Agile Construction and Evolution of Product-Line Architectures

PhD Thesis
November, 2012

Jessica Díaz Fernández

Supervisors:
Juan Garbajosa Sopeña
Jenifer Pérez Benedí
Agile Construction and Evolution of Product-Line Architectures

Jessica Díaz Fernández

Departmento de Lenguajes, Sistemas Informáticos e Ingeniería del Software, Facultad de Informática
Universidad Politécnica de Madrid (UPM) — Technical University of Madrid

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science

Supervised by
Dr. Juan Garbajosa Sopeña
Dr. Jenifer Pérez Benedí

November, 2012
Supervisors:
Dr. Juan Garbajosa Sopeña
E.U. Informática, Technical University of Madrid (Spain)

Dr. Jenifer Pérez Benedit
E.U. Informática, Technical University of Madrid (Spain)

External Reviewers:
Dr. Alexander L. Wolf
Department of Computing
Imperial College London (UK)

Dr. José Luis Fiadeiro
Department of Computer Science
Royal Holloway University of London (UK)

Thesis Defense Committee Members
Dr. Juan José Moreno Navarro, Universidad Politécnica de Madrid (Spain)

Dr. Pedro P. Alarcón Cavero, Universidad Politécnica de Madrid (Spain)

Dr. Miguel Toro Bonilla, Universidad de Sevilla (Spain)

Dr. Alexander L. Wolf, Imperial College London (UK)

Dr. Pekka Abrahamsson, University of Bozen-Bolzano (Italy)

PhD. Thesis
This work is subjected to copyright. All rights are reserved.

This thesis has been partially supported by: (i) UPM (Technical University of Madrid) under their Researcher Training program. (ii) The Spanish Ministry of Science and Innovation (MICINN) through the R&D+i projects INNOSEP (TIN2009-13849) and i-Smart Software Factory (IPT-430000-2010-038). (iii) The Spanish Ministry of Industry (MITYC) through the R&D project IMPONET (ITEA 2 09030 – TSI-02400-2010-103). And (iv) The Centre for Industrial Technological Development (CDTI) through R&D projects NEMO&CODED (ITEA2 08022 – IDI-20110864) and Energos (CEN-20091048).

Credits of the cover: Jessica Díaz, Universidad Politécnica de Madrid.
Picture (cover): (c) 1xpert-Fotolia.com
Software Product Line Engineering (SPE) has proved to have significant advantages in family-based software development, but also implies the up-front design of a product-line architecture (PLA) from which individual product applications can be engineered.

The big upfront design associated with PLAs is in conflict with the current need of "being open to change". However, the turbulence of the current business climate makes change inevitable in order to stay competitive, and requires PLAs to be open to change even late in the development. The trend of "being open to change" is manifested in the Agile Software Development (ASD) paradigm, but it is spreading to the domain of SPE.

To reduce the big upfront design of PLAs as currently practiced in SPE, new paradigms are being created, one being Agile Product Line Engineering (APLE). APLE aims to make the development of product-lines more flexible and adaptable to changes as promoted in ASD. To put APLE into practice it is necessary to make mechanisms available to assist and guide the agile construction and evolution of PLAs while complying with the "be open to change" agile principle.

This thesis defines a process for "the agile construction and evolution of product-line architectures", which we refer to as Agile Product-Line Architecting (APLA). The APLA process provides agile architects with a set of models for describing, documenting and tracing PLAs, as well as an algorithm to analyze change impact. Both the models and the change impact analysis offer the following capabilities:

- Flexibility & adaptability at the time of defining software architectures, enabling change during the incremental and iterative design of PLAs (anticipated or planned changes) and their evolution (unanticipated or unforeseen changes).

- Assistance in checking architectural integrity through change impact analysis in terms of architectural concerns, such as dependencies on earlier design decisions, rationale, constraints, and risks, etc.
• Guidance in the change decision-making process through change impact analysis in terms of architectural components and connections.

Therefore, APLA provides the mechanisms required to construct and evolve PLAs that can easily be refined iteration after iteration during the APLE development process. These mechanisms are provided in a modeling framework called FPLA.

The contributions of this thesis have been validated through the conduction of a project regarding a metering management system in electrical power networks. This case study took place in an i-smart software factory and was in collaboration with the Technical University of Madrid and Indra Software Labs.

La Ingeniería de Líneas de Producto Software (Software Product Line Engineering, SPLE) ha demostrado tener ventajas significativas en el desarrollo de software basado en familias de productos. SPLE es un paradigma que se basa en la reutilización sistemática de un conjunto de características comunes que comparten los productos de un mismo dominio o familia, y la personalización masiva a través de una variabilidad bien definida que diferencia unos productos de otros. Este tipo de desarrollo requiere el diseño inicial de una arquitectura de línea de productos (Product-Line Architecture, PLA) a partir de la cual los productos individuales de la familia son diseñados e implementados.

La inversión inicial que hay que realizar en el diseño de PLAs entra en conflicto con la necesidad actual de estar continuamente “abierto al cambio”, siendo este cambio cada vez más frecuente y radical en la industria software. Para ser competitivos es inevitable adaptarse al cambio, incluso en las últimas etapas del desarrollo de productos software. Esta tendencia se manifiesta de forma especial en el paradigma de Desarrollo Ágil de Software (Agile Software Development, ASD) y se está extendiendo también al ámbito de SPLE.

Con el objetivo de reducir la inversión inicial en el diseño de PLAs en la manera en que se plantea en SPLE, en los últimos años han surgido nuevos enfoques como la Ingeniería de Líneas de Producto Software Ágiles (Agile Product Line Engineering, APLE). APLE propone el desarrollo de líneas de producto de forma más flexible y adaptable a los cambios, iterativa e incremental. Para ello, es necesario disponer de mecanismos que ayuden y guíen a los arquitectos de líneas de producto en el diseño y evolución ágil de PLAs, mientras se cumple con el principio ágil de estar abierto al cambio.

Esta tesis define un proceso para la “construcción y evolución ágil de las arquitecturas de líneas de producto software”. A este proceso se le ha denominado Agile Product-Line Architecting (APLA). El proceso APLA proporciona a los arquitectos software un conjunto de modelos para describir, documentar y trazar PLAs, así como un algoritmo para analizar...
el impacto del cambio. Los modelos y el análisis del impacto del cambio ofrecen:

- Flexibilidad y adaptabilidad a la hora de definir las arquitecturas software, facilitando el cambio durante el diseño incremental e iterativo de PLAs (cambios esperados o previstos) y su evolución (cambios no previstos).
- Asistencia en la verificación de la integridad arquitectónica mediante el análisis de impacto de los cambios en términos de dependencias entre decisiones de diseño, justificación de las decisiones de diseño, limitaciones, riesgos, etc.
- Orientación en la toma de decisiones derivadas del cambio mediante el análisis de impacto de los cambios en términos de componentes y conexiones.

De esta manera, APLA se presenta como una solución para la construcción y evolución de PLAs de forma que puedan ser fácilmente refinadas iteración tras iteración de un ciclo de vida de líneas de producto ágiles. Dicha solución se ha implementado en una herramienta llamada FPLA (Flexible Product-Line Architecture) y ha sido validada mediante su aplicación en un proyecto de desarrollo de un sistema de gestión de medición en redes de energía eléctrica. Dicho proyecto ha sido desarrollado en una fábrica de software global en colaboración con la Universidad Politécnica de Madrid e Indra Software Labs.

**Palabras Clave:** Líneas de Producto Software, Arquitecturas Software, Arquitecturas de Líneas de Producto, Desarrollo Ágil de Software, Conocimiento Arquitectónico, Análisis de Impacto del Cambio, Componentes Plásticas y Parciales.
A Carlos,

y

A mis padres

Nuestra forma de vida depende más de cómo vivimos las cosas,
que de las cosas que vivimos

Anónimo
Acknowledgements

A mi marido, Carlos, por darme su apoyo incondicional, su cariño y por animarme a continuar. Sin él no lo habría conseguido.

A mis padres, Gregorio y Consuelo, que me han enseñado el valor de esfuerzo y del trabajo para conseguir aquellas metas que uno se propone en la vida, por su paciencia durante todos estos años de estudios, y su continuo apoyo y cariño.

A mi hermano, y resto de familiares, por estar ahí. En especial para aquellos que ya no están aquí...

A mis directores de tesis, Juan Garbajosa y Jennifer Pérez, por la confianza que han depositado en mí, el apoyo prestado y por haber compartido todo su conocimiento y sabiduría conmigo.

A mi amiga Jennifer, compañera de risas y lágrimas, de consejos y buenos ratos, de apoyo y motivación para seguir con esta carrera, pero sobre todo una compañera en la vida.

A Agustín y Pedro sin cuya ayuda tampoco hubiese sido posible este logro.

Al resto de miembros del grupo SYST: Carlos F., Carlos S., Daniel, Iria, Lili, Pili y todos aquellos que han pasado por el grupo, por los buenos ratos y hacer más agradable el trabajo.

A todos aquellos que han participado en la implementación de la herramienta FPLA: David, Raúl, Miguel, Eva, Cristina, Gonzalo y Sabela. A todos, gracias.

Al resto de profesores del dpto OEI de la Escuela de Informática.

Al personal de Indra Software Labs, en particular a Eloy y Mercedes, que me han proporcionado en todo momento la información necesaria para llevar a cabo el caso de estudio.

Thanks to Prof. Alex Wolf for his support, helpful advice and comments during my stay at Imperial College London. Also to Prof. José Fiadeiro for his exhaustive review.

Gracias a todos,
Table of contents

List of Figures xiv
List of Tables xvii

I INTRODUCTION 1
1 Introduction 3
   1.1 Research Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   1.2 Research Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
   1.3 Main Goal and Objectives . . . . . . . . . . . . . . . . . . . . . . . . . 10
   1.4 Research Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . 13
   1.5 Research Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
   1.6 Thesis Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

II STATE OF THE ART 21
2 Background 23
   2.1 Software Evolution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
      2.1.1 Evolution as part of the Development Process . . . . . . . . . . . 24
      2.1.2 Negative effects of Software Evolution . . . . . . . . . . . . . . . 25
      2.1.3 Change Impact Analysis . . . . . . . . . . . . . . . . . . . . . . . 27
      2.1.4 Traceability Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 28
      2.1.5 Dependency Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 30
      2.1.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
   2.2 Software Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
      2.2.1 Definition of Software Architecture . . . . . . . . . . . . . . . . . 32
      2.2.2 Main Concepts of Software Architectures . . . . . . . . . . . . . . 34
      2.2.3 Architecture Description Language . . . . . . . . . . . . . . . . . 36
      2.2.4 Architecture Evolution . . . . . . . . . . . . . . . . . . . . . . . . 37
         2.2.4.1 Architecture Evolution Taxonomy . . . . . . . . . . . . 38

ix
# TABLE OF CONTENTS

2.2.4.2 Architecture Evolution Support .......................... 40
2.2.4.3 Challenges in Architecture Evolution ................. 41
2.2.5 Architectural Knowledge (AK) .......................... 42
  2.2.5.1 Documenting & Formalizing Architectural Knowledge . 44
  2.2.5.2 Managing Architectural Knowledge .................. 45
2.2.6 Agile (Evolutionary) Architecting ..................... 46
2.2.7 Conclusions ............................................ 47
2.3 Software Product Lines .................................... 49
  2.3.1 Main Concepts of Software Product Lines ............ 50
  2.3.2 Product-Line Development Process .................. 51
  2.3.3 Variability Management ................................ 53
    2.3.3.1 Modeling Variability .......................... 54
    2.3.3.2 Feature Modeling ............................. 56
    2.3.3.3 Product-Line Architecture Modeling .......... 58
    2.3.3.4 Documenting Variability Design Rationale .... 60
  2.3.4 Evolution of Product-Line Architecture ............. 62
  2.3.5 Conclusions ............................................ 64
3 Agile Product Line Engineering ............................... 65
  3.1 Introduction ............................................. 66
  3.2 Systematic Literature Review: the method ............. 68
    3.2.1 First Phase: Planning the Review ................ 68
    3.2.2 Second Phase: Conducting the review phase ...... 72
    3.2.3 Third Phase: Reporting the review ............ 73
  3.3 Key Findings: Implications for Practitioners and Researchers 83
    3.3.1 How can APLE be put into practice? ............ 84
    3.3.2 Which are the open research challenges? ....... 88
    3.3.3 Final considerations ............................. 90
  3.4 Conclusions ............................................. 90

III CONSTRUCTION AND EVOLUTION OF PLAs .................. 93
4 Preliminaries ................................................ 95
  4.1 Empirical Pilot Study: A SPL for banking systems .... 95
  4.2 Scrum .................................................. 104
  4.3 Conclusions ............................................. 106
5 Flexible Product Line Architectures ........................ 107
  5.1 External vs. Internal variability ........................ 108
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>The Flexible-PLA Model</td>
<td>110</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Plastic Partial Components (PPC)</td>
<td>110</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Abstract Syntax: Metamodel description</td>
<td>113</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Concrete Syntax: Graphical language description</td>
<td>118</td>
</tr>
<tr>
<td>5.3</td>
<td>Flexible-PLA Model in MDE</td>
<td>120</td>
</tr>
<tr>
<td>5.4</td>
<td>The Flexible-PLA 4-view model</td>
<td>122</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Exemplification</td>
<td>123</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Considerations</td>
<td>125</td>
</tr>
<tr>
<td>5.5</td>
<td>Discussion</td>
<td>128</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusions</td>
<td>134</td>
</tr>
<tr>
<td>6</td>
<td>Product Line Architectural Knowledge</td>
<td>137</td>
</tr>
<tr>
<td>6.1</td>
<td>Variability Design Rationale</td>
<td>138</td>
</tr>
<tr>
<td>6.2</td>
<td>The PLAK Model</td>
<td>139</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Product Line Architectural Knowledge (PLAK)</td>
<td>139</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Abstract Syntax: Metamodel description</td>
<td>143</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Concrete Syntax: Graphical language description</td>
<td>147</td>
</tr>
<tr>
<td>6.3</td>
<td>The PLAK 4+1 view model</td>
<td>149</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Exemplification</td>
<td>150</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Considerations</td>
<td>151</td>
</tr>
<tr>
<td>6.4</td>
<td>Discussion</td>
<td>153</td>
</tr>
<tr>
<td>6.5</td>
<td>Conclusions</td>
<td>158</td>
</tr>
<tr>
<td>7</td>
<td>Change Impact Analysis in Product Line Architectures</td>
<td>159</td>
</tr>
<tr>
<td>7.1</td>
<td>Architecture-based CIA</td>
<td>160</td>
</tr>
<tr>
<td>7.2</td>
<td>The CIA technique</td>
<td>161</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Change Typology</td>
<td>162</td>
</tr>
<tr>
<td>7.2.2</td>
<td>A traceability-based algorithm</td>
<td>163</td>
</tr>
<tr>
<td>7.2.3</td>
<td>A rule-based inference engine</td>
<td>163</td>
</tr>
<tr>
<td>7.3</td>
<td>Exemplification</td>
<td>165</td>
</tr>
<tr>
<td>7.4</td>
<td>Discussion</td>
<td>169</td>
</tr>
<tr>
<td>7.5</td>
<td>Conclusions</td>
<td>170</td>
</tr>
<tr>
<td>IV</td>
<td>AGILE CONSTRUCTION AND EVOLUTION OF PLAs</td>
<td>171</td>
</tr>
<tr>
<td>8</td>
<td>Agile Product-Line Architecting Process</td>
<td>173</td>
</tr>
<tr>
<td>8.1</td>
<td>Agile Architecting</td>
<td>174</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Agile Architecting through Flexible &amp; Adaptive Architectures</td>
<td>175</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Agile Architecting through Change-Impact Analysis</td>
<td>177</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

8.2 The Agile Product-Line Architecting Process ........................................ 179
  8.2.1 APLA in Scrum .......................................................... 179
  8.2.2 Discussion .............................................................. 183
8.3 The FPLA Modeling Framework ............................................................. 186
  8.3.1 FPLA main characteristics ............................................... 186
  8.3.2 FPLA: Empirical Assessment ............................................ 189
    8.3.2.1 Test I: Description .............................................. 190
    8.3.2.2 Test I: Results .................................................. 193
    8.3.2.3 Test II: Description ............................................. 196
    8.3.2.4 Test II: Results .................................................. 197
    8.3.2.5 Conclusions & Limitations of the FPLA Empirical Assessment .... 200
8.4 Conclusions ....................................................................................... 201

V CASE STUDY ......................................................................................... 203

9 Case Study ............................................................................................ 205
  9.1 Context: the i-smart software factory .............................................. 206
  9.2 Reporting the case study ............................................................... 206
    9.2.1 Introduction ........................................................................ 207
      9.2.1.1 Problem statement .................................................. 207
      9.2.1.2 Research objectives .............................................. 208
    9.2.2 Case study design .................................................................. 209
      9.2.2.1 Research Questions ................................................ 209
      9.2.2.2 Data collection procedure .................................... 213
      9.2.2.3 Analysis & Validity procedure ............................... 214
      9.2.2.4 Case study description .......................................... 216
      9.2.2.5 Subject description ................................................. 220
    9.2.3 Results .................................................................................. 220
      9.2.3.1 Case study execution: Optimeter I ............................ 220
      9.2.3.2 Analysis and interpretation ................................... 226
      9.2.3.3 Evaluation of validity .......................................... 232
    9.3 Conclusions ................................................................................. 233

VI CONCLUSIONS & FURTHER WORK .................................................. 235

10 Conclusions & Further Work ................................................................ 237
  10.1 Conclusions ............................................................................... 237
  10.2 Research Contributions ................................................................. 239
# TABLE OF CONTENTS

10.2.1 Results ....................................... 239  
10.2.2 Description of Key Publications ................. 240  
10.2.3 Other related publications ........................ 243  
10.3 Further Research .................................. 244  

Bibliography ........................................... 247  

Appendix A: Flexible-PLA OCL Constraints ............. 271  
Appendix B: APLA in Scrum — Model process compliant to SPEM 273  
Appendix C: FPLA Experiment Questionaries ......... 275  

xiii
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Top Five WWW Smartphone Vendors, 1Q 2012, Five Quarter Market Share Changes (Units)</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Technical Debt</td>
<td>8</td>
</tr>
<tr>
<td>1.3</td>
<td>Thesis’ contributions</td>
<td>13</td>
</tr>
<tr>
<td>1.4</td>
<td>Thesis overview</td>
<td>19</td>
</tr>
<tr>
<td>2.1</td>
<td>Approaches for representing variability</td>
<td>55</td>
</tr>
<tr>
<td>2.2</td>
<td>Feature metamodel</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Exemplar of banking software systems</td>
<td>96</td>
</tr>
<tr>
<td>4.2</td>
<td>Banking systems SPL - Feature model</td>
<td>100</td>
</tr>
<tr>
<td>4.3</td>
<td>Banking systems SPL - Architecture model. Component &amp; Connector view</td>
<td>101</td>
</tr>
<tr>
<td>4.4</td>
<td>Banking systems SPL - Architecture model - Services specification</td>
<td>102</td>
</tr>
<tr>
<td>4.5</td>
<td>Active redundancy</td>
<td>103</td>
</tr>
<tr>
<td>4.6</td>
<td>Passive redundancy</td>
<td>104</td>
</tr>
<tr>
<td>4.7</td>
<td>Front end services</td>
<td>105</td>
</tr>
<tr>
<td>4.8</td>
<td>Scrum lifecycle</td>
<td>106</td>
</tr>
<tr>
<td>5.1</td>
<td>Plastic Partial Component</td>
<td>112</td>
</tr>
<tr>
<td>5.2</td>
<td>Flexible-PLA metamodel</td>
<td>115</td>
</tr>
<tr>
<td>5.3</td>
<td>Flexible-PLA graphical modeling language</td>
<td>118</td>
</tr>
<tr>
<td>5.4</td>
<td>Flexible-PLA graphical modeling language: Banking systems PLA</td>
<td>119</td>
</tr>
<tr>
<td>5.5</td>
<td>Flexible-PLA graphical modeling language: Banking systems PLA</td>
<td>120</td>
</tr>
<tr>
<td>5.6</td>
<td>The Flexible-PLA metamodel in MDD</td>
<td>121</td>
</tr>
<tr>
<td>5.7</td>
<td>Flexible-PLA model - Core View of banking systems PLA</td>
<td>124</td>
</tr>
<tr>
<td>5.8</td>
<td>Flexible-PLA model - Variability View (variability point definition) of banking systems PLA</td>
<td>125</td>
</tr>
<tr>
<td>5.9</td>
<td>Flexible-PLA model - Variability View (weaving definition) of banking systems PLA</td>
<td>126</td>
</tr>
</tbody>
</table>
Figure 5.10 Excerpt from Figure 5.7 without using the PPC variability mechanism

Figure 6.1 Product-line architecture knowledge: Closed DD

Figure 6.2 Product-line architecture knowledge: Open, Optional and Alternative DDs

Figure 6.3 PLAK Metamodel

Figure 6.4 The PLAK Metamodel in MDD

Figure 6.5 PLAK graphical modeling language

Figure 6.6 PLAK graphical modeling language: Banking systems PLA

Figure 6.7 PLAK model of the banking systems SPL

Figure 6.8 Approaches for documenting variability

Figure 7.1 PLAK model of the banking systems SPL

Figure 7.2 Change in the banking systems feature model

Figure 7.3 Change in the banking systems PLA

Figure 8.1 Agile Architecting mechanisms

Figure 8.2 APLA in Scrum

Figure 8.3 APLA in Scrum: Model process compliant to SPEM (I)

Figure 8.4 APLA in Scrum: Model process compliant to SPEM (II)

Figure 8.5 APLA in Scrum: Model process compliant to SPEM (III)

Figure 8.6 FPLA Modeling Framework

Figure 8.7 FPLA Modeling Framework

Figure 8.8 FPLA model - Feature model of home automation systems SPL

Figure 8.9 FPLA model - Core View of home automation systems SPL

Figure 8.10 FPLA model - Variability View of home automation systems SPL

Figure 8.11 FPLA model - Service & interface definition of home automation systems SPL

Figure 8.12 FPLA model - Variability View (weaving definition) of home automation systems SPL

Figure 8.13 Test I.a

Figure 8.14 Graph I

Figure 8.15 Test I.b

Figure 8.16 Graph II

Figure 8.17 Graph III

Figure 8.18 Test II.a

Figure 8.19 Test II.b

Figure 8.20 Graph IV

Figure 8.21 Graph V
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.22</td>
<td>Graph VI</td>
<td>200</td>
</tr>
<tr>
<td>9.1</td>
<td>Smart grid overview</td>
<td>217</td>
</tr>
<tr>
<td>9.2</td>
<td>Metering management system</td>
<td>218</td>
</tr>
<tr>
<td>9.3</td>
<td>OPTIMETER SPL</td>
<td>219</td>
</tr>
<tr>
<td>9.4</td>
<td>Optimeter I - Feature model</td>
<td>221</td>
</tr>
<tr>
<td>9.5</td>
<td>Optimeter I - PLAK model: Sprint 1</td>
<td>222</td>
</tr>
<tr>
<td>9.6</td>
<td>Optimeter I - PLAK model: Sprint 2</td>
<td>223</td>
</tr>
<tr>
<td>9.7</td>
<td>Optimeter I - PLAK model: Sprint 3</td>
<td>224</td>
</tr>
<tr>
<td>9.8</td>
<td>Optimeter I - PLAK model: Sprints 4 &amp; 5</td>
<td>225</td>
</tr>
<tr>
<td>9.9</td>
<td>Optimeter I - PLAK model: Sprint 6</td>
<td>226</td>
</tr>
<tr>
<td>9.10</td>
<td>Optimeter I - Sonar dashboard</td>
<td>227</td>
</tr>
<tr>
<td>10.1</td>
<td>APLA in Scrum: Model process compliant to SPEM</td>
<td>274</td>
</tr>
</tbody>
</table>
List of Tables

Table 1.1 Summary of research objectives, results and key publications.

Table 2.1 Traceability Types

Table 3.1 SLR - Search resources.

Table 3.2 SLR - Quality criteria

Table 3.3 SLR - Results of the systematic search for primary studies.

Table 3.4 SLR - Selected papers

Table 3.5 SLR - Distribution of studies according to the publication channel.

Table 3.6 Matching ASD and SPLE

Table 3.7 SLR - Summary of contributions/approaches working on a challenge.

Table 3.8 Summary for Application Engineering.

Table 3.9 Summary for Domain Engineering.

Table 4.1 Architectural Tactics for Availability

Table 5.1 Plastic Partial Components Cardinality

Table 5.2 Discussion summary

Table 6.1 Linkages Rule Logics for Design Decision Traceability Links

Table 6.2 Numbers of the PLAK model of the banking systems SPL

Table 6.3 Discussion summary

Table 7.1 Rule-based inference engine: propagation rules

Table 9.1 Case study - Summary of research questions

Table 9.2 Case study - Summary of research validation

Table 9.3 Case study - Data collection methods and data sources

Table 9.4 $RQ_2$: Descriptive statistics

Table 9.5 Optimeter I - Sonar dependency matrix

Table 9.6 Alternative Optimeter - Sonar dependency matrix

Table 9.7 $RQ_6$: Descriptive statistics
Part I

INTRODUCTION
Chapter 1

Introduction

In the past decade, the software industry has recognized the strategic importance of software product lines [Linden et al., 2007; Pohl et al., 2005], especially for those companies that develop products of the same family or domain. Software Product Line Engineering (SPLE) has proved to have significant advantages in family-based software development. These include faster time-to-market, better quality and lower cost by means of systematic reuse and mass customization [Clements & Northrop, 2002].

SPLE takes advantage of the common features of a family of products and anticipates the expected degree of variation over the product’s lifetime. SPLE exploits the commonality found in the products of a same family by using reusable core assets (typically developed in a phase called domain engineering) which are customized and assembled into customer-specific products (typically developed in a phase called application engineering). This domain-then-application model [Pohl et al., 2005] requires a “big upfront design” (BUFD) of what is known as a product-line platform before individual product applications can be engineered. This product-line platform consists of a detailed commonality and variability analysis, the design of a domain architecture —also known as Product-Line Architecture—, and the implementation of the set of reusable core assets. Therefore, SPLE demands a large investment upfront that small to medium organizations might not afford. Even large organizations might not be able to afford investing in long-term upfront design, given the fast-paced technological advancement nowadays and turbulence of the current business climate [Catal, 2009; Ghanam et al., 2011].
1. INTRODUCTION

The turbulence of the current business climate makes change inevitable in order to stay competitive (see Figure 1.1 which shows the volatility of the smartphone market in just over one year). The capacity to respond to this turbulence by harnessing, rather than avoiding change, is key to survival [Highsmith 2002]. Hence, for over a decade now, there has been a proliferation of methods under the general term of Agile Software Development (ASD, [Beck et al., 2001]) under which agility is defined as “the ability to both create and respond to change in order to profit in a turbulent business environment” [Highsmith, 2002]. ASD emphasizes rapid and flexible construction of working products in an iterative and incremental way through continuous delivery of valuable software by short time-framed iterations, as well as welcoming changing requirements even late in development.

![Figure 1.1: Top Five WWW Smartphone Vendors, 1Q 2012, Five Quarter Market Share Changes (Units)](source: IDC Worldwide Mobile Phone Tracker, May 1, 2012)

Nokia’s Symbian phone shipments declined precipitously last quarter. The company’s current smartphone woes make a speedy transition to products powered by the Windows Phone operating system, upon which it has bet its smartphone future, critical.

Recently, there has been an interest in finding ways to combine reuse and customization as practiced in SPLE with concepts such as iterative development and embracement to change as encouraged in ASD [Ghanam et al., 2011]. A new approach called Agile Product Line Engineering (APLE) [Cooper & Franch, 2006] advocates the integration of SPLE and ASD with the aim of reducing the BUFD associated with the product-line platform while making the development of software product lines more flexible and adaptable to changes. Although APLE is promising [Ali Babar et al., 2009b; Hanssen 2010; Mohan et al., 2010], many of the foundations of SPLE and ASD are completely different, or even opposite, making it necessary to deal with several challenges to put APLE fully into practice, i.e. to harmonize ASD and SPLE.
One of the main challenges that APLE faces consists of designing and evolving the product-line architecture while complying with the agile principles [Ali Babar et al., 2009b; Ghanam & Maurer, 2008; Hanssen & Fígri, 2008; Tian & Cooper, 2006]. In fact, the reconciliation of software architecture [Perry & Wolf, 1992] and agile [Shore & Warden, 2007] communities is a controversial issue which has been extensively discussed at the 2009 Software Architecture Challenges in the 21st Century Workshop, the 2010 Special Issue of IEEE Software on Agility and Architecture — Oil and Water? and the 2009 and 2010 Workshops on Architecture-Centric Methods and Agile Approaches. Kruchten [2009] and Booch [2009], among others, advocate the iterative and incremental evolution of the architecture to reduce the BUFD and keep the system in sync with changing conditions. Namely, “the architecture develops with the system, and includes only features that are necessary for the current iteration or delivery” [Chivers et al., 2005]. However, how to perform this iterative architecture refinement is still a challenge [Abrahamsson et al., 2010; Ghanam et al., 2009b]. This challenge is addressed in this thesis.

This thesis aims to reconcile product-line architecture and agility. To that end, we define a process and supporting mechanisms for “the agile construction and evolution of product-line architectures”, which we refer to as Agile Product-Line Architecting (APLA). This APLA process and their supporting mechanisms assist and guide the iterative and incremental construction and evolution of product-line architectures, while remaining close to the “be open to change” agile principle. Specifically, the supporting mechanisms assist and guide architects in the following three tasks: (i) defining flexible and adaptive product-line architectures that are able to respond to change, (ii) reducing the risk of unexpected consequences of changes and enabling the preservation of the integrity of the architecture, and (iii) providing architects with knowledge that assists and guides them in the change decision-making process, iteration after iteration, during the APLE development process.

The structure of this chapter is as follows: Section 1.1 introduces the motivation of this work, Section 1.2 defines the context of this research, and Section 1.3 defines the main goal and the specific objectives pursued by this thesis. Section 1.4 presents the main contributions and lists a number of publications, authored or coauthored by the author of this thesis, which present several of the thesis results. Section 1.5 presents the research methodology that has been followed during the thesis development. Finally Section 1.6 summarizes the structure of the thesis.
1. INTRODUCTION

1.1 Research Motivation

It is a well-accepted fact in software engineering that architectures make software systems simpler and easier to understand [Garlan & Perry, 1995]. Software architectures describe the structure of a software system by hiding low-level details and abstracting high-level important features [Perry & Wolf, 1992]. The design, specification, and analysis of the structure of software-intensive systems have become critical issues in software development [Garlan, 2001]. Hence, software architectures emerged as a solution for the design and development of large and complex software systems.

The role of architectures in the development of software product lines is even more important than in the case of single-products. Product-Line Architectures (PLA) define the common and variable structure for all products of a product-line, which has a high strategic value and deep impact on the success of product-lines [Bosch & Ran, 2000]. The upfront investment on a PLA, before individual product applications can be engineered, has become a great barrier which small or even large organizations might not be able to afford, given the volatile business situation nowadays [Catal, 2009; Ghanam et al., 2011]. The new development paradigm known as Agile Product Line Engineering (APLE) [Cooper & Franch, 2006] advocates the integration of SPL and ASD with the aim of (i) reducing the upfront analysis and design as practiced in SPL, and (ii) making the development of product-lines more flexible and adaptable to changes, as encouraged in ASD. Because SPL is tailored in favor of an agile incremental development in short iterations, the PLA must also support its incremental and iterative refinement.

Although agile methods do not explicitly reject the concept of architecture, they are all rather silent on how they can coexist. In fact, most agile methods have no explicit guidance for architecture [Kruchten, 2009]. Therefore, one of the major challenges to put APLE into practice is the problem of reconciling architecture and agility. The importance and motivation behind reconciling architecture and agility can be analyzed from two points of view: the role of software architecture in ASD, and the role of agility in software architecture.

The role of software architecture in ASD has been extensively discussed over the last few years. There are many advocates for and opponents against giving the importance in ASD to architectures as they have in other development approaches. Hence, literature is full of references that advocate against architecture in ASD, as customers can rarely appreciate the value that architecture delivers. Agile practitioners often consider that the upfront design and definition of software architectures is an investment in
time and effort that may not be paid off. Basically the agile aversion to architecture is due to the association of architecture with BUFD leading to massive documentation and implementation of YAGNI (you aint gonna need it) features, being perceived as a “fixed artifact” which is incompatible with the response to change (excerpt from Abrahamsson et al., 2010; Erdogmus, 2009). A common belief is that “If you are sufficiently agile, you don’t need an architecture - you can always refactor it on the fly” (excerpted from Cockburn, 2006). However, recently, there has been a growing recognition of the importance of paying more attention to good architecture design in ASD (Babar & Abrahamsson, 2008) which is actually key to keeping the cost of change low, and more and more authors emphasize the main role of software architectures to scale up the traditional scope of ASD to large software-intensive systems (Abrahamsson et al., 2010; Ali Babar et al., 2009b; Booch, 2010; Erdogmus, 2009; Falesi et al., 2010; Kruchten, 2009; Madison, 2010; Nord & Tomayko, 2006). It has been argued that an inaccurate architectural design leads to the failure of large software systems and that large code refactoring might create significant defects without considering architectural issues (Bowers et al., 2002). As reported by Dyba & Dingsoyr, 2008, several authors advocate that the lack of focus on architecture is bound to engender suboptimal design-decisions. This lacking element is in contradiction to an agile principle which establishes that “continuous attention to technical excellence and good design enhances agility”. Cockburn, 2006 claims that the issue between architecture and agility is not either architecture yes or architecture no: he thinks that the issue is how much effort should be invested in architecture, assuming that the architecture can be valuable for the customer. In his work, Kruchten, 2008 analyzes the cost and value of software architectures in agile software development. Then, the key question is: Are we able to avoid the obstacles that hamper agile practitioners to design software architectures without renouncing their values and principles?

The role of agility in software architecture is intended to make flexible and adaptable architectures which are able to respond to change in order to make profit in a turbulent business environment. Many authors refer to flexibility and adaptability as anticipation and adaptation, respectively (Highsmith, 2009; Kruchten, 2011), i.e. the ability to deal with changes that can be anticipated or planned versus changes that cannot be anticipated, nor foreseen. Agile values aim “to respond to change over following a plan”. This way, the flexibility and adaptability as preached in ASD are due to the iterative and incremental construction and evolution of working products, by including only the features at hand —i.e. “keeping the engineering team grounded in what is known today” (Highsmith, 2009). —, simple design, continuous integration, automated testing, and continuous refactoring. However, this kind of flexibility and adaptability in the process is not enough to deal with the turbulence of current business situations.
1. INTRODUCTION

There is a growing recognition that “designs should be based upon what we know today and a willingness to engage in redesign in the future — an evolutionary design process” [Highsmith, 2009]. This responds to the current trend of managing the technical debt (see Figure 1.2) to keep the cost of change (CoC) low so that customer responsiveness remains as high as possible during a product’s life [Highsmith, 2009]. If architecture design is not given proper attention, we may incur in large technical debts [Kruchten, 2011]. As a result, it would also be desirable to have flexibility and adaptability in the product architecture design itself in order to reduce the cost of change. Hence, architecture again plays an important role in ASD as the sole focus on what we know today could incur in technical debt, and in presence of technical debt the cost of iteratively adding new features gets increasingly higher iteration after iteration [Kruchten, 2011].

Like Kruchten [2009], this thesis considers architecture and agility an oxymoron. The challenge is to provide the means for the agile construction and evolution of architectures in order to achieve all the abovementioned benefits that both offer separately. Although several approaches present successful cases of agile architecture [Ibike & Abrahamsson, 2005] or iterative architecture [Chivers et al., 2005], they are specific solutions in the domains of mobile and security-based applications respectively, and they are not easily generalized to other domains. How to perform agile architecting and which general-purpose architectural mechanisms could support it, is in fact one of the main challenges [Abrahamsson et al., 2010; Ghanam et al., 2009b], and the main goal of this thesis. Specifically in the case of PLAs, agile product-line architecting is still an open challenge [Ali Babar et al., 2009b].

Figure 1.2: Technical Debt - Source: [Highsmith, 2009]. A technical debt strategy attempts to keep the CoC low so that customer responsiveness remains as high as possible during a product’s life.
1.2 Research Context

The work presented in this thesis is concerned with the problem of reconciling agility and the design of those kinds of architectures used to realize product-lines. As proposed by Kruchten [2009] and Booch [2009], this thesis studies support for the agile construction and evolution of PLAs. Next, we list the software engineering areas into which this thesis is framed:

- **Agile Product Line Engineering**: This thesis focuses on the APLE paradigm, which pursues to combine reuse and customization as practiced in SPLE with concepts such as iterative development and embracement to change as promoted in ASD. Specifically, this thesis focuses on reconciling agility and architecture.

- **Agile Product-Line Architecting**: This thesis focuses on the iterative and incremental construction and evolution of PLAs according to the “be open to change” agile principle: APLA. This iterative and incremental construction process can be seen as an evolutionary design process. This thesis addresses both evolutionary design and evolution from the how perspective that Lehman et al. [2000] describe, i.e. the methods and tools intended to facilitate software evolution and the tasks that implement it. The focus on architecture is motivated by the high-level insight software architecture provides, which permits restructuring where design decisions, rationale, constraints, and tradeoffs are made [Garlan et al., 2009]. Specifically, this thesis focuses on static evolution, i.e. the offline changes to an architecture that are implemented at design time, as opposed to dynamic evolution which refers to modifications in the system while the system is executing, i.e. runtime.

- **Model-Driven Software Evolution**: The APLA process that this thesis defines is based on models as the primary artifacts for developing and evolving software. This approach follows the Model-Driven Development paradigm (MDD) Beydeda et al., 2005; Selic 2003, and more specifically the emerging paradigm called Model-Driven Software Evolution (MoDSE) Deridder et al., 2011. MoDSE pursues to (semi-)automatically support software change by using models and traceability between models. Hence, a metamodeling approach is used to define the domain-specific languages needed to represent each participating artifact which is amenable to be manipulated in the APLA process.
1.3 Main Goal and Objectives

The main goal of this thesis can be formulated as follows:

*To define a process and supporting mechanisms to assist and guide the iterative and incremental construction and evolution of product-line architectures, while complying with the “be open to change” agile principle.*

Therefore, the main goal is to define a process to support the agile construction and evolution of product-line architectures, which we refer to as *Agile Product-Line Architecting (APLA).* From the study of relevant literature and background related to this thesis, it has been concluded that this APLA process should provide at least the following capabilities:

- **Flexibility & adaptability** at the time of defining software architectures through mechanisms that facilitate change during the incremental and iterative design of PLA (anticipated or planned changes), as well as its evolution (unanticipated or unforeseen changes).

- **Assistance in checking the architectural integrity** at the time of constructing and evolving PLAs through mechanisms that facilitate change impact analysis, in terms of architectural concerns such as dependencies on earlier design decisions, rationale, constraints and risks, etc which may be impacted by change.

- **Guidance in the change decision-making process** at the time of constructing and evolving PLAs through mechanisms that facilitate change impact analysis in terms of architectural components and connections which may be impacted by change.

To achieve these capabilities, the APLA process should provide mechanisms for (i) describing flexible and adaptive PLAs, (ii) documenting knowledge of PLAs—design decisions, their dependencies, rationale, constraints, among others—, and (iii) tracing architecturally significant features with their realization in PLAs. Architectural knowledge and traces are traversed to automatically analyze the impact of adding or changing features over PLAs, i.e. to determine the potential effects upon a system resulting from a proposed change. This change-impact knowledge helps preserve structural integrity while the architecture is iteratively and incrementally designed in the APLE.
development process. It is also useful for reasoning about a proposed change in features and guiding the change decision-making process. Additionally, as the role of variability at the time of defining PLAs is essential, how to support this variability during the APLA process is one of the major (and implicit) challenges to be dealt with in this thesis. Therefore, the process should be based on mechanisms and techniques that integrate and manage variability during the construction and evolution of PLAs.

The criteria to validate the achievement of the main goal of this thesis are defined in terms of the assistance and guidance that these mechanisms provide for agile product-line architecting.

The specific objectives that have been addressed to achieve the expected results are formulated as follows:

**OBJ 1.** To define a mechanism for describing PLAs. The aim of this mechanism is to provide software architecture with flexibility and adaptability at the time of their definition. To provide software architectures with flexibility —i.e. the ability to easily incorporate anticipated or planned changes—, this thesis proposes the modeling of architectural variability. To provide software architecture with adaptability —i.e. the ability to easily incorporate unanticipated or unforeseen changes—, this thesis proposes to have minimal dependencies and coupling between components in which variants are independent of the linking context in order to facilitate the possibility of low cost plug-in/plug-out variants.

The criteria to validate the achievement of this specific objective are defined as follows:

- expressiveness in describing variability at the level of (i) configuration of the architecture (external variability), and (ii) internal structure of architectural elements (internal variability)
- capabilities to provide agile architects with flexibility and adaptability while agile product-line architecting

**OBJ 2.** To define a mechanism for documenting product-line architectural knowledge and tracing architecturally significant features during their realization in the PLA. Capturing architectural knowledge allows one to revisit important design decisions over time, whereas tracing features to architecture assists software engineers in understanding the relationship and dependencies between features and PLA. This mechanism aims to help preserve the integrity of the architecture.
1. INTRODUCTION

The criteria to validate the achievement of this specific objective are defined as follows:

- expressiveness in documenting variability design rationale including the two levels —external and internal variability—
- expressiveness in defining traceability of variability through the artifacts of features and product-line architectures, including the two levels of variability —external and internal variability—
- capabilities to guide agile architects in preserving architectural integrity while agile product-line architecting

OBJ 3. To define a mechanism for traversing PLA descriptions, architectural knowledge documentation, and traceability definitions. Traversing PLA descriptions, their knowledge and traces allows one to analyze and determine the potential impact upon the architecture resulting from the implementation of a change in features. This mechanism aims to provide guidance in the change decision-making process.

The criteria to validate the achievement of this specific objective are defined as follows:

- effectiveness in locating the impacted architectural design decisions and elements resulting from a proposed change in features
- capabilities in guiding agile architects to reason about changes in order to make better evolution decisions based on impact and viability of the change

OBJ 4. To automate or semi-automate the previous objectives through a tool. The aim of the tool is to provide modeling primitives for describing, documenting and tracing PLAs during the iterative and incremental construction and evolution of product-line architectures. The tool also implements the algorithm and transformation functionalities for automating change impact analysis and generating code.

The criteria to validate the achievement of this specific objective are defined as follows:

- usability of the tool and understandability of its models
- guarantee of the correctness of models (models of models)
1.4 Research Contributions

This section presents the main results and key publications that result from this thesis, which is summarized in Table 1.1.

As the first step of this thesis, a systematic literature review of APLE is conducted in order to identify the current challenges of applying APLE to the software industry. This review of the state-of-the-art research in APLE is the initial point for defining the problem statement and establishing the main goal of this thesis. This systematic literature review has been published in:


The main result of this thesis is the definition of the APLA process to assist and guide architects in the iterative and incremental construction and evolution of product-lines architectures, while complying with the “be open to change” agile principle (see the top of the pyramid of Figure 1.3). The APLA process provides the capabilities of: (i) flexibility & adaptability at the time of defining PLAs, (ii) assistance in checking the integrity of the architecture, and (iii) guidance in the change decision-making process at the time of constructing and evolving PLAs. The mechanisms in which the APLA process is based on to provide these capabilities are also results of this thesis, and they are described as follows:

<table>
<thead>
<tr>
<th>Objective</th>
<th>Result</th>
<th>Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem statement</td>
<td>Systematic literature review of APLE</td>
<td>P1</td>
</tr>
<tr>
<td>OBJ 1.</td>
<td>Flexible-PLA model</td>
<td>P2</td>
</tr>
<tr>
<td>OBJ 2.</td>
<td>PLAK model</td>
<td>P3</td>
</tr>
<tr>
<td>OBJ 3.</td>
<td>CIA technique</td>
<td>P4</td>
</tr>
<tr>
<td>OBJ 4.</td>
<td>FPLA modeling framework</td>
<td>–</td>
</tr>
<tr>
<td>Main goal</td>
<td>Agile Product Line Architecting (APLA)</td>
<td>P5 &amp; P6</td>
</tr>
</tbody>
</table>

- Flexible-PLA model

The Flexible Product-Line Architecture (Flexible-PLA) model provides modeling primitives to specify flexibility and adaptability at the time of designing PLAs.
1. INTRODUCTION

This model supports the left-hand pyramid of Figure 1.3 and it has been published in:


• PLAK model

The Product-Line Architectural Knowledge (PLAK) model provides modeling primitives to document variability design rationale as well as to define the basic traceability linkage between features and product-line architectures. This model supports the right-hand pyramid of Figure 1.3 and it has been published in:


• Techniques for analyzing change impact

The techniques for analyzing change impact consist of a traceability-based algorithm and a rule-based inference engine. These traverse Flexible-PLA and PLAK models in order to determine the potential impact upon product-line architectures resulting from adding or changing features in each iteration of an APLE development process (see the pyramid core of Figure 1.3). This change impact analysis (CIA) technique has been published in:


• FPLA modeling framework

A modeling framework prototype called FPLA\(^1\) (see the pyramid base of Figure 1.3) implements the modeling primitives for describing, documenting and tracing product-line architecture, as well as the algorithm for analyzing change impact and model-to-text transformations to generate code.

\(^1\)It is available on https://syst.eui.upm.es/FPLA/home
1.4 Research Contributions

How these mechanisms and the FPLA modeling framework work together to support the APLA process is initially described in a first paper that focuses on flexible *working architectures*, whereas, the complete approach is described by a paper currently pending publication:


![Diagram](image)

**Figure 1.3: Thesis’ contributions** - This pyramid shows the supporting mechanisms to assist and guide the APLA process. The two pyramid cornerstones show the Flexible-PLA and PLAK models used for describing and documenting flexible and adaptive architectures. The pyramid core shows the change impact analysis technique. The pyramid base shows the modeling framework that supports the previous mechanisms. Finally, the top of the pyramid shows the APLA process that defines how previous mechanisms work.
1. INTRODUCTION

1.5 Research Methodology

Research in software engineering entails the creation of new models, methodologies and tools that are meant to help software engineers both to understand complex problems and to improve effectiveness and efficiency. Validity of the research results relies on the process of obtaining these results and the process of validating these results [Shaw, 2002]. Most importantly, it depends on the process of building the results, as well as the process of obtaining evidence that proves that the results are sound. Software engineering, as any other research discipline, needs guidance on the research process, i.e., a research methodology that assures good research, including validation techniques. According to the literature on research methodologies, and specifically research methods in software engineering, this thesis relies on a research methodology consistent with the principles of design science, as well as specific techniques such as systematic literature review and the use of case studies for specific issues such as literature analysis and research result validation.

Hevner et al. [2004] defines guidelines for design science in information systems research. Design science is essentially a problem-solving paradigm that seeks to create and validate artifacts intended to solve identified problems [Hevner et al., 2004]. Software engineering fits quite well the design-science paradigm in that understanding a problem domain and its solution are achieved through the building and application of designed artifacts. In software engineering, such artifacts are represented in a structured form that include models, methods, tools or frameworks. The design-science’s guidelines proposed by Hevner et al. [2004], and which have guided this thesis, are the following:

- Research must provide solutions to important and relevant problems. Section 1.1 stated the importance and relevance of the problem that this thesis pursues to solve.
- Research must provide verifiable contributions. Section 1.3 stated the specific objectives of this thesis and the criteria to validate the achievement of these objectives.
- Research must produce a viable artifact resulting from a search process. The resulting artifacts from our research are described in Chapters 5-8.
- Design artifacts must be rigorously validated. Chapter 9 provides empirical evidence that validates the achievement of the objectives of this thesis by evaluating the resulting artifacts.
1.5 Research Methodology

- Research must be communicated. Chapter 10 presents the key publications of the main contributions of this thesis.

Although there is extensive literature addressing validation in general, there is little literature addressing validation in software engineering. Shaw [2002] identified types of research validation in software engineering: analysis, experience, example, evaluation and persuasion. The most common kind of validation is evaluation through empirical research methods: observational methods (e.g. case studies or field studies) or experimental methods (e.g. controlled experiments or simulation). Evaluation methods are based on a set of criteria to validate the claims about artifacts to be evaluated [Hevner et al.] 2004. This thesis uses the case study method to validate the achievement of the proposed objectives based on the criteria that are defined for each objective. A case study is an empirical method that investigates contemporary phenomena in its natural context [Yin 2008] in order to search evidence, gain understanding, or test theories by using primarily qualitative analysis [Runeson & Höst 2009]. Therefore, the goal of the case study presented in Chapter 9 is to provide empirical evidence that validates the achievement of the objectives of this thesis in a real setting.

Finally, systematic literature review is a method to identify, evaluate and interpret all available relevant research on a specific research question or topic area by using a rigorous and auditable methodology [Kitchenham 2004]. Systematic literature reviews are important for different reasons [Budgen & Brereton 2006]: (i) to summarize existing evidence concerning a practice or technology, (ii) to identify where there are gaps in current research, (iii) to help position new research activities; and (iv) to examine how far a given hypothesis is supported or contradicted by the available empirical evidence. A formal and systematic literature review of agile product-line engineering has been conducted to build common understanding and to identify existing challenges in its implementation. The literature review method has been also used to acquire base knowledge on the foundations upon which the thesis relies: software evolution, software architectures, and software product lines.
1. INTRODUCTION

1.6 Thesis Overview

The remainder of this thesis is organized in the following chapters:

• **Chapter 2: Background.** This chapter provides the background required to understand the contribution of this thesis. This background focuses on: (i) *Software Evolution*. This section describes the trend in recent years to consider evolution a part of the development process in particular of agile software development processes, the well-known negative effects that evolution may cause unless an effort is made to prevent them, the techniques for supporting software evolution, and current challenges in software evolution. (ii) *Software Architecture*. This section establishes a conceptual base for the notion of software architecture, the main concepts of this field, including architecture knowledge, and provides an introduction to the role of software architecture in evolution. (iii) *Software Product Lines*. This section presents an overview of software product lines and presents the main concepts of this field, with special attention to variability, how variability is modeled in PLAs, and the role of variability in PLA evolution.

• **Chapter 3: Agile Product Line Engineering.** This chapter presents a systematic literature review of APLE and key findings, which uncover important challenges about how to integrate SPL and ASD. It is the starting point of this thesis, which aims to enable APLE.

• **Chapter 4: Preliminaries.** This chapter describes an empirical pilot study which has been conducted to exemplify the mechanisms that are described in the following chapters, as well as to prepare the case study that validates the complete approach presented in this thesis. In addition, this chapter also describes the agile method Scrum in which the main result of this thesis, i.e. the APLA process, is deployed.

• **Chapter 5: Flexible Product-Line Architectures.** This chapter defines, formalizes, and exemplifies the main concepts of the FPLA model which has been defined to describe PLAs.

• **Chapter 6: Product-Line Architectural Knowledge.** This chapter defines, formalizes, and exemplifies the main concepts of the PLAK model which has been defined to document knowledge on PLAs and trace requirements during their realization in the PLA.
1.6 Thesis Overview

- **Chapter 7: Change Impact Analysis in Product-Line Architectures.** This chapter defines, formalizes, and exemplifies the main concepts of the techniques that are used to analyze the impact of adding incremental functionality or changing requirements during the iterative and incremental construction and evolution of PLAs.

- **Chapter 8: Agile Product-Line Architecting process.** This chapter presents the APLA process, which assists and guides APLE practitioners in iteratively and incrementally constructing and evolving PLAs. This chapter also describes the FPLA modeling framework which provides software architects with the modeling primitives for describing, documenting and tracing PLAs, as well as (semi-)automatically change impact analysis and code generation.

- **Chapter 9: Case Study.** This chapter introduces the case study to validate the results of this thesis.

- **Chapter 10: Conclusions & Further Work.** This chapter summarizes the main contributions of the thesis and discusses future research.

The organization of this thesis is graphically represented in the Figure 1.4.
1. INTRODUCTION
Part II

STATE OF THE ART
Chapter 2

Background

This chapter provides the background to relevant research and practice of the software evolution, software architecture and software product line engineering areas. Main concepts and overviews of some of the important research themes in these areas are reviewed Finally, current challenges still open for each one of these areas are identified.

2.1 Software Evolution

Software engineering at large has recognized that software systems require continuous changes and enhancements to satisfy new and evolving user requirements, to adapt to new and emerging business models and organizations, to cope with technology innovation and to preserve the system from obsolescence. Therefore, the causes of requirements change range from those that are technical, due to the changing market of technology platforms, to those that come from business, due to the inherent volatility of the business context. Moreover, software systems are usually embedded in environments that are also continually changing [Godfrey & German, 2008].

In 1985, Lehman et al. stated the Law of Continuing Change [Lehman & Belady, 1985] which says the following: *A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.* In fact,
the largest part of lifecycle cost is concerned with software maintenance and evolution [Sommerville, 2006].

Software evolution, as it is understood in this thesis, can be defined as: *The process to implement changes and enhancements in requirements and technologies, that influence the system’s architectural structure while maintaining the architectural integrity* (based on [Bode & Riebisch, 2010; Breivold et al., 2011]).

The complexity of current software systems makes it harder to understand systems, as well as to make system changes. This may lead to making decisions that damage the architecture integrity of software systems. Several well-known negative effects of software evolution are architectural erosion [Perry & Wolf, 1992; van Gurp & Bosch, 2002], degeneration [Hochstein & Lindvall, 2005], drift [Perry & Wolf, 1992] and aging [Parnas, 1994]. Architectural erosion occurs when earlier design decisions are intentionally or accidentally violated during the implementation of software changes, which in turn causes software degradation and degeneration. Often, architecture and implementation drift apart so far that it is much easier to violate the architecture. Consequently, these problems make it much easier to fail in modifying software to meet changing needs, which is also known as software aging.

This section provides the background on software evolution required in order to understand the contribution of this thesis, including the description of the role of software evolution in the software development process, the negative effects of software evolution, the techniques supporting evolution and, finally, current challenges and conclusions.

### 2.1.1 Evolution as part of the Development Process

Traditionally, evolution came last in the software development lifecycle. Late lifecycle changes occur after at least one cycle of the development process has been completed and a working version of the software system has been deliberated. They usually make for high cost, both in money and effort [Williams & Carver, 2010]. Assuming that requirements are known before starting the software design phase seems very unrealistic [Mens et al., 2008]. On the contrary, requirements change during the entire software lifecycle.
Nowadays, evolution takes place in the initial stages of the lifecycle in what is now known as *incremental and evolutionary development*. The nature of software evolution has changed to a continuous process in which there’s no clear boundary between development and evolution [Boehm & Beck, 2010]. Early lifecycle changes occur on some higher-level views of the system such as models and documents. In this current paradigm, evolution becomes a natural part of development [Boehm & Beck, 2010]. As a result, the design for change is a vital part of development, including change impact analysis in order to plan the succession of changes required to implement evolution. Many development methods are iterative in nature, ensuring that evolution will occur. This is the case of *agile software development*.

Agile Software Development (ASD) or agility is just an umbrella term for a variety of methods that are based on the Agile Manifesto [Beck et al., 2001]. Several of the existing ASD methods are: Scrum [Schwaber & Beedle, 2002], eXtreme Programming (XP) [Beck, 2004], and Lean Development [Poppendieck & Poppendieck, 2006]. All of them implement (evolutionary) iterative incremental lifecycles and share some common values and principles. One of the main principles of ASD methodologies is to welcome changing requirements, even late in development. Therefore, agile teams do not try to avoid changes but try to understand what is behind them, seek to “embrace” them. Change is considered a normal condition of software development and is characteristic of every software development project. That means, ASD values responding to change over following a plan. The relevance of agile values and principles is increasing as large organizations are requiring their application [Lindvall et al., 2004]. These agile principles have been significantly accepted by industrial software companies [Chow & Cao, 2008] and have proved their effectiveness in projects with a large number of changing requirements [Dingsoyr et al., 2008; Dyba & Dingsoyr, 2008].

### 2.1.2 Negative effects of Software Evolution

From *Lehman’s laws of Evolution*, software systems must continually change to correct, improve or satisfy new requirements. Managing continuous change is not easy and comes at a cost. Therefore, because of such changes, the system is likely to degrade and its complexity and coupling increase over time, the design may be unintentionally ruined, and architecture and its implementation may drift apart too much. This makes it much easier to violate the original architectural design. Although the intention of software evolution is to enhance, improve, adapt, correct or extend the current functionality of a software system, changes may also lead to making decisions that
violate its fundamental architectural rules. The accumulation of such changes over time damages the architecture integrity of software systems making maintenance untenable unless an effort is made to minimize, prevent or repair these damages [de Silva & Balasubramaniam, 2012]. Although software aging is inevitable, “we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable” (based on [Parnas, 1994]).

These negative effects are known as erosion [Perry & Wolf, 1992], degeneration [Hochstein & Lindvall, 2005], drift [Perry & Wolf, 1992], loss of evolvability (aka decay [Mens et al., 2008]), and software aging [Parnas, 1994]. Architectural erosion is defined as “violation of the architecture which often leads to an increase in problems in the system and contribute to the increasing brittleness of a system” (based on [Perry & Wolf, 1992]). Architectural drift “results in a lack of coherence and clarity of form, which in turn makes it much easier to violate the architecture that has now become more obscured” (based on [Perry & Wolf, 1992]). When this occurs in an uncontrolled manner and the system is no longer maintainable, the system has degenerated [Hochstein & Lindvall, 2005]. These problems make it much easier to fail in modifying the software to meet changing needs, which is also known as software aging.

According to the observations of industrial case studies carried out by [van Gurp & Bosch, 2002], suboptimal realization of changes is due to the increasing complexity of the system not being fully understood. Often, earlier design decisions are not understood because they are not documented (a problem known as knowledge vaporization [Bosch, 2004] or knowledge dissipation [Visser, 2012]). Thus, these design decisions could be intentionally or accidentally violated during the implementation of software changes. In addition, design decisions could depend on other design decisions, and these dependencies must be reconsidered when a change has to be implemented. Therefore, a full understanding of the system and the changes to be made may help to counter, or even avoid, the abovementioned negative effects, and therefore preserve the architecture integrity of the system. This approach is classified as one that attempts to minimize erosion within the classification framework defined in [de Silva & Balasubramaniam, 2012]. Specifically, this approach is characterized in de Silva’s work as “those processes that ensure architecture conformance during development and maintenance activities”, such as the processes of architecture design documentation, architecture analysis, or dependency analysis, which in turn are the basis to support change impact analysis.
2.1 Software Evolution

2.1.3 Change Impact Analysis

Software organizations are aware of the fact that without sufficient understanding of the software systems they develop, evolution becomes expensive and unpredictable [Callo Arias et al., 2011]. Moreover, software systems are usually dependent on other systems, in such a way that changes in any of the latter may induce changes in any of the former. One of the major challenges in software evolution is the need to determine the effects that changes to a system have on the network of systems that depend on it [Visser, 2012]. Since changes may have ripple-effects, it is necessary to follow them up, a process known as change propagation. Change propagation was already identified [Yau et al., 1978] as an essential activity for accommodating the fact that software changes are rarely isolated [Mens et al., 2008]. To perform change propagation, several techniques have been proposed, such as change impact analysis, traceability analysis, dependency analysis, graph traversal, and effort estimation.

Software change impact analysis (CIA) [Arnold, 1996; Bohner, 1996] is a fundamental technique in software evolution, as it allows one to determine the potential effects upon a system resulting from a proposed change [Lee et al., 2000]. Therefore, CIA can be used to predict the effects of a change before it is implemented, possibly giving an estimate of the effort/cost to implement the change [Ramil & Lehman, 2000], as well as the potential risk involved in making the change [Mens et al., 2008]. This analysis can then be used to make better evolution decisions, such as whether or not the change should be carried out based on economic viability or other risks such as degradation of the system to evolve. CIA also permits trade-offs between a group of candidate or alternative solutions and can be used to select the most beneficial solution from among them [Pfleeger & Bohner, 1990].

Software change classifications and taxonomies [Buckley et al., 2005; Williams & Carver, 2010] have been used to qualitatively analyze the impact of making certain types of changes [Sun et al., 2010]. To quantitatively analyze the change impact, most approaches focus on source-code analysis [Kagdi et al., 2008; Kim et al., 2008], often limited to dependence (data and control flow) or call relationships, and therefore tied to the coding technique and programming language. This restricts the kinds and visibility of changes that can be analyzed. Therefore, it is more difficult to locate the impacted code when the changes originate in higher-level abstractions, such as requirements, than low-level code abstractions.

The work reported in Perry & Wolf, 1992 and Garlan, 2003 clearly show that software architectures bridge the gap between requirements and implementation.
2. BACKGROUND

Reasoning about changes at an architecture-level of abstraction is fundamental to evolve software systems [Bass et al., 2003; Garlan et al., 2009]. This is due to the fact that software architecture enhances system understanding by hiding low-level details and permitting restructuring where design decisions and rationale, constraints, and tradeoffs are made. The analysis of high-level structures, as well as architectural design decisions and rationale, is necessary to reason about the evolution of complex, large software systems. The impact analysis upon software architectures of changing requirements is an important area of CIA research, as requirements are almost never stable and fixed. Architecture-based change impact analysis gives insight into how much the architecture will be impacted in order to handle changing requirements —specifically architecturally significant requirements. Dependencies among the components of a system can make CIA more difficult since a change can lead to ripple-effects which are not obvious to detect. “The more we understand the impact, the less risk we take when making each change and the better that we can control software degradation resulting from change” [Pfleeger & Bohner, 1990].

2.1.4 Traceability Analysis

Traceability is the ability to describe and follow the life of a software artifact and a means of modeling the relations between software artifacts in an explicit way [Gotel & Finkelstein, 1994]. Traceability mechanisms, such as matrices, graph-based representations and cross referencing schemes, define and maintain relationships and links between artifacts involved in the software lifecycle [Aizenbud-Reshef et al., 2006]. These traceability links make it easier for software engineers to understand the relations and dependencies among software artifacts created during the software lifecycle.

The artifacts that are subjected to tracing are from requirements to source code, documentation or tests. The necessity of explicitly defining traceability between requirements and architecture has already been addressed by [Pohl et al., 2001; Ramesh & Jarke, 2001]. Ramesh & Jarke establish the basis for requirements traceability by defining a Reference Model for Requirements Traceability which describes four traceability types to establish relationships between requirements and design assets. They are: satisfaction, dependency, evolution, and rationale (see Table 2.1).

It is commonly understood that traceability supports the process of (i) checking the consistency between these artifacts in both forward and backward directions, i.e.
### 2.1 Software Evolution

Table 2.1: Traceability Types. From [Ramesh & Jarke, 2001].

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satisfaction</td>
<td>Occurs when a high-level object defines some kind of constraint or goal which should be fulfilled by one or more lower-level objects.</td>
</tr>
<tr>
<td>Dependency</td>
<td>Helps manage dependencies among objects typically at the same stage of development (e.g., dependencies imposed by a constraint, or created between objects when trying to satisfy some goals).</td>
</tr>
<tr>
<td>Evolution</td>
<td>Tracks the modification, refinement history of various objects.</td>
</tr>
<tr>
<td>Rationale</td>
<td>Identifies the reasons behind the creation, elimination or evolution of a lower-level object.</td>
</tr>
</tbody>
</table>

from requirements to code and from code to requirements, respectively [Aizenbud-Reshef et al., 2006]; (ii) analyzing the change impact over any development artifact, such as requirements specifications, models, architectures, documents, code, and tests [Mirakhorli & Cleland-Huang, 2011; Rochimah et al., 2007]; (iii) facilitating team communication, as traceability serves as a bridge between different development artifacts, as well as different stakeholders; (iv) preventing the software degeneration that evolution produces [Hochstein & Lindvall, 2005], (v) offering quality control to verify that requirements are met throughout all the development artifacts [Aizenbud-Reshef et al., 2006], and (vi) providing quantitative data to predict software quality before evolving a software product [Hayes et al., 2005].

Regarding software evolution, several works over the past decades have been proposed which successfully support software evolution by means of traceability analysis. Traceability is a widely-known technique to preserve knowledge through the product development lifecycle that can help with change impact analysis. Basically, when a change occurs in one artifact, traceability links are traversed to retrieve the linked, and therefore impacted, artifacts. A summary of these works can be found in [Rochimah et al., 2007]. Additionally, the work of Rochimah also identifies one factor that is essential to take into account when a traceability mechanism is defined: the granularity of the artifacts involved in a traceability link ranging from *coarse-grained*, such as requirements or components, to *fine-grained* since links can be created at the method level in source code.
2. BACKGROUND

2.1.5 Dependency Analysis

As software systems become more and more complex, dependencies become stronger, and thus evolving software systems become harder [Visser, 2012]. A general definition of dependency is a connection between two items in which changes to one item may cause changes to the other item as well. Generically, the concept of dependency in software engineering has been understood as: the degree to which each component relies on each one of the other components in the software system, in such a way that the fewer and simpler the connections between components, the easier it is to understand each component without reference to other components [Stevens et al., 1974]. A wide range of code-dependency analysis techniques exists, such as code searching techniques, repository mining of historical change data, and retrieval techniques by probabilistic analysis, several of which are based on structure matrices or conceptual graphs. Several of these solutions permit the detection of the most common occurrences of architectural violations (mainly regarding modularity and coupling) by detecting dependencies in the code that break constraints in the intended architecture [de Silva & Balasubramaniam, 2012].

However, this conceptual notion limits the scope of dependencies that can be analyzed to the solution space of software systems, as dependencies are necessary between high-level development artifacts. In [Callo Arias et al., 2011] the authors carry out a systematic review of dependency analysis solutions and classify dependencies as: (i) structural dependencies [Allen & Garlan, 1997; Stafford & Wolf, 1998], such as control or data flow dependencies; (ii) behavioral [Stafford & Wolf, 1998] or interaction [Allen & Garlan, 1997] dependencies, such as the use of interfaces; or (iii) traceability dependencies [Gotel & Finkelstein, 1994] which extend the scope of dependencies to both the problem space and the solution space of software systems.

2.1.6 Conclusions

The software research community has agreed that evolving software is more difficult than developing software from scratch, in part because more research in tools, processes, languages and mechanisms is necessary to support evolution. [Mens, 2010] and [Visser, 2012], supported by a group of experts in software evolution, discussed the future challenges (timeframe of 2015 and beyond) in their respective works. These challenges are as follows:
2.1 Software Evolution

- Better mechanisms for estimating and analyzing change. Automating the propagation of changes is necessary to take software dependencies into account and, thus, changes can be synchronized if required.

- Formalisms, tools, and techniques to deal with knowledge vaporization [Bosch 2004]. Rationalization of architectural decisions made during the architecture solution design helps to understand systems, and thus enables evolution.

- Formalisms, tools, and techniques to deal with software aging and assuring software product quality [Bennett & Rajlich 2000; Godfrey & German 2008; Mens 2009].

- Formalisms, tools, and techniques to deal with software degeneration [Hochstein & Lindvall 2005].

- Traceability between artefacts involved during software development (including models).

- Evolution at higher levels of abstraction, such as software architecture, and in emerging paradigms, such as model-driven software development [Beydeda et al. 2005], aspect-oriented software development [Filman et al. 2004; Kiczales et al. 2001], or agile software development [Beck et al. 2001; Shore & Warden 2007].

- Evolution of complex software systems, such as software product line evolution.

Changes in requirements are propagated to the architecture in order to design and implement them. In this sense, this thesis focuses on architecture-based evolution (aka architecture evolution), and specifically on static evolution, though it will be simply referred to as software evolution or evolution. This evolution may cause the negative effects mentioned before: architectural erosion, degeneration, drift, or aging, which threatens the integrity of systems. To preserve the architectural integrity of software systems, it is necessary to invest in mechanisms to support software evolution. This is described by [Lehman et al. 2000] as the “how” perspective of software evolution, i.e. the methods and tools intended to facilitate software evolution and the tasks that implement it. In this regard, this section has provided an overview of the techniques supporting evolution, such as traceability analysis and dependency analysis, both of them being key to address change impact analysis. This section has also introduced the role of software architecture in all of these techniques, as well as the necessity of architecture design documentation in order to avoid knowledge vaporization, thus minimizing architecture erosion. This thesis relies on architecture design documentation, as well as traceability and dependency analysis, to support change impact analysis. This provides architects with knowledge that guides and assists them in the change decision-making process and therefore helps them maintain the architectural integrity of software systems.
Presently, model-driven engineering (MDE [Schmidt, 2006]) is being applied to software evolution in such a way that models can be managed and transformed to facilitate and automate tasks involved in evolution by employing high-level abstractions. This emerging area of software engineering is known as model-driven software evolution (MoDSE) [Deridder et al., 2011]. This utilizes models to increment productivity and quality and reduce evolution costs by automating basic activities in software evolution. This thesis is based on this emerging area as models and traceability between models are used to (semi-)automatically support software evolution.

2.2 Software Architecture

Most of the research in the field of software architecture has focused on design [Bass et al., 2003; Buschmann et al., 1996], description [Medvidovic & Taylor, 2000] and assessment [Svämmberg & Mårtensson, 2007] of software architectures. This can be demonstrated by the large number of architecture evaluation methods and architecture description languages (ADL) that have been defined over recent years (see Section 2.2.3). In fact, in the ECSA 2010 keynote, Prof. P. Kruchten suggested the idea that “we do not need YAADL (Yet Another Architecture Description Language)”.

Architecture evolution has received less attention, although over the past few years most researchers have agreed with the fact that understanding and reasoning about system software architectures can provide the insight necessary to guide software maintenance and evolution tasks [Breivold et al., 2011; Garlan, 2000].

This section provides the background on software architectures required to understand the contribution of this thesis. This includes the definition of software architecture, main concepts and architecture description languages, architectural evolution and current challenges, architectural knowledge and current challenges, the role of architecture in Agile Software Development (ASD) and new approaches for agile (evolutionary) architecting, and finally conclusions.

2.2.1 Definition of Software Architecture

The work reported in [Perry & Wolf, 1992] represents the starting point for a community that actively focused on the notion and practical application of software architecture.
Perry & Wolf defined software architecture as a model composed of *elements, form* and *rationale* in which elements are either processing, data, or connecting elements with a particular form—properties and relationships—, whereas the rationale captures the motivation for the choice of elements and form. Since then, other definitions have been put forward. [Garlan & Shaw][1993], as well as the software architecture community, generally agree that the key elements of an architecture are the components, the connectors and the configuration. Hence, software architecture has been defined as follows:

*The architecture of a system describes its gross structure: top level design decisions, how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts [Garlan, 2000].*

*Software architectures provide high-level abstractions for representing the structure, behavior, and key properties of complex software systems [Garlan, 2003].*

*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass et al., 2003].*

Due to the importance of the architecture-level of systems development, a reliable consensus with a precise definition of a system’s architecture was required. The [IEEE Std 1471-2000][2007], and subsequently [ISO/IEC/IEEE Std 42010][2011], were created to provide a basis for thinking about the architecture of software-intensive systems. That is to say that they were created to facilitate the expression and communication of architectures through standardization of elements and practices for architectural description.

*The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. [IEEE Std 1471-2000][2007]*

Software architectures comprise a wide-scope of tasks: (i) analysis and description of the properties of systems at a high level of abstraction; (ii) validation of software requirements; (iii) estimation of the cost of the development and maintenance processes;
2. BACKGROUND

(iv) reuse of software; and (v) establishment of the bases and guidelines for the design of large complex software systems [Perry & Wolf 1992]. It follows that software architectures play a vital role in developing large software systems. In fact, it has long been recognized that architecture has a strong influence over the lifecycle of a system and as a critical element in successful development as well as in the successful evolution of software-intensive systems [Brown & McDermid 2007; Kruchten et al. 2006b].

2.2.2 Main Concepts of Software Architectures

The software architecture community generally agrees that the building blocks of an architecture are the components, connector and their configuration. They are defined as follows:

- **Components** are considered as black boxes showing a high level of encapsulation and abstraction, interactions with which being restricted to their interfaces. There are a lot of definitions for the term component, although the most widely used definition in the software architecture field is the one proposed by [Szyperski 1998].

  A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

- **Connectors** model interactions among components and rules that govern those interactions [Medvidovic & Taylor 2000]. There are two different perspectives for defining connectors. Connectors can be considered (i) first-class entities in software architecture representation or (ii) merely the connection between two components. In regard to the first-class consideration, [Shaw 1996] defines connectors as follows:

  Connectors are the locus of relations among components. They mediate interactions but are not things to be hooked up (they are, rather, the hookers-up). Each connector has a protocol specification that defines its properties. These properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction such as ordering, performance, etc.

- **Architectural configurations**, or topologies, are connected graphs of components and connectors that describe architectural structure [Medvidovic & Taylor 2000].
2.2 Software Architecture

Therefore, configurations specify components connected to other components through connectors which enable proper communication. The points through which components can interact are called *ports*.

- Ports publish the behavior —services— provided and required by components, this means, ports publish an *interface* or part of itself.
- Interfaces specify the services a component provides or requires.
- Finally, *attachments* establish the communication channels between components and connectors.

A key quality at configuration level is *compositionality* which allows describing systems at different levels of abstraction. Hierarchical composition allows architects to define complex components that several authors have referred to as *composite components* [Magee & Kramer, 1996], *subsystems* [Jacobson et al., 1997], *representations*¹ [Garlan et al., 2000], or what the architecture community generally refers to as *systems*.

- Systems represent architectural configurations that are made up of connectors and components that can be built in a hierarchical way. Compositional relationships, i.e. the relationship between a system and an architectural element of at higher abstraction level, are defined by means of *bindings*.
- Bindings establish the mappings between the internal and external interfaces of a system [Garlan, 2000], this means that bindings establish a connection between a system port and a port of one of its sub-components.

Other concepts that are used throughout this document are view and viewpoint which are defined as follows:

- A *view* is a representation of a whole system from the perspective of a related set of concerns of the system stakeholders. The term *view* is used to refer to the expression of a system’s architecture with respect to a particular viewpoint [IEEE Std 1471-2000, 2007].

¹The Acme ADL defines representations as hierarchical descriptions of architectures. Acme permits any component or connector to be represented by one or more detailed, lower-level descriptions [Garlan et al., 2000].
2. BACKGROUND

- A viewpoint is a specification of the conventions for constructing and using a view. A viewpoint defines a pattern or template from which it is possible to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis [IEEE Std 1471-2000 2007].

View-based description has emerged as the “best of breed” approach for dealing with software architectures [Clements et al. 2003]. The V&B approach [Clements et al. 2010] is based on describing software architectures by using relevant views and then documenting the information that applies across the views. There are several approaches that propose a fixed set of viewpoints to construct software architectures, as well as the views that materialize them. The Rational Unified Process (RUP) is built on the Kruchten’s 4+1 view model which defines the logical, process, physical, development and scenarios views [Kruchten 1995]. The work by Rozanski & Woods, 2005 defines six main viewpoints: functional, information, concurrency, development, deployment, and operational. Based on the intended uses of the architecture (eg. implementation guidelines or change impact analysis), other approaches permit architects the freedom to select the most relevant views according to these uses. Hence, Bass et al. [2003] and Clements et al. 2010 propose to select the most relevant views based on the architecture’s intention and/or stakeholder’s concerns, among the module, component & connector, and allocation views. The IEEE Std 1471-2000 2007 mentions the structural, behavioral and physical viewpoints, among others. Whereas, the RM-ODP ISO/IEC 10746 1998 establishes the enterprise, information, computational, engineering, and technology viewpoints.

2.2.3 Architecture Description Language

Software architectures are generally expressed using an Architecture Description Language (ADL). ADLs are formal notations for representing architectural designs by providing both a conceptual framework and a concrete syntax for characterizing software architectures [Garlan 2000]. Rigorous representations of software architecture allow architectural analysis and even code generation. As a result, ADLs have been proposed as modeling notations to support architecture-based development [Medvidovic & Taylor 2000] as they may capture the elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.
2.2 Software Architecture

The work by Medvidovic & Taylor [2000] was the first attempt to get consensus in the architectural community on what an ADL should provide. The work by Medvidovic & Taylor establishes the minimum concepts which should be supported by an ADL: components, their interfaces, connectors, and configurations. It also summarizes the wide-range of concepts that the different ADLs, defined over recent years, have tried to provide: components, connections, compositionality, configurations, semantics, constraints, non-functional properties, communication protocols, underlying formal model, and tool support for modeling, analysis, evaluation, and code generation.

Several examples of ADLs are: xADL [Dashofy et al., 2002], Acme1 Garlan et al. [2000], AspectualACME Batista et al. [2006], Wright Allen & Garlan [1997], Darwin Magee & Kramer [1996], Unicon Shaw et al. [1995], Leda Canal et al. [1999], C2SADEL Medvidovic et al. [1996], Rapide Luckham et al. [1995], PiLar Cuesta [2002] or PRISMA Pérez [2006]. They provide mechanisms for describing architectural structure and most of them provide mechanisms for adding semantics2 to that structure. Several mechanisms used to specify system semantics are pi-calculus and finite state process languages.

2.2.4 Architecture Evolution

Software architecture is inevitably subject to evolution. Knowing and understanding the set of principal design decisions made during the system’s conceptualization and development is essential to successful system evolution [Taylor et al., 2009]. Architecture evolution permits planning and system restructuring at a high level of abstraction where design decisions, quality and business tradeoffs can be understood and analyzed [Garlan et al., 2009].

Most of the architecture evolution research has focused on the goal of designing software architectures which anticipate, facilitate and embrace change by emphasizing quality attributes such as evolvability, changeability/modifiability, flexibility, extensibility, and portability, among others [Bode & Riebisch, 2010] Breivold & Crnkovic [2009]. Analyzing and improving software evolvability at an architecture-level has been a recurring issue, which is summarized in [Breivold et al., 2011]. Another part of the research has focused on capturing architectural changes as first-class entities. Several approaches in this research line propose formal representations for change in software

1Strictly speaking, ACME is an interchange language and not an ADL although it contains a number of ADL-like features [Medvidovic & Taylor, 2000].
2Also referred as behavior or dynamism.
2. BACKGROUND

Architecture descriptions and the use of verification mechanisms to ensure that the evolved architecture is valid [Barais et al., 2008; Jansen & Bosch, 2005; McVeigh et al., 2011]. On the same topic, another research line has focused on automating architectural evolution by considering changes as model transformations in the context of model-driven engineering [Biehl & Töndrungren, 2010; Graaf et al., 2008; Navarro & Cuesta, 2008]. Finally, Garlan et al. [2009] propose the definition of architectural evolution patterns, whereas van der Hoek et al. [2001], Roshandel et al. [2004], Abi-Antoun et al. [2006], and Garg et al. [2003] propose architectural versioning to track the evolution of the architectural elements such as components and connectors. Several of these proposals provide supporting tool such as Mae [Roshandel et al., 2004] and Ménage [Garg et al., 2003].

The following three subsections delve into architecture evolution taxonomies, the support that current ADLs offer and the mechanisms that they use to realize the change, and the current challenges that still exist in architecture evolution, respectively.

2.2.4.1 Architecture Evolution Taxonomy

Architecture evolution has been commonly categorized in terms of “the time of change” as static or dynamic evolution. Static evolution refers to offline changes to an architecture that are implemented at design time [Medvidovic & Taylor, 2000], as opposed to dynamic evolution that refers to modifying the architecture and enacting those modifications in the system while the system is being executed [Andrade & Fiadeiro, 2003; Costa-Soria, 2011]. For the purpose of this thesis, the following taxonomy, which focuses on the effect of evolution, is especially interesting. In short, changes may affect the structure, the behavior, or the rationale. The following taxonomy has been adapted from [Buckley et al., 2005; Díaz et al., 2011; Kim et al., 2008; Nedstam et al., 2004; Svaenbø & Bosch, 2000; Tahvildari et al., 1999; Williams & Carver, 2010]:

- **Structural evolution** refers to changes that affect only the architecture structure and do not modify the external functionality of systems. With changes that affect only the architecture structure, we refer to changes in the configuration of components and the connectors which model the interactions among components. These changes may be unnoticeable to users and are implemented by means
of architectural restructuring\(^1\) and refactoring\(^2\). Their purpose is to (i) enhance (design-time) quality attributes such as modifiability, understandability or reusability, among others —preventative changes— or (ii) accommodate new platforms or standards —adaptive changes.

- **Interface evolution** refers to perfective changes that affect user-observable features—functional or non-functional characteristics—of a software system. These changes affect the portion of the architecture that is responsible for providing functional features and/or non-functional features (i.e. run-time quality attributes such as availability, performance, reliability, etc.). These changes could impact the architecture structure or not. That means that interface evolution could imply structural evolution or not. These types of changes include adding, removing and/or modifying components, their interfaces or connectors. Since these types of changes imply semantic change (i.e. change in behavior), there may be ripple-effects which cannot be completely analyzed or done automatically, making manual intervention necessary.

- **Design rationale evolution** refers to changes in architectural design decisions, their rationale, or the dependencies between architectural design decisions, which may imply (or not) either structural or interface evolution, or both of them.

Structural, interface and rationale evolution, may have ripple-effects, also known as *change propagation*. That means that changes may have a direct effect due to specific relationships between the change and one or more architectural elements, or may have an indirect effect due to dependencies between design decisions, or dependencies between components and connections (coupling).

Structural evolution, in its strictest sense or mixed with interface evolution, may affect the *external structure* of the architecture or the *internal structure* of their elements, as is described below:

- External structural evolution may affect the architectural configuration. It can be realized through: (i) addition, deletion or replacement of architectural elements; (ii) *kidnapping*, i.e. movement of an entire architectural element from

\(^1\)The transformation from one representation form [of a software system] to another, at the same relative abstraction level, while preserving the subject system’s external behavior (functionality and semantics) [Chikofsky & Cross 1990].

\(^2\)Architecture-oriented refactoring is an approach to restructure the software architecture of a system and to improve its internal software quality [Wohlfarth & Riebisch 2006].
2. BACKGROUND

one subsystem to another; (iii) splitting, i.e. division of the functions of an architectural element into two or more distinct elements; and (iv) relocating, i.e. movement of functionality from one architectural element to another. Several of these changes may imply the connection, reconnection or disconnection of the involved architectural elements.

- Internal structural evolution may affect the components and connectors. Internal structural evolution at component level, aka component evolution \cite{Medvidovic2000}, can be defined as the modification of component interfaces, behavior, or implementation. Internal structural evolution at connector level, aka connector evolution \cite{Medvidovic2000}, can be defined as the modification of the connector interaction protocols.

2.2.4.2 Architecture Evolution Support

ADLs can support static, dynamic evolution or both of them. For example, static evolution is supported by Acme, C2SADEL and Wright, while dynamic evolution is addressed by PiLar \cite{Cuesta2002} and Dynamic PRISMA \cite{Costa-Soria2011}. In addition, several evolution environments exist, such as ArchStudio \cite{Oreizy1998}, DRADEL \cite{Medvidovic1999}, Ménage \cite{Garg2003} or Mae \cite{Roshandel2004, van der Hoek1999, van der Hoek2001}, which complement evolution support with versioning or configuration management.

ADLs, such as Acme, C2SADEL, Darwin, Rapide, or Wright, support external structural evolution by changing the architectural configuration. Most of them support internal structural evolution by employing compositionality mechanisms of software architectures such as subtyping of (component/connectors) types\footnote{Types are abstractions that encapsulate functionality into reusable blocks. Hence, a component type can be instantiated multiple times in a single architecture or it may be reused across architectures \cite{Medvidovic2000}.} and refinement of components \cite{Medvidovic2000}, or the use of representations \cite{Garlan2000}. This is the case of C2SADEL or Acme, respectively. Others ADLs (e.g. Rapide) rely on implementation techniques such as inheritance, extensions, parameterization, configuration, or aspect-oriented programming\footnote{AOP \cite{Kiczales2001}} \cite{Svahnberg2005}. However, over the past few years, several of these techniques such as aspect-oriented software development (AOSD)\footnote{AOSD proposes the separation of the crosscutting-concerns of software systems into separate entities, which are called aspects.} has emerged not only to apply aspect-oriented to the implementation.
stage, but also to apply it to every stage of the software lifecycle. One of these stages is software architecture. Aspect-oriented software architectures \cite{Cuesta:2005,Navasa:2009,Perez:2006,Perez:2006} claim to improve reusability and adaptability of software architecture based on the concepts of \textit{separation of concerns} of architectural components. Therefore, internal structural evolution at architecture-level could be addressed in terms of aspects, as AspectualACME \cite{Batista:2006} and TranSAT \cite{Barais:2008} do.

### 2.2.4.3 Challenges in Architecture Evolution

As software systems become more complex and large, traditional ADLs do not provide mechanisms to completely support architecture evolution due to the lack of first-class entities that represent the main architectural design decisions made during conceptualization and development \cite{Jansen:2004}. As a result, support for architecture evolution is lacking as architects are not able to understand the decisions and reasons which have driven a specific architectural design, how the architecture will be impacted, or whether the changes may violate any design constraints or early design decisions.

Software evolution needs assistance in reasoning about what kind of changes can be made independently, and which changes require coordinated modifications \cite{Garlan:2009}. Reasoning about changes at an architecture-level of abstraction is fundamental to evolve software systems as software architecture provides system understanding, hiding low-level details and permits restructuring where design decisions, rationale, constraints, and tradeoffs are made. However, most of the research has focused on the elements and form of software architectures, whereas the rationale has often been neglected. As a result, architects are not able to understand the decisions and reasons which have driven a specific architectural design when a change impacts that design. This fact makes it easier to violate any design constraints or early design decisions. This problem is known as \textit{knowledge vaporization} and it is the key cause of architectural erosion, degeneration, drift or aging during architecture evolution. Without the knowledge of the decisions, their dependencies and the reasons driving the design, the integrity of architecture can not be guaranteed.

The first author to introduce the problem caused by the focus on the resulting architectural design, instead of the decision driving the design, was \cite{Bosch:2004}. Subsequently, \cite{Kruchten:2004} and \cite{Jansen:2005} introduced architectural design decisions as first-class entities in software architecture representation, although
2. BACKGROUND

previously other authors also recognized the importance of documenting software architecture [Clements et al., 2002] for which they coined the term known as Architectural Knowledge (AK). This emphasis on capturing, documenting and managing architectural knowledge has quickly achieved a great popularity [Ali Babar et al., 2009a; Dutoit et al., 2006; Kruchten et al., 2009; Tang et al., 2006].

2.2.5 Architectural Knowledge (AK)

Existing notational approaches to software architectures, such as ADLs, typically focus on the resulting architecture itself, i.e. the architectural design, and neglect to explicitly document the design decisions and related knowledge that represent the reasoning behind this result, i.e. the rationale of the architecture.

Because architectures manage complex interactions and interdependencies between architectural elements, architectural descriptions should not only include the design of the solution but also should explicitly document the design decisions that resulted in the architecture, as well as the rationale underlying the design decisions. Design decisions typically crosscut the architecture, affecting multiple architectural elements, such as components and connectors, and they often become intimately intertwined with other design decisions [Bosch, 2004].

Letting this knowledge resides only implicitly in people’s heads is particularly harmful during staff turnover, whereas, if this knowledge is explicitly codified it can be automated or semi-automated with appropriate tool support [Liang & Avgeriou, 2009]. This opens up the possibility for reasoning, analysis, and large-scale knowledge reuse. The first approach is called personalization, the second is called codification [Hansen et al., 1999]. Also, hybrid approaches exist that combine the previous two [Farenhorst et al., 2007]. It is possible to distinguish between two types of explicit knowledge based on the form in which it is codified: documented knowledge, which is expressed in natural language and/or images, and formal knowledge, which is expressed using a formal language or model where semantics are defined (e.g. ADLs or domain-specific models) [Liang & Avgeriou, 2009]. When systems grow in size and complexity, formalized approaches for documenting knowledge are necessary in order to support a great number of concepts and relationships. It provides a common language for communication, traceability, and to automate or semi-automate tasks such as change impact analysis, or checking the completeness of an architecture [Jansen et al., 2009].
As a result, in the recent years, most researches of the area emphasize the need to formalize architectural knowledge (AK). There is not a single encompassing definition of what AK entails. It has been agreed upon that a software architecture should be seen as the set of principal design decisions made during the system’s conceptualization and development [Taylor et al., 2009]; “the result of a set of design decisions rather than a set of components and connectors” [Bosch, 2004]. From these statements, some authors have established the following definitions to describe AK:

- **Architectural Design Decisions** are descriptions of the set of architectural changes to the software architecture that (partially) realize one or more requirements on a given architecture (adapted from [Jansen et al., 2007]).

- **Design Rationale** is the abstract grouping of reasons that justify the decisions that are made during the design process [Tang et al., 2006].

- **AK = Design + Design Decisions** [Kruchten et al., 2006a]

- **AK = Solution (Design) + Design Decisions + Rationale** [van Vliet, 2008].

AK provides the basis to improve understanding and rationalization of architectural decisions made during the design of an architectural solution, i.e. reconstruction and explanation of the rationale behind them [Ali Babar et al., 2009a; Bass et al., 2003; Bosch, 2004; Clements et al., 2010; Dutoit et al., 2006; Kruchten et al., 2009]. Other researchers also emphasize the need to document design rationale for maintaining and evolving architectural artifacts [Bosch, 2004; Farenhorst & de Boer, 2009; Kruchten et al., 2009] and to avoid architectural erosion [de Silva & Balasubramaniam, 2012], degeneration, drift, or aging. The works by [Tang et al., 2006] and [Falessi et al., 2006] present empirical evidence about how important design rationale is considered by practitioners, and it has been recently incorporated into [ISO/IEC/IEEE Std 42010, 2011] which is the revision of [IEEE Std 1471-2000, 2007].

The failure of documenting AK is often due to (i) the time and budget constraints of projects, and (ii) the lack of standards and processes to guide why, how, what and when design rationale should be documented [Tang et al., 2006]. This causes the knowledge vaporization that prevents reasoning about the architecture itself and its evolution, and may result in expensive system evolution, lack of stakeholders communication, and limited reusability of architecture [Jansen et al., 2007].
2. BACKGROUND

To prevent failure of documenting AK, the software architecture community has worked to provide some formal representations and tools supporting design rationale [Babar & Gorton, 2007; Capilla et al., 2007; Jansen et al., 2007; Tang et al., 2007a]. Although these representations and tools initially create additional overhead work for capturing design decisions and their rationale during the early development stages, it has been demonstrated that this overhead is paid off in later maintenance and evolution stages [Kruchten et al., 2009]. The expected benefits from explicit capture and documentation of design rationale are:

- General understanding of a system, which is particularly useful during staff turnover [Kruchten et al., 2009].
- Comprehension of the implications of design decisions on meeting the requirements or satisfying constraints [Bass et al., 2003].
- Better understanding of change impact [Kruchten et al., 2009; Tang et al., 2007b] by reasoning about changes (or new requirements) at an architecture-level of abstraction.
- Knowledge sharing [Farenhorst et al., 2008] and improvement of stakeholders communication.

Therefore, although a significant effort is necessary in documenting the AK, stakeholders must be convinced that they will get a good return on their investment [Jansen et al., 2009].

2.2.5.1 Documenting & Formalizing Architectural Knowledge

Several approaches for documenting AK have been proposed. Interesting comparisons of these approaches are presented in [Farenhorst & de Boer, 2009; Shahin et al., 2009; Tang et al., 2010, 2006]. Next, an overview of these approaches is presented, which we have categorized as follows:

- **Structural approaches**: These approaches explicitly represent the design deliberation process, including alternatives and arguments. Some examples are: the Issue-Based Information System (IBIS) [Kunz & Rittel, 1970] or the Procedural Hierarchy of Issues (PHI) [McCall, 1987].
2.2 Software Architecture

- **Template-based approaches**: These approaches provide a guideline for textual descriptions of most relevant information about design decisions. Tyree & Akerman [2005] present a template to codify architectural design decisions together with their rationale and several other properties relevant to that decision, such as the associated constraints, a timestamp, and a short description.

- **High-level structural approaches**: These approaches are based on structures that capture and store architecture design decisions in such a way that learning and reasoning is stimulated. Kruchten et al. [2006a] ontology proposes a more formal approach of organizing and visualizing different types of decisions. According to the Kruchten’s ontology, a design decision comprises the decision itself, the scope of the decision, the rationale of the decision, cost, risk, author, timestamp and state. Relationships between design decisions (e.g. enables or conflicts with) are also considered.

- **Annotating approaches**: These approaches annotate architectural documents to make their knowledge explicit. The approach of Jansen et al. [2009] enriches traditional documentation sources (e.g. word documents) according to a formal metamodel.

- **Design decision link approaches**: These approaches propose models to describe the relationships between the traditional software artifacts and the design decisions, including the semantics of these relationships — or links. Several of these approaches pursue tracing knowledge of the problem to knowledge of the solution Babar et al. [2006], Königemann & Zimmermann [2010] by connecting decisions to requirements and architectures. Others also manage the dependencies between decisions Zimmermann et al. [2009]. Some of them are supported by tools: AREL Tang et al. [2007a], ADDSS Capilla et al. [2007], Archium Jansen et al. [2007], and PAKME Babar & Gorton [2007]. Similarly, Galster et al. [2006] and de Boer & van Vliet [2009] explore common ground between the requirements and architecture. Based on the idea that there is no fundamental difference between architecturally significant requirements and architectural decisions, they propose integrated methods and tools for architecture knowledge management that overarch requirements engineering and architecting.

### 2.2.5.2 Managing Architectural Knowledge

Three main approaches for managing AK have been proposed: those based on reasoning about codified architectural solutions, those based on discovering architectural
knowledge, and those based on reusing codified generic knowledge. Next, an overview of these approaches is presented, which we have categorized as follows:

- Approaches for reasoning about codified architectural solutions: These approaches focus on modeling architectural knowledge concepts in such a way that learning and reasoning is stimulated. It allows the rationalization of made architectural decisions, i.e. reconstruction and explanation of the “why” behind them. Several examples are the tools: AREL [Tang et al., 2007a], ADDSS [Capilla et al., 2007], Archium [Jansen et al., 2007], PAKME [Babar & Gorton, 2007], or Kruchten’s ontology. The proper visualization techniques of the architectural knowledge is very useful for architects in the decision making process.

- Approaches for discovering architecture knowledge: They provide architects with a reading guide when looking for specific architectural knowledge [de Boer & van Vliet, 2008]. To that end, the authors propose the use of an intelligent discovery method based on latent semantic analysis to quickly reach the important knowledge while skipping less relevant documentation.

- Approaches for the reuse of codified generic knowledge: They propose the use of decision templates, architectural guidelines, patterns, etc. to make decisions for a single application. “Engineers need reference material that organizes what we know about architecture into an accessible, operational body of knowledge” [Shaw & Clements, 2006].

2.2.6 Agile (Evolutionary) Architecting

The work of [Falessi et al., 2010] found that agile practitioners perceive software architecture as relevant on the basis of aspects such as (i) communication and understanding of software systems, (ii) rationalization of previous design decisions, which is the input for subsequent design decisions, (iii) documentation of rationale, assumptions, constraints and other dependencies necessary to evaluate design alternatives, (iv) scaling of agile practices to large projects, (v) documentation of points of flexibility within the system to support future requirements, and (vi) system planning and budgeting.

Advocates of a balance between architecture and agility propose that the architecture emerges gradually iteration after iteration, as a result of small successive refactoring [Abrahamsson et al., 2010; Booch, 2010; Madison, 2010; Nord & Tomayko, 2000]. In this sense, hybrid approaches, such as the concepts of agile architecting
2.2 Software Architecture

or evolutionary architecting, have gained increasing acceptance in recent years. Bohm & Turner 2004 demonstrate a solution for adapting XP to develop large-scale software systems by introducing elements of high-level architectural plans to provide essential big-picture information as well as using design patterns and architectural solutions, rather than simple design, to handle foreseeable change. Cockburn 2006 proposes starting with a simple architecture that handles all the big rocks. Then, it can evolve or be refactored as other requirements appear; but it should not be an objective to have the architecture at the end of the project. McMahon 2005 recommends agile architecting in two levels. The first level develops a high-level agile architecture including the major system components, assumptions, and a brief description of each component. The second level focuses on the high-risk areas for each iteration (big rocks). Finally, Ali Babar et al. 2009b analyze the role of the architecture in ASD through an industrial case study in which software product lines and agile practices are integrated. The authors describe a development process that consists of three sub-processes: product line platform, exploration before agile product development, and agile product development. It was found that architecture and architectural communication support these three processes and, reciprocally, these processes may update product architectures by means of refactoring.

Most of these approaches invest in a first architecture —what some people call a zero-feature release Beck 2004, McMahon 2005: “getting an architecture sufficiently right early without necessarily resorting to big upfront design” Kruchten 2009. It means that it will take longer to get to code, i.e. in such a release, the architecture is in place, but the organization does not deliver any user-visible features to the customer Nord & Tomayko 2006. Conversely, other authors believe in continuous architectural refactoring, starting on simplicity and flexibility Booch 2010.

2.2.7 Conclusions

It has long been recognized that architecture has a strong influence over the lifecycle of a system. Architecture is also been recognized as a critical element in the successful development as well as in the successful evolution of software-intensive systems Brown & McDermid 2007, Kruchten et al. 2006b. Reasoning about the architecture can provide the insight necessary to make decisions about proposed changes Bass et al. 2003 and to guide the software maintenance and evolution tasks. Supporting evolution at the architecture-level allows planning and system restructuring at a high level of abstraction, where design decisions are made, and quality and business tradeoffs can
be analyzed. In fact, design decisions capture principles and guidelines for designing and evolving the architecture.

Many existing notational approaches for describing and documenting software architectures typically focus on architectural design (solution). In the recent years, research results have emphasized the need to document the design decisions that are made to define the solution, as well as the rationale behind these design decisions. Documenting and codifying all this AK is essential for evolving software in a controlled way, as it helps to (i) avoid the problem of the so-called knowledge vaporization, and (ii) minimize the negative effects that evolution has over the architecture without compromising software integrity. For that purpose, some formal representations and tools have been proposed for capturing, documenting and managing AK with the goal of reasoning about architecture solutions, reusing generic AK, or discovering AK. However, to the best of our knowledge, there is little work on the fundamental issue of how to use these rationale-based techniques to support software system evolution. This thesis considers that the great challenge is to provide a comprehensive methodology to guide how rationale-based techniques could be used to successfully evolve systems. In this regard, it would be desirable to exploit the architectural knowledge to automatically or semi-automatically analyze the impact over the architecture of making a change in requirements to evolve a software system. Change-impact knowledge may help to understand the change, and thus to minimize architectural erosion and degeneration. To that end, design decisions, their rationale, dependencies, constraints, etc. have to be considered in order to preserve the architectural integrity of software systems.

Finally, starting from the hypothesis that architecture plays an important role in evolutionary development paradigms (such as ASD), by enabling communication, documenting the design decisions and rationale, and scaling of agile practices to large projects among others, our understanding is that techniques supporting architecture evolution can help agile architecting. In this way, architecture can be aligned with agile values and principles. This thesis pursues to extend the “be open to change” agile principle also to large and complex software systems, specifically to family-based software development (also known as software product line engineering), through the agile construction and evolution of product-lines architectures.
2.3 Software Product Lines

Over the last decade, new software development paradigms emerged to increase the productivity of software companies and to reduce time to market. One of them is Software Product Line Engineering (SPLE) [Clements & Northrop, 2002; Pohl et al., 2005]. SPLE promises better quality, lower cost, and faster time-to-market [Linden et al., 2007] through systematic reuse of software assets and mass customization. Hence, SPLE exploits the commonality found in a set of products by investing in the upfront design of the product-line platform —i.e. the design of the common set of reusable core-assets, their variability and the PLA. Then, these assets are assembled into customer-specific products just by deriving the existing variability.

The software industry has recognized the strategic importance of software product lines [van der Linden, 2002], especially those companies that develop products for the same family or domain. Representative examples are mobile phones, car electronics or financial services.

A software product line (SPL) can be defined as follows: A set of software applications —software-intensive systems or software products— sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Clements & Northrop, 2002].

This prescribed way refers to the principles of SPLE [Pohl et al., 2005]. SPLE is a paradigm to develop software product lines using reusable core assets, i.e. assets that are common to the products of a family and that are susceptible to mass customization. SPLE is based on the idea that products of the same domain or family share a lot of commonalities, therefore the definition of a platform, from which products are derived and customized, may make sense. Therefore, the process of developing SPLs consists in several basic steps as follows: (i) identification of commonalities and variabilities over the family, (ii) definition of a platform for building products in a specific domain, (iii) effective management of variations across the products, and (iv) reuse and exploitation of the existing variability to build applications of the product-line.

This section provides the required background of software product lines to understand the contribution of this thesis, including the description of the main concepts in SPLE and the product-line development process, the review of most relevant approaches for modeling variability and documenting its rationale, and finally the description of the state-of-the-art research in product-line architecture evolution (PLA evolution).
2. BACKGROUND

2.3.1 Main Concepts of Software Product Lines

The SPL community generally agrees that the building blocks of a product-line are reusable core assets, which support different degrees of customization through defining variability points. These concepts are defined as follows:

- **Core asset base** [Clements & Northrop, 2002] or **platform** [Pohl et al., 2005]: assets that forms the basis—a common structure—from which a set of products of a same family can be efficiently developed. That core asset base could include requirement statements, documentation and specifications, domain models, architecture descriptions, reusable software components, test cases, work plans, process descriptions, etc.

- **Asset**: package of relevant artifacts that provide a solution to a given problem. Assets can be of different granularity, may allow different degrees of customization (variability), can be applied to different phases of software development, and can be reused in different phases [Bachmann et al., 2004].

- **Artifact**: piece of information (requirements specification, architecture, code, tests, etc.) that is produced, modified, or used by a process and may take various shapes (e.g. models, model elements, documents) [Thiel & Hein, 2002].

- **Feature**: a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems. This term starts to be extensively-used when the Feature-Oriented Domain Analysis (FODA) method [Kang et al., 1990] introduces the feature modeling technique for capturing commonality and variability of SPLs.

- **Variability**: ability to quickly achieve change in preplanned ways [Clements et al., 2010], also known as **anticipated change**, i.e. change that is mostly foreseen [Bachmann & Bass, 2001]. Variability is the basis for mass customization [Pohl et al., 2005], i.e. ability to specify flexibility to enable the development of customized applications from a SPL. Variability is achieved by intentionally defining the variation type—optional, alternative and multiple—, the time in which variation occurs—at design, compilation or run time—, the places where products can differ, and its rationale—i.e. intent and motivation to define that variability. It is specified in terms of variability points (aka variation points) and variants.

- **Variability Point**: explicit engineering decision which permits several alternative, multiple or optional variants in regard to selected assets of system development [Bachmann et al., 2004]. Variability points are the exact places in the core
2.3 Software Product Lines

assets where a specific kind of flexibility has been built in, i.e. the locations
at which variations occur [Clements et al., 2010]. A variability point could
represent, for instance, the choice among a number of functional features available
to the users, different structures and interaction patterns in the product-line
architecture, or alternative software components in product implementation.

- **Variant**: option for a specific decision that has been left open.

### 2.3.2 Product-Line Development Process

The literature has established two essential processes to successfully develop SPLs [Pohl et al., 2005]: Domain Engineering and Application Engineering. Although there is a consensus about these processes, several approaches propose different sets of activities and practices [Clements & Northrop, 2002]. The SPL processes and a summary of the most relevant activities and practices are described below.

**Domain engineering** consists of creating a set of reusable assets for building systems in a specific problem domain. Domain Engineering determines the scope of the product-line and handles the commonalities and variabilities for all the products of the SPL. Once the domain has been well understood, an architecture is designed. The **product-line architecture** is considered flexible in the sense that it is capable of accommodating potential members of the SPL and, proactively, addressing the variability. Reusable assets are implemented and tested. The results of the entire process are known as the common platform or core asset base. The domain engineering process is summarized in the following set of sub-processes and activities [Clements & Northrop, 2002; Northrop, 2008; Pohl et al., 2005]:

1. Domain Identification and Scoping,
2. Domain Requirements Engineering,
3. Domain Analysis,
4. Domain Architecture Design,
5. Domain Realization or Core Assets Development,
6. Traceability Management,
7. Evolution and Maintenance.

**Application engineering**, also known as product derivation, consists of developing products through systematic reuse of core assets by deriving the variability points.
2. BACKGROUND

This means, core assets are extended from variability points with the selected variants for a specific product. Therefore, the results of this process are the software applications/products from a SPL. The application engineering process is summarized in the following set of sub-processes and activities [Clements & Northrop, 2002; O’Leary et al., 2009b; Pohl et al., 2005]:

1. Product Requirement Engineering
2. Product Configuration (application analysis and application architecture design),
3. Product Development (application realization),
4. Test, Integration and Deployment,
5. Traceability Management,

According to the scope of this thesis, the sub-processes Domain Analysis and Domain Architecture Design are especially relevant.

On one hand, domain analysis can be defined as the process by which information used in developing software systems within the domain is identified, captured, and organized with the purpose of making it reusable (to create assets) when building new products” [America et al., 2001]. Therefore, this process focuses on identifying the commonality and variability in requirements, and interdependencies of variability [Khumram & Gorschek, 2009]. Numerous methods for domain analysis can be found in literature: Feature-Oriented Domain Analysis (FODA), Domain Analysis and Design Process (DADP), and Stability-Oriented Domain Analysis (SODA), among others. A summary and comparison of these domain analysis methods can be found in [Mili et al., 2001]. One of the most widely used is FODA [Antkiewicz & Czarnecki, 2004; Kang et al., 1990, 2002], which our work is based on.

On the other hand, domain architecture design aims to produce the Product-Line Architecture (PLA), which can be defined as the concepts and structures to achieve variation in features of different products, while sharing as many parts as possible in the implementation [Jazayeri et al., 2000]. As a result, a PLA must serve the needs of many potential products of the same family or domain, and therefore must capture the commonality of these products. They must also deal with their differences to make the variation points explicit among the products constituting the PLA [Linden et al., 2007]. An interesting comparison of PLA design methods is presented in [Matinlasis, 2004].
2.3.3 Variability Management

Variability plays a key role in the success of SPLs, since it enables the derivation of a family of products, each of which is customized to satisfy different user needs. Its management is orthogonal to the all activities and phases mentioned in the previous section. The following activities are especially relevant to the research community:

1. eliciting, representing and documenting variability in software artefacts,
2. managing dependencies among variabilities,
3. implementing the variabilities, and
4. exploitation of the variabilities for building and evolving product applications from a product-line (configuration or product derivation).

The systematic literature review developed by Chen & Babar [2011] reveals that there has been an extensive research on supporting these four activities. In regard to the representation of variability, majority of approaches are based on feature modeling [Antkiewicz & Czarnecki 2004; Kang et al. 1990], architectural variability modeling [Adachi Barbosa et al. 2011; Bachmann & Bass 2001; Dashofy & Hoek 2002; van der Hoek et al. 1999; van Ommering et al. 2000; Weiler 2003] and UML-based modeling [Atkinson et al. 2001; Razavian & Khosravi 2008; Webber & Gomaa 2002]. The management of dependencies among variabilities has been addressed by Mohan & Ramesh [2007]; Sinnema et al. 2006; Taborda 2004c. Most of the research tends to concentrate on the third activity —i.e. the implementation-level of variability. A taxonomy of the different proposals for implementing variability can be found in Svahnberg et al. [2005]. Finally, in regard to the exploitation of the variabilities for building product applications, the work by O’Leary et al. [2009b] defines the key activities for the derivation process.

According to the main goal of this thesis, it is especially relevant to review the different approaches for (i) modeling variability, specifically at the domain-analysis and domain-architecture levels, and (ii) documenting the rationale behind architectural variability as well as managing the dependencies among architectural variabilities.
2. BACKGROUND

2.3.3.1 Modeling Variability

There are two main approaches to specify variability in SPLs [Metzger & Pohl, 2007]: (i) separating the variability in a separate dimension from the development artifacts, and (ii) representing variability throughout the development artifacts including requirements specification, architecture description or implementation. After reviewing both approaches, pros and cons of these approaches are analyzed.

(i) Separating the variability in another dimension from the development artifacts.

These approaches propose a separate model for defining variability, also known as dedicated variability model (see a, Figure 2.1). The works by [Bachmann et al., 2004; Loughran et al., 2008; Pohl et al., 2005] are based on the assumption that variability should be independently specified. Chronologically, [Bachmann et al., 2004] are the first authors to propose a uniform representation of variability instead of explicit representation of variability throughout the different development artifacts, which is known as the Variation Model. The Variation Model captures variability points, their rationale, variants, and their dependencies. Each variant refers/traces to one or more assets, therefore this model acts as a dedicated view which interacts with all other system’s views. These include requirements, architecture or implementation, according to the nature of the assets which its variants refer to. [Pohl et al., 2005] also propose to define variability in a separate model which conforms to the Orthogonal Variability Metamodel (OVM), and consequently trace this model to all development artifacts. The variability modeling language proposed by [Loughran et al., 2008] is clearly influenced by Pohl and Bachmann’s work. Loughran et al. take Pohl and Bachmann’s variability documentation approach and add primitives to perform composition of architectural variabilities, i.e. they add the primitives to reference the variation points and variants of a dedicated variability model to architectural elements. Finally, the Base-Variation-Resolution approach [Bayer et al. 2006] describes a formalized way of specifying the relationship between a separate variability model, which is specified by the common variability language [Haugen et al., 2008], and other models.

(ii) Integrating variability in the development artifacts.

These approaches define variability in the artifacts that are involved in the SPL development (see b, Figure 2.1). Some of them are specific to domain-analysis
models, such as Feature-Oriented Domain Analysis (FODA) [Antkiewicz & Czarnecki, 2004; Kang et al., 1990, 2002] or product-line architecture models [Adachi Barbosa et al., 2011; Bachmann & Bass, 2001; van der Hoek et al., 1999; van Ommering et al., 2000] (see Section 2.3.3.2). These approaches may rely on others that define the relationships of variability between the different models [Czarnecki, 2005; Zschaler et al., 2010].

Figure 2.1: Approaches for representing variability - a) Separating the variability in another dimension from the development artifacts. b) Integrating variability in the development artifacts

Dedicated variability models provide several advantages: they are smaller and less complex, which improves communication of variability among different stakeholders [Metzger & Pohl, 2007]. Additionally, dedicated variability models help to understand and model the dependencies between variabilities [Bachmann et al., 2004]. However, the initial advantages of separating variability from existing artifacts imply several issues:

First, the variability model on its own is not able to represent the full meaning of variability in SPLE [Linden et al., 2007]. The traditional views of requirements, design, etc. are necessary, as well as the relationship between these views and the variability. Separating variability tends to result in an overhead of traceability links in order to relate the dedicated variability model to the other artifacts created during the development process, such as features, architecture or implementation models (see a, Figure 2.1). This makes it difficult to manage traceability links and maintain them. In fact, managing the traces to and from the artifacts to the variability defined in the dedicated model is an important challenge, which was already identified by [Bachmann et al., 2004], but it has not been completely dealt with yet.
Second, the high-level of abstraction in which variability is specified in a dedicated model, such as that proposed by Bachmann et al. [2004] or Pohl et al. [2005], is not easily and explicitly mapped to the architecture-level and much less to the implementation level. Variability, as it is specified in these dedicated models, is closer to the problem space than to the solution space. The distance between the problem space and the solution space, motivated by the exponential complexity that arises when the product-line architecture’s artifacts are built, makes it difficult to relate dedicated variability models to architectural models or source code.

Third, dedicated variability models are not enough to support the development and evolution of complex SPLs if variability in software architectures is not specified [Loughran et al. 2008]. Architecturally-relevant variabilities cannot be solely expressed using OVM models or conventional features models [Czarnecki & Eisenecker 2000]. Variability support must be an integral part of the product-line architecture [Linden et al. 2007].

These three issues can be addressed by using integrated variability models. Hence, the first issue is addressed as variability is specified within the different development artifacts, either requirements, architecture or others, and traces to/from a dedicated variability model are not necessary. The second issue is addressed due to the fact that variability is specified in different levels of abstractions, from requirements to design or code. Finally, the third issue is addressed because architecturally-relevant variability can be specified within product-line architectures.

Once the pros and cons of dedicated and integrated variability models have been described, the approaches that integrate variability in the development artifacts will be focused on. The following two subsections present the most relevant approaches for modeling variability at the domain analysis and architecture levels.

### 2.3.3.2 Feature Modeling

Feature-Oriented Domain Analysis (FODA) introduces the feature modeling technique for capturing commonality and variability of SPL in terms of features. Feature modeling is graphically described through the feature diagram notation, which specifies all products of a family (all possible configurations of the SPL) through a hierarchical tree-like structure. That means, the feature model structures the requirements into a tree which shows common and variable features for all products of a family. Some extensions
were defined to compensate the ambiguity and lack expressiveness in feature models [Schobbens et al. 2007]. Since there is no universally accepted definition of feature modeling, the extended feature metamodel definition proposed by [Czarnecki et al., 2005] is used in this thesis (see Figure 2.2). It includes the following concepts:

- *RootFeature* metaclass is used to modularize the model in a tree-like structure where exactly one root is marked as the main one in the model.
- *SolitaryFeature* and *GroupedFeature* metaclasses represent ungrouped and grouped characteristics of a software system.
- *FeatureGroup* metaclass links *GroupedFeatures* a set of *GroupedFeatures*.
- *Feature* metaclass is the generic definition of feature. It is *hasFeatures* zero or more *SolitaryFeatures* and is *hasGroups* zero or more *FeatureGroups*. The metaclasses *SolitaryFeature*, *GroupedFeature* and *RootFeature* inherit from *Feature* metaclass.

Following the Czarnecki’s approach, the metaclasses *FeatureGroup* and *SolitaryFeature* have the attributes *groupcardinality* and *featurecardinality*, respectively. These attributes allow the specification of relationships AND, X-OR, OR, mandatory and optional (see Figure 2.2). The attribute *groupCardinality* describes how many *GroupedFeatures* can actually be used, whereas the attribute *featureCardinality* describes how many times it can be used to compose the parent feature.
2. BACKGROUND

2.3.3.3 Product-Line Architecture Modeling

Architectural commonality represents design decisions that are bound during the architecting process, whereas architectural variability represents optional, alternative and multiple design decisions that are not bound during the architecting process. This variability has to be explicitly specified at the architecture-level to model the PLA, and then instantiated to derive the product architectures (PA).

Since PLAs define the common and variable structure for all products of a product-line, they consist of a set of reusable building blocks that can be configured to build the products that make up the product-line. From the structural viewpoint, these building blocks are components and connectors, which should be designed to be configured to build the products—that is, they should provide variability. In fact, variability should be one of the most important architecture drivers.

An interesting comparison of PLA design methods is presented in [Matinlassi, 2004]. A literature review of the most important approaches for representing PLAs has allowed us to classify the different approaches to represent architectural variability as: annotative, declarative, compositional, and transformational approaches.

- Annotative approaches consider one model representing variability and the relationship with the architectural model, in the same way that dedicated variability models do. This approach is proposed by the *variability modeling language* [Loughran et al., 2008] and the *variability guide* [Clements et al., 2010].

- Declarative approaches define specific concepts to specify variability points within ADLs and/or architectural models. ADLs supporting the specification of PLAS typically distinguish core elements from variability points. Hence, variability points are architectural elements themselves. xADL [Dashofy & Hoek, 2002], Unicon [Shaw et al., 1995], and PL-AspectualACME [Adachi Barbosa et al., 2011] offer extensions for describing PLAs by means of (i) option and variant extensions, (ii) variant implementations of components, and (iii) aspectual connectors, respectively. Other approaches, although they are not ADLs in the strictest sense of this term, also offer modeling primitives to describe architectural variability. Therefore, several of these approaches propose optional interfaces such as the Koala Component Model [van Ommering et al., 2000]; variant components, variant connectors and multi-versioning connectors [van der Hoek et al., 1999, 2001]; elastic components [Kakarontzas et al., 2007]; or port variability [Mann, 2009].
2.3 Software Product Lines

- Compositional approaches reuse well-known concepts to specify variability within architectural models and/or ADLs. These concepts are based on: (i) the compositionality mechanism of software architecture, such as the use of representations (eg. Acme [Garlan et al., 2000]) or subtyping (eg. C2 [Medvidovic et al., 1996]); (ii) the UML compositionality and inheritance mechanisms, such as the approaches of [Razavian & Khosravi, 2008; Webber & Gomaa, 2002]; or (iii) the invasive software composition principles [Assmann, 2003], such as the AOP extension for SPL by [Lee et al., 2006], aspect-oriented model [Noda & Kishi, 2008], elastic components [Kakarontzas et al., 2007], or PL-Aspectual ACME [Adachi Barbosa et al., 2011].

- Transformational approaches use model transformation to capture variability. This approach is followed by the common variability language [Haugen et al., 2008] that represents the variability of a base model using rules; these rules describe how modeling elements of the base model have to be substituted to obtain a product model. [Hendrickson & van der Hoek, 2007] represent architectural variability through change sets containing additions and removals of components and connections. Finally, [Haber et al., 2011] provide modification operations to capture variability.

As it is proposed by annotative approaches, the main disadvantage of separate models for modeling variability is the resulting overhead of tracing the separate variability model to the traditional artifacts, such as requirements, architecture, design, etc.

Although transformational approaches are very promising, they are characterized by the complexity in defining the transformations or rules to model variability. In addition, the problem of following the life of variability through the different software artifacts created during the software lifecycle has still not been solved.

As it is proposed by declarative and compositional approaches, variability has to be explicitly represented at the architecture level. They consider essential to integrate variability modeling concepts into architectural models for successfully developing SPLs. These concepts specify places in the PLA where differences exist among specific product architectures. Furthermore, unlike annotative and transformational approaches, compositional and declarative approaches enable the capability to trace the life of variability.

Most of these mechanisms for describing architectural variability specify external variation in the architecture. This provides flexibility to enable the development of
2. BACKGROUND

customized applications from a SPL by modifying the configuration of the architecture. Specifically these mechanisms specify variability by:

- Adding or removing components and connections through the mechanisms defined by declarative approaches [Dashofy & Hoek, 2002; van der Hoek et al., 1999; van Ommering et al., 2000].

- Modifying the configuration of composite components, also known as subsystems, through the use of compositionality mechanisms of software architecture, such as the use of subtyping of components and connectors [Medvidovic et al., 1996] or the use of representations [Adachi Barbosa et al., 2011; Garlan et al., 2000].

However, it is also necessary to specify variations inside components (internal variation), i.e. variations of simple components —or non-composite components.

The work of [Bachmann & Bass, 2001] is a reference for variability management in software architectures as it provides a complete support for variability, both internal and external. They laid the basis for addressing internal variability by means of components that realize all possible variations. Their work proposes different techniques to realize variations of components, such as generators, configuration management systems, compiler switches or external configuration files. However, in recent years, the approaches based on the aspect-oriented software architectures [Pérez et al., 2006] are emerging to apply aspect-oriented to the specification of the internal variation of components [Adachi Barbosa et al., 2011].

2.3.3.4 Documenting Variability Design Rationale

The explicit capture and documentation of architecture knowledge improves understanding and rationalization of architectural decisions made during the design of an architecture solution and may allow one to reason about change, and thus, may help to maintain and evolve the architecture without compromising its integrity. These advantages are still higher in PLAs.

The documentation of design rationale has a high strategic value and deep impact on the success of PLAs design, and consequently on the SPL development [Knodel & Muthing, 2006]. However, documenting the knowledge of PLAs is more complex
than in the case of single-product architectures. The difficulty of documenting PLAs lies in the fact that PLAs require documenting architecture that realizes variability. This may involve new kinds of design decisions to recognize the role of variability and new critical dependencies in order to manage the variability found in the multiple products supported by the product-line. Hence, design decisions related to a variability point may affect several others. Examples of this are quality attributes which may have variability points and thus, affect many different products. There could even be conflicting quality attributes in different products of the same family.

Therefore, documenting the knowledge of PLA requires documenting the design decisions, dependencies and rationale behind the commonality, as well as the design decisions, dependencies and rationale behind the variability, also known as *variability design rationale*. In fact, both documentation of design decisions associated with variations and the capability to trace the life of these variations are key to effectively develop and evolve software product-lines [Mohan & Ramesh, 2002, 2007]. It is also recognized that the design and exploitation of PLAs demand even more architectural knowledge than that required of general reuse-based software engineering [Bosch, 1999]. This is due to the fact that PLAs not realize the design of a single product, PLAs realize the design of a family of products. Namely, various stakeholders (of various products of a SPL) must agree on all design decisions related to the definition of the PLA.

In spite of the necessity of capturing and exploiting variability design rationale, the documentation of PLAs has received little attention so far. Only a few approaches partially address variability design rationale. [Clements et al., 2010] propose an annotative approach for documenting variability design rationale. [Capilla & Ali Babar, 2008] propose a unified data model to explicitly support the relationships between design decisions and variability models of SPLs. Finally, [Galvão et al., 2010] propose a grammar to specify design rationale, including variability design rationale.

Finally, other approaches focus on the problem of the dependencies between variants, which has to be considered when binding the overall variability to derive products from the family. This means, the possible products are restricted, due to dependencies that exist between variants, and the constraints that are imposed upon these dependencies. Consequently, being aware of the dependencies between variants is essential, otherwise incorrect products will result. A representative example of research area is the COVAMOF framework [Sinnema et al., 2006]. However, dependencies between variants are not only important when configuring products, but also when effectively aiding evolution of software product lines. Dependencies between design decision, as
2. BACKGROUND

well as architectural elements, including variability points and variants, are essential to analyze which parts of the PLA will be impacted by a proposed change.

2.3.4 Evolution of Product-Line Architecture

SPLE must coordinate the design, development, and also the evolution of a set of related products. SPLE implies a high cost in developing a SPL platform which will only be profitable if this platform evolves over several years. In fact, the typical pattern is to start with a small set of products (often just one), and after the business starts to generate profits or looks profitable in the future, the decision is made to develop a PLA and introduce new products [Riebisch & Philippow 2001; Riva & Del Rosso 2003]. Even though a PLA was designed with a range of planned products in mind, it will inevitably deal with unforeseen requirements, improvements or even new products. Therefore, a SPL may evolve to enhance the PLA, impacting all the derived products, or to add more products as new members in the product family [Cho et al., 2011].

The challenges of software evolution are even more problematic in SPLE, since the PLA simultaneously defines the architectural structure for a set of closely-related systems or products. Therefore, making changes on the PLA requires the consideration of multiple variability points, variants, dependencies, decisions and constraints from different stakeholders and users of the product line family [Cho et al., 2011]. Certainly, one of the major issues regarding SPL evolution is variability, as it involves more critical dependencies than single-products, or even possible conflicting requirements [Svahnberg & Bosch 1999]. The variability of the products must be considered when evolving the architecture. It must be carefully verified if a change in architecturally-relevant variabilities can break the integrity of the PLA, i.e. if a change in architecturally significant requirements for a product can violate early design decisions causing the erosion and degradation of the PLA.

The importance and complexity of evolution of software product lines has been recognized by [Bosch & Ran 2000; Garg et al. 2003; Lago & van Vliet 2004; Riva & Del Rosso 2003; Svahnberg & Bosch 1999; Taborda 2004b]. A small number of approaches address several faces of the evolution problem in SPLE. The approach by [Taborda 2004e] proposes a release matrix for capturing the inter-dependencies between multiple products and components in order to plan and track the evolution of the system architecture over time. The work by [Garg et al. 2003] presents an environment for managing the evolution of PLAs in joining the Ménage tool and xADL 2.0 [Dashofy &
2.3 Software Product Lines

Hoek, 2002; Dashofy et al., 2005. It offers support for (i) specifying PLAs as a set of core architectural elements which are augmented with variability points, (ii) versioning or architectural elements, and (iii) product configuration. But neither of them address the other part of the problem: How can we manage the change in such a way that we can understand which parts of the architecture are impacted by the change while maintaining the integrity of the architecture?

A categorization of product-line architecture evolution may help to understand the change. This categorization has been defined by Bosch & Ran, 2000; Svahnberg & Bosch, 1999. They define eight categories of evolution which range from improving existing functionality or quality attributes to adding a new product to the SPL. Therefore, a change may affect (i) the entire PLA, i.e. common architectural elements to all products from the product-line; (ii) specific product-architectures (PA), i.e. optional, alternative or multiple variants; or (iii) both of them. At the realization level, these changes fit with the architectural change types defined in Section 2.2.4.1, specifically with structural, interface and rationale evolution.

However, despite the fact that software evolution has clearly advanced these last years in traditional software development, evolution is still challenging in SPL. The missing traceability among software artifacts becomes a major challenge for evolving SPL. The capability of tracing variable features to their realization in the PLA, as well as the identification of the artifacts that are impacted by changes in variable features is essential to completely manage variability. Change impact analysis may provide criteria upon which the decisions to evolve or not evolve an architecture could be made based on economic viability of product-line evolution Bosch & Ran, 2000; Schmid & Verlage, 2002a. Change impact analysis may even provide criteria that may help prioritize changes to the architecture.

Change impact analysis may also help to cope with the architectural dependencies between variants by recovering past design decisions. In fact, variability points are not usually independent of each other Thiel & Hein, 2002 but they maintain complex dependencies on core architectural elements and/or other variabilities. Therefore, it is important to take into account, with change impact analysis in SPL, that there might be dependencies between variants, also known as variability dependencies Bachmann et al., 2004. These dependencies have significant implications in the evolution of PLAs. Hence, a given variability point or variant may depend on other variability points or variants. We consider that there are different types of dependencies, which are specific to the artifact in which dependencies are analyzed, as well as the viewpoint from which
2. BACKGROUND

the artifact is represented. Hence, the software research community has described the following dependencies: feature dependencies [Lee & Kang, 2004], architectural dependencies [Stafford & Wolf, 1998], design-decisions dependencies [Babar & Gorton, 2007; Zimmermann et al., 2009], traceability dependencies [Gotel & Finkelstein, 1994], code dependencies [Law & Rothermel, 2003; Podgurski & Clarke, 1990], etc. In accordance to the main goal of this thesis, the following are especially relevant: (i) traceability dependencies between features, architectural design decisions and architecture elements, and (ii) variability dependencies at the architecture-level which may be modeled through design-decision dependencies.

2.3.5 Conclusions

Managing variability is a key task when developing SPLs. Variability is spread across several different development artifacts (requirements, architecture, design, code, etc.) making it necessary to capture, represent and model this variability. When selecting an approach to model variability, the pros and cons described in Section 2.3.3.1 must be taken into account. Hence, the sole use of dedicated variability models is not sufficient as their composition mechanisms are not tailored to express architecture-level variabilities. PLA embodies the commonality, variability and earliest design decisions for all products of a product-line, providing a framework where reusable components can be developed. As a result, mechanisms to explicitly specify the commonalities and variabilities of SPLs at the architecture-level are required.

The documentation of design rationale also has a high strategic value and deep impact on the success of the development of SPLs. This is based on the need to document variability design rationale, which may involve new kinds of design decisions in order to recognize the role of variability, as well as more critical dependencies, than in the case of documenting the architecture of single-products. Documenting the design decisions associated with architectural variability, the dependencies between these design decisions, the rationale behind the use of specific mechanisms to realize this variability, and the capability to trace the life of variability, enables more effective change impact analysis in SPLE. Therefore, reasoning with models specified through these metamodels may help making the right decision when adopting and evolving a product line.
Chapter 3

Agile Product Line Engineering

This thesis is framed into the Agile Product Line Engineering (APLE) [Cooper & Franch, 2006] paradigm which pursues to combine reuse and customization as practiced in Software Product Line Engineering (SPLE) with concepts such as iterative development and embracement to change as described by Agile Software Development (ASD).

This chapter reports the results and key findings of a systematic literature review of experiences and practices of APLE. It has been conducted in the scope of this thesis, in order to bring the published work on this emerging paradigm together. A review of the current state-of-the-art research in APLE may provide understanding when analyzing the reasons to move towards APLE. It also helps identify what barriers have been dealt with and what challenges have to be addressed in the near future in order to apply APLE to the software industry. In fact, this review provides the problem statement which is resolved in this thesis.

The remainder of this chapter is structured as follows: Section 3.1 provides an overview to APLE; Section 3.2 describes the method used for the literature review of the state-of-the-art research in APLE and reports the contributions of the collected studies; Section 3.3 summarizes key findings as well as implications for practitioners. Finally, conclusions are presented in Section 3.4.
3. AGILE PRODUCT LINE ENGINEERING

3.1 Introduction

Software product line engineering (SPLE) [Clements & Northrop, 2002; Pohl et al., 2005] exploits the commonality and variability found in products from the same family by an upfront long-term design of a common set of core-assets/platform (Domain Engineering, DE). The core-assets are then assembled into customer-specific products by deriving the existing variability found in the products (Application Engineering, AE). This means that SPLE takes advantage of common features among the products of a family through the systematic reuse of the core-assets and the effective management of variabilities across the products. Significant improvements have been made using SPLE. These include reducing cost and time-to-market as well as providing flexibility to respond to planned changes [Schmid & Verlage, 2002b] in contexts where anticipated changes can be predicted with certain accuracy. However, when large, complex Software Product Lines (SPL) projects have to deal with changing market conditions, i.e. unplanned changes, alternatives to supplement SPLE are required [Känsäla, 2004]. Agile software development (ASD) may be one of these alternatives, since agile processes harness change for the customer’s competitive advantage [Shore & Warden, 2007]. However, scalability is still a challenge in agile projects, therefore the reusability and variability management among the products from the same family are not easily scaled up [Ghanam et al., 2010a]. As a consequence, a sensible approach resulting from the integration of SPLE and ASD would be to have them complement each other. Some advocates of this approach have coined the term agile product line engineering (APLE) [Cooper & Franch, 2006]; others coined the term agile software product lines (ASPL) [Känsäla, 2004] to refer to the same approach.

Therefore, APLE advocates the integration of SPLE and ASD with the goal of addressing what both approaches lack when they are applied individually to software development. Both of them have proved to have significant benefits in software development. Although they pursue common promises (faster time-to-market, better quality and lower cost), many of their foundations are completely different, or even opposite.

Although it is difficult to provide a date, it is understood that the consideration of SPLE and ASD as a joint approach was recognized in 2002. In conjunction with the 2002 International Conference on Software Reuse, a new workshop on Software Reuse and Agile Approaches was offered [Yoder, 2002]. This workshop focused on combining DE and Agile methods. Later on, in conjunction with the 2006 SPL Conference, a new workshop called ‘Agile Product Line Engineering’ (APLE’06 [Cooper & Franch, 2006]) was arranged. In this workshop, it was acknowledged that both Agile and SPLE approaches share several common goals and could work together. According
3.1 Introduction

to the panel ‘Agile and Product Line Engineering’ (APLE’08 [McGregor 2008a]), organized by the 2008 SPL Conference, the applicability of agile methods to AE looked feasible, but it was not clear how agile methods could be applied to DE. Finally, two workshops on APLE [Ghanam et al. 2009a, 2010b] in conjunction with the eXtreme Programming Conference, were held in 2009 and 2010. In these workshops, issues such as the traditional way of creating and managing SPLs were analyzed. They focused on how the upfront investments in SPL platforms might be profitable in changing market conditions.

Although practitioners and the academic community have tackled the integration of SPLE and ASD from various points of view, no definitive approach has been achieved so far. Organizations still have to face numerous barriers to put APLE into practice. Until now no systematic literature review has been undertaken to bring together the published work on APLE. Systematic literature reviews (SLR) identify, evaluate and interpret all available relevant research on a specific research question or topic area through the use of a rigorous and auditable methodology [Kitchenham 2004]. Systematic literature reviews are important for different reasons [Budgen & Brereton 2006]: (i) to summarize existing evidence concerning a practice or technology, (ii) to identify where there are gaps in current research, (iii) to help position new research activities; and (vi) to examine how far a given hypothesis is supported or contradicted by the available empirical evidence. As a result, a systematic literature review on APLE is necessary to build this common understanding and to identify existing challenges in the APLE implementation. In addition, it could be useful for practitioners who want to stay up-to-date with the current research and which problems have been solved, as well as for researchers who want to identify what open gaps still exist.

Keeping these purposes in mind, the systematic literature review has been conducted following the guidelines proposed by Kitchenham [Kitchenham 2004]. The search strategy identified over 370 unduplicated scientific papers, of which only 39 were directly related to APLE. The key findings uncover important challenges about how to integrate the ‘domain-then-application’ SPLE model [Ghanam et al. 2008] with an agile iterative approach.
3. AGILE PRODUCT LINE ENGINEERING

3.2 Systematic Literature Review: the method

A review on the state-of-the-art research in APLE has been conducted following the guidelines put forward by Kitchenham [Kitchenham, 2004], who proposes three main phases for doing a systematic review process: (i) planning the review, which aims to develop a review protocol (see Section 3.2.1); (ii) conducting the review, which executes the planned protocol in the previous phase (see Section 3.2.2); and (iii) reporting the review, which is responsible for presenting the review steps to the community (see Section 3.2.3). The execution of the overall process involves iteration, feedback, and refinement of the defined process.

3.2.1 First Phase: Planning the Review

The phase planning the review consists of developing a review protocol. The review protocol defines the methods to undertake a specific systematic review, reducing the possibility that this review can be driven by research expectations. Protocol development must specify (i) the review objective and research questions; (ii) the search strategy; (iii) the explicit inclusion and exclusion criteria; (iv) the criteria to evaluate each study; and (v) the data extraction strategy and the strategy for synthesizing extracted data [Kitchenham, 2004]. All of these steps are described in the following subsections.

Review objective and research questions

The review objective is to identify experiences and practices in APLE in order to identify the current APLE approaches, i.e. how SPLE and ASD can be integrated, what barriers have been dealt with, and what challenges have to be addressed in the near future to apply APLE to the software industry. In detail, the research questions that were identified to achieve the outlined objective are the following:

- RQ1: What are the reasons for combining SPLE and ASD? When is it advantageous to put APLE into practice?
- RQ2: How are the principles of SPLE and ASD related to each other?
3.2 Systematic Literature Review: the method

- RQ3: How are SPLE and ASD positioned in respect to business strategic objectives?
- RQ4: Which current approaches combine SPLE and ASD satisfying AE activities?
- RQ5: What are the challenges and gaps in current approaches that combine SPLE and ASD during AE activities?
- RQ6: Which current approaches combine SPLE and ASD satisfying DE activities?
- RQ7: What are the challenges and gaps in current approaches that combine SPLE and ASD during DE activities?
- RQ8: Which current approaches combine SPLE and ASD satisfying both DE and AE activities through agile principles?
- RQ9: Are there successful industrial experiences putting APLE into practice?

Search strategy

A formal search strategy is required in order to find the entire population of scientific papers which may be relevant when answering the identified research questions. The formal definition of this search strategy allows us to make a replicable and open review of external assessments. The search strategy consists of defining the search space: electronic databases and conference proceedings that are considered key spaces for the review objective (see Table 3.1). These searches retrieve a list of scientific papers called primary studies. Our search strategy also defines two more steps: secondary searches based on references found in primary studies, and tertiary searches in DBLP based on the authors found in primary studies in order to look for previous or subsequent studies by these authors. Additionally, some authors were directly e-mailed to find out whether they had any published material.

Once the searching space has been defined, it is necessary to define the search terms to be introduced into the search inquiry forms of electronic databases. The search terms have been divided into two criteria: (i) key words related to ASD and specific agile methods such as Scrum, eXtreme Programming (XP), Feature Driven Development (FDD), or Lean; and (ii) key words related to SPLE such as product line(s), product family (-ies), Domain Analysis, Domain Engineering, Core Assets, Product Derivation, or Application Engineering. Acronyms for each of these terms were also added in the search inquiry. Next, some examples of searches in EI Compendex and ISI Web of Knowledge are shown:
Table 3.1: SLR - Search resources.

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eletronic databases</td>
<td>ACM Digital library</td>
</tr>
<tr>
<td></td>
<td>IEEE Xplore</td>
</tr>
<tr>
<td></td>
<td>SpringerLink</td>
</tr>
<tr>
<td></td>
<td>EI Compendex</td>
</tr>
<tr>
<td></td>
<td>Inspec</td>
</tr>
<tr>
<td></td>
<td>ISI Web of Knowledge</td>
</tr>
<tr>
<td></td>
<td>ScienceDirect</td>
</tr>
<tr>
<td>Conference proceedings &amp; hand searches</td>
<td>SPLC (Software Product Line Conference)</td>
</tr>
<tr>
<td></td>
<td>XP (Agile Processes and eXtreme Programming in SE)</td>
</tr>
<tr>
<td></td>
<td>Agile Conference</td>
</tr>
<tr>
<td></td>
<td>APLE (Workshop on Agile Software Product Line Eng.)</td>
</tr>
</tbody>
</table>

**Inclusion and exclusion criteria**

The review protocol also specifies inclusion criteria (IC) and exclusion criteria (EC), which determine whether each potential study should be considered or not for this systematic review. The list of IC and EC for this APLE systematic review is shown below.

- **Inclusion Criteria (IC)**
  - **Type of studies:** Scientific material written in English, according to the search terms defined in the previous section. Scientific material is a general term that includes papers, short papers, experience reports, summaries of workshops, panels, and poster sessions, subjected to the normal scientific peer review process, normally double blind reviewed by 2-3 reviewers.
3.2 Systematic Literature Review: the method

- **Time**: The scientific material published until June 2010.
- **Exclusion Criteria (EC)**
  - **Agility as adjective**: Those studies that use the term agility as an adjective, but they do not refer to agile methodologies. These studies use the term ‘agile’ or ‘agility’ to mean flexibility but not ASD.
  - **Poor arguments**: Those studies that are based on general opinion and/or poor arguments.
  - **Reductio ad absurdum**: Those studies that do not fulfill the IC.

**Quality assessment**

Kitchenham’s guidelines [Kitchenham, 2004] suggest performing a quality assessment of each included study; it complements the IC and EC as defined above. However, there is no universal agreed definition of “quality”. The Critical Appraisal Skill Programme (CASP)\(^1\) defines criteria for assessing the quality of qualitative research. This systematic review has used the quality criteria defined for CASP and those proposed by Dybå et al. [Dyba & Dingsoyr, 2008]. The criteria cover three main issues: rigor, credibility, and relevance. The quality assessment form defined by [Dyba & Dingsoyr, 2008] is briefly summarized in Table 3.2.

<table>
<thead>
<tr>
<th>Table 3.2: SLR - Quality criteria. Adapted from [Dyba &amp; Dingsoyr, 2008].</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Is there a clear statement of the aims and objectives of the research? (YES/NO)</td>
</tr>
<tr>
<td>2. Is there an adequate description of the context in which the research was carried out? (YES/NO)</td>
</tr>
<tr>
<td>3. Is there an adequate description of the proposed contribution, method, or approach? (YES/NO)</td>
</tr>
<tr>
<td>4. Is there a clear statement of findings? (YES/NO)</td>
</tr>
<tr>
<td>5. Is the evidence obtained from experimental studies or observational studies? (YES/NO)</td>
</tr>
<tr>
<td>6. Is the study of value for research or practice? (YES/NO)</td>
</tr>
</tbody>
</table>

**Data extraction and data synthesis strategies**

The extraction process consists of identifying the required data to answer the research questions. In response, we created a set of forms to store verbatim key concepts

\(^1\)http://www.phru.nhs.uk/Pages/PHD/CASP.htm
3. AGILE PRODUCT LINE ENGINEERING

regarding goals, findings, and conclusions of each included study. The synthesis process consists of organizing the key concepts to enable comparisons between studies and the reciprocal translation of findings into higher-order interpretations.

### 3.2.2 Second Phase: Conducting the review phase

The phase conducting the review includes a sequence of steps: (i) search for primary studies; (ii) selection of primary studies applying inclusion/exclusion criteria; (iii) study quality assessment; (iv) data extraction; and (v) data synthesis.

#### Search for studies

Following the guidelines described in Section 3.2.1, a search for primary studies was performed. It retrieved over 536 results, but these results include a certain degree of redundancy between the search engines. Secondary and tertiary searches retrieved 32 citations by scanning reference lists, scanning web pages of prominent authors, and contacting them. This adds up to 568 results. Table 3.3 shows the number of studies that were retrieved from each search engine, including duplicated studies. We have rejected 198 duplicated results; hence only 370 unduplicated studies were included to be evaluated in the next step.

<table>
<thead>
<tr>
<th>Database</th>
<th>Retrieved</th>
<th>Included</th>
<th>Excluded</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM Digital Library</td>
<td>48</td>
<td>4 (8.33%)</td>
<td>44 (91.67%)</td>
</tr>
<tr>
<td>EI Compendex</td>
<td>23</td>
<td>8 (34.78%)</td>
<td>15 (65.22%)</td>
</tr>
<tr>
<td>IEEE Xplore</td>
<td>14</td>
<td>5 (35.71%)</td>
<td>9 (64.29%)</td>
</tr>
<tr>
<td>Inspec</td>
<td>35</td>
<td>9 (25.71%)</td>
<td>26 (74.29%)</td>
</tr>
<tr>
<td>ISI Web of Knowledge</td>
<td>15</td>
<td>6 (40%)</td>
<td>9 (60%)</td>
</tr>
<tr>
<td>ScienceDirect</td>
<td>312</td>
<td>8 (2.56%)</td>
<td>304 (97.44%)</td>
</tr>
<tr>
<td>SpringerLink</td>
<td>89</td>
<td>7 (7.87%)</td>
<td>82 (92.13%)</td>
</tr>
<tr>
<td>Secondary and Tertiary searches</td>
<td>32</td>
<td>5 (15.63%)</td>
<td>27 (84.38%)</td>
</tr>
<tr>
<td>Total</td>
<td>568</td>
<td>52 (9.15%)</td>
<td>516 (90.85%)</td>
</tr>
</tbody>
</table>

Table 3.3: SLR - Results of the systematic search for primary studies.
3.2 Systematic Literature Review: the method

Study selection

After retrieving all of the unduplicated scientific papers relevant to the research questions, we selected the publications relevant to the review objective, according to the inclusion and exclusion criteria defined in Section 3.2.1. After evaluating the title and abstract, 168 studies were rejected. After a quick read-through of the bodies of the papers, 150 studies were rejected. Therefore, a total of 516 studies were rejected (see Table 3.3). After this process, a total of 52 relevant studies (see Table 3.3) were entered into a spreadsheet, along with their bibTex source. Electronic versions of publications were also stored in a filesystem directory for easy access during the systematic review.

Study quality assessment, data extraction and data synthesis

Each selected study was assessed on the basis of the quality criteria defined in Section 3.2.1. 75% of studies passed the defined quality criteria (39 scientific papers). The data extraction consisted of coping verbatim goals, findings, and conclusions reported by the studies into a commercial tool for organizing and tracking information. Finally, results were synthesized in tabular forms that show the APLE roadmap. They allowed us to compare studies and to identify current challenges and gaps (see Section 3.2.3).

3.2.3 Third Phase: Reporting the review

The systematic review retrieved 39 scientific papers which have been identified from P1 to P39 in chronological order of publication (see Table 3.4). They have been obtained from different publication channels as Table 3.5 shows. The research questions defined in Section 3.2.1 have been addressed through the 39 selected papers in this systematic review.

RQ1: What are the reasons for combining SPLE and ASD? When is it advantageous to put APLE into practice?

Several studies have concluded that there are sufficient reasons to move towards a combination of SPLE and ASD. The three main goals that lead to combination of SPLE and ASD are:
3. AGILE PRODUCT LINE ENGINEERING

Table 3.4: SLR - Selected papers: list of the correspondences between the selected papers and the references in chronological order.

<table>
<thead>
<tr>
<th>Id</th>
<th>Reference</th>
<th>Id</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Kane, 2003</td>
<td>P11</td>
<td>Paige et al., 2006</td>
</tr>
<tr>
<td>P2</td>
<td>Kansala, 2004</td>
<td>P12</td>
<td>Kircher et al., 2006</td>
</tr>
<tr>
<td>P3</td>
<td>Taborda, 2004a</td>
<td>P13</td>
<td>Feng et al., 2007</td>
</tr>
<tr>
<td>P4</td>
<td>Cooper &amp; Franch, 2006</td>
<td>P14</td>
<td>Hanssen, 2007</td>
</tr>
<tr>
<td>P5</td>
<td>Carbon et al., 2006</td>
<td>P15</td>
<td>Noor et al., 2008a</td>
</tr>
<tr>
<td>P6</td>
<td>Kurmann, 2006</td>
<td>P16</td>
<td>Noor et al., 2008b</td>
</tr>
<tr>
<td>P7</td>
<td>Noor et al., 2006</td>
<td>P17</td>
<td>Trinidad et al., 2008</td>
</tr>
<tr>
<td>P8</td>
<td>Tian &amp; Cooper, 2006</td>
<td>P18</td>
<td>Hanssen &amp; Figri, 2008</td>
</tr>
<tr>
<td>P9</td>
<td>Navarrete et al., 2006</td>
<td>P19</td>
<td>Karam et al., 2008</td>
</tr>
<tr>
<td>P10</td>
<td>Trinidad et al., 2006</td>
<td>P20</td>
<td>McGregor, 2008a</td>
</tr>
<tr>
<td>P21</td>
<td>Raatikainen et al., 2008</td>
<td>P31</td>
<td>McGregor, 2008b</td>
</tr>
<tr>
<td>P22</td>
<td>Carbon et al., 2008</td>
<td>P32</td>
<td>McGregor, 2008c</td>
</tr>
<tr>
<td>P23</td>
<td>Kakarontzas et al., 2008</td>
<td>P33</td>
<td>Ghanam et al., 2009b</td>
</tr>
<tr>
<td>P24</td>
<td>O’Leary et al., 2007a</td>
<td>P34</td>
<td>Martinez et al., 2009</td>
</tr>
<tr>
<td>P25</td>
<td>O’Leary et al., 2007b</td>
<td>P35</td>
<td>Codenie et al., 2009</td>
</tr>
<tr>
<td>P26</td>
<td>O’Leary et al., 2008</td>
<td>P36</td>
<td>Ali Babar et al., 2009b</td>
</tr>
<tr>
<td>P27</td>
<td>O’Leary et al., 2009a</td>
<td>P37</td>
<td>Ghanam &amp; Maurer, 2010a</td>
</tr>
<tr>
<td>P28</td>
<td>Ghanam et al., 2008</td>
<td>P38</td>
<td>Hanssen, 2010</td>
</tr>
<tr>
<td>P29</td>
<td>Ghanam &amp; Maurer, 2008</td>
<td>P39</td>
<td>Mohan et al., 2010</td>
</tr>
<tr>
<td>P30</td>
<td>Ghanam &amp; Maurer, 2009</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. To cut down the long-term investment during the DE phase. Kansala (P2) and Taborda (P3) identify the challenges to be faced by SPL in volatile, dynamic market conditions. DE requires to completely define and plan the commonality and variability for all products of the SPL. It means an upfront long-term investment in designing a set of core-assets/platform (P18) [Schmid & Verlage, 2002b]. The upfront long-term design results in a SPL platform that is flexible, in the sense that it is capable of accommodating variation for those anticipated changes, i.e. the identified variabilities across the products of the SPL. Companies can leverage this investment when new products are developed through systematic reuse of core-assets by deriving the SPL variability points with the selected variants. Although it has been considered successful [Pohl et al., 2005], this strict “domain-then-application” model is resource intensive and risky (P28). Since core-assets are planned with a long term perspective, there is a risk of developing core-assets that will become obsolete and never be used in product derivation.

2. To deal with volatile business situations. When market stability decreases, the risk of developing core-assets that will become obsolete and never be used in
3.2 Systematic Literature Review: the method

Table 3.5: SLR - Distribution of studies according to the publication channel.

<table>
<thead>
<tr>
<th>Publication channel</th>
<th>Type</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>APLE</td>
<td>Workshop</td>
<td>7</td>
</tr>
<tr>
<td>SPLC</td>
<td>Conference</td>
<td>6</td>
</tr>
<tr>
<td>J Syst Software</td>
<td>Journal</td>
<td>4</td>
</tr>
<tr>
<td>XP</td>
<td>Conference</td>
<td>4</td>
</tr>
<tr>
<td>CEE-SET</td>
<td>Conference</td>
<td>3</td>
</tr>
<tr>
<td>Agile</td>
<td>Conference</td>
<td>2</td>
</tr>
<tr>
<td>J of Object Technology</td>
<td>Journal</td>
<td>2</td>
</tr>
<tr>
<td>PROFES</td>
<td>Conference</td>
<td>1</td>
</tr>
<tr>
<td>VaMoS</td>
<td>Workshop</td>
<td>1</td>
</tr>
<tr>
<td>ADC</td>
<td>Conference</td>
<td>1</td>
</tr>
<tr>
<td>Essay</td>
<td>Workshop</td>
<td>1</td>
</tr>
<tr>
<td>SDG</td>
<td>Conference</td>
<td>1</td>
</tr>
<tr>
<td>CIMCA,IAWTIC, ISE</td>
<td>Conference</td>
<td>1</td>
</tr>
<tr>
<td>Rise</td>
<td>Workshop</td>
<td>1</td>
</tr>
<tr>
<td>EuroSPI</td>
<td>Conference</td>
<td>1</td>
</tr>
<tr>
<td>PLEASE</td>
<td>Workshop</td>
<td>1</td>
</tr>
<tr>
<td>Split</td>
<td>Workshop</td>
<td>1</td>
</tr>
<tr>
<td>IEEE Software</td>
<td>Magazine</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>39</strong></td>
</tr>
</tbody>
</table>

Product derivation increases. Then, the upfront effort for planning and designing SPLs might be wasted on changing market situations (P3). Käänsälä (P2) defines a software process knowledge-base for Nokia, called *Business Framework for SW Process* to map software processes to the size and complexity of software and business volatility. This framework identifies a *sweet zone*, where traditional approaches tackle the increasing size and complexity of software through SPLE. It also points out a *difficult zone*, where large, complex software projects have to deal with volatile business situations. According to Käänsälä, business volatility requires alternatives to supplement SPLE, and ASD is proposed as a promising alternative.

3. To deal with situations where there is not enough knowledge about DE. DE and the resulting built-in flexibility have proven to be a crucial success factor in contexts where anticipated changes can be predicted with certain accuracy. If SPL developers do not have enough knowledge to perform the DE, ASD may facilitate the elicitation of further knowledge (P8). ASD expects limited predictability targeting smaller-scale development efforts, assuming that future changing scenarios cannot be anticipated (P18). As a result, when anticipated
changes cannot be predicted and the software product lifecycle is not known, it
would be advantageous to use an incremental approach, such as APLE (P5).

Therefore, the long-term investment that is carried out during the DE phase of a
typical SPL development, could be replaced by an agile approach when changes cannot
be anticipated. Changes cannot be anticipated when volatile business situations or the
knowledge on DE is not sufficient. Since efforts are focused on limited predictability,
risks are reduced. According to (P8), trade-offs between SPLE and ASD provide the
opportunity to apply the APLE approach to a wider variety of projects, rather than
those served by ASD or SPL methods. This is the case for critical systems, large-scale
systems, or those systems that require distributed teams during their development.
Most of the authors agree that SPLE and ASD are complementary when dealing with
the issues of large-scale product families in volatile markets (P2-P5, P8, P9, P11, P14,
P18, P20, P22, P24-39).

RQ2: How are the principles of SPLE and ASD related to each other?

Tian et al. (P8), Hanssen et al. (P14, P18, P38), and Mohan et al. (P39) compare
SPLE and ASD with the aim of determining the feasibility and conflicts of bringing
together both approaches. Although both SPLE and ASD pursue common goals such as
improving customer satisfaction and flexibility, and reducing time-to-market, SPLE and
ASD apply different strategies to achieve these goals (P4-P10, P18, P39). On one hand,
SPLE stresses the importance of predicting changes at the beginning of the process,
and the need to define a PL Architecture to support customization. In this sense, it
is capable of accommodating predicted changes to potential members of the product
line. On the other hand, ASD emphasizes value delivery to the customer and welcomes
changes by means of incremental development and close iterations with customers.
However, ASD methods (e.g. XP and Scrum) advocate for minimal investment in
an upfront architecture when knowledge is not readily available, and encourage the
continuous improvement and refactoring of the architecture to achieve the business
goals. In fact, agile methods have a reputation for paying very little attention to
software architecture (P18) and some agile practitioners even advocate against investing
effort in architecture specification, as it is perceived as wasted effort [Cockburn, 2006].

McGregor (P31, P32) presents a theoretical attempt to reconstruct a hybrid method
from SPLE and ASD principles. He concluded that although the integration of both
paradigms is difficult, SPLE and ASD can be tailored under the condition that both
3.2 Systematic Literature Review: the method

should retain their basic principles. Following this same idea, Hanssen et al. (P18) assert that, although at first sight SPLE and ASD may seem contradictory, they actually complement each other. An overall mapping between SPLE and ASD principles is carried out by Hanssen et al. (P18) to identify similarities, as well as differences between both the approaches. Table 3.6 summarizes this mapping, but the complete mapping with the 12 agile principles can be found in (P18). If we focus on principles, it seems feasible to tailor SPLE with ASD to obtain an approach that (i) analyzes the most significant commonalities in a domain, rather than an exhaustive set; and (ii) meets changing customer requirements, rather than just simply customizing core-assets (P8).

Table 3.6: Matching ASD and SPLE. Adapted from [Hanssen & Figri, 2008].

<table>
<thead>
<tr>
<th>Agile Principles</th>
<th>SPLE correspondence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliver working software frequently</td>
<td>Product delivery might be slowed down compared to ASD, due to overhead in maintaining integrity with the SPL platform</td>
</tr>
<tr>
<td>Welcome changing requirements</td>
<td>Changes planned in variability models are cheaply and quickly implemented (configurability)</td>
</tr>
<tr>
<td>Continuous attention to technical excellence</td>
<td>SPL platforms encode technical excellence and design to support rapid product derivation</td>
</tr>
<tr>
<td>The best architectures, requirements, and design emerge from self-organizing teams</td>
<td>Investments in requirements, architectures, and designs can be reused across multiple products</td>
</tr>
<tr>
<td>Continuous delivery of valuable software</td>
<td>Dependent upon SPLE approach</td>
</tr>
</tbody>
</table>

RQ3: How are SPLE and ASD positioned in respect to business strategic objectives?

This question is especially significant as it is close to the organization top management, which will provide the basic operations guidelines. At a very high level, both SPLE and ASD, by definition, are aligned with business strategy: SPLE scoping is obtained from this business strategy; ASD claims that value delivery to the customer is an utmost important objective, and customer participation is a usual practice. Strategic objectives are dealt with by (P18, P39). While both combine SPLE and ASD, they follow a different approach in respect to strategic objectives. In (P18), SPLE is used at a strategic level while ASD is used at a tactical level, i.e. project management. (P39) highlights the importance of top management to (i) manage the relationships
among technological, organizational, and strategic objectives, (ii) continuously assess the alignment of these objectives, and recognizing the evolutionary nature of these relationships; and (iii) cope with a considerable flexibility in development processes. While radically different, the (P18) and (P39) approaches provide a good basis for their justification. Hanssen et al. (P18) work with the hypothesis that ASD is useful for the tactical level, which is to assume that the scope of ASD planning is short term. Mohan et al. (P39) recommend having ASD at strategic level, which is mid to long term planning, in order to foster flexibility. Examples of deploying the tactical level can be found in (P3, P6, P16). This different interpretation and scoping on how to take SPLE and ASD into practice, finds an explanation and guidance in (P35). Codenie et al. (P35) explains how it can be sensible to consider: the kind of application, all kind of customer needs, strategic objectives, and the flexibility degree of the approach. Which means that organizations should analyze the consequences of deciding a higher or lower level of mass customization and/or flexibility in terms of investment, as well as, business, organizational and technological factors.

From these studies we can conclude that the current literature has addressed the theoretical mapping between the principles of both SPLE and ASD, and how these principles can live together. All authors agree that specific trade-offs for each project are necessary to successfully support APLE.

RQ4: Which current approaches combine SPLE and ASD satisfying Application Engineering activities? RQ5: What are the challenges and gaps in current approaches that combine SPLE and ASD during Application Engineering activities?

The research work of Taborda (P3) proposes the implementation of a release matrix to support Release Planning and tracking of SPLs in ASD. This release matrix allows us to plan and track the evolution of the PL Architecture over time in an incremental release strategy, addressing multiple products, components and their inter-dependencies. However, it lacks formalization, and deals with release planning documentation rather than how to deal with the actual product release.

Together with Release Planning, Configuration Management (CM) is a key activity to ensure successful SPLs. The management, maintenance, and tracking of multiple artifacts derived from a SPL requires a substantial and complex effort. Therefore, more powerful and robust CM tools are required than that of a single-system development. Kurman (P6) addresses this problem arguing that: (i) automatic product configuration
3.2 Systematic Literature Review: the method

is a key issue for APLE, and (ii) automatic building and deployment have to be 100% reproducible at any time.

Carbon et al. (P5) strongly believe that SPLE and ASD must be combined, arguing that ‘a combined SPLE approach would thus build-in flexibility for the most probable anticipated changes (proactive), while being agile enough to quickly incorporate changes that were not anticipated (reactive) and for no flexibility was built in’. The authors describe how Agile methods are used in a product derivation process called PuLSE-I [Bayer et al., 2000], where planning games and incremental design are normal practices. Software products are developed in such a way that a first version includes the core functionality, and further iterations add new functionality. It satisfies the agile principle of early and continuous delivery of working software. The authors also highlight an experiment to study upfront flexible design vs. incremental design from a return on investment perspective. However, the experiment results are not described in detail. In a later publication, Carbon et al. (P22) stress the importance of feedback from application to domain engineering in order to minimize the degeneration of the product line infrastructure, and thus maximize its viability. The feedback mechanism proposed by Carbon et al. is the planning game, where application engineers take over the role of customers and domain engineers take over the role of developers. The application engineers specify suggestions to the domain engineers on how to improve the reusability of core-assets for specific product applications.

O’leary et al. (P24-P27) provide a much more comprehensive framework to cover all activities for Product Derivation (PD) than the previous authors. They define an Agile Framework for Product Derivation (AFPD) which assists organizations in using a structured approach for PD activities and provides a means of integrating Agile practices in the PD process. AFPD considers the adoption of early and continuous delivery strategy, automation of product derivation, product derivation iterations, and agile testing techniques. O’leary’s work on SPLE area has provided a set of key activities that any PD approach should consider, as well as a set of issues to be addressed [O’Leary et al., 2009c] in a short-medium term. These issues are related as they provide mechanisms for supporting effort estimation, traceability and synchronization between platform and product teams (DE and AE teams, respectively). They also show how to represent product line variability from different points of view, for instance, how to represent variability in the language of the product derivation stakeholders. These issues include a number of challenges that are still no completely solved for SPL.

RQ6: Which current approaches combine SPLE and ASD satisfying Domain Engineering activities? RQ7: What are the challenges and gaps
3. AGILE PRODUCT LINE ENGINEERING

in current approaches that combine SPLE and ASD during Domain Engineering activities?

According to the panel “Agile and Product Line Engineering”, APLE’08 (P20), it is not clear how agile methods could be applied to DE. Nevertheless, some attempts have been made to address its applicability.

The work of Noor (P7), Feng (P13), and Trinidad (P17) tackle very specific DE activities such as product-line scoping, requirements engineering or feature modeling. Noor et al. (P7, P15, P16) provide an approach to collaboratively support the product line scoping activity, which includes agile principles, such as stakeholder involvement, rapid feedback, or value-based prioritization. Feng et al. (P13) provide some initial results on how to select a requirements engineering (RE) process, which provides the necessary degree of agility and the support for specifying a product line application. They gather the knowledge that influences the RE process selection with a tailored level of agility. Trinidad et al. (P17) present an automated error analysis method to detect errors (e.g. dead features) in feature models. Since feature models are usually strongly affected by changes in requirements, and one of the objectives of agile methods is a rapid response to requirement changes, (P17) makes more agile feature modeling.

The work of Paige (P11) and Kakarontzas (P23) tackle the design of PL Architectures. Paige et al. (P11) propose building software product lines by using the agile method Feature Driven Development (FDD), in such a way that SPL development process can benefit from agile development techniques. This can be used to identify SPL features, configurations, and variability points. Hence, the main challenge is how to integrate PL Architecture design into FDD. To address this challenge, Paige et al. propose that architecture should be incrementally designed, and SPL variations should be generated as a result of the agile refactoring practice. Even though this proposal of SPL and FDD integration shows potential, no concrete data, process description, or cases were presented. Moreover, considerations involving incremental design of PL Architecture were not described. Meanwhile, Kakarontzas et al. (P23) present an approach based on Test Driven Development (TDD) and elastic components, to specify product line variability. Elastic components address the configuration and evolution of components by means of adding/deleting/modifying variants that hook from the root component. TDD ensures that the functionality and quality of the product is not degraded during evolution. This work is a first step to show how PL Architectures can be tailored to be more Agile. However, since their variants are context-dependent, additional mechanisms are needed to make variants independent of the root component. Context-independent variants may (i) facilitate the incremental and iterative evolution
of PL Architectures in ASD by reducing the number of changes; and (ii) make the reuse of variants among the products of a SPL more flexible.

Finally, Raatikainen et al. (P21) introduce a tool based approach to monitor, plan and measure the development process of a SPL. This tool integrates Kumbang (a feature modeling tool), and Agilefant (an agile backlog tool). Even though the combination seems promising, no concrete data or cases were presented and the agility of the method was not described.

RQ8: Which current approaches combine SPLE and ASD satisfying both Domain Engineering and Application Engineering activities through Agile Principles?

The integration of SPLE and ASD principles and practices, satisfying both DE and AE, is quite challenging. It is only addressed by the work of Ghanam et al. (P28-P30, P37). While all previous works target SPL-based organizations that want to make their product development more agile, Ghanam et al. (P28-P30, P37) target agile organizations that wish to establish a SPL. The authors highlight the importance of mining systems, keeping in mind the reuse, and the formalization of commonality and variability through acceptance tests. Hence, they introduce an iterative model for APLE and the use of TDD to support this process. Specifically, they propose a bottom-up application-driven approach that relies on automated acceptance tests to derive core-assets from existing code. Their approach uses executable acceptance tests to support variability management and traceability: an acceptance test has a generic fixed component and a variable component. To bridge the gap between SPLs and the XP agile method, executable acceptance tests are used to describe the system and act as anchor points for traceability relations. Through a bottom-up approach, the SPL is iteratively built from existing product instances, the SPL platform evolves progressively in an iterative approach, and variability is handled on-demand, i.e. reactively. (P37) presents a tool for assisting the refactoring of code when a developer introduces a variation. The code-based tool requests the method that is causing the variation from the developers, and then creates an abstract factory for this method, the corresponding concrete classes, and the required test classes. This approach provides a novel contribution based on TDD to model variability dealing with DE and AE in a context where the XP method is used. Despite the fact that this contribution offers a significant advance in the area, more work is still necessary to support coarse-grained variability. This means that it is not always possible to trace a variation with a class or a method.
3. AGILE PRODUCT LINE ENGINEERING

Otherwise, a variation could be traced to a set of methods, classes, attributes, or even components.

RQ9: Are there successful industrial experiences putting APLE into practice?

Kane (P1) describes his experience incrementally introducing agile development techniques into a bioinformatic institute. This experience report identifies several challenges, including how to better manage a SPL with agile techniques and specifically how to balance the priorities between the products of a SPL. Another industrial experience, by Hanssen et al. (P18), describes a successful case study of integrating SPLE and ASD practices. They use SPL practices as a strategic platform for developing products with the agile EVO method [Johansen 2005]. Kircher et al. (P12) identify a collection of successful best practices when they introduce a SPL approach in Siemens business groups. They assert that SPLs cannot be considered in isolation in innovative business, as the usage of agile processes may facilitate fast feedback cycles between requirements engineering, development, and field trial. A key issue that is not addressed, is how a SPL approach integrates the best practices of Agile methodologies.

Another empirical study was developed by Ali Babar et al. (P36) who describe a successful industrial case study and analyze the organizational processes and practices that were used to integrate SPLs and ASD. The authors describe a development process that consists of three sub-processes: product line platform, exploration before agile product development, and agile product development. It was found that architecture and architectural communication support these three processes and, reciprocally, these processes may update product architectures by means of refactoring. Although this approach is valid and compelling, it focuses on a reuse-centric AE process, and does not discuss the role of agile methods in DE. As a result, it is closer to the concept of agile software development using a product line platform than the concept of APLE.
3.3 Key Findings: Implications for Practitioners and Researchers

From this systematic literature review two main ideas can be highlighted.

- First, practitioners can conclude that there are sufficient reasons to move towards a combination of SPLE and ASD.
- Second, researchers can conclude that there are still some important challenges in the area, and therefore, more research work is required to completely put APLE into practice.

In regard to the first idea, the literature review reports that APLE would be applicable to business situations where the convenience of going towards a product line has been identified, but at the same time the market situation is not stable enough for different reasons, including technological factors. The review has identified four main advantages of putting APLE into practice, and when it is advantageous:

1. If SPL developers do not have enough knowledge to completely perform the DE, ASD may facilitate the elicitation of further knowledge (P8).
2. Trade-offs between SPLE and ASD provide the opportunity to apply the APLE approach to a wider variety of projects than those served by only applying ASD or SPL methods (P8).
3. When anticipated changes cannot be predicted and the product lifecycle is not known, it would be advantageous to use an incremental approach such as APLE (P5).
4. Agile processes may facilitate fast feedback cycles between requirements engineering, development, and field trial in innovative business (P12).

In regard to the second idea, the literature review has reported the following three main findings:

1. The applicability of agile methods to DE requires more effort than the application of AE. This is due to the fact that is difficult to reduce the upfront design with the aim of getting closer to agile principles and values, while achieving the typical
3. AGILE PRODUCT LINE ENGINEERING

goals of DE, such as reuse (P33). Nevertheless, the applicability of agile methods to AE does seem feasible. The works by Carbon et al. (P5, P22) and O’Leary et al. (P24-P27) provide a complete support for product derivation.

2. Synchronization between platform and product teams (DE and AE teams, respectively) is vital in APLE, as DE and AE should not be separated. The platform should be synchronized with the application needs to avoid the platform becoming obsolete (P26). It is still a challenge for APLE practitioners (P33).

3. Business objectives should be used to identify the right level of flexibility. This level should be useful in determining the combination of SPLE and ASD: either SPLE and ASD at strategic level, or SPLE at strategic level and ASD at tactical (P8, P18, P39).

Since this systematic literature review pursues to be useful for both practitioners and researchers, the analysis of the most relevant findings have been classified into two points of view: (i) How can APLE be put into practice?, i.e. what problems have been solved in APLE deployment and which approaches/practices can be considered for APLE adoption; and (ii) Which are the open research challenges?, i.e. what is still missing and what are the challenges/barriers researchers should cope with? The analysis from these points of view is detailed in the following subsections.

3.3.1 How can APLE be put into practice?

Table 3.7 shows a global view of the contributions and approaches (exactly 15 approaches) that practitioners interested in APLE can put into practice in their developments. This table highlights practices and experiences (30 approx.) which completely or partially solve the most recurrent challenges in APLE. Table 3.7 also classifies these practices and experiences based on: the process/es (DE or AE) they address, which specific activity they deal with, and which research question of this systematic literature review they answer. We have categorized the contributions and approaches for each challenge that has been identified in this systematic literature review as follows:

1. Whereas SPLE stresses the importance of the upfront investment, planning and design, ASD emphasizes delivering value to the customer rapidly and frequently, advocates the welcoming of changes, and has the reputation for paying little attention to the design. As a result, their combination is a challenge. Five contributions/approaches address this challenge. Specifically, the work of Hanssen
3.3 Key Findings: Implications for Practitioners and Researchers

et al. (P14, P18, P38) and Ghanam et al. (P28-P30, P37) provide industrial case studies (P18) and tool support (P37) in regards to applying XP and EVO agile methods to SPL development.

2. The welcoming of changes, that ASD suggests, requires mechanisms to support it. Three contributions/approaches address this challenge by means of mechanisms to deal with the change in feature models (P10, P17), product line scoping (P7, P15, P16), and requirements engineering processes (P13). These contributions provide successful industrial case studies and tool support. However, an issue not dealt with is how changes are managed in successive activities such as architecture, design, product derivation, etc.

3. As SPLE is tailored in favor of an agile incremental development in short iterations, the PL Architecture must also support its incremental design. Two contributions/approaches address this challenge (P11, P23), but they do not report case studies to illustrate their applicability.

4. As SPLE is tailored in favor of an agile incremental development in short iterations, the product derivation process must also support the frequent delivery of value to the customer. Five contributions/approaches address this challenge (P3, P5, P6, P24-P27, P36). These contributions provide successful industrial case studies and tool support for the entire product derivation process and specific activities, such as release planning or product configuration.
### Table 3.7: SLR - Summary of contributions/approaches working on a challenge.

<table>
<thead>
<tr>
<th>Challenges</th>
<th>Contributions &amp; Approaches</th>
<th>Practices</th>
<th>Experiences</th>
<th>DE/AE Activities</th>
<th>Activities</th>
<th>Research Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>A business framework maps software process with software size and complexity and business volatility (P2).</td>
<td>–</td>
<td>–</td>
<td>Nokia experience</td>
<td>–</td>
<td>–</td>
<td>RQ1</td>
</tr>
<tr>
<td>A lighter-weight approach by tailoring a SPL approach with Agile practices (P8).</td>
<td>Analysis of the most significant commonalities in a domain, rather than an exhaustive set. Welcome to change customer requirements, rather than just simply customization.</td>
<td>–</td>
<td>DE</td>
<td>–</td>
<td>DE</td>
<td>RQ2</td>
</tr>
<tr>
<td>Combination of the up-front investment, effort, planning and design of SPL with the incremental and “welcome to changes” development of ASD (P18,P3,P2).</td>
<td>Fusion process of SPL and ASD using the agile method EVO (P14,P18,P38).</td>
<td>SPL addresses strategic long-term objectives and ASD addresses tactical short-term objectives.</td>
<td>Industrial case study.</td>
<td>DE/EA All</td>
<td>All</td>
<td>RQ2</td>
</tr>
<tr>
<td>An APLE approach based on an iterative model and the agile method TDD (P23,P30,P37).</td>
<td>Bottom-up application-driven approach that relies on automated acceptance tests (ATs) to derive core-assets from existing code. Formalization of commonality and variability through ATs. ATs act as anchor points for traceability relations. Agile refactoring to support variations.</td>
<td>Tool + case studies.</td>
<td>DE/EA All</td>
<td>All</td>
<td>All</td>
<td>RQ8</td>
</tr>
<tr>
<td>Approach to reconcile the Agile and CMMI Contexts in Product Line Development (P9).</td>
<td>Approach to reconcile the Agile and CMMI Contexts in Product Line Development (P9).</td>
<td>SPL, CMMI and Agile</td>
<td>–</td>
<td>DE/EA Orthogonal</td>
<td>Orthogonal</td>
<td>RQ9</td>
</tr>
<tr>
<td>Rapid response to changes in requirements</td>
<td>An automated error analysis to detect errors in feature models due to changes in requirements (P10,P17).</td>
<td>Release planning and tracking by means of a Release Matrix.</td>
<td>Tool + industrial case study</td>
<td>DE</td>
<td>Feature Modeling</td>
<td>RQ6</td>
</tr>
<tr>
<td>Collaborative support for the product line scoping activity (P7, P15, P16).</td>
<td>Stakeholder involvement, rapid feedback, or value-based prioritization.</td>
<td>Tool + industrial case study</td>
<td>DE</td>
<td>Product Line Engineering</td>
<td>RQ6</td>
<td>RQ6</td>
</tr>
<tr>
<td>A framework to select a RE process with the necessary degree of agility (P13).</td>
<td>A survey to collect expertise in APLE RE</td>
<td>–</td>
<td>DE</td>
<td>Requirements Engineering</td>
<td>RQ6</td>
<td>RQ6</td>
</tr>
<tr>
<td>Incremental PL Architecture</td>
<td>An APLE approach based on an iterative model and the agile method FDD (P11).</td>
<td>PL Architecture is realized incrementally, and SPL variations are generated as a result of the agile refactoring practice.</td>
<td>Case study.</td>
<td>DE</td>
<td>PLA Design</td>
<td>RQ6</td>
</tr>
<tr>
<td>Approach based on TDD and elastic components to specify variability in PL Architectures (P23).</td>
<td>Elastic components address the configurability and evolution of components. TDD ensures that the functionality and quality of the product is not undermined during the evolution.</td>
<td>–</td>
<td>DE</td>
<td>PLA Design</td>
<td>RQ6</td>
<td></td>
</tr>
</tbody>
</table>
### 3.3 Key Findings: Implications for Practitioners and Researchers

| Plan and track the evolution of a PL Architecture over time in an incremental release strategy (P3). | Release planning and tracking by means of a Release Matrix. | Trials of the Release Matrix in a large software development organization. | AE | Release Planning | RQ4 |
|---|---|---|---|---|---|---|
| Automatic product configuration (P6). | SPL approach acts as an overall development strategy and is combined with some agile software development techniques: daily user deployment test, welcome to changes, etc. Architecture and architectural communication support these three processes and, reciprocally, these processes may update product architectures by means of refactoring. | Industrial case study. | AE | Release Planning and Product Configuration | RQ4 |
| Agile Product Configuration and Derivation Development process made of three sub-processes: product line platform, exploration before agile product development, and agile product development (P36). | Agile methods in a product derivation process called PuLSE-I (P5). | Incremental design (early and continuous delivery of working software) Feedback from AE to DE to minimize the degeneration of the product line infrastructure, and thus maximize its viability. The feedback mechanism is the agile practice “planning game”. | AE | All | RQ4 |
| Agile Framework for Product Derivation (AFPD) (P24-27). | Adoption of early and continuous delivery strategy, automation of product derivation, product derivation iterations, and agile testing techniques. | Industrial case study. | AE | All | RQ4 |

| 4 challenges | 15 contributions/approaches | 30 practices aprox. | 7 industrial CS, 3 tools, 3 CS/trials, and 1 survey | 10 | DE | AE |
3. AGILE PRODUCT LINE ENGINEERING

3.3.2 Which are the open research challenges?

Tables 3.8 and 3.9 describe the activities and practices for AE and DE, respectively. These tables reveal that gaps exist and more research is required. These tables only consider the subset of papers that is relevant to the activities and practices being analyzed. Each row represents the work of an author that is relevant to a specific activity/practice, and the last row shows the total number of papers that address each activity/practice. The ✓ mark means that authors explicitly propose a solution to support an activity; the x mark means that authors explicitly identify a challenge or gap in that activity; finally the - mark means there is no evidence.

In view of Table 3.8 the applicability of agile methods to AE seems to be highly feasible. Twelve papers address AE, specifically ten of them propose solutions related to release planning, 11 of them are related to product configuration, 10 are related to product development, 10 are related to test, integration and deployment, 6 are related to traceability, and 10 of them are related to maintenance and evolution. Only one gap has been identified by the papers included in this systematic review. From the 6 papers that provide solutions for traceability, 4 papers report the problem to define traceability among the different SPL artifacts and the synchronization among product and platform teams.

In view of Table 3.9 the applicability of agile methods to DE presents lack of evidence for most of the activities/practices. Nevertheless, some solutions are provided: three papers tackle SPL scoping, one paper addresses requirements engineering, seven papers tackle feature model, and four papers address traceability. However two activities deserve special attention: architecture and traceability. Tian et al. (P8) explicitly recognized potential challenges and risks associated with APLE: (i) traceability management and maintenance of components might be difficult in agile approaches without explicit knowledge; and (ii) if PL Architectures are tailored to be more Agile, there is a danger that a valuable architecture supporting other products of the family may be damaged. As a result, APLE architecture and APLE traceability are still open research challenges.

In regard to the traceability challenge, work in progress (P28-30) has tackled traceability management. This work is inclined toward TDD approaches to take advantage of iteratively track and test software. However, additional effort is still necessary to be able to support traces between features and core-assets in order to easily implement maintenance tasks in a systematic and (semi)-automatic way.
3.3 Key Findings: Implications for Practitioners and Researchers

In regard to the architecture challenge, most approaches converge towards iteratively and progressively building the SPL platform (P8, P11, P18, P29, P30), and specifically, the PL Architecture (P11, P23). In practical terms, the PL Architecture should have mechanisms to support evolution, i.e. to flexibly be adapted. But, the agile design of PL Architectures has been pointed out as a key challenge to overcome, since none of the mentioned authors have completely solved it. Dealing with this challenge is essential for the goal of APLE, and can be widely adopted by the software industry (P8, P18, P29, P36). The key question is: how can the long-term planning and upfront architecture required by SPLE be taken care of while still being able to deliver value to the customer on time? (P36). Or from the opposite point of view: how can architecture be tailored to be more agile without losing the SPL reusability and flexibility? It should not be forgotten that software architectures are a key factor in the success of SPLs, while Agile considers that the architecture will gradually emerge in an iterative and incremental way. Thereby, Agile will pay attention to design, and much more attention to early delivery of valuable software (P18), putting customization in second place (P8). Following (P33), it may be useful to have a mechanism to specify qualities for a good PL Architecture which will allow us to perform a trade-off between upfront long-term design of SPLs and ASD short-term design.

Table 3.8: Summary for Application Engineering.

<table>
<thead>
<tr>
<th>References</th>
<th>Release Planning</th>
<th>Product Configuration</th>
<th>Product Development</th>
<th>Test, integration, deployment</th>
<th>Traceability and evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P3)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(P5,P22)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(P6)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(P24-27)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(P26-30,P37)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3.9: Summary for Domain Engineering.

<table>
<thead>
<tr>
<th>References</th>
<th>Scoping</th>
<th>Requirements Engineering</th>
<th>Feature Model</th>
<th>Architecture</th>
<th>Core Assets Development</th>
<th>Traceability and Maintenance</th>
<th>Evolution and Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P8)</td>
<td></td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>(P7,P15,P16)</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(P10,P17)</td>
<td></td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(P14,P18,P38)</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(P11)</td>
<td></td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(P13)</td>
<td></td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(P23)</td>
<td></td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(P24-27)</td>
<td></td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>(P26-30,P37)</td>
<td></td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

89
3. AGILE PRODUCT LINE ENGINEERING

3.3.3 Final considerations

To keep this systematic literature review up-to-date on publications, we have made an effort during the review process to include the latest papers and advances on APLE. Although these papers have not been included in the systematic analysis, it is important to mention some new approaches and techniques that contribute to putting APLE into practice. In [Ghanam et al., 2010a], the authors propose a framework to incrementally and reactively construct variability profiles for existing and new systems. The framework leverages common agile practices, such as iterative software development, refactoring, continuous integration and testing, to introduce variability into systems only when it is needed. In [Pérez et al., 2010], the authors attempt to solve the problem of designing agile software architectures by using the Plastic Partial Component (PPC) concept. PPCs are highly malleable components that can partially be described as that which increases the flexibility of architecture design. In fact, the notion of PPC was originally defined for SPLE to support the definition of internal variation of architectural components [Pérez et al., 2009]. Hanssen et al. [Hanssen et al., 2010] report how agility and entropy\(^1\) are negatively related. Although their contribution is not specific for SPLE, otherwise used for product evolution in general, a case study shows (i) how a successful software product line organization has adopted the agile development method Evo [Johansen, 2005]; and (ii) how to manage entropy while the process maintains agility. Finally, in [Ghanam & Maurer, 2010b], the authors propose a traceability mechanism to ensure consistency between feature models and code artifacts during evolution processes where ASD is practiced.

3.4 Conclusions

This chapter presented the results of a systematic literature review of practices and experiences with APLE. Although some authors disagree about the combination of SPLE and ASD, practically all of our findings support the fact that SPLE and ASD can work together as they both pursue the same high-level goals, but using different methods to achieve them. It has been acknowledged that the main advantage of APLE is that SPLE may be applied to volatile markets where changes cannot be easily predicted, as well as being supported by ASD to develop large-scale product families (SPLs). A significant issue is to identify the right level of flexibility according to the business

---

\(^1\)This concept is defined in [Hanssen et al., 2010] as how the maintainability of a system may degrade over time due to continuous change.
3.4 Conclusions

objectives. The level of flexibility is key in deciding if the approach is SPLE and ASD at strategic level, or SPLE at strategic and ASD at tactical.

Therefore, practitioners can conclude that there are sufficient reasons to move towards a combination of SPLE and ASD. Several researchers and practitioners conceive APLE as an evolutionary development process for product-lines. Hence, the works by [Díaz et al., 2011; Ghanam & Maurer, 2008; McGregor, 2008b] propose an approach in which domain-then-application phases are incrementally iterated. This provides a reactive approach in which reusable assets respond to current customer demands while (i) being able to iteratively, incrementally construct a flexible SPL platform and (ii) being able to deliver features to the customer on time. However, most authors agree that the applicability of agile methods in domain engineering requires more effort to meet the challenge of reducing the upfront design, while being able to get closer to agile principles and values. This is the starting point of this thesis. In this way, the thesis focuses on one of the main challenges that slows down putting APLE into practice: the reconciliation of agility and architecture.

This chapter is an excerpt from the following article:


Copyright ©Wiley & Sons, Ltd.
3. AGILE PRODUCT LINE ENGINEERING
Part III

CONSTRUCTION AND EVOLUTION OF PLAs
Chapter 4

Preliminaries

This chapter describes an empirical pilot study which has been conducted to exemplify the mechanisms that are described in the following chapters, as well as to prepare the case study that validates the complete approach presented in this thesis. In addition, this chapter also describes the agile method Scrum in which the main result of this thesis, i.e. the APLA process, is deployed.

4.1 Empirical Pilot Study: A SPL for banking systems

The empirical pilot study is applied to the banking system domain. A banking system is a well-known and commonly used exemplar in the software engineering area for validating new approaches, methods and tools [Gondal et al., 2011; Liu, 2010; Mitra et al., 2005; Wang & Mylopoulos, 2009]. Specifically, the architectural design of banking systems has been extensively studied [Buede, 2009; Nord et al., 2004] which facilitates its comprehension and allows one to easily compare previous approaches with new ones. The architectural design of banking systems has been extended to the SPL paradigm so that it can be used to illustrate how describing variability of PLAs, as well as how documenting product-line architecture knowledge.

Figure 4.1 shows an overview of the SPL for banking systems. Banking systems typically consist of a set of core components that offer their functionality to Automatic
4. PRELIMINARIES

Teller Machines (ATM) and bank web applications (WebApp). Both ATM and WebApp aim to provide a cost-effective service to bank customers that is convenient, safe, and secure 24-hour access for realizing a common set of banking transactions and reducing the cost of providing these basic transaction. Among the most common banking transactions, this pilot study focuses on the functionalities of account balance, withdrawal and deposit of funds, and money transfer. ATM customers can make balance inquiries and money transfers, withdraw and deposit money, whereas WebApp customers can make balance inquiries and money transfers.

ATMs and WebApps connect to the main banking system through an access point. The first step during the connection is the process of validation, which is different in the case of ATMs and WebApps. In the case of ATMs, valid general identification information is a debit/credit card, so that customers manually enter their card into the ATM and a unique identification information — the PIN. In the case of WebApps, valid general identification information is a code: the customer enters their ID, birthdate, and next they must enter a unique identification code. Once the validation has been successful, a ticket issuer module generates a number that uniquely identifies the banking transaction. Subsequent message exchanges with the bank indicate the ticket number.

![Figure 4.1: Exemplar of banking software systems](image)

**Figure 4.1: Exemplar of banking software systems** - Banking systems typically consist of a set of core components that offer their functionality to ATMs and bank web applications.

The operational concept of banking systems is comprised of a group of specific scenarios that are based upon the requirements of both ATM and WebApp customers. These scenarios are an adapted excerpt from those defined by Buede, 2009:
1. Customer makes deposits
   a. Customer provides valid general identification information.
   b. ATM requests unique identification information
   c. Customer enters unique identification information.
   d. ATM requests activity selection.
   e. Customer selects deposit.
   f. ATM requests type of deposit (cash vs. check).
   g. Customer identifies type of deposit—cash/check.
   h. ATM provides a means to physically insert cash/check into ATM.
   i. Customer enters deposit.
   j. ATM transmits the transaction to the main banking system, gives customer receipt, returns to main menu.

2. Customer requests cash to be withdrawn from an account.
   a. Customer provides valid general identification information.
   b. ATM requests unique identification information
   c. Customer enters unique identification information.
   d. ATM requests activity selection.
   e. Customer selects withdrawal.
   f. ATM requests amount of withdrawal.
   g. Customer identifies amount of withdrawal ($C_{req}$).
   h. ATM contacts the main banking system and requests the amount of available funds from the customer’s account ($F_{max}$).
   i. If $C_{req} > F_{max}$, ATM denies request.
   j. If $C_{req} > C_{lim}$, ATM denies request. ($C_{lim}$ is the maximum cash withdrawal allowed.)
   k. Else, ATM transmits the transaction to the main banking system, gives customer receipt, gives the customer money, and returns to the main menu.

3. Customer requests transfer of funds from one account to another.
   a. Customer provides valid general identification information.
   b. ATM/WebApp requests unique identification information
   c. Customer enters unique identification information.
4. PRELIMINARIES

d. ATM/WebApp requests activity selection.

e. Customer selects transfer of funds.

f. ATM/WebApp requests account for destination of funds transfer

g. Customer identifies destination account.

h. ATM/WebApp queries the main banking system to determine the availability of funds from the source account ($F_{max}$).

i. ATM/WebApp requests the amount of the funds transfer.

j. Customer identifies the amount of funds to be transferred ($F_{trns}$).

k. If $F_{trns} > F_{max}$, ATM/WebApp denies the request.

l. Otherwise the funds are transferred, ATM/WebApp transmits the transaction to the main banking system, gives the receipt, and returns to the main menu.


   a. Customer provides valid general identification information.

   b. ATM/WebApp requests unique identification information

   c. Customer enters unique identification information.

   d. ATM/WebApp requests activity selection.

   e. Customer selects balance status of an account.

   f. ATM/WebApp queries the main banking system to obtain the needed information, gives customer receipt, and returns to the main menu.

5. The main banking system is not working.

   a. Customer attempts to provide valid general identification information or is performing either of the scenarios 1-4.

   b. If there are too many requests, the main banking system rejects the request (overload status).

   c. ATM/WebApp informs customer that the main banking system is not working and aborts the session with the customer.

This pilot study also focuses on the non-functional feature of availability 24/7. Banking system applications should guarantee availability 24 hours 7 days per week of their core functionality to ATMs and WebApps. Several stakeholders require their banking system applications to have strict 24/7 availability, while others permit a weaker, non-strict availability. Strict availability must provide recovery and repair in
milliseonds, whereas non-strict availability is less available and cheaper. Therefore, the strictness of availability is a variability point.

Figure 4.2 shows the feature model of the described above banking systems SPL. This model is composed of the following features: the core banking transactions, ticketing and access functions offered to ATMs and WebApps, as well as the variability point that defines the variations strict and non-strict availability for banking system functionality.

Figure 4.3 shows the product-line architecture of the banking systems SPL, i.e. the components, connectors and interactions among them which make up the core configuration of the PLA. The components ATM and WEBApp implement the external systems to which banking system applications offer their functionality. ATMs and WEBApps connect to the main banking system through two access points which have been architecturally designed as front end services. The components ATMFrontend and WEBFrontend abstract and simplify the communication with the underlying components that provide the common banking transactions by providing a well-defined application programming interface. These underlying components implement the core functionality of banking systems applications: Balance, Withdrawal, Deposit and Transfer (see Figure 4.3). Finally, the component TicketIssuer manages the unique identification of banking transactions.

The components Balance, Withdrawal, Deposit and Transfer implement the services: balance, withdraw, deposit, and transferToInternalAccount, transferToExternalAccount, and transferToForeignAccount, respectively (see Figure 4.4), which will be provided by a set of interfaces. The component TicketIssuer implements the service getUniqueIdentifier (see Figure 4.4), which will be provided by an interface. The components ATMFrontend and WEBFrontend provide different implementations of the service processRequest (see Figure 4.4), which will be provided by two different interfaces. Finally, through their respective interfaces, the components ATMFrontend and WEBFrontend also require the services: balance, withdraw, deposit, transfers and getTicket (see Figure 4.4).
Figure 4.2: Banking systems SPL - Feature model
Figure 4.3: Banking systems SPL - Architecture model. Component & Connector view
Various architectural tactics used to realize availability are proposed in the literature [Bass et al., 2003; Scott & Kazman, 2009]. We have selected active redundancy and passive redundancy tactics to implement strict and non-strict availability, respectively. These tactics are described as follows:

The tactic Active Redundancy (see Figure 4.5) is based on a “configuration wherein all of the nodes (active or redundant spare) in a protection group receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s)” [Scott & Kazman, 2009]. Therefore, from the architectural view, this tactic requires: (i) a load balancer in order for all nodes —active and redundant nodes— to process identical inputs, and (ii) a synchronizer in order for the active and redundant nodes to maintain an identical state. If there is a failure, the repair occurs on time as the redundant spare has an identical state to the active node.
4.1 Empirical Pilot Study: A SPL for banking systems

The cost of this tactic is high due to the cost of synchronization between redundant spare and active node(s).

![Figure 4.5: Active redundancy](source: Scott & Kazman, 2009)

Active or redundant nodes receive and process identical inputs in parallel.

The tactic Passive Redundancy (see Figure 4.6) is based on a “configuration wherein only the active members of the protection group process input traffic, with the redundant spare(s) receiving periodic state updates” [Scott & Kazman, 2009]. Therefore, from the architectural point of view, this tactic requires: (i) a router to ensure that only the active node process all the inputs, as well as to change the route to the redundant node(s) when there is a failure, and (ii) a periodic data controller in order for active and redundant node(s) to maintain periodic state updates. If there is a failure, the router selects a redundant spare after checking the state update. This tactic achieves a balance between (i) the more highly available but more complex active redundancy tactic and (ii) the less available but significantly less complex spare tactic. Table 4.1 summarizes the mechanisms and results of both tactics.

![Figure 4.6: Passive redundancy](source: Scott & Kazman, 2009)

Only active nodes process input traffic.

From the descriptions of active and passive redundancy, the following architectural design decisions are made: On one hand, the component types \{Balance, Withdrawal, Deposit, TicketIssuer and Transfer\} described in Figure 4.3 are replicated by multiple instances in a set of nodes that constitute a protection group\(^1\). On the other hand, the service processRequest of components ATMFrontend and WEBFrontend is in charge of

\(^1\)A group of processing nodes where one or more nodes are “active” with the remaining nodes in the protection group serving as redundant spares [Scott & Kazman, 2009].
4. PRELIMINARIES

Table 4.1: Architectural Tactics for Availability. From Scott & Kazman [2009]

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Mechanism</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Redundancy</td>
<td>Configuration wherein all of the nodes (active or redundant spare) in a protection group receive and process identical inputs in parallel</td>
<td>Redundant spare possesses an identical state to the active processor, so recovery and repair can occur in milliseconds</td>
</tr>
<tr>
<td>Passive Redundancy</td>
<td>Configuration wherein only the active members of the protection group process input traffic, with the redundant spare(s) receiving periodic state updates</td>
<td>Achieves a balance between the more highly available but more complex active redundancy tactic and the less available but significantly less complex spare tactic</td>
</tr>
</tbody>
</table>

doing the load balance in the case of active redundancy, whereas it is in charge of routing the requests only to the active node in the case of passive redundancy. The service `processRequest` needs other services to implement the active and passive redundancy tactics: The service called `updateStateDaemon` is in charge of synchronizing active and passive nodes in the case of active redundancy, whereas it is in charge of doing periodic data synchronizations in the case of passive redundancy. The service called `processFault` is in charge of changing the route of requests when the active node fails in the case of passive redundancy. These services are also provided by both `ATMFrontend` and `WEBFrontend` as Figure 4.7 shows.

![Figure 4.7: Front end services - Services specification](image)

4.2 Scrum

The APLA process and the supporting mechanisms defined in this thesis, are deployed in the agile method Scrum [Schwaber & Beedle, 2002]. Scrum implements an iterative and incremental lifecycle (see Figure 4.8). Three roles, the Product Owner, the Scrum
4.2 Scrum

Master, and the Team, together make up the Scrum Team. The product owner represents the key stakeholder interests, the team is in charge of developing the product functionality, whereas the scrum master is in charge of the process.

The Scrum process starts with the capture of the requirements of the product owner from the product vision (features). Then, features may be decomposed into a list of user stories (US) known as product backlog (see Figure 4.8). US describe the product features using scenarios written by customers without techno-syntax and including the acceptance criteria that validate them. After that, US are prioritized, based on business value, and assigned to sprints, understanding a sprint as a 2-4 weeks period of development time.

The Scrum lifecycle is composed of a set of these sprints. Sprints have a sprint planning meeting at their beginning in which the product owner and the team plan together what has to be done (see Figure 4.8).

At the end of each sprint, a working product is delivered (see Figure 4.8). In the sprint review meeting the product owner assesses the working product to validate that US were met or to introduce changes into the US (see Figure 4.8). The sprint retrospective meeting is held in order to put continuous improvement into practice as follows: what went well and what could be improved for the next sprint (see Figure 4.8).

**Figure 4.8: Scrum lifecycle** - Scrum implements an iterative and incremental lifecycle that is divided into sprints

105
4. PRELIMINARIES

4.3 Conclusions

This chapter has detailed the agile method Scrum and the software requirements and operational scenarios of an exemplar which will be used in the next chapters to illustrate the APLA process and supporting mechanisms that this thesis defines. This exemplar consists of a family of banking systems which specifies several common features for all banking system applications, such as the functionalities of balance, withdrawal, deposit or transfer of money. It also points out variable features which are specific for several banking system applications, such as the different levels of strictness of the availability of their functionality through ATMs and bank websites. We have analyzed two architectural tactics to realize strict and non-strict availability. These are: active redundancy and passive redundancy.
Chapter 5

Flexible Product Line Architectures

This chapter presents a mechanism for describing PLAs from the structural point of view. Its main purpose is to deal with objective 1 (OBJ 1) defined in Section 1.3, i.e., to provide PLAs with flexibility and adaptability at the time of their definition. Flexibility is directly related to the definition of variability at a given point of time [Pohl et al., 2005], i.e., the architecture capability of changing according to anticipated changes [Clements et al., 2010]. Adaptability is concerned with variation over time to adapt different conditions and environments [Subramanian & Chung, 2001], i.e., the architecture capability of evolving to unanticipated or unplanned changes. In this regard, this chapter presents a model for the description of flexible and adaptive PLAs: the Flexible Product Line Architecture model, known as Flexible-PLA model. The Flexible-PLA model extends current approaches for modeling architectural variability by providing a wider scope for specifying variability, not only at the level of the external architecture configuration, but also at the level of internal specification of components. To support internal variability, the Flexible-PLA model is based on a novel concept called Plastic Partial Component.

This chapter presents the concepts of internal & external variability and Plastic Partial Component. It describes the formalization of these concepts by means of the Flexible-PLA metamodel, the guidelines to be deployed in a model-driven development framework, its division into views, and finally a discussion about this contribution.
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

5.1 External vs. Internal variability

PLAs consist of a set of reusable building blocks that can be configured to build the products that make up the product-line. From the structural viewpoint these building blocks are components and connectors, which should be designed to be configured to build the products — i.e. they should provide variability. In fact, variability should be one of the most important drivers of these kinds of architectures that support the set of members that make up a product-line. However, the modeling of the architectural variability has become a challenge for SPLE and a recurring issue on numerous workshops and conferences\(^1\) due to the relevance that architectural variability plays in the construction of flexible and adaptive architectures.

Architectural variability modeling enables the maintenance of backward and forward traceability of variability between requirements and architecture and between architecture and detailed design, or even implementation. It also helps understand the configuration of a product (product derivation). Many researchers suggest the idea of explicit description of variability in the architecture [Adachi Barbosa et al., 2011; Bachmann & Bass, 2001; Dashofy & Hoek, 2002; van der Hoek et al., 1999]. Most of them consider essential to integrate variability modeling concepts into architectural models in order to successfully develop SPLs. Architectural variability modeling requires variability concepts to be consistently integrated with the other architectural elements.

Most of the mechanisms for describing architectural variability, proposed by the abovementioned researchers, specify what we have called external variation of the architecture. External variation allows the specification of flexibility points by modifying the structural configuration of the architecture. Specifically, these mechanisms define variability by:

- Adding or removing components and connections by means of optional, alternative and multiple components and connectors [Dashofy & Hoek, 2002; van der Hoek et al., 1999; van Ommering et al., 2000].

5.1 External vs. Internal variability

- Modifying the configuration of composite components, also known as systems, by means of the compositionality mechanisms of software architecture, such as the use of subtyping of components and connectors [Medvidovic et al., 1996] or the use of representations [Adachi Barbosa et al., 2011; Garlan et al., 2000].

However, external variation is not enough to completely define all kinds of variabilities [Bachmann & Bass, 2001] and trace these variabilities from requirements to the architecture [Weiler, 2003]. Sometimes it is not possible to materialize the variable requirements of a family of products simply through additions or removals of components or connectors in the PLA structural configuration. Sometimes variations may happen inside components. As a result, it is necessary to specify internal variation, i.e. variations of simple components—or non-composite components. In these components part of their functionality is common to the SPL and part of their functionality changes depending on the product to be derived according to these internal variations.

Internal variation is especially relevant when describing variability that refers to secondary or supporting concerns with respect to a primary or dominant decomposition. An example of a supporting concern with respect to a dominant decomposition could be availability. This availability may crosscut several components of a PLA and may have variability points (eg. strict or non-strict availability). This could affect many different products or there could even be conflicting quality attributes (e.g. tradeoffs between availability and performance) in different products of the same family. The description of this variability, which is internal to one or many components, is as important as the description of the external variability.

The work of [Bachmann & Bass, 2001] is a reference for variability management in software architectures as it provides a complete support for variability, both internal and external. They laid the basis for addressing internal variability by means of components that realize all possible variations. Their work proposes different techniques such as generators, configuration management systems, compiler switches, and external configuration files, to realize variations of components. In recent years, approaches based on the aspect-oriented software architectures [Pérez et al., 2006] have been proposed that apply aspect-oriented to the specification of the internal variation of components [Adachi Barbosa et al., 2011]. These approaches are based on invasive software composition principles [Assmann, 2003] which define components as fragment boxes that hook a set of reusable fragments of code. Specifically, invasive software composition proposes these fragments of reusable code to be aspects, which makes components easier to be maintained, and by extension software architectures.
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

In summary, the specification of both external and internal variability is essential to completely model variability. In addition, it is critical to select the appropriate mechanism in such a way that variations can be specified without tangling their specification with the core components of PLAs. In other words, variabilities should be independently specified from commonalities.

The Flexible-PLA model that this thesis presents explicitly specifies the commonalities and variabilities of the structural view of PLAs—including both external and internal variations. The Flexible-PLA model is based on the notion of Plastic Partial Component (PPC) [Pérez et al. 2009] which is a solution to specify the internal variation of architectural components. Both of them are presented in the next section.

5.2 The Flexible-PLA Model

This section presents a solution for describing PLAs. This solution, based on models, is called the Flexible-PLA model. Flexible-PLA follows Model-Driven Engineering (MDE [Hutchinson et al. 2011; Schmidt 2006]), specifically Model-Driven Development (MDD [Beydeda et al. 2005]). MDD is a software development approach in which the focus and primary artifacts of development are models—as opposed to programs—and model transformations [Beydeda et al. 2005]. MDE sets out that a model must be described by a well-defined domain-specific (modeling) language (DSL, Taha 2009). Therefore, to be able to specify Flexible-PLA models it is necessary to define a DSL for explicitly specifying the commonalities and variabilities of the structural view of PLAs—including external and internal variations. The next subsections describe (i) the main concept our DSL is based on for describing PLAs, i.e. the concept of Plastic Partial Component, (ii) the DSL abstract syntax through the definition of the Flexible-PLA metamodel, its domain concepts, relationships and rules, and (iii) the DSL concrete syntax by defining a graphical language representation.

5.2.1 Plastic Partial Components (PPC)

The main concept underlying the Flexible-PLA model is the concept of Plastic Partial Component (PPC [Pérez et al. 2009]). A PPC is a kind of component that allows one to specify internal variability; therefore, it is a component such that part of its behavior
5.2 The Flexible-PLA Model

corresponds to the core of a SPL and part of its behavior is specific of a product or set of products from that SPL.

The variability mechanism underlying PPCs is based on the principles of invasive software composition and the combination of two approaches to define software architectures: the Component-Based Software Development (CBSD) [Szyperski, 2002] and the Aspect-Oriented Software Development (AOSD) [Kizcales et al., 1997].

The variability of a PPC is specified using variability points, which hook fragments of code known as variants to the PPC (see Figure 5.1). The specification of a variability point must include the definition of the weavings between the PPC and the variants. The notion of weaving originally comes from Aspect-Oriented Programming (AOP) [Kiczales et al., 2001; Kizcales et al., 1997] and provides the functionality needed to specify where and when to extend components through the use of aspects. Unlike AOP, our weavings specify where and when to extend PPCs through the use of variants. AOP defines a set of weaving primitives: pointcuts and advices, which are applied to weave a PPC with variants. Pointcuts define where the code of a variant is going to be inserted. Specifically, a pointcut is the call to one or a subset of services that a PPC provides. The services of a PPC, which can be intercepted during their execution to insert code, are called services for derivation. The fragment of code to be inserted in a PPC, widely named advice in AOP terminology, is provided by variants. Each variant is composed of several advices, and each advice is modeled as a service for derivation. The definition of the weaving operator consists of establishing when to insert the advice of the variant in regard to a pointcut. It could be before, after or insteadOf the call of the pointcut.

Variants (as well as components and PPCs) realize the features that have been defined at domain analysis. These features of the domain analysis can be related to concerns which crosscut the software architecture (crosscutting-concerns) or not (non-crosscutting-concerns). In the case of crosscutting-concerns, it would be desirable that variants could be easily reused by different PPCs. To that end, variants should be unaware of the linking context. Regarding this issue, it is important to notice that there are some differences between our definition of weavings and variants and the definition of weavings and aspects that AOP provides. By our definition, variants only specify the advice. As a result, our variants do not make reference to the pointcut of the PPC and do not specify the weaving operator (after, before or around) as AOP does. The pointcut and the weaving operator are specified outside the variants. Specifically, they are specified in the weavings, which in turn, are specified in the variability points. As a result, variants are unaware of the linking context, and they are completely reusable.
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

![Figure 5.1: Plastic Partial Component](image)

**Figure 5.1: Plastic Partial Component** - Graphical notation of a PPC with a variability point and two variants

Namely, variants are software pieces that can be reused by any component, even any PLA, as they are context-independent.

Therefore, a PPC is defined by specifying (see Figure 5.1): (i) its variability points, (ii) the variants that are necessary to complete the definition of the component for any software product, (iii) the hooks between the variability points and the variants, and (iv) the weavings that pinpoint when a pointcut—service(s) of a PPC or set of PPCs—is intercepted by the advice of a variant. As a result, a PPC can initially be partially defined, consequently it can be completely defined for a specific product using the selection of variants through the variability points and the weavings the last ones define. This is why we have characterized these components as partial and plastic components:

- PPCs are *partial* because they can be incompletely specified. They are partially described in the product-line architecture and ready to be extended or modified in the product architecture.

- PPCs are *plastic* because they are highly malleable. This means that they are easily configured to specific products from the SPL due to their variability mechanism, which allows one to flexibly adapt software components by easily
5.2 The Flexible-PLA Model

(un-)weaving variants. Hence, they are ready to be extended or modified at any moment.

The PPC component definition makes it possible to flexibly compose pieces of software —components and variants— as if we were building a jigsaw-puzzle. It allows the improvement of flexibility and adaptability, as well as reusability and traceability of software.

- Flexibility & Adaptability. Flexibility of software architectures is given by PPCs and variability points while adaptability is given by the capabilities for changing variants, as they are unaware of the linking context. PPCs reduce dependences and coupling between components and their variants. They also make the easy (un-)weaving of variants possible and cheap.

- Reusability & Traceability. Reusability is improved because PPCs and their variants are context-unaware. Traceability is also improved as traceability of variability (which is internal to one or many components) is supported, in addition to the traceability of the external variability. PPCs enable a fine-grained traceability of variability, i.e. the trace of variants, either backward to requirements or forward to implementation.

5.2.2 Abstract Syntax: Metamodel description

A metamodel is a model of models, hence, it describes models and establishes their properties in a precise way, allowing the verification of those models that are constructed and conformed to them [Object Management Group, 2006]. For this reason, this work defines a metamodel to completely describe the structural viewpoint of PLAs.

The Flexible-PLA metamodel contains a set of inter-related metaclasses (see Figure 5.2). These metaclasses define a set of properties and services for each concept considered in the model. On one hand, metaclasses, their properties and their relationships describe the structure and the information that is necessary to specify the architecture and its variations. On the other hand, the services of metaclasses allow us to develop models by creating, destroying, adding or removing elements which are compliant to the constructors of the metamodel. Those constraints that cannot be defined through the use of relationships and their cardinality are specified by using the Object Constraint Language (OCL [Object Management Group, 2011b]).
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

The common concepts of PLAs are specified, as it is usually done in common ADLs [Medvidovic & Taylor, 1997, 2000]. Hence, components are described in the metamodel by the metaclass **Component**. A component is characterized by a set of properties (see the metaclass **Property** and the aggregation relationship **characterizedBy** in Figure 5.2), and offers a set of services (see the metaclass **ServiceForCore** and the aggregation relationship **offers** in Figure 5.2). A component has a set of ports that publish its services (see the metaclass **Port** and the aggregation relationship **hasPorts** in Figure 5.2). Properties, services and ports are part of the component and do not have their own entity without the component, hence the aggregations between component and {property, service, and port} are inclusive.

Ports publish the services of a component through an interface. This relationship is represented by the aggregation **publishesInterfaces** between the metaclasses **Port** and **Interface** (see Figure 5.2).

Connectors model interactions among components. Although we advocate the approach of considering connectors as first-class entities in software architecture representations [Shaw, 1996], the Flexible-PLA metamodel implements a simplification of connectors as simple attachments. Hence, connectors define the communication channels between the ports of two components. This connection between components is represented in the metamodel by the association relationship **linksport**, which relates the metaclasses **Port** and **Connector** (see Figure 5.2). The metaclass **Connector** has two attributes, **name** and **mandatory**. The attribute name stores the identifier of the connector, whereas the attribute mandatory determines whether the connection is optional, i.e. whether it belongs to the core of the PLA —mandatory value is true—or whether it is specific to a product, or subset or products, from the SPL that is being modeled —mandatory value is false. As a result, a component that has all optional connections, is naturally optional. Therefore, external variability is realized by adding or removing optional connections and components to/from PLAs, i.e. by modifying their structural configuration.

A PPC is a specialization of a component as the metaclass **PlasticPartialComponent** inherits all the properties and behavior from the metaclass **Component** (see Figure 5.2). Internal variations are specified using PPCs, i.e. components that define a set of variability points which hook variants. Hence, a PPC is characterized by the definition of a set of variability points, i.e. the place where the different variants are hooked to one or more PPCs. This relationship is modeled by means of the association relationship called **defines** (see Figure 5.2), which relates the metaclass **PlasticPartialComponent** to the metaclass **VariabilityPoint**.
Figure 5.2: Flexible-PLA metamodel
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

The set of variants that a variability point provides is specified through the association relationship hooks between the metaclasses VariabilityPoint and Variant (see Figure 5.2). The metaclass Variant defines an attribute to name a variant. The metaclass VariabilityPoint defines two attributes that permit the naming of a variability point and the specification of the kind of variation. The kind of variation refers to the cardinality of the variability point (see the attribute cardinalitySelection in Figure 5.2). Its value is based on the variability management of software architectures that Bachmann & Bass [2001] described, which is summarized in Table 5.1.

Table 5.1: Plastic Partial Components Cardinality

<table>
<thead>
<tr>
<th>Type of Cardinality</th>
<th>Min</th>
<th>Max</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional and unique and unique</td>
<td>0</td>
<td>1</td>
<td>0 or 1 among n variations</td>
<td>When a product is applied to the product line, it is optional to select a variant of the variability point.</td>
</tr>
<tr>
<td>optional and multiple</td>
<td>0</td>
<td>n</td>
<td>0 or 1 or m among n variations (m ≤ n)</td>
<td>When a product is applied to the product line, it is optional to select a set of variants from the variability point.</td>
</tr>
<tr>
<td>mandatory and unique - simple</td>
<td>1</td>
<td>1</td>
<td>only 1 among n variations</td>
<td>When a product is applied to the product line, it is mandatory to select a variant of the variability point.</td>
</tr>
<tr>
<td>mandatory and unique - multiple</td>
<td>1</td>
<td>n</td>
<td>1 or m among n variations (m ≤ n)</td>
<td>When a product is applied to the product line, it is mandatory to select at least one variant of the variability point.</td>
</tr>
<tr>
<td>mandatory and multiple - multiple</td>
<td>m</td>
<td>n</td>
<td>m among n variations (m ≤ n)</td>
<td>When a product is applied to the product line, it is mandatory to select at least m variants of the variability point.</td>
</tr>
</tbody>
</table>

Finally, variability points are characterized by the weaving that pinpoints where and when to extends PPCs through the use of variants. The weaving is defined by the metaclass Weaving. The metaclass Weaving is part of the variability point as it specifies the weaving between PPCs and variants, and it is dependent on the linking context. As a result, a variability point must specify all the weavings that can be applied to the PPC(s) that define it and its variants. This relationship is defined in the metamodel through the aggregation relationship weaves, which is inclusive. The metaclass Weaving has three attributes: name, (weaving) operator, and selection. The first one allows the identification of the weaving, whereas the second one establishes
5.2 The Flexible-PLA Model

when to insert the advice of the variant in regard to a pointcut: before, after, instead (see Figure 5.2). The attribute selection determines if a weaving has to be applied to or not. This means, at the time of deriving or configuring a specific product from a SPL, this attribute determines which pointcuts —service(s) of a PPC or set of PPCs— are intercepted by the advice of a variant. The pointcut and the advice are represented by the metaclass ServiceForDerivation. This is why the metaclasses PlasticPartialComponent and Variant are composed by ServicesForDerivation (see the aggregation relationships composedof and constitutedby in Figure 5.2). They are the services that participate in the weaving to specify where to insert the code of the advice in the pointcut (see the aggregation relationships pointcut and advice in Figure 5.2). The metaclass ServicesForDerivation is a specialization of services that inherits all the properties and behavior of a common service of the metaclass Service (see Figure 5.2). They are a specialization because they are services that participate in a variability point. Hence, common components (non-PPCs) cannot be composed of this kinds of services, only PPCs and variants can be composed of services that participate in a variability point, i.e. ServicesForDerivation.

As has been mentioned above, those constraints that cannot be defined through the use of these relationships and their cardinality have been specified by using OCL. These constraints are described below and their OCL definition is presented in Appendix A.

**Constraint 1** All services which are published by an interface through the ports of a component, must be offered by this component.

**Constraint 2** An interface can be published by more than one port, but all ports must belong to the same component.

**Constraint 3** A ServiceForDerivation must belong to a PPC or to Variant (XOR).

**Constraint 4** In the relationship advice between Weaving and ServiceForDerivation, the ServiceForDerivation must belong to a Variant.

**Constraint 5** In the relationship pointcut between Weaving and ServiceForDerivation, the ServiceForDerivation must belong to a PPC.

**Constraint 6** Every ServiceForDerivation must have at least one relationship as advice or pointcut.
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

5.2.3 Concrete Syntax: Graphical language description

A graphical modeling language is defined as this kind of language is usually more intuitive. This language has not been defined from scratch: the common graphical metaphors of components [Object Management Group, 2009] as well as variability points [Pohl et al., 2005] have been reused. Its usage has been pursued to be as friendly as possible for use in the PLA community. Figure 5.3 describes the main concepts and their graphical representations.

![Figure 5.3: Flexible-PLA graphical modeling language](image)

Figure 5.3: Flexible-PLA graphical modeling language - Graphical modeling language used by FPLA modeling framework

Components are represented by blue rectangles that have one pin for each of the ports associated to them. PPCs are represented as components, with the only difference being the two white triangles on both sides to denote that they are not completely specified. Connectors are represented by a line that connects two ports. Interfaces are represented by yellow rectangles that specify their name and the list of services that they publish. Variability points (VP) are represented as by [Pohl et al., 2005], so that a common notation is possible. The Weavings are presented by a rectangle that specifies its name and operator. This rectangle is differentiated from the rest as it is drawn using dashed lines. Variants are represented by horizontal rectangles. Finally, we distinguish between the graphical notation of ServicesForCore and ServicesForDerivation.

Figures 5.4 and 5.5 show several snapshots of the FPLA modeling framework where the Flexible-PLA graphical language is illustrated by using the pilot study of the SPL for banking systems (see Chapter 4). Specifically, Figure 5.4 shows the PPC
ATMFrontend which implements availability in its two (mutually exclusive) alternative variations, strict and non-strict. This variability is internal to the component specification of the front end, i.e. the ATMFrontend has a common part which is independent from the banking system application to be derived, and a variable part which is specific to the banking system application to be derived. This variable part refers to the two abovementioned variants of the availability concern. Hence, the PPC ATMFrontend defines two variability points: RequestManaging and Updating (see Figure 5.4). RequestManaging defines the variability regarding the management of requests from ATMs and it hooks the variants LoadBalancing and Routing for strict and non-strict availability, respectively (see Figure 5.4). Updating defines the variability regarding the updating process of the nodes (active or redundant spare) and it hooks the variants Synchronization and DataMonitoring (see Figure 5.4).

In order to pinpoint where and when to extend the PPC ATMFrontend through the use of variants, the variability points RequestManaging and Updating have to define the weavings. We have defined four weavings (see Figure 5.4). These weavings define where and when the code of each one of the four variants is injected. Figure 5.5 details the pointcut and advices definition for two weavings of the PPC ATMFrontend. The pointcut is represented by the service (for derivation) processRequest (see Figure 5.5). The service ProcessRequest is intercepted by the advices balance and routeIP, which manage the ATM requests in order that (i) active and redundant nodes process identical inputs, or (ii) only the active node processes all the inputs, respectively (see Figure 5.5).
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

Figure 5.5: Flexible-PLA graphical modeling language: Banking systems PLA
- The figure shows the weaving definition of a PPC

5.3 Flexible-PLA Model in MDE

The OMG Meta-Object Facility (MOF) 2.0. specification \[Object Management Group, 2006\] defines an architecture to support meta-modeling and MDD. Its main purpose is the management of model descriptions at different levels of abstraction. The MOF 2.0. architecture defines a number of layers greater than or equal to two. The two layers are established as it is the minimum requirement to represent and navigate from a class to its instances and vice versa (layers M0-instances and M1-classes). If we also introduce metamodels in the MOF architecture, the minimum number of layers needed is four (layers M0-M3). The well-known four-layered metamodel architecture of MOF 2.0., can be described as:

- The layer M3 (meta-metamodel layer) defines the abstract language used to describe the entities of the lower layer (metamodels). The MOF specification proposes the MOF language as the abstract language for defining all types of metamodels, such as the metamodel of UML.

- The layer M2 (metamodel layer) specifies the structure and semantics of the models defined at the lower layer.

- The layer M1 (model layer) comprises the models that describe data of the lower layer. These models are described using the primitives and relationships described in the metamodel layer (M2).

- The layer M0 (information layer) comprises the instances of the models that are defined at the model layer (M1).

MOF 2.0 with its reflection model (instance-class relationships) can be recursively applied to handle any layers (sometimes referred to as metalevels) as modelers (analyst, architect, programmer, etc) define. Specifically, the model and language that this thesis
defines to describe PLAs is based on a four-layered architecture which uses MOF 2.0. and UML 2.0. to specify its metamodel. This four-layered architecture is described as follows (see Figure 5.6).

![Figure 5.6: The Flexible-PLA metamodel in MDD](image)

The Flexible-PLA metamodel is defined at the layer M2 of the MOF four-layer architecture.

The layer M3 defines the abstract language MOF which is used to describe the UML metamodel. This in turn is used to describe the entities of the lower layers. This layer closes the reflection structure of MOF 2.0. (see layer M3, Figure 5.6). The metamodel used to describe PLAs, the Flexible-PLA metamodel, is defined at layer M2 (see layer M2, Figure 5.6). The layer M1 comprises the Flexible-PLA models which describe types (classes) conforming to the Flexible-PLA metamodel (see layer M1, Figure 5.6). These models describe data of the lower layer. Finally, the lowest level comprises the instances of the models that are defined at the layer M1 (see layer M0, Figure 5.6).

Since this work is based on MOF and UML, any architectural model with a metamodel specification at the layer M2 can easily introduce the Plastic Partial Component concept by extending its metamodel with the Flexible-PLA metamodel. This task could be automated by model transformations using any model management tool or model...
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

transformation language, such as medini QVT [Object Management Group, 2011a] and ATL [ATL, 2011].

5.4 The Flexible-PLA 4-view model

The description of software architectures can be done from different viewpoints. The Flexible-PLA metamodel focuses on the description of the structural viewpoint of software architectures, which is based on the works of Perry & Wolf [1992] and Garlan & Shaw [1993] and proposed by the IEEE Std 1471-2000 [2007].

The structural viewpoint requires the preservation of the descriptions of software system structures by hiding the low-level details, abstracting the important high level features, and providing the black box view of architectural elements [Perry & Wolf, 1992]. In the specific case of PLA descriptions, architectural structure description should not be mixed with variability specifications. In order to guarantee these properties, different views have been defined for specifying architectural structure and variability. Additionally, PLA construction must meet the SPLE processes at architectural level (see Chapter 2.3), i.e. PLA models must be described by preserving the Domain Engineering and Application Engineering processes and their main tasks. On one hand, the Domain Engineering process requires us to: (i) specify the core architecture and (ii) define the variation of all the products of the family. On the other hand, the Application Engineering process requires us to: (iii) derive the product by selecting the corresponding variations and (iv) obtain the final product.

This is the reason Flexible-PLA metamodel is built on a 4-view model which refines and decomposes the structural viewpoint, proposed by the IEEE Standard 1471 [IEEE Std 1471-2000, 2007], into views in order to support the variability description that PLA definition requires. This model defines 4 conceptual views: Core, Variability, Derivation and Product.

The Core View describes the mandatory components and connections, that is, the (invariable and common) core components.

The Variability View comprises (i) optional components and connections, and (ii) the description of PPCs, variability points, weavings, and variants.
The Derivation View allows architects to (i) reconfigure the core architecture by adding or removing the optional components and connectors; (ii) to complete the partial specification of PPCs by weaving variants with the core functionality of the PPC(s). The first task consists of changing the attribute mandatory to the value true in those connectors that are optional for the SPL, but mandatory for the product that the architect wants to derive (see Figure 5.2). The second task consists of selecting the weavings of each variability point that are required by the product, i.e. changing the attribute selection to the value true in the selected weaving for the product (see Figure 5.2).

Finally, the Product View shows the architecture of a specific product as a result of applying the reconfigurations and selections of the Derivation View. As a result, the product architecture is an architecture without PPCs, i.e. the architecture is constituted by components that are completely specified and do not have variation points associated to them. Hence, the product architecture is unaware of the variability specification.

### 5.4.1 Exemplification

The FPLA modeling framework conforms to this 4-view model. Using these views and several snapshots from FPLA, the Flexible-PLA model is exemplified by using the pilot study of the SPL for banking systems (see Chapter 4). Figure 5.7 shows the first step in defining its PLA. It consists of defining the Core View, which defines the components that realize the common banking transactions. These components are Balance, Withdrawal, Deposit, Transfer, and TicketIssuer (see Figure 5.7). The services provided by these components are offered to other systems through two different front ends for managing the requests from ATMs and WEBApps.

The feature availability 24/7—that most common banking transactions must provide—refers to a secondary or supporting concern in respect to the primary/dominant functional decomposition. This feature defines a variability point in such a way that banking systems can offer their core banking transactions upon strict availability or non-strict availability. The two front ends are the components in charge of implementing availability in its two (mutually exclusive) alternative variations, strict and non-strict. Therefore, this variability is internal to the component specification of the two front ends. That is why the two front ends are specified by using PPCs: ATMFrontend and WEBFrontend (see Figure 5.7). Both of them have a common part which is independent
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

Figure 5.7: Flexible-PLA model - Core View of banking systems PLA - This figure shows the core view for the pilot study that was described in Chapter 4 from the banking system application to be derived, and a variable part which is specific to the banking system application to be derived. This variable part refers to the two abovementioned variants of the availability concern.

To implement strict and non-strict availability, the architectural tactics active redundancy and passive redundancy have been selected, respectively, as it is explained in Chapter 4. To implement these two tactics, the PPCs ATMFrontend and WEBFrontend define two variability points: RequestManaging and Updating. Figure 5.8 shows the variability view in which these variability points are modeled. RequestManaging defines the variability regarding the management of requests from ATMs and WEBApps; namely, it implements the variants LoadBalancing and Routing for strict and non-strict availability, respectively (see Figure 5.8). Updating defines the variability regarding the updating process of the nodes (active or redundant spare); namely, it implements the variants Synchronization and DataMonitoring (see Figure 5.8). Therefore, strict availability (active redundancy) is supported by the Synchronization and LoadBalancing variants, and non-strict availability (passive redundancy) is supported by the Routing and DataMonitoring variants.

In order to pinpoint where and when to extend PPCs through the use of these variants, the variability points RequestManaging and Updating have to define the weavings. Eight weavings are defined (see Figure 5.8). These weavings define where
5.4 The Flexible-PLA 4-view model

and when the code of each one of the four variants is injected within the code of the PPCs ATMFrontend and WEBFrontend. Hence, Figure 5.9 details the pointcuts and advices definition for each one of the four weavings of the PPC ATMFrontend. The pointcuts are represented by the ServicesForDerivation: processRequest, processFault, and updateStateDaemon (see Figure 5.9). The service processRequest is intercepted by the advices balance and routeIP which manage the ATM and WebApp requests in order that (i) active and redundant nodes process identical inputs, or (ii) only the active node processes all the inputs, respectively (see Figure 5.9). The service processFault is intercepted by the advice routeIP which changes the route of requests to the redundant node(s) when the active node fails (see Figure 5.9). Finally, the service updateStateDaemon is intercepted by the advices synchronizeState and periodicUpdateState in order to (i) active and redundant nodes maintain identical state at all time, or (ii) with periodic state updates, respectively (see Figure 5.9).

5.4.2 Considerations

Applying the PPC variability mechanism to the scenario of the banking system SPL, we are able to glimpse several of the attributes and advantages that Flexible-PLA model provides to the PLA design.
Firstly, the flexibility attribute of software architectures is given by the primitives to specify both kinds of variability: external variability by using optional connections, and internal variability by using PPCs to specify their variability points and variants. The pilot study focused on implementing the variability of the feature availability 24/7 in its two variations: strict and non-strict. To define these two variations, the Flexible-PLA metamodel provides the primitives to define: the PPCs \textit{ATMFrontend} and \textit{WEBFrontend}, the variability points \textit{Updating} and \textit{RequestManaging}, and the variants \textit{Synchronization}, \textit{LoadBalancing}, \textit{Routing} and \textit{DataMonitoring}. At the time of deriving specific product applications from the banking system SPL, this flexibility allows the implementation of one of the variations strict or non-strict availability. This happens through the selection of the corresponding variants.

Secondly, the adaptability attribute of software architectures is given by the capability for changing variants, as they are unaware of the linking context. The use of PPCs in this pilot study facilitates the addition of new architectural tactics to realize availability, such as \textit{sparse} or \textit{exception handling} \cite{ScottKazman2009}. It also facilitates their removal and the replacement of some tactics by others, thanks to the primitives for (un-)weaving variants. Namely, the architecture is adaptable by adding, removing or replacing the variants regarding availability tactics to/from/in
the variability points *Updating* and *RequestManaging*. This happens through the simple reconfiguration of the weavings and without affecting the code of the PPCs *ATMFrontend* and *WEBFrontend*.

Finally, the code of the four abovementioned variants —*Synchronization*, *LoadBalancing*, *Routing* and *DataMonitoring*— is not scattered through the two front ends that need to implement availability 24/7 (or other components that could need it), it is modularized, which might make it easier to further changes in that code. In addition, the code of these four variants can be reused by both front end services, *ATMFrontend* and *WEBFrontend*. This is possible because the variants of the Flexible-PLA metamodel are unaware of the linking context. In fact, AOP is a solution that avoids the *scattering* of crosscutting concerns throughout many components, as well as the *tangling* of the core with crosscutting concerns in each component [Laddad, 2010]. As in AOP, the PPC variability mechanism is a solution that avoids the *scattering* of variants throughout many components, as well as the *tangling* of the core and variants (either crosscutting or non-crosscutting concerns) in each component. It is a fact that code scattering and code tangling make traceability, reusability and evolution difficult [Laddad, 2010]. Alternatively, Figure 5.10 shows how a banking system SPL implements strict and non-strict availability using only external variability conventional techniques, such as optional connectors. Even when using well-designed components that implement availability and that offer well-defined interfaces (see the components *StrictAvailabilty* and *Non-strictAvailability* in Figure 5.10), each client of these interfaces (see the components *ATMfrontend* and *WEBfrontend* in Figure 5.10) still needs the code to invoke the interfaces in order to support availability and fault recovery. This code may be scattered across multiple components, and there is no single place to identify the availability concern. The overall effect is an undesired tangling between the components which need to support availability and fault recovery and the components which implement the active and passive redundancy. This issue is also documented in [Laddad, 2010] for the security concern. Additionally, a final consideration is that several concerns cannot be designed as components. This is due to the fine-granularity of their functionality, which may be closer to a service than a component. Therefore, PPCs and the AOP underlying mechanism minimize code scattering and code tangling, which in turn enable traceability, scalability, reusability and evolution.

In conclusion, this pilot study uncovers that although adding these variability points and their respective variants implies more effort in specifying their weavings, we gain in flexibility and adaptability, code modularity, and in turn, traceability, scalability, reusability and maintainability.
5. FLEXIBLE PRODUCT LINE ARCHITECTURES

Figure 5.10: Excerpt from Figure 5.7 without using the PPC variability mechanism - This figure shows a banking system SPL that implements strict and non-strict availability, using only external variability conventional techniques. The overall effect is an undesired tangling between the components which need to support availability and the components which implement the strict and non-strict redundancy.

5.5 Discussion

Over the last few years, a wide variety of ADLs, models and UML extensions have been proposed to describe PLAs (see Table 5.2). They can be differentiated by the following criteria: (i) the main architectural concepts that are supported as first-class entities in software architecture representation, such as components, connectors, etc.; (ii) support to explicitly specify external and internal variability; (iii) the formalism used to describe PLAs; (iv) the architectural views where PLA models are broken down; (v) whether the formalism provides forward or backward traceability to requirements or implementation, respectively; and finally (vi) the tool that supports the approach. These criteria can be used to compare the different approaches and to identify what the lacks are in the research topic of PLA descriptions.

We use a tabular format to compare related work in chronological order of publication, this way the two first columns identify the approach. If an approach has a name, we have used that name and its reference, otherwise we only use its reference. If a work does not provide any evidence to fill a column we use the - mark.

As Table 5.2 shows, most approaches account for external variability. However, the focus of this comparison is on internal variability. First, this review starts by
analyzing those approaches which are based on the compositionality mechanisms of software architectures (see rows 1,3,14,15 in Table 5.2).

The ADLs C2SADEL [Medvidovic et al., 1996] and ACME [Garlan et al., 2000] (see rows 1 and 3 in Table 5.2) are based on compositional specification of components through the use of subtyping of (components and connectors) types and refinement of components, and the use of representation, respectively. Although neither of them offer full support to explicitly describe optionality nor variability, these works are included in this comparison as the techniques in which they are based on, have been reused by other works to specify variability. This is the case of PL-Aspectual ACME [Adachi Barbosa et al., 2011; Batista et al., 2008] that uses the representation concept to define multiple representations for a given component (see row 14 in Table 5.2). The Koala Component Model [van Ommering et al., 2000] (see row 4 in Table 5.2) is also based on the compositionality mechanism of software architectures. Hence, it supports the specification of component variability through optional interfaces and hierarchical specification of its subcomponents; then selection between subcomponent variants is realized by variation points called switches. Finally, VEIA [Mann, 2009] (see row 15 in Table 5.2) also proposes modeling variability by means of a compositional specification of components and a definition of ports variability.

Although the abovementioned approaches enable the specification of internal variability of composite components, neither of these approaches can offer full support for the internal variability of simple or non-composite components. This is why Table 5.2 shows the mark $\times$. Namely, the approaches which are solely based on compositionality mechanisms cannot offer the primitives to specify the internal variability of non-composite components. The Flexible-PLA metamodel that this chapter presents (see row 16 in Table 5.2) provides the primitives to specify the internal variability of non-composite components by using PPCs.
<table>
<thead>
<tr>
<th>Approach</th>
<th>Architectural concepts</th>
<th>Explicit Specification of Variability into PLA</th>
<th>Formalism</th>
<th>Architectural view</th>
<th>Traceability</th>
<th>Supporting Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[Oreizy et al., 1998]</td>
<td>C2 architectural concepts (C2 architectural style)</td>
<td>C2SADEL</td>
<td>C&amp;C view</td>
<td>ArchStudio → Java, DRADEL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Medvidovic et al., 1999]</td>
<td>Compositional specification of components: heterogeneous subtyping to evolve components although neither optionality nor variability is addressed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[van der Hoek et al., 1999, 2001a]</td>
<td>Components, connectors, ports, configuration, representation, systems, and families</td>
<td>—</td>
<td>Language-independent extension for describing architectures</td>
<td>—</td>
<td>Mae (Ménage + DRADEL)</td>
</tr>
<tr>
<td>3</td>
<td>ACME [Garlan et al., 2000]</td>
<td>Compositional specification of components: use of representations and its “extends” feature although neither optionality nor variability is addressed</td>
<td>ACME</td>
<td>C&amp;C view</td>
<td>—</td>
<td>ACME Studio</td>
</tr>
<tr>
<td>4</td>
<td>Koala [van Ommering et al., 2001]</td>
<td>Component, connections, interfaces, configuration</td>
<td>Koala</td>
<td>Component Model and Language</td>
<td>Logical view</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>dashofy &amp; Hook, 2002; Dashofy et al., 2005</td>
<td>Components, connectors and interfaces</td>
<td>xADL</td>
<td>C&amp;C view</td>
<td>—</td>
<td>xArch/xADL 2.0</td>
</tr>
<tr>
<td>7</td>
<td>QUASAR [Thiel &amp; Hein 2002]</td>
<td>Variability points</td>
<td>—</td>
<td>UML extension</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>Variation Point Model [Webber &amp; Gomaa, 2002]</td>
<td>Components, Interfaces and Variants</td>
<td>—</td>
<td>UML extension</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Table</td>
<td>Document</td>
<td>Feature Components</td>
<td>Definition, Refinement, Redefinition and Extension</td>
<td>UML Annotation</td>
<td>Invasive Software Composition: AOP</td>
<td>Invasive Software Composition: Metamodel</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>--------------------</td>
<td>---------------------------------</td>
<td>---------------</td>
<td>-----------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>9</td>
<td>[Weiler, 2003]</td>
<td>Common feature components</td>
<td>Variable feature components and specific feature components</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>AOP extension for SPL [Lee et al., 2006]</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Invasive Software Composition: AOP</td>
<td>Invasive Software Composition: AOP</td>
</tr>
<tr>
<td>11</td>
<td>Elastic Components [Karakontzas et al., 2007]</td>
<td>Components, interfaces and elastic components</td>
<td>—</td>
<td>—</td>
<td>Invasive Software Composition: AOP</td>
<td>Metamodel</td>
</tr>
<tr>
<td>12</td>
<td>[Razavian &amp; Khosravi, 2008]</td>
<td>Components, connector, and interfaces</td>
<td>Optional or alternative architectural elements</td>
<td>Inheritance</td>
<td>UML Profile C&amp;C view</td>
<td>—</td>
</tr>
<tr>
<td>13</td>
<td>Aspect-Oriented Model [Noda &amp; Kishi, 2008]</td>
<td>Aspects, optional aspects and alternative aspects</td>
<td>—</td>
<td>—</td>
<td>Invasive Software Composition: AOSA</td>
<td>Everything is an aspect</td>
</tr>
<tr>
<td>15</td>
<td>VEIA [Mann, 2009]</td>
<td>Component variation point (Mandatory and optional) connectors and ports</td>
<td>Component variation point (Mandatory and optional) connectors and ports</td>
<td>—</td>
<td>Language-independent extension for describing PLAs</td>
<td>Minimal view</td>
</tr>
<tr>
<td>16</td>
<td>Flexible-PLA [Perez et al., 2009]</td>
<td>Components, connectors, interfaces, ports, etc. Plastic Partial Components, VP and variants</td>
<td>Optional connectors</td>
<td>—</td>
<td>Invasive Software Composition: AOSA</td>
<td>FPLA Metamodel (language-independent extension for describing PLAs)</td>
</tr>
</tbody>
</table>

ADL: Architecture Description Language
AOP: Aspect-oriented programming
AOSA: Aspect-oriented software architecture
PLAs: Product-Line Architectures
C&C: Component and connector
In addition to the compositionality mechanisms, the abovementioned approaches and others propose the use of specific concepts to explicitly describe variability points within ADLs and/or architectural models. Hence, xADL [Dashofy & Hoek, 2002], Koala Component Model [van Ommering et al., 2000] and PL-AspectualACME [Adachi Barbosa et al., 2011] offer extensions for describing PLAs by means of (i) option and variant extensions, (ii) optional interfaces, and (iii) aspectual connectors (see rows 4, 5, 14 in Table 5.2, respectively). Other models which are not ADLs themselves, otherwise known as language-independent extensions, also offer modeling primitives to describe architectural variability. Hence, several of these models propose variant components, variant connectors and multi-versioning connectors [van der Hoek et al., 1999, 2001], elastic components [Kakarontzas et al., 2007], or port variability [Mann, 2009] (see rows 2, 11, 15 in Table 5.2, respectively). The Flexible-PLA metamodel that this chapter presents (see row 16 in Table 5.2) defines PPCs, variability points and variants to explicitly describe variability.

Internal variability is mostly addressed by using UML compositionality and inheritance mechanisms, such as the approaches of Bachmann & Bass [2001], Razavian & Khosravi [2008], Webber & Gomaa [2002], Weiler [2003] (see rows 5, 8, 9, 12 in Table 5.2). The work of Bachmann & Bass is a reference for variability management in software architectures. In fact, our approach is based on their work in order to define the kinds of variation. Their work identifies two types of variation in architectures: variation of components and variation of connections. Hence, a component can be optional, a component can have a set of alternative implementations, or variations can occur because of connections between components. Bachmann & Bass laid the basis for addressing internal variability by means of a root component, from which hangs components that implement the variation. The works by Razavian & Khosravi [2008], Webber & Gomaa [2002], Weiler [2003] extend this contribution and formalize it by using UML profiles. Hence, Weiler introduces the concept of incomplete specification to implement variability within components. However, all these abovementioned approaches use inheritance and aggregation patterns to establish relationships between components and their variants. Therefore, this notation is closer to class diagrams rather than to architectural descriptions. Additionally, the use of inheritance to support internal variability makes reusability of variants difficult and hinders the flexibility to evolve PLAs. Hence, the reuse of a variant by other components implies (i) replicating the variant, which consequently might create a maintenance issue, or (ii) using multiple inheritance, which may lead to ambiguity. The Flexible-PLA metamodel that this chapter presents (see row 16 in Table 5.2) defines variants which are completely reusable as they are independent of the linking context. Hence, PPCs are invasively composed of variants which only provide the behavior to be inserted in the PPC, i.e. they do not
5.5 Discussion

specify the pointcuts and the weaving operator as AOP does. The pointcut and the weaving operator are specified outside the variants, and therefore variants are unaware of the linking context. As a result, variants are completely reusable, and architectures more flexible.

There is a growing number of approaches that combine Aspect-Oriented Programming (AOP) and SPLs [Kakarontzas et al., 2007; Lee et al., 2006; Noda & Kishi, 2008] (see rows 10,11,13 in Table 5.2). The Aspect-Oriented Modeling (AOM) [Noda & Kishi, 2008] proposes a PL-ADL where everything is an aspect, rejecting the main concepts of formal ADLs. This has the inconvenience that the semantics of aspects and components is not preserved. Kakarontzas et al. [2007] propose a similar approach to address internal variability by adding or deleting variants that hang from elastic components. They use AOP with pointcuts defined by methods of the component interface. PL-AspectualACME [Adachi Barbosa et al., 2011] is an ADL used for describing PLAs that extends AspectualACME [Batista et al., 2006]. It uses the ACME representation concept to define multiple representations for a given element which can be selected by means of aspectual connectors. As its variability mechanism is solely based on the compositionality mechanism of software architecture, it only defines variability of composite components —thus, it does not offer full support for internal variability of non-composite components. Finally, a work that is very close to the notion of Plastic Partial Component is proposed by Lee et al. [2006]. They implement variable features that crosscut several modular units by using aspects. These aspects modify the internal behavior of components following invasive composition. This work is defined at the implementation level using the programming languages AspectJ [Kiczales et al., 2001]. However, the use of AspectJ makes their aspects dependent on the linking context (i.e. the component), and thus, they are not reusable. The Flexible-PLA metamodel that this chapter presents (see row 16 in Table 5.2) also defines variants by using invasive composition, but at an architecture-level. These variants specify their pointcuts and weaving operators outside the variants, thus they are reusable. AspectJ code is generated from Flexible-PLA models as a result of a model-to-code transformation.

Finally, only three approaches frame their models into views (see rows 7,8,15 in Table 5.2), although none of them have specific architectural views to separate core architectural elements from variable architectural elements and to differentiate domain and application engineering processes. The Flexible-PLA metamodel that this chapter presents (see row 16 in Table 5.2) defines the Core and Variability view used to avoid tangling variability specifications with software architecture descriptions. These
two views are related to the domain engineering process, whereas the Derivation and Product views are related to the application engineering process.

In summary, our contribution addresses these gaps with the following key solutions:

- The PPC variability mechanism provides the primitives to specify internal variability of simple—or non-composite—components.
- The PPC variability mechanism hooks variants which are unaware of the linking context, and therefore they are completely reusable. Thus, we avoid the disadvantages of using inheritance mechanisms.
- The Flexible-PLA metamodel is built on 4 views, which avoids tangling variability specifications with software architecture descriptions, as well as domain engineering tasks with application engineering tasks.
- The Flexible-PLA metamodel is supported by the FPLA modeling framework. It provides the modeling primitives for the definition of PLA models which conform to the Flexible-PLA metamodel, as well the model-to-code transformations. Therefore, our proposal is ready to be involved in a MDD process to transform its outputs into platform dependent models and to automatically generate code.

5.6 Conclusions

This chapter presented mechanism to describe PLAs. The Flexible-PLA model integrates CBSD and AOSD to extend the traditional notion of software architecture with the concepts of Plastic Partial Components (PPCs), which provide modeling primitives to capture internal variability of components. The most important contribution of the Flexible-PLA model is a precise representation for capturing variability as an integral part of software architectures. The Flexible-PLA model doest not tie to a single ADL. Instead, it enables the capture of the external and internal variability of components for a multitude of ADLs, as it is based on the common architectural concepts of components, connectors, interfaces, etc. We have adapted and borrowed a variety of concepts that have been established in software architecture research for quite some time now. Still, the Flexible-PLA model differs considerably from most architecture modeling languages defined to date.
5.6 Conclusions

The main characteristics that the PPC variability mechanism provides are: flexibility, adaptability, reusability and traceability. Flexibility of software architectures is given by PPCs and variability points, while adaptability is given by the capabilities of changing variants as they are unaware of the linking context. PPCs reduce dependences and coupling between components and their variants, and enable the easy (un)-weaving of features to be possible and cheap.

Based on the full support for variability that the Flexible-PLA model offers, and the characteristics that the PPC variability mechanism provides, agile architects can take advantage of variability mechanisms to flexibly adapt software architectures and to incrementally develop them. Namely, the Flexible-PLA metamodel offers primitives for agile architecting. Specifically, agile architects can take advantage of the PPC primitives for incrementally and iteratively refine and evolve architectures.

The contribution of this chapter has been published in:

5. FLEXIBLE PRODUCT LINE ARCHITECTURES
Chapter 6

Product Line Architectural Knowledge

This chapter presents a mechanism for documenting and tracing PLAs. Its main purpose is to deal with objective 2 (OBJ 2) defined in Section 1.3, i.e., help preserve the integrity of the architecture. Capturing architectural knowledge allows software architects to revisit important design decisions over time, whereas tracing architecturally significant features with their realization in the PLA assists software architects in understanding the relationships and dependencies between features and PLAs. In this regard, this chapter presents a model for documenting and tracing PLAs: the Product-Line Architectural Knowledge, known as PLAK model. The PLAK model is based on the Flexible-PLA model to document flexible and adaptive PLAs.

This chapter presents the concepts of variability design rationale and product-line architectural knowledge. It describes the formalization of these concepts by means of the PLAK metamodel, the guidelines to be deployed in a model-driven development framework, the PLAK 4+1 views, and finally a discussion about this contribution.
6. PRODUCT LINE ARCHITECTURAL KNOWLEDGE

6.1 Variability Design Rationale

There has been clear progress in architecture documentation and the use of design rationale for reasoning about the architecture and its evolution. However, the documentation of PLAs is still a challenge. Documenting the knowledge of PLA requires not only documenting the design decisions and rationale behind the commonality, but also the design decisions and rationale behind the variability, also known as variability design rationale. As a consequence, documenting the knowledge of PLA requires new kinds of design decisions and new critical dependencies to manage the variability across the multiple products supported by the SPL.

Although documenting the knowledge of PLA is more complex than documenting (single-product) architectures, it is also essential, not only to support the rationalization of design decisions made and the assurance of architecture integrity during evolution, but also to configure valid products according to architectural dependencies between product variants. As many other authors assert, the documentation of architectural design rationale has a high strategic value and deep impact on the success of SPLs [Bosch & Ran, 2000; Knodel & Muthing, 2006; Mohan & Ramesh, 2007]. However, common approaches to document AK do not specifically recognize the concepts of variability point and variant, which are specific to SPL engineering. The documentation of architectural design rationale related to a certain variability point or variant has received very little attention so far, and a few approaches partially address variability design rationale [Capilla & Ali Babar, 2008; Clements et al., 2010; Galvão et al., 2010]. A complete approach for documenting knowledge of PLA must be provided. Its aim should be to support the rationalization, evolution and configuration of product applications, and to provide complete expressiveness for (i) variability design rationale and its dependencies, including both the external variability of the architecture configuration and the internal variability of components, and (ii) traceability between requirements and PLA descriptions.

This chapter presents the notion that we have coined as Product-Line Architectural Knowledge (PLAK) which aims to support: (i) variability design rationale, as well as the architectural variability dependencies, and (ii) traceability links between requirements and PLA descriptions. The notion of PLAK must be codified in a formal way to support reasoning over the space of architectural knowledge. Models provide this formality. This formalization has been performed using models: The PLAK model provides the modeling primitives to represent and document the new types of design decisions that support variability design rationale, the dependencies between these design decisions and their rationale, as well as the capability of tracing variability. In fact, regarding
traceability, we present a decision-centric approach as Mirakhorli & Cleland-Huang [2011] do, in which traceability links are created around the design decisions to trace requirements with PLA descriptions. In this regard, the PLAK model has been built based on (i) the feature modeling (see Section 2.3.3.2) approach for modeling variability of requirements, and (ii) the Flexible-PLA modeling (see Chapter 5) for modeling both variability inside architectural components and variability in the architecture configuration. This is especially relevant as it is just as important to document the architectural knowledge of external variability of the architecture configuration as it is to document the internal variability of components. Finally, the PLAK model has also been built based on existing models from the design rationale management research Babar et al. [2006] Capilla et al. 2007 Kruchten et al. 2006a Tang et al. 2007a.

6.2 The PLAK Model

This section presents a solution for documenting and tracing PLAs. This solution, based on models, is called the PLAK model. It follows MDE, and is specifically framed into the emerging paradigm known as Model-Driven Software Evolution (MoDSE Deridder et al. 2011), which pursues to apply the main advantages of model-driven development, such as abstraction and automation, to evolving existing systems as well. MoDSE aims to (semi-)automatically support software evolution through the use of models and traceability between models. To be able to specify PLAK models is necessary to define a domain-specific (modeling) language for explicitly documenting the design decisions, as well as dependencies and rationale of commonalities and variabilities of PLAs—including external and internal variations. Hence, the next subsections describe (i) the main concept our DSL is based on for documenting and tracing PLAs, i.e. the concept of PLAK, (ii) the DSL abstract syntax through the definition of the PLAK metamodel, its domain concepts, relationships and rules, and (iii) the DSL concrete syntax by defining a graphical language representation.

6.2.1 Product Line Architectural Knowledge (PLAK)

PLAK extends the main types of AK—design decisions, dependencies and rationale—with (i) variability design rationale and its dependencies, and (iii) traceability links between requirements and PLA descriptions.
6. PRODUCT LINE ARCHITECTURAL KNOWLEDGE

The variability design rationale is related to the definition of a certain variability point or variant, as well as the architectural interactions (dependencies) of variability with the other architectural elements. PLAK makes it possible to document the AK on both the external variability of the architecture configuration and the internal variability of components. This is due to the modeling primitives of the Flexible-PLA metamodel and the concept of PPC in which it is based on. Namely, PLAK provides modeling primitives for documenting the AK of {optional, alternative and multiple} simple and composite components —external variations—, as well as variations inside simple components —internal variations. The internal variation is especially relevant to describe variability concerning quality attributes that may crosscut one or many simple components. (e.g. the availability concern that is mentioned in Chapter 4). If architectural internal variability is neither described, documented nor traced, important concerns related to quality attributes could remain unknown and its knowledge could evaporate.

To support variability design rationale, this thesis presents four kinds of design decisions. They are classified as follows: Closed Design Decisions (Closed DDs), Open Design Decisions (Open DDs), Optional Design Decisions (Optional DDs) and Alternative Design Decisions (Alternative DDs).

- **Closed DDs**: They document the realization of the common structure of SPLs (core-assets). Figure 6.1 shows a Closed DD which documents the rationale of using a three-tier architecture to separate business logic from presentation and data storage. This DD documents the realization of the baking transactions which are common to all products that construct the banking systems SPL described in Chapter 4.

- **Open DDs**: They document the realization of the variability of SPLs. Figure 6.2 shows an Open DD which documents the rationale of using the aspect-oriented paradigm to enable flexibility, adaptability and reusability. This DD documents the realization of the feature availability which is a variability point in the banking systems SPL described in Chapter 4.

- **Optional DDs**: Open DDs consist of a set of Optional DDs. Optional DDs document each of the variants of an Open DDs. Figure 6.2 shows two Optional DDs which document the rationale of two availability tactics active redundancy and passive redundancy. These DDs document availability in two levels of strictness (e.g. recovery in milliseconds or less strict), as described in Chapter 4.

140
6.2 The PLAK Model

- **Alternative DDs**: Closed and Open DDs could have zero or more alternatives which document the realization of the common structure and the variability of SPLs, respectively. Figure 6.2 shows an Alternative DD which documents the alternative design for passive redundancy: the *exception tactic*.

![Figure 6.1: Product-line architecture knowledge: Closed DD](image1)

**Figure 6.1**: Product-line architecture knowledge: Closed DD - The figure shows the notation of a Closed DD that documents the rationale of using a *three-tier architecture* in the banking systems PLA.

![Figure 6.2: Product-line architecture knowledge: Open, Optional and Alternative DDs](image2)

**Figure 6.2**: Product-line architecture knowledge: Open, Optional and Alternative DDs - The figure shows the notation of Open, Optional and Alternative DDs that document the rationale of two availability tactics in the banking systems PLA.

The terms *open* and *closed* have been widely used to qualify variability points which can be left open allowing to add new variants, or closed not allowing to add variants [Atkinson et al., 2001] [Gurp et al., 2001]. This terminology refers to the time of binding variability points to specific variants (design time, compiling time, linking time or running time), as [Gurp et al., 2001] propose. This thesis follows the
6. PRODUCT LINE ARCHITECTURAL KNOWLEDGE

definition of binding time made by the work of Atkinson et al. [2001] and Gurp et al. [2001]. Specifically, the PLAK concept focuses on design time which is refined into two (sub-)binding times: the domain architecting process (or domain engineering ¹) in which DDs could be left open, and the application architecting process (or application engineering ²) in which Open DDs are closed.

Therefore, the four abovementioned types of DDs offer complete support for documenting the commonality and variability of SPLs. Specifically, Closed DD are completely closed or bound during the domain architecting process, whereas Open DDs are defined during the domain architecting process but their binding is intentionally delayed. It is at the time of deriving product applications that Open DDs are bound. This means that the application architecting process implies the binding of all Open DDs, i.e. Open DDs are bound to the appropriate Optional DD, whereas from the architectural point of view, variability points are bound to the appropriate variants according to these decisions.

It is necessary to emphasize that the PLAK concept focuses on architecture design and (architectural) design decisions as first-class citizens of the architecture, in contrast with OO-like design [Atkinson et al., 2001; Gurp et al., 2001]. In PLAK, the binding process to derive specific products may be realized at the architectural design decisions level, in contrast to OO-like design where the binding process is realized at design or even code level. After binding, the code for specific products can be automatically generated by means of model-to-text transformations.

To support variability design rationale, PLAK also entails the definition of dependencies between two of the any abovementioned DDs: Open, Closed, Optional or Alternative. Additionally, these design decisions entail a set of concepts which are needed to completely support reasoning and rationalization of rationale behind them: constraints, assumptions, rationale, design and patterns³. Constraints specified by requirements, methodologies, processes, or business goals, should be taken into account when selecting an appropriate design decision among various alternatives before a decision is made. Design decisions rely on assumptions that architects make. Rationale justifies decisions that are made in the design process based on some factors such cost, risk, or tradeoffs. Finally, the design defines the particular solution; this (solution) design may apply a pattern, which is a known solution to a recurring problem.

¹See Chapter 2, Section 3
²See Chapter 2, Section 3
³Farenhorst & de Boer [2009] define the major elements that most design rationale models have consensus on.
Finally, architectural design decisions may be used to establish links between requirements and architectures and vice-versa, which can be traced forward and backward [Könemann & Zimmermann 2010; Kruchten et al. 2009; Mirakhorli & Cleland-Huang 2011]. Similarly, this thesis presents a decision-centric approach where traceability links are created around the design decisions. In this case, tracing features to their realization in PLAs is focused on. Therefore, the design decisions turn into the links between feature and PLA models, which this thesis has called design decision traceability links. Design decision traceability links comprise the semantics of the linkage rules that establish the bridge between feature and PLA models. These linkage rules define the logics to create links between specific concepts of the Feature and Flexible-PLA metamodels.

Summarizing, PLAK can be defined as the joining of the following concepts: \( \text{PLAK} = \{ \text{Open DD, Closed DD, Optional DD, Alternative DD, Dependency, Rationale, Constraint, Assumption, Design, Pattern, and Design decision traceability link} \} \). These concepts have been included in a single model to completely document the knowledge of PLAs in an assisted and formal way.

### 6.2.2 Abstract Syntax: Metamodel description

The PLAK Metamodel formalizes the notion of PLAK. The PLAK Metamodel provides modeling primitives to represent and document the new types of design decisions that support variability design rationale —{Open, Closed, Optional or Alternative} DDs. This includes the design rationale of internal and external variability, their dependencies and rationale, as well as the linkage rules between the Feature and Flexible-PLA metamodels.

The PLAK Metamodel contains a set of inter-related metaclasses. These metaclasses define a set of properties and services for each concept considered in the PLAK definition. On one hand, metaclasses, their properties and their relationships describe the structure and the information that is necessary to capture the design decisions, their rationale, and the linkage rules between features and PLAs. On the other hand, the services of metaclasses allow us to manage models by creating, destroying, adding or removing elements which are compliant to the constructors of the metamodel. Those constraints that cannot be defined using modeling primitives (such as those that have been described by annotations in Figure 6.3) are specified using OCL [Object Management Group 2011b].
Specifically, the PLAK Metamodel contains the metaclasses that describe the following concepts:

- Open DD, Closed DD, Alternative DD, Optional DD, Rationale, Constraint, Assumption, Design, and Pattern (see marker A in Figure 6.3).
- Linkage Rule, the Feature Model concepts, and the Flexible-PLA Model concepts (see marker B in Figure 6.3).

The metaclass DesignDecision offers the primitives to instantiate a design decision. It defines five properties: name, description, stakeholder of the design decision, its status and version. A design decision consists of Alternative DDs, which is specified in the metamodel through the relationship consistsOf between the metaclasses DesignDecision and AlternativeDesignDecision. As defined in the previous subsection, we distinguish between Open DDs and Closed DDs. These types of design decisions are specified in the metamodel by means of the metaclasses OpenDesignDecision and ClosedDesignDecision, respectively. An Open DD is composed of a set of Optional DDs. This primitive is specified in the metamodel through the aggregation isComposedOf between the metaclasses OpenDesignDecision and OptionalDesignDecision. All of the {Open, Closed, Optional, Alternative}DesignDecision metaclasses inherit from the DesignDecision metaclass. A design decision may dependsOn other design decisions. Finally, the metaclass DesignDecision is composed of the metaclasses Constraint, Assumption, Design and Rationale. The metaclass Rationale defines four properties: why, cost, risk and tradeoffs. These properties justify the design decision, make an effort estimation of implementing the design decision, evaluate its risks, and evaluate its pros and cons, respectively. Finally, the metaclass Design may apply zero or more Patterns. It is specified in the metamodel through the relationship applies (see marker A in Figure 6.3).

Because the design decisions act as traceability links between features and architecture concepts, the metaclasses {Open, Closed, Optional}DesignDecision offer the primitives to define Linkage Rules. The key Feature concepts that are involved in the linkage rules are: Solitary Feature, Feature Group, and Grouped Feature. The key Flexible-PLA concepts that are involved in the linkage rules are: Component, Plastic Partial Component, Connector, Variability Point and Variant. The linkage rules are specified in the metamodel through the following metaclasses:
6.2 The PLAK Model

- *ClosedDD_LinkageRule* (see marker B.1 in Figure 6.3).
- *ExternalOpenDD_LinkageRule* (see marker B.2 in Figure 6.3).
- *InternalOpenDD_LinkageRule* (see marker B.3 in Figure 6.3).
- *ExternalOptionalDD_LinkageRule* (see marker B.4 in Figure 6.3).
- *InternalOptionalDD_LinkageRule* (see marker B.5 in Figure 6.3).

The logics of the possible linkage rules between Feature and Flexible-PLA metamodels have been defined in Table 6.1, taking the following considerations into account:

- A Design Decision may involve one or more Feature concepts, and one or more PLA concepts.
- Variability in the Feature Model is specified by means of *Solitary Features* whose cardinality attribute is 0..n, *Feature Groups*, and *Grouped Features*.
- Variability in the Flexible-PLA model is specified by means of optional {*Connectors, Components* and *PPCs*} which describe the external variability of architecture configuration, whereas *VariabilityPoints* and (optional and alternative) *Variants* describe the internal variability of PPCs.

**Table 6.1:** Linkages Rule Logics for Design Decision Traceability Links

<table>
<thead>
<tr>
<th>Linkage Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClosedDD</td>
<td>A mandatory <em>Solitary Feature</em> can trace with a <em>Component</em></td>
</tr>
<tr>
<td></td>
<td>A mandatory <em>Solitary Feature</em> can trace with a <em>PPC</em></td>
</tr>
<tr>
<td>ExternalOpenDD</td>
<td>An optional <em>Solitary Feature</em> can trace with an optional <em>Connector</em></td>
</tr>
<tr>
<td></td>
<td>A <em>Feature Group</em> can trace with an optional <em>Connector</em></td>
</tr>
<tr>
<td>InternalOpenDD</td>
<td>A <em>Feature Group</em> can trace with a <em>Variability Point</em></td>
</tr>
<tr>
<td>ExternalOptionalDD</td>
<td>A <em>Grouped Feature</em> can trace with an optional <em>Component</em></td>
</tr>
<tr>
<td></td>
<td>A <em>Grouped Feature</em> can trace with an optional <em>PPC</em></td>
</tr>
<tr>
<td>InternalOptionalDD</td>
<td>An optional <em>Solitary Feature</em> can trace with a <em>Variant</em></td>
</tr>
<tr>
<td></td>
<td>A <em>Grouped Feature</em> can trace with a <em>Variant</em></td>
</tr>
</tbody>
</table>
Figure 6.3: PLAK Metamodel
It is important to emphasize that the PLAK metamodel, as well as the Flexible-PLA metamodel, is compliant with the MOF four-level architecture (see Section 5.3). Hence, the PLAK, Flexible-PLA and Feature metamodels are defined at the M2 layer, whereas the models that conform to these metamodels are specified at the M1 layer. The MOF architecture provides the facilities to construct models at layer M1 using the PLAK metamodel primitives and guarantee model correctness. Finally, when specifics models of the M1 layer are executed, their instances are created at the M0 layer (see Figure 6.4). The formalization through the MOF architecture allows PLAK concepts to be exported to other models by extending other metamodels defined at the M2 layer, or by model transformation.

Figure 6.4: The PLAK Metamodel in MDD - The PLAK metamodel is defined at the laye M2 of the MOF four-layer architecture

### 6.2.3 Concrete Syntax: Graphical language description

A graphical modeling language has been defined as this kind of language is usually more intuitive. Figure 6.5 describes the main concepts and their graphical representations, and Figure 6.6 shows a snapshots from the FPLA modeling framework in which the
PLAK graphical language is illustrated by using the pilot study of the SPL for banking systems (see Chapter 4).

The four kinds of architectural DDs (Closed, Open, Optional and Alternative DDs) share a common representation: they are represented by blue rectangles in which compartments, which store rationale, assumptions, constraints, and design items, can be added. Closed DDs can be identified by an icon that depicts a closed lock, whereas Open DDs can be identified by an icon that depicts an open lock (see the Open DD AvailabilityTactic in Figure 6.6). Optional DDs can be identified by an icon that depicts a graph, whereas alternative DDs can be identified by two arrows with opposite directions and colors (see the Optional DDs ActiveRedundancy and PassiveRedundancy in Figure 6.6). Regarding the information that DDs capture, Figure 6.6 shows the DD ActiveRedundancy that contains the rationale for this DD. It presents (i) the justification of why this DD permits recovery of failure in milliseconds, (ii) the cost estimation, (iii) how the risk is high due to the problem of synchronization, and finally (iv) the tradeoff of being highly available (see Figure 6.6). Figure 6.6 also shows the DD FacadeStrategy which defines a constraint to be considered at the time of implementing the front ends.

Dependencies between DDs are represented by dashed lines. Figure 6.6 shows the dependency between the access to the common banking system functionality from ATMs and/or web application and its availability 24/7 (see the DDs FacadeStrategy and AvailabilityTactic).
6.3 The PLAK 4+1 view model

Figure 6.6: PLAK graphical modeling language: Banking systems PLA - The figure shows two Open DDs and the Optional DD of one of them. These DDs trace variability in features with its realization in the PLA.

Links between DD and features and DD and architectural elements are represented by solid lines. Figure 6.6 shows the Open DDs AvailabilityTactic which traces the variability in the feature availability 24/7 to its realization in the PLA through the variability points RequestManaging and Updating. In addition, Figure 6.6 shows the Optional DDs ActiveRedundancy and PassiveRedundancy which trace the variants Strict and Non-strict to their realization in the PLA through the variants LoadBalancing and Synchronization for strict availability, and Routing and DataMonitoring for non-strict availability.

6.3 The PLAK 4+1 view model

The PLAK 4+1 view model extends the Flexible-PLA 4-view model\(^1\) with the PLAK view. The PLAK view provides architects with the modeling primitives for documenting the architectural knowledge of Flexible-PLA models and tracing features with their

---

\(^1\)Flexible-PLA 4-view model decomposes the structural viewpoint of PLA definitions into four views: Core, Variability, Derivation and Product (see Section 5.4)
6. PRODUCT LINE ARCHITECTURAL KNOWLEDGE

realization in the PLA. The FPLA modeling framework conforms to this 4+1 view model.

6.3.1 Exemplification

The PLAK view is exemplified by using the pilot study of the SPL for banking systems (see Chapter 4) and several snapshots from FPLA. This pilot study has certain needs of architectural documentation and traceability which should be captured and modeled. This has permitted to check if PLAK modeling primitives satisfy the expected expressiveness to document the banking systems PLA. In the pilot study it is necessary to document: (i) The architectural tactics that implement the strict and non-strict availability that the banking systems SPL requires. (ii) The implementation of strict availability 24/7 through the use of the active redundancy tactic, and specifically by implementing the variants Synchronization and LoadBalancing. (iii) The implementation of non-strict availability through the use of the passive redundancy tactic, and specifically by implementing the variants Routing and DataMonitoring. (iv) The dependency between the access to the common banking system functionality from ATMs and/or web application and its availability 24/7. Finally (v) the traces between features and their realization in the PLA are required for this pilot study.

The PLAK model resulting from modeling the above knowledge is shown in Figure 6.7. This figure shows an excerpt from the feature model on the left-side and an excerpt from the (Flexible-)PLA model on the right-side. The DDs FacadeStrategy, AvailabilityTactic, ActiveRedundancy, and PassiveRedundancy constitute part of the PLAK model of the banking system SPL.

To give evidence that the PLAK modeling primitives are expressive enough to document the knowledge required in the banking systems PLA, the PLAK model of Figure 6.7 is analyzed as follows:

- PLAK provides the modeling primitives to create OpenDesignDecisions that store the design rationale of product-line variability. This is the case of the decisions regarding the implementation of Availability 24/7, which is intentionally left open to two mutually exclusive options: strict and non-strict availability (see label A in Figure 6.7).
6.3 The PLAK 4+1 view model

- PLAK provides the modeling primitives to create OptionalDesignDecisions. This is the case of implementing each one of the design decisions: (i) ActiveRedundancy (see label C in Figure 6.7) and (ii) PassiveRedundancy (see label D in Figure 6.7) which implement strict and non-strict availability, respectively.

- PLAK provides the modeling primitives to create dependencies between DDs. This is the case of defining the dependency between the Open DDs FacadeStrategy and AvailabilityTactic (see labels A and B in Figure 6.7).

- PLAK provides the modeling primitives to create links between FeatureGroups and optional Connectors. This is the case of the links that define the relationship between the feature Access and the optional connections, defined between the two front ends and the components that implement the banking transactions—Balance, Withdrawal, Deposit and Transfer— (see label La in Figure 6.7).

- PLAK provides the modeling primitives to create links between FeatureGroups and VariabilityPoints. This is the case of the links that define the relationship between the feature Availability 24/7 and the (Aspects)Variability Points which architecturally realize this feature (see label Lb in Figure 6.7).

- PLAK provides the modeling primitives to create links between GroupedFeatures and Variants. This is the case of the links that define the relationship between the grouped features strict and non-strict and the variants which architecturally realize these features (see labels Lc and Ld in Figure 6.7).

6.3.2 Considerations

An empirical assessment was conducted in order to evaluate how useful the PLAK model described in Figure 6.7 is. This empirical assessment consisted of a set of interviews to a group of students\(^1\). A group of eight post-graduate students participated in these interviews. The students belonged to a course of Advance Construction of Software Products in Computer Science and Technology Master and each had a different amount of years of work experience. These students were interviewed, and their answers were recorded, transcribed, grouped by quotes and coded. Coding means that parts of the text are given a code representing a certain topic of interest. Then, the coded material was translated into English and enriched with comments and reflections (i.e. memos). The evidence from this material is summarized as follows:

\(^1\)The script of the interview is available on https://www.surveymonkey.com/s/J6ZX6CR
Figure 6.7: PLAK model of the banking systems SPL
6.4 Discussion

- Without the knowledge provided by the PLAK model of Figure 6.7, it would be very difficult to explain why the components ATMFrontend and WEBFrontend have variability, as well as to be able to determine the valid configuration for a banking system application requiring strict or non-strict availability.

- Without the knowledge provided by the PLAK model of Figure 6.7, it may be difficult to know (i) if a banking system application requiring strict availability has to implement the Synchronization and LoadBalancing, or (ii) if a banking system application requiring non-strict availability has to implement the Routing and DataMonitoring. To capture this knowledge traceability needs to be made available in order to know which of the two alternatives holds, specifically traceability (i) between the feature strict availability 24/7 and the DD Active Redundancy and between the latter and the aspects Synchronization and LoadBalancing, and (ii) between the feature non-strict availability 24/7 and the DD Passive Redundancy and between the latter and the aspects Routing and DataMonitoring.

- PLAK models may be useful to identify where a feature is implemented in the PLA. As a result, it may also be useful to identify, given a change in a feature, where the change impacts the PLA. From the PLAK model of Figure 6.7 it is easy to observe that a change in the active redundancy tactic may impact two variants — Synchronization and LoadBalancing — which hook from different variability points. Perhaps this is not easy to locate in the code, but by making it available at the architecture-level, PLAK models facilitate this task. This impact knowledge may help to correctly implement the change while maintaining the integrity of the architecture.

Finally, Table 6.2 shows the number of PLAK modeling primitives (see Figure 6.7) which were used to document the knowledge of the banking systems PLA, i.e. the effort in documenting this PLA. These data uncover that, with a relatively low effort it is possible to achieve the abovementioned benefits.

6.4 Discussion

In the recent years the software architecture community has emphasized the need of formal approaches representing architectural design rationale and tools for managing this knowledge [Ali Babar et al., 2009a]. Table 6.3 summarizes some related work in
6. PRODUCT LINE ARCHITECTURAL KNOWLEDGE

Table 6.2: Numbers of the PLAK model of the banking systems SPL

<table>
<thead>
<tr>
<th>Modeling concepts</th>
<th>Number of PLAK modeling primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features (Feature Model)</td>
<td>15</td>
</tr>
<tr>
<td>Components</td>
<td>7</td>
</tr>
<tr>
<td>PPCs</td>
<td>2</td>
</tr>
<tr>
<td>Variability Points</td>
<td>2</td>
</tr>
<tr>
<td>Variants (Aspects)</td>
<td>4</td>
</tr>
<tr>
<td>Design decisions</td>
<td>6</td>
</tr>
<tr>
<td>Traceability Links</td>
<td>15</td>
</tr>
</tbody>
</table>

the AK area. Also discussed in this section is the contribution of this chapter to this area.

Over the last few years, the software community has worked to provide some formal representations and tools supporting design rationale [Babar & Gorton, 2007; Capilla et al., 2007; Jansen et al., 2007; Tang et al., 2007a]. Some of them are supported by tools such as AREL [Tang et al., 2007a], PAKME [Babar & Gorton, 2007], Archium [Jansen et al., 2007], and ADDSS [Capilla et al., 2007] (see rows 1-4 in Table 6.3).

Works, such as the [Kruchten et al., 2006a] ontology, Jansen et al. [2009] tool suite, or van Heesch et al. [2012] documentation framework are not included in Table 6.3 although they are reference works in the architecture documentation area. This is due to the fact that these works focus on (architectural)DD while neglecting their traceability with the architecture.

Although interesting comparisons of the above mentioned approaches are presented in Farenhorst & de Boer [2009]; Shahin et al., 2009; Tang et al., 2010, the key issue for this research is to analyze those approaches and tools that support variability design rationale. It is true that the abovementioned approaches have paid little attention to variability, hence, most tools have not been originally designed for managing variability design rationale. In a later work Capilla & Ali Babar [2008] propose a unified data model to explicitly support the relationships between design decisions and variability models of SPLs. Although their approach is promising by extending two existing tools to manage AK, such as PAKME [Babar & Gorton, 2007] (see row 2 in Table 6.3) and ADDSS [Capilla et al., 2007] (see row 4 in Table 6.3), it does not document the knowledge of “architectural variability”, instead, it documents the knowledge of a generic variability model. The authors consider a variability model as a decision model where variants have to be selected for configuring particular product applications.
From this point of view, this approach improves rationalization and communication of variability. However, since the architecture definition does not support specification of variability, it is quite difficult to store knowledge about how this variability is implemented, the architectural interactions of variability with the other architectural elements, as well as why that design variant was selected.

Clements et al. [2010] (see row 6 in Table 6.3) propose an annotative approach to document variability design rationale. They propose to design the architecture, while separately defining the variability points of the system in a variability guide where variants, their architectural effects, or dependencies among variants are textually specified. This work proposes annotations to link the architecture with the variability points which have been defined in the variability guide. Variability guides are textual documents that are difficult to process in an automated way. This approach does not support any specific mechanism for specifying architectural variability, otherwise it supports UML compositionality and inheritance mechanisms.

Additionally, separating variability from architecture definition — as Capilla & Ali Babar 2008 and Clements et al. 2010 proposed — usually implies an overhead of traceability links when the design rationale is documented (see a, Figure 6.8). This is due to the fact that the documentation of design rationale needs information regarding the requirement/s, architectural element/s and variability that are involved in each decision. This will cause that the design rationale is spread over these three models, creating an overhead of traceability links to maintain the connection and consistency of this rationale. That is why this thesis relies on an approach in which variability and software architectures were not independently specified. Design decision traceability links are used to link features with their realization in the architecture (see b, Figure 6.8). As a result, the traceability links for documenting variability design rationale are considerably reduced due to the complexity of the traceability graph is lower.
<table>
<thead>
<tr>
<th>Approach</th>
<th>Architectural concepts</th>
<th>Explicit Specification of Variability into PLA</th>
<th>Formalism</th>
<th>Architectural view</th>
<th>Traceability</th>
<th>Supporting Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>External variability (external architecture configuration)</td>
<td>Internal Variability (internal specification of non-composite components)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 AREL [Tang et al., 2007]</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Architecture rationale model</td>
<td>Decision view</td>
<td>ADD→Arch</td>
</tr>
<tr>
<td>2 PAKME [Babar &amp; Gorton, 2007]</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Architectural knowledge data model</td>
<td>Decision view</td>
<td>Req→ADD→Arch (coarse-grained)</td>
</tr>
<tr>
<td>3 Archium [Jansen et al., 2007]</td>
<td>Components and connectors</td>
<td>—</td>
<td>—</td>
<td>ADL + Archium model</td>
<td>Decision view and C&amp;C View</td>
<td>ADD→Arch→Java Archium</td>
</tr>
<tr>
<td>4 ADDSS [Capilla et al., 2007]</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ADD Metamodel</td>
<td>Decision view</td>
<td>Req→ADD→Arch (fine-grained) ADDSS</td>
</tr>
<tr>
<td>5 Flexible-PLA [Perez et al., 2009]</td>
<td>Components, connectors, interfaces, ports, etc. and Plastic Partial Components, VP and variants</td>
<td>Optional connectors</td>
<td>Invasive Software Composition: AOSA</td>
<td>Core, Variability, Derivation, and Product views</td>
<td>Arch→AspectJ</td>
<td>FPLAv1.0.0</td>
</tr>
<tr>
<td>7 [Galvao et al. 2010]</td>
<td>Components and connectors</td>
<td>Aspectsual components and connectors</td>
<td>—</td>
<td>Variability design rationale grammar</td>
<td>C&amp;C View + variability design rationale</td>
<td>—</td>
</tr>
<tr>
<td>8 PLAK [Diaz et al., 2011b]</td>
<td>Components, connectors, interfaces, ports, etc. and Plastic Partial Components, VP and variants</td>
<td>Optional connectors</td>
<td>Invasive Software Composition: AOSA</td>
<td>Core, Variability, Derivation, Product and Decision views</td>
<td>Feature→ADD→Arch</td>
<td>FPLAv1.1.0 (fine-grained)</td>
</tr>
</tbody>
</table>

ADL: Architecture Description Language
ADD: Architectural Design Decision
AOSA: Aspect-oriented software architecture
PLAs: Product-Line Architectures
C&C: Component and connector
Finally, Galvão et al. [Galvão et al., 2010] (see row 7 in Table 6.3) propose a grammar to specify design rationale, including variability design rationale. Although they consider aspectual components and connectors to describe architectural variability, they do not explicitly support the relationship between the design rationale and the architecture.

Summarizing, the presented proposal (see row 8 in Table 6.3) differs from previous approaches in the following terms:

1. The PLAK metamodel captures variability design rationale, including the documentation of both external and internal variability. Because the Flexible-PLA metamodel supports the definition of external and internal variability, it is possible to store useful knowledge about how and why variability is implemented in PLA models.

2. The PLAK metamodel defines the linkage rules between the Feature and Flexible-PLA metamodels. These linkage rules guide architects in defining valid traceability links between feature and PLA models. Additionally, the overhead of traceability links is drastically reduced due to (i) a decision-centric approach that focuses on traceability links around the architectural design decisions, and (ii) an approach that integrates variability in the development artifacts, specifically in the PLA (see b, Figure 6.8).

3. PLAK metamodel primitives guarantee the correctness of PLAK models.
6. PRODUCT LINE ARCHITECTURAL KNOWLEDGE

6.5 Conclusions

This chapter presented a mechanism to capture and document the architectural knowledge that is present in PLAs. The PLAK model comprises the modeling primitives to i) document variability design rationale, ii) variability dependencies, and iii) traceability links between the feature and PLA models. The documentation of the design decisions associated with variations, and the capability to trace the life of these variations enable more effective product-line evolution [Mohan & Ramesh, 2007].

The PLAK model supports the documentation of the external variability of the architecture configuration but also the internal variability of simple components, along with their dependencies and rationale. The documentation of internal variability is especially relevant as it is usually related to the variability of quality concerns that may crosscut the PLA. Neither the documentation of design decisions associated with internal variations nor the capability to trace the life of these variations, had been dealt with so far.

The documentation of PLAK improves the understanding of design decisions made, and provides the basis for reasoning about future design decisions or changes, including those design decisions that affect the variability of a SPL. This may be a useful tool for software maintenance and evolution tasks in SPLE. Applying models as the primary artifacts for documenting design decisions supports the fact that evolution can be (semi-)automatically assisted, for example through the traversal of PLAK models when used to aid change impact analysis. A better understanding of change impact upon the architecture helps to evolve the architecture while preserving its integrity and reducing architectural erosion [de Silva & Balasubramaniam, 2012; Kruchten et al., 2009].

Finally, PLAK is also useful during the derivation of product applications to configure valid products according to the dependencies between Open DDs, or dependencies between Optional DDs from different Open DDs.

The contribution of PLAK and its application has been published in:

Chapter 7

Change Impact Analysis in Product Line Architectures

This chapter presents mechanisms for traversing product-line architecture descriptions, architectural knowledge documentation, and traceability definitions. Traversing PLA descriptions, their knowledge and traces allows one to analyze and determine the potential impact upon the architecture resulting from the implementation of a change in features, also known as change impact analysis (CIA). Its main purpose is to deal with objective 3 (OBJ 3) defined in Section 1.3, i.e. to provide guidance in the change decision-making process. In this regard, this chapter presents a new technique for CIA that targets PLAs. It proposes to join a traceability-based algorithm and a rule-based inference engine to effectively traverse modeling artifacts that account for variability. This technique is built on the previously described mechanisms for (i) specifying variability in PLAs, (ii) documenting PLA knowledge, and (iii) tracing variability between features and PLAs.

This chapter introduces the concept of architecture-based CIA, presents a CIA technique for PLAs and its exemplification, and finally discusses this contribution.
7. CHANGE IMPACT ANALYSIS IN PRODUCT LINE ARCHITECTURES

7.1 Architecture-based CIA

Coping with changing requirements is an essential issue in the evolution of software systems. Improving the understanding of the impact of change is key when addressing software evolution. Change impact analysis (CIA) is fundamental in software evolution, as it allows one to determine the potential effects upon a system resulting from a proposed change [Arnold, 1996; Bohner, 1996]. Architecture-based CIA gives insight into how much the architecture will be impacted in order to handle changing requirements—specifically architecturally significant requirements. Reasoning about impacts on the architecture can provide the high-level insight necessary to make better requirements-driven evolution decisions [Bass et al., 2003].

The first step is to define what the term software architecture encompasses in analyzing change impact. Over the past few years, there has been a general consensus that a software architecture should be seen as “the result of a set of design decisions rather than a set of components and connectors” [Bosch, 2004], or as “the set of principal design decisions made during the system’s conceptualization and development” [Taylor et al., 2009]. For this reason, this thesis considers modeling architectural knowledge—defined as a set of architectural design decisions, their dependencies and rationale—as essential for analyzing change impact on architectures. As a result, dependencies, rationale, constraints and tradeoffs between design decisions must be captured so that the impact of a change on one decision is understood across the broader context of other architecturally significant decisions [Mirakhorli & Cleland-Huang, 2011]. Documenting the rationale behind the use of specific mechanisms to realize architectural variability, as well as dependencies between variants, enables more effective change management [Mohan & Ramesh, 2007]. Therefore, CIA techniques need to be adapted to assess a broader concept of architectural knowledge and, in the domain of SPLs, CIA techniques need to be adapted to assess product-line architectural knowledge, in which variability design rationale is documented. As a result, variability must be considered as a critical element in SPL change impact analysis. Additionally, traces are also essential for controlling the evolution of software systems [Capilla et al., 2007].

The purpose of traceability is to understand the relationship between requirements and their architectural realization, in both forward (from requirements to architecture) and backward (from architecture to requirements) directions.

This chapter addresses CIA of changing features upon PLAs. In this regard, this thesis proposes to join a traceability-based algorithm and a rule-based inference engine with the aim of traversing PLA models via a set of traceability links and propagation rules. To achieve this, variability plays a key role. For this reason, it is important that...
the models of PLA and traceability must be able to completely support variability. This is why we consider mechanisms for (i) specifying variability in PLAs, (ii) documenting PLA knowledge, and (iii) tracing variability between features and PLAs. Therefore, this CIA technique is based upon the contributions of this thesis: the Flexible-PLA metamodel, which addresses the problem of specifying variability in PLAs; and the PLAK metamodel, which supports the documentation of design decisions associated with variability, the design rationale behind the variability, the dependencies between design decision, as well as the definition of the basic traceability linkage between features and PLAs. These metamodels form the basis for identifying the architectural artifacts that are impacted by changes in SPL requirements, as well as in product-specific requirements. What remains is to define the technique for effectively traversing the models to obtain a CIA method that can account for variability.

7.2 The CIA technique

The CIA technique that this thesis defines consists of both a traceability-based algorithm and a rule-based inference engine, which work together to analyze change impact in PLAs from the structural point of view. This involves the traversal of PLAK models (which encompass all others: feature, PLA, and PLA knowledge models), based on a set of traceability links and propagation rules, to determine the potential impact of implementing a change. Both the algorithm and the inference engine work as follows:

Traceability-based algorithm Given a change in features, the traceability-based algorithm allows us to determine (i) the first-order design decisions that are involved with the feature to be changed, (ii) the n-order design decisions that depend on the first-order design decisions, and (iii) the first-order architectural elements that are involved in each (first and n-order) design decision. The algorithm traverses the traceability links that bridge features and PLA elements, and the dependency relationships between design decisions.

Rule-based inference engine Given a change in the PLA that realizes a change in features, the rule-based inference engine fires propagation rules to assess how the change may propagate to the architecture. In other words, when a modification over the PLA is applied, propagation rules are fired to simulate the effects on the rest of the PLA. The n-order architectural elements that are impacted by the change are thereby obtained.
The traceability-based algorithm, the rule-based inference engine, and the types of changes that are take into account are presented in the next subsections.

7.2.1 Change Typology

Change can be classified as: structural evolution, interface evolution, and design rationale evolution (see Section 2.2.4.1). Interface and rational evolution are not the focus of this work, since there may be ripple-effects which cannot be completely analyzed automatically, making manual intervention necessary. Hence, this thesis focuses on structural evolution, i.e. those changes in requirements that affect the architectural structure. Specifically, these changes may affect the external structure of the architecture (external structural evolution) or the internal structure of their elements (internal structural evolution) as described in Section 2.2.4.1.

Structural evolution in the SPL domain is considered in this thesis as follows. A change in features may affect (i) the entire SPL (core components and their connections), (ii) specific products (optional components and connections), or (iii) both (plastic partial components with common and variable functionality). The specific cases which we have considered are:

- Additions/deletions of components and PPCs (and their connections), variability points and variants.
- Additions/deletions of interfaces and services\(^1\) to/from components and PPCs.
- Additions/deletions of variability points and variants.
- Additions/deletions of services to/from variants\(^2\).

Given a change in features, the traceability-based algorithm retrieves the design decisions and architectural elements which may be impacted by the change. Then, given a change in the PLA (either of the abovementioned changes) that realizes the change in features, the rule-based inference engine returns the change propagation in the architecture.

---

\(^1\)These services can be public or private

\(^2\)These services can be published or not by one or more PPCs
7.2 The CIA technique

7.2.2 A traceability-based algorithm

The traceability-based algorithm retrieves the effects of changes in features. It returns the set of (first and n-order) design decisions, as well as first-order architectural elements that are impacted by changes. Since a change may impact many architectural elements, due to dependencies between design decisions, it is necessary to narrow the traversal of the traceability model, as well as control the infinite loops. Several techniques that address these issues were studied and, in the end, latent semantic analysis (LSA) [Lan-1998] was selected. LSA infers the meaning of words from natural text by statistical analysis of the context where words are used. LSA is applied to the assumptions, constraints, and rationale of a design decision and a set of key words related to the features to be changed. LSA measures the similarity between design decisions and the features to be changed.

The traceability-based algorithm that has been developed based on LSA is presented in Code 1. Code 1 describes the pseudo-code algorithm and explains each step through comments.

7.2.3 A rule-based inference engine

The rule-based inference engine that has been designed implements the rules that determine the propagation of changes in PLAs. The propagation rules are presented in Table 7.1 where $C$ is a component; $PPC$ is a plastic partial component; $VP$ is a variability point; $RI(C)$ is the required interface of $C$; $RI(PPC)$ is the required interface of $PPC$; $PI(C)$ is the provided interface of $C$; $PI(PPC)$ is the provided interface of $PPC$; $Hooks$, $Weaves$, $Pointcut$, $Advice$ and $Defines$ are the primitives to address variability inside PPCs; $UpdateC$ is the modification to the cardinality of a variability point; and $N$ is a neighbor of a component or a PPC. These rules are a set of statements following the pattern $Relation(element, subelement)$ in such a way that $element$ is related to $subelement$, and the addition/deletion of $Relation$ implies the addition/deletion of the subelement from the element. For instance, the deletion of $PI(C,s)$ consists of deleting a service $s$ that is provided by $C$ through its provided interface; the addition $Hooks(VP,variant)$ consists of adding the relation hooks from a variability point to a variant.
7. CHANGE IMPACT ANALYSIS IN PRODUCT LINE ARCHITECTURES

Code 1 Traceability-Based Algorithm

//foDD : first-order design decisions
//noDD : n-order design decisions
//foAE : first-order architectural elements

CONST LSAvalue; //stop condition to narrow the traversal

[vector] noDD;

[vector] traceabilityBasedAlgorithm (changedFeature, keywords) {
    [vector] foDD;
    [vector] foAE;

    //Traversing the PLAK model to get the DDs which are involved given a change in one feature
    [vector] foDD = traversalFromFeaturesToDDs (changedFeature);
    for each designdecision in foDD do {
        if (designdecision.isOpenDD && designdecision.hasOptinalDDs)
            foDD.addOptionalDDs(designdecision);
        else if (designdecision.isOptionalDD)
            foDD.addOpenDDs(designdecision);
    }
    noDD = foDD;
    for each designdecision in foDD do {
        //Traversing the PLAK model to get the AE which are impacted through a design decision
        foAE += traversalFromDDsToAE (designdecision);
        //Traversing the dependencies of the designdecision with other design decisions
        foAE += processDependencies (designdecision, keywords);
    }
    return foAE;
}

[vector] processDependencies (designdecision, keywords) {
    foAE = new [vector];
    [vector] dependencies = designdecision.getDependsOf();
    if (dependencies.isEmpty())
        return foAE;
    else{
        for each dependency in dependencies do {
            currentLSAvalue = LSA (keywords, designdecision, dependency);
            if ((currentLSAvalue>LSAvalue) and (not noDD.contains(dependency))) {
                noDD.add(dependency);
                if (dependency.isOpenDD && dependency.hasOptionalDDs)
                    noDD.addOptionalDDs(dependency);
                else if (dependency.isOptionalDD)
                    noDD.addOpenDDs(dependency);
                foAE+= processDependencies (dependency, keywords);
                foAE+= traversalFromDDsToAE (dependency);
                return foAE;
            }else
                return foAE;
        }
    }
}

### Table 7.1: Rule-based inference engine: propagation rules

<table>
<thead>
<tr>
<th>Type</th>
<th>ID</th>
<th>Rule Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intra-Component</td>
<td>R1</td>
<td>deletion RI(C,s) may cause deletion PI(C,s)</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>addition PI(C,s) may cause addition RI(C,s)</td>
</tr>
<tr>
<td>Intra-PPC</td>
<td>R3</td>
<td>deletion RI(PPC,s) may cause deletion PI(PPC,s)</td>
</tr>
<tr>
<td></td>
<td>R4</td>
<td>addition PI(PPC,s) may cause addition RI(PPC,s)</td>
</tr>
<tr>
<td></td>
<td>R5</td>
<td>addition Hooks(VP,variant) cause additions Weaves(VP), Point-cut(PPC,w), Advice(variant,w), UpdateC(FVP) and may cause addition PI(PPC) and/or RI(PPC)</td>
</tr>
<tr>
<td></td>
<td>R6</td>
<td>deletion Hooks(VP,variant) cause deletions Weaves(VP), Point-cut(PPC,w), Advice(variant,w), UpdateC(VP) and may cause deletion PI(PPC)</td>
</tr>
<tr>
<td></td>
<td>R7</td>
<td>deletion VP cause deletion Defines(PPC,VP) and Hooks(VP,variants)</td>
</tr>
<tr>
<td></td>
<td>R8</td>
<td>deletion Defines(PPC,VP) or Defines(PPC,VP) may cause deletion PI(PPC)</td>
</tr>
<tr>
<td></td>
<td>R9</td>
<td>addition Defines(PPC,VP) or Defines(PPC,VP) may cause addition PI/RI(PPC)</td>
</tr>
<tr>
<td>Inter-Component</td>
<td>R10</td>
<td>deletion PI(C) or PI(PPC) cause deletion RI(N)</td>
</tr>
<tr>
<td></td>
<td>R11</td>
<td>addition RI(C) or RI(PPC) cause addition PI(N)</td>
</tr>
</tbody>
</table>

### 7.3 Exemplification

This section illustrates how the traceability-based algorithm and the rule-based inference engine work by using the pilot study of the SPL for banking systems (see Chapter 4).

Suppose an engineer is given the task to improve the apparent reliability of the interaction between banking systems and ATM machines when a customer asks for their account balance to be displayed. In the current formulation of the banking system SPL, if there are too many requests, banking systems reject the request, as shown in the constraint of the design decision `FacadeStrategy` in Figure 7.1. In this overload status, the ATM machine aborts the session with the customer and returns their card. In the new formulation of the banking system SPL, instead of simply a rejecting and aborting the customer session, the SPL must be changed in such a way that banking systems respond to an overload status by taking advantage of the function `retry`: the system should wait for a random amount of time before retrying. If some number of retries are rejected, then the session is aborted. In essence, the protocol between
banking systems and ATM machines needs to be modified to decrease the number of aborted sessions.

Figure 7.1: PLAK model of the banking systems SPL - The figure shows the traceability between the feature, PLAK and Flexible-PLA models

Given this change, CIA is applied as follows: the feature(s) that may be affected by the change are identified and traced forward to the design decisions and components that are further impacted. In the banking systems example, suppose that an engineer selects the feature that is affected by the improvement of the protocol between banking systems and ATM machines. In the feature model of the banking systems SPL (see Figure 7.1 in Chapter 4) it is possible to identify that this improvement is related to the feature Access, and specifically, there is a feature ATMService that is about the connection between banking systems and ATMs (see Figure 7.2). Then, suppose that an engineer decides to add a new functionality to the feature ATMService which allows a previously rejected request to be retried. This feature retry is optional, that is, it is specific to the bank system application to be configured (see Figure 7.2). The application of our technique in this scenario consists of first executing the traceability-based algorithm on the PLAK model of Figure 7.1 and then the rule-based inference engine.

The execution of Code 1 on the PLAK model of Figure 7.1 returns the set of design decisions and first-order architectural elements that are impacted by adding the new
7.3 Exemplification

Figure 7.2: Change in the banking systems feature model - Retry feature

The execution is described as follows. Because the new feature retry stems from the feature ATMSertvice, the input for the traceability-based algorithm is the feature ATMSertvice. The algorithm traverses the links from the feature ATMSertvice to the DDs of the PLAK model of Figure 7.1. There is only one DD impacted: the Optional DD ATMFrontend. Next, the traceability-based algorithm traverses the dependencies between this DD and others as follows: the DD ATMFrontend is an Optional DD that stems from the Open DD FacadadeStrategy, and the DD FacadadeStrategy depends on the DD AvailabilityTactic, which in turn, has two Optional DD: ActiveRedundancy and PassiveRedundancy. Finally, the traceability-based algorithm traverses the links from this set of DDs to the Flexible-PLA model of Figure 7.1. As a result, the traceability-based algorithm retrieves the following architectural elements: the PPC ATMFrontend, the variability points RequestManaging and Updating, and their respective variants.

Execution 1 Execution of the traceability-based algorithm

```
input: featureChanged = 'ATMSertvice'
output:
  noDD = ['ATMFrontend', 'FacadeStrategy', 'AvailabilityTactic', 'ActiveRedundancy', 'PassiveRedundancy']
  foAE = ['ATMFrontend', 'Updating', 'RequestManaging', 'DataMonitoring', 'Roating', 'Synchronization', 'LoadBalancing']
```

Once the algorithm returns the set of impacted elements, the engineer can manually examine them and determine why and how they could be affected by the change. The impacted elements are components and connectors, as well as the DDs which store constraints, assumptions, rationale, etc. In this scenario, the retrieved components and DDs do not require to be changed; instead, they determine how and where the new
function retry should be realized. From the dependency between front-end services and the availability tactic, the engineer is able to infer that the function retry must be implemented in a different way in the cases of active and passive redundancy. In the first case the retry should be automatically directed to another active node, while in the second case the retry requires explicit data synchronization, route change, and retry execution. As a result, the engineer proposes its realization by adding a new variability point that stems from the PPC ATMFrontend (see Figure 7.3).

Figure 7.3: Change in the banking systems PLA - Adding the variant retry

The input for the rule-based inference engine is the addition of the new variability point (PV) retry and the two variants SimpleRetry and ComplexRetry. The execution of the propagation rules of Table 7.1 in the Flexible-PLA model of Figure 7.1 automate the change propagation as follows: First, adding the new variability point (PV) retry, stemming from the PPC ATMFrontend, results in (by rule R9) the addition of a service that provides the function retry through an interface of the PPC ATMFrontend to the PPC ATM. Second, adding the new variants SimpleRetry and ComplexRetry requires adding the relationship hooks from the VP retry to these two variants. Third, adding the relationship hooks results in (by rule R5) the addition of the relationships weaves, pointcut, and advice, and the update of cardinality.

In essence, the purpose of automating the traversal of PLAK models is to gather the set of nodes of potential interest for the engineer, and the possible additions and deletions that could be automated. That includes detecting features or design decisions that turn out to be in conflict with the proposed change. The engineer would then need to examine the nodes in that set manually to validate the proposed change.
7.4 Discussion

Many approaches have been proposed to support CIA at the source-code level [Kagdi et al., 2008; Kim et al., 2008], but few have addressed CIA at the architectural level [Feng & Maletic, 2006; Hassan et al., 2010; Zhao et al., 2002]. It is commonly agreed that traceability is key for identifying artifacts affected by a change [Pohl et al., 2001], from requirements to architecture and vice versa. In this direction, there is a growing body of approaches that address traceability between requirements and architecture [Chen & Chen, 2009; Olsen & Oldevik, 2007; Ramesh & Jarke, 2001], and more specifically, variability traceability between requirements and architecture in SPL [Mohan & Ramesh, 2007; Moon et al., 2007; Satyananda et al., 2007]. Moon et al. propose a variability trace metamodel that connects two metamodels, requirements and architecture, considering variability. Satyananda et al. propose a framework to formally identify traceability between the feature and architecture models using formal concept analysis, functional decomposition, and a set of mapping analysis rules. These approaches show potential and are promising but do not address change impact analysis. Moreover, these approaches only support architectural variability by adding/removing components or connections (external variability). Consequently, they define traceability links using high-level abstractions of variability, that is, variability of coarse-grained components (complex components), neglecting variability inside components (internal variability). Additionally, only a few approaches consider architectural design decisions and design rationale to aid change impact analysis [Mohan & Ramesh, 2002; Riebisch & Wohlfarth, 2007; Tang et al., 2007a] or variability design rational in addition to architectural design decisions and rationale. Both the documentation of design decisions associated with variations and the capability to trace the life of these variations are key to effectively aiding CIA in SPLs [Mohan & Ramesh, 2002, 2007]. Their proposals are very close to our work, but they admit that their approach should be complemented by specialized design-modeling representations to model variation points.

To sum up, the novelty of the CIA technique that is presented in this chapter relies on coping with several key elements that can effectively aid CIA in SPLs: what the knowledge that aids CIA is, how this knowledge is modeled, and how this knowledge is utilized. They are addressed in our proposal by:

1. Tracing variability between features and PLAs. Our work differs from previous approaches [Mohan & Ramesh, 2007; Moon et al., 2007; Satyananda et al., 2007] in the concept of variability. The variability concept is fine-grained, considering
architectural variability by adding components and connectors, but also variability inside components by adding variability points and variants. This is why this thesis relies on (i) the Flexible-PLA metamodel to specify variability in PLAs, and (ii) the PLAK metamodel to trace features to PLAs.

2. Documenting architectural knowledge. Our work differs from previous approaches [Mohanty & Ramesh, 2002; Riebisch & Wohlfarth, 2007; Tang et al., 2007a] in the architectural knowledge that has been proposed. Conventional models for supporting architectural knowledge have been adapted and extended to capture PLA knowledge. This is the reason why the PLAK metamodel has been defined.

3. Combining a traceability-based algorithm and a rule-based inference engine to take advantage of the two techniques. Most authors only address one of them: a graph-based analysis (based on links) [Tang et al., 2007a; Zhao et al., 2002] or rule-based systems [Feng & Maletic, 2006; Hassan et al., 2010; Vora, 2010]. Some others have incorporated additional techniques to improve CIA by using Bayesian networks [Tang et al., 2007b]. Instead of this route, our work relies on latent semantic analysis over the documentation that is generated in the software development cycle [de Boer & van Vliet, 2008], which is showing promising results in architectural knowledge discovery.

7.5 Conclusions

Ineffective CIA complicates the decision-making process and seriously jeopardizes the success of software evolution. The identification of the architectural elements that are affected by changes in variability is critical to appropriately evolve SPLs. Tracing and documenting the design rationale behind variability enables more effective CIA. The CIA technique that was presented in this chapter combines a traceability-based algorithm and a rule-based inference engine for analyzing change impact in PLAs. It traverses the knowledge contained in two metamodels —Flexible-PLA and PLAK—that together provide the basis for (i) specifying variability in PLAs, (ii) documenting PLA knowledge, and (iii) tracing variability between features and PLAs.

The contribution of this chapter has been published in:

Part IV

AGILE CONSTRUCTION AND EVOLUTION OF PLAs
Chapter 8

Agile Product-Line Architecting Process

This chapter presents a process for the agile construction and evolution of product-line architectures. Its main purpose is to reconcile architecture and agility with the goal of dealing with one the main challenges that APLE faces: the iterative and incremental construction and evolution of product-line architectures while complying with the “be open to change” agile principle (see Section 1.3). In this regard, this chapter presents the Agile Product-Line Architecting (APLA) process that assists and guides architects in the following three tasks: (i) defining flexible and adaptive product-line architectures that facilitate the change, (ii) reducing the risk of unexpected consequences of changes and enabling the preservation of the architecture integrity, and (iii) providing architects with knowledge that assists and guides them in the change decision-making process, iteration after iteration during the APLE development process.

This chapter introduces mechanisms that enable agile architecting, describes the APLA process, presents the FPLA modeling framework that support it, and finally discusses this contribution.
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

8.1 Agile Architecting

This section describes the strategy for approaching architecture and agility. Aligning fruitfully software architectures and agility requires leveraging the inherent qualities of software architectures (e.g. abstraction, communication, reuse) while getting closer to the “open to change” agile principle. This alignment can be achieved as long as practitioners are able to count on mechanisms for enabling:

- The incremental design of features. Agile architects need mechanisms to flexibly construct PLAs by adding small increments of complete or partial features\(^1\) in each iteration of the APLE development process. This activity is called agile product-line architecting, or simply, agile architecting.

- The accommodation of new features or customizations on existing features. Agile architects need mechanisms to adapt PLAs, as new customer requests come in and as market conditions change over time through successive iterations of the APLE development process. This activity is called agile product-line evolution, or simply agile evolution.

While both agile architecting and agile evolution activities are conceptually different, they require the same mechanisms to carry them out. Throughout this chapter and the rest of the thesis, they will be referred to either in the case of incremental design, or in the case of accommodating changes. Similarly, the inputs of these activities —feature increments and feature evolution— are generically considered as changes in features.

It would be highly convenient and desirable if the mechanisms for enabling agile architecting would assist and guide agile architects, specifically in the following three tasks: (i) the design of PLAs which anticipate, facilitate and embrace change, (ii) the decision-making process of implementing changes in each APLE iteration, and (iii) the maintenance of architectural integrity, i.e. the preservation of earlier architectural design decisions, iteration after iteration, during the APLE development process. Regarding the first task, having flexibility and adaptability at the time of defining software architectures is essential so that architecting can be aligned with the “welcome to change” agile principle. Regarding the second task, the knowledge about the effects of a change upon the PLA provides information to reason about how and where to implement that change, and permits better evolution decisions based on risks, cost or

\(^1\)An increment is often smaller than a feature, understanding a feature as a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [Kang et al., 1990]
8.1 Agile Architecting

viability of the change. Regarding the last task, the continuous process of architecting should never result in the software’s degradation as consequence of intentionally or accidentally violating earlier design decisions and constraints [Hanssen & Figri, 2008]. In this sense, agile architects need knowledge about dependencies between design decisions, constraints, tradeoffs, etc., which assists them in countering or even avoiding several well-known negative effects that software evolution causes, such as architectural erosion [Perry & Wolf, 1992; van Gurp & Bosch, 2002], degeneration [Hochstein & Lindvall, 2005], drift [Perry & Wolf, 1992], or aging [Parnas, 1994].

Therefore, our understanding is that it is essential to have flexibility and adaptability at the time of defining software architectures, while guiding the change decision-making process and preserving architectural integrity. Consequently, this allows architecting to be aligned with the “be open to change” agile principle. It facilitates the evolvability characteristic of software architectures, defined as: “the ability of a software system throughout its lifespan to accommodate to changes and enhancements in requirements and technologies, that influence the system’s architectural structure, with the least possible cost while maintaining the architectural integrity” (excerpt from Bode & Riebisch, 2010).

Besides this point, there is a peculiarity in the case of PLAs which make them different from traditional architectures: PLAs have to manage the variability found in the products of a product-line. One of the major (and implicit) challenges to be dealt with is knowing how to support this variability during the iterative and incremental construction and evolution of PLAs. Although, it is necessary to emphasize that variability has also become a means to make software architecture more flexible in developments that do not need to manage variability among the different product applications from a product-line.

8.1.1 Agile Architecting through Flexible & Adaptive Architectures

Agile architects can take advantage of variability mechanisms to flexibly adapt software architectures and to incrementally develop them. Although variability has primarily been addressed in the domain of SPLE, variability is also a relevant characteristic of the architecture enabling the last responsible moment$^{1}$ for a decision$^{2}$, the planned

---

$^{1}$The Lean Construction Institute coined the term last responsible moment. See www.leanconstruction.org.

$^{2}$The moment at which failing to make a decision eliminates an important alternative [Poppendieck & Poppendieck, 2003].
evolutionary software development, or quality attributes such changeability [Galster & Avgeriou, 2011]. In fact, several authors distinguish product-line variability vs. software variability as “the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context” [Svahnberg et al., 2005].

Specifically, agile architects can take advantage of the PPC primitives that were presented in Chapter 5. Flexibility of software architectures is given by PPCs and variability points, while adaptability is given by the capabilities for changing variants as they are unaware of the linking context. PPCs reduce dependences and coupling between components and their variants, and enable easy and cheap (un-)weaving of variants. Hence, PPC primitives can be used for incrementally and iteratively refining and evolving the architectural components that compose a working PL architecture, or simply working architecture.

A working architecture is one that develops with the system, and includes only features that are necessary for the current iteration or delivery. This idea was also proposed by [Booch, 2009; Kruchten, 2009] as continuous architecting. Continuous architecting may help reduce big upfront architecture design and keep the system in-sync with changing conditions. Although the PPC variability mechanism has been successfully applied to SPLs to support internal variation of architectural components among the products of a SPL —i.e. to define variations among products—, it can also be used to flexibly add, remove and modify variants throughout the iterations of an agile lifecycle. In this case, variants behave as extensions (see Figure 8.1.a). Namely, the PPC variability mechanism behaves as an extension mechanism to flexibly compose pieces of software, as if we were building a puzzle. As a result, PPCs get closer and closer to customer needs by means of specifying the variants only when they are strictly required by a working product.

Therefore, the PPC variability mechanism is the backbone to support incremental development of architectural components through (i) the incomplete specification of components, and (ii) their extension by hooking new variants. Weavings specify where and when to extend PPCs through the use of variants (see Figure 8.1.a). Hence, working architectures can be incrementally and iteratively designed and evolved in each iteration by weaving/unweaving extensions, and/or by modifying the architecture configuration through optional components and connectors.
8.1 Agile Architecting

Figure 8.1: Agile Architecting mechanisms - a) PPC extension mechanism & b) Documenting “open to change” decisions

8.1.2 Agile Architecting through Change-Impact Analysis

Agile architects can take advantage of knowledge-based architecting \cite{Lago et al., 2008} to guide the decision-making process of implementing changes in each APLE iteration, as well as to facilitate the maintenance of the architecture integrity in each of these iterations.

We propose CIA as the main driver for agile architecting. The CIA technique was presented in Chapter 7 and is supported by the architectural models that were presented in Chapters 5 and 6: Flexible-PLA and PLAK models, respectively. These models promote communication between individuals and agile teams working on the system, and support (semi-) automatically reasoning over the space of architectural knowledge iteration after iteration. Both the models and the CIA technique assist architects during the agile architecting process. To use these models and technique during the agile architecting process, several adaptations were necessary. Briefly, the adaptations are described as follows.

Regarding the PLAK model, it was necessary to adapt their modeling primitives in order to capture the knowledge of adding feature increments or changing features in each agile iteration. This adaption consists of the following primitives:

- **Closed design decisions** (Closed DDs) are completely closed (or bound) in a given iteration and support the realization of those features that can be completed in one iteration and that architects considered unchanging over time.
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

• **Open design decisions** (Open DDs) are intentionally left open (or delayed) and support the realization of those features that cannot be completed in one iteration and that architects plan to complete iteration after iteration (see the design decision with the open lock of Figure 8.1.b). Open DDs consist of a set of optional design decisions.

• **Optional design decisions** (Optional DDs) support each of those increments in each agile iteration of an Open DD (see the optional design decision of Figure 8.1.b).

• **Alternative design decisions** (Alternative DDs) support the alternative realization of Closed and Open DDs, respectively.

These four types of DDs offer complete support for documenting the knowledge derived from the agile architecting process. The PLAK model supports the documentation of flexible and adaptive architectures through decisions which can be intentionally left delayed, and later closed in the following iterations.

Regarding the CIA technique, it was also necessary to adapt it in order to analyze the impact of adding or changing features in each iteration of the agile lifecycle. As presented in Chapter 7, this CIA technique consists of a traceability-based algorithm and a rule-based inference engine which traverse Flexible-PLA and PLAK models based on a set of traceability links and propagation rules. The process that this CIA technique implements consists of the two main steps described below:

(1) Given a change in features (adding, deleting or updating), the traceability-based algorithm determines (i) the first-order design decisions that are involved with the feature to be changed, (ii) the n-order design decisions that depend on the first-order design decisions, and (iii) the first-order architectural elements (PPCs, components, and connectors) that are involved in each (first and n-order) design decision. The algorithm traverses the traceability links that bridge features and architectural elements, and the dependency relationships between design decisions.

(2) Given a change in the working architecture that realizes the change in features, the rule-based inference engine fires propagation rules to obtain the change propagation in the working architecture. Namely, when a modification over the working architecture is applied to, propagation rules are fired to simulate the effects on the rest of the working architecture. We thereby obtain the n-order architectural elements that are impacted by the change.
To summarize, this work relies on the use of the concept of variability to iteratively and incrementally construct/evolve software architectures, as long as this variability is documented and traced through Open and Optional DDs. The CIA technique described within this section fits perfectly with the needs to analyze the impact of agile architecting, as it traverses Flexible-PLA and PLAK underlying concepts which support architectural variability. The output of the CIA technique provides change-impact knowledge that may be useful for reasoning about a proposed change in features and guiding the change decision-making process. It also might help to preserve the architectural integrity iteration after iteration during the agile lifecycle.

8.2 The Agile Product-Line Architecting Process

This section describes how the mechanisms for describing flexible and adaptive architectures, and the mechanisms for supporting knowledge-based architecting, work together to enable agile product-line architecting, or simply agile architecting. In this regard, this thesis defines the APLA process this section describes. This APLA process has been deployed in the agile method Scrum (see Scrum description in Chapter 4) as is described and discussed in following subsections.

8.2.1 APLA in Scrum

This thesis proposes a tailored Scrum development process where agile architecting is considered as a key activity to prepare the sprint (see Figure 8.2). Like Kruchten [2009], this proposal advocates the role of the architect on the agile team. In this way, the architecture team is part of the agile team and interacts with the rest of the members at the decision-making process by tracking architectural concerns and balancing them with business priorities. Additionally, this tailored Scrum fits into common frameworks for SPLE, such as that defined by Pohl et al. [2005]. This framework defines two subprocesses: domain engineering and application engineering. Domain engineering focuses on defining the common platform for all products of the SPL, while application engineering focuses on developing products through systematic reuse of the common platform. The tailored Scrum, according to these considerations, is described as follows:

The first step consists of capturing the requirements of the Product Owner from the product vision (features). Features may be decomposed into a list of user stories (US)
known as *SPL backlog* (see Figure 8.2). US describe the SPL features using scenarios written by customers without techno-syntax and including the acceptance criteria that validate them. Then, US are prioritized, based on business value, and assigned to *sprints* (also known as iterations). A sprint is a 2-4 weeks period of development time.

Scrum implements an iterative and incremental lifecycle based on these sprints. Sprints have a *sprint planning meeting* in the beginning when the *SPL Owner* and *Team* plan together what has to be done. In this tailored Scrum, the *agile architecting* tasks are developed in conjunction with the sprint planning meetings (see Figure 8.2).

The abovementioned Flexible-PLA and PLAK modeling artifacts as well as the CIA technique drive *agile architects* as follows:

- The Flexible-PLA model provides agile architects with primitives to iteratively and incrementally construct and evolve the *working architecture*. Changes are realized by using the extension mechanism of PPCs or/and changing the architecture configuration. Namely, changes are realized by adding/removing or modifying:
  - *Components*: Units of basic functionality, known as major software components. They are components of the working architecture, which are not able to support internal variability or extensions in the next iterations.
  - *PPCs*: Units of basic functionality, also known as major software components. They are components of the working architecture, which are able to support both variability and extensions in the next iterations, i.e. they are incrementally and iteratively refined (“open to changes”).
  - *Variants*: Functional or non-functional features which are not relevant enough to be major software components. They are part of the functionality that a PPC provides. They work as a mechanism to customize products from a SPL, as well as a mechanism to incrementally and iteratively extend the functionality and quality attributes of the products.
  - *Variability Points*: They define the place where PPCs can differ or extend their functionality.
  - *Connections*: They coordinate components and PPCs, and configure the working architecture.
  - *Optional Connections*: They coordinate components and PPCs, and support variability of the working architecture configuration.
The PLAK model provides agile architects with the primitives to document design decisions, dependencies, constraints, tradeoffs, risks, etc. of the working architecture. It also traces features with their realization in the working architecture.

The CIA technique is applied to the working architecture of the previous sprint (from the second sprint, where there is the first working architecture resulting from the first sprint). It provides agile architects with the change-impact knowledge resulting from the changes—or new features—planned for the sprint. From the impacted components and connections, the architects can reason about where and how to implement that change. From the impacted design decisions, the architects are aware of the effects of that change over previous constraints, tradeoffs, risks, and they therefore have knowledge to preserve the integrity of the architecture.

It is important to emphasize that the only difference between the first iteration and the rest is the fact that the first iteration starts the software architecture from scratch. In this work, we do not define a ZFR (Zero Feature Release) where the customer does not participate, as other approaches propose [Beck, 2004; McMahon, 2005]. We consider that the investment of time and cost in this ZFR does not guarantee that the decisions made will be definitive and the ZFR architecture will be preserved. Therefore, our first iteration is just one more where the customer participates. In each iteration, PPCs, variants or connections from the software architecture are updated, added and removed in a flexible way and without any restriction.

As a result of this tailoring, agile architects interact with the rest of the team at the sprint planning meetings in planning the features to be done in the current sprint. This planning is performed by means of tracking architectural concerns, such as constraints, risks, viability, etc., and balancing them with the business priorities. The output of sprint planning meetings consists of: (i) the sprint backlog, (ii) the tasks that must be performed to achieve the sprint goal, and (iii) the working architecture models (Flexible-PLA and PLAK models) to be implemented during the sprint (see Figure 8.2).

During the sprints, developers implement the working architecture, i.e. the common and variable assets which compose the SPL platform (domain engineering). From this SPL platform it is possible to configure specific products (application engineering) through a process where variability is derived (see Figure 8.2).
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

Figure 8.2: APLA in Scrum - The APLA process has been deployed in Scrum. The agile architecting tasks are developed in conjunction with the sprint planning meetings.

At the end of each sprint, the working architecture and a set of working products are delivered (see Figure 8.2). In the sprint review meeting (see Figure 8.2) the SPL Owner assesses the working products to validate that US were met or introduce changes into the US. The retrospective meeting (see Figure 8.2) is held to put continuous improvement into practice: what went well and what could be improved for the next sprint.

The process model of the above described tailored Scrum is presented below. The purpose of process models is to document and communicate processes so that processes can be better understood and executed. It also enhances the re use of processes. The OMG (Object Management Group) has developed the Software Process Engineering Metamodel (SPEM [Object Management Group, 2008]) which supports all method and process modeling primitives. Figures 8.3-8.5 show the process model of the tailored Scrum conforms to SPEM (see Appendix B to see the complete view of the process model, as well as legend of Figures).

Figure 8.5 shows the activity Agile Architecting, which is performed after US are estimated and selected to be done in the current sprint. Agile Architecting consists of three tasks: Change Impact Analysis, PLA Modeling and PLAK Modeling. The input of these three tasks is the Sprint Backlog, which is the output of the task US Estimation & Selection. The output of the task Change Impact Analysis is the Change-impact Knowledge which is used by the task Implementation. The output of the tasks PLA
8.2 The Agile Product-Line Architecting Process

*Modeling* and *PLAK Modeling* are the Flexible-PLA Model and the PLAK Model, respectively. To perform these models, these tasks use the *guidelines* Flexible-PLA Metamodel and PLAK Metamodel, respectively. These models are a specialization of the product *Working Architecture* which is used by the task *Implementation*.

![Figure 8.3: APLA in Scrum: Model process compliant to SPEM (I)](image1)
- This figure shows the activities *Definition SPL Backlog* and *Plan Sprint*.

![Figure 8.4: APLA in Scrum: Model process compliant to SPEM (II)](image2)
- This figure shows the activities *Development* and *Assess Sprint*.

183
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

Figure 8.5: APLA in Scrum: Model process compliant to SPEM (III) - This figures shows the activity Agile Architecting

8.2.2 Discussion

The modeling artifacts and the CIA algorithm that are described above, provide agile architects with the mechanisms to effectively make agile architecting:

- The Flexible-PLA model.

Agile architecting is performed in each sprint by taking advantage of the flexibility and adaptability that the Flexible-PLA modeling primitives provide. Each item of the product backlog is transformed into a product increment sprint after sprint by using the extension mechanism provided by PPCs. At the end of each sprint the result is a working product and working architecture.

PPCs and working architectures match the agile value responding to change over following a plan: PPCs easily accept changes by (un-)weaving variants and they are connected between them to configure working architectures. In regard to the twelve agile principles, the following details those that could be enriched through the presented proposal:
8.2 The Agile Product-Line Architecting Process

- **(P1)** Our highest priority is to satisfy the customer through early and continuous delivery of valuable software: PPCs help get closer and closer to customer needs over iterations, and by extension, the working architecture.

- **(P2)** Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage: PPCs are plastic, i.e. they are ready to be extended or modified at any moment. The architecture is open to changes due to the adaptation capabilities that PPCs and working architectures provide.

- **(P4)** The most efficient and effective method of conveying information to and within a development team is face-to-face conversation: The architecture team participates in every meeting of the project and embrace new customer needs. The architecture team shares architectural information among its members and with others involved in every scheduled meeting.

- **(P6)** Continuous attention to technical excellence and good design enhances agility: PPCs support the intuitive modularization and scaling of software by using its variability mechanisms, which can be advantageously used for adding or removing pieces of software throughout the different iterations that comprise the development of a working product. In addition, PPCs help us to easily apply a major technique used in ASD to cope with changes: refactoring. PPCs help us to extend and reorganize code by its encapsulation into variants, which avoids the inherent tangled-code that crosscutting concerns generate.

- The PLAK model and the CIA technique.

Knowledge-based architecting is becoming increasingly important in the agile community. Hence, the traditional Scrum lifecycle has been tailored over the last few years with the backlog grooming sessions [Leffingwell 2010; Pichler 2010] due to the need of focusing on what is coming up in the next sprint. Backlog grooming sessions give agile teams the opportunity to look further into the future of the product, and can alert them to technical challenges. The purpose of these sessions is to make improvements to the product backlog: break down user stories that are too big (epics), estimate backlog items, look deeper into the backlog to do longer-range technical planning, and prioritize the backlog [Pichler 2010]. Prioritization entails making decisions about the technologies, architecture and design options to deliver requirements. These decisions may aim to (i) implement functionally-complete features early, (ii) implement requirements exhibiting uncertainty or risk (fail as early as possible) early, or (iii) delay decisions
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

until the last responsible moment. Delaying decisions provides agile teams with more time to evaluate options, and gather feedback from customers.

In backlog grooming sessions, the role of software architecture is vital to understand the working product, looking further into future, technical planning, and prioritizing the backlog items. In this regard, the PLAK modeling primitives and the CIA techniques provide knowledge for understanding and reasoning about the increments and changing features, while trying to maintain the integrity of the architecture by taking risks, dependencies, and tradeoffs with earlier architectural decision decisions, into account.

8.3 The FPLA Modeling Framework

Flexible-PLA (FPLA) is an open-source graphical modeling framework that deploys the Feature, Flexible-PLA & PLAK models and its respective views. The FPLA modeling framework is available for the community as an Eclipse plug-in, specifically, it is supported by the Eclipse Indigo Modeling framework. In addition, it is important to mention that FPLA has been constructed following a model-driven development (MDD) approach. Hence, it has been semi-automatically generated from the Feature, Flexible-PLA & PLAK metamodels by using modeling tools such as Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF).

8.3.1 FPLA main characteristics

The main characteristics that FPLA provides are described as follows:

1. FPLA provides modeling primitives and graphical notation for specifying feature models.

2. FPLA provides modeling primitives for describing PLA: Flexible-PLA models. It provides intuitive graphical notation and expressiveness for specifying both internal and external variability.

https://syst.eui.upm.es/FPLA/home
3. FPLA provides modeling primitives for documenting and tracing PLAs: PLAK models. It provides intuitive graphical notation and expressiveness for documenting and tracing variability design rationale.

4. FPLA provides different views to facilitate architects with the design tasks, while making the PLA description more intuitive and friendly. These views are: Core, Variability, Services, Weaving, PLAK, Derivation, and Product.

5. FPLA guarantees the correctness of PLA models, which conform to Flexible-PLA & PLAK metamodels by its verification mechanism based on OCL (Object Constraint Language [Object Management Group, 2011b]) rules.

6. FPLA provides CIA. Specifically, FPLA incorporates the traceability-based algorithm that traverses the PLAK models and retrieves the design decisions and architectural elements which are impacted by a change in features.

7. FPLA supports the selection of a particular configuration out of the multitude of available architectural configurations that the PLA implements (application engineering process or derivation process).

8. FPLA supports the description of PLA models ready to be involved in a MDD process by transforming their outputs into semi-automatically generated code, specifically AspectJ [Kiczales et al., 2001]. FPLA automatically generates code skeletons and composes the code from external sources by using model-to-text transformations. Its aim is to link architecture and code. The traceability between architecture and implementation may avoid another common problem in ASD when code drifts so far apart that it makes it much easier to erode the software system architecture.

Figure 8.6 shows a snapshot from the FPLA modeling framework. The right-side shows the toolbar with all the modeling primitives that allow one to describe, document and trace PLAs. The complete set of palette tools used for all along with each one of the models that FPLA supports —Feature, Flexible-PLA and PLAK models— is shown in Figure 8.7. The left-side shows how these modeling primitives are represented in the drawing sheet when they are dragged and dropped.
Figure 8.6: FPLA Modeling Framework
8.3 The FPLA Modeling Framework

![Figure 8.7: FPLA Modeling Framework](image)

Figure 8.7: FPLA Modeling Framework - Tool palettes that provide the primitives to model feature, Flexible-PLA and PLAK models

8.3.2 FPLA: Empirical Assessment

The usability of the FPLA modeling framework and the understandability of its models were studied in two experimental settings, referred to as Test I and Test II (see Appendix C). Test I evaluates the understandability of Flexible-PLA models by using a set of examples that model a home automation systems SPL, whereas Test II evaluates the usability of the FPLA modeling framework for modelling the PLA of the banking systems SPL that is described in Chapter 4. A group of eight post-graduate students participated in these practical exercises. The students belonged to a course of Advance Construction of Software Products in Computer Science and Technology Master and each had a different amount of years of work experience. The students were asked to answer Test I and II after a twenty-minute class session where the main concepts of Flexible-PLA models and the main characteristics of the FPLA modeling framework were explained to students.
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

8.3.2.1 Test I: Description

After the 20-minute class session, the students answered Test I during the following 20 minutes. Test I consists of five groups of True/False questions (see Appendix C). Each group consists of the questions regarding a specific view: feature view for describing feature models, and core, variability, service & interface, and weaving views for describing Flexible-PLA models. Each group of questions starts with a figure that shows the view corresponding to the modeling of a home automation systems SPL. Briefly, home automation systems are described as follows: Smart homes are characterized by a high degree of automation; their devices are integrated into a home network, so that smart homes constitute entire systems that are made up of individual subsystems and whose behavior is determined by the software controlling it: the home automation systems. Home automation systems make it possible to coordinate the functions provided by different devices, this way users of home automation systems can access and control home devices and their functions remotely via the Internet and a common (centralized) user interface. The home automation systems SPL defines a family made of different home automation systems which are customized according to specific users needs.

The first group of questions from Test I shows the feature model of the home automation systems SPL (see Figure 8.8). In this group of questions, the students were questioned about mandatory, optional and alternative features of the model to check if they understood the meaning of these concepts. For instance, they were questioned about the modeling of optionality or obligation to implement features such as smart lighting or intrusion detection.

![Figure 8.8: FPLA model - Feature model of home automation systems SPL](image)

Figure 8.8: FPLA model - Feature model of home automation systems SPL
8.3 The FPLA Modeling Framework

The second group of questions from Test I shows the core view of the Flexible-PLA model of the home automation systems SPL (see Figure 8.9). In this group of questions, the students were questioned about components, PPCs and connections of the model to check if they understood the meaning of these concepts. Hence, they were questioned about the modeling of components such as lighting control and intrusion detector, and PPCs such as door lock and surveillance.

![Figure 8.9: FPLA model - Core View of home automation systems SPL](image)

The third group of questions from Test I shows the variability view of the Flexible-PLA model of the home automation systems SPL (see Figure 8.10). In this group of questions, the students were questioned about variability points, variants and weavings of the model to check if they understood the meaning of these concepts. Hence, they were questioned about the modeling of variability points, such as the lock type and their variants — keypad and fingerprint scanner — and the surveillance device and their variants — motion sensors and cameras.

The fourth group of questions from Test I shows the service & interface view of the Flexible-PLA model of the home automation systems SPL (see Figure 8.11). In this group of questions, the students were questioned about services and interfaces of the model to check if they understood the meaning of these concepts. Hence, they were questioned about the modeling of services, such as lock, authentication, set alarm, etc.

Finally, the fifth group of questions from Test I shows the weaving view of the Flexible-PLA model of the home automation systems SPL (see Figure 8.12). In this group of questions, the students were questioned about weavings, pointcuts and advices of the model to check if they understood the meaning of these concepts. Hence, they
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

Figure 8.10: FPLA model - Variability View of home automation systems SPL

![Figure 8.10: FPLA model - Variability View of home automation systems SPL](image)

Figure 8.11: FPLA model - Service & interface definition of home automation systems SPL

![Figure 8.11: FPLA model - Service & interface definition of home automation systems SPL](image)
were questioned about the modeling of the weavings to inject the services of variants, such as key pad or fingerprint scanner into the PPC doorlock, depending on whether the lock type of a specific home automation system requires an advanced configuration or a basic configuration.

Figure 8.12: FPLA model - Variability View (weaving definition) of home automation systems SPL

8.3.2.2 Test I: Results

The results from Test I are shown in Figure 8.13 in a tabular format. Each row is related to a question (corresponding to one of the five groups mentioned above) and the columns entitled Student1 to Student8 show whether the student’s answer was right or not (values 1 and 0, respectively). The last column shows the average of the questions answered correctly. These data reveal that several questions regarding the concepts of PPCs, Service, ServiceForDerivation, and Weaving present the lowest averages (see Figure 8.13: Group 2, questions D and E; Group 4, questions C and I; and Group 5, question E). This means that these questions reveal the most difficult concepts at the time of understanding their meaning in a specific view of a model. By grouping the concepts and calculating the averages among all questions regarding the same concept, Figure 8.14 graphically shows the concepts that presented more difficulties at the time of understanding their meaning in the complete model. Hence, as shown in Figure 8.14 the PPC, Pointcut & Advice and ServiceForDerivation concepts are the least understandable. For instance, a generic mistake among students was that
PPCs were conceived as optional components, or when the pointcut and advice concepts were often exchanged.

**Figure 8.13: Test I.a - Understandability of Flexible-PLA models. Results from eight post-graduate students for each question**

Figure 8.15 shows for each group of questions the average of the questions answered correctly by every student (see last column of the table shown in Figure 8.15), for each student the average of the questions answered correctly from Test I (see the seventh row of the table shown in Figure 8.15) and other relevant information such as work experience, the highest position in the last job or age (see the last rows of the table shown in Figure 8.15). From these data, several conclusions can be drawn:

- Weaving is the view that presents the most difficulties at the time of understanding every concept that this view offers to modelers, coinciding with the understanding of the concepts of pointcut and advice (see Figure 8.16).
8.3 The FPLA Modeling Framework

Figure 8.14: Graph I - Understandability of Flexible-PLA models: concepts

Figure 8.15: Test I.b - Understandability of Flexible-PLA models. Results from eight post-graduate students for each group of questions

Figure 8.16: Graph II - Understandability of Flexible-PLA models: views
The students with more than five years of work experience achieved an average of 80% of the questions answered correctly. They understood the concepts of Flexible-PLA models much better (see Figure 8.17). These results reveal that the understandability of Flexible-PLA models depends on the variable work experience.

Figure 8.17: Graph III - Understandability of Flexible-PLA models depending on the expertise

8.3.2.3 Test II: Description

After finishing Test I the students started to answer Test II. Test II evaluates the usability of the FPLA modeling framework in terms of easy to learn and easy to use, as described by [Rubin, 1994]. The students were provided with computers with the FPLA modeling framework installed. Test II (see Appendix C) consists of four exercises for modeling the PLA of the banking systems SPL that is described in Chapter 4. Each of these exercises focuses on a specific view —core, variability, service & interface, and weaving views— all of which are necessary to completely describe the PLA of the banking systems SPL. Each exercise starts with a description of what has to be modeled according to the specifications of the banking systems SPL. Next, the students use FPLA for modeling the given specifications. Finally, the students answer a set of Yes/No questions which evaluate the usability in terms of (i) easy to learn (ex. whether icons and graphical notation used by FPLA for modeling PLAs are representative and easy to find and identify), and (ii) easy to use, i.e. in terms of achievement of the tasks necessary to model the proposed PLA (ex. whether the components were modeled according to the specifications of the banking systems SPL).
8.3.2.4 Test II: Results

The results from Test II are shown in Figure 8.18 in a tabular format. Each row is related to a question (corresponding to one of the four exercises mentioned above) and the columns entitled Student1 to Student8 show whether the student evaluated the usability of the FPLA modeling framework positively or negatively (values 1 and 0, respectively). The average of the questions that students positively evaluated can be found in last column of the table shown in Figure 8.18. In addition, for each exercise there is a row that shows how long it took students to model the specifications and answer the questions.

![Figure 8.18](image)

**Figure 8.18: Test II.a - Usability of the FPLA modeling framework. Results from eight post-graduate students**

Figure 8.19 shows for each exercise the average of the questions answered correctly by every student (see last column of the table shown in Figure 8.19). This table also shows, for each student, the average of the questions answered correctly from Test II (see the sixth row of the table shown in Figure 8.15), and the total time students spent in modeling the specifications and answer the questions. Finally, other relevant information is shown, such as work experience, the highest position in the last job or age (see the last rows of the table shown in Figure 8.19). From these data, several conclusions can be drawn:
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

![Figure 8.19: Test II.b - Usability of the FPLA modeling framework. Results from eight post-graduate students for each exercise](image)

- Usability, in terms of easy to learn, is 100% positively evaluated except for the concepts of *services for derivation*, *pointcuts* and *advices*. In these cases several students had difficulties in finding and identifying the icons for modeling these concepts (see Figure 8.18: Exercise 3 question B, and Exercise 4 questions B and C).

- Weaving and variability are the views that took longer for students (see Figure 8.20). These data reveal that these views are also the most difficult to learn at the time of modeling the proposed PLA.

- Figure 8.21 shows the relationship between the time students spent in modeling the specifications and answer the questions (see Y axis) and their experience (see X axis). These data reveal that there is no relationship. Therefore, the learnability of FPLA modeling framework is not affected by users’ experience.

- Usability, in terms of easy to use, is also 100% positively evaluated except for the concepts of *services for derivation*, *pointcuts* and *advices*. Like Test I, the greatest difficulties that students had were the modeling of *pointcuts* and *advices* (see Figure 8.18: Group 4, questions D and E), followed by the modeling of *services for derivation* (see Figure 8.18: Group 3, questions C and D) where tasks were not completed by 82% of the students, specifically in the case of modeling pointcuts and advices.

- Figure 8.22 shows the relationship between the percentage of tasks that were completed by students (see Y axis) and their experience (see X axis). These data reveal that there is no relationship. Therefore, the usability of FPLA modeling framework is not affected by users’ experience.
8.3 The FPLA Modeling Framework

Figure 8.20: **Graph IV** - Usability of the FPLA modeling framework. Time spent in modeling each view

Figure 8.21: **Graph V** - Usability (easy to learn) of the FPLA modeling framework depending on the expertise
8. AGILE PRODUCT-LINE ARCHITECTING PROCESS

8.3.2.5 Conclusions & Limitations of the FPLA Empirical Assessment

The results from Test I and II have allowed FPLA’s developers to identify the issues when using FPLA and understanding its models. The results from Test I show that it may be necessary to improve the understandability of Flexible-PLA models for those users with low work experience. The results from Test II reveal the positive results of usability, in terms of being easy to learn and easy to use, for most concepts implemented by the FPLA modeling framework. However, it may be necessary to improve the usability of those concepts related to the weaving process, including the modeling of pointcuts and advices.

Regarding the limitations of this empirical assessment, it is necessary to mention the time restriction when doing the tests. This restriction affects the number of questions that were not answered by the students, which does not mean these questions wouldn’t have been answered correctly without this restriction (see Figure 8.18 Exercise 4 questions D and E).
8.4 Conclusions

This chapter presented the APLA process whose goal is to assist and guide agile architects in the decision-making process at the time of adding or changing features by: (i) describing PLAs which are easily adaptable iteration after iteration, (ii) tracking the effects of changes upon the architecture —components and connections which are impacted by the changes—, and (iii) analyzing architectural concerns, such as dependencies on earlier design decisions, rationale, constraints, risks, etc., which are also impacted by changes. This analysis enables the preservation of the integrity of the architecture, as long as it helps to reduce the risk of unexpected impacts from changes. The analysis is instrumented by a CIA technique and supported by the FPLA modeling framework.

The agile architecting process, on which the APLA process is based, has been published in:

Part V

CASE STUDY
Chapter 9

Case Study

This chapter aims to provide empirical evidence that validates the achievement of the objectives of this thesis. In this regard, the mechanisms that this thesis proposes to satisfy the objectives and to meet the main goal of the thesis are evaluated. Case study research is a technique that consists of the investigation of contemporary phenomena in their natural context [Yin 2008] to search for evidence, gain understanding, or test theories by primarily using qualitative analysis [Runeson & Höst 2009]. Since most of the objectives to be validated in the context of this thesis are qualitative, this thesis uses the case study technique to search for evidence to see if the proposed mechanisms really do satisfy these objectives. This chapter presents the case study which has been conducted in an experimental i-smart software factory (iSSF [Martin et al. 2010]). The case study was performed within an industrial project in the energy power networks domain where the FPLA Modeling Framework was used. The author and directors of this thesis have been involved since 2011 with this particular investigation.

This chapter briefly describes the running environment where the case study has been conducted and reports the case study according to the guidelines for conducting and reporting case study research in software engineering by [Runeson et al. 2012].
9. CASE STUDY

9.1 Context: the i-smart software factory

This section briefly describes the running environment in which the case study has been conducted: an i-smart software factory (iSSF [Martin et al., 2010]) which is deployed in the Technical University of Madrid (UPM) and Indra Software Labs.

The iSSF is a software engineering research and education setting in close cooperation with the top industrial and research collaborators in Europe. It is a global and distributed software development initiative set up at the end of 2011. Indra Software Labs leads this initiative at the corporate level in Spain, in conjunction with UPM, although it is framed into a broader-scope that includes other software factories such as that located at the Univ. of Helsinki, Univ. of Eastern Finland and Univ. of Bolzano and companies such as Tieto, and Indra in Spain.

The iSSF aims to create models and tools that will contribute both toward the implementation of the new processes and methodologies, and the monitoring and tracking of the results. As it has been above mentioned, the iSSF in which this case study has been run, comprises laboratories in two different geographical locations in Madrid (UPM and Indra’s factories), equipped with sophisticated computer and monitoring equipment. This equipment facilitates tracking of the project’s progress using real-time data from development tools. The iSSF facility continuously runs projects in eight week cycles. This case study focused on one of these projects which was ran in the iSSF, specifically in a project to develop a metering management system in electric power networks.

9.2 Reporting the case study

Runeson et al. [2012] establish a set of steps to conduct and report a case study. They are the following:

1. Case study design: Objectives are defined and the case study is planned.
2. Preparation for data collection: Procedures and protocols for data collection.
3. Collecting evidence: Execution with data collection on the studied case.

---

1http://www.upm.es/internacional
2http://www.indracompany.com/en
4. Analysis of collected data.
5. Reporting.

The goal of reporting a case study is twofold: to communicate the findings of a study, and to work as a source of information for judging the quality of the study. With this twofold goal, the reporting of the case study is described as follows.

9.2.1 Introduction

This section summarizes the problem statement and the research objectives which are evaluated in this case study. Each one of these objectives defines a mechanism that intends to satisfy the capabilities that the APLA process should provide in order to meet the main goal of this thesis. The aim of the case study described is to provide empirical evidence of the achievement of these objectives in the context of the iSSF, i.e. evaluate each one of the mechanisms provided.

9.2.1.1 Problem statement

As described in Chapter [1] Agile Product Line Engineering (APLE) aims to reduce the upfront analysis and design as practiced in SPLE, and to make the development of product-lines more flexible and adaptable to changes as promoted in ASD. The main challenge to put APLE into practice consists of designing and evolving product-line architectures while getting close to the “be open to change” agile principle. To address this challenge, it is necessary to count on mechanisms to assist and guide the agile construction and evolution of architectures, specifically in the following three architecting tasks: (i) defining flexible and adaptive product-line architectures that facilitate the change, (ii) providing architects with knowledge that assists and guides them in the change decision-making process, and (iii) reducing the risk of unexpected consequences of changes and enabling the preservation of the architecture integrity, iteration after iteration during the APLE development process.
9. CASE STUDY

9.2.1.2 Research objectives

The main goal of this thesis is precisely to define a process and supporting mechanisms for “the agile construction and evolution of product-line architectures”, i.e. 
*agile product-line architecting (APLA)*. The APLA process intends to provide agile architects with the following capabilities:

- **Flexibility & adaptability** at the time of defining software architectures through mechanisms that facilitate change during the incremental and iterative design of PLAs (anticipated or planned changes), as well as their evolution (unanticipated or unforeseen changes).
- **Assistance in checking** architectural integrity at the time of constructing and evolving PLAs through mechanisms that facilitate change impact analysis in terms of architectural concerns such as dependencies on earlier design decisions, rationale, constraints, and risks, etc which may be impacted by change.
- **Guidance in the change decision-making process** at the time of constructing and evolving PLAs through mechanisms that facilitate change impact analysis in terms of architectural components and connections which may be impacted by change.

The specific objectives that have been addressed to achieve the abovementioned capabilities are defined as follows (excerpt from Chapter 1, Section 1.3):

**OBJ 1.** To define a mechanism for describing variability of PLAs in which variants are independent of the linking context. The aim of this mechanism is to provide software architecture with flexibility and adaptability.

**OBJ 2.** To define a mechanism for documenting product-line architectural knowledge and tracing architecturally significant features during their realization in the PLA. The aim of this mechanism is to provide knowledge to help preserve the integrity of the architecture.

**OBJ 3.** To define a mechanism for traversing PLA descriptions, architectural knowledge documentation, and traceability definitions. The aim of this mechanism is to analyze and determine the potential impact upon the architecture resulting from the implementation of a change in features in order to provide guidance in the change decision-making process.
9.2 Reporting the case study

**OBJ 4.** To automate or semi-automate the previous objectives by tool support. The aim of the tool is to provide modeling primitives for describing, documenting and tracing PLAs, as well as the algorithm and transformation functionalities for automating change impact analysis and code generation, during the iterative and incremental construction and evolution of product-line architectures.

Chapter 1 (see Section 1.3) described the criteria to validate the achievement of each specific objective. The case study described evaluates the mechanisms resulting from each specific objective based on these criteria, with the aim of searching empirical evidence that validates whether these mechanisms satisfy the capabilities that the APLA process should provide to meet the main goal of this thesis. These mechanisms are materialized as follows (excerpt from Chapter 1, Section 1.4):

- The Flexible-PLA metamodel
- The PLAK metamodel
- The techniques for analyzing change impact
- The FPLA modeling framework

9.2.2 Case study design

This section describes the case study that has been designed to provide empirical evidence that validates the achievement of the objectives of this thesis. The pilot study that is described in Chapter 4 was previously conducted to help prepare this case study design (see also Chapters 5-7, Sections 5.4, 6.3 and 7.3 respectively), as well as elicit the research questions, along with their dependent, independent and hidden variables.

9.2.2.1 Research Questions

A case study searches for evidence for formal propositions or hypotheses by collecting data regarding events happening in a real life setting. The hypotheses of our case study are derived from each specific objective of this thesis. As previously mentioned, these objectives define a set of mechanisms that claim to satisfy the capabilities that the APLA process should provide to meet the main goal of this thesis. To validate these claims and hypotheses each of the mechanisms has to be evaluated. To this aim,
these claims and hypothesis are refined into a set of research questions to be answered through the cases study analysis. These research questions are formulated around the criteria defined in Chapter 1 (see Section 1.3). The research questions are formulated as follows:

$RQ_1$: Do Flexible-PLA modeling primitives effectively provide the required expressiveness for describing PLAs, including internal and external variability?

$RQ_2$: Are Flexible-PLA modeling primitives effective in providing architectures with flexibility and adaptability while agile architecting?

$RQ_3$: Do PLAK modeling primitives effectively provide the required expressiveness for documenting PLAs, including the rationale of internal and external variability and variability dependencies?

$RQ_4$: Do Flexible-PLA & PLAK modeling primitives effectively provide the required expressiveness for tracing features with PLA models, including variability traces?

$RQ_5$: Do PLAK models assist and guide architects in preserving the architecture integrity while agile product-line architecting?

$RQ_6$: Is the CIA algorithm effective in locating the impacted architectural design decisions and elements resulting from a proposed change in features?

$RQ_7$: Does the CIA algorithm assist and guide agile architects in reasoning about the changes while agile product-line architecting?

$RQ_8$: Is the FPLA modeling framework usable in terms of easy to learn?

$RQ_9$: Is the FPLA modeling framework usable in terms of easy to use?

$RQ_{10}$: Are models constructed with the FPLA modeling framework understandable?

Table 9.1 summarizes the intended research questions to be answered through the case study analysis described. However, during the planning of the case study, it was observed that several of these questions may be better answered through the pilot study previously conducted to prepare the case study. The reason is that the answer to these questions does not require a real life setting, but instead an empirical assessment closer to experimental settings where it is possible to have a high level of control over the variables (e.g. experience of the subjects). The research questions which obtain evidence from the pilot study are: $RQ_1$, $RQ_3$ and $RQ_4$ regarding the expressiveness of
Flexible-PLA and PLAK models, and $RQ_8$, $RQ_9$ and $RQ_{10}$ regarding the usability of the FPLA modeling framework and the understandability of its models.

Next, the research questions are analyzed in detail. For each question, its dependent variables (variables being explained) and independent variables (variables which affect dependent variables) are identified and discussed.

Research questions $RQ_1$, $RQ_3$ and $RQ_4$ aim to find out if Flexible-PLA and PLAK models provide the required expressiveness for describing, documenting and tracing the architectural knowledge of PLAs. Expressiveness can be defined in terms of how and in which level the tasks of describing, documenting, and tracing the architecture are supported by the modeling primitives that the Flexible-PLA & PLAK Metamodels provide. The level in which these tasks are supported by the modeling primitives is explored through questions asked and comments expressed by a set of modelers after describing, documenting and tracing a proposed PLA. The potential independent variable that might have an influence on the dependent variable is the architect experience.

Research question $RQ_2$ aims to find out if Flexible-PLA models effectively provide architects with mechanisms to achieve flexibility and adaptability while agile architecting, i.e. if Flexible-PLA models facilitate change during the incremental and iterative construction and evolution of PLAs. Two dependent variables are defined for $RQ_2$: First, this case study quantitatively validates if the flexibility given by PPCs and variability points provides the ability to incorporate those changes that can be anticipated and planned to the resulting architecture of this case study. Second, this case study quantitatively validates if the adaptability given by PPCs and the weaving mechanism provides the ability to incorporate those changes that cannot be anticipated —unforeseeable— to the resulting architecture of this case study. Therefore, the dependent variables are the number of planned changes that the architecture is able to incorporate and the number of unforeseeable changes that the architecture is able

### Table 9.1: Case study - Summary of research questions

<table>
<thead>
<tr>
<th>Thesis Objective</th>
<th>Mechanism (Result)</th>
<th>Research Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJ 1.</td>
<td>Flexible-PLA metamodel</td>
<td>$RQ_1$ &amp; $RQ_2$</td>
</tr>
<tr>
<td>OBJ 2.</td>
<td>PLAK metamodel</td>
<td>$RQ_3$, $RQ_4$ &amp; $RQ_5$</td>
</tr>
<tr>
<td>OBJ 3.</td>
<td>CIA algorithm</td>
<td>$RQ_6$ &amp; $RQ_7$</td>
</tr>
<tr>
<td>OBJ 4</td>
<td>FPLA Modeling Framework</td>
<td>$RQ_8$, $RQ_9$ &amp; $RQ_{10}$</td>
</tr>
</tbody>
</table>
to incorporate. The potential independent variable that might have an influence on the two dependent variables is the total number of changes that the architecture must incorporate to meet the requirements. Additionally, the appreciation of architects by comparing the flexibility and adaptability of our approach regarding the use of conventional modifications of the architecture configuration and refactoring, although subjective, is also evaluated through interviews and surveys.

Research question $RQ_5$ aims to find out if PLAK models effectively facilitate the maintenance of the integrity of the architecture, i.e. if PLAK models provide knowledge to assist and guide agile architects in trying to preserve architectural integrity iteration after iteration. As a dependent variable, the level of assistance and guidance to preserve the architectural integrity of PLAK models is qualitatively estimated by analyzing questions asked to architects through a set of interviews. These questions ask architects about specific situations in which the assistance and guidance of PLAK models is analyzed. Hence, the architects are asked if PLAK models help them when trying to maintain the architectural integrity when changes are implemented. Specifically, the questions are focused on dependencies, assumptions, tradeoffs, or constraints of those design decisions which may be in conflict to a proposed change, and therefore could jeopardize the integrity of the architecture. The potential independent variables which might have an influence on the dependent variable are the architects experience, the project size, and the architecture complexity.

Research question $RQ_6$ aims to find out if the CIA algorithm that this thesis proposes effectively determines the impact on the architecture that results from a proposed change in features. As a dependent variable, the level of effectiveness is quantitatively evaluated by counting the number of components that are impacted by a proposed change. First the architects analyzed change impact without the assistance of the CIA technique and PLAK models. Then, the architects analyzed change impact with the assistance of the FPLA modeling framework that implements the CIA algorithm. This procedure allowed to verify if the CIA algorithm had determined impacts that architects manually did not, or vice versa if the architects had manually determined impacts that the CIA technique did not. From this procedure, it is possible to define the dependent variable for quantitatively measuring $RQ_5$: the percentage of impacts that the CIA algorithm automatically determines that were not manually determined by architects. The potential independent variable that might have an influence on the dependent variable is the total number impacts that exist given the proposed change(s) and the architects experience.
Research question \( RQ_7 \) aims to find out if the CIA algorithm effectively assists and guides agile teams in making decisions about the changes in the iterations of an agile process. As a dependent variable, the level of assistance and guidance in making decisions about the changes is qualitatively estimated by analyzing questions asked to architects. These questions ask about specific situations where the assistance and guidance in the change decision-making process of the CIA algorithm is analyzed. Hence, the architects are asked if the change-impact knowledge provides guidance in reasoning about changes in features. This may be the case of recognizing the dependencies between design decisions that could reveal hidden ripple-effect changes. The potential independent variables that might have an influence on the dependent variable are the architects experience, the project size, and the architecture complexity.

Research questions \( RQ_8 \), \( RQ_9 \) and \( RQ_{10} \) aim to find out if the FPLA modeling framework is usable and if its models are understandable. Usability can be defined in terms of easy to learn and easy to use [Rubin, 1994] whereas the understandability can be defined as the users capabilities to interpret the mean of the concepts modeled in Flexible-PLA models. The potential independent variable that might have an influence on the dependent variable is the architect experience.

Table 9.2 summarizes dependent and independent variables for each research question answered through this case study. It is necessary to mention that it is in the nature of case studies that independent variables cannot be controlled [Runeson & Höst, 2009]. This and other potential threats to validity are discussed in Section 9.2.3.3.

9.2.2.2 Data collection procedure

Both quantitative and qualitative data are gathered. The collection methods which have been used are described as follows:

- Observation. Two observers follow the tailored Scrum that was described in Figure 8.2 in which the APLA process is deployed. These observers attend the sprint agile architecting and planning meetings, review meetings, and visit the developing team twice week. They take notes from these meetings and, thanks to the iSSF technologies, agile architecting meetings are video recorded, transcribed, and analyzed using the constant comparison method as described in [van Heesch et al., 2012].
## 9. CASE STUDY

Table 9.2: Case study - Summary of research validation

<table>
<thead>
<tr>
<th>Research Question</th>
<th>Dependent Variable</th>
<th>Independent Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$RQ_1$</td>
<td>Expressiveness.</td>
<td>Architects experience.</td>
</tr>
<tr>
<td>$RQ_2$</td>
<td>Number of planned changes that are incorporated to the architecture. Number of unforeseeable changes that are incorporated to the architecture.</td>
<td>Total number of changes that the architecture must incorporate to meet the requirements.</td>
</tr>
<tr>
<td>$RQ_3$</td>
<td>Expressiveness.</td>
<td>Architects experience.</td>
</tr>
<tr>
<td>$RQ_4$</td>
<td>Expressiveness.</td>
<td>Architects experience.</td>
</tr>
<tr>
<td>$RQ_5$</td>
<td>Assistance and guidance in preserving the architecture integrity.</td>
<td>Architects experience, project size, and architecture complexity.</td>
</tr>
<tr>
<td>$RQ_6$</td>
<td>Percentage of impacts that the CIA algorithm automatically determines and that were not manually determined by architects.</td>
<td>Total number impacts given a proposed change. Architects experience.</td>
</tr>
<tr>
<td>$RQ_7$</td>
<td>Assistance and guidance in making decisions about the changes.</td>
<td>Architects experience, project size, and architecture complexity.</td>
</tr>
<tr>
<td>$RQ_8$</td>
<td>Easy to learn.</td>
<td>Architects experience.</td>
</tr>
<tr>
<td>$RQ_9$</td>
<td>Easy to use.</td>
<td>Architects experience.</td>
</tr>
<tr>
<td>$RQ_{10}$</td>
<td>Understandability.</td>
<td>Architects experience.</td>
</tr>
</tbody>
</table>

- **Questionnaire and Interview.** The architects are interviewed following a questionnaire\(^1\) open to the discussion. Architects’ opinions on the object under study are collected and their questionnaire answers are analyzed. These interviews are video recorded, transcribed, and analyzed using the constant comparison method.

- **Archival data.** In addition to the video recordings, the information about the project is collected in Redmine\(^2\).

- **Analysis of work artifacts.** The data of Flexible-PLA and PLAK models generated with the FPLA modeling framework, the CIA algorithm running under FPLA, and the code under subversion\(^3\) are gathered.

- **Metrics.** The metrics captured by Sonar\(^4\) are collected at the end of each sprint.

---
\(^1\)The script of the interview is available on [https://www.surveymonkey.com/s/TSYCCN6](https://www.surveymonkey.com/s/TSYCCN6)
\(^2\)Redmine is web-based project management and bug-tracking tool [http://www.redmine.org/](http://www.redmine.org/)
\(^3\)Subversion is an open source version control system [http://subversion.apache.org/](http://subversion.apache.org/)
\(^4\)Sonar is an open platform to manage code quality [http://www.sonarsource.org/](http://www.sonarsource.org/)
9.2 Reporting the case study

Table 9.3 summarizes the data collection methods and data sources used to answer each research question, as well as when they were used.

Table 9.3: Case study - Data collection methods and data sources

<table>
<thead>
<tr>
<th>Data collection method</th>
<th>Session</th>
<th>Data source</th>
<th>Research Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interviews</td>
<td>Pilot study</td>
<td>Post-graduate students</td>
<td>$RQ_1, RQ_3, RQ_4$</td>
</tr>
<tr>
<td>Analysis of work artifacts</td>
<td>Pilot study</td>
<td>Flexible-PLA &amp; PLAK models</td>
<td>$RQ_1, RQ_3, RQ_4$</td>
</tr>
<tr>
<td>Analysis of work artifacts</td>
<td>After each sprint</td>
<td>Flexible-PLA models</td>
<td>$RQ_2$</td>
</tr>
<tr>
<td>Metrics</td>
<td>After each sprint</td>
<td>Sonar</td>
<td>$RQ_2$</td>
</tr>
<tr>
<td>Interviews</td>
<td>After each sprint</td>
<td>Architect</td>
<td>$RQ_5$</td>
</tr>
<tr>
<td>Observation</td>
<td>Sprints</td>
<td>Architect</td>
<td>$RQ_5$</td>
</tr>
<tr>
<td>Analysis of work artifacts</td>
<td>After sprints 3-6</td>
<td>Change impact analysis</td>
<td>$RQ_6$</td>
</tr>
<tr>
<td>Interviews</td>
<td>After sprints</td>
<td>Architect</td>
<td>$RQ_7$</td>
</tr>
<tr>
<td>Observation</td>
<td>Sprints</td>
<td>Architect</td>
<td>$RQ_7$</td>
</tr>
<tr>
<td>Questionnaire (Tests)</td>
<td>Pilot study</td>
<td>Post-graduate students</td>
<td>$RQ_8, RQ_9, RQ_{10}$</td>
</tr>
<tr>
<td>Interviews</td>
<td>Pilot study</td>
<td>Post-graduate students</td>
<td>$RQ_8, RQ_9, RQ_{10}$</td>
</tr>
</tbody>
</table>

9.2.2.3 Analysis & Validity procedure

In this case study, both quantitative and qualitative analysis were used to examine the data gathered. For quantitative data, this case study uses analysis of descriptive statistics. For qualitative data, the procedure to explore the chain of evidence [Runeson & Höst, 2009] from collected data is described as follows: Interviews and meetings are recorded, transcribed, grouped by quotes and coded. Coding means that parts of the text are given a code representing a certain topic of interest — one code is usually assigned to many pieces of text, and one piece of text can be assigned more than one code and codes can form a hierarchy of codes and sub-codes [Runeson & Höst, 2009].
9. CASE STUDY

The coded material is enriched with comments and reflections (i.e. memos). From this material it is possible to identify evidence that answers one or more research questions.

As data gathered in case studies is mainly qualitative [Runeson & Höst, 2009], and it is typically less precise than quantitative data, it is important to use triangulation to increase the precision of the study. There are several types of triangulation [Stake, 1995]: (i) methodological triangulation, i.e. the use of different methods to measure the same concern e.g. interviews, observations, or the analysis or archival data; (ii) data source triangulation, i.e. the use of multiple data sources at potentially different occasions; and (iii) observer triangulation, i.e. the use of more than one observer in the case study [van Heesch et al., 2012]. The three types of triangulation were used in this case study. For instance, Table 9.3 shows the application of methodological triangulation for all research questions which are qualitatively evaluated (see RQ1, RQ3, RQ4, RQ2, RQ5 and RQ7). Data source triangulation is applied by interviewing architects both separately and together. Finally, observer triangulation is applied by replicating specific data collection sessions by two different observers.

9.2.2.4 Case study description

The case study consists of a project to develop a metering management system in electrical power networks. The metering management system is part of two larger ITEA2 projects called IMPONET¹ and NEMO&CODED² and a third national project called ENERGOS³. The first one focuses on supporting complex and advanced requirements in energy management, specifically electric power networks which are envisioned like smart grids (see Figure 9.1). IMPONET aims for (i) continuous monitoring and bi-directional communication with customers to promote sustainability, and (ii) prevention of congestions, faults, and peak loads in real-time. NEMO&CODED focuses on the modelling, design, implementation and operation of networked hardware/software smart devices for the electrical distribution application domain and IT platform management domain. Finally, ENERGOS focuses on providing the real-time platform to integrate and manage the operations from those stakeholders involved in smart grids (e.g. consumers, producers, etc.).

¹Intelligent Monitoring of Power NETworks http://innovationenergy.org/imponet/
²NEtworked MOntoring & COntrol, Diagnostic for Electrical Distribution http://innovationenergy.org/nemocoded/
³Technologies for automated and intelligent management of power distribution networks of the future http://www.indracompany.com/sostenibilidad-e-innovacion/proyectos-innovacion/energos-technologies-for-automated-and-intelligent-
Figure 9.1: Smart grid overview Source: Overview of Microgrid research and development activities in the EU. Manuel Sánchez, European Commission, Symposium on Microgrids, Montreal 2006.
9. CASE STUDY

The metering management system captures and manages meter data from a huge number of distributed energy resources. It validates, stores and processes these meter data, and provides them to external systems. Figure 9.2 shows the high level architecture of the metering management system and the external systems. The metering management system communicates to the external system to capture or provide metering data. The overview of the system functionality is as follows:

1. Meter capturing. It integrates all meter capturing processes which are currently being supported by telemetering systems and batch processes that collect measurements at substations (see box Input in Figure 9.2). The purpose is to have a single database with the energy metering data (see Meter Capturer in Figure 9.2).

2. Meter processing. It includes three operations: validation of meter data according to an established validation formula, calculation of the optimal vector for a measuring point for a type and period of energy data, and estimation of energy data according to a established estimation formula (see Meter Processor in Figure 9.2).

3. Meter providing. It defines the interface (see Meter Provider in Figure 9.2) with information client systems to provide exchanging data with other information systems, such as billing and settlements, energy demand forecast, and energy purchases (see box Output in Figure 9.2).

![Figure 9.2: Metering management system - An overview and interfaces with external systems](image)

This case study focuses on the central box: the metering management system which is developed in the project OPTIMETER. The project OPTIMETER has been conducted in the iSSF. The goal of the project OPTIMETER is to provide metering management with real time data processing. To select the data storing technology, it is necessary to account for the performance in loading the large amounts of energy data coming from the meter capturing processes, as well as the performance in querying
these data. In this regard, OPTIMETER started with just one of the large data storing technologies shown in Figure 9.3, specifically the object-oriented NoSQL database Oracle Big Data. However, later the project was conceived as the development of a family (SPL) of metering management applications, each of which is intended to support the different data storing technologies shown in Figure 9.3. The objective is to carry out various proof of concept of large data storing technologies to evaluate their performance. The need to be open to new data storing technologies makes OPTIMETER a valuable case study to apply the APLE principles, and specifically the APLA process that this thesis presents. The agile construction and evolution of the OPTIMETER PLA through the application of the APLA process and their supporting mechanisms is put into practice in this case study.

The OPTIMETER SPL is being iteratively and incrementally developed in the iSSF in Scrum subprojects of 8 sprints (1sprint = 2 weeks). The Scrum lifecycle followed by each one of these subprojects was described in Chapter 8. In that chapter, a tailored Scrum which deploys the APLA process was defined (see Figures 8.2, 8.5). This tailored Scrum provides a framework to put the mechanisms to be evaluated in this case study into practice: the Flexible-PLA and PLAK metamodels, the CIA algorithm and the FPLA modeling framework.
9. CASE STUDY

This case study focuses on the first Scrum subproject carried out in the iSSF in collaboration with the Indra Software Labs, which we refer to as Optimeter I. This project initially aimed to develop a metering management application running over the database manager Oracle Big Data: Hadoop over Berkeley DB. However, as it will be described in the case study execution, an unplanned change in customers needs made it necessary to change the database manager during the execution of Optimeter I.

9.2.2.5 Subject description

In total, 10 people participated in the project Optimeter I: four developers, two product owners, one scrum master (who performs both the tasks of the Scrum master and of a part-time architect), one full-time architect and two observers. The observers had access to all project information and collaborated directly with product owners and fellow team members.

9.2.3 Results

9.2.3.1 Case study execution: Optimeter I

This section describes the case study execution, i.e. the execution of Optimeter I according to the lifecycle shown in Figure 8.2. The description of the case study execution focuses on the agile construction and evolution of the OPTIMETER PLA, and specifically on following three architecting tasks that intend to (i) define flexible and adaptive architectures, (ii) make change decisions, and (iii) preserve the integrity of the architecture.

First of all, it is necessary to describe the requirements that the OPTIMETER PLA must meet. Hence, the first step of our tailored Scrum is the capture of requirements from the product vision (features). These features were described in detail by the product owners as follows:

1 Apache Hadoop is a framework for running applications on large clusters built of commodity hardware. http://hadoop.apache.org/


9.2 Reporting the case study

- **F1.** Meter reading. It consists of reading metering data associated with different energy resources, type of periods (quarterly, hourly, daily and monthly) and periods —intervals. Metering data is provided by text files for a specific date.

- **F2.** Meter storing. It consists of a large data store running over an object-oriented NoSQL database (specifically, Big Data Oracle running over Berkeley DB).

- **F3.** Meter data access. It consists of the initial data loading of historical metering data of one month and query of these data. Both loading and querying require to leverage a high performance through the use of clustering technologies (specifically Hadoop).

- **F4.** Meter data process. It consists of the algorithms for validating raw and optimal data, as well as calculating the optimal vector (integrated processing) of raw and optimal data. Namely, the energy data for a specific origin, period and date is retrieved and the system adds the energy data to obtain the energy data of the next period.

- **F5.** Graphical interface. It consists of the interface that provides the query of metering data.

Figure 9.4 graphically shows the abovementioned features through a hierarchical tree-like structure: the feature model. Then, these features were progressively decomposed into US and the product backlog was created, prioritized based on business value, and assigned to sprints. Sprints are described below:

![Feature Model Diagram]

**Figure 9.4: Optimeter I - Feature model**

**Sprint 1** focused on implementing feature F2 (Meter storing). The US planned for this sprint are the following: (i) installation and configuration of the database manager (Berkeley DB), and (ii) several conceptual proofs to create and access the database\(^1\). In

\(^1\)Some of the US here described are epics and needed to be decomposed although we prefer this abstraction level to hidden not relevant details.
9. CASE STUDY

this regard, the architects modeled the PLAK model\(^1\) that is shown in Figure \[9.5\]. This is the working architecture resulting from Sprint 1. It shows the components and design decisions that were made to realize the planned US for this sprint: The component DBManager implements the configuration of the Berkeley DB, the component Proof implements the access the database, and finally the design decision DD001 keeps the rationale behind using Berkely DB, eg. it ensures mass access to data, although the learning curve is high (see Figure \[9.5\]).

![Figure 9.5: Optimeter I - PLAK model: Sprint 1](image)

**Sprint 2** focused on implementing feature F1 (Meter reading) and partially implementing F3 (Meter data access). The US planned for this sprint are the following: (i) reading text files of metering data associated to different energy resources, type of periods (quarterly, hourly, daily and monthly) and periods, (ii) processing the previously read data to form pairs of key/value, (iii) data loading —pairs key/value— in the database, and (iv) data query. Following the tailored Scrum defined in Chapter 8, change impact analysis is carried out by architects to analyze the impact of adding the new features on the current working architecture —i.e. the working architecture from Sprint 1 (see Figure \[9.5\]). At this time, the working architecture of Sprint 1 was in too early of a stage that the CIA algorithm could not provide any relevant result that could have assisted architects in the decision-making process of adding these features.

Next, the architects modeled the PLAK model that is shown in Figure \[9.6\]. This is the working architecture resulting from Sprint 2. It shows the components, PPCs and design decisions that were made to realize the planned US for this sprint: The component MeterCapturer implements the reading text files of metering data and processing the previously read data to form pair of key/value. The PPCs DataLoader and

---

\(^1\)First, architects describe the architecture by using Flexible-PLA models, and then they document the architecture using PLAK models. PLAK models explicitly include Feature and Flexible-PLA models to traces features with their realization in the architecture.
9.2 Reporting the case study

DataQuery implement data loading and data query, respectively. As the functionalities for data loading and data query could not be completely implemented in this sprint, and increments in following sprints could refine these components, the architects decided to implement them using PPCs. The design decision DD002 keeps the rationale behind reading metering data in pairs of key/value. Finally, between DD002 and DD001 there is a dependency that keeps the relation between the need of reading data in pairs key/value and the use of the database manager BerkeleyDB.

Sprint 3 focused on completing feature F3 (Meter data access) and implementing feature F4 (Meter data process). The US planned for this sprint are the following: (i) high performance by using clustering, and (ii) initial operations over raw and optimal data to calculate the optimal vector. The impact analysis retrieved the design decisions and components which could have been impacted as a consequence of adding the new features on the current working architecture — i.e. the working architecture from Sprint 2 (see Figure 9.6). From Figure 9.4 and Figure 9.6, we can see that high performance is a subfeature of feature F3, and F3 has the subfeatures DataQuerying and DataLoading which are implemented by the PPCs DataQuery and DataLoader, respectively. The CIA algorithm retrieved these PPCs as potential candidates to be impacted by the feature increment high performance.

Next, the architects modeled the PLAK model that is shown in Figure 9.7. This is the working architecture resulting from Sprint 3. It shows the components, PPCs and design decisions that were made to realize the planned US for this sprint: The variant
(extension) HadoopMAP/REDUCE implements the operations for clustering and distributing work around a cluster in order to improve the performance for data accessing (data loading and data query). The PPCs DataLoader and DataQuery were extended with this functionality through the variability point clustering (see Figure 9.7). The design decision DD005 keeps the rationale behind clustering as well as a dependency with the design decision DD001 (see Figure 9.7). Finally, the PPC MeterProcessor implements initial operations over raw and optimal data to calculate the optimal vector. As the functionality for meter processing could not be completely implemented in this sprint and increments in following sprints might refine this component, the architects decided to implement it using a PPC.

Sprint 4 focused on completing feature F4 (Meter data process). The US planned for this sprint are the following: (i) validating metering data, and (ii) calculating optimal vectors. The impact analysis retrieved the design decisions and components which could have been impacted as a consequence of adding the new features on the current working architecture — i.e. the working architecture from Sprint 3 (see Figure 9.7). The CIA algorithm retrieved the PPC MeterProcessor as a potential candidate to be impacted by the change.

Next, the architects modeled the PLAK model that is shown in Figure 9.7. This is the working architecture resulting from Sprint 3. It shows the components, PPCs and design decisions that were made to realize the planned US for this sprint: the PPC MeterProcessor was extended by the algorithms for validating metering data.
and calculating optimal vectors through the variability points RawValidation, OptimalValidation, RawIntegrated and OptimalIntegrated (see Figure 9.8). These algorithms are implemented in the variants (extensions) shown in Figure 9.8 eg. RawEnergyDataValidation.

At this time, the project continued for four more sprints but two members of the Scrum team were substituted, specifically one of the architects and one of the developers.

**Sprint 5** focused on implementing feature F5 (Graphical interface). This feature has no a prior dependencies on others, so the CIA algorithm did not retrieve any impact (see Figure 9.8).

**Sprint 6**, the SPL owner required the following conceptual proof: a change of the database manager with the aim of evaluating the performance of other solutions. Specifically, the product owner decided to evaluate Oracle Real Application Clusters (RAC\(^1\)) over Oracle 11g (see the design decision DD006 in Figure 9.9). This meant there was an unplanned or unconsidered change in the SPL Backlog. The impact of this action is analyzed in detail in the following subsection.

\(^1\)Software for clustering and high availability in Oracle db environments
9. CASE STUDY

Next, the architects modeled the PLAK model that is shown in Figure 9.9. This is the working architecture resulting from Sprint 6. It shows the components, PPCs and design decisions that were made or changed to realize the planned US for this sprint: The variant (extension) RAC implements the operations for clustering and distributing work around a cluster in order to improve the performance for data accessing (data loading and data query). The PPCs *DataLoader* and *DataQuery* were extended with this functionality through the variability point *clustering* (see Figure 9.9). The design decision *DD007* keeps the rationale behind clustering as well as a dependency with the design decision *DD006* (see Figure 9.9).

![PLAK model: Sprint 6](image)

Figure 9.9: Optimeter I - PLAK model: Sprint 6

The above mentioned six sprints provided the necessary data to conduct the case study analysis and interpretation.

9.2.3.2 Analysis and interpretation

Quantitative and qualitative analysis was used to examine the data gathered during the case study. The data collected consisted of: the architectural models resulting from each sprint (see Figure 9.5-9.8), archival data from Redmine and Sonar, the output of executing CIA by using the FPLA modeling framework, as well as the questionnaires and interviews performed with the architects and a set of developers. Figure 9.10 shows...
9.2 Reporting the case study

several relevant data from the Sonar system that provide an overview of the size and complexity of the project Optimeter I. The analysis of these data has permitted to find evidence to answer each one of the research questions:

<table>
<thead>
<tr>
<th>Lines of code</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,547</td>
<td>114</td>
</tr>
<tr>
<td>11,396 lines</td>
<td>6 packages</td>
</tr>
<tr>
<td>3,632 statements</td>
<td>230 methods</td>
</tr>
<tr>
<td>74 files</td>
<td>+1,281 accessors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code coverage</th>
<th>Unit test success</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.3% ▲</td>
<td>100.0% ▲</td>
</tr>
<tr>
<td>13.1% line coverage ▲</td>
<td>0 failures ▲</td>
</tr>
<tr>
<td>15.1% branch coverage ▲</td>
<td>0 errors ▲</td>
</tr>
<tr>
<td></td>
<td>244 tests ▲</td>
</tr>
<tr>
<td></td>
<td>2.3 sec ▲</td>
</tr>
</tbody>
</table>

Figure 9.10: Optimeter I - Sonar dashboard - Optimeter can be considered a medium size project with 8Kloc

\( RQ_1: \) Do Flexible-PLA modeling primitives effectively provide the required expressiveness for describing PLAs, including internal and external variability?

\( RQ_3: \) Do PLAK modeling primitives effectively provide the required expressiveness for documenting PLAs, including the rationale of internal and external variability and variability dependencies?

\( RQ_4: \) Do Flexible-PLA \& PLAK modeling primitives effectively provide the required expressiveness for tracing features with PLA models, including variability traces?

The evidence to answer \( RQ_1 \) \( RQ_3 \) and \( RQ_4 \) is explored through questions asked and comments expressed by a set of post-graduate students who were enquired about the Flexible-PLA models and PLAK models resulting from executing the pilot study, —i.e. the description and documentation of the banking systems PLA. The models resulting from executing the pilot study are shown in Chapters 5 and 6 (see Sections 5.4, 6.3) whereas the evidence from the questionnaires is summarized in Chapters 6 (see Section 6.3).

227
9. CASE STUDY

**RQ₂:** Are Flexible-PLA modeling primitives effective in providing architectures with flexibility and adaptability while agile architecting?

The evidence to answer **RQ₂** is explored through descriptive statistics that measures the flexibility and adaptability of Flexible-PLA modeling primitives at the time of incorporating increments and changes while architecting the OPTIMETER PLA. Due to the fact that the architecture has to accommodate small increments and changes in each project sprint, we have measured if every change required in each sprint could be realized due to the flexibility and adaptability that is provided by the Flexible-PLA modeling primitives. In this regard, the working architecture resulting from each sprint, specifically the Flexible-PLA models, have been analyzed. Increments are considered as anticipated and planned changes, therefore we are going to evaluate if the flexibility given by PPCs and variability points was able to incorporate those increments from Sprints 1-5. However, in Sprint 6 the product owner required a change of the database manager with the intention of evaluating the performance of other solutions. This change could not be anticipated, it was unforeseeable, therefore we are going to evaluate if the adaptability given by PPCs and the weaving mechanism and its capability for changing variants were able to incorporate this change in Sprint 6.

For each sprint, Table 9.4 shows the total number of changes that the architecture had to incorporate to meet the requirements, the number of planned changes that were incorporated in the architecture by using PPCs and variability points, and the number of unforeseeable changes that were incorporated in the architecture by using PPCs and the weaving mechanism. Next, two examples of changes that were easily incorporated to the architecture due to these Flexible-PLA modeling primitives are described. The first one is a planned change, and the second one is an unforeseeable change.

During Sprint 3, it was planned to implement the feature high performance by using clustering. This consisted of refining the components `DataLoader` and `DataQuery`. The increment to refine `DataLoader` from Sprint 2 to Sprint 3 was easily implemented due to the fact that `DataLoader` was defined as a PPC (see Figure 9.6 and Figure 9.7). The high performance of the data loading planned for Sprint 3 was implemented by defining the variability point `clustering` and the variant `hadoopMAP/REDUCE`. Therefore, the PPC mechanism permitted to flexibly extend `DataLoader` through the use of variants and variability points.

During Sprint 6, there was a change of the database manager. The database manager that the SPL owner proposed was `Oracle 11g`. This database manager does not work over the clustering of Hadoop, but over `Oracle Real Application Clusters`.  

228
The architects were able to (i) incorporate a new database manager, i.e. Oracle 11g, (ii) incorporate a new variant that implemented the code for clustering over Oracle 11g (see Figure 9.9), (iii) define its weaving, and (iv) select this weaving for the new product and not the previous one defined for Hadoop. Therefore, the weaving of this new variant permitted to easily adapt the architecture to the new database manager. These changes were implemented in the PLAK model of Figure 9.9 and the new code was added to the new database manager component and the new variant. Due to the code generation capabilities of the FPLA Modeling Framework, the code of the new product was automatically regenerated.

Table 9.4: \( RQ_2 \): Descriptive statistics for the variables defined in Table 9.2

<table>
<thead>
<tr>
<th>Sprint</th>
<th>Total changes</th>
<th>Planned changes incorporated to the architecture</th>
<th>Unforeseeable changes incorporated to the architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint 1</td>
<td>2</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>Sprint 2</td>
<td>4</td>
<td>4</td>
<td>—</td>
</tr>
<tr>
<td>Sprint 3</td>
<td>2</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>Sprint 4</td>
<td>2</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>Sprint 5</td>
<td>1</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>Sprint 6</td>
<td>1</td>
<td>—</td>
<td>1</td>
</tr>
</tbody>
</table>

To answer \( RQ_2 \), in addition to the analysis of the Flexible-PLA models resulting from the abovementioned sprints, we evaluated the comments from the architect and a subset of developers. A set of interviews was conducted to evaluate the participants opinion of the flexibility and adaptability, although subjective, of our approach in regards to the sole use of conventional modifications of the architecture configuration and refactoring. The open format of the interviews allowed the architect and the developers to discuss the advantages of using Flexible-PLA modeling primitives versus conventional architecture modeling primitives. The discussion was based on analyzing the coupling and cohesion of the components that make up the architecture in two cases: (i) using PPCs and their extension mechanism, and (ii) not using them. To that end, the architects focused on (i) the working architecture of Figure 9.8 in which the Hadoop implementation has been encapsulated in a variant that extends the functionality of the PPCs DataLoader and DataQuery, and (ii) an alternative working architecture in which Hadoop had been encapsulated in an independent component. To measure the coupling and cohesion of the components, the architects relied on the number of dependencies between the components provided by the Sonar system. Table 9.5 shows the dependency matrix of components of the working architecture of Figure 9.8 whereas
Table 9.6 shows the dependency matrix of components of the alternative working architecture. From Table 9.6, the architects concluded that the dependencies between the components DataLoader and HadoopMAP/REDUCE and between DataQuery and HadoopMAP/REDUCE may make it difficult to implement the change that is proposed by the product owner in Sprint 6 (see Table 9.6: 24 and 35 dependencies, respectively). Therefore, the architects found evidences showing that the Flexible-PLA modeling primitives provide flexibility and adaptability while architecting the OPTIMETER PLA.

**Table 9.5:** Optimeter I - Sonar dependency matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>isl.optimeter.DBManager¹</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.DataLoader²</td>
<td>-</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.DataQuery³</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.MeterCapturer⁴</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.MeterProcessor⁵</td>
<td>19</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 9.6:** Alternative Optimeter - Sonar dependency matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>isl.optimeter.DBManager¹</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.DataLoader²</td>
<td>-</td>
<td>24</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.Hadoop³</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.DataQuery⁴</td>
<td>-</td>
<td>35</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.MeterCapturer⁵</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl.optimeter.MeterProcessor⁶</td>
<td>19</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RQ₆:** Is the CIA algorithm effective in locating the impacted architectural design decisions and elements resulting from a proposed change in features?

The evidence to answer **RQ₆** is explored through descriptive statistics that measures the effectiveness of the CIA algorithm in locating impacts on the OPTIMETER PLA given a change in features. Table 9.7 shows the total number of components which are impacted by those changes planned in Sprints 3-6, those impacts that are automatically identified by the CIA algorithm and those impacts that are manually identified by the architects. These data are analyzed below.

In Sprints 3 and 4, the CIA algorithm is 100% effective in locating the design decisions and components which are impacted by the feature increments that are planned.
9.2 Reporting the case study

during the sprint planning meetings. The complexity of the working architectures resulting from these first sprints is relatively low, in such a way that the architects also manually determined 100% of the impacted design decisions and components.

In Sprint 6 the architects ran the CIA algorithm to traverse the architectural model of Sprint 5 and retrieve the design decisions and components which could have been impacted by the change of database manager. The algorithm retrieved the PPCs \texttt{DataLoader} and \texttt{DataQuery}, the variability point \textit{clustering}, and the components \texttt{DBManager} and \texttt{MeterCapturer} as potential candidates to be impacted by the change (see row 2, Sprint 6 of Table 9.7). In this retrieval, two dependencies between architectural design decisions participate in the propagation of the change. These dependencies keep the relation of Hadoop and the data reading in pairs key/value with the database manager —Berkeley DB (see dependencies between \texttt{DD005} and \texttt{DD001}, and \texttt{DD002} and \texttt{DD001} in Figure 9.7). The manual analysis performed by the architects did not determine the propagation of the change. This is due to the fact that the dependencies between the \texttt{DD005} and \texttt{DD002} with \texttt{DD001} became hidden for the new architect who incorporated in Sprint 5 to the team, so that they only found that three components could have been impacted (see row 3, Sprint 6 of Table 9.7).

| Table 9.7: | RQ$_6$: Descriptive statistics for the variables defined in Table 9.2 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| | Sprint 3 | Sprint 4 | Sprint 5 | Sprint 6 |
| | Impacts | % over total | Impacts | % over total | Impacts | % over total | Impacts | % over total |
| Total number of impacts | 2 | — | 1 | — | 0 | — | 5 | — |
| CIA algorithm | 2 | 100% | 1 | 100% | 0 | 100% | 5 | 100% |
| Manual CIA | 2 | 100% | 1 | 100% | 0 | 100% | 3 | 60% |

Therefore, the CIA algorithm automatically retrieved two components which were not manually determined by the architects. Namely, the percentage of impacts that the CIA algorithm automatically determined, and that were not manually determined by the architects, is 40%. Hence, 40% of the DDs and components, which are impacted by the change, would have stayed hidden if the CIA algorithm had not worked effectively.

RQ$_5$: Do PLAK models assist and guide agile architects in preserving the architecture integrity while agile product-line architecting?

RQ$_7$: Does the CIA algorithm assist and guide agile architects in reasoning about the changes while agile product-line architecting?
9. CASE STUDY

*RQ*$_5$ and *RQ*$_7$ are jointly assessed by analyzing the interviews given to the architect and two developers. From these interviews, the following excerpts can be highlighted:

<<The use of PLAK models was particularly useful to understand the system during the staff turnover that took place between Sprints 4 and 5>>.

<<Without the knowledge provided by the PLAK model and the CIA algorithm, it would be extremely difficult to understand the impact of changing the database manager>>. It’s likely that several components had remained unchanged […] we had likely implemented the necessary functionality in other or new components, so the former would have had “dead code” >>. Dead code is an indicator of bad smell >>. This bad smell could have been identified after we had implemented the change by analyzing the dashboard of sonar platform >>. By using PLAK models and the CIA algorithm we were able to proactively determine all the impacts and to avoid this software degradation >>. We took previous design decisions into account among which there were dependencies >>. It did allow us to maintain the architectural integrity >>.

These excerpts from the architect and the two developers put in evidence that our approach for agile architecting assisted and guided architects in the change decision-making process and in the tasks of maintaining the architectural integrity iteration after iteration while they were architecting the OPTIMETER PLA.

*RQ*$_8$: *Is the FPLA modeling framework usable in terms of easy to learn?*

*RQ*$_9$: *Is the FPLA modeling framework usable in terms of easy to use?*

*RQ*$_{10}$: *Are models constructed with the FPLA modeling framework understandable?*

The evidence to answer *RQ*$_8$ *RQ*$_9$ and *RQ*$_{10}$ is explored through the empirical assessment that is described in Chapter 8 (see Section 8.3.2). Section 8.3.2 describes the tests used to conduct the empirical assessment and reports the evidence from these tests.

9.2.3.3 Evaluation of validity

Case studies are qualitative in nature. The objective judgment of the collected data of this kind is not possible. To improve the internal validity of the results presented,
the independent variables that could influence this case study have been identified as follows: The architect’s experience has a great influence. Its influence has been reduced in this case study, as the expertise of the two architects who participated were very different (1 year vs. 7 years). However, the influence of project’s size and architecture’s complexity cannot be reduced due to the inherent nature of case studies, which normally focus on one project. Also to improve the internal validity of the results, triangulation of source data has been used to increase the reliability of the results. In this regard, interviews were individually conducted with the two architects, although several questions were asked in a group setting to encourage discussion.

However, the major limitation in case study research concerns external validity, i.e. “the generality of the results with respect to a specific population” [van Heesch et al., 2012], as only one case is studied. In return, case studies allow one to evaluate a phenomenon, a model, or a process in a real setting. This is something important in software engineering in which a multitude of external factor may affect to the validation results, and that other techniques such as formal experiments, although they permit replication and generalization, do not consider as they are conducting under controlled settings.

9.3 Conclusions

The viability of the APLA supporting mechanisms is proved by a case study which has been run in an experimental laboratory called i-Smart Software Factory. It combines both academic and industrial efforts in R&D, with remarkable facilities for tracking the projects’ progress. The case study puts the proposed agile product-line architecting solution into practice within the agile development of a metering management system in the electric power networks domain. The results prove that (i) the Flexible-PLA & PLAK model provide the required expressiveness for describing and documenting the proposed PLA, (ii) the Flexible-PLA modeling primitives are effective in providing architectures with flexibility and adaptability while agile architecting, (iii) the CIA algorithm is effective in locating the impact resulting from a change, and (iv) this change-impact helps architects to take better decisions, especially when architectural knowledge may be lost or vaporized as a result of a staff turnover. These promising results did not interfere with other agile practices and did not incur a big upfront design, making the agile construction and evolution of PLAs possible. Therefore, our approach supports iterative, evolutionary software architecting by providing (i) flexibility and
adaptability at the time of defining software architectures, (ii) guidance in evolution-decisions, and (iii) assistance in preserving the integrity of the architecture. However, the use of the Flexible-PLA and PLAK modeling primitives requires to know and understand the concepts on which they are based. The learning curve of these concepts could slow down the process of putting APLA into practice.
Part VI

CONCLUSIONS & FURTHER WORK
Chapter 10

Conclusions & Further Work

This chapter presents and analyzes the main contributions of the thesis. It also presents future work that can be performed to continue this research and to extend the results that have already been obtained.

10.1 Conclusions

The turbulence of the current business climate makes change inevitable, and requires software architectures to be open to change even late in the development of software products. The trend of “being open to change” is manifested in the ASD (Agile Software Development) paradigm, but is spreading to the domain of SPLE (Software Product Line Engineering). APLE (Agile Product Line Engineering) is an emerging paradigm that aims to reduce the upfront design as currently practiced in SPLE, and to make the development of product-lines more flexible and adaptable to changes as advocated in ASD.

This thesis defined a process and supporting mechanisms for “the agile construction and evolution of product-line architectures”. We refer to this process as Agile Product-Line Architecting (APLA). The supporting mechanisms consist of a set of models for describing, documenting and tracing PLAs (Flexible-PLA and PLAK models), and a (traceability-based) algorithm to analyze change impact. These mechanisms were
evaluated through the conduction of a case study, as well as a previous pilot study that helped prepare the case study design.

The pilot study consisted of two tests: Test I and Test II. Test I consists of the analysis of the models of a home automation systems SPL, whereas Test II consists of the modeling of the PLA of a banking system SPL. The first one evaluated the understandability of Flexible-PLA models, whereas the second one evaluated the usability of the FPLA Modeling Framework.

The case study consisted of developing a metering management system in electrical power networks, which is part of a family of metering management systems (OPTI-METER SPL). In this development, we put the APLA process and their supporting mechanisms into practice. Specifically, the Flexible-PLA model provided the primitives for specifying working architectures, which were iteratively refined in each sprint of the development process. The PLAK model provided the primitives for documenting the design decisions, the dependencies and rationale resulting from architecting, as well as tracing features to the PLA. Finally, the CIA algorithm provided change-impact knowledge that guided the decision-making process of implementing the changes (planned by the product owner but also unanticipated changes) sprint after sprint of the development process.

Therefore, the contributions of this thesis were validated and put in practice in both well-known exemplars (i.e. home automation and banking systems) and in an industrial case study. The results of both studies revealed that the APLA mechanisms provided:

- Capabilities to provide agile architects with flexibility and adaptability while agile product-line architecting.
- Capabilities to guide agile architects in preserving architectural integrity while agile product-line architecting.
- Effectiveness of in locating the impacted architectural design decisions and elements resulting from a proposed change in features.
- Capabilities in guiding agile architects to reason about changes in order to make better evolution decisions based on impact and viability of the change.
- Usability of the FPLA Modeling Framework which implements the models and the algorithm to analyze change impact.
10.2 Research Contributions

The main result of this thesis is the definition of the **APLA process to assist and guide architects in the iterative and incremental construction and evolution of product-lines architectures, while complying with the “be open to change” agile principle** (see the top of the pyramid of Figure 1.3).

The mechanisms on which the APLA process is based are also results of this thesis, and they are described as follows:

**Flexible-PLA model**

The Flexible Product-Line Architecture (Flexible-PLA) model provides modeling primitives to variability by using PPCs, so that agile architects can take advantage of variability mechanisms to flexibly adapt software architectures and to incrementally develop them. This model supports the left-hand pyramid of Figure 1.3.

As a result, it has been demonstrated that the APLA is a well-supported solution for the agile construction and evolution of product-line architectures.

10.2 Research Contributions

This section presents the main results of this thesis, as well as the description of key and secondary publications that resulted from this research.

10.2.1 Results

An initial contribution of this thesis is a systematic literature review of APLE which was conducted in order to identify the current challenges of applying APLE to the software industry. This review of the state-of-the-art research in APLE was the initial point for defining the problem statement and establishing the main goal of this thesis.

The mechanisms on which the APLA process is based are also results of this thesis, and they are described as follows:

- Expressiveness of Flexible-PLA and PLAK models.

**Flexible-PLA model**

The Flexible Product-Line Architecture (Flexible-PLA) model provides modeling primitives to variability by using PPCs, so that agile architects can take advantage of variability mechanisms to flexibly adapt software architectures and to incrementally develop them. This model supports the left-hand pyramid of Figure 1.3.
10. CONCLUSIONS & FURTHER WORK

• PLAK model
The Product-Line Architectural Knowledge (PLAK) model provides modeling primitives to document variability design rationale as well as to define the basic traceability linkage between features and product-line architectures. The PLAK model offers complete support for documenting PLAs, which improves the understanding of design decisions. It also provides the basis for reasoning about future design decisions or changes, including those design decisions that affect the integrity of the architecture. This model supports the right-hand pyramid of Figure 1.3.

• Techniques for analyzing change impact
The techniques for analyzing change impact consist of a traceability-based algorithm and a rule-based inference engine. These traverse Flexible-PLA and PLAK models in order to determine the potential impact upon product-line architectures that results from adding or changing features. This change-impact knowledge that is useful for comprehending and reasoning about a proposed change in features and guiding the change decision-making process iteration after iteration during the APLE lifecycle. The CIA technique supports the pyramid core of Figure 1.3.

• FPLA modeling framework
A modeling framework prototype called FPLA\(^1\) (see the pyramid base of Figure 1.3) implements the modeling primitives for describing, documenting and tracing product-line architecture, as well as the algorithm for analyzing change impact and model-to-text transformations to generate code.

Finally, although the APLA process has been initially conceived for agile processes, any non-agile evolutionary process can take advantage of the iterative and incremental construction and evolution of product-line architectures, even single-product architectures.

10.2.2 Description of Key Publications

The results of this thesis have been published in a number of refereed research articles, published in international journals and proceedings of conferences and workshops, authored or coauthored by the author of this thesis. The publications are listed as follows:

\(^{1}\text{It is available on https://syst.eui.upm.es/FPLA/home}\)
10.2 Research Contributions


This paper presents a review of experiences and practices on APLE. The key findings uncover important challenges on how to integrate the SPLE model with an agile iterative approach to fully put APLE into practice. The most important finding is the difficulty in reducing the upfront design required by SPLE with the aim of getting closer to agile principles and values, and specifically the tension between agility and architecture, as well as the lack of mechanisms for designing PLAs which can be flexibly adapted in each agile iteration. This work has motivated the study of the iterative and incremental construction and evolution of product-line architectures that is addressed in this thesis.


This paper addresses the issue of defining variability at the architectural level. To that end, the notion of *Plastic Partial Components (PPC)* is proposed to support internal variations of architectural components. The specification of these components is performed using invasive software composition techniques and without tangling the core and product architectures of the product-line.


This paper, presented in “Second International Workshop on Variability, Adaptation and Dynamism in software systEms and seRvices (VADER 2011)”, focuses on capturing and documenting architectural knowledge of product-lines. In recent years, researches have emphasized the need to document architectural knowledge in order to maintain and evolve software, i.e. the need to document not only the design of the solution, but also the decisions driving the design and their rationale. However, few approaches document the architectural knowledge behind variability, known as *variability design rationale*. This paper presents the process for documenting variability
10. CONCLUSIONS & FURTHER WORK

design rationale of flexible and adaptive architectures, alongside their architectural description.


This paper focuses on the analysis of change impact in product-line architectures. Change impact analysis is fundamental in software evolution since it allows one to determine potential effects upon a system resulting from changing requirements. While prior studies have generically considered change impact analysis at the architecture-level, there is a distinct lack of support for the kinds of architectures used to realize product-lines, i.e. PLAs. In particular, prior approaches do not account for product-line variability. This paper presents a new technique for change impact analysis which targets PLAs. We propose to join a traceability-based algorithm and a rule-based inference engine to effectively traverse modeling artifacts that account for variability.


This paper proposes the design of working architectures in agile iterations as an attempt to solve the problem of designing software architectures in agile software development. This contribution is based on the concept of PPC. PPCs are used as highly malleable components that can be partially described, which increases the flexibility of architecture design. PPC-based architectures allow some of the agile values and principles to be reinforced.


This paper focuses on providing knowledge to assist agile architects in reasoning about changes. This paper introduces the novel solution of using change-impact knowledge as the main driver for agile architecting. The solution consists of a Change Impact Analysis technique and a set of models to assist agile architects in the change (decision-
making) process by retrieving the change-impact knowledge over the architecture of adding or changing features iteration after iteration.

10.2.3 Other related publications

The following paper presents a future research line to continue this work, and therefore, is not directly included in this thesis. This paper is described below:


This paper puts APLE into practice by tailoring the agile method Scrum by means of three concepts that we have defined: PPCs, working PLAs, and reflective reuse. Finally, the third short paper briefly highlights the benefits of APLE and the barriers of putting it into practice.


This paper focuses on understanding the interdependencies among various artifacts of the software development lifecycle, specifically on how the value chain is preserved from the problem space to the solution space. The aim is to study the feasibility of using traceability as the backbone to preserve the value chain between the prioritized product requirements and the product architecture. This paper is the starting point for an approach that traces requirements and architectures (see Chapter 6).


This paper proposes a method for prioritizing product-line features in APLE. This method is based on a formal approach used to analyze the impact of adding, modifying, or deleting features in each agile iteration, in terms of which components will be affected, which components will require refactoring, etc. The results of this evaluation may
provide useful data to facilitate the decision-making process during the prioritization of features to be implemented in each agile iteration.


Finally, the following is a list of papers addressing product line engineering in different domains:


10.3 Further Research

The APLA process provides agile architects with: (i) Flexibility & adaptability at the time of defining software architectures. (ii) Guidance in the change decision-making process, and (iii) Assistance in checking the architectural integrity at the time of constructing and evolving PLAs. But APLA can be extended by supporting: (i) Flexibility & adaptability not only at design-time but also at run-time. (ii) Guidance and assistance while evolving PLAs by automatically generating alternative solutions for implementing a change which have minimum impact over the architecture.

The necessity of “being open to change” in current business situations gives an essential role to changes in software development. Considering changes as first-class
citizens in software architectures “opens the doors” to the automation of software evolution. The Flexible-PLA model could be extended to model changes, and the PLAK model could be also extended to document the rationale of changes, in such a way that iterative and incremental construction and evolution of software architecture is completely modeled.

It is necessary to emphasize that, although most contributions of this thesis have been implemented by the FPLA Modeling Framework, this framework can be improved and extended with additional functionalities:

- The PLAK model could be extended to capture more types of knowledge, such as architectural innovation knowledge or domain experience. Agile architectures are a potential framework to guarantee speed and innovation; the challenge is to store and manage this knowledge. Hence, the agile architecting process can take advantage of this repository of knowledge, favoring the creation of new innovation knowledge and the application of innovation decisions during agile architecting.

- The CIA algorithm could be improved through different statistics, metrics, or even with different analysis perspectives, to provide richer impact-knowledge that supports agile architecting. Hence, CIA could evaluate change impact over the value of the product architecture, as well as analyze tradeoffs and conflicts, which allows architects to prioritize features iteration after iteration during an agile development.

- The rule-based inference engine could be integrated into the FPLA Modeling Framework.

Finally, we are planning a new case study in another project, running under the iSSF, for comparative studies.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


Johansen, T. (2005). Using evolutionary project management (evo) to create faster, more userfriendly and more productive software. experience report from firm as, a norwegian software company. In *Product Focused Software Process Improvement*, volume 3547 of *Lecture Notes in Computer Science* (pp. 216–223).: Springer Berlin / Heidelberg. 82, 90


BIBLIOGRAPHY


Visser, J. (2012). Change in constant. ERCIM NEWS 88. keynote. 26, 27, 30


269


Appendix A: Flexible-PLA OCL
Constraints

This appendix describes and formalizes those constraints of the Flexible-PLA Meta-
model that cannot be defined through the use of relationships and their cardinality. 
These constraints are specified by using the Object Constraint Language (OCL [Object
Management Group [2011b]] as follows:

**Constraint 1** All services which are published by an interface through the ports of a
component, must be offered by this component.

```
Context Component inv:
self.offers -> forAll(cs | cs.describedby ->
    exist ( ss1 | self.hasPorts ->
        exist(p | p.publishesInterfaces ->
            exist(i | i.publishesSignatures ->
                exist (ss2 | ss2=ss1)
            )/*end Exist
        )/*end Exist
    )/*end Exist
)/*endforall
```

**Constraint 2** An interface can be published by more than one port, but all ports must
belong to the same component.

```
Context interface inv:
Forall(i:interface) select (p | i.publishesInterfaces -> size() > 1)
    -> forAll (p1, p2 | exist (c | c.hasports = p1 and c.hasports=p2))
```

**Constraint 3** A ServiceForDerivation must belong to a PPC or to Variant (XOR).

```
Context sd: ServiceForDerivation inv:
exist ( sd1 | PlasticPartialComponent.ComposedOf -> sd1 =sd)
  XOR
exist ( sd2 | Variant.ConstitutedBy -> sd2 =sd)
```

**Constraint 4** In the relationship advice between Weaving and ServiceForDerivation,
the ServiceForDerivation must belong to a Variant.

271
Constraint 5 In the relationship pointcut between Weaving and ServiceForDerivation, the ServiceForDerivation must belong to a PPC.

Constraint 6 Every ServiceForDerivation must have at least one relationship as advice or pointcut.
Appendix B: APLA in Scrum — Model process compliant to SPEM

This appendix shows and formalizes the APLA process by using the Software Process Engineering Metamodel (SPEM [Object Management Group, 2008]) which supports all method and process modeling primitives.
Figure 10.1: APLA in Scrum: Model process compliant to SPEM
Appendix C: FPLA Experiment Questionaries

This appendix includes the questionnaires of Test I and Test II. It is necessary to emphasize that, although Test II descriptions were written in English, all questions were written in Spanish to ensure they were well-understood by students.
APPENDICES

Nombre y Apellidos: ______________________________________________________________

Experiencia laboral:

<table>
<thead>
<tr>
<th>Puesto</th>
<th>Duración</th>
<th>Tipo de Tarea Desempeñada</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Áreas de especialidad: ____________________________________________________________
_______________________________________________________________________________

Estudios: _______________________________________________________________________
_______________________________________________________________________________

Hombre/Mujer: ____
Edad : ____
Nacionalidad: ____

TEST 1
Contestar Verdadero (V) o Falso (F)
Anote la hora actual (HH:MM): ________________

1. Entendimiento del modelo de la vista Features (Características) de la herramienta FPLA:

A. La característica HomeApplianceControl es opcional ____
B. Las características SmartLighting e IntrusionDetection son obligatorias ____
C. Las características SmartLighting, IntrusionDetection, MotionSensor y Keypad son obligatorias ____
D. Las características SmartLighting, IntrusionDetection, CameraSurveillance y FingerprintScanner son obligatorias ____
E. Las características SmartLighting, IntrusionDetection, MotionSensor, Keypad, CameraSurveillance y FingerprintScanner son obligatorias ____

F. La característica IntrusionDetection tiene dos variantes: una configuración básica (basic) y otra avanzada (advanced) las cuales son mutuamente excluyentes, es decir, un producto de la familia HomeAutomation, implementa la configuración básica o la avanzada, pero nunca las dos ____

2. Entendimiento del modelo de la vista Core de Componentes y Componentes Plásticas y Parciales de la herramienta FPLA:

A. El modelo muestra siete componentes core ____

B. El modelo muestra cinco componentes plásticas y parciales (PPC) y dos componentes core ____

C. El modelo muestra cinco componentes core y dos componentes plásticas y parciales (PPC) ____

D. La PPC DoorLock es opcional ____

E. La PPC Surveillance es opcional ____

F. La componente IntrusionDetection es opcional ____

G. Se han definido tres conectores para modelar la interacción entre los componentes Home Automation y los componentes ApplianceControl, LightingControl y IntrusionDetection ____

H. Se ha definido un conector para modelar la interacción entre los componentes HomeAutomation, ApplianceControl, LightingControl y IntrusionDetection ____

I. HomeAutomation es un conector entre la componente GUI y las componentes ApplianceControl, LightingControl y IntrusionDetection ____

J. IntrusionDetection es un conector entre la componente HomeAutomation y las componentes DoorLock y Surveillance ____
3. Entendimiento del modelo de la vista de variabilidad de las Componentes Plásticas y Partiales de la herramienta FPLA:

A. El modelo muestra cuatro puntos de variabilidad (AVP – Aspect Variability Points)
B. La PPC DoorLock define dos puntos de variabilidad (AVP – Aspect Variability Points)
C. La PPC DoorLock define dos puntos de variabilidad (AVP – Aspect Variability Points) con una propiedad cada uno de ellos, MonitorSensor y Camera
D. La PPC Surveillance define un punto de variabilidad (AVP – Aspect Variability Points) llamado deviceType
E. La PPC Surveillance define un punto de variabilidad (AVP – Aspect Variability Points) con una propiedad llamada deviceType
F. El modelo muestra cuatro variantes que son cuatro aspectos
G. El modelo muestra cuatro variantes que son 2 aspectos y 2 features
H. El modelo muestra exactamente dos weavings
I. El weaving motionsensorWeaving define el proceso de “tejido” de las variantes MotionSensor y Camera
J. El weaving keypadWeaving define el proceso de “tejido” de la variante KeyPad
4. Entendimiento del modelo de la vista de interfaces & servicios de la herramienta FPLA:

A. El modelo muestra cuatro interfaces: KeyPad, FingersprintScanner, MotionSensor y Camera
B. El modelo muestra dos interfaces: IDoorLock e ISurveillance
C. La PPC DoorLock define un único servicio que es público para otras componentes a través de la interfaz IDoorLock
D. La PPC DoorLock define dos servicios que son públicos para otras componentes a través de la interfaz IDoorLock
E. La PPC DoorLock define un servicio especial (ServiceForDerivation) que no es público para otras componentes
F. La PPC Surveillance define un único servicio que es público para otras componentes a través de la interfaz ISurveillance
G. La PPC Surveillance define dos servicios que son públicos para otras componentes a través de la interfaz ISurveillance
H. La PPC Surveillance define un servicio especial (ServiceForDerivation) que no es público para otras componentes
I. Cada una de las variantes (aspectos) define un servicio especial (ServiceForDerivation)
5. Entendimiento del modelo de la vista de weavings de la herramienta FPLA:

A. El modelo muestra cuatro weavings ___
B. El modelo muestra exactamente dos pointcuts y cuatro advices ___
C. El modelo muestra exactamente cuatro pointcuts y dos advices ___
D. El weaving `keypadWeaving` define el siguiente proceso de “tejido”: el código del servicio `accessKP` se “inyecta” justo antes de la llamada al servicio `authentication` que define la PPC `DoorLock` ___
E. El weaving `fingerprintWeaving` define el siguiente proceso de “tejido”: el código del servicio `accessFP` se “inyecta” justo después de la llamada al servicio `authentication` que define la PPC `DoorLock` ___
F. El weaving `motionsensorWeaving` define el siguiente proceso de “tejido”: el código del servicio `start` se “inyecta” en lugar de de la llamada al servicio `switchon` que define la variante `MotionSensor` ___

Anote la hora de finalización (HH:MM)
TEST 2

Anote la hora actual (HH:MM)

Descripción del scenario: A SPL for banking systems

Requirements.

This section describes a specific scenario of a SPL for banking systems. Banking systems typically consist of a set of core components that offer their functionality to Automatic Teller Machines (ATM) and bank web applications (WebApp) (see Figure 1). Both ATM and WebApp are to provide a cost-effective service to bank customers that is convenient, safe, and secure 24-hour access to a common set of banking transactions and reduce the cost of providing these basic transaction. Among the most common banking transactions, this scenario focuses on the maintenance of the account balance, the withdrawal and deposit of funds, and money transfer. ATM customers can make balance enquiry and money transfers, withdraw and deposit money, whereas WebApp customers can make balance enquiry and money transfers.

In our scenario, ATMs and WebApps connect to the main banking system through their respective front-end services. The first step during the connection is the process of validation, which is different in the case of ATMs or WebApps. In the first case, valid general identification information is a debit/credit card, so that customers manually enter their card into the ATM and a unique identification information—the PIN. In the second case, valid general identification information is a code: customer enter the customer's ID, birthdate, and next customers must enter a unique identification code. Once the validation has been successful, a ticket issuer module generates a number that uniquely identifies the banking transaction. Subsequent message exchanges with the bank indicate the ticket number.
This scenario also focuses on the non-functional feature of *availability*. Banking system applications should guarantee the *availability 24 hours 7 days per week* of their core functionality to ATMs and WebApps. Several stakeholders require for their banking system applications *strict 24/7 availability*, while others permit a weaker, *non-strict availability*. The first one must provide recovery and repair in milliseconds, whereas the second one is less available and cheaper. Therefore, the strictness of availability is a variability point.

Figure 2 shows the feature model of a SPL for banking systems in which core banking transactions, ticketing and access functions offered to ATMs and WebApps are specified, as well as, the variability point that defines the variants strict and non-strict availability of these transactions.

**Architectural decisions.**

Various architectural tactics to realize availability are proposed in literature [Bass et al., 2003; Scott & Kazman, 2009]. We have selected *active redundancy* and *passive redundancy* tactics to implement strict and non-strict availability, respectively.

The tactic Active Redundancy (see Figure 3) is based on a “*configuration wherein all of the nodes (active or redundant spare) in a protection group receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s)*” [Scott & Kazman, 2009]. Therefore, from the architectural view, this tactic requires: (i) a *load balancer* in order to all nodes |active and redundant nodes| process identical inputs, and (ii) a *synchronizer* in order to active and redundant nodes maintain identical state. If there is fault, the repair occurs in time because the redundant spare has an identical state to the active node. The cost of this tactic is high due to the cost of synchronization between redundant spare and active node(s).
The tactic Passive Redundancy (see Figure 4) is based on a “configuration wherein only the active members of the protection group process input traffic, with the redundant spare(s) receiving periodic state updates” [Scott & Kazman, 2009]. Therefore, from the architectural point of view, this tactic requires: (i) a router in order to only the active node processes all the inputs, as well as to change the route to the redundant node(s) when there is a fault, and (ii) a periodic data control in order to active and redundant node(s) maintain periodic state updates. If there is a fault, the router selects a redundant spare after checking the state update. This tactic achieves a balance between the more highly available but more complex active redundancy tactic and the less available but significantly less complex spare tactic.
Se pide: modelar la PLA (product-line architecture) del scenario propuesto.

Anote la hora actual (HH:MM): __________________

Aprox. 15 minutos

a) **Modelar la vista CORE**: Model the PLA for the banking systems SPL, i.e. the components, connectors and interactions among them which make up the core configuration of the PLA.

- The components *ATM* and *WEBApp* implement the external systems to which banking system applications offer their functionality.
- The PPCs *ATMFrontend* and *WEBFrontend* abstract and simplify the communication with underlying components which provide the common banking transactions by providing a well-defined application programming interface.
- The components *Balance*, *Withdrawal*, *Deposit* and *Transfer* implement the common banking transactions such as balance, withdrawal, deposit and transfer.
- The component *TicketIssuer* manages unique identification of banking transactions.
- There is a connector *Ca*, which connects the component ATM and the PPC ATMFrontend.
- There is a connector *Cw*, which connects the component WEBApp and the PPC WEBFrontend.
- There are five connectors (*C1...C5*) that manage the connection and communication of the PPCs ATMFrontend & WEBFrontend with the components *Balance*, *Withdrawal*, *Deposit*, *Transfer* and *TicketIssuer*. They are connected as follows:
  - The PPC ATMFrontend connects to *Balance*, *Withdrawal*, *Deposit*, *Transfer* and *TicketIssuer* through *C1...C5*.
  - The PPC WEBFrontend connects to *Balance*, *Transfer* and *TicketIssuer* by reusing three of the five connectors.

Contesta a las siguientes preguntas: Sí (S), No (N), No lo encontré (x)

A. ¿Encontraste la vista para modelar los elementos core de la PLA? ____
B. ¿Identificaste el icono para modelar las componentes en la paleta de herramientas? ____
C. ¿Identificaste el icono para modelar las PPCs en la paleta de herramientas? ____
D. ¿Identificaste el icono para modelar los puertos en la paleta de herramientas? ____
E. ¿Identificaste el icono para modelar los conectores en la paleta de herramientas? ____
F. ¿Identificaste los iconos para modelar los “attachments” en la paleta de herramientas? ____
G. ¿Modelaste los componentes, PPCs y sus puertos? ____
H. ¿Modelaste los conectores y sus “attachments”? ____

Anote la hora actual (HH:MM): ________________
Anote la hora actual (HH:MM): ________________

Aprox. 15 minutos

The scenario also specified a non-functional feature: the availability 24/7 of the common banking transactions provided by a banking system. The feature availability 24/7 defines a variability point in such a way that banking systems can offer their common banking transactions upon strict availability or non-strict availability. The two front ends are the components in charge of implementing availability in its two (mutually exclusive) alternative variants, strict and non-strict. Therefore, this variability is internal to the component specification of the two front ends. This is why the two front ends are specified by using two PPCs: ATMFrontend and WEBFrontend. Both of them have a common part which is independent from the banking system application to be derived, and a variable part which is specific for the banking system application to be derived. This variable part refers to the two abovementioned variants of the feature availability.

b) Modelar la vista VARIABILITY: Model the variability points and variants.
   - The PPCs ATMFrontend and WEBFrontend define two aspect-variability points: RequestManaging and Updating. The first one defines the variability regarding with the management of requests from ATMs and WEBApps. The second one defines the variability regarding with the updating process of the nodes (active or redundant spare).
   - The aspect-variability point RequestManaging links the aspects (variants) LoadBalancing and Routing for defining the strict and non-strict availability, respectively.
   - The aspect-variability point Updating links the aspects (variants) Synchronization and DataMonitoring for defining strict and non-strict availability, respectively.

Therefore, strict availability (active redundancy) is supported by the Synchronization and LoadBalancing variants, and non-strict availability (passive redundancy) is supported by the Routing and DataMonitoring variants.

Contesta a las siguientes preguntas Sí (S), No (N), No lo encontré (x)

A. ¿Encontraste la vista para modelar la variabilidad de la PLA? ____
B. ¿Identificaste el icono para modelar los puntos de variabilidad (AVP1) en la paleta de herramientas? ____
C. ¿Identificaste el icono para modelar las variantes (aspects) en la paleta de herramientas? ____
D. ¿Identificaste el icono para modelar la relación “defines” entre las PPCs y los puntos de variabilidad (AVP)? ____
E. ¿Identificaste el icono para modelar la relación “links” entre los puntos de variabilidad (AVP) y las variantes (aspects) en la paleta de herramientas? ____
F. ¿Modelaste los puntos de variabilidad (AVP)? ____
G. ¿Modelaste las variantes (aspects)? ____
H. ¿Modelaste la relación “defines” entre PPCs y puntos de variabilidad (AVP)? ____
I. ¿Modelaste la relación “links” entre puntos de variabilidad (AVP) y variantes (aspects)? ____

Anote la hora actual (HH:MM): ________________

---

1 Aspect Variability Point
In order to pinpoint where and when extending PPCs using these variants, the variability points *RequestManaging* and *Updating* have to define weavings.

The variability point *RequestManaging* defines four weavings:
- wStrictAvailabilityRequestATM
- wNonStrictAvailabilityRequestATM
- wStrictAvailabilityRequestWEB
- wNonStrictAvailabilityRequestWEB

The variability point *Updating* defines four weavings:
- wStrictAvailabilityUpdatingATM
- wNonStrictAvailabilityUpdatingATM
- wStrictAvailabilityUpdatingWEB
- wNonStrictAvailabilityUpdatingWEB

These weavings define for each one of the two PPCs where the code of each one of the four variants is injected.

Contesta a las siguientes preguntas Sí (S), No (N), No lo encontré (x)

A. ¿Identificaste el icono para modelar los weavings en la paleta de herramientas? _____
B. ¿Identificaste el icono para modelar la relación “weaves” entre los (aspect)variability points y los weavings en la paleta de herramientas? _____
C. ¿Modelaste los weavings? _____
D. ¿Modelaste la relación “weaves” entre los (aspect)variability points y los weavings? _____

Anote la hora actual (HH:MM):______________
c) Modelar las vistas SERVICES & WEAVING

- **Service definition:** Model the services (specifically, ServicesForDerivation) for the PPC ATMFrontend and for all the variants as follows:
  - The PPC ATMFrontend implements the service *processRequest*, *processFault*, and *updateStateDaemon*.
  - The variant *LoadBalancing* implements the service *balance*.
  - The variant *Synchronization* implements the service *synchronizeState*.
  - The variant *Routing* implements the service *routeIP*.
  - The variant *DataMonitoring* implements the service *periodicUpdateState*.

Contesta a las siguientes preguntas Sí (S), No (N), No lo encontré (x)

A. ¿Encontraste la vista para modelar los servicios de la PLA? ____
B. ¿Identificaste el icono para modelar los servicios (ServiceForDerivation) en la paleta de herramientas? ____
C. ¿Modelaste los servicios (ServiceForDerivation) de la PPC ATMFrontend? ____
D. ¿Modelaste los servicios (ServiceForDerivation) de las variantes (aspects)? ____

- **Weaving definition:** Model the pointcuts and advices definition for each one of the four weavings of the PPC ATMFrontend as follows:
  - The pointcuts are represented by the ServicesForDerivation: *processRequest*, *processFault*, and *updateStateDaemon*.
  - The service *ProcessRequest* is intercepted by the advices *balance* and *routeIP* which manage the ATM and WebApp requests in order to active and redundant nodes process identical inputs or only the active node processes all the inputs, respectively.
  - The service *ProcessFault* is intercept by the advice *routeIP* which changes the route of requests to the redundant node(s) as the active node has failed.
  - Finally, the service *updateStateDaemon* is intercepted by the advices *synchronizeState* and *periodicUpdateState* which in order to active and redundant nodes maintain identical state at all time or periodic state updates, respectively.

Contesta a las siguientes preguntas Sí (S), No (N), No lo encontré (x)

A. ¿Encontraste la vista para modelar los weavings de la PLA? ____
B. ¿Identificaste el icono para modelar la relación “pointcut” en la paleta de herramientas? ____
C. ¿Identificaste el icono para modelar la relación “advice” en la paleta de herramientas? ____
D. ¿Modelaste los pointcuts para todos los weavings de la PPC ATMFrontend? ____
E. ¿Modelaste los advices para todos los weavings de la PPC ATMFrontend? ____

Anoté la hora actual (HH:MM): __________________