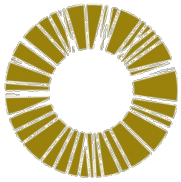

LOUVAIN
School of Management

An Agile Feature-Driven Framework for Managing Evolving Software Product Lines

The AgiFPL Method

Hassan HAIDAR

Doctoral Thesis 01 | 2020

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

LOUVAIN RESEARCH INSTITUTE IN MANAGEMENT AND ORGANIZATION

UCLouvain

Louvain School of Management

An Agile Feature-Driven Framework for Managing Evolving Software Product Lines

The AgiFPL Method

Hassan HAIDAR

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in Economics and Management Sciences of the UCLouvain

Examination Committee:

Advisor: Prof. Manuel Kolp – UCLouvain
Advisor: Prof. Yves Wautelet – KU Leuven
Examiner: Prof. Pierre Semal – UCLouvain
Examiner: Prof. Jean Vanderdonckt – UCLouvain
Examiner: Dr. Sara Shafiee – Danmarks Tekniske Universitet (DTU)

February 2020

To my mother and father,

لأمي التي أفنت روحها وجسدها من أجل تربيتي وإعلاء شأني،

لأبي الذي جاب الأرض شرقاً وغرباً كي يؤمّن لي العيش الكريم والتحصيل العلمي المميز،

أهدي هذا الجهد المتواضع أمام جهودكما الجبّارة ...

وكذلك أهدي هذا الكتاب لزوجتي وأولادي ...

Acknowledgements

This research project would not have been possible without the help and support of many persons and organizations. Therefore, I would like to thank all the people who, in several different ways, have made this research work possible and turned it into a fabulous experience.

I would like to express my deep and sincere gratitude to my advisors: ***Prof. Manuel Kolp***, Chairman of the FNRS-accredited Doctoral School of Management at UCLouvain and head of CEMIS - Center in Management Information Systems (LouRIM), and ***Prof. Yves Wautelet***, head of the Research Centre for Information Systems Engineering (LIRIS) at KU Leuven campus Brussels. Manuel and Yves were always available for advice, guidance, help, encouragement, fruitful stimulating discussions, and continuing interest in my work. This dissertation was possible because Manuel gave me the opportunity to work for the LouRIM and LSM at UCLouvain. I believe that our discussions have largely enhanced my scientific experience and allowed me to introduce the research contributions presented in this thesis. In addition, I would like to deeply thank them for their thorough and careful review of the different parts of this thesis. I would also like to thank them for supporting, and encouraging me to participate in several conferences worldwide. They gave me the opportunity to learn a lot about research, how to tackle new problems, and how to develop technique to solve them. I thank them for the friends that they have become and will continue to be. Thanks Manuel, thanks Yves.

I would like to express my thanks to the members of my PhD committee: Prof. Jean Vanderdonckt (UCLouvain), Prof. Pierre Semal (UCLouvain), and Dr. Sara Shafiee (Technical University of Denmark – DTU) for accepting to participate in the jury, for carefully reviewing my thesis and for their many useful suggestions and stimulating questions and feedback. Their comments have improved the text and are greatly acknowledged.

In addition, I would like to thank all my friends and colleagues of Louvain School of Management (LSM) and Louvain Research Institute in Management and Organization (LouRIM) for their friendship, help, support and the nice work environment. Many sincere thanks go to Prof. Marco Saerens, Kamila Moulaï, Sylvain Courtain, Iyad Khadam, Ghazaleh Aghakhani, Pierre Leleux, Soreangsey Kiv, Mehdi Ousmer, Nicolas Burny, Vu Nguyen Huynh Anh, Paul Chatelain, Samedi Heng, Nesrine Mezhoudi, and Manuel Herrera Rodriguez. Moreover, warm regards go to Sylvie Baudine, Sandrine Delhaye, Heike Rämer, Prof. Valérie Swaen (President of LouRIM) for their help and encouragement.

I am very thankful to my friends in Belgium, Lebanon, and around the world who have encouraged me and have made my life joyful during all these years. Warm thanks go to Prof. Anne-Marie Polomé, Dr. Saïf-Eddine El Bouhali, and Loay Moghnieh.

My deep gratitude and sincere thanks go to my mother Nassima and my father Mohamad for their motivation and support throughout my studying years. Without their efforts, fine education, support, and guidance, I would probably not have taken up the eagerness or motivation to do this PhD. I would like to sincerely thank them for their encouragement and love, which gave me so much joy. Thank you so much for the education you gave me. Dear and Mom Dad, this thesis is dedicated to you in particular. I would also like to thank my sisters and my brother for their support and encouragement.

Finally yet importantly, I would like to sincerely thank my wife Sahar for her love, support, encouragement, and patience during the PhD period. I thank her for being there for me in times when I needed her most which gave me the strength to perform this research project. I would also like to thank my lovely and adorable kids: Jude, Zahraa, and Mohamad. Their warm and lovely smiles were the source of my power.

It was not possible to mention all the people who contributed in one way or another to this PhD. My apologies to the people I forgot and my warmest thanks for everything.

Hassan Haidar

Abstract

Agile Product Line Engineering is a paradigm that has emerged as a solution for responding to the need for managing changes in requirements, reducing time-to-market, promoting product quality, and decreasing development costs in software organizations. Agile Product Line (APL) approaches are the results of combining agile methods with Software Product Lines (SPL). The main goal of this thesis is to propose an efficient APL method. In general, software development methodologies consist of two integral parts. The first one is dedicated for requirements engineering and the second one is dedicated for the development process. In this thesis, we have proposed a Feature-Oriented Agile Product Line method called "Agile Framework for managing evolving Software Product Lines – AgiFPL method". AgiFPL has been defined, designed and implemented after studying and analyzing the existent APL methodologies with the aim to take advantage of the strengths of the studied methodologies and to overcome their weaknesses. AgiFPL was proposed to address the development process part. For the requirement engineering part, we have proposed an Integrated Requirements Engineering framework for Agile Software Product Lines, which is the part that provides the syntax and semantics used for expressing the products of an APL method. The aim is to allow analysts and developers to specify requirements that precisely capture the stakeholder's needs and intentions as well as to manage product line variabilities. Finally, this thesis proposes an assessment model called AgiPL-AM (Assessment Model for Agile Product Lines) for the assessment of the situation of agile adoption within agile product line approaches. In fact, assessing the current situation, regarding the combination of agile practices and activities with Software Product Lines, is an essential step towards a successful integration of agile methods into Software Product Lines.

Table of contents

List of figures

List of tables

List of abbreviations

2GPLE	Second Generation Product Line Engineering
AD	Application Design
AE	Application Engineering
AgiFPL	Agile Framework for managing evolving Software Product Lines
AgiPL-AM	Assessment Model for Agile Product Lines
APL	Agile Product Line
APLE	Agile Product Line Engineering
AR	Application Realization
ARE	Application Requirements Engineering
ASD	Agile Software Development
AT	Application Testing
BAPO	Business-Architecture-Process Organization
BUD	Big Upfront Design
CBE	Criteria-Based Evaluation
CDD	Component-Driven Development
COPA	Component-Oriented Platform Architecting Method
CVL	Common Variability Language
D	Descriptive Form
DD	Domain Design
DE	Domain Engineering
DOD	Definition of Done
DR	Domain Realization
DRE	Domain Requirements Engineering
DSL	Domain Specific Language
DT	Domain Testing
FAST	Family-Oriented Abstraction, Specification, and Translation process
FODA	Feature-Oriented Domain Analysis
FORM	Feature-Oriented Reuse Method
FOSD	Feature-Oriented Software Development
KobrA	Component-Based Application Development Komponentenbasierte Anwendungsentwicklung
MDA	Model Driven Architecture
ML	Modeling Language
N/A	Not relevant to the context or properties of the methodology
N/D	Not defined in the methodology
PL	Product Line
PLA	Product Line Architecture
PLE	Product Line Engineering
PLUS	Product Line UML-Based Software Engineering
ProM	Product Management

QADA	Quality-driven Architecture Design and quality Analysis
RE	Requirements Engineering
RQ	Research Question
SC	Scale Form
SDLC	Software Development Life-Cycle
SE	Software Engineering
SIP	Stories in Progress
SLR	Systematic Literature Review
SM	Simple Form
SPL	Software Product Line
SPLE	Software Product Line Engineering
WIP	Work In Progress

Part I – Introduction

1 Introduction

Software Product line engineering (SPLE or PLE) is a well-established software engineering discipline that provides an efficient way to build and maintain *"portfolios of systems"*, which share common *"features"* and *"capabilities"* [Pohl et al., 2005] [van der Linden et al., 2007]. Systems built with SPLE have, for decades now, demonstrated improvements in *"development time"*, *"cost"*, *"quality"*, and *"engineering productivity"*. Significant advantages in family-based software development have been proven by SPLE, such as *"faster time-to-market*, *better quality*, and *lower costs by means of systematic reuse and mass-customization"* [Clements and Northrop, 2001]. Fundamentally, implementing a SPLE method requires considerable upfront planning and design (i.e. Big Upfront Design – BUD) with a heavyweight software process to achieve organization business goals. When the anticipated changes in *"core-assets"* have been predicted with certain accuracy, SPLE has provided significant improvements. However, when large or complex software product-line projects have to deal with *"changing market conditions"*, alternatives to supplement SPLE are required [Díaz et al., 2011].

The difficulty of predicting the level of *"demand uncertainties"* of software products and the *"rapid changes"* in the current software business environment have made alteration to software products development inevitable. The change is required in order to stay competitive in the market [Beck and Andres, 2004]. Agile Software Development (ASD) methods are considered able to *"both create and respond to change in order to profit in a turbulent business environment"* [Highsmith, 2002]. Basically, agile methods ensure *"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"* [Highsmith, 2009] [Cockburn, 2007]. Unlike SPLE approaches, Agile Software Development (ASD) approaches achieve the organization strategies (i.e. business goals) through the *practices*, *principles*, and *values* focused on *"people and interactions"*, *"working software"*, *"customer collaboration"*, *"responding to change"*, and *"continuous improvement"* [Manifesto Agile, 2001]. Contrary to SPLE, ASD targets a *"lightweight process and low upfront planning and design"* [Larman and Vodde, 2008].

Software Product Line Engineering (SPLE) processes and Agile Software Development (ASD) processes advocate identical *"business goals"*, such as *"reducing time-to-market"*, *"increasing productivity"*, and *"gaining cost effectiveness and efficiency of software development efforts"* [Tian and Cooper, 2006]. Since both processes share common goals, there has been growing interest in whether the integration of *"Agile"* and *"Software Product Lines"* could provide further benefits and solve many of the remaining issues surrounding software development [de Souza and Vilain, 2013].

1.1 Research Motivation and Context

The idea of combining *"Agile"* and *"Software Product Lines"* has become a trending research field, shifting focus and taking the attention of many researchers and practitioners within the software industry. The main interest lies in discovering how to combine reusability and customization, as practiced in Software Product Lines, with concepts such as iterative development and embracement to change as encouraged in agile methods [Ghanam et al., 2011]. To deal with the growing complexity of information systems and to handle the competitive and changing needs of the IT production industry, practitioners and researchers have proposed several approaches with the intention of combining agile and product lines techniques [da Silva et al., 2011]. The goal was to make Software Product-Line methodologies evolving from predictive to iterative and incremental, and to agile approaches.

Agile Product Line Engineering (APLE) is a new paradigm that promotes the integration of *"agile"* principles and *"Software Product Lines"* with the target of *reducing the big upfront design (BUFD)* associated with the *product-line platform*, while making the development within the Software Product Lines more *flexible* and *adaptable* to change [Cooper and Franch, 2006] [Díaz, 2012]. APLE is presented as *"promising"*, however, there are several challenges when putting APLE into practice. In [Hohl et al., 2017a] and [Hohl et al., 2017b], Hohl et al. have identified some challenges that could relate to activities, practices, or even the organization itself. The main challenges listed by Hohl et al. are the following:

- *Organizational challenge*: The exact coordination is still a challenge for introducing agile elements to the existing SPL. When introducing agile practices into the existing SPL processes, it is recognized that the *"coordination of the software development"* impedes a faster flow of activities [Hohl et al., 2017a];
- *Challenge related to worldwide-distributed development team*: a major challenge is the collaboration with suppliers and a worldwide distribution of the development team. Different cultures and different mindset are likely to impede an agile development [Hohl et al., 2017a];
- *Challenges related to management*: some cases exist where the management does not want to give up any responsibility. With less responsibility on the managerial level, scheduling of the development and reporting will be challenging, it is unclear for the managers how the agile software product line could be planned and features are scoped [Hohl et al., 2017b];
- *Challenges related to dependencies and synchronization*: a dynamic coordination is deemed to a necessity to introduce agile development practices into SPL development process. The development process across several domains must therefore be synchronized [Hohl et al., 2017a];
- *Challenges related to validation and release*: with an APLE approach, it is a challenge to scale the test framework to test all variants within the SPL. It is unclear how far the automation of tests could help in the process. Testing strategies must be context specific and scalable [Hohl et al., 2017a];

- *Challenges related to Software Development*: One of the major challenges that face an APLE approach is the software development itself. The identified challenges in the software development are clustered as technical challenges, challenges related to costs, requirements management challenges, challenges related to software architecture, challenges related to software quality, and challenges related to the use of SPL and variants [Hohl et al., 2017a].

In addition to these challenges, it was recognized that several foundations of SPLE and ASD are highly contrasting, or sometimes even opposite [Hanssen, 2011]. It is therefore necessary to deal with these challenges and differences when integrating agile practices with SPLs, or even when adopting an existent APL method. In other words, APLE's actual context implies that organizations have to face several barriers to achieve its adoption effectively. The significant advantage in combining the *"agile" attribute* and SPLE approaches is the *"synergy"*. This synergy means that each approach has the capacity to address the weaknesses of the other. As mentioned, although there are many advantages in combining the two approaches, some difficulties do still exist. These difficulties are mainly due to the *"inherent differences"* of the approaches [Farahani and Ramsin, 2014]. The core differences all lie in the following [Hanssen and Fægri, 2008]:

- The strategies for handling changing requirements;
- The degree of focus on documentation;
- The level of user involvement required;
- The development roles involved.

Despite the emphasized difficulties, large companies make great efforts to achieve a successful combination of both methods. Agile Software Product Line Engineering is driven by the assumed improvements for the customers and the software developers (companies). In fact, companies assume that the development could benefit from both a working *"reuse strategy"* and an *"increased flexibility"* with an agile Software Product Line. This *"flexibility"* is very important in order to react appropriately to *"customer needs"* and *"changing requirements"* during the development process [Hohl et al., 2016]. Accordingly, Hohl et al. [Hohl et al., 2017] have argued that combining SPLs and agile development methods is not *trivial*. To face the new market trends of the software industry, companies are willing to take the risk and spend all the needed efforts in order to overcome the obstacles of combination. Their aim is to adopt an *Agile Product Line approach* that ensures the achievement of *"shorter time-to-market"*, *"shorter release cycles"*, as well as the introduction of development practices such as *"continuous integration"* and *"continuous delivery"*.

Several methods have been proposed to provide a practical process for applying APLE in organizations, but none of the identified methods covers all the required APLE features. The survey of Farahani and Ramsin [Farahani and Ramsin, 2014] has shown that any new APL method has to cover the following features:

- Full coverage of the generic Software Development Life-Cycle (SDLC);
- Comprehensive and precise definition of the methodology;
- Sufficient attention to the non-SDLC activities;

- Prescription of a specific modeling framework of requirements (Requirements Engineering approach);
- Provision of model examples;
- Attention to active user involvement;
- Management of expected and unexpected changes.

This thesis aims to propose a new Agile Product Line method. The intention behind the definition of a new APL method is to address the deficiencies identified in the current methods, while making use of their advantages. An important and first step when combining *"Agile"* and *"Software Product Lines"* is to identify the basis of the process of the designed *"APL method"*; whether it is SPLE or ASD. In the literature, two main scenario of combining of agile methods with Software Product Lines (SPLs) were presented. In the first scenario, researchers and practitioners propose APL approaches by starting with a SPL process as the main process, and then adding the agile practices. In the second scenario, researchers have proposed APL approaches, in which the agile method is considered as the main development process, where the integration starts from the agile method, and the SPL practices are then added.

Defining a new APL method is actually defining a new software development methodology. Chen and Babar [Chen and Babar, 2011] argued that a software development methodology mainly consists of two integral parts:

1. *Modeling Language*, which provides the syntax and semantics used to express the products. This part, in fact, represents some practices and activities of the requirements engineering;
2. *Process*, which prescribes the flow of required activities and explains how the products should be produced, enhanced and exchanged along this flow. Thus, this part essentially represents the software development process.

To reach the target, a development process should be designed and aptly provide modeling modules and rules.

1.2 Problem Statement

After an in-depth review of the available literature on Agile Product Line Engineering and existing APL methods, two main findings could be deduced, namely:

1. *Practitioners could conclude that there are sufficient reasons to move towards a combination of Agile Software Development with Software Product Line Engineering* [Díaz et al., 2014].
2. *Researchers could conclude that there are still some important challenges in the area of APLE, and therefore, more research work is needed to completely put Agile Product Line Engineering into practice* [da Silva et al., 2015].

Regarding the first finding, it was acknowledged that APL methods would be applicable to business cases where a clear convenience of going towards a Software Product Line existed. However, it should be considered that the current market situation is not stable enough for different reasons; including but not limited to technological factors. It does, however, remain advantageous to deploy APLE into practice in many situations:

a. When developers do not have sufficient knowledge to completely perform the Domain Engineering (DE), ASD may facilitate the elicitation of further requirements, specifications, and knowledge [Tian & Cooper, 2006];

b. Trade-offs between ASD and SPL provide the opportunity to apply the APL method to a wider variety of projects than those served by only applying a single of these two methods [Tian & Cooper, 2006];

c. When anticipated changes cannot be predicted and the product lifecycle is unknown, it would be advantageous to use an incremental method, such as an APL method [Carbon et al., 2006];

d. Agile processes may facilitate fast feedback cycles between requirements engineering (RE), software development, and field trial in innovative business [Kircher et al., 2006].

Regarding the second finding, the review of the literature on APL methods has reported the following main ascertainments:

i. The integration of agile methods to Domain Engineering (DE) requires more effort than its integration to the Application Engineering (AE). In fact, it is difficult to reduce the upfront design with the aim of getting closer to agile principles and values, while achieving the typical goals of DE, such as reuse [Ghanam et al., 2009b]. However, the integration of ASD to AE seems feasible. This feasibility was confirmed in several works, such as [O'Leary et al., 2009b] [Ghanam et al., 2010] [da Silva et al., 2015] [Farahani and Ramsin, 2017] [Hohl et al., 2017b];

ii. Synchronization between platform and product teams 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 [O'Leary et al., 2008]. According to [Ghanam et al., 2009b] the synchronization between DE and AE teams still remains a challenge for APLE practitioners;

iii. Business goals have to be considered in order to identify the extent of flexibility required. Which should prove to be useful in determining the combination of SPLE and ASD, i.e. either SPLE and ASD at the strategic level, or SPLE at the strategic level and ASD at the tactical level [Tian & Cooper, 2006] [Hanssen & Fægri, 2008] [Mohan et al., 2010] [Hohl et al., 2017a] [Farahani and Ramsin, 2017].

Consequently, the main problematic issue that has been tackled by this thesis is to propose a new APL method that:

- *Combines agile practices with Software Product Lines effectively ;*
- *Limits at the maximum "Big Upfront Design" ;*

- *Ensures 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 ;*
- *Presents practical ways to integrate agile practices to Domain Engineering lifecycle, while preserving the core architectures and main principles of Software Product Lines;*
- *Ensures practical and effective synchronization between DE and AE teams is highly ensured.*

1.3 Research Design

The research activities in Software Engineering (SE) lead to the introduction of new *models*, *methodologies* and *tools* that intend to aid software engineers and developers to *understand the project contexts*, *complex problems*, and *to improve effectiveness and efficiency* [Shaw, 2002]. Like any other research discipline, research in software engineering requires guidance on the research process in order to ensure good research. This thesis relies on different research methods, such as *"Design Science"* principles [Hevner et al., 2004], and specific research techniques, such as *"Systematic Literature Review (SLR)"*.

This thesis is built up from three main parts; each part has been based on a different research method and approach. Figure 1.1 shows the chapters that address the different research questions defined in this dissertation. The research process designed to reach the target of this thesis could be presented as the following:

I. *Part 2* involves a literature study of the relevant approaches and techniques;

II. *Part 3* focuses first on defining the key elements and the software architecture required for building the required methodology. Then, it focuses on building the methodology up upon the defined concepts, artifacts and architecture for the full lifecycle coverage;

III. *Part 4* focuses on the evaluation and the validation of the designed methodology.

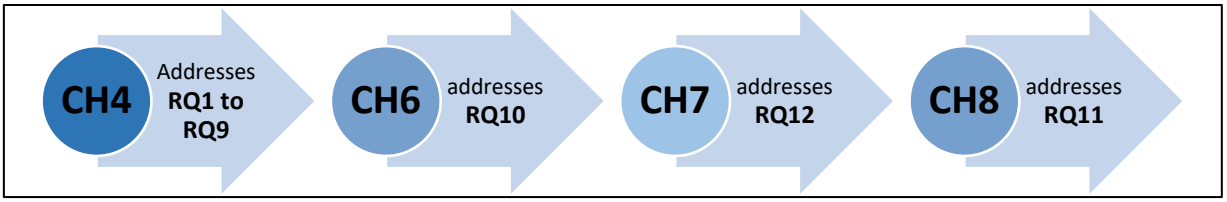

Fig. 1.1 Chapters that address the research questions of the dissertation.

1.3.1 Literature Review

To successfully achieve this part of the thesis, a review the scientific sources related to the main topics of interest was conducted. The literature review part focused on research works related to *Software Product Line (SPLE) methodologies*, *Agile (ASD) methodologies*, *Agile Product*

Line (APL) methodologies, as well as some related *Requirements Engineering (RE) frameworks*.

Part 2 of this thesis was built in three steps. During the first step, the main SPLE and ASD methodologies related to this research project were reviewed. By this step, the *state of the art* related to SPLE and ASD methodologies were explored. The intention is to cover the vertical and the horizontal dimensions of the well-known methodologies in the SPLE and ASD fields. In the second step, the relevant APL methodologies, or more precisely their *phases*, *workflows* and *the global processes*, were presented. After presenting the selected APL methods, a *"Systematic Literature Review (SLR)"* was conducted, in order to respond to the following *Research Questions (RQs)*:

- RQ1: What are the purposes of the combination of SPLE and ASD? What are the expected benefits of the combination of ASD and SPLE?
- RQ2: How the Agile principles are related to the SPL principles?
- RQ3: How the combination of SPLE and ASD respects the business strategic goals?
- RQ4: Which current APL approaches satisfy the Application Engineering (AE) activities?
- RQ5: What are the challenges and gaps in current APL approaches that relate to AE activities?
- RQ6: Which current APL approaches satisfy the Domain Engineering (DE) activities?
- RQ7: What are the challenges and gaps in current APL approaches that relate to DE activities?
- RQ8: Which current APL approaches satisfy both DE and AE activities through agile principles?
- RQ9: Do successful experiences putting an APL approach into practice exist?

The conducted systematic literature review (SLR) has given a *macro-view* of the different approaches that combine ASD with SPLE, and thus, a macro-view about the current APL approaches. This SLR has presented a sufficient macro-view about the *activities*, *practices*, and *experiences* with APLE. However, to achieve the ultimate objective of this thesis, which is to propose a strong APL method, a more detailed *micro-view* about the reviewed studies on APLE was needed. Accordingly, a *"Criteria-Based Evaluation (CBE)"* was performed to evaluate a set of relevant APL method. By means of the CBE, it was intended to evaluate:

- The global processes of the methodologies;
- The characteristics related to the agile principles and values;
- The characteristics related to the SPLE principles;
- How these characteristics fit the common goals of agile and SPLE;
- The characteristics related to the combination of the two approaches.

At the end of the *Part 2*, an attempt was made to successfully identify how Software Product Lines deal with the Requirements Engineering (RE) issues, as well as how agile methods do. In addition, it was intended to identify the main RE practices and techniques that could help to propose a RE framework for Agile Software Product Lines.

1.3.2 Building-up the targeted APL method

Part 3 addresses the core objective of this thesis directly, which is *"Proposing an Agile Product Line method"*. The main research questions that have been tackled are the following:

- RQ10: How to combine agile practices with Software Product Lines and therefore build an APL method that takes the strengths of the previous APL methods and overcomes their weaknesses?
- RQ11: What are the Requirements Engineering activities and practices that fit the Agile Software Product Lines principles? How could they be integrated to the proposed APL method?
- RQ12: How to assess the agility attribute of an agile product line method?

Defining a new APL method (i.e. responding to RQ10) is essentially defining a new software development methodology that has two main integral parts: *Modeling Framework* and *Development Process*.

To design the development process of the targeted APL method, *first* the results of the *Part 2* are taken as a basis to build-up the targeted APL method. *Second*, the design of the targeted process is initiated by selecting the *SPL approach* that would be the basis of the targeted APL, and then the agile method(s) that will be integrated to the chosen SPL is selected. After that, the APL process was designed. *Third*, the designed process has been reviewed in order to optimize it. To do this, two academics and one software engineering expert were chosen. Each person has reviewed the designed process separately.

To answer research question (RQ11), a research procedure (proposed by Engels and Sauer [Engels and Sauer, 2010]), that has the following steps, was used:

- Define domain and disciplines;
- Produce domain model of software engineering concepts;
- Select notations;
- Define artifacts types;
- Define the software engineering process models;
- Select tools, techniques and utilities.

To respond to research question RQ12 an assessment model to assess the agility attribute of an agile product line method was proposed. In order to reach the target here, a

research procedure of three phases was followed. *The first phase* starts by a review of literature on maturity models that concern Software Product Line Engineering, Agile Product Lines, and Agile Software Development. *The second phase* involves the construction of the proposed "agile assessment model". After defining the main objectives of the required assessment model, the assessment model have been designed and developed in an iterative way. *In the third phase*, the model was applied and evaluated. At this stage, the model has been reviewed and refined in order to optimize and finalize the proposed assessment model.

1.3.3 Assessment and Validation

In this part, a research procedure of two phases was followed. First, the APL method is validated utilizing case study techniques. Second, the proposed method is evaluated through a specific evaluation strategy.

The *case study methodology* has been used in this thesis, as it is a commonly used research strategy in Software Engineering. It is a demonstration case that exhibits the implementation of some software technology or programming concept. In fact, it involves an interpretive, naturalistic approach to the world, investigating things in their natural settings (i.e. real industrial case), attempting to make sense of, or interpret, phenomena in terms of the meaning people bring to them. To demonstrate the real world feasibility and benefits of our proposed APL method, a case study has been performed by focusing on a specific project called *TransLogisTIC* [Wautelet, 2011].

The evaluation of the proposed APL method and the validation of its applicability in practice as well as the identification of its strengths and weaknesses were ensured in two steps. First, we evaluate it by performing a *Criteria-Based Evaluation (CBE)*. Second, we assess its level of agility by using the *Assessment Model* developed in Part 2.

1.4 Overview of the proposed solution

This thesis proposes three complementary solutions. *First*, it proposes an Agile Product Line method called AgiFPL (Agile Framework for managing evolving Product Lines). *Second*, it proposes an Integrated Requirements Engineering framework for Agile Software Product Lines, which provides the syntax and semantics used for expressing the products of an APL method. *Third*, it proposes an assessment model called AgiPL-AM (Assessment Model for Agile Product Lines) for the assessment of the situation of agile adoption within agile product line approaches.

AgiFPL has been defined, designed and implemented after studying and analyzing the existent APL methodologies with the aim of taking advantage of the strengths of the studied methodologies and to overcome their weaknesses. In this thesis, AgiFPL was proposed to address the development process. In fact, the proposed process consist of a set of *units of work (activities, tasks, steps)*, *guidelines*, *roles*, and *work-products* that allow the people (Experts,

Owners, Developers, stakeholders, etc.) involved with the Software Product Line (SPL) to perform the development processes of Domain Engineering (DE) and Application Engineering (AE) aligned with *Scrumban* and *Scrum* practices. AgiFPL has indeed presented a Scrumban-inspired process for the DE tier and a Scrum-inspired process for the AE tier. Scrum was adopted, as it is considered the mainstream method within development teams and its practices fit the AE process by handling *feedback*, *iterativeness*, *incrementally*, and *adaptability* in a systematic way. Scrumban was adopted, as it ascribes more importance to the requirements engineering part than the other agile methodologies; it establishes a structure of limited number of *"Production Lines"* within its development process. It is furthermore commonly used for *maintenance* projects, where it helps to deal with the *traceability* issue in the DE tier. Moreover, a 3-phases approach to implement AgiFPL in practice was proposed.

The "Integrated Requirements Engineering framework for Agile Software Product Lines" represents some practices and activities of the requirements engineering. In fact, Requirements engineering (RE) techniques play a determinant role within Agile Product Lines development methods; as they notably establish the relevance fueling the decision to adopt the product line approach for software-intensive systems production. This framework proposes an integrated goal and feature-based metamodel for agile software product lines development. The aim is to allow analysts and developers to specify requirements that precisely capture the stakeholder's needs and intentions, as well as to manage product line variabilities. Adopting practices from requirements engineering, especially goal and feature models, helps designing the domain and application engineering tiers of an agile product line. Such an approach allows for a holistic perspective integrating human, organizational and agile aspects to understand product-lines dynamic business environments. It aids to bridge the gap between product-lines structures and requirements models, and proposes an integrated framework to all actors involved in the product-line architecture.

The AgiPL-AM model is an assessment model to evaluate the situation of agile adoption within agile product line approaches. In fact, assessing the current situation, regarding the combination of agile practices and activities with Software Product Lines, is an essential step towards a successful integration of agile methods into Software Product Lines. Following a specific research approach AgiPL-AM was built, which allows for self-evaluations within the team in order to determine the current state of agile software development in combination with Software Product Lines. AgiPL-AM is comprised of six categories (five are related to agile principles and one to product-line architecture) and five levels of maturity. Furthermore, AgiPL-AM has the ability to reveal and pinpoint agile product-line approach *strengths* and *weaknesses*.

1.5 Threats to validity and limitations

15.1 Threats to validity related to the literature review

The outcome of the presented literature study is biased by different factors. In fact, there are several threats to the validity of the conducted studies. This section presents the threats to

validity according to the different steps of the literature review. These threats are categorized according to Wohlin et al. [Wohlin et al., 2003] as internal, external, and construct validity.

- The ***"Construct Validity"*** threats concern mainly the relationships between theory and observation. These threats are essentially due to the method used to assess the outcomes of the conducted literature review. This kind of threats were mitigated through the multiple sources of evidence that allow establishing chains of evidence (e.g. field observations, analysis of documents, etc.).The main construct validity threats are the following:

  - *Concerning the defined Research Questions (RQs)*: the defined research questions might not provide complete coverage of the Agile Software Product Line (APL) area. This issue was considered as a feasible threat. Thus, several discussion meetings with the research team were held in order to calibrate the adopted research questions;

  - *Concerning the selection of databases*: several databases publish studies and papers related to the area of Software Engineering (SE). It is high probably that some publications are listed in more than database while others are not. The research team has mitigated this threat of missing studies by using seven databases that commonly publish related work.

- The ***"Internal Validity"*** threats are concerned with factors that might affect the dependent variables without the researcher's knowledge. The internal validity was mitigated mainly through the defined steps to analyze the collected data. The main internal validity threats are the following:

  - *A subjectivity in the study selection (Publication bias)*: it not possible to guarantee that all relevant primary studies were selected. In fact, it is possible that some relevant papers were not chosen. In order to mitigate this threat, an automatic search was performed, and it was complemented by performing manual search to try to collect all primary studies in this field. More precisely, the references in the primary studies were followed;

  - *A subjectivity in the data extraction*: during the data extraction process, the primary studies were classified based on the judgement of one researcher (i.e. the author of this thesis) of the research team, which may increases the amount of research bias. In order to reduce biases due to subjective decisions, specific criteria for inclusion and exclusion of a paper were formulated. In fact, the inclusion and exclusion criteria can retain a certain objectivity of the results. In case of doubts, all the concerned researchers (i.e. 3 researchers) have decided about the inclusion or exclusion of the study. A second researcher reviewed the whole process. Despite the double-checking, it is possible that some studies have been classified incorrectly.

- The ***"External Validity"*** is related to the ability to generalize the results of the conducted study. The main external validity threats are the following:

  - *Concerning the results*: because of the construction of the study, the research team cannot ensure that they have found all the publications that are relevant for the topic of Agile Software Product Line approaches.

  - *Concerning the repeatability of the systematic process*: to deal with the risk that involves the ability to replicate or extend the conducted studies, a detailed description of the systematic processes used in this dissertation were described (e.g. SLR protocol, etc.).

1.5.2 Threats to validity related to the proposed solutions

The APL method as presented in this dissertation and pointed out as its principal contribution is tailored to feature-driven Software Product Lines (SPLs). As such, its use with different SPLs may require recalibrating the proposed processes, which leads to the need for several test cases. In addition, the proposed ALP method has presented a way to combine agile techniques with SPLs, not to integrate to an agile development process a set of SPL practices.

Another threat to validity of the research is that the generalizability of the findings of this research project is limited only to the contexts of the cases used within this thesis. However, the analysis and integration of other similar cases could extend the obtained results of this thesis, with context similar to any of the case companies may find the findings of this thesis applicable.

Finally, a few examples and one case study were presented in this dissertation. Thus, there are threats to validity that limit the generalization of the case study results. To ensure the credibility of the obtained results, several future research projects, which adopt a large enough set of case studies developed in a real software industrial environment and that involve more professionals in the research team will be launched.

1.5.3 Limitations

The proposed solutions as presented in this thesis bear potential for improving the Agile Product Line Engineering area, however, still carries several limitations. This section discusses the boundaries and limitations of the work presented in this thesis. The most important ones are the following:

- The absence of a complete and implemented tool that support the different presented approaches may influence the quality and the usability of the presented solutions. Thus, there still a small limitation to the approach's real world applicability.

– A more comprehensive empirical validation of the proposed approach and solutions would be required. Some parts of the validation presented in "Part IV" are still rather preliminary and some others are limited to some extent. Therefore, a well-established tool support and further case studies could provide a significantly strong empirical validation in the future.

1.6 Research Contributions

This thesis was partially shared with the scientific community as it is shown in Figure 1.2. The main results and key publications that result from this dissertation are based on the following contributions:

- *(C1)*: Haidar, H., Kolp, M. and Wautelet, Y. (2017). Agile Product Line Engineering: The AgiFPL Method. In *Proceedings of the 12th International Conference on Software Technologies - Volume 1: ICSOFT*, Madrid, Spain, pp. 275-285. DOI: 10.5220/0006423902750285

- *(C2)*: Haidar, H., Kolp, M. and Wautelet, Y. (2018). Formalizing Agile Software Product Lines with a RE Metamodel. In *Proceedings of the 13th International Conference on Software Technologies - Volume 1: ICSOFT*, Porto, Portugal, pp. 90-101. DOI: 10.5220/0006849001240135

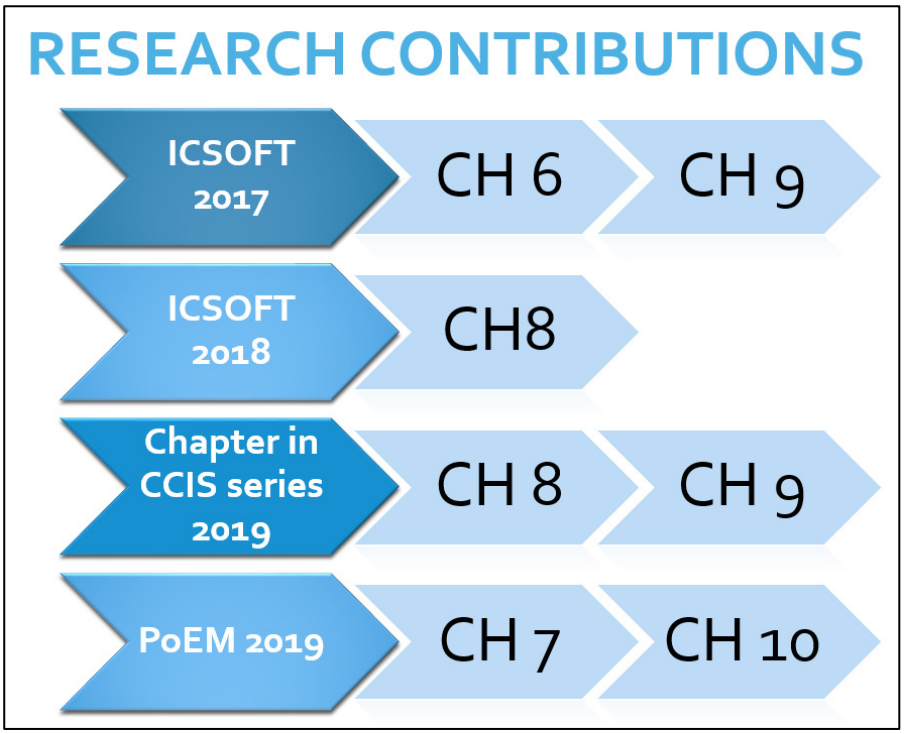

Fig. 1.2 Matching between published papers and chapters of this dissertation.

- *(C3)*: Haidar, H., Kolp, M., and Wautelet, Y. (2019). An Integrated Requirements Engineering Framework for Agile Software Product Lines. In *van Sinderen M., Maciaszek L. (eds) Software Technologies. Communications in Computer and Information Science, vol 1077*. Springer, Cham, pp. 124-149. DOI: https://doi.org/10.1007/978-3-030-29157-0_6

- *(C4)*: Haidar, H., Kolp, M., and Wautelet, Y. (2019). Assessing the adoption level of agile development within Software Product Lines: the AgiPL-AM model. In *Guédria W., Gordijn J., and Proper H. A. (eds) The Practice of Enterprise Modeling. PoEM 2019*. Lecture Notes in Business Information Processing, vol 369. Springer International Publishing, pp. 134-148. DOI: 10.1007/978-3-030-35151-9

1.7 Reading Map

This dissertation consists of four parts and eleven chapters. Figure 1.3 provides the structure and the roadmap of the thesis. The remainder of this thesis is structured as outlined below:

- *Chapter 2* and *Chapter 3* give an overview of Software Product Line Engineering (SPLE) and Agile Software Development (ASD) states of the art. Chapter 2 introduces Software Product Line methodologies. It comprehensively presents the key concepts and the motivation behind product-line development, the promises of product line engineering, and how the product lines are characterized in the context of handcrafting and mass production. In addition, it presents some frameworks and development processes of software product lines and their specific characteristics. Chapter 3 considers *"Agility"* from a management perspective by introducing a considerable overview of agile software development and by focusing on implementation, organization and people of two main methods, namely, *Scrum* and *Scrumban*.

- *Chapter 4* reviews the context related to APLE. This chapter appraises significant APL approaches, presents a systematic literature review, and uses a criteria-based evaluation to compare relevant APL approaches in order to increase the understanding of agile SPL.

- *Chapter 5* gives an overview of the different aspects of the Requirements Engineering (RE) discipline. In addition, it reviews how Software Product Lines (SPL) deal with the RE issues, as well as how agile methods do. Further, it presents the main RE practices and techniques that have been used to propose the RE framework for Agile Product Lines presented in Chapter 8.

- *Chapter 6* presents the detailed description of the processes of the proposed APL method (i.e AgiFPL). This description introduces the roles, work products, activities, tasks, steps, and workflow of the processes of Domain Engineering (DE) and Application Engineering (AE), according to AgiFPL. In addition, this chapter has introduced an approach to implement AgiFPL in practice, which is a 3-phases approach.

- *Chapter 7* presents the AgiPL-AM model, which is an assessment model to evaluate the situation of agile adoption within agile product line approaches.

- *Chapter 8* proposes an integrated goal and feature-based metamodel for agile software product lines development. The proposed approach allows analysts and developers to specify requirements that precisely capture the stakeholder's needs and intentions as well as to manage product line variabilities.

- *Chapters 9 and 10* introduce a case study, as well as specific evaluations in order to validate the results of this dissertation.

- *Chapter 11* concludes this thesis, summarizes the work achieved, highlights the main contributions of the thesis and points out possible efforts in future work.

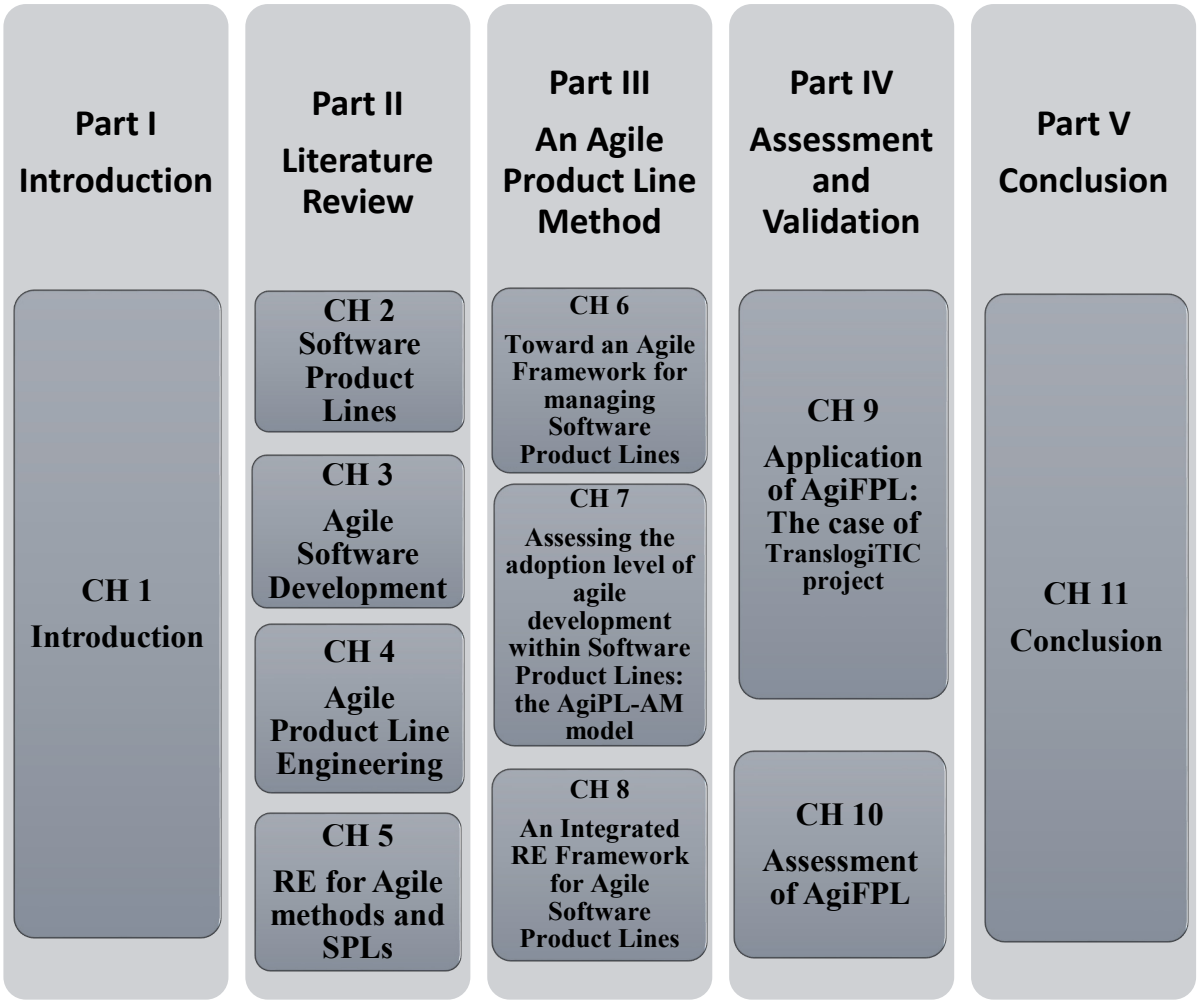

Fig. 1.3 Structure of the thesis - This figure presents the organization of the thesis, its parts and their respective chapters.

Part II – Literature Review

2 Software Product Lines

Abstract. *Software product lines* aim to empower software vendors to tailor products to the individual requirements of their customers. Their goal is to maximize the reusability of any previously developed software artifacts for the further development (derivation) of a multitude of other software products. The adoption of a *Software Product Line approach* is highly feasible when existing and/or planned software products exhibit a significant amount of commonality; and therefore have differences that can be considered as mere variability. This chapter provides a concise introduction to Software Product Lines. It comprehensively presents the key concepts and the motivation behind product-line development, the promises of product line engineering, and how the product lines are characterized in the context of handcrafting and mass production. In addition, it presents some frameworks and development processes of software product lines and their specific characteristics.

2.1 Introduction

There are differences between the software industry and classic mass production industries, such as automobile manufacturing, where product line engineering originally became the norm. Fundamentally, software is immaterial and could be duplicated with minimal material effort. Therefore, the main issue with software is the development, not production. However, the principles of "reusing" common parts for various products, and the further systematic management of their variability, could realistically be applied to software development [KäKölä & Dueñas, 2006].

Historically, work on software product lines (SPL) dates back to the 1970s, with the introduction of the *"program families approach"* introduced by Dijkstra [Dijkstra, 1972] and Parnas [Parnas, 1976]. Thereafter, in the early 1980s, Neighbors [Neighbors, 1980] developed the approach of *"Domain Engineering"*. Two main works followed these pioneering efforts. The first, presented by Kang et al., consist of systematic approaches of *"feature-oriented domain analysis (FODA)"* [Kang et al., 1990]. The second, developed by the *"Software Productivity Consortium Services Corporation"*, constituted a synthesis method called *"Reuse-driven software processes"* [Software Productivity Consortium Services Corporation, 1993]. The first systematic approaches to software product line engineering started emerging in the

late 1990s and early 2000s [Brownsword and Clements, 1996], [Weiss and Lai, 1999], [Clements and Northrop, 2001], [Atkinson, 2002], [Gomaa, 2005], or [Linden et al., 2007].

In the late 1990s, the need for *increased productivity* and *reduction of the time-to-market* were amid the essential factors that lead to the rise of new software development paradigms. Among these, the (Software) Product Line Engineering (SPLE or PLE) paradigm has gained significant momentum within the software industry [Apel et al., 2013]. The main principle is based on the construction of software systems from reusable parts, rather than developing them from scratch. Therewith offering the ability to tailor to stakeholders' individual requirements, by granting the opportunity to select from a large variety of configuration options [Clements & Northrop, 2001]. In fact, all systematic approaches to SPLE aim to reuse the commonality of software products and systematically handle the variability.

According to van der Linden, the strategic importance of PLE has been recognized by the software industry, especially companies developing within the same family of products or domain; such as smart phone manufacturers, car electronics companies, and financial services enterprises [van der Linden, 2002].

2.2 What is a Software Product Line (SPL)?

By reviewing the literature, several definitions for "Software Product Line" can be identified. The definition of SPL has evolved over time, as can be seen below on the schematic timeline in Figure 2.1; other (less common) definitions exist in the literature.

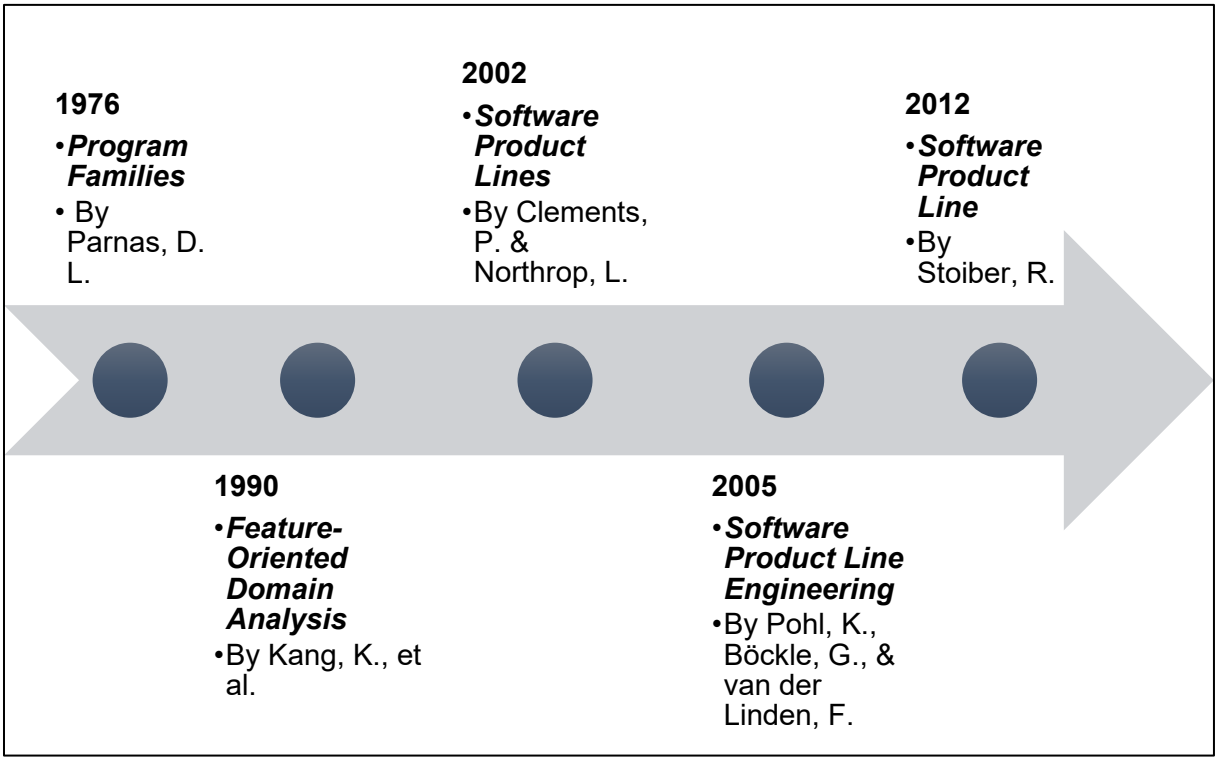

Fig. 2.1 SPL definitions timeline

In the 1970s, Dijkstra [Dijkstra, 1972] and Parnas [Parnas, 1976] have introduced in their publications the concept of *"Program Families"*. Parnas defines **"Program Families"** as *"sets of programs whose common properties are so extensive that it is advantageous to study the common properties to programs before analyzing individual members"* [Parnas, 1976].

Kang et al. [Kang et al., 1990] have advanced the field of SPL by introducing the concept of "Feature-Oriented Domain Analysis (FODA)". They consider that *the primary focus of the method is the identification of prominent or distinctive features of software systems in a domain. These features are user-visible aspects or characteristics of the domain. They lead to the creation of a set of products that define the domain and also give the method its name:* ***Feature-Oriented Domain Analysis (FODA)****. The features define both common aspects of the domain as well as differences between related systems in the domain*.

In 2002, Clements and Northrop [Clements & Northrop, 2001] have defined a ***software product line*** as *a set of software-intensive systems 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*.

Pohl et al. [Pohl et al., 2005] have presented the "**Software Product Line Engineering (SPLE)**". They defined SPLE as *paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization*. Where they consider a **platform** as *any base of technologies on which other technologies or processes are built*. The **mass customization** as *the large-scale production of goods tailored to individual customers' needs*. **Domain engineering** as *the process of software product line engineering in which the commonality and the variability of the product line are defined and realized.*

In his new approach to Product Line Engineering (PLE), Stoiber [Stoiber, 2012] define the **software product line** (SPL) as *a set of software products that all share a significant amount of commonality and differ in their specific configuration of variability*.

All the presented definitions contain two essential words explicitly or implicitly, namely *"commonality"* and *"variability"*. Identifying, documenting, and managing the communalities and variabilities are the key properties characterizing software product lines. Clements and Northrop [Clements & Northrop, 2001] focus their work less on variability modeling (e.g., feature modeling [Kang et al., 1990]), but rather on the managerial, social, and organizational aspects; which are decisive in the successful introduction of software product lines within companies. Contrastingly, Pohl et al. [Pohl et al., 2005] and Stoiber [Stoiber, 2012] have presented comprehensive approaches to PLE, in which they have introduce and emphasize key concepts for variability management (i.e. variability modeling, etc.). Therefore, PLE allows maximal reuse of both commonality and variability. Maximizing the reuse of commonality by developing all products on a common product platform. Maximizing the variability by utilizing a more modular development of variable functionalities, which can be added or removed at will.

2.3 Promises of Product Lines

The rise in Product Line Engineering (PLE) research was mainly driven by the potential of significant advancement in developmental efficiency. PLE promises better quality, lower cost,

and faster time-to-market [van der Linden et al, 2007] by systematic reuse of software assets and mass customization. The PLE paradigm takes advantage of the commonality found in sets of products by investing in the upfront design of the product-line platform [Pohl et al., 2005]. The upfront design is mainly composed of the common set of reusable core-assets, their variability, and the Product Line Architecture (PLA). Hence, the developed assets are assembled into customer-specific products by deriving the existing variability [Northrop & Jones, 2012]. The most important benefits are presented below [Apel et al., 2013] [Clements & Northrop, 2001]:

- **Tailor-made products:** adopting a product line approach to software production facilitates tailoring products to individual customers (stakeholders). A software vendor could produce a whole set of customized products rather than providing a standardized product, or alternatively a smaller set of preconfigured products, such as Community, Professional, and Enterprise editions of the same software product;

- **Increased software quality:** Industrial mass production has improved quality due to the use of systematically controlled standardized parts. In fact, by adopting a PLE approach, the extensive quality assurance implies a significantly higher efficacy of error detection and regulation, thereby increasing the quality of all further derived products. In contrast to handcrafting, commonly used parts lead to more stable, lean, and reliable products;

- **Significant reduction of development costs:** While providing each customer their desired software solution, product-line vendors do not need to pay the cost of designing and developing each product from scratch. Instead, they develop reusable parts (artifacts) that can be combined in different ways. Figure 2.2 illustrates the economic promise of software product lines. This figure shows the ***cost/effort*** needed to develop ***n*** different products. The blue line sketches the costs/effort required for developing the product from scratch, while the green line sketches the costs/effort for product-line engineering. The development cost per product per customer can be reduced by selecting which parts to combine (potentially add missing ones) and by testing the resulting product. While the *upfront investment* required for such an approach is certainly larger than developing a single software product (i.e. the need to design reusable parts and implement variations that might not be required for the first product), the approach pays off in the long term; when multiple tailored products are requested. The location at which both curves intersect marks the *break-even point*. At this point, the costs are the same for developing the systems separately as for developing them by product line engineering. The precise location of the break-even point depends on various characteristics of the organization and the market it has envisaged, such as the customer base, the expertise, and the range and kinds of products. The strategy that is used to initiate a product line also influences the break-even point significantly [McGregor et al., 2002];

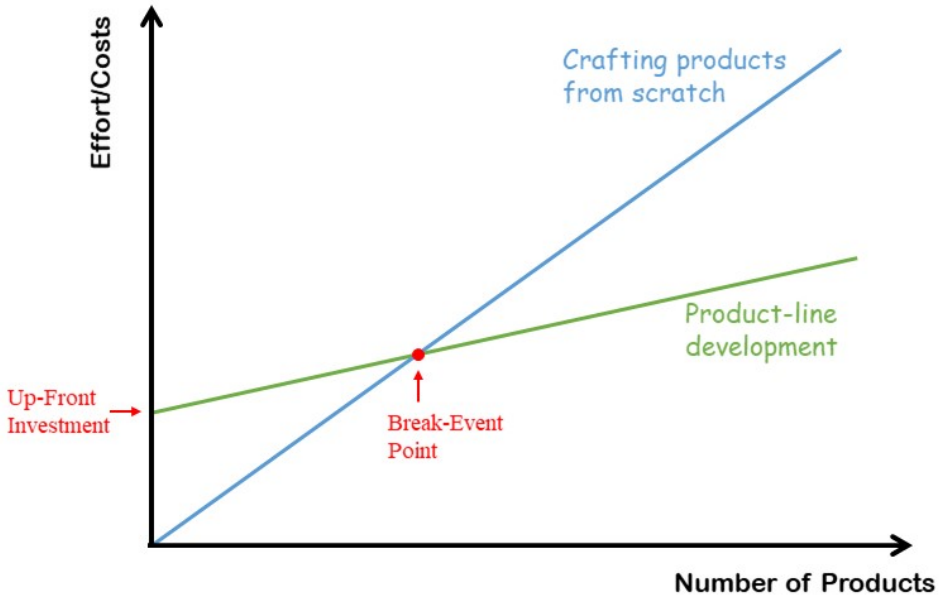

Fig. 2.2 Effort/costs of crafting products individually versus product-line development [Pohl et al., 2005].

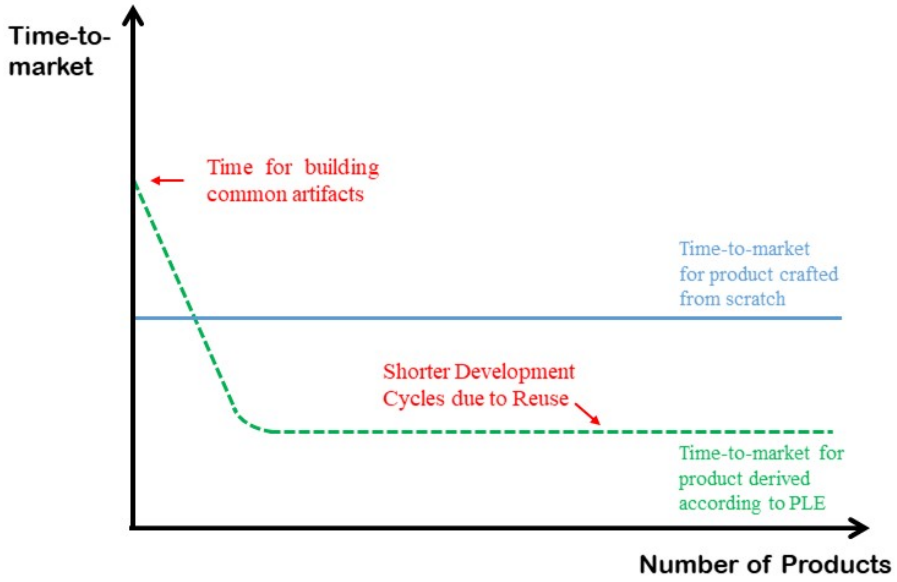

Fig. 2.3 Time to market with and without product line engineering [Pohl et al., 2005].

- **Faster product development cycles and time-to-market:** While standard software is readily available, handcrafted software products entail significantly higher costs and, more importantly, longer development time, before they can be released. If a customer

selects only from predefined configuration options (that map to existing parts), a software vendor can quickly produce the corresponding software product by assembling the existing, corresponding parts. Even if a customer requests functionality that has not been prepared, building a new product on top of existing well-designed, reusable parts is much faster than developing it entirely from scratch. A *well-designed* platform that can be extended for new products promises the possibility to quickly react to market changes, more so than standardized software or individual development ever could [Apel et al., 2013]. In Figure 2.3, the time-to-market for single-product development is assumed to be roughly constant[1]. Figure 2.3 shows that for PLE, the time-to-market is initially higher; as the common artifacts have to be built first. Yet, after having passed this hurdle, the time to market is considerably shortened as many artifacts can be reused for each new product [Pohl et al., 2005].

In addition to that, Clements and Northrop [Clements & Northrop, 2001] list even more benefits such as *"productivity gains in general"*, *"increased market presence"*, *"unprecedented growth"*, *"improved customer satisfaction"*, *"higher reuse goals"*, *"mass customization"*, and *"compensation for an inability to hire software engineers"*.

2.4 Key concepts of Software Product Lines

In general, the PLE community agrees that the structural entities of a product-line are the reusable core assets or artifacts, which support different degrees of customization through defining variability points. The main concepts of PLE are defined as follows:

- ***Core asset base [Clements & Northrop, 2001] or Platform [Pohl et al., 2005]***: set of assets that form the basis (i.e. a common structure) from which a set of products of the same family can be efficiently developed. That core asset base may include requirements statements, documentation and specifications, domain models, architecture descriptions, reusable software components, test cases, work plans, process descriptions, etc. Therefore, a *"core asset base"* or *"platform"* is any *"base of technologies on which other technologies or processes are built"* [Pohl et al., 2005];

- ***Asset***: package of relevant artifacts which provides 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***: a piece of information (requirements specification, architecture, code, tests, etc.) that is produced, modified, or used by a process and may take various shapes [Thiel and Hein, 2002]. Pohl et al. [Pohl et al., 2005] define "Development Artifact" as the output of a sub-process of domain or application engineering that encompass

[1] In practice, the time-to-market for single-system varies, but to exhibit the effect of single-system vs. product line engineering this assumption is sufficiently accurate [Pohl et al., 2005].

requirements, architecture, components, and tests. In addition, they name *"Domain Artifacts"* the reusable *"artifacts created in the sub-processes of domain engineering"*;

- ***Feature***: a feature[2] is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all the phases of the software's life cycle [Apel et al., 2013]. This term started 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 product lines;

- ***Variability***: variability is the ability to derive different products from a common set of artifacts [Clements et al., 2010], also known as anticipated change, i.e. change that is mostly foreseen [Bachmann and Bass, 2001]. Variability is the basis for mass customization i.e. the ability to specify flexibility and therewith enable the development of customized applications from a product-line (PL) [Pohl et al., 2005]. According to [Kang et al., 1990], [Pohl et al., 2005], and [Apel et al., 2013] variability is achieved by intentionally defining the variation type (i.e. optional, mandatory, alternative, and multiple), the time in which variation occurs (i.e. at design, compilation, or runtime), the places where products can differ, and rationale (i.e. intent and motivation to define that variability). Variability is specified in terms of variability points (or variation point) and variants;

- ***Variability Point***: Also known as, (variation point) is an explicit engineering decision, which permits several *optional*, *mandatory*, *alternative*, or *multiple* variants in regard to selected assets of system development [Bachmann et al., 2004]. Pohl et al. [Pohl et al., 2005] consider that a variation point is a representation of a variability subject within domain artifacts enriched by contextual information. According to [Clements et al., 2010], variability points are the exact places in the core assets where a specific kind of flexibility has been built-in (i.e. the locations at which variations occur). A variability point could represent the choice among a number of functional features available to the stakeholders, 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 [Clements et al., 2010]. A variant is a representation of a variability within domain artifacts [Pohl et al., 2005];

- ***Product***: a product, of a product line, is specified by a valid feature selection (a subset of the features of the product line), where a feature selection is valid if, and only if, it fulfills all feature dependencies [Apel et al., 2013];

[2] Features are a fundamental notion in modern software engineering. The notion of a feature has many definitions presented in Chapter 5. In addition, in Chapter 5 a further explaination will be give regarding the choice of this particular definition for the presentwork.

- ***Domain***: a domain is an area of knowledge that is scoped to maximize the satisfaction of the requirements of its stakeholders. It includes a set of concepts and terminology understood by practitioners in that area, and includes the knowledge of how to build software systems (or parts of software systems) within that area [Czarnecki and Eisenecker 2000] [Apel et al., 2013];

- ***Development***: the generic term used to describe how core assets (or products) are realized. The term *development*, used within this thesis, may actually involve building, acquisition, purchase, retrofitting earlier work, or any combination of these options. Throughout the work, these options are recognized and addressed, but the term *development* is mostly used to refer to the broader meaning.

The literature review shows that some practitioners use different sets of terms to convey essentially the same meaning. For a few, a *product line* is a profit and loss center solely concerned with turning out a set of products; it refers more to a business unit, than a set of products. The *product family* is the set of products, which is called the *product line*. The software assets in the *core asset base* are sometimes called a *platform*. What was identified as *core asset development* is sometimes referred to as *domain engineering (DE)*, and what was identified as *product development* is sometimes referred to as *application engineering (AE)*.

2.5 Product Line Architecture (PLA)

According to Nakagawa et al. [Nakagawa et al., 2011], "*the Product Line Architecture (PLA) refers to a structure that encompasses the behavior from which software products are developed*". Moreover, Pohl et al. [Pohl et al., 2005] defined PLA as the *"core"* architecture that represents the product-line high-level design; considering variation points and variants documented in the variability model. The Product Line Architecture (PLA) is a fundamental artifact of the product line engineering, and contains all commonalities and variabilities of the product-line (PL), and thus, it represents an abstraction of the products that can be generated [van der Linden et al., 2007]. The PLA plays a central role to successfully generate specific products taking into account the development and evolution of a PL. It abstractly represents the architecture of all potential products from a specific domain. The quality attributes of a PLA have impacts on the PL performance. For example, establishing the adequate PLA will (1) increase the productivity of the product line process and the quality of the products; (2) provide a means to understand the potential behavior of the products and, consequently, (3) decrease their time-to-market; and, (4) improve the handling of the product line variability.

The PLA addresses the product-line design decisions by means of their commonalities, as well as variabilities [Taylor et al., 2009]. The PLA design can benefit from the use of PL characteristics such as feature modularization. In addition, the adoption of architectural styles benefits the PLA design and provides flexibility, maintainability and understandability [Gomaa, 2004]. In addition, an architectural style determines ways to select and show parts of an architecture, providing a specific organization of the elements, in order, to improve the architecture understandability. To do this, an architectural style defines specific components, connectors and a set of rules of how these can be combined [Colanzi et al., 2014]. The adoption

of a style makes the system easier to maintain, and can reduce fault propagations due to changes [Mariani et al., 2016].

There are two common styles that are usually adopted in the SPL context, namely, *layered Style* and *Client/Server Style*. The *layered architectural style* eases extension and contraction. In the PL context, software contraction regards to the design of mandatory elements, and software extension regards to the design of variables elements. The use of the *client/server style* also eases PLA evolution, since servers can be easily added or removed [Gomaa, Hussein, 2004].

Several methods were proposed in order to answer the needs of product lines from the software architectural point of view. Mentioned hereafter are some of the most common product line approaches:

i. COPA [America et al., 2000];
ii. FAST [Weiss et al., 1999] [Harsu, 2002];
iii. KobrA [Atkinson and Muthig, 2002];
iv. FORM [Kang et al., 2002];
v. QADA [Matinlassi et al., 2002];
vi. PLUS [Gomaa, 2005];
vii. CVL [Haugen et al., 2013];
viii. Product line engineering and management [ISO/IEK 26550:2017].

2.5.1 Component-Oriented Platform Architecting Method (COPA)

A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products (COPA) [America et al., 2000] is currently under development at the Philips Research Labs. The COPA method is one of the results of the Gaudi project [Philips et al., 1999]. The ambition behind the Gaudí project is "*to make the art and emerging methodology of System architecture more accessible and to transfer this know-how and skills to a new generation of system architects*" [Matinlassi, 2004].

The COPA method is a component-based product line methodology that provides a set of component-based subsystems and interfaces (with their associated processes, documentation and tools) from which a stream of derivative and composite products can be developed and produced, according to a domain specific architecture or product family architecture [America et al., 2000]. The main goal of the COPA method is to harmonize business, architecture, process and organization [Matinlassi, 2004]. COPA uses the *Business-Architecture-Process Organization (BAPO) model* [van Ommering, 2002] to cover multiple aspects of the product line life cycle like the business drivers, architecture, processes and organizational concerns. BAPO starts by identifying the *business needs* for the product line; which might be an improvement of an existing product line, or the need for a new one. After, BAPO defines the

PLA, in which, the *domain* of the product line is defined. In fact, at this step, the systems and components are defined and structured to fit the PLA. The process phase of BAPO creates the architecture previously defined, while identifying component dependency, commonality and variability. The organization aspect of BAPO covers *organizational support* for the product-line. It ensures that the product-line matches the organization's business needs and it provides management support. In addition, it defines processes for product-line maintainability and evolution [van Ommering, 2002] [Tzeremes, 2016].

COPA is an extensive method targeted to all interest groups of a software company. Especially, the architecture stakeholders of the COPA method are the customers, suppliers, business managers and engineers. The incentive to use COPA is a promise to efficiently manage size and complexity, obtain high quality, manage diversity and a significant reduction in lead-time [Muller, 2004].

2.5.2 Family-Oriented Abstraction, Specification, and Translation process (FAST)

Weiss et al. [Weiss et al., 1999] have introduced a practical, family-oriented software production process known as the Family-Oriented Abstraction, Specification, and Translation process. The FAST [Weiss et al., 1999] process is an alternative to the traditional software development process. It is highly applicable when creating multiple versions of a product sharing significant common attributes, such as common behavior, interfaces, or codes. The objective of FAST is to make the software engineering process more efficient by reducing multiple tasks, decreasing production costs, and shortening the marketing time [Harsu, 2002]. According to [Weiss et al., 1999], the FAST method considers the product line aspects and defines a full product line engineering process complete with activities and artifacts. The FAST process can be divided into three sub processes of the product line:

1. Domain Qualification;
2. Domain Engineering;
3. Application Engineering.

During domain qualification, product families are identified and a justification is made for their creation. Domain engineering covers the analysis and implementation of the domain. During domain analysis, product line functionality is abstracted and a common platform for product line family creation is designed. Domain implementation creates and implements the common platform. Application engineering uses the platform, created in domain engineering, to create product line family members.

The FAST method originated from the industrial world and has a highly practical background. Therefore, FAST seems to be aimed at software engineers and designers currently working in the industry. The use of the FAST method can alleviate the problems, which make the software developers' task such a lengthy and costly one.

2.5.3 Component-Based Application Development (KobrA)

The KobrA method was developed at Fraunhofer-IESE [Fraunhofer-IESE, 2001] as a methodology for modeling architectures. The KobrA method [Atkinson et al., 2002] stands for *Komponentenbasierte Anwendungsentwicklung,* which translated from German, means "*Component-Based Application Development*".

According to [Atkinson et al., 2000] [Atkinson and Muthig, 2002] [Atkinson et al., 2002], KobrA is a component-based incremental product line development approach or a methodology for modeling architectures. It is also designed to be suitable for both single system and family based approaches in software development. In addition, the approach can be viewed as a method that supports a Model Driven Architecture (MDA) [Frankel, 2003] approach to software development; in which the essence of a system's architecture is described as independent from platform idiosyncrasies. Another important goal of this method is to be as concrete and prescriptive as possible and make a clear distinction between the products (i.e. artifacts) and processes [Matinlassi, 2004].

Based on KobrA, software elements are created individually and are synthesized in different ways to create different members of the product line. KobrA has two main phases:

1. Framework Engineering;
2. Application Engineering.

Framework engineering analyses the commonality and variability of the product line and creates generic framework that represents all variations of the product line, whilst also including information about the common and variant features. *Application engineering* is responsible for instantiating the generic framework and create different product variants based on customer specifications.

KobrA is stated as a simple, systematic, scalable and practical method [Atkinson et al., 2002]. By *simple,* the authors mean that the method is as highly economically efficient with its concepts, and that the features in a method should be as orthogonal as possible. The term s*ystematic* refers to the fact that the concepts and guidelines defined in the method should be precise and unambiguous. A *scalable* method provides two aspects of scalability; namely granularity scalability and complexity scalability. The first one means that a method should be able to accommodate large-scale and small-scale problems in the same manner using the same basic set of concepts, whereas fulfillment of the latter refers to incremental application of the method concepts. *Practicality* requires a method to be compatible with as many commonly used implementation and middleware technologies as possible.

2.5.4 Feature-Oriented Reuse Method (FORM)

Kang et al. [Kang et al., 1998] have proposed a feature-oriented method called "Feature-Oriented Reuse Method". The FORM method extends FODA method [Kang et al., 1990] for

the software design and implementation phases and prescribes how the feature model is used to develop domain architectures and components for reuse.

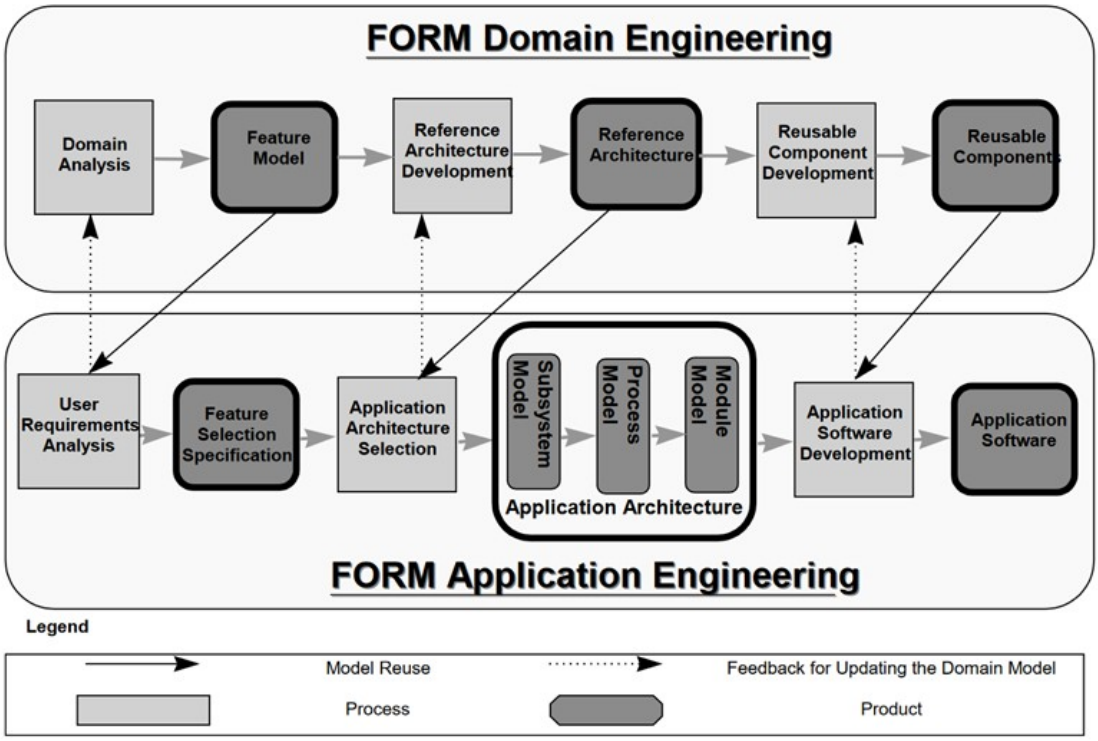

Fig. 2.4 FORM Engineering Processes [Kang et al., 1998].

According to [Kang et al., 2002], when examining a class/family of related systems and the *commonality* underlying those systems, it is possible to obtain a set of reference models, i.e., *software architectures* and *components* needed for implementing applications in the class. FORM (Feature-Oriented Reuse Method) supports development of such reusable architectures and components (through a process called the *Domain Engineering*) and development of applications (through a process called the *Application Engineering*) using the domain *artifacts* produced from the domain engineering.

FORM has a specialized manner of applying domain analysis results (i.e. commonality and variability) to the engineering of reusable and adaptable domain components with specific guidelines. It starts with feature modeling to discover, understand, and capture commonalities and variabilities of a product line. Domain engineering starts from the beginning of the software development (i.e. context analysis). The primary input is the information of the systems that share a common set of capabilities and data.

Figure 2.4 shows the engineering processes of FORM. Domain engineering creates the feature model, reference architecture, and reusable components as an output. Application engineering creates the application software after; features selection from the feature model, application architecture selection from reference architecture and reusable components have been selected from reusable components. FORM caters to the wide spectrum of domain and

application engineering, including the development of reusable architectures and code components. It is applied during software engineering in many industrial processes.

2.5.5 Quality-driven Architecture Design and quality Analysis (QADA)

Quality-driven Architecture Design and quality Analysis (QADA) is a methodology that provides a set of methods and techniques to develop high-quality software architectures for single systems and system families.

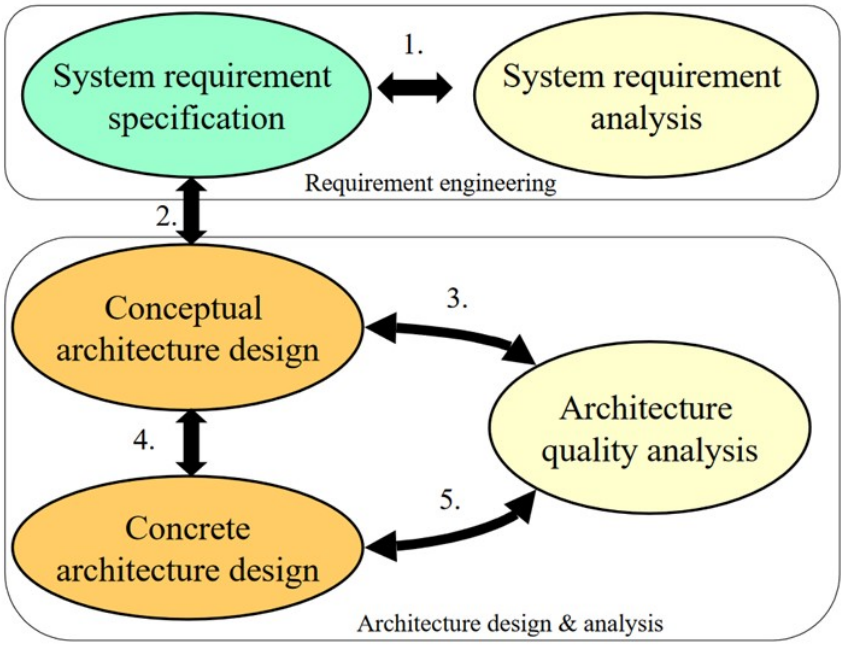

Fig. 2.5 QADA method main phases [Matinlassi et al., 2002].

The methodology is mostly intended for the development of service architectures, applied in pervasive computing environments and other networked systems [Ovaska, 2008]. [Matinlassi et al., 2002] have developed the QADA method at VTT (Technical Research Centre of Finland). According to the QADA method, the quality requirements are the driving force when selecting software structures and each viewpoint concerns certain quality attributes [Matinlassi, and Niemelä, 2002]. Architecture design is combined with quality analysis, which determines whether the designed architecture meets the quality requirements set from the very beginning.

The QADA method describes the architectural design part of the software development process, including steps and artifacts produced in each step. It also covers the description language used in the artifacts. It does, however, not cover any organizational or business aspects. The method starts with the Requirements Engineering (RE) phase. The aim is to collect the "*driving ideas of the system and the technical properties on which the system is to be designed*" [Matinlassi et al., 2002]. In addition to functional properties, the quality requirements and constraints of the system are captured as input.

Figure 2.5 presents the main phases of the QADA method. In fact, the output of the QADA method is twofold: *design* and *analysis*.

- *Design* covers software architecture at two abstraction levels: *conceptual* and *concrete*.
  - *Conceptual* architecture covers the conceptual components, relationships and responsibilities, which are intended to be used by certain high level stakeholders related to product line, e.g. product line architects or management.
  - *Concrete* architecture is closer to the so-called 'traditional' architecture description aimed for software engineers and designers.
- *Analysis* provides precious information concerning the quality of the design. Analysis results in feedback on whether the design meets the quality requirements defined for the system.

The QADA method does not produce final implementation artifacts. The method users are mainly product line architects, software architects or an architecting team. However, the group of stakeholders that use the method output is much wider. At the conceptual level, the stakeholders include system architects, service developers, product architects and developers, maintainers, component designers, service users, project manager and component acquisition. The concrete level, on the other hand, has the architectural descriptions, which are aimed at component designers, service developers, product developers, testing engineers, integrators, maintainers and assets managers. These groups continue to implement, test or maintain the deigned architecture. As almost all methods do, QADA claims to be a systematic method, simple to learn and conforming to the IEEE standard for architectural description [IEEE, 2007].

2.5.6 Product Line UML-Based Software Engineering

Product Line UML-Based Software Engineering (PLUS) is defined as a design method for software product lines that describes how to conduct *requirements*, *analysis*, and *design modeling* for software product lines in UML [Gomaa, 2005]. PLUS builds on the COMET method by considering the added dimension of variability in each of the modeling views.

Figure 2.6 shows the evolutionary software process model for the software product lines according to PLUS method [Gomaa, 2011]. According to [Gomaa, 2011] [Gomaa, 2005] [Tzeremes, 2016], the PLUS requirements phase detects the product-line *Use Cases* and tags them as optional and/or variant. Feature analysis identifies the product line features and maps them to the use cases. During the analysis phase, PLUS examines the problem domain and develops the system context diagram, collaboration, or sequence diagrams, and state diagrams. The analysis phase concludes with feature or class dependency diagrams and tables that show the classes that implement features. In the design phase, PLUS examines the solution domain, develops the product-line architecture, and structures the system into subsystems and components. The design phase ends with defining the communication interface of each component. In the component implementation phase, software engineers select a subset of the designed functionality for development. The product-line testing phase performs integration testing among the components developed on the increment with the existing components of the

product line and functional testing assessing the functionality of the increment. All artifacts generated by PLUS are stored in the software product line repository.

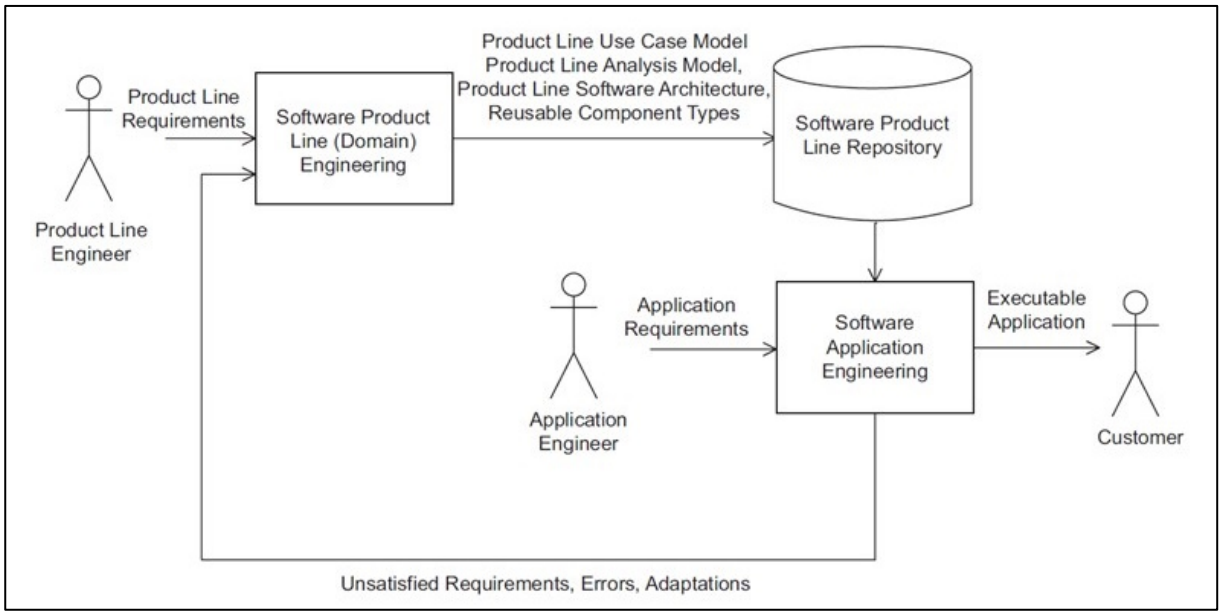

Fig. 2.6 Evolutionary software process model for software product lines according to PLUS method [Gomaa, 2011].

2.5.7 Common Variability Language (CVL)

According to [Haugen et al., 2013], the Common Variability Language (CVL) is a domain-independent language for specifying and resolving variability. It facilitates the specification and resolution of variability over any instance of any language defined using a *Meta Object Facility (*MOF)-based meta-model.

The Common Variability Language (CVL) is used to add variability to Model Driven Architecture (MDA) models. In particular, CVL is a *Domain Specific Language (DSL)* for modeling variability in models based on *MOF* standard defined by the Object Management Group (OMG) [Reinhartz-Berger et al., 2014]. CVL operates on three models:

1. The *base model* is a domain model for a particular system;
2. The *variability model* describes variations of the system;
3. The *resolution model* captures a set of options on the variability model.

To create a new system, CVL takes as input the aforementioned three models and generates new resolved models. Existing DSL tools can operate on the resolved models and transform them to working software.

The CVL approach is, by nature, an orthogonal composition-based approach, as elements of the base model can, through the CVL variation points, be composed, removed, substituted, etc. Variation points specify how the elements of the base models are modified by defining specific modifications to be applied [Horcas et al., 2018].

2.5.8 Product line engineering and management (ISO/IEK 26550:2017)

The international standard for software and systems engineering, being the "Reference model for product line engineering and management" [ISO/IEC 26550:2017, 2017], aims to create a common vocabulary and standardized process for product line creation. The standard covers the *domain* and *application* engineering aspects for creating a product line.

Domain engineering covers: *product line scoping*, *domain requirements engineering*, *domain design*, *domain realization* and *domain validation and verification*. During domain engineering, the organizational management works with the technical management to perform the initial product line scoping. Product line scoping involves the identification of market groups, product categories, common and variable features, functional domains for envisioned features (with sufficient possibility of reuse), reusable assets for creating products, and a cost benefit analysis for each domain asset. After the product line is scoped domain requirements engineering is performed, which identifies the product line stakeholders and captures detailed requirements.

Domain design is used to perform commonality and variability analyses, feature modeling and finally to define the domain architecture. Domain realization is responsible for component design and implementation. Domain validation and verification assure the quality of the product line. All domain assets defined during domain engineering are stored on the domain asset repository.

Application engineering process in the *ISO* involves *application requirements engineering*, *application design*, *application realization* and *application verification and validation*. Application requirements engineering develops application-specific requirements reusing common and variable requirements defined during domain requirements engineering. Application design derives the application architecture from the domain architecture in order to meet application requirements. Application realization implements product line members by drawing upon the application requirements and architecture; reusing and configuring domain components and interfaces. Application verification and validation ensures that the right member product and the right application assets have been modeled, specified, designed, built, verified, and validated. All artifacts created by the application engineering process are stored in the application asset repository.

2.6 A Framework for Software Product Line Engineering

In 2005, Pohl et al. [Pohl et al., 2005] have proposed a reference framework for software product-line engineering that incorporates the central concepts of traditional product line engineering, namely the use of *platforms* and the ability to provide *mass customization*.

In their proposed framework, they consider a layered architectural style for the product line. Idem as [Weiss and Lai, 1999] [Boeckle et al., 2004] [Pohl et al., 2001], [Pohl et al., 2005] consider that the SPLE paradigm separates two processes:

1. *Domain engineering (DE):* This process is responsible for establishing the reusable platform and thus for defining the commonality and the variability of the product line. The platform consists of all types of software artifacts (requirements, design, realization, tests, etc.). Traceability links between these artifacts facilitate systematic and consistent reuse;

2. *Application engineering (AE):* This process is responsible for deriving product line applications from the platform established in domain engineering. It exploits the variability of the product line and ensures the correct binding of the variability according to the applications' specific needs.

The advantage of this division is that there is a separation of the two concerns of building a robust platform and creating customer-specific applications in a short time. To be effective, the two processes must interact in a manner that is beneficial to both. For example, the platform must be designed in such a way that it is of use for application development, and application development must be aided in using the platform.

The separation into two processes also indicates a partition of concerns with respect to variability. Domain engineering is responsible for ensuring that the available variability is appropriate for producing the applications. This involves common mechanisms for deriving a specific application. The platform is defined with the right amount of flexibility in many reusable artifacts. A large part of application engineering consists of reusing the platform and binding the variability as required by the different applications.

2.6.1 The framework

The PLE framework of Pohl et al. [Pohl et al., 2005] is based on the approach of [Weiss and Lai, 1999], which makes the differentiation between the domain and application engineering processes. Moreover, the proposed framework has its roots in several projects, namely the ITEA[3], ESAPS[4], CAFÉ[5], and FAMILIES[6] projects.

Figure 2.7 presents the framework of Pohl et al. proposed as a software product line engineering approach for product lines. The upper part of the figure depicts the Domain Engineering (DE) Process. The DE process is composed of five key sub-processes: product management, domain requirements engineering, domain design, domain realization, and domain testing. The domain engineering process produces the platform, including the commonality of the applications and the variability to support mass customization.

[3] Information Technology for European Advancement
[4] Engineering Software Architectures, Processes and Platforms for System-Families
[5] Concepts to Application in System-Family Engineering
[6] FAct-based Maturity through Institutionalization Lessons-learned and Involved Exploration of System-family engineering

The lower part of Figure 2.7 depicts the Application Engineering (AE) Process, which is composed of the sub-processes application requirements engineering, application design, application realization, and application testing.

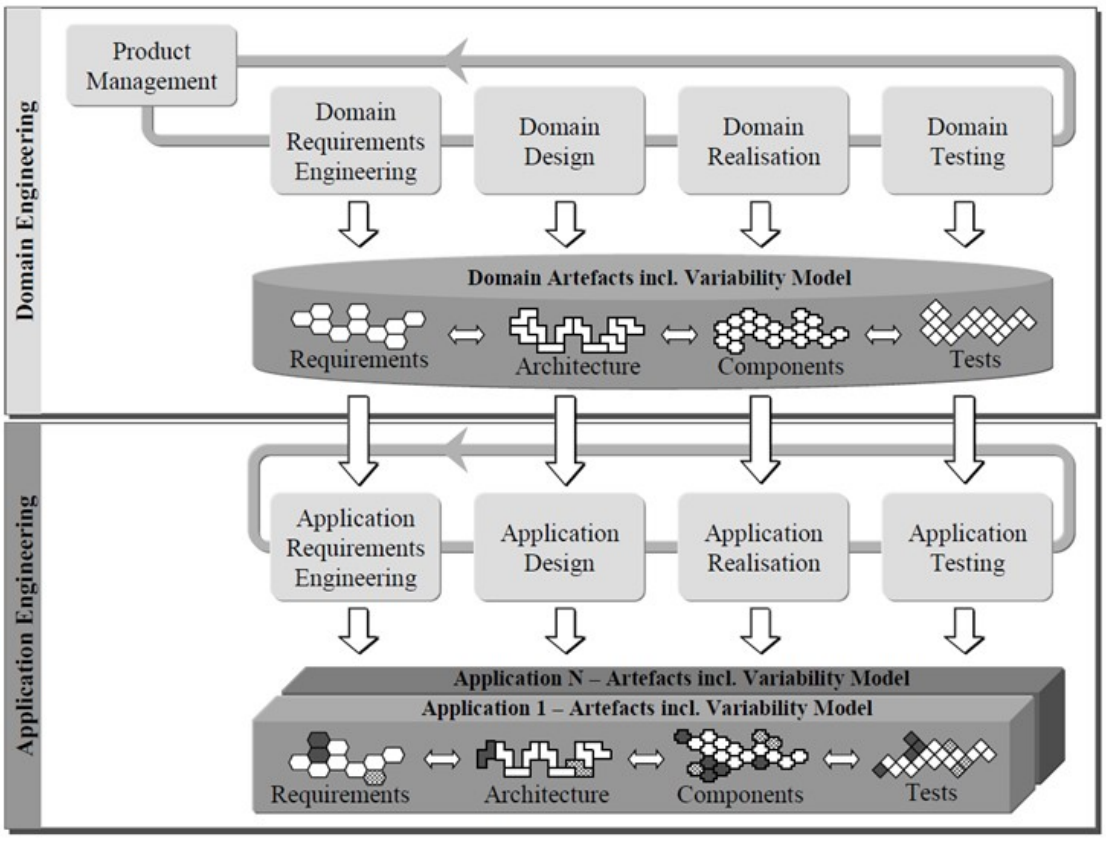

Fig. 2.7 Pohl et al. Framework for Software Product Line Engineering (SPLE / PLE) [Pohl et al., 2005].

The framework differentiates between different kinds of development artifacts. In their book [Pohl et al., 2005], they define *development artifact* as the output of a sub-process of domain or application engineering. In fact, development artifacts encompass requirements, architecture, components, and tests. The development artifacts are:

1. *Domain artifacts*, which are reusable development artifacts created in the sub-processes of domain engineering;
2. *Application artifacts*, which are the development artifacts of specific product line applications.

2.6.2 Domain Engineering (DE)

The domain engineering process has two main goals:

1. Define the commonality and the variability of the software product line;
2. Define the set of applications the software product line is planned for, i.e. define the scope of the software product line.

The sub-processes of the domain engineering stated above accomplish these goals. Each one of these sub-processes has to:

- Detail and refine the variability determined by the preceding sub-process;
- Provide feedback about the feasibility of realizing the required variability to the preceding sub-process.

2.6.2.1 Sub-processes of Domain Engineering

Each sub-process of the domain engineering sub-processes has a specific mission and description. The domain engineering sub-processes are the following:

1. *Product Management (ProM)*: ProM deals with the economic aspects of the software product line, in particular the market strategy. Its main concern is the management of the product portfolio of the company or business unit. In product line engineering, product management employs scoping techniques to define what is within and outside of the scope of the product line;

2. *Domain Requirements Engineering (DRE)*: the DRE sub-process encompasses all activities for eliciting and documenting the common and variable requirements of the product line;

3. *Domain Design (DD)*: The DD sub-process encompasses all activities for defining the reference architecture of the product line. The reference architecture provides a common, high-level structure for all product line applications;

4. *Domain Realization (DR)*: The DR sub-process deals with the detailed design and the implementation of reusable software components;

5. *Domain Testing (DT)*: DT is responsible for the validation and verification of reusable components. Domain testing assesses the components on their specification, i.e. requirements, architecture, and design artifacts. In addition, domain testing develops reusable test artifacts to reduce the effort for application testing.

2.6.2.2 Domain Engineering artifacts

Domain artifacts (or domain assets), produced by the aforementioned sub-processes, compose the platform of the software product line and are stored in a common repository. The artifacts are interrelated by traceability links to ensure the consistent definition of the commonality and the variability of the software product-line throughout all artifacts. In the following list, each kind of artifact, including the variability model, is briefly characterized. Domain artifacts are the following:

1. *Product Roadmap*: The product roadmap describes the features of all applications of the software product line and categorizes these into common features, which are part of each application, and variable features, that are only part of some applications. In addition, the roadmap defines a schedule for market introduction;

2. *Domain Variability Model*: The domain variability model defines the variability of the software product line. It defines what can vary, i.e. it introduces variation points for the product line. It also defines the types of variation offered for a particular variation point, i.e. it defines the variants offered by the product line. Moreover, the domain variability model defines variability dependencies and variability constraints, which have to be considered when deriving product line applications;

3. *Domain Requirements*: Domain requirements encompass requirements that are common to all applications of the software product line, as well as variable requirements; which enable the derivation of customized requirements for different applications;

4. *Domain Architecture*: The domain architecture or reference architecture determines the structure and the texture of the applications in the software product line. The structure determines the static and dynamic decomposition that is valid for all applications of the product line. The texture is the collection of common rules guiding the design and realization of the parts, and how they are combined to form applications;

5. *Domain Realization Artifacts*: Domain realization artifacts comprise the design and implementation artifacts of reusable software components and interfaces. The design artifacts encompass different kinds of models that capture the static and the dynamic structure of each component. The implementation artifacts include source code files, configuration files, and make files;

6. *Domain Test Artifacts*: Domain test artifacts include the domain test plan, the domain test cases, and the domain test case scenarios. The domain test plan defines the test strategy for domain testing, the test artifacts to be created, and the test cases to be executed.

2.6.3 Application Engineering (AE)

Application engineering process has several operational goals. The key goals of the application engineering process are the following:

- Achieve an as high as possible reuse of the domain assets when defining and developing a product line application;

- Exploit the commonality and the variability of the software product line during the development of a product line application;

- Document the application artifacts, i.e. application requirements, architecture, components, and tests, and relate them to the domain artifacts;

- Bind the variability according to the application needs from requirements over architecture, to components, and test cases;

- Estimate the impacts of the differences between application and domain requirements on architecture, components, and tests.

2.6.3.1 Sub-processes of Application Engineering

The proposed SPLE framework introduces four Application Engineering sub-processes:

1. *Application Requirements Engineering (ARE)*: The ARE sub-process encompasses all activities for developing the application requirements specification. The achievable amount of domain artifact reuse depends heavily on the application requirements. Hence, a major concern of application requirements engineering is the detection of deltas between application requirements and the available capabilities of the platform;

2. *Application Design (AD)*: The application design sub-process encompasses the activities for producing the application architecture. Application design uses the reference architecture to instantiate the application architecture. It selects and configures the required parts of the reference architecture and incorporates application specific adaptations;

3. *Application Realization (AR)*: The application realization sub-process creates the considered application. The main concerns are the selection and configuration of reusable software components as well as the realization of application-specific assets. Reusable and application-specific assets are assembled to form the application;

4. *Application Testing (AT)*: The application testing sub-process comprises the activities necessary to validate and verify an application as well as its specification.

2.6.3.2 Application Engineering artifacts

Application artifacts (or application assets) comprise all development artifacts of a specific application including the configured and tested application itself. The sub-processes already

described produce them. The application artifacts are interrelated by traceability links. These artifacts are:

1. *Application Variability Model (AVM)*: The application variability model documents, for a particular application, the binding of the variability, together with the rationales for selecting those bindings. It is restricted by the variability dependencies and constraints defined in the domain variability model. Moreover, the application variability model documents extensions to the domain variability model that have been made for the application;

2. *Application Requirements (AR)*: Application requirements constitute the complete requirements specification of a particular application. They comprise reused requirements as well as application-specific requirements;

3. *Application Architecture (AA)*: The application architecture determines the overall structure of the considered application. It is a specific instance of the reference architecture. To ensure the success of a product line, it is essential to reuse the reference architecture for all applications. Its built-in variability and flexibility should support the entire range of application architectures;

4. *Application Realization Artifacts (ARA)*: ARA encompass the component and interface designs of a specific application, as well as the configured, executable application itself. The required values for configuration parameters can be provided, for example, via configuration files. “Make files” or the “run-time system” evaluates these parameter values, for example. The values can be derived from the application variability model;

5. *Application Test Artifacts (ATA)*: ATAs comprise the test documentation for a specific application. This documentation makes application testing traceable and repeatable. Many application test artifacts can be created by binding the variability of domain test artifacts, which is captured in the orthogonal variability model.

2.7 Feature-Oriented Product line: a development process

Nineteen years after the introduction of the FODA method by Kang et al. [Kang et al., 1990], Apel and Kästner [Apel and Kästner, 2009] have proposed a feature-oriented approach named *feature-oriented software development* (FOSD). This approach was proposed in order to highlight another research area that focuses on the development of product *features*. In the proposed approach, they consider a *feature* as a *“unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option”*.

The basic idea of FOSD is to decompose a software system in terms of the features this system provides. Furthermore, they call the set of software systems (products) generated

(derived) from a set of features a *"software product-line"*. Apel and Kästner have demonstrated that the key principle for FOSD is the use of feature as *first-class entities* to analyze, design, implement, customize, debug, or evolve a software system [Apel and Kästner, 2009].

In 2013, Apel et al. [Apel et al., 2013] have proposed a Feature-Oriented approach for software product lines. In this approach, they detail the related processes, concepts and implementation.

2.7.1 A Process for Product-Line Development

A feature-oriented development process for software product lines has to take into account two main issues that play a crucial role on defining the structure of the process:

1. The explicit handling of variability;
2. The systematic reuse of implementation artifacts.

In fact, for both issues, it is imperative to have appropriate structuring of process and software artifacts.

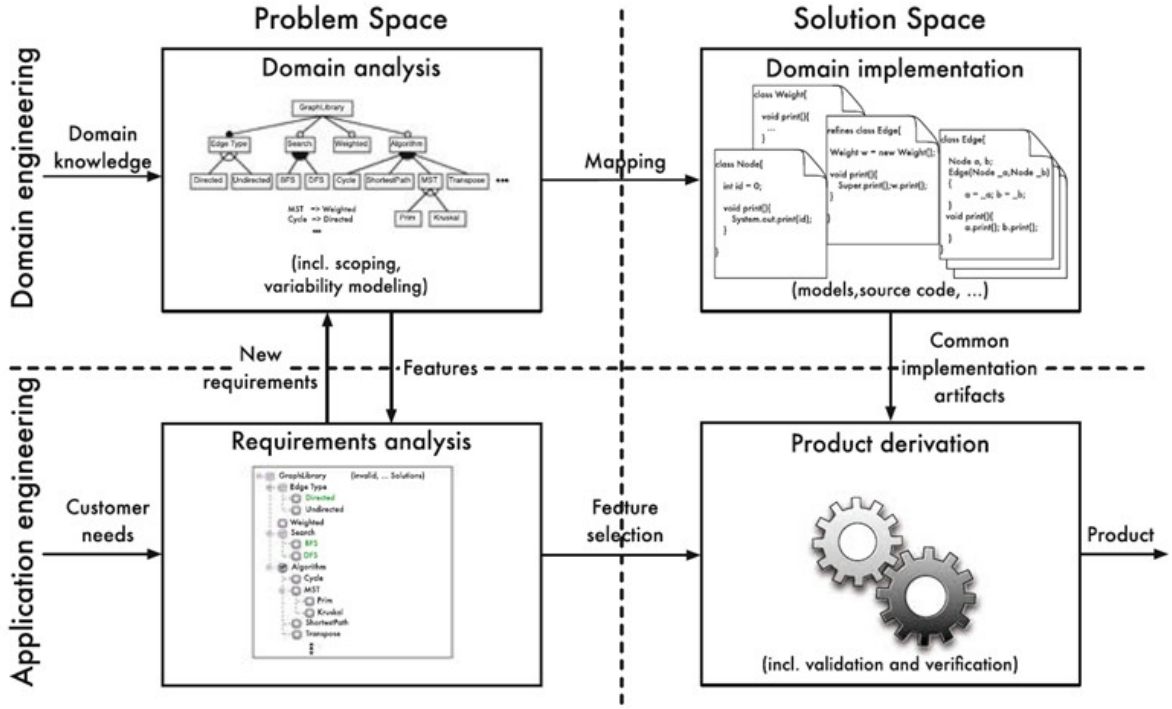

Fig. 2.8 A feature-oriented engineering process for software product lines [Apel et al., 2013].

The proposed framework distinguishes between *Domain Engineering (DE)* and *Application Engineering (AE)* and between *Problem Space (PS)* and *Solution Space (SS)*. Figure 2.8 illustrates a two-dimensional structure with four clusters of tasks in product-line development and mapping between them.

The top half of Figure 2.8 is depicted for the *domain engineering*, which is the process of analyzing the domain of product line and developing reusable artifacts. Domain engineering does not result in a specific software product, but prepares artifacts to be used in multiple, if not all, products of a product-line. Domain engineering targets *development for reuse*. The bottom half of Figure 2.8 depicts the *application engineering,* which has the goal of developing a specific product for the needs of a particular customer (or stakeholder). It corresponds with the process of single application development in traditional software engineering, but reuses artifacts from domain engineering where possible. It targets *development with reuse*. Application engineering is repeated for every product of the product line that is to be derived.

The distinction between the problem space and solution space highlights two different perspectives. The left half of Figure 2.8 depicts the *problem space*, which takes the perspective of stakeholders and their *problems*, *requirements*, and *views* of the entire *domain* and *individual products*. Features are, in fact, domain abstractions that characterize the problem space. The right half of Figure 2.8 depicts the *solution space*. This space represents the developer's and vendor's perspectives. It is characterized by the terminology of the developer, which includes names of functions, classes, and program parameters. The solution space covers the *design*, *implementation*, and *validation and verification* of features and their combinations in suitable ways to facilitate systematic reuse.

2.7.2 4 clusters of tasks in product-line development

Apel et al. [Apel et al., 2013] have adopted an orthogonal distinction between domain and application engineering as well as the problem and solution space. This distinction gives rise to four clusters of tasks in product-line development:

1. ***Domain analysis*** is a form of requirements engineering for an entire product line. In this cluster, the scope of the domain is to be determined. Therewith deciding which products should be covered by the product line and, consequently, which features are relevant and should be implemented as reusable artifacts. The results of domain analysis are usually documented in a *feature model*;

2. ***Requirements analysis*** investigates the needs of a specific customer as part of application engineering. In the simplest case, a customer's requirements are mapped to a feature selection, based on the features identified during domain analysis. If new requirements are discovered, they can be fed back into domain analysis, which may result in a modification of the feature model (and the reusable domain artifacts);

3. ***Domain implementation*** is the process of developing reusable artifacts that correspond to the features identified in domain analysis. There are many kinds of artifacts relevant in software product lines such as design, test, and documentation artifacts, and implementation artifacts (i.e. source code). Moreover, the basic ideas and techniques apply also to non-code artifacts. Depending on how variability is implemented, developers might produce very different artifacts in this step, from run-time parameters and preprocessor directives to plug-ins and components, and many more;

4. ***Product derivation*** is the production step of application engineering, where reusable artifacts are combined according to the results of requirement analysis. Depending on the implementation approach, this process can be more or less automated, possibly, involving several development and customization tasks.

According to [Apel at al., 2013], a goal of product-line development, is to move development efforts from application engineering to domain engineering as much as possible. Further, the more application engineering is evolved into a series of generation tasks, the lower the costs per product will be. A major goal of feature-oriented product lines is to fully automate product derivation.

2.8 Factory-oriented approach for Product Line Engineering (PLE)

BigLevers and others [Clements & Northrop, 2001] [Clements, and BigLever, 2015] [Krueger, and Clements, 2017] [Young, and Clements, 2017] [Bolander, et al., 2016] have proposed a definition for what they call "Product Line Engineering". The consider that *"Systems and Software Product Line Engineering, abbreviated as Product Line Engineering (or PLE for short), is defined as the engineering of a portfolio of related products using a shared set of engineering assets and an efficient means of production"* [Productlineengineering.com, 2016].

In this definition, there are three remarkable concepts, namely "Products", "Assets", and "Means of production". They have defined these concepts as following:

- The *products* in a Product Line Engineering (PLE) portfolio are described by the properties they have in common and the variations that set them apart. The descriptions are in terms of the products' features. Products can comprise any combination of *Software*, *Systems* in which software runs, or *Non-software systems* that have software-representable artifacts (such as engineering models or development plans) associated with them;
- *Assets* are the "soft" artifacts associated with engineering lifecycle of the products; the building blocks of the products in the product line. Assets can be whatever artifacts that are representable with software or either artifacts that compose a product or support the engineering process to create a product. These can include, but are not limited to *"Requirements*, *Design specifications*, *Design models*, *Source code*, *Build files*, *Test plans and test cases*, *User documentation*, *Repair manuals and installation guides*, *Project budgets*, *Schedules and work plans*, *Product calibration and configuration files*, *Data models and parts lists*, etc.";
- The *means of production* is the mechanism that exercises the assets' *variation points (VP)* to produce configured versions that, together, constitute the artifact set for one of the products in the product line. Configuring the shared assets for each product in turn produces the entire set of products.

According to [Young, and Clements, 2017] and [Bolander, et al., 2016], early approaches to Product Line Engineering (PLE) (e.g. Program Families [Parnas, 1976], FODA

[Kang et al., 1990], SATRS [Foreman, 1996], FAST [Weiss and Lai, 1999]) employed many means for producing the products from shared assets. The assets could be manually configured, or (more likely) an *ad hoc* collection of techniques was employed to configure the various assets separately. These approaches have yielded a rich legacy of successful product-line solutions [Clements et al., 2014] [van der Linden et al, 2007] [Northrop et al., 2012]. However, according to [Northrop et al., 2012], no approach in these early years reached the level of becoming a repeatable, prescriptive, methodological engineering discipline They considered the scheme scales poorly, and may not trace well across different kinds of artifacts.

The *Second Generation Product Line Engineering (2GPLE)* approaches use the same kind of variation across all assets. Second Generation PLE is centered on the creation of a PLE Factory [Clements et al., 2014] [Krueger, and Clements, 2014]. The advent of the 2GPLE – which is an alignment of PLE approaches that are centered on the factory approach – has resolved many of the weaknesses of the early approaches [Clements and BigLever, 2015].

2.8.1 The Second Generation Product Line Engineering (2GPLE) approaches

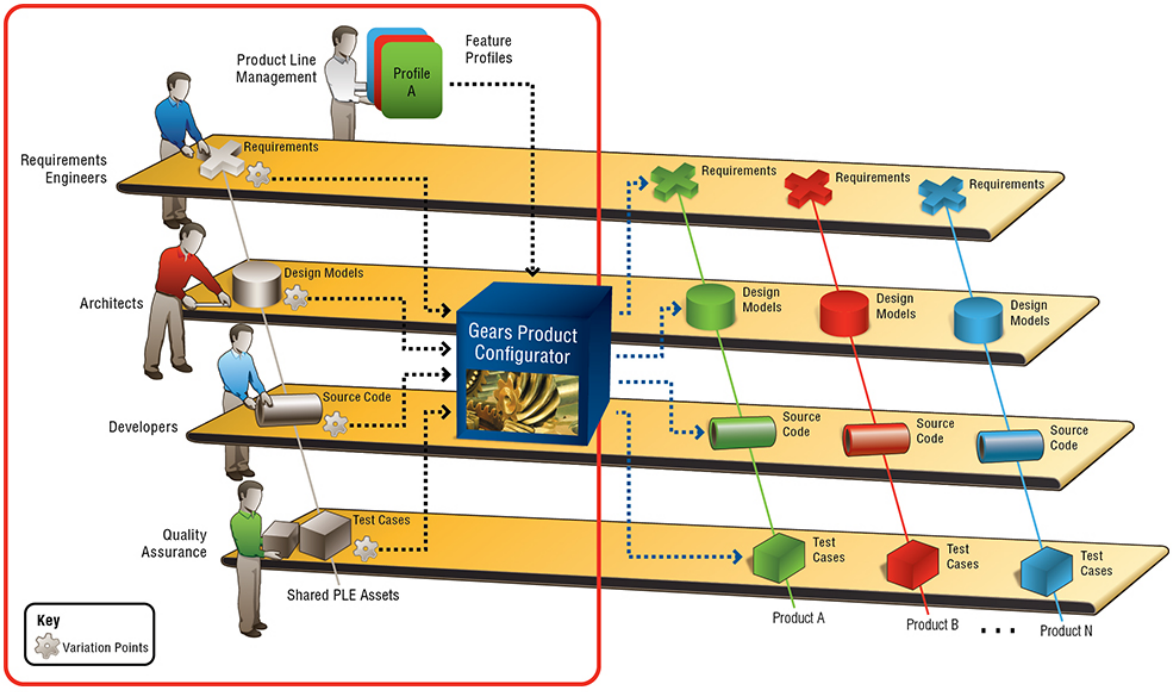

Fig. 2.9 The Second Generation Product Line Engineering (2GPLE) factory paradigm [Clements et al., 2014].

The second generation PLE (2GPLE) was built on the first-generation efforts of PLE. 2GPLE has embodied a more well-defined and repeatable process, centered on a strong *factory paradigm* [Clements et al., 2014]. As the modern product lines are more and more complex, the intricacies of these product lines require industrial-strength automation. The 2GPLE approaches provide the required commercial industrial-strength automation tools. The automation is called *"Configurator"*, which takes a feature-based description of a product and configures all of the assets (using their variation points) to produce instances for that product [Bolander, et al., 2016].

Figure 2.9 exhibits the essence of 2GPLE with *Gears*. These gears are considered as configurators. At the left of Figure 2.9, the factory's supply chain is illustrated in the form of shared configurable assets. The assets are configurable because they include *variation points (VP)* (i.e. places where the artifact needs to be different depending on what feature set it is asked to support), shown by the gear symbols in the figure. In fact, variation points are expressed in terms of the available *features* in each products. A *feature-based* product specification, as seen at the top of Figure 2.9, instructs the configurator on how to configure the assets coming in from the left. The resulting products, assembled from the configured assets, emerge on the right part of the figure. This enables the rapid production of several variants of the assets for any of the products in the portfolio. Once the production line capability is established, products are instantiated (i.e. derived from the shared assets by the configurator) rather than manually created.

The second-generation product line engineering (2GPLE) has several distinguished characteristics. Among these characteristics, the following important characteristics distinguish the approach:

- **Features express product variation:** In the *factory paradigm*, a production protocol is required, so the shared assets (requirements, designs, code, test cases, user manuals, etc.) can be configured appropriately. Rather than adopting a different *"language"* and mechanism for each type of artifact (e.g. compiler directives for code, attributes for requirements, text variables for documents, etc.), 2GPLE uses a small and consistent set of variation mechanisms [Bachmann and Clements, 2005] for all of the artifacts. Each product is described by giving a list of its features. Here, features are used to express product differences in all lifecycle phase artifacts. This streamlines the development process and lets all stakeholders speak the same language.

- **Shared assets come from all lifecycle phases, not just the software:** In *large-scale* product lines, automated production of complete and consistent sets of lifecycle artifacts is essential. Managing these artifacts means imbuing them with variation points [Bachmann and Clements, 2005]. In 2GPLE, assets are designed with built-in variation points, which are places within the asset that can change depending on the product in which they are used. When a product is built, the configurator uses the product's feature-based description to "exercise" these variation points (that is, cause the change in the asset to occur as to meet the product needs) [Clements et al., 2014]. Variation point mechanisms comprise [Clements, and BigLever, 2015]:
    - Including or omitted the artifact;
    - Choosing one variant of the artifact (from an available set) to use in the product;
    - Making fine-grained choices within an artifact, such as including or omitting a section, model element, or block of code.

- **Industrial-strength automation** is employed in the form of a *configurator*, which is a tool that takes a feature-based description of a product and exercises the variation points in the shared assets to produce an artifact set that supports the named features. Product

development thereby becomes automated, so that application engineering (so important in first-generation approaches) becomes negligible [Clements et al., 2014].

- **Configuration management (CM) that maintains assets, not products or asset instantiations:** Under the 2GPLE discipline, the full superset of available PLE assets (and not the individual products or systems) are managed under configuration management (CM). A new version of a product is not derived from a previous version of the same product, but rather from the shared superset of PLE assets themselves.

- **Supporting product lines across organizational boundaries:** This characteristic of 2GPLE involves feature languages that facilitate modular and hierarchical product lines developed across organizational boundaries [Flores et al., 2012]. This allows a *system-of-systems* to become *product-line-of-product-lines* [Clements, and BigLever, 2015].

From several real world case study, such as the cases of General Motors [Flores et al., 2012] and the US Army [Rivera et al., 2008], 2GPLE has proved its benefits and has represented a more clearly formulated methodology that organizations can use directly. It simultaneously generalizes and simplifies the concepts and deployments ways from its first-generation roots [Clements et al., 2014]. Employing 2GPLE has shown substantial benefits in reliability, sustainability, and responsiveness in organizations [Lanman et al., 2009] [Clements et al., 2014].

2.8.1.1 PLE as a factory

An analogy with *factory-based manufacturing* serves well to properly illuminate the important concepts. Manufacturers have long used engineering techniques to create a product line of similar products using a common factory, that assembles and configures parts to produce the varying products in the line. According to the principles of the 2GPLE, the *"configurator"* is the factory's automation component. The *"parts"* are the assets in the factory's supply chain. A statement of the properties desired in the product tells the configurator how to configure the assets.

Figure 2.10 illustrates how a product line could be structured as a *"factory"*. At the left of Figure 2.10, the factory's supply chain is depicted. The presented supply chain is expressed in the form of shared assets that are configurable, as they include variation points that are expressed in terms of the features available in each of the products. A product specification at the top of figure 2.10 tells the configurator how to configure the assets coming in from the left. The resulting products, assembled from the configured assets, emerge on the right of Figure 2.10. This enables the rapid production of several variant of any assets for any of the products in the portfolio. Once this production line capability is established, products are instantiated – derived from the shared assets – rather than manually created.

In this context, the term *product* means not only the primary entity being built and delivered, but also all of the *artifacts* that are produced along with it. Some of these support the engineering process (e.g. *Requirements*, *Project Plans*, *Design Modes*, and *Test*

Cases), while others are necessary byproducts, such as *User Manuals*, *Shipping Labels*, and *Parts Lists*. These artifacts are the product-line's assets.

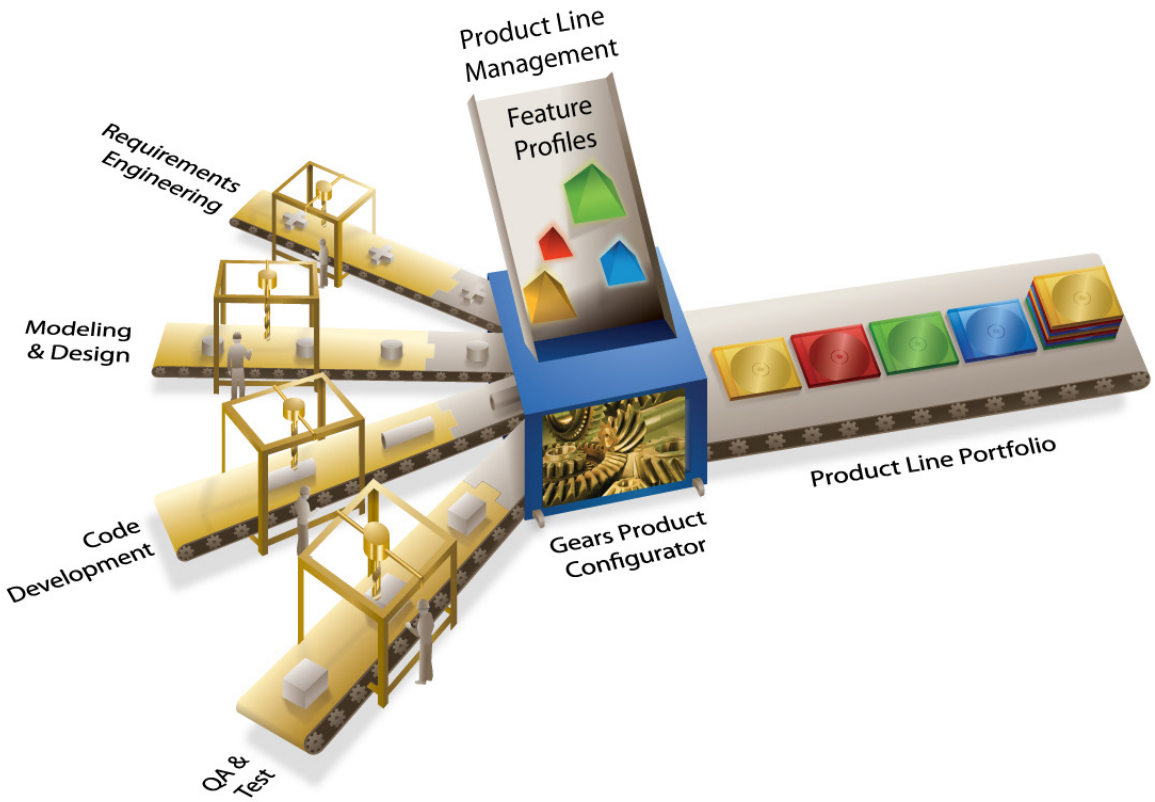

Fig. 2.10 The Product Line Engineering envisioned as a factory [Clements et al., 2014].

Assets can be whatever artifacts are representable digitally and either constitute part of a product or support the engineering process to create a product. Four kinds of *shared assets* are shown in Figure 2.10, but those are just examples. Shared assets can include, and are not limited to, *Requirements*, *Design Specifications*, *Design Models*, *Source Code*, *Build Files*, *Test Plans and Test Cases*, *User Documentation*, *Repair Manuals and Installation Guides*, *Project Budgets*, *Schedules*, and *Work Plans*, *Product Calibration and Configuration Files*, *Data Models*, *Parts Lists*, and more. Assets in PLE are engineered to be shared throughout the product-line.

2.8.1.2 PLE contrasted with product-centric development

According to BigLever's white-paper [Productlineengineering.com, 2016], *"PLE contrasts highly with traditional product-centric development; in which each individual product is developed and evolves independently from other products, or (at best) starts out as a cloned copy of a similar product that is then changed to suit the new product's specific needs. This product-centric development takes very little advantage of the commonalities among products in a portfolio after the initial clone operation. In particular, it derives very little benefit from*

commonality in a product's sustainment or maintenance phase, where, data shows, most products consume up to 90% of their project's resources" [Clements, and BigLever, 2015].

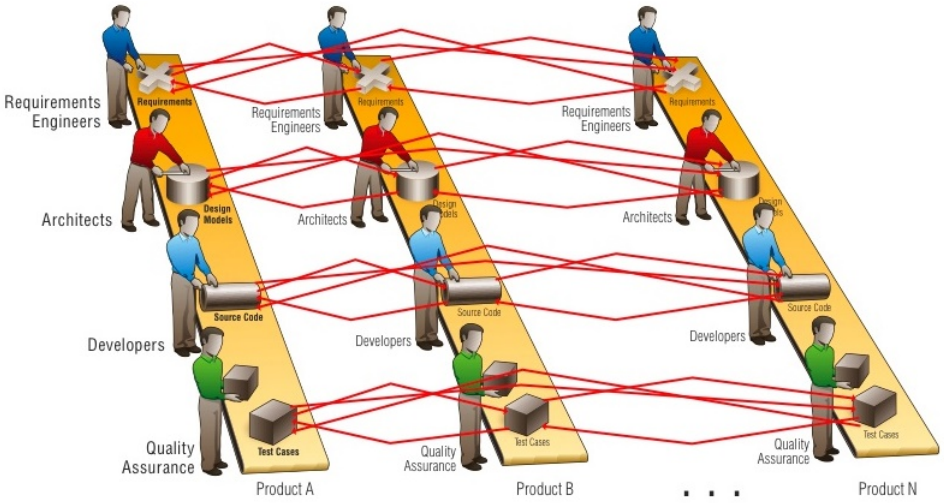

Fig. 2.11 Product-centric development yields $O(N^2)$ complexity [Bolander, et al., 2016].

Figure 2.11 shows a production shop in which *N products* are developed and maintained. In this stylized view, each product encompasses requirements, design models, source code, and test cases. Each engineer in this shop primarily works on a single product. When a new product is launched, the most similar assets are copied, and the project then starts by adapting them to meet the new product's needs.

Coordination among projects, if present, is *ad hoc* and de-centralized, meaning that, each of the *N* product teams should really confer with each of the other *N-1* product teams. These communication paths are shown in *red* in Figure 2.11. This communication obligation imposes an overhead that grows as the square of the number of products. This complexity will quickly overwhelm any engineering staff; in order to get their products out the door on time and on budget, each product team will focus more on their product silo and less on taking advantage of the commonalities and interdependencies among the other products. The result is divergent product silos, low degrees of sharing, and high duplication of effort across the product silos to fix the same defect multiple times in multiple products, or to independently implement the same enhancements in different ways in different products [Clements, and BigLever, 2015].

Under the PLE approach, all development takes place in the factory and not in project silos. This assures the maximum amount of *cross-project* sharing on an ongoing basis. Coordination happens between products and the factory, which for a *portfolio* of *N* products is an $O(N)$, as opposed to an $O(N^2)$, proposition [Clements, and BigLever, 2015].

2.8.2 Ecosystem support for three dimensions of PLE

By definition an *ecosystem* is considered as *"a system, or a group of interconnected elements, formed by the interaction of a community of organisms with their environment"* or, perhaps

more helpfully to the context of the thesis, *"any system or network of interconnecting and interacting parts, as in a business"* [John et al., 2012]. John et al. have pointed out that ecosystems are believed to contain the necessary elements to sustain life of the ecosystem's elements. Therefore, if the elements have value to us, then the ecosystem also has value to us, and is therefore worth studying to understand how it sustains that life and) what can be done to enhance that sustainment [Bolander, et al., 2016]. In the context of PLE, the term *ecosystem* in reference to *Tools*, *Technologies*, *Products*, and *Suppliers* of all of those that, together, provide an *industrial-strength PLE technology* solution [Productlineengineering.com, 2016].

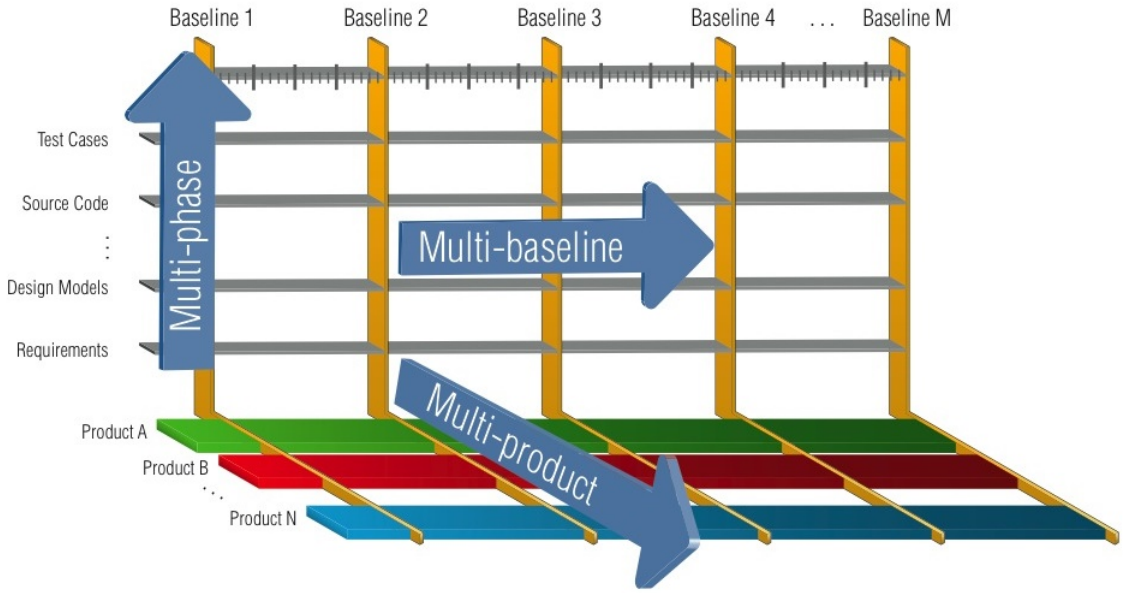

Fig. 2.12 Three dimensions of Product Line Engineering [Bolander, et al., 2016].

Figure 2.12 illustrates these three concerns. Organizations have to, *first*, manage the popularity of products (i.e. illustrated by the outward-pointing axis of Figure 2.12). The 2GPLE paradigm, discussed above, and the automation-centric PLE approach for which 2GPLE is a specific instance, is aimed largely at this dimension. This is the realm of mainstream PLE tools, such as one highlighted earlier [Bolander, et al., 2016]. *Second*, they have to evolve the portfolio over time (i.e. illustrated by the horizontal axis of Figure 2.12). The horizontal axis of Figure 2.12 represents the 'multi-baseline', which deals with the usual temporal concerns of product engineering, such as version, configuration and change management. Configuration management (CM) systems track the evolution of shared assets over time. Just as no PLE tool is going to incorporate all requirements engineering or testing tools, no PLE tool is going incorporate all CM system capabilities. At best, they will be compatible with general CM systems, or even be agnostic to the CM systems in use by simply working on files checked out into workspaces. However, to handle the critical need to manage evolution, CM systems are now added to the PLE ecosystem [Clements, and BigLever, 2015]. *Third*, they have to manage the shared assets and the products to which they contribute across lifecycle phases or disciplines (i.e. vertical axis of Figure 2.12). The ecosystem, discussed in the previous section is aimed

largely at this dimension. However, in a true systems engineering environment, traceability across the lifecycle artifacts (e.g., from requirements to code to tests, or from design models to parts list) plays a critical role. That traceability must be maintained as the product-line and its shared assets evolve over time, and must persist when shared assets' *variation points* are exercised to derive products [Bolander, et al., 2016].

2.8.3 Establishing a PLE Factory approach

Organizations may move to establish a *PLE Factory* for competitive advantage or to increase their bottom-line. According to [Productlineengineering.com, 2016] and [Young, and Clements, 2017], the successful organizational adoption of a PLE Factory approach often involves three stages or tiers.

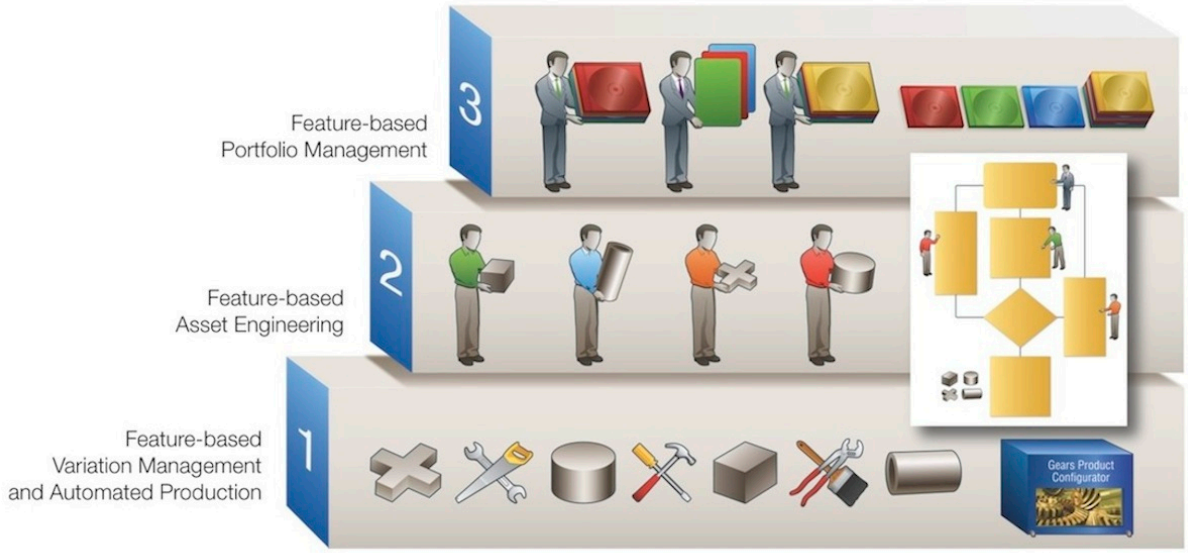

Fig. 2.13 Three tiered approach for adopting a PLE Factory [Productlineengineering.com, 2016].

Figure 2.13 illustrates the adoption approach for establishing product lines following the principles of 2GPLE. The tiers of the proposed approach are the following:

- **Tier 1:** this tier focuses on incorporating the *PLE Factory configurator* into the organization, and using it to define *feature models* and a shared *asset* superset for the product line;

- **Tier 2:** this tier starts by *re-engineering* the products' assets into a collection of shared assets with *variation points*. In this tier, new roles specific to PLE are defined and filled; roles that move the engineers away from *product-specific* responsibilities and towards *asset-specific*, *product-independent* roles;

- **Tier 3:** this tier concerns the organization's management and allows it to steer the portfolio in strategic directions by defining products with new features or new feature combinations to, for example, enter a new market where the organization's ability to produce new products quickly and efficiently will provide competitive advantage.

Organizations can begin building capabilities in each tier together. Further, adoption can be *incremental*, and need not happen all at once. Under a principle known as *"incremental return on incremental investment"* each step towards complete adoption brings commensurate benefit [Productlineengineering.com, 2016] [Clements, and BigLever, 2015].

2.9 Adoption strategies of a Product-Line Approach

The adoption of a Software Product Line (PLE) approach within an organization represents a big concern for the PLE community. Many adoption strategies have been presented in literature. According to Krueger [Krueger, 2002], there are three different adoption paths following:

i. Proactive approach;
ii. Extractive approach;
iii. Reactive approach.

The adoption strategy may have a significant effect on the selection of implementation methods.

2.9.1 Proactive approach

The *proactive approach* develops a product-line from scratch by carefully using analysis and design methods. Following this approach, in a *design process*, the developers model the domain and implement all relevant features before the first product is generated. Proactive approach gathers key tasks, which are the following:

- Domain analysis and scoping;
- Determination of the product-line implementation approach;
- Implementing the entire product-line.

Using the proactive approach, developers can plan the product-line's variability perfectly for the desired variability. Consequently, one can reach a high-level of code quality and maintainability. However, the drawback of this approach is a *high upfront investment* and the corresponding *risks* before the first product arrives on the market. Moreover, with existing products, a company has to essentially stop production for a substantial period-of-time to restructure or even rewriting the code.

Several success stories from companies that adopted product-line with a proactive approach were presented in literature. For example, Clements and Kruger [Clements and Kruger, 2002] have presented a successful case with a full production stop. However, it is debatable how applicable this process is in general. Often, some products are already in productive use and a long delay to transit to product-line technology is not acceptable. The

proactive approach is often seen as idealistic and academic, which, in practice, has to be combined partly with ideas from the other two adoption strategies (i.e. Extractive and Reactive approaches) [Apel el al., 2013].

2.9.2 Extractive approach

The *extractive approach* starts with a collection of existing products and incrementally refactors them to form a product-line. This approach is useful when a company already has a portfolio of related products that target a common domain, but those projects are not engineered in a systematic way yet. The main target of the extractive approach is to make a *transition* from one or multiple legacy products to a more structured product line. The main tasks of the extractive approach are the following:

- Identification of commonalities and differences of existing products, based on domain knowledge and stakeholder requirements;
- Extraction or implementation of the core functionality in the form of common reusable domain artifacts;
- Extraction and realization of the variation using appropriate implementation techniques.

According to Apel et al. [Apel el al., 2013], the extractive approach advocates an incremental adoption of product-line technology. Common parts are extracted, and some cloning is eliminated step by step. Due to its incremental nature, risks and upfront investment are much lower compared to the proactive approach. Note that the quality of the extracted product line relies on the quality of the tools supporting the extraction.

2.9.3 Reactive approach

The *reactive approach* begins with a small, easy to handle product line (possibly consisting only of a single product) and is extended *incrementally* with new features and implementation artifacts, thus extending the scope of the product-line.

This approach was presented as an instance of Boehm's "Spiral Model" [Boehm, 1988]. In fact, this approach is considered as an agile method to adopt a product-line approach. According to this approach, developers start with a software product line SPL_0, which realizes an initial version of the envisioned software product line. In incremental steps from SPL_i to SPL_{i+1}, the product line progressively grows toward its ideal version; covering the full variation spectrum, as defined during *Domain Analysis*, which can also be incremental. The main tasks in the reactive approach are the following [Apel et al., 2013]:

- Exploration and characterization of the requirements leading to a new product currently not covered by the product line;
- Describing the delta leading to the improved product;
- Implementing the delta in a suitable way.

In addition to be an adoption path, the reactive approach also describes a typical pattern for maintaining and evolving a product line during its lifetime. *Reactive strategy* is positioned between the proactive and the extractive approach. It requires less *upfront planning* than the proactive approach, but may require more invasive and expensive changes to the product-line. At the same time, the reactive approach is typically considered to be more structured than the extractive approach, because each iteration follows clear planning steps. Overall, the *reactive process* aligns well with *agile methods* of software construction [Apel el al., 2013].

2.10 Conclusion

Software Product Lines approaches are well presented in literature. Researchers and practitioners have proposed a number of approaches, techniques, tools, and practices tailored specifically to software product lines. This chapter has revisited what are considered as core approaches of Software Product Lines. Reviewing these approaches and their related principles and conceptual foundations has fortified the required knowledge that was essential to attempt the main target of this thesis.

Software Product Line approaches aim at the development of similar software products in an efficient and coordinated manner. They allow maximizing the reuse of commonality and of variability. The reuse of commonality is maximized by developing all products on a common product platform. The variability is fully exploited by using a modular development of variable functionality, which can be added to or removed from the product more easily.

Each approach has its advantages and disadvantages. Therefore, the adoption of an approach depends on the business model of the software vendor and its context, the targeted domain, stakeholder's requirements, and other issues. This chapter has introduced different definitions of the Software Product Line Paradigm, its promises, the different product line architecture, some famous Product Line Engineering approaches, and the different adoption strategies.

3 Agile software development

Abstract. *Agile software development* refers to a group of software development methods, in which requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. It promotes adaptive planning, evolutionary development, early delivery, continuous improvement, and further encourages rapid and flexible response to change. Its adherents promote the fast delivery of higher quality systems, which match customer needs and expectations much better. This chapter considers *"Agile"* from a management perspective by introducing a considerable overview of agile software development and by focusing on implementation, organization and concerned people of two main methods, namely, *Scrum* and *Scrumban*.

3.1 The rise of Agile Methodologies

The concept of *Agile* emerged in the late 1990s, in the effort to address perceived difficulties with existing solution development processes that were rooted in, and owed their rigidity to *Plan-Driven* practices [Moran, 2015]. The roots of *Agile* date back to the 1980s, prompted by the rise of new technology and the increasing volatility of the business environment, the shortcomings of traditional methods were becoming more evident [Cockburn, 2005].

Agile takes its origins in part from the Japanese manufacturing and industrial sectors to which many of its concepts owe their heritage [Boehm and Turner, 2003]. These include the visual control concept found in the Toyota Production System [Monden, 1993], that later anticipated agile information radiators, the *Kanban* [Anderson, 2010] charts, used in agile task assignment and tracking, and the continual influence of lean thinking on *Agile* today. The synthesis of *Eastern* and *Western* thinking so persuasively laid out by the authors that introduced the term *"scrum"* [Cohn, 2013], reflects the spirit in which the *Agile Manifesto* [ManifestoAgile, 2001] itself was conceived and points to a genesis founded in organizational learning and team empowerment [Takeuchi et al., 1995].

Figure 3.1 illustrates the timeline of the main agile methods since the early 1990s. The first agile development method is called Dynamic Systems Development Method (DSDM) [Stapleton, 1997]. The DSDM has emerged from Rapid Application Development (RAD)

methodology [Martin, 1991], which is an independent framework that has evolved over time to encompass a wider scope than one would traditionally associate with agile projects (e.g., inclusion of explicit governance, quality and risk). By 2007, DSDM had become an open methodology that had assumed the mantle of a generic agile project management and solution delivery framework. At that moment, DSDM was briefly introduced under the marketing name "Atern". Today, the DSDM culminates in agile project management an Agile Project Framework (AgilePF) [DSDM Consortium, 2014], the DSDM Agile Project Management (AgilePM) [Agile Business Consortium, 2014], and the DSDM Program Management Frameworks (AgilePgM) [Messeger et al., 2014].

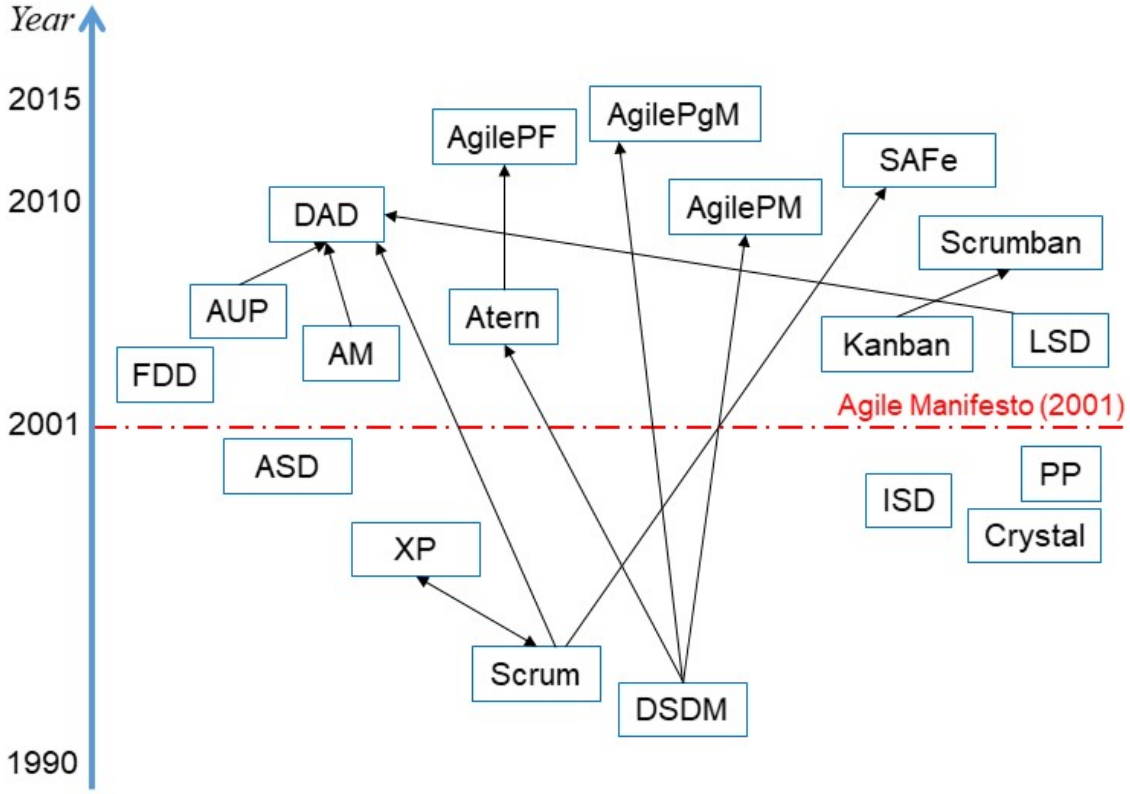

Fig. 3.1 Historical Development of Agile Methods. Adapted from [Abrahamsson et al., 2003] and [Moran, 2015].

In 2003, RAD gave rise indirectly to Adaptive Software Development (ASD), which actively sought to embrace change in speculate-collaborate-learn cyclical patterns of work [Highsmith, 2013]. The method *Scrum* [Cohn, 2013] [Cohn, 2004] was conceived as a manufacturing development methodology that brought about "*innovation continuously*, *incrementally*, and *spirally*". The roots of Scrum can be traced back to its industrial heritage in 1986. Moreover, Scrum took shape from ideas stemming from organizational learning and went on to exert considerable influence within the agile world and, thus, affecting almost all other methodologies [Takeuchi and Nonaka, 1995].

The 'eXtreme Programming' (XP)method [Auer and Miller, 2001] is a software engineering method focused on a set of principles and practices that grew of the development

of the Chrysler Comprehensive Compensation System payroll system. According to [Beck and Andres, 2004], XP has enjoyed a period of *"cross-fertilization"* with Scrum, setting the terms of reference for many agile practices today. In the 1990s, XP became hugely influential and nowadays, its practices constitute the essential tool set of every IT project that employs an agile approach.

In February 2001, the "Agile Manifesto" [ManifestoAgile, 2001] was born. Seventeen software developers formulated it in order to establish a common ground for their perceptions of the software development process. Around the time of the formulation of the Agile Manifesto, several other methodologies were to be found, such as:

- The design and build focused Feature Driven Development (FDD) [Palmer and Felsing, 2002];
- The Internet-speed Development (ISD) [Baskerville et al., 2001], which practiced fast release and delivery;
- The practical toolkit that constituted Pragmatic Programming (PP) [Hunt and Thomas, 2000];
- The flexible Crystal family of methodologies [Cockburn, 2006].

Furthermore, many efforts were deployed in order to make the Rational Unified Process (RUP) [Gibbs, 2009] *Agile*. These efforts are manifested in the Agile Unified Process (AUP) [Ambler, 2006a], which together with Agile Modelling (AM) [Ambler, 2002], focused on modeling practices and cultural principles, and, therefore, became the predecessors for Disciplined Agile Delivery (DAD) [Ambler and Lines, 2012]. Beside the mentioned developments, other influential approaches were introduced, including Lean Software Development (LSD) [Poppendieck and Poppendieck, 2003], which applied lean principles to *Agile* and the attempt to architecturally scale agile product development using the Scaled Agile Framework (SAFe) [Leffingwell, 2018].

Back in 2004, David Anderson designed a pull system, which later evolved into the Kanban method [Anderson, 2010]. This pull system was designed for Microsoft IT teams. Just a few months later, the team achieved its highest productivity per person, shortest lead-time and highest customer satisfaction. Moreover, by the introduction of *Scrumban* [Ladas, 2009], organizations have layered the *Kanban* Method alongside *Scrum* to help them achieve several different kinds of outcomes.

Reviewing the historical development of Agile reveals a rich texture of humanist, organizational, and technological traditions, for which a mature body of literature concerning both its culture and its practices already exists [Cohn, 2005] [Moran, 2015]. In addition, numerous comparative surveys of agile methodologies have been found in literature, where these studies highlight both the commonality rooted in the manifesto and its principles together with the uniqueness of focus and purpose of each approach [Moran, 2014].

3.2 The "Agility" attribute

Supporters of *agile* have promoted the notion that project *uncertainties* should be taken into account in order to balance planning and control with *execution* and *feedback* [Moran, 2014]. Agile projects exhibit features of *"open communication amongst heterogeneous stakeholders"*, *"emergent behavior within self-organizing teams"* and a *"culture of openness and learning"* [Book et al., 2016].

Agile Software Development (ASD) has two central notions, namely *"iterative"* and *"incremental"*. The notions of *iterative development* and *incremental delivery* are based on shared values stipulated in the *"agile manifesto"* [ManifestoAgile, 2001] which expresses preferences towards *'individual interactions and customer collaboration'*, *'working solutions over comprehensive documentation'*, and *'responsiveness to change'*. Thus, the agile manifesto articulates the convictions that interactions among *"project team members"* and their *"customers"* (i.e. *product owner*) should support efforts to create *"working solutions"* in a flexible manner [Cockburn, 2005].

According to [Moran, 2014], being agile starts by defining what *"agile"* means for the organization. In fact, defining *agile* is harder than it seems, as much of what practitioners understand to be the essence of agile arises from principles and practices experienced as emergent characteristics. Thus, it could be hard to attribute *agility* to specific individual techniques or rituals, as it is widely accepted that many of these techniques existed prior to the agile community. Nor can agile be characterized purely by reference to the agile manifesto and its principles alone, as some principles are likely to prove troublesome if used as the basis of a definition.

Consequently, *agile* can be understood, as a structured solution development paradigm that embodies the following core elements, surrounded by a regular iterative development and incremental delivery driven by business needs [Cohn, 2005] [Moran, 2015]:

- *Adaptive:* it is widely admitted that change is inevitable and that the pursuit of reward entails risk. Agile promises adaptive planning and effective feedback loops. On the one hand, with agile, the high-level plans are revised later into detailed plans once the necessary information becomes available. On the other hand, reviews and retrospectives guide and direct the solution development and the process behind it. Therefore, dealing with the inevitable change requires a decentralized and iterative approach to solution development, which is responsive to the changing needs of business;

- *Value-Driven:* in order to focus on individual business needs and requirements, agile advocates direct assessments of progress (e.g., working solutions rather than status reports) and draws on the direct experience and input of those who need the solution. Thus, to meet this target, this requires tighter integration of stakeholders, leaner production practices and incremental delivery of working solutions;

- *Collaborative:* Agile avoids specialists working to specifications, preferring instead to employ multi-disciplinary and highly communicative teams. Teams that share their experiences and tacit knowledge in order to gain consensus regarding the solution, which may entail engagement of stakeholders outside of the team;

- *Empowered:* adopting an agile approach needs integrated teams. Establishing integrated teams requires humanistic values of trust, respect and courage supported by an environment of empowerment and self-organization wherein the traditional role of management is replaced by one of servant-leadership.

All these core elements are found to work well in practice. Therefore, *“Agile”* is as much of a cultural stance on the process of solution development, as well as it is a set of *practices* and *values*. Furthermore, Schwaber [Schwaber, 2006], one of leading supporters of agile, has described agility as *“hard and disruptive”*. However, there are several ideas that regarded ambiguity during the development process as a strength, provoking new perspectives and challenging established ideas, rather than a weakness that must be managed through precise planning [Moran, 2015].

3.2.1 Adopting a definition for the term “Agile”

Nowadays, more and more people use the term ***“Agile”*** when talking about the project management field, mainly in software development. The word “agile” is one of those trending words that is almost used without understanding the true essence of it. What does the term “agile” actually mean? Several definitions of “agile” are presented in literature. The intention of this section is to adopt a definition of the “agile” term that fit the context of the present research in order to use it in this thesis and to use it in developing a definition of the “APLE” concept.

The term “agile” is defined in Oxford Dictionary of English [Oxford University Press, n.d.] as *“Able to move quickly and easily”* or *“Relating to or denoting a method of project management, used especially for software development, that is characterized by the division of tasks into short phases of work and frequent reassessment and adaptation of plans”*. Goldman has defined “agility” as “*A comprehensive response to the business challenges of profiting from rapidly changing, continually fragmenting, global markets for high-quality, high-performance, customer-configured goods and services. It is dynamic, context-specific, aggressively change-embracing, and growth-oriented. It is not about improving efficiency, cutting costs, or battening down the business hatches to ride out fearsome competitive “storms”, it is about succeeding and about winning: about succeeding in emerging competitive arenas, and about winning profits, market share, and customers in the very center of the competitive storms many companies now fear.”* [Goldman et al., 1995]. In addition, Table 3.1 gathers some definitions of *“Agile”* found in literature.

Table 3.1 Definitions of the term "Agile". Adapted from [Laanti et al., 2013].

	References	Definitions	Emphasis of the corresponding definition
1	[Cockburn, 2002]	Being effective and maneuverable. Use of light-but-sufficient rules of project behavior and the use of human and communication-oriented rules.	Effective, Steerable, Rule-based, People, Communication
2	[Anderson, 2003]	Ability to expedite	Speed
3	[Larman, 2003]	Rapid and flexible response to change.	Speed, flexibility, responsiveness
4	[Schuh, 2004]	Building software by empowering and trusting people. Acknowledging change as a norm, and promoting constant feedback. Producing more valuable functionality faster.	People, empowerment, change, feedback, value, speed
5	[Lyytinen, 2006]	Discovery and adoption of multiple types of Information Systems Development innovations through garnering and utilizing agile sensing and response capabilities.	Delivery, innovations, responsiveness
6	[Subramaniam, 2005]	Uses feedback to make constant adjustments in a highly collaborative environment.	Feedback, adaptability, collaboration
7	[Ambler, 2007]	Iterative and incremental (evolutionary) approach to software development, which is performed in a highly collaborative manner by self-organizing teams with “just enough” ceremony that produces high-quality software in a cost-effective and timely manner which meets the changing needs of its stakeholders.	Iterative, incremental, self-organizing, less process-driven, collaborative, cost-conscious, speed, customer-driven
8	[Nerur and Balijepally, 2007]	Define “Agile” via strategic thinking (of uncertainty), holographic organization theory, “emergent metaphor of design” and Agile Methods as people-centric, competent people and their relationships, high customer satisfaction through quick delivery of quality software, active participation of concerned stakeholders; creating and leveraging change. Evolutionary delivery through short iterative cycles, intense collaboration, self-organizing teams and high degree of developer discretion. Learning, teamwork, self-organization and personal empowerment. Responsiveness and flexibility. Interchangeability of roles and jobs based on autonomy.	Strategic thinking, uncertainty, chaos theories, holographic organization, non-traditional, emergent design, people-centric, competent people and their relationships, high customer satisfaction, quick delivery, active participation, creating and leveraging change, short iterative cycles, intense collaboration, self-organizing teams, developer discretion, learning, teamwork, self-organization, personal empowerment, responsiveness, flexibility, hierarchy, role interchangeability and autonomy.
9	[IEEE, 2007]	Capability to accommodate uncertain or changing needs up to a late stage of the development (until the start of the last iterative development cycle of the release).	Iterative, responsive
10	[Conboy, 2008]	Continual readiness of an entity to rapidly or inherently create change, proactively or reactively embrace change, and learn from change while maximizing value, through its collective components and its relationships with its environment	Speed, flexibility, responsiveness to change
11	[Association for Project Management, 2019]	An umbrella term refers to a project management approach based on delivering requirements iteratively and incrementally throughout the life cycle.	Umbrella term, project management, Iterative, incremental

The definitions mentioned above partially cover the same points of emphasis as the *“Agile Principle (see Table 3.1)”*. However, these definitions use slightly different terms or viewpoints on *“Agility”* [Laanti et al., 2013]. For example, Cockburn [Cockburn, 2002] considers that agile is about *“communication”*, and Ambler [Ambler, 2007] considers that agile it is about *“collaboration”*. According to Laanti et al. [Laanti et al., 2013], these kinds of nuances might seem irrelevant, but they can cause confusion in a large organization when Agile Methods are being used.

Researchers seem to avoid defining agile/agility, or define it via reference to a few existing sources, or define agility via the methods researched. Today, when searching the agile definition, it can be found that the newest definitions of *“Agile”* have ceased to mention effectiveness. Nevertheless, *“Agile”* is described as *a “set of practices that they can be used when doing systems improvements”* [Laanti et al., 2013]. It was concluded that these agile activities and practices are organized in ways that prescribe the workflows, that should be performed, and explain how the products should be produced and handled, along these flows. The flow of activities respects agile principles. Therefore, in this thesis, research works, and related projects, it is consider that the ***agility attribute of Agile Product Lines methodologies concern the software development process and not the developed product***. From this perspective, the following definition of “agile or agility” are considered:

Definition 3.1:

Agile or agility term refers to an umbrella term used for a group of related approaches to software development [process] based on iterative and incremental development. [Kenneth, 2013]

In this definition, the emphasis lies on *“a group of related approaches”*, *“iterative”*, and *“incremental”*. Incremental development is a staging and scheduling strategy, in which various parts of the system are developed at different times or rates and integrated as they are completed. Iterative development is a rework scheduling strategy in which time is set aside to revise and improve parts of the system.

Accordingly, Agile Software Development (ASD) is a way of managing and organizing the development process, emphasizing direct and frequent communication, delivering frequently of *working-software increments*, having short iterations, involving active customer engagement throughout the whole development life cycle, and being change responsive rather than change avoidance. This is in contrast to waterfall-like models, which emphasize thorough and detailed planning and upfront design, and conformance to consecutive stages of the plan [Hanssen et al., 2018].

3.3 Iterative and Incremental

Generally, Agile is understood as more of an evolution rather than a revolution. The school of iterative development and incremental delivery, which was well established by the 1980s, heavily influences agile methods. In fact, all existent agile processes are *iterative* and *incremental* approaches to software development. The terms iterative and incremental, each have a unique meaning and are often used together [Cohn, 2013] [Larman, and Basili, 2003].

Incremental development (or delivery) involves building a system (or product) piece by piece. At the beginning, the first part of the product is developed, and then a next part is added to the first part, and so on. Alistair Cockburn [Cockburn, 2008] describes the incremental development as a *"staging and scheduling strategy, in which various parts of the system are developed at different times, or rates and integrated as they are completed"*. An alternative strategy is to develop the entire system with a *big-bang integration* at the end.

Iterative development is referred to as a *"rework scheduling strategy, in which time is set aside to revise and improve parts of the system"* [Cockburn, 2008]. Therefore, this approach – in whatever method – prefers to browse the entire solution development lifecycle (i.e., analysis, implementation and testing, deployment) with the aim of producing a self-contained, tested and partially functional product within a fixed timeframe [Moran, 2015]. During successive iterations, the product is further refined thereby enabling lessons learned from earlier iterations to be fed back into the process. However, the iterative development process acknowledges the impossibility (or at least improbability) of getting a feature right the first time [Cohn, 2013]. The alternative strategy to iterative development is to plan to get everything right the first time [Cockburn, 2008].

Briefly, in an incremental process, a development team fully develops one feature and then moves onto the next feature. By contrast, in an iterative process, they build the entire system, but with some imperfectly at first, and then they use subsequent passes across the entire system to improve the built system.

Several agile methodologies have merged these two approaches into one in order to refer to the process of *"iterative and incremental development"* (e.g. Scrum). According to Jeff Sutherland [Sutherland, 2010], one of the leading advocates of Scrum, the Scrum method describes iteration as the act of traversal of the entire process during each pass of which the product gradually developed. In addition, he considers that increments are concerned with the concept of "incremental development", which is iterating overall the *Sprint* and that each iteration should conclude with *"minimal usable feature set that is potentially shippable"* [Sutherland, 2010]. In practice, there is a little consensus in the agile community concerning the precise definition of an increment. Furthermore, there is no precise definition about where effectively the boundaries of agility lie [Moran, 2015].

3.4 The “Agile Manifesto”

In 2001, in Utah, seventeen software practitioners were gathered; they had written the Agile Manifesto [ManifestoAgile, 2001] for agile software development. These practitioners were the representatives from *“eXtreme Programming”*, *“Scrum”*, *“Dynamic Systems Development Method (DSDM)”*, *“Adaptive Software Development (ASD)”*, *“Crystal”*, “*Feature-Driven Development (FDD)”*, *“Pragmatic Programming”*, and others. The main objective of that meeting was to discuss the practices of each methodology that were successful in software development in the late 1990s and to try to understand the common ground of each methodology [Heng, 2017]. While the participants did not agree about much, they found consensus around four main values [Agile Alliance, 2018]. In fact, participants did not agree upon many issues. However, they have agreed a few things that become the *Manifesto for Agile Software Development*. The two main achievements of the Agile Manifesto were to provide a set of *value statements* that form the foundation for Agile software development and to coin the term *Agile software development* itself [Agile Alliance, 2018].

In his book, Sommerville [Sommerville, 2010] has stated that the Agile Manifesto is a *“set of principles encapsulating the ideas underlying agile methods of software development”*. Actually, the manifesto for agile software development gather four values and twelve principles.

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:			
1	**2**	**3**	**4**
Individuals and interactions over processes and tools	**Working software** over comprehensive documentation	**Customer collaboration** over contract negotiation	**Responding to change** over following a plan
That is, while there is value in the items on the right, we value the items on the left more”.			

Fig. 3.2 Values of the Agile Manifesto [ManifestoAgile, 2001].

Figure 3.2 highlights the values of the agile manifesto. It is clear that the manifesto was written in a structured sentence with the word *“over”* in the middle. As the values are partially written in bold, this means that the agile manifesto emphasizes these parts of the values more [Ambler, 2002]. In addition, based on Figure 3.2, agile methods focus on *individuals* and *interactions* more than on *process* and *tools*. According to Hunt, the project success relies mostly on the *people* involved and the way in which they communicate more than on the processes, methodologies and tools that are used [Hunt, 2006] [Hazzan, and Dubinsky, 2008].

A set of *12 principles* have been identified from these value statements. These principles have two main objectives:

i. They are intended to help people gain a better understanding of what agile software development is all about;
ii. They can be used to help to determine whether the developer is following an agile methodology or not.

Moreover, these principles do not stipulate a specific method. Instead, they define a set of guiding statements, where any method has to be conform to these statements in order to belong the banner "Agile". Therefore, agile methodologies should be conform to these principles. Table 3.2 presents the twelve principles that are based on the Agile Manifesto.

Table 3.2 Principles of Agile Manifesto [ManifestoAgile, 2001].

P1	*Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*	**P7**	*Working software is the primary measure of progress.*
P2	*Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*	**P8**	*Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
P3	*Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*	**P9**	*Continuous attention to technical excellence and good design enhances agility.*
P4	*Business people and developers must work together daily throughout the project.*	**P10**	*Simplicity – the art of maximizing the amount of work not done – is essential.*
P5	*Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*	**P11**	*The best architectures, requirements, and designs emerge from self-organizing teams.*
P6	*The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*	**P12**	*At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

Table 3.3 shows the dependencies of the agile manifesto values and its principles. This table was adapted from [Heng, 2017] and the dependency relationship is represented by "X".

Table 3.3 Dependencies of agile manifesto values and principles [Heng, 2017].

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Value (1)	X			X	X	X		X				X
Value (2)	X		X				X		X	X	X	
Value (3)	X	X		X				X				
Value (4)	X	X	X				X					(X)

In a nutshell, *"Agile"* is a mindset informed by the values contained in the *Agile Manifesto* and the 12 Principles behind this manifesto. Those values and principles provide guidance on how to create and respond to change and how to deal with uncertainty [Agile Alliance, 2018].

3.5 Agile Software Development in practice

In general, an *agile team* has to balance the *"need for adaptation (i.e. innovation)"* against the potential *"pressure to standardization"*. To do this, it was suggested that *"lightweight"* methods (e.g. Scrum and XP) maintain high adaptation and low optimization environments, whereas, *"heavier"* methods (e.g. DSDM and SAFe) are to be found in low adaptation and high optimization contexts [Cockburn, 2006] [Moran, 2015]. It was argued that an organization typically requires both.

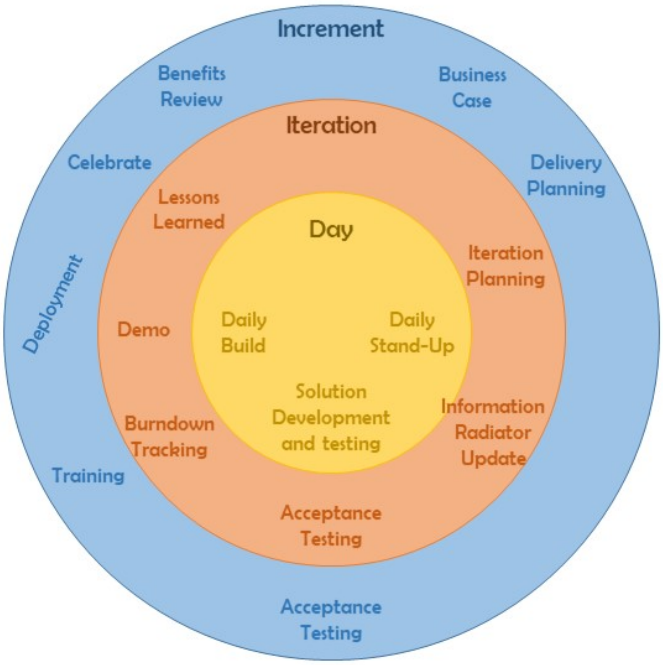

Fig. 3.3 Agile Chart for a Generic Agile Process. Adapted from [Moran, 2015].

Figure 3.3 sketches the agile chart for a generic agile process. The *small circle* represents the day of an agile team. In practice, a typical agile day starts with a stand-up meeting of the project team members. Then, during the day, the code could be developed, tested and integrated into a shared repository by using continuous integration practices and by that means, ensuring a tight feedback loop. At the end of each day, a complete build and deployment may be performed to assess the stability and readiness of the code, as well as demonstrating the working software. Furthermore, technical practices performed on a daily basis tend to be highly automated. The *middle circle* represents the iteration of an agile process. Continuing with this next cycle, each iteration begins with a planning session in which estimates and priorities are set (i.e. definition of *user stories*, etc.). In every part of the iteration, the communal information radiator has to be updated with relevant information, user acceptance testing could be performed

and the progress is tracked by using some form of burndown charts that exhibit a high degree of transparency. With regard to the end of the iteration, the project team meets stakeholders in order to demonstrate the work completed during the iteration. To conclude an iteration, the team reflects on its experiences and lessons learned (i.e. retrospective) and considers what might be done to improve the process [Derby, and Larsen, 2006]. Generally, an iteration length typically varies between two to four weeks. The *bigger circle* represents the increment of an agile process. In fact, an increment needs affirmations of the business case and high-level delivery planning. Which often results in a feature list (i.e. backlog) that describes the requirements in the language of the customer at a level of detail commensurate with the information available. Acceptance testing of the evolving product usually occurs on all levels. However, it is a definitive character at the increment level and is expected to complement the integration testing that occurs during the iteration and the unit testing in the daily cycle. At the end of the increment, a final deployment of a part (or complete) solution provides the welcome opportunity to celebrate delivery of business value [Derby and Larsen, 2006] [Moran, 2014].

According to [Dingsøyr et al., 2010], agile methods are seen as a reaction to *plan-based* or traditional methods that emphasize a rationalized and engineering-based approach. In fact, since the creation of the Agile Manifesto, in 2001, agile methods have gained a lot of popularity in the software industry due to its numerous success stories. The popularity of agile methods has been verified in several ways. For example, in the *CHAOS report* [Hastie and Wojewoda, 2015] it was demonstrated empirically that (in practice) agile methods are more successful and less likely to fail comparing to waterfall-based methods.

Table 3.4 Agile Vs. Waterfall. Adapted from 2015 CHAOS Report [Hastie and Wojewoda, 2015].

SIZE	METHOD	SUCCESFUL	CHALLENGED	FAILED
Small size projects	Agile	58 %	38 %	4 %
	Waterfall	44 %	45 %	11 %
Medium size projects	Agile	27 %	62 %	11 %
	Waterfall	7 %	68 %	25 %
Large size projects	Agile	18 %	59 %	23 %
	Waterfall	3 %	55 %	42 %
All sizes	Agile	39 %	52 %	9 %
	Waterfall	11 %	60 %	29 %

Table 3.4 shows the importance of agile methods for practitioners. In fact, it highlights the results of the CHAOS report related to success and failure of using agile and waterfall methods in surveyed software projects.

Every year, *CollabNet-VersionOne* introduces the annual State of Agile Report. Figure 3.4 shows the percentages of common agile methodologies used within organizations. According to the 13th annual report, issued in 2019, *Scrum* still is the most popular agile methodology with being used in 54% of all agile projects. The use of *Scrumban* method has increased by 1% since 2017. *XP* has become unpopular with a use of only 1%. However, a lot

of XP's relevant engineering techniques (e.g., User stories, Pair Programming, Planning game, etc.) are still very popular and used in many agile methods.

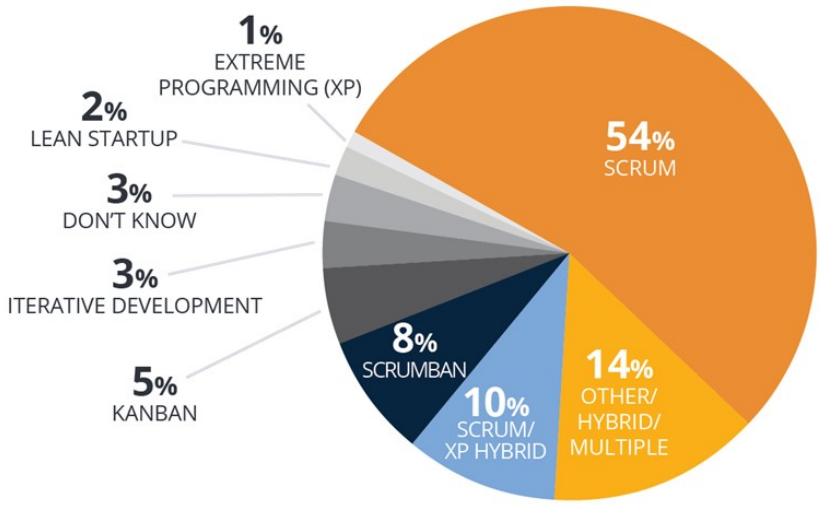

Fig. 3.4 Common agile methodologies used in respondents' organizations [CollabNet VersionOne, 2019].

3.6 Overview of main Agile Methods

This section provides an overview on two agile methods, namely Scrum and Scrumban. Section 3.6.1 reviews the Scrum methodology and Section 3.6.2 presents the Scrumban methodology. Scrum has been chosen because it is widely used within organizations and integrates project management in the same family of methods. The Scrumban has been selected because it integrates relevant engineering practices that fit Software Product Lines and integrates project management techniques as well.

3.6.1 Scrum

In the early 1990s, Jeff Sutherland introduced the Scrum methodology for software development [Sutherland and Schwaber, 2011]. This methodology was inspired by a paper of Takeuchi and Nonaka [Takeuchi and Nonaka, 1986]. Scrum is an exceptionally elegant, effective, and popular software development framework. Scrum's value increases as teams and organizations advance their understanding and application of its core principles and practices [Kenneth, 2013].

According to Craddock [Craddock, 2013], Scrum could be described as a product development methodology with mild project management aspirations (e.g., lightweight tracking and reporting),; its emphasis lies mainly on the management of software requirements and development. As Scrum focuses on the project management aspects for software development, the scope of Scrum does not reach other activities, such as business change

management, systems development or data migration, and it therefore defers to existing practices within an organization to cover project initiation, risk management, release and deployment and change management processes [Abrahamsson, 2003]. Indeed practitioners of agile methodologies, which have a wider scope, have argued that Scrum can be successfully embedded within their frameworks [Schwaber and Sutherland, 2016].

Scrum has three main concepts: *transparency*, *inspection*, and *adaption* [Tsui et al., 2013]. In addition, it has a common definition of what can be defined as a *completed item*. In practice Scrum is an *iterative* and *incremental* methodology based on short time-box iterations of maximum 30 days, which are known as *"sprint"*; each sprint produces a piece of workable software [Cohn, 2013]. It is important to mention that at the beginning Scrum consisted only of a few practices. Later, the authors were inspired by the values of "Agile Manifesto" and thus, they came out with five principles, namely *commitment*, *focus*, *openness*, *respect*, and *courage* [Schwaber and Beedle, 2001]. In few words, Scrum is a refreshingly simple, people-centric framework based on the values of honesty, openness, courage, respect, focus, trust, empowerment, and collaboration [Kenneth, 2013]. The Scrum practices are embodied in specific *roles*, *activities*, *artifacts*, and their associated *rules*.

3.6.1.1 Scrum roles

According to [Cohn, 2004], [Cohn, 2013], and [Kenneth, 2013], *Scrum* presents the three following core roles:

- *Scrum Master:* the Scrum Master is responsible for making sure that the team is as productive as possible. He/she does this by helping the team use the Scrum process, by removing impediments to progress, by protecting the team from outside, and so on. Notice that the Scrum Master is not acting in a managing role, but rather as a coach of the team, as a true part of the development team. He/she interacts with the development team, customers and the management during the project;

- *Product Owner*: the product owner is the project's key stakeholder and represents the users, customers and others in the process. The product owner is often someone from product management or marketing, a key stakeholder or a key user. The product owner is responsible for creating and prioritizing the Product Backlog, setting the scope of the sprint, and reviewing and accepting the software at the end of each sprint;

- *Scrum Team (Development Team)*: A typical Scrum team counts between five and nine members, but Scrum projects can easily scale into the hundreds. However, Scrum can also easily be used by one-person teams and often is. This team does not include any of the traditional software engineering roles such as a programmer, designer, tester or architect. Everyone on the project works together to complete the work they have collectively committed to complete within a sprint. Scrum teams develop a deep form of camaraderie and a feeling that "they're all in it together". In other words, a Scrum team is a self-organizing team that is involved in estimating, creating the Product

Backlog (described in the next section), decomposing task for implementation of each Product Backlog item, and reviewing Product Backlog, etc.

3.6.1.2 Scrum artifacts

According to [Cohn, 2014] and [Kenneth, 2013], Scrum methodology proposes three main artifacts that ensure the control of the project. These artifacts are openly accessible for any member of the team. These artifacts are the following:

- *Product Backlog*: The product backlog is a prioritized feature list containing every desired feature or change to the product. The term "backlog" can get confusing, because it is used for two different things. To clarify, the product backlog is a list that stores the current requirements of the product. It does not constitute the complete requirements set of the software, because the items that have already been implemented are discarded from the Product Backlog and other items can be added afterwards. An item of the Product Backlog normally consists of a feature, a functionality of the system and its priority. Sometimes, a user story card expresses the feature;

- *Sprint Backlog*: It is similar to the Product Backlog, but it is for a specific sprint. It is the subset of the Product Backlog. The Product Owner decides and selects which Product Backlog item should go into the current Sprint Backlog for the implementation. This selection will produce a workable software at the end of the sprint. Every item in the Sprint Backlog is further decomposed in a set of small task and each task is assigned to a specific developer;

- *Burndown Chart*: this chart displays the remaining efforts of the project. It is a tool for measuring the progress in software development. The Scrum Master updates it at the end of each sprint.

3.6.1.3 Scrum practices

According to [Tsui et al., 2013], [Cohn, 2014] and [Kenneth, 2013], Scrum integrates several practices, sometimes called "Scrum Events". The four core practices of Scrum methodology are the following:

- *Sprint planning meeting*: At the start of each sprint, a planning meeting is held, during which the product owner presents the top items on the Product Backlog to the team. The Scrum team selects the work they can complete during the coming sprint. That work is then moved from the product backlog to a sprint backlog, which is the list of tasks needed to complete the product backlog items the team has committed to complete in the sprint.

- *Daily Scrum*: It is a short meeting (maximum 15 minutes) that occurs every day during the sprint. The main target of this meeting is to ensure that the team members are on the track. It is a session of asking the three (in)famous questions:
    - What have you done since the last Scrum?
    - What will you do between now and the next Scrum?
    - What got in your way of doing work?

  If a member encounters a problem, she/he can ask for help from the others. The Scrum Master normally organizes this meeting. This meeting helps set the context for each day's work and helps the team to stay on track. All team members are required to attend the daily scrum.

- *Sprint review meeting*: At the end of each sprint, the team demonstrates the completed functionality at a sprint review meeting, during which, the team shows what they accomplished during the sprint. Typically, this takes the form of a demonstration of the new features, but in an informal way; for example, PowerPoint® slides are not allowed. The meeting must not become a task in itself, nor a distraction from the process.

- *Sprint retrospective*: Besides this, at the end of each sprint, the team conducts a sprint retrospective: a meeting during which the team (including its Scrum Master and product owner) reflects on how well Scrum is working and what changes could be made for it to work even better. The team considers three things:
    - What went well?
    - What didn't?
    - What improvements could be made for the next sprint?

  It is essential for the team to have this meeting as a self-evaluation and to strive for self-improvement.

3.6.1.4 Scrum Process

Abrahamsson et al. [Abrahamsson et al., 2002] have divided the Scrum process in three phases. (Figure 3.5)[7] introduces a graphical representation for the following phases of Scrum process:

1. *PreGame Phase*: The Scrum process starts with the requirements gathering by creating the *Product Backlog*. It is not required to be complete from the beginning and it can be updated after each sprint. This phase is separated into two sequential steps:
    - *Planning*: During the planning, each item in the Product Backlog is estimated with priority and efforts for its implementation, and the scope of the sprint is set—i.e., moving the selected items from Product Backlog to *Sprint Backlog*, with schedule and cost.

[7] Used icons are royalty-free.

– *Architecture*: The architecture defines how the items in the Sprint Backlog are implemented. It involves the system architecture modification and *high level-design*.

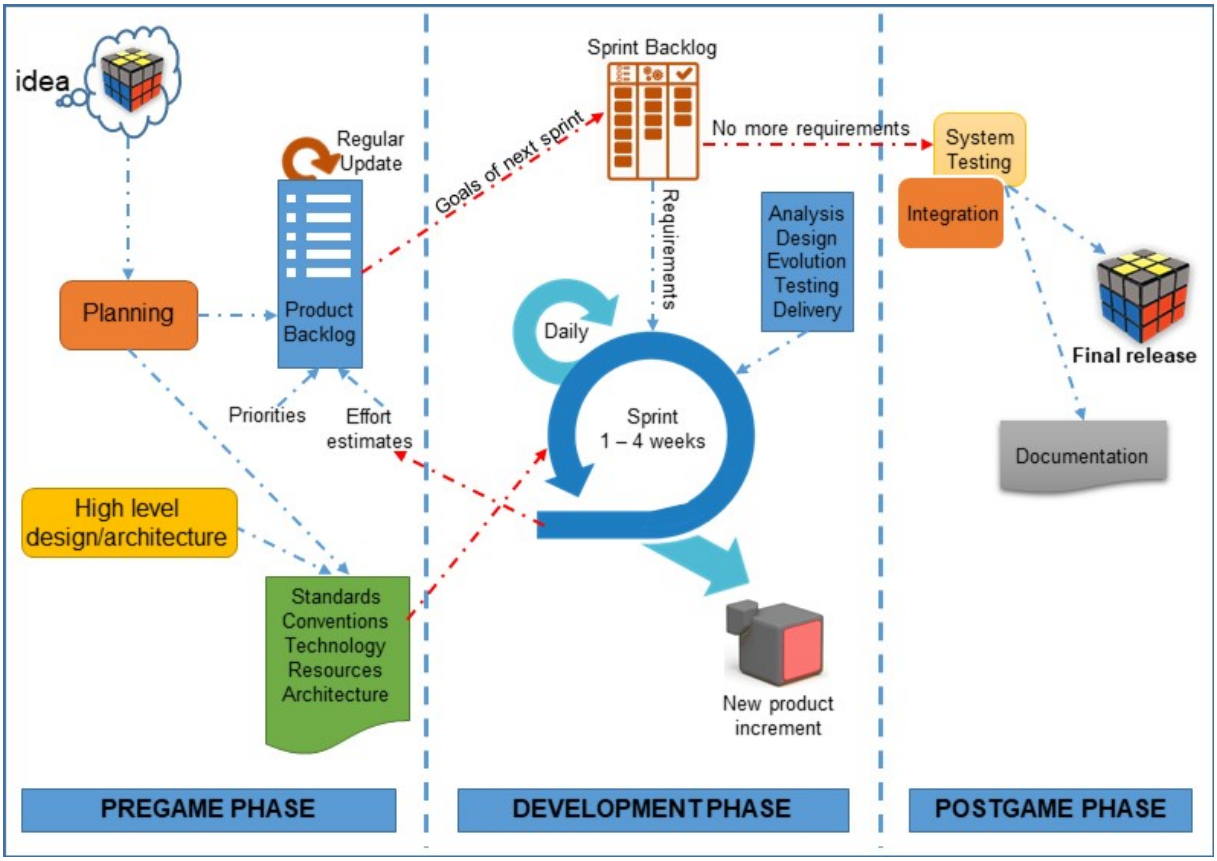

Fig. 3.5 Scrum's lifecycle. Adapted from [Abrahamsson et al., 2002].

2. *Development Phase (*also called the *Game Phase)*: the development phase is considered as the *agile* part of the Scrum approach. It is treated as a *"black box"*, where unpredictability is to be expected. The various and ever-changing environmental and technical variables (such as *timeframe*, *quality*, *requirements*, *resources*, *implementation technologies and tools*, and even *development methods*), identified in Scrum, are observed and controlled through various Scrum practices during the *Sprints* of the development phase. Rather than considering these matters only at the beginning of the software development project, Scrum aims at controlling them continuously in order to be able to flexibly adapt to the changes. In addition, any changes to aspects such as requirements, priority, efforts, etc. are prohibited.

3. *PostGame Phase*: This phase contains the closure of the release. This phase is entered when an agreement has been made on whether the environmental variables, such as the requirements, have been completed. In this case, no additional items and issues can be found nor can any new ones be invented. The system is now ready for release and the final preparations for this are performed during the PostGame Phase, including the tasks such as the integration, system testing and documentation.

3.6.1.5 Discussion

As stated above, Scrum is not a standardized process where a series of sequential steps are methodically followed, which are guaranteed to produce a high-quality product that delights customers, both on time and on budget. Scrum is presented as a project management framework that is applicable to any project with aggressive deadlines, complex requirements and a degree of uniqueness. Kenneth [Kenneth, 2013], Craddock [Craddock, 2013] and Cohn [Cohn, 2014] argue that organizations that apply Scrum diligently experience various benefits, such as: delighted customers, improved return of investment, reduced costs, fast results, confidence to succeed in a complex world, etc. It is demonstrated that with Scrum, not only customers are delighted, but also the people doing the work actually enjoy it. They enjoy frequent and meaningful collaboration, leading to improved interpersonal relationships and greater mutual trust among team members.

According to Ken Schwaber [Schwaber, and Sutherland, 2016], co-creator of Scrum, "*75% of those organizations using Scrum will not succeed in getting the benefits that they hope for from it…*". Thus, Scrum also presents some disadvantages. For Cline [Cline, 2015], usually, the Scrum methodology is applied for small 5-8 people teams. Team members must be committed to the project, as this framework requires an experienced team. If a team consists of novices in this area, there might be a risk of not completing the project on time. Moreover, strict control over the team might put a lot of pressure on them, which on its own may also lead to failure. In addition, Cockburn [Cockburn, 2007] and Misevičiūtė, [Misevičiūtė, 2016] have highlighted that the *"Scrum method may consume a lot of time if a longer sprint is planned. Unexpected issues may also hinder the process of completing a sprint on time, thus more time will be needed to remove those issues. Moreover, the scrum estimation is one of the most strenuous parts, as tasks must be well defined, otherwise estimated project costs and time will not be precise"*.

Briefly, Scrum is a constrained process, where tasks are assigned to each team member (i.e. all tasks might be bounded by deadlines). According to Abrahamsson et al. [Abrahamsson et al., 2002], and Sutherland and Schwaber, [Sutherland, and Schwaber, 2011] Scrum works well for large projects, because work is organized in *sprints,* which in part are planned by all scrum team members. After one sprint ends, a new sprint is planned, therefore the board is reset for each sprint. Scrum has predefined roles like *Scrum Master*, *Product Owner* and *Development Team*. Scrum teams are cross-functional, track their workflow using *Burndown Charts,* and have daily meetings where each member elaborates on the current situation of his/her work. Larger enterprises, with larger projects usually use scrum to achieve high efficacy within a working process.

3.6.2 Scrumban

Scrumban is a relative newcomer to the world of agile software development. Ladas [Ladas, 2009] has defined Scrumban as *"a transition method for moving software development teams from Scrum to a more evolved development framework"*. More precisely, Scrumban is *"a composite of Scrum and Kanban methods, as it contains basic properties of Scrum and*

flexibility of Kanban" [Brezočnik, and Majer, 2016]. Organizations have placed the Kanban Method alongside Scrum to help them achieve varying kinds of outcomes. In other words, Scrumban is a management framework that emerges when teams employ Scrum as their chosen way of working and use the Kanban Method as a lens through which to view, understand and continuously improve how they work [Reddy, 2015].

According to [Nikitina et al., 2012], complementing the principles of Scrum with practices of Kanban can yield a more sophisticated development methodology (i.e. Scrumban) and thus, it can guarantee the highest product quality. In fact, Scrumban combines the benefits of both Scrum and Kanban. Scrum can effectively facilitate the management of projects by providing principles that urge cooperation among the *development team* members. This all in order to accomplish the required work, which has been divided in several *sprints* with *fixed length,* and provide aid where the *scrum team* should be more *"a cross-functional"*. Kanban limits the *work in progress*, *measures the lifetime of project*, and *monitors and manages the workflow* by applying Kanban board.

Experiences demonstrate that Scrumban has evolved to become a family of principles and practices that create complementary tools and capabilities. Over the years, Scrumban has been used to help teams and organizations accelerate their transitions from Scrum to other development methodologies. It has been used to help teams and organizations overcome a variety of common challenges that Scrum is designed to force them to confront. When the context requires, it has been used to help organizations evolve new *Scrum-like* processes and practices that work best for them— not simply as a means to accommodate inadequacies and dysfunctions Scrum exposed, but rather as a strategy to resolve those problems in a manner, that is most effective for that environment [Reddy, 2016].

Following Reddy [Reddy, 2015], Scrumban can be distinguished from *Scrum* in the way that it emphasizes certain principles and practices that are substantially different from Scrum's traditional foundation. These include the following:

- Recognizing the important role of organizational management (self-organization remains an objective, but within the context of specific boundaries);
- Allowing for specialized teams and functions;
- Applying explicit work policies ;
- Applying the laws of flow and queuing theory;
- Deliberate economic prioritization.

On the other part, Scrumban is distinct from the *Kanban Method* in the following principles and/or practices [Reddy, 2015]:

- It prescribes an underlying software development process framework (Scrum) as its core;
- It is organized around teams;
- It recognizes the value of time-boxed iterations when appropriate;
- It formalizes continuous improvement techniques within specific ceremonies.

From practice, it was learnt that the principles and practices of Scrumban are not unique to the software development process. These principles and practices can be easily applied in many different contexts, providing a common language and shared experience across *interrelated business functions*. Consequently, this enhances the kind of organizational alignment that is an essential characteristic of success [Reddy, 2015] [Misevičiūtė, 2016] [Johnson, 2016].

3.6.2.1 Scrumban roles

Like Scrum and Kanban, Scrumban has several roles. According to mm1 Technology GmbH [mm1, 2016] and Reddy [Reddy, 2016] these roles are the following:

1. *Product owner:* the person responsible for maintaining the product backlog by representing the interests of the stakeholders, hereby ensuring the value of the work the development team performs.

2. *Scrumban sensei:* the person responsible for correct use of the Scrumban process. Although the designation of a Scrumban sensei, and his/her presence in (Scrumban) meetings, is generally advisable, teams with a lot of Scrumban experience may also work without this role.

3. *Development team:* a cross-functional group of people responsible for delivering potentially shippable increments of the product (at the end of every production cycle).

4. *Stakeholders:* the people enabling the project. They are only directly involved in the process during the reviews. Aside from that, they may solely influence the team by discussing their needs with the product owner. Typically, the main stakeholders are managers, customers and users.

3.6.2.2 Scrumban artifacts

According to mm1 Technology GmbH [mm1, 2015], Reddy [Reddy, 2016] and Nikitina [Nikitina et al., 2012], Scrumban framework proposes ten main artifacts. Namely, the following:

1. *Product backlog**:*** an ordered list of requirements that the team maintains for a product. In Scrumban, one should document requirements in “User Story” format. Anyone can edit the backlog, but the product owner is ultimately responsible for ordering the user stories. Stories in the product backlog contain rough estimates of both business value and development effort.

2. *Selected backlog:* a list of work the development team must address next. It has a defined capacity limit. As soon as capacity is available, it is filled up with user stories/features from the top of the product backlog.

3. *Story in Progress (SIP) Backlog:* a list of user stories, which the development team currently addresses. Team members pull user stories from the selected backlog when there are no more remaining tasks in the task backlog.

4. *Task backlog:* a table structured along the phases that are necessary for completing the project, e.g. design, development, and test. The development team breaks the user stories/features from the SIP backlog down into single tasks. Once a task has finished one phase, a team member from the consecutive phase eventually pulls the task to process it further.

5. *User Story:* a description of a certain product feature or behavior, written strictly from the user's point of view. Usually, the product owner writes the user stories.

6. *Task:* a unit of work, which should be achievable within one working day or less. To implement a user story, you must accomplish all associated tasks.

7. *Work in Progress (WIP) Limit:* Limits the number of stories and tasks in each productive steps of the production flow and thus prevents work overload.

8. *Parking lot:* a space for tasks, which the team could not finish, due to external dependencies. For example, another team has to review a document. Placing a task in the parking lot prevents the team from encountering deadlocks, where unfinished tasks block production lines.

9. *Cumulative flow diagram (CFD):* a publicly displayed chart showing a detailed view of the teams' past and present performance. The CFD allows the identification of bottlenecks within the production flow. It also enables the product owner to predict the time a new requirement will most probable need to complete.

10. *Impediment backlog:* a list maintained by the sensei, including all current impediments.

3.6.2.3 Scrumban practices

Since Scrumban combines Scrum and Kanban and contains the best rules and practices of both methods. The Scrumban method integrates many practices from both. On one hand, it uses the practices that reflect the sanctioned nature of Scrum to be *"agile"*. On the other hand, it encourages teams to constantly improve their processes along with Kanban's aim of continuous improvement (i.e. Kaizen) [Pahuja, 2018]. According to [mm1, 2015], [Reddy, 2016] and [Ladas, 2009] the main practices of Scrumban method are the following:

1. *Planning 1:* (The "what": Whenever the product owner pulls new user stories into the selected backlog.) The product owner holds the decision of "what?" in order to select the next user stories to work on, explaining the user stories of the product backlog and answering open questions. After this analysis, the development team should understand

the requirements. Therewith, the team is able to estimate the complexity of each user story.

2. *Planning 2:* (The "how": Whenever team members pull new user stories into the production flow.) Here, the team discusses solutions for new user stories in the SIP backlog and accordingly creates tasks for each user story.

3. *Daily:* (15 min max.) A short, time-boxed meeting, taking place every day at the same time. Every team member answers three questions:

    a. What have I done since yesterday?
    b. What am I planning to do today?
    c. What are my impediments?

4. *Review:* (Whenever the team ships an increment.) The team uses this meeting to present and review the work it has completed since the last delivery. Usually, it also includes a demonstration of the features created in the last product increment.

5. *Retrospective:* (After any review.) The Scrumban sensei holds the retrospective to reflect on the past production cycle in order to ensure continuous process improvements. The sensei always asks two questions in the retrospective:

    a. What went well during the last cycle?
    b. What should improve in the next cycle?

6. *Andon:* (Whenever a problem occurs.) The Scrumban sensei organizes an Andon meeting whenever problems in the production flow occur. For example, a story goes over the expected cycle time, or a task is frequently re-assigned and not yet solved. Both the development Team and the product owner take part in this meeting and work on a solution to solve the pending issue.

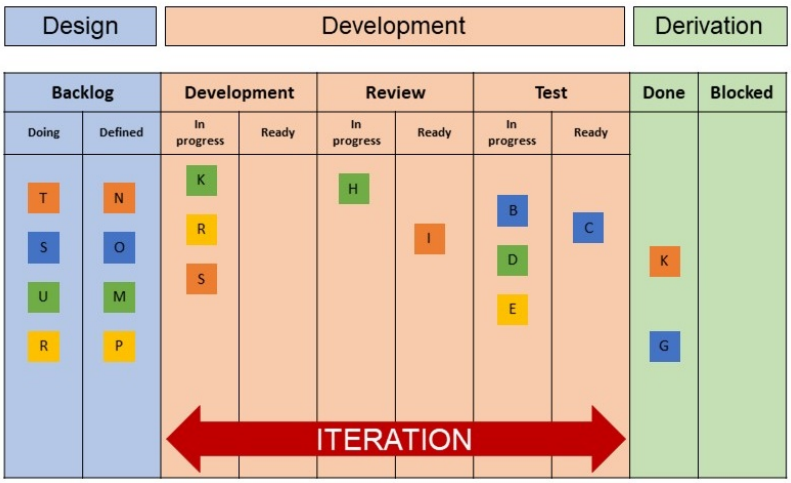

Fig. 3.6 Example of Scrumban Board that gives an overview of a process workflow.

7. *Extend Board:* the workflow is visualized so that the team can follow how the tasks move from the initial request to completion. This provides both a sense of project scope and understanding of the end goal. It consists of columns, headed with *"To do"*, *"Work in progress (WIP)"* and *"Done"*. Column *"Work in progress"* can be divided into more sections, then new columns indicate the particular stages a task goes through, therefore everybody knows the current situation and tasks that are to be completed as soon as possible (See Figure 3.6).

8. *Limit the WIP (Backlog limit):* Each team member should be working on no more than one task at a time. To reinforce this rule WIP limits from Kanban are used, limiting the number of tasks in the progress columns. This reinforces team collaboration and ensures any bottlenecks are resolved quickly.

9. *Plan on demand:* to save time and minimize waste, the planning is done only when necessary. The amount of tasks to be planned for an iteration is controlled, by putting a limit on the backlog column. The task limit is based on team capacity and prior iterations.

10. *Bucket size planning:* this approach is used for long term planning within the Scrumban framework. It is based on three different phases of planning – 1 year for the long-term perspective, 6 months for committing to specific goals and 3 months for setting up clear requirements. This assists Scrumban teams in having a roadmap of actions for the long-term perspective (See Figure 3.7).

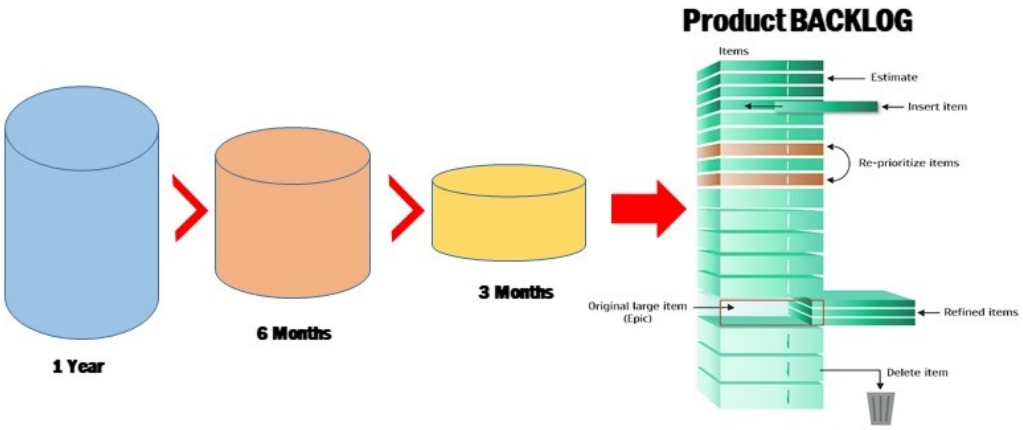

Fig. 3.7 Bucket size planning. The backlog icon is adapted from [Kenneth, 2013].

11. *Lead and cycle time (Metrics performance): Lead and cycle time* are terms taken from Kanban, it defines the required time from the initial request to task completion, or the time from starting the task to its completion (See Figure 3.8). It is used in Scrumban to estimate how long the iteration will last and what should the backlog limit be for the team. Scrumban uses average lead and cycle time as its key metrics for performance. If lead and cycle time is under control, then one can understand how long does it takes for a task to reach the end consumer, how long it takes to develop and how long does it take to manage the change (i.e. change management). With these metrics, one can predict

how long it will take to provide a certain amount of value (i.e. product value) or earn some amount of money. In addition, to measure performance, *Cumulative Flow Diagram (CFD)* could be useful. CFD is a chart showing the cumulative number of arrivals and departures from a process, or parts of a process, over a period-of-time (See Figure 3.8).

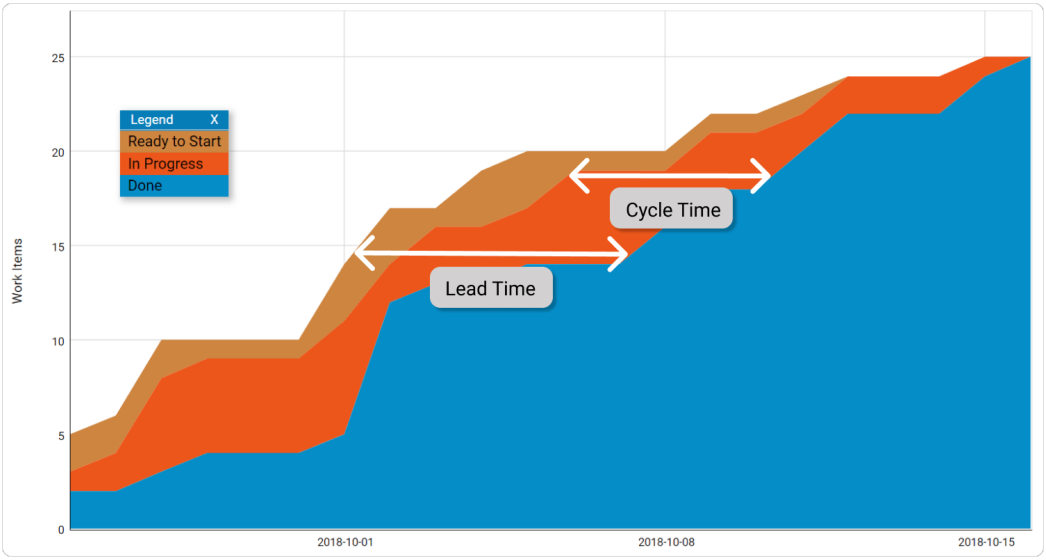

Fig. 3.8 An illustration of Lead, Cycle time and Cumulative Flow Diagram (CFD)

Concisely, Scrumban uses mixed techniques. In fact, it combines the basic features of Scrum and the flexibility of Kanban. Scrumban has a slightly constrained process, where prioritization is optional, but it is recommended during each planning, and planning is done by Kaizen events. Scrumban uses the "planning on demand principle" to fill the backlog and tasks are assigned exclusively using the pull system, like in Kanban. In addition, just like in Kanban, the board stays constant, while only the tasks and their priorities change. In Scrumban the work is usually focused more on planning than releasing; while in Scrum planning is done after each sprint, Scrumban planning is only done on demand. This method is mostly used for fast-paced process like startups or projects which require continuous product manufacturing, where the environment is dynamic [Mahnic, 2014], [Misevičiūtė, 2016], and [Shore Labs, 2017].

3.6.2.4 Scrumban process

In 2008, Ladas [Ladas, 2009] has introduced the term Scrumban in his white-paper on *"Scrumban-Essays on Kanban Systems for Lean Software Development"*. As stated above, on the one hand, Scrumban uses the perspective nature of Scrum in order to be agile. On the other hand, it encourages the process improvement of Kanban to allow teams to continually improve their process [Pahuja, 2018]. Many research works related to Scrumban are available in literature and there are two main schools of thoughts:

- Some apply Scrum to Kanban, where the process is more inclined towards Kanban;
- Other apply Kanban to Scrum, where the process is more inclined towards Scrum;

Both ways seem to take certain principles from Scrum and Kanban and accordingly adjust them to their organization or team needs and requirements [Stoica et al., 2016].

According to many practitioners such as [Suresh, 2018], the core difference between Scrum and Scrumban is in the board itself. Actually, Scrumban has changed the *pull* order of items. In fact, with Scrum, the order of the sprint planning process and all tasks are *pre-set* prior to working. This can cause a level of rigidity that can result in skipping problems that arise during the process, and which may never be addressed prior to release. Moreover, one of the biggest advantages of Kanban brought to Scrumban is that problems are identified throughout the process and can be acknowledged before the end-point is reached. The quality of the final output is generally better. However, as Kanban can give too much liberty to team members to choose their own workflow, this can result in pieces not being completed in a timely fashion, particularly if one task is awaiting the completion of another.

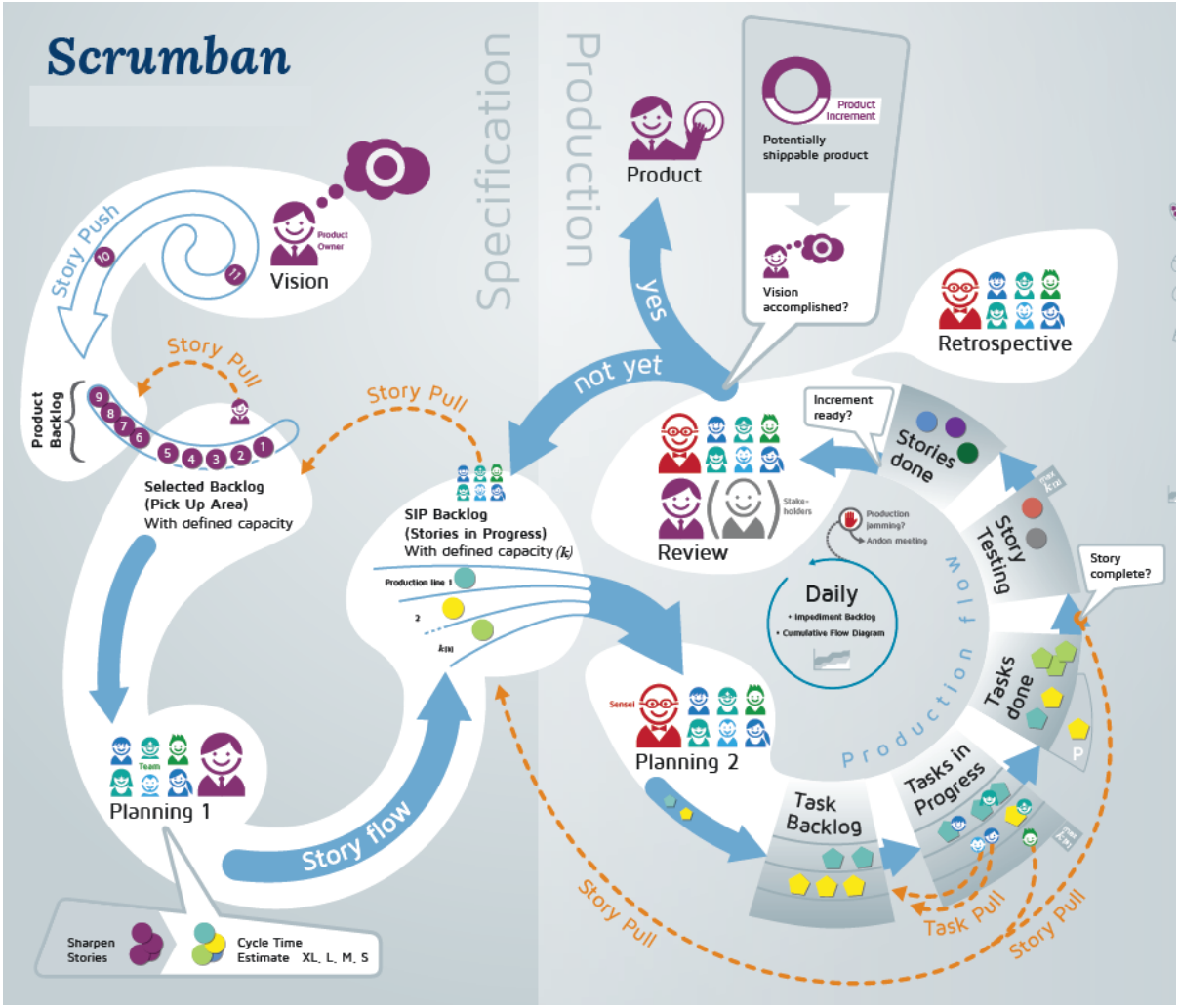

Fig. 3.9 Scrumban process [mm1, 2015].

The main trick with Scrumban is to increase the pool by adding a new area into the existing Scrum workflow. In addition, it does allow a smoother transition to a final goal without being bogged down in individual sprints.

Ladas [Ladas, 2009], Reddy [Reddy, 2016] and other researchers have divided the Scrumban process in two major phases. Figure 3.9 presents a graphical representation of the following phases of Scrumban process:

1. *Specification Phase*: In this phase, the first step is to run a "Kickstart Event", which is a meeting in which the team will nail down the foundational elements of Scrumban, so they can start using it to manage their work. Thus, the phase of specification starts by gathering the requirements from the product owner in order to create the *Product Backlog*. This phase is separated into three sequential steps:

    a. *Step 1.1 – Building of the "Product Backlog" and defining the "Selected Backlog"*: On the one hand, to build the Product Backlog, the *Product Owner* with the *Development Team* lists the requirements and documents them in *"User Story"* format. Anyone can edit the backlog, but the product owner is ultimately responsible for ordering the user stories. Stories in the product backlog contain rough estimates of both business value and development effort. On the other hand, the Development Team defines the Selected Backlog, which is a list of work the development team must address next. It has a defined *capacity limit*. As soon as capacity is available, it is filled up with user stories/features from the top of the product backlog.

    b. *Step 1.2 – Holding of the meeting "Planning 1"*: in this meeting people define the "WHAT?". In fact, the product owner holds this meeting prior to pulling new user stories into the selected backlog, to select the next user stories to work on. The product owner explains the user stories of the product backlog and answers open questions during this gathering. After this analysis, the development team should understand the requirements. Therefore, the team is able to estimate the complexity of each user story.

    c. *Step 1.3 – Construct the "Stories in Progress" Backlog with a defined capacity (k)*: at this stage, the *Development Team* lists the user stories, which they currently address. Team members pull user stories from the selected backlog when there are no more remaining tasks in the *Task Backlog*.

2. *Production Phase*: this phase starts with a planning meeting held to plan the production flow. This phase is separated into three sequential steps:

    a. *Step 2.1 – Holding the "Planning 2"*: at this step the Development Team addresses the "HOW?". Team members pull new user stories into the production flow. In this meeting, they discuss solutions for new user stories in the *Stories In Progress* backlog and create tasks for each user story accordingly.

    b. *Step 2.2 – Production Flow – Defining the Scrumban Board (adapted from Kanban principles)*: here the production flow is organized in a "Kanban Board", which contains several columns. Among these columns, the following columns

are mandatory: “Task Backlog”, “Tasks in Progress”, “Tasks Done”, “Story Testing”, and “Stories Done”.

c. *Step 2.3 – End of Increment/Product derivation*: at the end of each increment, the Development Team presents and reviews the work it has completed since the last delivery. Usually, the “Review” meeting includes a demonstration of the features created in the last product increment. After any review, the Scrumban sensei, and the development team hold the “Retrospective” meeting to reflect on the past production cycle in order to ensure continuous process improvements. Once the Product Owner’s vision is accomplished and the Product is in fact ready; the final product can be delivered.

As a combination of the Kanban Method and Scrum, Scrumban has evolved to become much more than just the “best elements of both.” It encourages the search for improved understandings from all sources, and facilitates the integration other models and frameworks.

3.6.2.5 Discussion

One of the main intentions behind combining Lean and Agile methodologies is to allow project members to receive fast and iterative feedback, while they have the ability to implement the necessary changes and respond to the feedback [Banijamali et al., 2017]. According to Auerbach and McCarthy [Auerbach and McCarthy, 2014], by combining Agile and Lean in co-located projects the coordination between team members is enhanced, the team morale is increased, and therefore, better results are produced. While Scrum helps to make the process flexible, Kanban increases the scale of the development process and makes it more efficient. Ladas [Ladas, 2009] has argued that Scrumban incorporates the iterative planning of Scrum, but is more responsive and adaptive to changes in stakeholders’ requirements. With Scrumban, researchers and practitioners hope to ensure more flexibility in projects as well as the iterative and incremental development.

According to Pahuja [Pahuja, 2018], the adoption of Scrumban method may ensure several advantages, such as the following:

- ✓ Increased quality;
- ✓ Ensuring the “Just-in-Time” principles (i.e. decisions and facts just when they are required);
- ✓ Ensuring short lead time
- ✓ Ensuring continuous improvement (i.e. Kaizen);
- ✓ Minimizing waste (i.e. everything that is not an added value for the customer);
- ✓ Ensuring process improvements by adding some values of Scrum as and when needed;
- ✓ Etc.

However, the implementation of Scrumban presents several challenges as well. According to Karvonen et al. [Karvonen et al., 2012], the flexibility regarding production changes may generate new challenges (e.g., challenges in assigning resources and project timetables). Due to the lean nature of Scrumban that calls for considering the entire organization during the implementation. In addition, the combination of Kanban and Scrumban may increase the

complexities of planning of activities across the whole organization. Furthermore, Rodriguez et al. [Rodriguez et al., 2014] have argued that it is not always possible to include business personnel or management executives to develop product backlogs or receive regular feedback.

In his book, Reddy [Reddy, 2016] presents why Scrumban is interesting from a business perspective. According to him, there is many interesting and important things taking place in the upstream process. As a framework, Scrumban provides views and capabilities that enable a business to evaluate and manage these work processes more effectively [Reddy, 2016]. It emphasizes measuring real experience to:

- evaluate past outcomes in order to make better decisions about present actions
- compare against common benchmarks; as an indicator of relative health
- provide data upon which to base a forecast of future results
- influence the behavior of individuals

Concisely, Scrumban can give teams the power to adapt and change to stakeholder and production needs, without feeling overburdened by their project methodology. It is designed to remove metrics that encourage undesired outcomes. It can restore working time to the team, and avoids unnecessary meetings. Most importantly, it can limit the team's *"work in progress (WIP)",* so that they can finish what they start to a high standard. Scrumban can remove overhead stress for the development team, increase efficiency, and increase the overall satisfaction for the customer [Gambill, 2013].

3.6.3 Comparison between Scrum and Scrumban

As stated earlier in this chapter, agile methodologies range from lightweight approaches, which have a strong product development focus, to heavier methodologies that emphasize architecture orientation or project management. Both Scrum and Scrumban are considered as lightweight approaches. However, Scrumban contains richer elements of project management than Scrum.

Table 3.5 Definitions of methodological dimensions [Moran, 2015]

Dimension	Description
Principles	The core values that characterize the methodology and imbue it with meaning in the eyes of its practitioners
Roles	Distinct roles cited by the methodology. Note that several roles may be assigned to an individual and thus no conclusions should therefore be drawn regarding team size
Artifacts	The intermediate products generated and consumed by the process (omitting final project deliverables). The necessity to create indirect artifacts is a reliable indicator of the weight of a methodology
Practices	The techniques explicitly cited by the methodology as being core to the effective and efficient operation of the process
Phases	The distinct phases of the model underpinning the methodology through which the process must traverse

It was shown that each method has its own culture, practices and language. Accordingly, to distinguish between each method and to identify their appreciation of focus and sphere of application, a brief survey has been highlighted in this section. The survey takes into account

how these methods represent software engineering, product development, project management, and portfolio architectural perspectives on Agile [Moran, 2015]. In other words, Scrum and Scrumban are compared based on the five methodological dimensions of principles, roles, artifacts, practices, and phases. Moran [Moran, 2015] has presented a definition for each dimension. Table 3.5 gathers the definitions of these methodological dimensions. It is clear that each methodology elaborates its own practices and components at different degrees of detail and therefore a certain amount of interpretation is required in order to make reasonable comparisons. This chapter has taken some caution when making the targeted comparison based on the primary sources of Scrum and Scrumban methodologies. In addition, this chapter has adopted the definitions of dimensions of Moran [Moran, 2015] and has followed Moran's analysis, which uses multiple sources about Scrum and Scrumban.

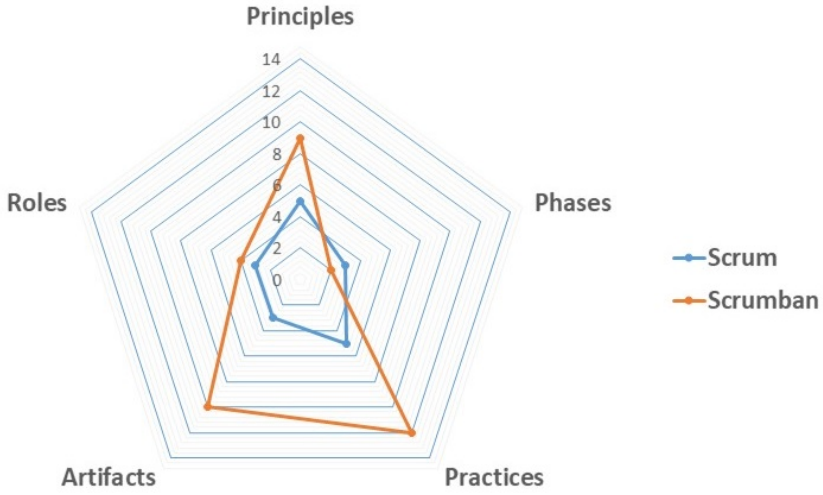

Fig. 3.10 Dimensional Comparison of Scrum and Scrumban.

Table 3.6 Scrum and Scrumban Methodological Dimensions [Banijamali et al., 2017] [Reddy, 2016].

Dimension	Scrum	Scrumban
Principles	Focus, Courage, Openness, Commitment, Respect	Transparency, Balance, Understanding, Flow, Customer Focus, Agreement, Respect, Leadership, Collaboration
Roles	Product Owner, Scrum Master, Development Team	Product Owner, Scrumban sensei, Development Team, Stakeholders
Artifacts	Product Backlog, Sprint Backlog, Iteration	Product Backlog, Selected Backlog, Story in Progress (SIP) Backlog, Task Backlog, User Story, Task, Work in Progress (WIP) Limit, Parking Lot, Cumulative Flow Diagram (CFD), Impediment Backlog
Practices	Sprint, Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective	Planning 1, Planning 2, Daily, Review, Retrospective, Andon, Pull work, Backlog Limit, Plan On Demand, Visualization of the workflow (Extend Board), Bucket size planning, Metrics performance
Phases	PreGame, Development, PostGame	Specification, Production

Figure 3.10 illustrates the compared methodologies on the basis of the adopted five methodological dimensions. Figure 3.10 and Table 3.6 indicate that there are simultaneously substantial similarities and differences in term of the methodological dimensions. In addition, this figure and table highlight that Scrumban has more principles and artifacts than Scrum. This could be explained by the fact that Scrumban combines the best features of both Scrum and Kanban. Moreover, [Gambill, 2013], [Mahnic, 2014], [Misevičiūtė, 2016], and [Reddy, 2016] have performed deep analysis of the nature of Scrum and Scrumban. The main differences that characterize Scrum and Scrumban are presented in the Table 3.7.

Table 3.7 Differences between Scrum and Scrumban, adapted from [Gambill, 2013], [Mahnic, 2014], [Misevičiūtė, 2016], [Reddy, 2016].

#	Criteria	Scrum	Scrumban
1	*Iterations*	1-4 week sprints	Continuous work with short cycles for planning and longer cycles for release
2	*Work routines*	Push and pull principle mixed with early binding to team members	Pull principle with late binding to team members
3	*Scope limits*	Sprint limits total work amount	Work in progress limits current work amount
4	*Planning routines*	Sprint planning	Planning on demand for new tasks
5	*Estimation*	Must be done before start of sprint	Optional
6	*Performance metrics*	Burndown	Velocity is optional, use lead time and cycle time as default metrics for planning and process improvement, Cumulative Flow Diagram is optional
7	*Continuous improvement*	Sprint retrospective	Short Kaizen event as an option
8	*Meetings*	Sprint planning, daily scrum, retrospective	Planning1, planning2, daily, review, retrospective, Andon – (Short Kaizen event)
9	*Roles*	Product owner, Scrum master, team	Product owner, Scrumban sensei, development team, stakeholders
10	*Team members*	Cross-functional team members	Specialization or preference to tasks
11	*Task size*	The size that can be completed in sprint	Any size
12	*New items in iteration*	Forbidden	Allowed whenever queue allows it
13	*Ownership*	Owned by a team	Supports multiple teams ownership
14	*Board*	Defined/reset each sprint	Persistent
15	*Prioritization*	Through backlog	Recommended on each planning
16	*Rules*	Constrained process	Slightly constrained process
17	*Fit for*	Enterprise maturity for teams working on product or especially project which is longer than a year	Startups, fast-pace projects, continuous product manufacturing

3.7 Conclusion

Current software companies tend to establish their production entities in different agile ways in order to be flexible and optimize skilled workforces to produce higher quality products and

lower cost. This chapter has introduced state of the art agile methods that concern the scope and aims of the current thesis, namely Scrum and Scrumban.

Agile thinking is a people-centric view to software development. Agile methods are a set of methodologies that follow the four values and twelve principles of the *agile manifesto*. Rather than just being a methodology on its own, the agile manifesto guides the agile methods. It has been shown that agile methods are generally characterized as iterative, incremental, less documented and people-oriented.

The Scrum method is introduced as a project management framework applicable to any project with aggressive deadlines, complex requirements and a degree of uniqueness. Scrumban can give teams the power to adapt and change to stakeholder and production needs, without feeling overburdened by their project methodology. However, the use of Scrum and/or Scrumban in software development presents challenges to the organizations that adopt agile methods. It is challenging to select one method over the other. When selecting Scrum, Scrumban, or the both methods, a range of criteria has to be considered. Thus, an in-depth analysis should be conducted in order to make the right decision when forming a Scrum and/or Scrumban method.