5 Requirements Engineering for Agile methods and Software Product Lines

Abstract. *Requirements Engineering (RE)* is defined by [van Lamsweerde, 2009] as *"a coordinated set of activities for exploring, evaluating, documenting, consolidating, revising and adopting the objectives, capabilities, qualities, constraints and assumptions that the system-to-be should meet based on problems raised by the system-as-is and opportunities provided by new technologies"*. This chapter gives an overview of the different aspects of the RE discipline. In addition, it reviews how Software Product Lines and agile methods deal with the Requirements Engineering issues. Further, it presents the main RE practices and techniques that have been used to propose an integrated RE framework for Agile Product Lines presented in Chapter 8.

5.1 A brief outset

In the late nineties, Dorfman and Thayer highlights that "*the inability to produce complete, correct, and unambiguous software requirements is still the major cause of software* [project] *failure*" [Dorfman and Thayer, 1997]. Nowadays, identifying and specifying the requirements play a crucial role in any successful software development project. The adequate and precise specification of key system requirements helps to avoid considerable risks of development. In addition, when rectifying misunderstandings of requirements later in the software development process can results major additional costs and delays the delivery date [Boehm, 1976].

Important studies have highlighted in their findings that the origin of a high percentages of the errors and failures in software products can actually be traced back to the requirements phase. Sheldon et al. [Sheldon et al., 1992] studied the software development life cycle (SDLC) of "US Air Force" systems and found that overall 41% of errors could be traced back to requirements. In addition, Hall et al. found that 48% of all problems experienced in software development projects in 12 software companies in different domains were requirements-related [Hall et al., 2002]. Moreover, Boehm and Basili reported that finding and fixing a software

problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase [Boehm, and Basili, 2001]. Consequently, requirements engineering (RE) must be a major concern in any software development project. Therefore, ignoring or diminishing the role of requirements activity would be irresponsible and makes software development projects likely to result in low quality and increased cost. However, exaggerating too much when performing requirements activities can be dangerous and lead to delay the project and increase the budget [Davis, 2005]. Therefore, a good and clear trade-off needs to be found with the objective of performing the adequate amount of requirements engineering for every software project.

Requirements engineering (RE) emphasizes the use of systematic and repeatable techniques that ensure the completeness, consistency, and relevance of the system requirements [Sommerville, and Sawyer, 1997]. According to Pohl [Pohl, 2010], RE is the discipline to elicit, analyze, specify and document, verify and validate, and manage the stakeholder's requirements for a planned system where:

- *Requirements elicitation* is the process of discovering, reviewing, and understanding the stakeholder's needs, intentions, and constraints for the system.
- *Requirements analysis* is the process of refining the stakeholder's needs, intentions, and constraints.
- *Requirements specification and documentation* is the process of documenting the stakeholder's needs, intentions, and constraints clearly and precisely.
- *Requirements verification and validation* is the process of ensuring that the system requirements are complete, correct, consistent, and clear.
- *Requirements management* is the process of scheduling, coordinating, and documenting the requirements engineering activities (i.e. the stakeholder's needs, intentions, and constraints).

For several reasons, software companies find themselves putting more effort into customizing existing software systems, instead of developing *new-to-the-world* software products. In fact, reusing existing software (i.e. software reuse) has become of paramount importance to ensure an efficient development and high-quality software. Omitting the reuse of identical and already previously developed functionalities will inevitably lead to unnecessary additional costs and to lower software-quality. Software reuse has demonstrated its effectiveness and has enhanced the quality of software products, because reused components were previously quality-assured and many bugs are already known or fixed.

As stated earlier in this thesis, several methods have been proposed as APLE methodologies. Among these methods, only those that propose a concrete *process* have been considered in the present work. Chen and Babar [Chen and Babar, 2011] argued that a software development methodology consists mainly of two integral parts:

1. *Modeling Language* that provides the syntax and semantics used for expressing the products. This part represents some practices and activities of the requirements engineering.

2. *Process* that prescribes the flow of activities that should be performed and explains how the products should be produced enhanced and exchanged along this flow. Thus, this part represents essentially the software development process.

After surveying and studying some significant APLE methodologies (see Chapter 4), it was shown that the agility attribute of APLE methodologies has deemphasized the role of modeling, and hence the modeling language. In this chapter, the current state of research of requirements engineering (RE) practices and techniques used in agile product lines (APLs) are reviewed and synthesized with respect to their degree of empirical validation, origin and context of use, level of expressiveness, maturity, and industry adoption.

5.2 Requirements Engineering disciplines

According to Wiegers and Beatty [Wiegers and Beatty, 2013], Requirements Engineering (RE) is actually divided into two mains areas, namely, *requirements development* and *requirements management*. Requirements development is mainly, *identifying*, *analyzing*, *agreeing upon*, and *recording requirements*. The purpose of Requirements Management is simply to manage changes in requirements.

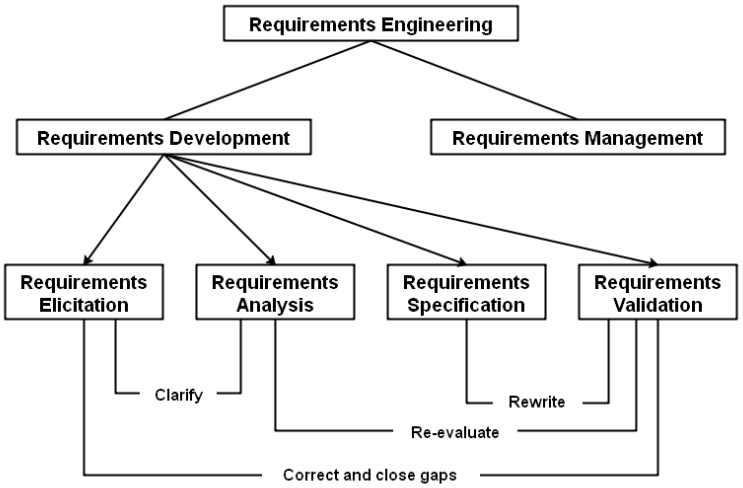

Fig. 5.1 Requirements Engineering (RE) areas - adapted from [Wiegers, 2005].

As shown in Figure 5.1, the RE discipline is structured in four main activities. These activities are briefly presented in this section and are the following:

1. Requirements elicitation;
2. Requirements analysis;
3. Requirements specification;
4. Requirements validation.

5.2.1 Requirements Elicitation

Requirements Elicitation activity is concerned with understanding the application domain, services the system should provide, system performance, hardware constraints [Sommerville, 2010]. According to Nuseibeh and Easterbrook [Nuseibeh and Easterbrook, 2000], the clear understanding of the needs of stakeholders, the system constraints as well as the limitations and deficiencies of the *system-as-is* are of critical importance. Information that has been gathered during this activity often has to be *interpreted*, *analyzed*, *modeled* and *validated*. Thus, this is the reason why this activity of eliciting requirements is often related to the other activities of the RE process [108]. In practice, there are many techniques that can be used for eliciting requirements from stakeholders, such as *interviews*, *Use-Cases*, *scenarios*, *user-task elicitation*, *user stories*, *observation and social analysis*, *focus groups*, *brainstorming sessions*, and *prototyping*.

5.2.2 Requirements Analysis

The Requirements Analysis activity is dedicated for analyzing and modeling the requirements gathered in the elicitation activity. It consists of checking and analyzing requirements for necessity, consistency, completeness and feasibility. In addition, during this activity, conflicting, overlapping and omitted requirements are identified. In fact, the conflicting concerns of stakeholders that generate inconsistent requirement specifications are solved through prioritization. Through dialogue with customers, a priority is attributed to the requirements. An analysis of both risk and impact of requirements is also included in this activity [Elshandidy and Mazen, 2013].

Within Requirements Analysis activity, modeling tasks are also performed. Models make requirements easier to understand, because they imply the possibility of collecting, processing, organizing and analyzing smaller amounts of relevant information. There are many techniques, modeling notations and languages. The most popular ones are Unified Modeling Language (UML) [OMG, 2017], Specification and Description Language (SDL) [Spivey, 1988], Structured Analysis Structured Design (SASD) [Yourdon, 1989], Goal-based techniques (e.g. [van Lamsweerde and Letier, 2004]) etc.

5.2.3 Requirements Specification

According to Wiegers and Beatty [Wiegers and Beatty, 2013], Requirements Specification is aimed at communicating the requirements between stakeholders and developers by means of the requirements document, which directly results from activities performed within the different processes in the RE cycle. In this activity, requirements and features of the *system-to-be*, which are the result of the previous activities, are detailed, structured and documented. The resulting requirements documents are used in different activities of software development, such as “project work-plan”, “project estimation”, “Software Prototype Mockup”, “Call for tenders, proposal evaluation”, etc.

Generally, requirements could be documented in three forms, that are, informal, semi-formal and formal [Pohl and Rupp, 2011]

- *The informal form* consists of using natural language or ad-hoc diagrams for documenting requirements.
- *The formal form* consists of using a rigorous mathematical basis or well-defined graphical notation with well-defined semantic for documenting requirements.
- *The semi-formal* consists of using natural language augmented with some (graphical) notations in formal approach.

5.2.4 Requirements Validation

Requirements validation aims at validating and verifying requirements. It ensures that all gathered, modeled and documented requirements actually fulfill the need of all different stakeholders in an accurate and complete way. This activity is performed by using the requirements document, organizational standards and organizational knowledge as input [Paetsch et al., 2003]. Bourque and Fairley [Bourque and Fairley, 2014] have mentioned that the techniques used for requirements validation are requirements reviews, prototyping, model validation and acceptance tests.

5.2.5 Requirements Management

Wiegers and Beatty [Wiegers and Beatty, 2013], *Requirement Management* is a complete and concrete process. It comprises all activities that are associated with *change control*, *version control*, *requirements tracing*, and *requirements status tracking*. Traceability is a technique used to keep track of the relationships between requirements, design, and implementation of a system in order to manage changes.

5.3 Requirements Categories

Sommerville [Sommerville, 2010] and van Lamsweerde [van Lamsweerde, 2009] have argued that there are three kinds of requirements:

1. Functional Requirements;
2. Nonfunctional Requirements;
3. Quality Requirements.

5.3.1 Functional Requirements

Functional requirements describe, first, the services a system should provide, second, how the system should react to particular inputs, and, third, how the system should behave in particular

situations [Sommerville, 2010]. They describe what the developers must implement to enable users to accomplish their tasks. In addition, functional requirements can be high level (e.g., Top management) and general (user requirements) or they can be detailed, expressing inputs, outputs, exceptions, and so on (system requirements) [Laplante, 2013].

5.3.2 Nonfunctional Requirements

In literature, there are various definitions for non-functional requirements. Sommerville defines nonfunctional requirements as *"Constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services"* [Sommerville, 2010]. In addition, van Lamsweerde [van Lamsweerde, 2009] defines nonfunctional requirements as *"constraints on the way the software-to-be should satisfy its functional requirements or on the way it should be developed"*. The difference between functional and non-functional requirements is that a functional requirement is completely satisfied and non-functional requirements are only satisfied up until a certain level.

5.3.3 Quality Requirements

van Lamsweerde [van Lamsweerde, 2009] has considered that quality requirements complement the *"what aspects"* with *"how well aspects"* of requirements. He defines quality requirements as *"additional, quality-related properties that the functional effects of the software should have"*. Quality requirements are previously known as 'quality attributes' of nonfunctional requirements.

5.4 Requirements engineering for Software Product Lines

The requirements engineering (RE) for product lines engineering differs from the requirements engineering for a single custom-built system. The RE for a family of software-intensive systems focuses more on systematic reuse, not only from the technical perspectives, but from the organizational, marketing, and process perspective as well [Bosch, 2000]. In particular, the following aspects must be considered in the RE for Software Product Lines (SPLs) [Alves et al., 2010] [Coplien et al., 1998]:

- *Scoping, commonality, variability (SCV) analysis*. This analysis aims to explicate the definition of a domain and the inclusion and exclusion criteria of the domain. Identifying and managing the common and variable requirements are key to the product line (PL).

- *Asset management* is unique to product line engineering (PLE). In order to manage the assets, the designated personnel of the product line (e.g. domain engineers) have to design, build, and evolve the reusable set of core assets, so that the application engineers can effectively reuse the asset base to derive customized individual products.

- *Stakeholders of the product lines*. The stakeholders include not only a single application's customers, users, developers, testers, maintainers, etc., but also the parties who are involved in asset development and management.

- *Marketing* plays an important role in launching a product line (PL) or transitioning from single-system development to product line engineering, so factors such as "reuse ratio (RR)" and "return on investment (ROI)" need to be considered when planning the product line requirements.

- *Organizational and process changes* are expected in product line engineering [Krueger, 2001]. These changes could be the result of whether the core assets are developed in a centralized or distributed manner, and whether the organization chooses proactive, reactive, or extractive adoption strategy.

- *The RE techniques*, most notably the modeling techniques, are different from single-system RE. Single-system requirements are often modeled from the *use perspective* (e.g. use case, sequence diagrams, etc.). However, PL requirements are modeled from the *reuse perspective* by explicitly by explicitly representing the commonality and variability information (e.g., feature models, orthogonal variability models, etc.).

Several critical decisions are made during the requirements engineering phase for developing Software Product Lines (PLs). In fact, after the definition of the PLs' scope, the *"Domain Engineer (DE)"* specifies the common requirements that are shared among family members and the optional ones that are unique to a specific product. Furthermore, the product line requirements are non-trivial, because:

- They reflect stakeholders' diverse perspectives;
- They have complex configuration dependencies (e.g. requires, excludes, …);
- They are expressed in various forms (e.g. textual, goals, …);
- They are expressed at different granularities (e.g. features, qualities …).

One of the key concepts in PLE is variability, which ensures the required flexibility for products (Apps) differentiation and diversification [Chen and Babar, 2011]. *Variability* is defined as the ability of product lines (PLs) to be exchanged, customized, configured or extended to be of use in a specific context [Sinnema and Deelstra, 2008]. *Variability management* is an aspect that distinguishes PLs from other software development approaches in the sense that it involves extremely complex and challenging tasks that must be supported by appropriate methods, techniques, and tools [Sinnema and Deelstra, 2008]. Up until now, several representations of variability, together with mechanisms that facilitate its systematic exploitation, have been reported in the literature, and some efforts have been made to unify the concepts and relations of the field of Product Lines [Bachmann et al., 2004].

In order to model such variability, methods can be broadly categorized into two groups. First, the feature-based group which comprises methods that model the common and variable

features of a product family (an analysis-oriented perspective). Second, the architecture-based group, which includes methods that describe variability in terms of components, interfaces, connectors and so on (a more design-oriented perspective) [Asikainen et al., 2007]. At the requirement engineering (RE), stage the focus is on the first group, that is, languages based on the modeling of features [Sepúlveda et al., 2016]. In fact, in domain engineering, the "*feature modeling*" approach is used to understand the target domain and develop reusable artifacts [Czarnecki et al., 2012].

Classen et al. [Classen et al., 2008] mention that in the requirements engineering (RE) context, the term *"feature"* is a problem-oriented concept that refers to '*a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems*'. Most of the complexity of feature languages resides in the interoperation of features. In fact, some features depend on each other, while others are mutually disruptive [Classen et al., 2008]. The popular way to express and model such interoperation is by means of a feature model (FM) [Asikainen et al., 2006]. A feature model (FM) has a tree-like structure, with its root nodes representing the software product family. The characteristics of the family are organized down the tree. The feature model (FM) represents individual or configured components that can be assembled to generate a specific application [Czarnecki and Wasowski, 2007]. At the beginning, the feature modeling was presented as a part of the Feature-oriented domain analysis (FODA) method [Kang et al., 1990], although this feature model is now present, with slight variations, in virtually all the product-line methods that rely on a visual representation of product characteristics [Sepúlveda et al., 2016].

Table 5.1 Requirement Engineering practices - Modeling languages for Software Product Lines.

#	RE technics / modeling language	e.g. of Reference
1	*Feature Model (FM)*	[Aple et al., 2013] [Kang et al., 2002]
2	*UML diagrams (Class Diagram (CD), Use-Case Diagram (UCD), Activity Diagram (AD), Sequence Diagram (SD))*	[Gomaa and Gianturco, 2002]
3	*Decision Model (DM)*	[Ajila and Kaba, 2008]
4	*Architecture Diagram*	[Arcega et al., 2016]
5	*Evolution Feature Model*	[Pleuss et al., 2012]
6	*Product Model (as-is, to-be)*	[Weyns et al., 2011]
7	*Component Model*	[Jahn et al., 2012]
8	*Configuration Model*	[Ferreira et al., 2012]
9	*Hyper Feature Model*	[Seidl et al., 2014]
10	*Mappings*	[Demuth et al., 2014]
11	*Model Fragments*	[Dhungana et al., 2010]
12	*Temporal Feature Model*	[Nieke et al., 2017]
13	*Orthogonal Variation Management (OVM)*	[Lee and Hwang, 2016]
14	*xDSL – variations of Domain Specific Language (DSL)*	[Spinellis, 2001] [Fowler, 2010]
15	*Ontology*	[Asikainen et al., 2007]
16	*Goal-Oriented Modeling*	[Asadi et al., 2014] [Asadi et al., 2011]
17	*Common Variability Language (CVL)*	[Font et al., 2017]

Researchers and practitioners have proposed several research works that deal with the requirements engineering issues in the context of Software Product Lines. Hereafter, we present the most important works and summarizes the modeling approaches proposed for Software Product Lines.

America et al. [America et al., 2005] have proposed a modeling language that includes the *Orthogonal Variation Management (OVM)* approach to deal with variability at architectural level for SPLs and takes into consideration the functional and quality properties. Asadi et al. [Asadi et al., 2011] [Asadi et al., 2014] have used goal modeling and mappings to FM in order to help application engineers to communicate with stakeholders and abstract them away from realization details. Sinnema et al. [Sinnema et al., 2004] have proposed a framework for modeling variability in software product families called COVAMOF. This framework allows to model variation points, hierarchical organization of variability and the first-class representation of dependencies.

Asikainen et al. [Asikainen et al., 2007] have defined a domain ontology called Kumbang for modeling variability in SPL. Park et al. [Park et al., 2011] have proposed a variability modeling approach to specify and control the common and distinguishing characteristics of service-oriented applications. Behjati et al. [Behjati et al., 2013] have introduced a specific approach called SimPL. The SimPL methodology provides a notation and a set of guidelines for modeling commonalities and variabilities in Integrated Control Systems (ICS) families. Abo Zaid et al. [Abo Zaid et al., 2011] have presented an approach called "Feature Assembly" to create Feature Model (FM); it is based on creating FM by composing and reusing features. Zhou [Zhou, 2010] has presented an approach to pair up goals with features; this can help to identify the traceability from a domain model down to requirement specifications. Gomaa and Gianturco [Gomaa and Gianturco, 2002] have proposed a Domain Modeling approach using UML for SPLs of WWW apps. It describes how the views of UML may be used for modelling such SPL. Zhang et al. [Zhang et al., 2004] have proposed a XVCL-based approach to SPL development. It performs domain analysis, capturing common and variant requirements for a SPL in a FM. Zhang et al. [Zhang et al., 2011] have proposed a Common Variability Language (CVL) Compare approach that provides a methodology to synthesize an SPL from existing models using the CVL Compare tool.

Table 5.1 presents the requirements engineering techniques and modeling languages used for specifying the requirements in the context of Software Product Line Engineering (SPLE). These techniques and practices as well as modeling languages could be combined, aligned and extended.

5.4 Agile Requirements Engineering

As stated in this chapter, the Requirements Engineering (RE) is a crucial discipline for any software development methodology. In general, RE and agile methods are often seen as being incompatible due to the fact the RE is frequently conceived as a phase heavily relying on documentation for sharing knowledge [Paetsch et al., 2003]. However, Wang et al. [Wang et al., 2014] have revealed that RE is critical for project success in agile methods.

Agile methods are usually applied in an environment where requirements are unstable or unknown and change is the norm. In such situations, it is hard to build high-quality requirements documents as recommended by RE. This situation has led to some agile RE practices. Table 5.2 presents the artifacts used for agile Requirements Engineering. Note that artifacts are used for communication, elaboration, validation and documentation of the requirements in agile environment.

Table 5.2 Artifacts used in Agile Requirements Engineering.

#	Artifact	Description	Reference
1	User Story	User story is a description of a feature written from the perspective of the person who needs this. It consists of a written text, conversation about it and acceptance criteria.	[Cohn, 2004]
2	Prototype	Prototype is a model of the software application that supports the evaluation of design alternatives and communication.	[Bellucci et al., 2015]
3	Use Case	Use case describes an action or event steps, which are needed to achieve a goal.	[Cockburn, 2000] [Wautelet et al., 2014]
4	Scenario	Scenario is a textual representation of a problem and describes the interaction between user and system in a specific context.	[Obendorf and Finck, 2008]
5	Story Card	Story Card is a physical representation for the written text and details from a user story.	[Kautz, 2010]
6	Persona	Persona is a description of a fictitious person that represents a larger part of the target group.	[Liskin et al., 2014]
7	Vision	Vision is an abstract description of the overarching goal that guides product development and aligns development, business people and other stakeholder.	[Buchan, 2014]
8	UML diagram	Unified Modeling Language (UML) provides a standard to visualize the design of a system.	[Lee et al., 2010]
9	Storyboard	With a storyboard, the workflow of the user is presented in a sequence of pictures.	[Näkki et al., 2011]
10	Task	One user story is split into more tasks, which describe requirements that are more technical.	[Cohn, 2004], etc.
11	Kanban board	Kanban board visualizes the progress of a requirement through the workflow of the development	[Anderson, 2010]
12	UI pattern	UI pattern describes an abstract solution for recurring design problems and give inspiration to designer.	[Memmel et al., 2007]
13	Essential use cases	"Essential Use-Case" describes user tasks and is a simplified and generalized form of use cases.	[Lee et al., 2010]
14	Pictures	Picture is a visual representation (e.g. photograph, painting)	[Bellucci et al., 2015]
15	Mind Map	Mind map is a diagram used to visualize and organize information.	[Wanderley et al., 2014]
16	UI specification	Written specification that describes the UI of a system. The text is enriched by mock-ups, icons, etc.	[Maguire, 2013]

The User Story (US) is the most used requirements artifacts in agile methods. User stories are more than only small pieces of functionality represented on story cards, they form the fundamental basis of communication between the development team and the customers [Cohn, 2004]. They are gathered at the beginning of the project, at the beginning of a release or at the beginning of an iteration. In addition, they are used as guidance within several conversations and discussions between developers and customers or users. A US driven

approach starts with gathering the different US and writing them on so-called story cards. The USs of the project backlog are considered as the scope elements of the coming iteration. At the end of each iteration agile practitioners proceed to a re-prioritization of requirements, as well as a re-evaluation of the criteria used in the prioritization.

Scrum method is widely used by the 'agile community' and was tailored to for the Application Engineering process of our Agile Product Line method presented in Chapter 6. That is why in this section, the mapping of Scrum's practices with the Requirements Engineering (RE) activities is presented. Table 5.3 presents this mapping.

Table 5.3 Requirements engineering implementation in Scrum - adapted from [Lucia and Qusef, 2010].

RE activity	Scrum's practices
Requirements Elicitation	– Product Owner formulates the Product Backlog; – Any stakeholder can participate in the Product Backlog.
Requirements Analysis	– Backlog Refinement Meeting; – Product Owner prioritizes the Product Backlog; – Product Owner analyzes the feasibility of requirements.
Requirements Specification	– Face-to-face communication.
Requirements Validation	– Review meetings.
Requirements Management	– Sprint Planning Meeting; – Items in Product Backlog for tracking; – Change requirements are added/deleted to/from Product Backlog.

5.5 Goal Models (GM) and Feature Models (FM)

Feature Modeling appears as a crucial technique to identify *commonalities* and *variabilities* of SPLs. However, Feature Models show only a specific perspective, which is not sufficient to express all the characteristics and constraints of an SPL [António et al., 2009].

Using a goal-oriented approach, such as iStar 2.0 [Dalpiaz et al., 2016], to complement (and help define) feature models would improve such models by enhancing the meaning and justification to features. Goal-oriented modeling provides a way to identify variabilities at an early phase of requirements, allowing alternative options to satisfy stakeholder's goals. Table 5.4 presents the distinctions between goal models and feature models. In addition, a comparison of variability types in both models are outlined.

Table 5.4 Comparison of Goal model (GM) and Feature model.

GM	FM
An artifact representing stakeholders' objectives and strategies. It describes the intentional space of a domain stakeholders	Describes the configuration space of a product line. Accordingly, it represents characteristics of software products that belong to a product line
Different terms are defined for goals: achieve, maintain, avoid, and cease	Features can only be selected or deselected
An intentional decomposition (except for soft goals) in a goal model is an entailment, i.e., an *AND-refinement* of source intentional elements is one possible combination (among others) of source intentional elements that implies the target intentional element	An *AND-decomposition* in a feature model is a *PartOf* relation that implies that a parent feature contains its child features
Behavioral variability represents the various behaviors of a single system that may be used by its user	*Product line variability* refers to differences between products in a product line, which may exist among their requirements, design models and implementation models
OR-refinement represent intentional/behavioral variability	*OR decompositions* represent product line variability

5.5.1 Variabilities

In goal models, OR relations represent variability in the stakeholders' goals (i.e., intentional variability), and the ways these goals can be achieved (i.e., behavioral variability) [Asadi et al., 2016]. When developing a reference design and implementation model from the family goal model, these variability relations can lead to either *intentional/behavioral variability* or *product line variability* in the reference models. Thus, there are two cases, which are the following:

- **First case:** *OR-refinement represents intentional/behavioral* variability: OR-refinement represent intentional/behavioral variability; if all the products, having a target intentional element involved in the OR relation, contain all source intentional elements of the OR-refinement in their goal models.

- **Second case:** *OR-refinement represents product line* variability: OR-refinement represents product line variability if source intentional elements involved in OR-refinement vary between different products that contain the target intentional element.

At this stage, it should be noted that feature models not only show variabilities in requirements, but also encapsulate product line variabilities in designing and implementing models [Kang et al., 1990]. Therefore, a feature model may contain several variability relations that do not exist in the goal model.

Consequently, investigation of the notion of variability in the goal and feature models disclose the differences in their semantics of variability. *Variability relations* in goal models can be either *product line* variability or *intentional/behavioral* variability while feature models only represent the *product line* variability.

5.5.2 Mapping between intentional elements and features

For Domain Engineering process, goal models could capture intentional variability [Liaskos et al., 2006] [Yu et al., 2008] and describe the intentions behind existing features in the Software Product Line. Hence, using the goal model, engineers can ensure that existing features and variability relations in feature models are aligned with intentional variability in the goal models. They can also trace back differences in products to differences in the intentions of the stakeholders [Asadi et al., 2016].

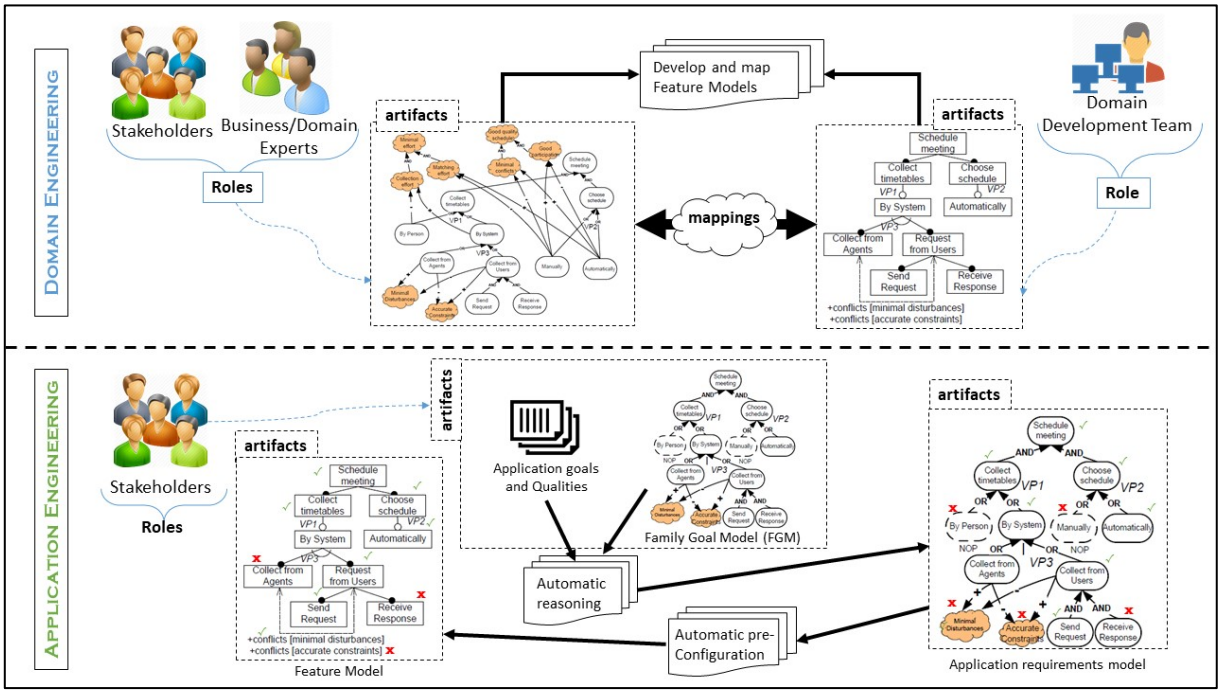

Fig. 5.2 Approach of mapping between intentional elements and features - Adapted from [Asadi et al., 2016].

According to the model proposed by [Asadi et al., 2016], in order to represent an *explicit mapping* (i.e., a mapping indicated by domain engineers), a mapping relation for each mapped task is developed (See Figure 9). In addition, if a *feature* is mapped to more than one *goal or/and task*, then the corresponding feature appears in the mapping relations of all those goals or/and tasks. After explicit mapping between tasks and features in a *feature model*, Development Team can drive implicit mappings between intermediate tasks and goals and features through existing relations in goal models and feature models. This chapter adopts the following definition of *mapping:*

Definition 5.1:

Let $FGM = \{\mathcal{G}, \mathcal{C}, \mathcal{D}^F\}$ be a Family Goal Model where $\mathcal{G} = (\mathcal{G}_g \cup \mathcal{G}_t \cup \mathcal{G}_s)$ and $FM = \{\mathcal{F}, \mathcal{F}_M, \mathcal{F}_O, \mathcal{F}_{IOR}, \mathcal{F}_{XOR}, \mathcal{F}_{incl}, \mathcal{F}_{excl}\}$ a Feature Model, $\boldsymbol{\Phi_i}$ $(\mathcal{G}_i, \mathcal{F}_i)$ is a mapping relation between an intentional element $\mathcal{G}_i \in (\mathcal{G}_g \cup \mathcal{G}_t)$ and a set of features $\mathcal{F}_i \subseteq \mathcal{F}$, and Φ denotes the set of all mappings between FGM and FM. (adapted from [Asadi et al., 2016])

5.6 Chapter Summary

This chapter briefly presented the state of the art of Requirements Engineering (RE) in both Software Product Lines and Agile Methods. It specifically focuses on the main RE techniques and activities that may help to propose a RE approach for Agile Product Lines.

It was concluded, from Chapter 5, that the current Agile Product Line methodologies, reviewed in this thesis, have not proposed complete RE approach that complement their APL processes. However, some of the reviewed APL methods have proposed some RE techniques without going beyond to propose how they could be used in practice. Others have neglected the RE part.

This chapter identifies the RE practices and modeling languages used by classical SPLE approaches to develop and manage requirements. In addition, it describes the main artifacts used in agile RE. Finally, it provides a clear explanation about the main practices and techniques that have been used to build the RE approach proposed in Chapter 8. In fact, it has made an overview about the feature modeling, goal modeling, and user story artifact. Furthermore, it has presented how goal models could help generating feature models.

Part III – An Agile Product Line Method

6 Toward an Agile Framework for managing Software Product Lines

Abstract. *The AgiFPL Method* is an APL method, called Agile Framework, for managing evolving Software Product Lines. 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. Since none of the reviewed work has presented an explicit definition of APLE, this chapter has first defined explicitly the concept "Agile Product Line Engineering – APLE". Moreover, this chapter has introduced the details about AgiFPL (i.e. the framework, different processes, defined roles, tailored activities, practices and artifacts, etc.). AgiFPL has presented a Scrumban-inspired process for the Domain Engineering (DE) tier and a Scrum-inspired process for the Application Engineering (AE) tier. Finally, this chapter has introduced a concrete approach to implement AgiFPL in practice, which is a 3-phase approach.

6.1 Introduction

As concluded from the literature review (Part I), there are several reasons that encourage companies and organizations to move toward the adoption of Agile Product Line methods. Thus, many questions might be asked before deciding which APL approach to adopt. This chapter addresses the core objective of this thesis directly, which is *"Proposing an Agile Product Line method"*. The main question that this chapter has tackled is the following:

How can agile practices be combined with Software Product Lines, and an APL method built which takes the strengths of the previous APL methods and overcomes their weaknesses?

Defining a new APL method is essentially 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 for expressing the products. This part represents some practices and activities of the requirements engineering;
2. *Process;* that prescribes the flow of activities that should be performed and explains how the products should be produced, enhanced and exchanged along this flow. Thus, this part in essence represents the software development process.

To reach the target, a development process should be designed and provide modeling modules and rules. However, before designing the APL process, it is preliminary to address, in this chapter, the issue of the term *"agile/agility"* of APLE and to propose a *definition* for "APL or APLE". Thereafter, the process, and its related complements, is designed following the appropriate research method.

6.2 Research Method

Figure 6.1 illustrates the research process followed in this chapter. First, the results of the conducted systematic literature review (SLR) and the criteria-based evaluation (CBE) were analyzed. In fact, in this first step, a cross-analysis of the advantages, findings and challenges was made. In addition, the studied APL methods are compared by filtering their strengths and weaknesses, and identifying their applicability. 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 selected SPL is selected. After that, the APL process was designed. Third, the designed process was 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.

Note that the review of the process was not realized through quantitative analysis approach; in contrary, the base of the review activity is similar to this done by conference reviewing commities and journal papers reviewers. In addition, the final optimized version of the designed APL process is obtained after the validation through the case study presented later in this thesis. In this chapter, the final version of the APL process is presented.

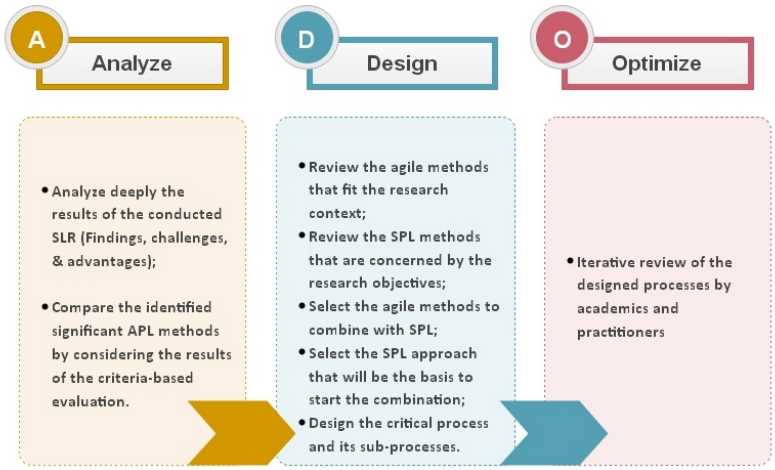

Fig. 6.1 Followed research process

6.3 A definition for “Agile Software Product Line Engineering (APLE)”

The definition of the APLE concept is presented in this chapter and not in the “Part I” for two main reasons. First, the presented definition is our own definition and highlights what is considered as APLE in this research project. Second, since “Part 2” presents the main contributions of this dissertation, thus we decided to join this definition to this chapter because it is considered as a related contribution.

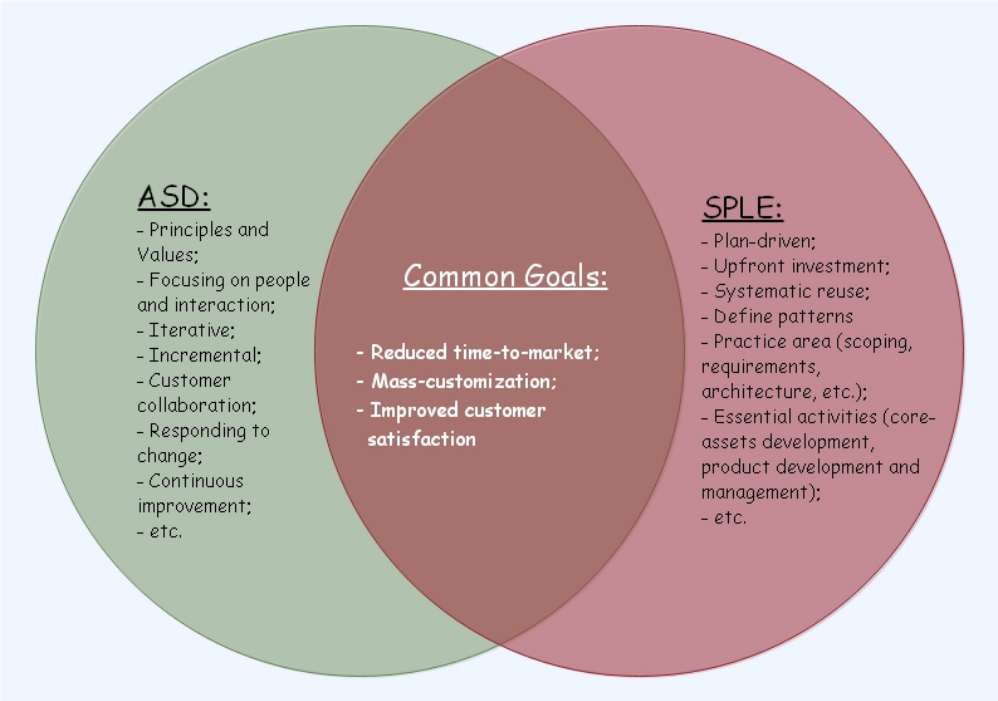

Fig. 6.2 Venn diagram that represents the characteristics of SPLE and ASD.

Agile methods do not address essential concepts for SPL methods such as reuse and variability management. SPL methods do not focus on agile values or principles. However, both approaches share common goals, such as reduced time-to-market. Therefore, both approaches could be combined to address common business goals. In other words, agile methods may address many of the problematic SPL issues (e.g. unstable domain, insufficient resources, etc.) through incremental and evolutionary development. An Agile SPL approach can handle scenarios unfavorable to systematic reuse in a more appropriate way. Further, on the one hand, an Agile SPL method could be built by starting with a SPL process and then integrate agile activities and practices in order to align the designed method with agile principles at a maximum degree. On the other hand, an Agile SPL method could also be built by starting with an agile method and then integrate progressively SPL principles. Figure 6.2, which is a Venn diagram, shows the common goals of SPLE and ASD as well as their differences.

According to these characteristics and structures of both Software Product Line Engineering and Agile Software Development, the following definition was proposed as a definition for what is called Agile Software Product Line Engineering (APLE):

Definition 6.1:

Agile Product Line Engineering is a paradigm that maximizes the benefits of common objectives of both approaches ASD and SPLE, by combining agile practices and activities to a Software Product Line approach or by integrating SPL principles to an agile approach, while addressing the weaknesses of each approach. Where any combination should ensure at least systematic reuse, iterative and incremental development, stakeholders' involvement, and high responsiveness to changes.

This definition focuses on the fact that any combination of SPLE and ASD, first, should ensure a maximization of the performance of the business infrastructure, and second, should be realizable in practice (not utopic). By this definition, a point of view is proposed in which, an organization becomes agile while preserving its Software Product Lines or an agile organization boost its agility by integrating SPL principles. Therefore, *an organization needs to know when it has become "agile".* The literature on APLE does not define any threshold (point) after which an organization has adopted enough practices to be called *"AGILE".* This issue will be tackled in next chapter of this thesis (i.e. Chapter 7).

6.4 Designing an Agile Software Product Line method

When designing a new APL process, first, the basis of the new process should be selected (i.e. start with SPL or ASD). Second, if the basic process is a SPL process, thus the type of the SPL architecture should be selected (i.e. COPA, KobrA, FORM, etc.). Third, an agile method or a hybrid of agile methods should be selected for the combination.

6.4.1 Basis of the APL process and its type of architecture

Nowadays, most of the companies that need to adopt APL methods are companies that have basic SPL structures and want to become "agile". In addition, most of the reviewed APL methods were proposed by starting with SPL processes as the basic process and move toward integrating agile practices and activities (see *Table 4.24*). These two main reasons are sufficient to encourage us to propose an APL method by starting with a SPL process as a basis to design the targeted APL process.

As shown in Chapter 2 (Section 2.4), most of SPL methods have Feature-Oriented architecture. According to Apel et al. [Apel et al., 2013], Software Product Lines reconcile mass production and standardization with mass customization in software engineering. Ideally, based on a set of reusable parts, a software manufacturer can generate a software product based on the requirements of its customer. The concept of *"feature"* is central to achieving this level of automation, because features bridge the gap between the customer requirements and the functionality a product provides. Thus, features are a central concept in all phases of *product-line development.* Therefore, to introduce a useful, and practical APL method, as well as to gain

popularity among practitioners and "SPL community", we have chosen to propose a *Feature-Oriented APL method*.

6.4.2 Agile methods to be tailored for the APL process

As shown in the Section 4.4 in Chapter 4, the *eXtreme Programming* and *Scrum* methods are the most frequently used in the context of agile SPL. In addition, it was shown in the systematic literature review that the applicability of agile methods for Application Engineering (AE) seems very feasible, however, the applicability of agile methods for Domain Engineering (DE) still remains challengeable, especially for traceability and product-line architecture.

Based on the findings obtained in Chapter 4, the decision was made to tailor the *"Scrumban method"* for the DE and *"Scrum method"* for the AE. Scrum and Scrumban were selected for several reasons. Sections 6.4.2.1 and 6.4.2.2 present the main reasons that encourage the choices.

6.4.2.1 Why a Scrumban-inspired process for Domain Engineering?

DE is the process of analyzing the domain of a 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. Thus, *Domain Engineering targets development for reuse* [Apel et al., 2013] [Pohl et al., 2005]. These reusable artifacts could be part of systems or mini-systems, (as well as codes, diagrams, etc.) that could be integrated to other systems in order to derive the final products.

As stated earlier in this thesis, the Kanban methodology [Anderson, 2010] was developed at Toyota, where the automotive manufacturer has large-scale and complex production lines. Kanban has been defined as a method for managing and improving service delivery gradually over time. According to Ladas [Ladas, 2009], who has introduced Scrumban methodology, Scrumban was created as a means of transitioning a development team from Scrum to Lean or Kanban. However, Scrumban has become a methodology of its own, combining elements of both Scrum and Kanban. Practitioners and academics [Smartsheet.com, n.d.], have asserted that Scrumban is most commonly used in *development* and *maintenance* projects. In practice, both development and maintenance teams need many of the same skills. Development teams need a means of managing their entire development process, while maintenance teams must be able to make updates and repairs to faulty software [Nikitina et al., 2012].

The nature of Scrumban it was a favorable criterion that have conducted us to choose to tailor Scrumban for DE. ***First***, the Scrumban methodology grants more importance to the requirements engineering than the other agile methodologies (e.g. Scrum). In fact, Scrumban process is decomposed in two main parts, namely, *"Specification"* and *"Production"*, where the specification part is less aggressive toward the upfront design than Scrum. ***Second***, the Scrumban methodology establishes in its development process a structure of limited number of

"Production Lines" (see Figure 3.9). This feature of Scrumban makes it suitable for the Domain Engineering. ***Third***, since the Scrumban methodology is commonly used for *maintenance* projects, it may open the opportunity to deal with the *traceability* issue. In fact, the traceability phase mainly traces information that can be visualized and presented to the user for comprehension tasks, and to the development team for testing and maintenance tasks. Therefore, the agile practices of maintenance proposed in Scrumban process could be tailored to the *DE maintenance*. Accordingly, it was considered that the Scrumban methodology might be suitable to be tailored for the integration of its agile practices and activities with the DE.

6.4.2.2 Why a Scrum-inspired process for Application Engineering?

In contrast to Domain Engineering (DE), Application Engineering (AE) has the goal of developing a specific product for the needs of a particular customer or other stakeholder. It corresponds to the process of *single application development* in traditional software engineering, but reuses artifacts from domain engineering where possible. It targets *development with reuse*. AE is repeated for every product of the product line that is to be derived [Apel et al., 2013] [Pohl et al., 2005]. Therefore, in the product lines of AE, each team uses the reusable artifacts required for the derivation of the final product as well as the development of the artifacts that do not exist in the platform. Consequently, Scrum brings agility to the AE with high effectiveness. Moreover, the systematic literature review has confirmed prior assumptions, as one of its findings is the following: *"applicability of agile methods for Application Engineering (AE) seems very feasible"*.

As shown in Figure 3.4, according to the 13th annual State of Agile Report, *Scrum* is still a popular agile methodology with a use of 54% in overall agile projects [CollabNet VersionOne, 2019]. Thus, Scrum was chosen to be combined in the AE due to its popularity and due to its effectiveness.

6.5 Agile Framework for managing evolving Software Product Lines: The AgiFPL method

The intention behind the AgiFPL framework is to develop a proper APL methodology, which combines agility with SPL effectively, and embodies the advantages of both approaches. This section presents how the APLE is tackled by means of a tailored Scrum, and Scrumban, in which the DE and AE processes are performed in an iterative and incremental way.

To overcome the Product Line Architecture (PLA) challenge and reducing the upfront design, it is necessary to define the mechanisms and strategies to be applied during the APLE development process. This is why this contribution is based on two main aspects: *requirement engineering (RE)* and *development process*. In this chapter, the AgiFPL development process is presented. The requirements engineering part is presented in a separate chapter (see Chapter 8). In fact, the proposed requirement engineering approach will provide the mechanism to easily evolve PLAs in an agile context and will establish a suitable reuse strategy. Furthermore,

adopting Scrumban and Scrum for the development processes in DE and AE will establish the agility of the selected method.

AgiFPL is thus an agile methodology designed to improve the agility within the SPL and to meet effectively any new emerged business expectations. The main goal of AgiFPL is to move teams from the classical approach to a more evolved APLE framework. In other words, on the one hand, for the DE stage, AgiFPL implements an iterative process that combines agile activities and practices from Scrumban method. At this stage the roles, activities, artifacts, board, and meetings of Scrumban are adapted to be used within the different parts of the DE process. On the other hand, for the AE stage, AgiFPL also implements an iterative process, which combines agile activities and practices from the Scrum method with the AE process. For the AE tier, roles, activities, artefacts, and meetings of Scrum are used by the different parts of AE process.

6.5.1 AgiFPL Description

AgiFPL is a two-tier framework where the first layer is dedicated to the *Domain Engineering (DE)* and the second one, is dedicated to the *Application Engineering (AE)*. Moreover, the framework considers two spaces i.e. *Problem Space* and *Solution Space*.

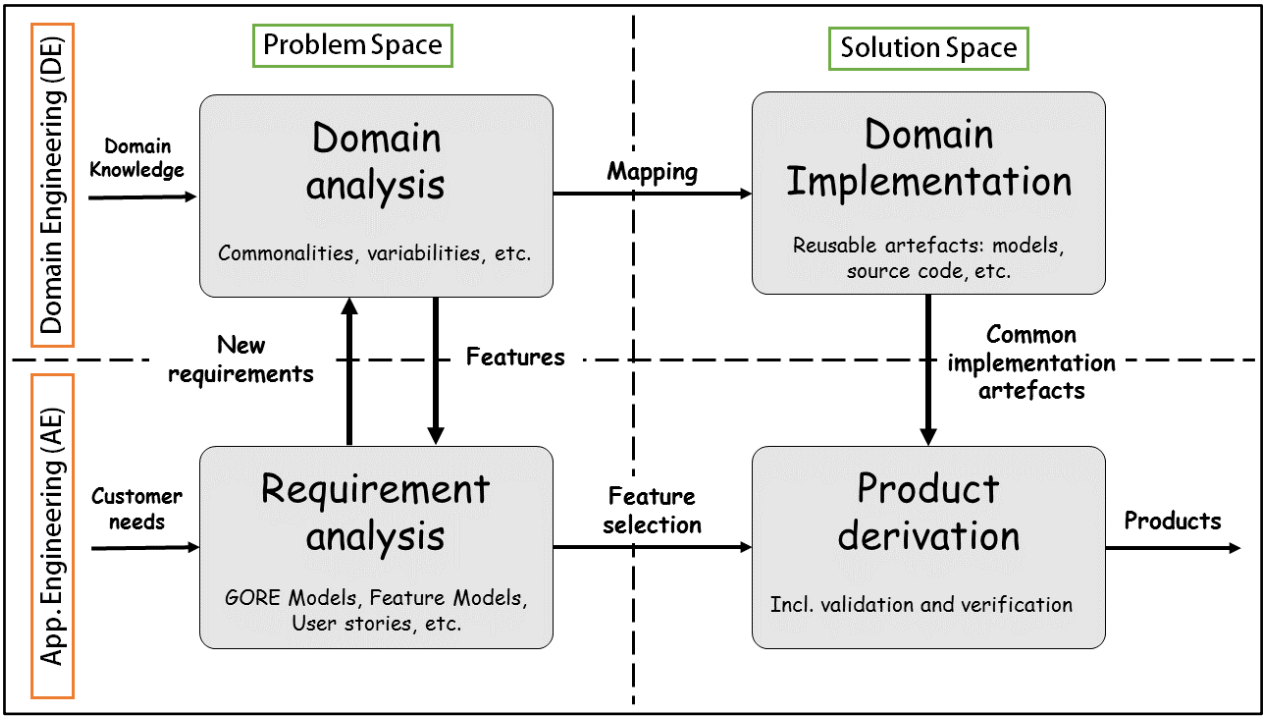

Fig. 6.3 Overview of the engineering process for AgiFPL. Revised and adapted from [Apel et al., 2013].

Figure 6.3 presents a macro-view of the AgiFPL process that is a two-dimensional structure with four main clusters of tasks in product-line development and the mapping between them. As the proposed APL method is a feature-oriented method, AgiFPL makes the distinction between two main spaces, namely, *Problem Space* and *Solution Space*. This distinction highlights two different perspectives. The problem space takes the perspective of stakeholders

and their problems, requirements, specifications, goals, and views of the entire domain and individual products. Features that are domain abstractions characterize this space. The problem space is decomposed in two big clusters of tasks that are the following:

- *Domain Analysis*: the domain analysis is a form of requirements engineering for the entire domain of the Software Product Lines (SPL). At this stage, people involved (i.e. Domain experts, Domain engineers, etc.) draw the scope of the domain. In fact, they decide which products have to be covered by the product line and, thus, which features are relevant and should be implemented as *reusable artifacts*.

- *Requirements Analysis*: the requirements analysis is the investigation and the elicitation of the needs of a specific client as part of the Application Engineering (AE). At this stage, the customer requirements are mapped to a feature selection, based on the features identified during domain analysis. When new requirements are discovered, they can be fed back into domain analysis, which may result in a modification of the feature model and, thus, the reusable domain artifacts.

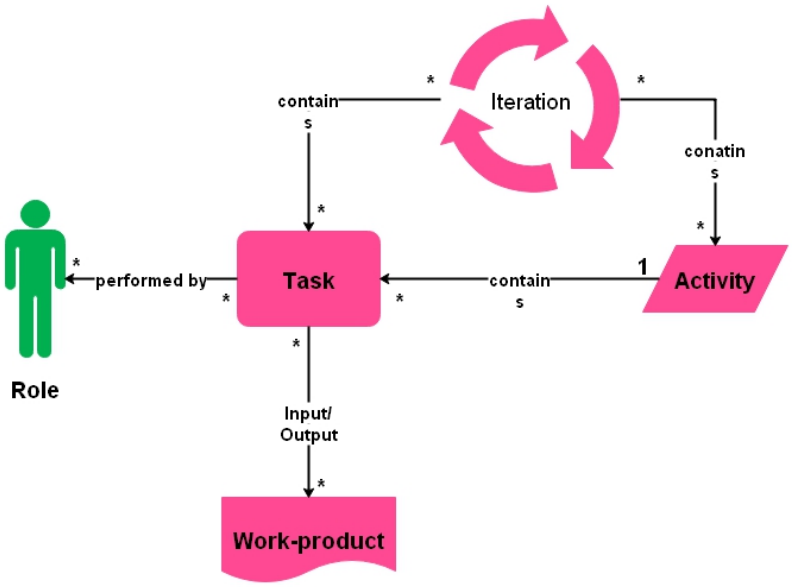

Fig. 6.4 Abstract relationship between the elements of AgiFPL processes.

The solution space represents the developer and vendor 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*, *validation* and *verification* of features and their combinations in suitable ways to facilitate *systematic reuse*. The solution space is also decomposed in two big clusters of tasks that are the following:

- *Domain Implementation*: this cluster of tasks is dedicated for the process of developing reusable artifacts that correspond to the features identified in Domain Analysis. Note that the reusable artifacts could be design, tests, source code, models, etc.

- *Product Derivation*: this phase is dedicated for the production step of Application Engineering, in which 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.

Furthermore, the AgiFPL critical processes are built on the roles, work-products (i.e. artifacts), units of work (activities and tasks), and iterations. Figure 6.4 shows the relationships among the elements that construct the processes of AgiFPL.

6.5.1.1 AgiFPL Roles

By definition, *roles* are the set of related responsibilities and skills. The role concept does not represent individual people, but rather the responsibilities that the stakeholders take on to perform a task. According to the AgiFPL framework, there are two categories of roles, namely, roles related to the Domain Engineering (DE) and roles related to the Application Engineering (AE).

6.5.1.1.1 Domain Engineering Roles according to AgiFPL

The core roles of the Domain Engineering (DE) tier defined according to the AgiFPL framework are the following roles:

- Business experts;
- Domain experts;
- Domain Sensei;
- Domain Development Team;
- Domain Stakeholders.

Business Expert

The *"Software Vendor"* strategy is defined mainly by the Business Experts. According to the AgiFPL method, a *Business Expert* plays a key role in aligning business goals, prioritizing major features, selecting business goals and market strategies. These elements lead to the definition of the Domain Scope.

Figure 6.5 represents the core tasks and elements of the "Business Expert" role. Based on the *marketing strategy* and the *market analysis*, the Business Experts define business goals, customer needs, legal and cultural constraints, competitors, and so on. In addition, Business Experts play an additional role in eliminating possible conflicts between the *Business*, the *Product-Line*, and the *Customer* goals.

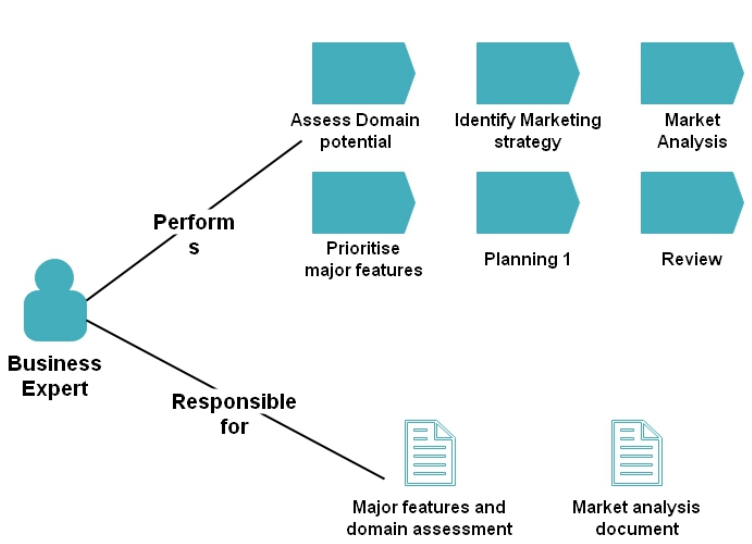

Fig. 6.5 Role of Business expert

Furthermore, a Business Expert *prioritizes the major features* according to the criteria stemmed from the market vision, which gives information about market potential or value, potential for reuse, risks, and potential for change or stability. The criteria or questions may be filtered based on goals.

This role, of Business Expert, requires good communication and collaboration skills. Moreover, the domain knowledge, market, Software Product-Line goals, and competitors are essential to the Business Expert to make appropriate decisions.

Domain Expert

The Domain Experts could come from different backgrounds. In fact, they represent the end users, clients, sponsors, business analysts, or any mix of these. A Domain Expert uses deep knowledge of the business to explain to the developers in various levels of details the tasks that the system must perform. The Domain Experts represent the interests of the top management of the *"Software Vendor"*. They actually take the role of *Product Owner* of the Scrumban framework. Figure 6.6 illustrates the main agile tasks and activities related to the role of Domain Expert.

Based on the Scrumban-inspired DE process, the Domain Expert takes decisions from his or her domain knowledge, such as the changes in the legislation and the impact of the technology on the domain. The Domain Expert provides legacy features to the Domain Design, helps resolving name or description conflicts, granularity issues, etc. During the prioritization of major activity, the Domain Expert prioritizes the features through the potential for reuse, risks, maturity, potential for change, and commonality and variability criteria. In addition, during the tasks "Develop a Feature Model" and "Develop a Product Map", the Domain Expert helps to define what features are common to all products and which are variable and how these variable features must be specified (i.e. commonalities and variabilities).

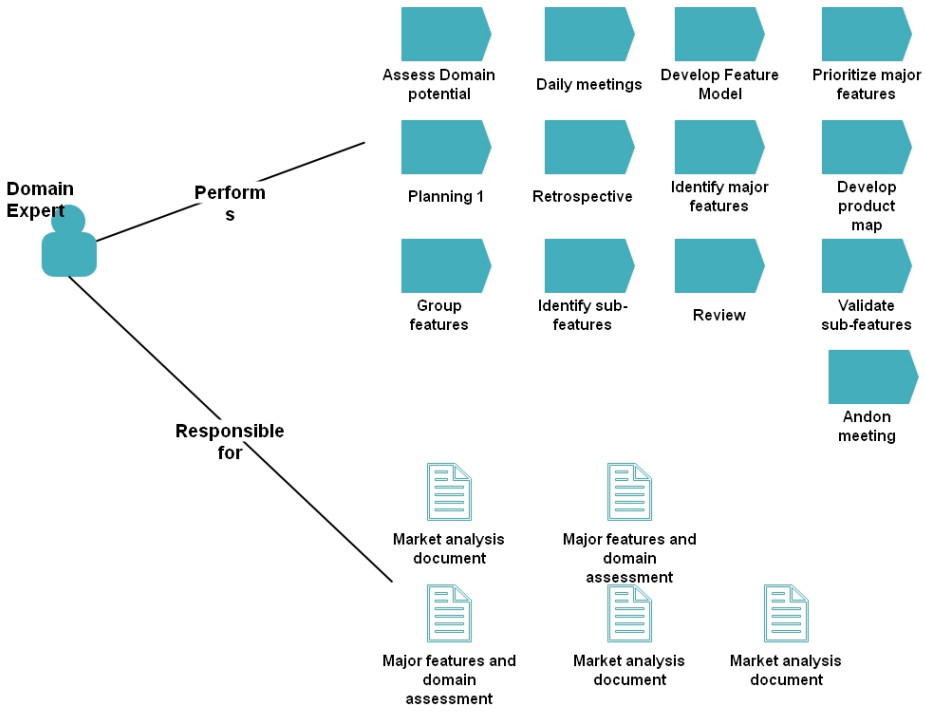

Fig. 6.6 Role of Domain Expert.

The role of "Domain Expert" requires good (written) communication and collaboration skills in order to facilitate the use of agile principles and practices. Furthermore, the Domain Expert should pay attention to innovations and changes in the market, technology, or governmental politics. Business Experts and Domain Experts should closely work together to ensure a solid synergy in "Domain Design" phase.

Domain Sensei

The Domain Sensei role is involved in all tasks and activities of *Scrumban-inspired Domain Engineering Process*. The Domain Sensei role is basically a role of supporting of the agile principles and practices for the other roles. In other words, the Domain Sensei is the responsible for the correct use of the Scrumban-inspired DE process. Although the designation of a Domain Sensei and his/her presence in all meetings (i.e. Planning 1, Planning 2, Daily, Review, Retrospective, and Andon) is generally advisable.

Figure 6.7 shows the different tasks and activities adopted from Scrumban and tailored for the Domain Engineering. The Domain Sensei should have a deep knowledge about agile values and principles, as well as, the adopted Scrumban practices in the DE process. In fact, the Domain Sensei removes obstacles for the team and guides the project members focused on the Domain Design goal. According to AgiFPL, the Domain Sensei must know the value of each Scrumban practice and activity related to DE Meetings and Production Flows.

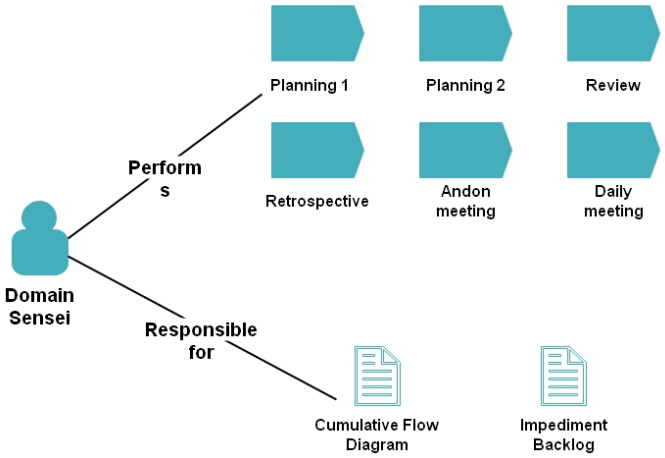

Fig. 6.7 Role of Domain Sensei.

Domain Development Team(s)

The Domain Development Team is a group of developers with complementary skills that perform all the defined stories and task.

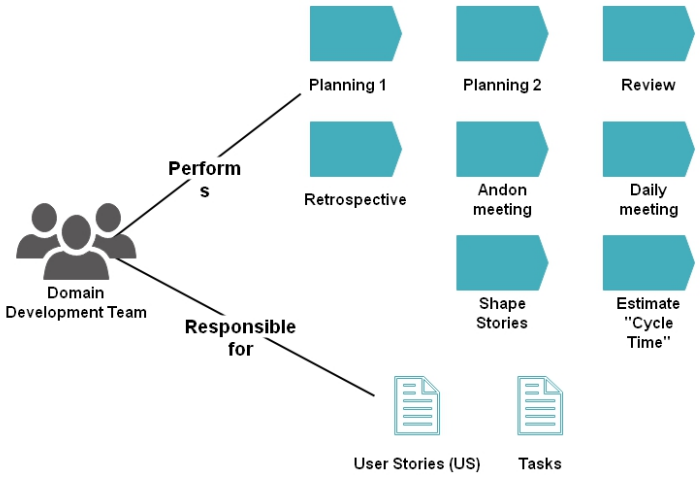

Fig. 6.8 Role of Domain Development Team.

Figure 6.8 represents the adopted Scrumban practices for the Domain Development Team members. The Domain Development Team is a cross-functional group of people responsible for delivering potentially shippable increments of reusable artifacts (at the end of every production cycle). In fact, the members of the Domain Development Team develop the reusable artifacts for the Software Product Lines. They perform the selection of tasks for the different product lines during the “Planning 1” and “Planning 2”. They also pay attention to the commonalities and variabilities that have to be specified.

The members of the Domain Development Team have deep technical programming skills and they have good collaboration skills when coding. In addition, they apply all the agile practices that allow them to pay attention to the commonalities and variabilities according to the defined goals.

Domain Stakeholders

Domain Stakeholders are the people involved that enable 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 external consultants, managers, customers and users related to the Domain of the Software Product Lines.

6.5.1.1.2 Application Engineering Roles according to AgiFPL

As stated earlier in this chapter, the Scrum method was tailored for the Application Engineering (AE) tier. In AE there are several production lines. Each line of the AE phase has its own team called ***"Line i Team"*** where ***i ∈[1, 2,…, N]***. The core roles of ***"Line i Team"*** defined according to the AgiFPL framework are the following roles:

- App i Owner;
- Line i Scrum Master;
- Line i Development Team;
- App i Stakeholders.

App i Owner

The "App i Owner" is the person responsible for maintaining the *"App i backlog"* by representing the interests of the stakeholders, ensuring the value of the work the development team does. Figure 6.9 illustrates the role of the *App i Owner* according to the Scrum-inspired AE process of AgiFPL.

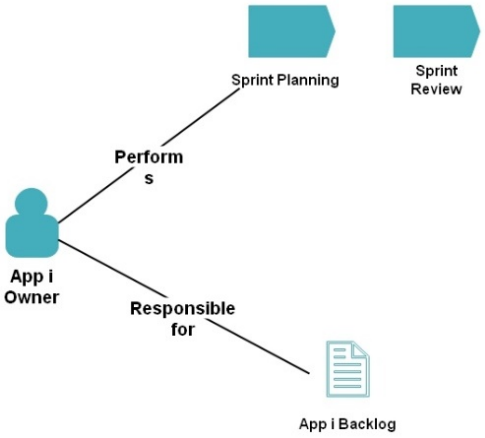

Fig. 6.9 Role of App i Owner.

Line i Scrum Master

The Line i Scrum Master is the person responsible for the product line i process, making sure it is used correctly and maximizing its benefits. In fact, the role of the Line i Scrum Master is involved in all tasks that are balanced with Scrum practices. This role is a role of supporting agile principles and practices for the other roles of the *"Line i"*.

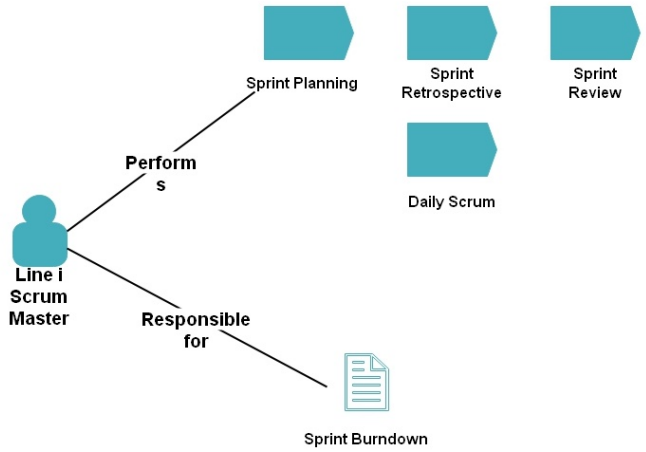

Fig. 6.10 10 Role of the Line i Scrum Master.

Figure 6.10 shows the main tasks of the Scrum Master of the line i. Basically, Line i Scrum Master removes most of the obstacles for the team, and guides the project members focused on the sprint goal.

Line i Development Team

The *"Line i Development Team"* is a cross-functional group of people responsible for delivering potentially shippable increments of the App i at the end of every sprint. Figure 6.11 represents the agile practices and activities of the role of the "Line i Development Team". The members of the development team have complementary skills that perform the sprint tasks.

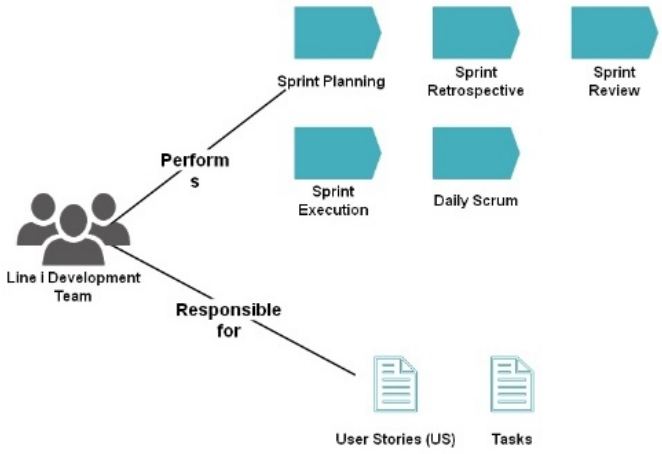

Fig. 6.11 Line i Development Team.

App i Stakeholders

"App i Stakeholders" role represents the people involved that enable the project related to the "Product i". 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 "App i Owner". Typically, the main stakeholders are managers, customers and users that are related to the "Product i".

6.5.1.2 AgiFPL's Units of work

Activities, Tasks, and Steps are *"Units of Work"* performed by the roles defined by the AgiFPL method with specific goals, and defined in- and output. The activities group the tasks with common goals. The tasks group steps and guidelines that guide the roles to reach their goals.

Agile activities, tasks, and steps for the DE process are adopted from the Scrumban practices, where these practices are tailored to fit the SPL structure of the DE. In addition, the Scrum practices are tailored to fit the SPL structure of the AE.

6.5.1.2.1 Meetings in the Scrumban-inspired Process for Domain Engineering

According to the AgiFPL principles, there are six kinds of meetings for the DE process that were adopted from Scrumban practices to ensure the collaboration and communication between customers, stakeholders and development teams, as well as the communication amongst the development team members. These meetings are the following:

1. *Planning 1*: The "WHAT": Whenever the product owner pulls new user stories into the selected backlog. The *"Domain Analysts"* holds it to select the next user stories to work on, explaining the user stories of the Features Backlog and answering open questions. After this analysis, the development team should understand the requirements. Therefore, 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 (i.e. "Domain Sensei" and "Domain Development Team") discusses solutions for new user stories in the SIP backlog and creates tasks for each user story accordingly.

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

    i. What have I done since yesterday?
    ii. What am I planning to do today?
    iii. 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 increment.

5. *Retrospective*: (After any review) The "Domain 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:

    i. What went well during the last cycle?
    ii. What should improve in the next cycle?

6. *Andon*: (*Whenever a problem occurs*) the "Domain Sensei" organizes an *Andon* meeting whenever problems in the production flow occur. For example, a story is 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 place in this meeting and work on a solution to solve the open issue.

6.5.1.2.2 Meetings in the Scrum-inspired Process for Application Engineering

To ensure an effective feedback channel between customers, developers aim to establish a deep involvement of customer in the development process. The AgiFPL method has also defined six type of meetings adopted from Scrum method for the Scrum-inspired process for AE. These meetings are the following:

1. *Sprint Planning 1*: (60 min per sprint week) is held to select the work to be done for the next sprint (the "WHAT"). The "App i Owner" explains the stories of the App backlog to the team and answers their question. After the planning, the team should have understood the requirements and its commits the scope for the sprint.

2. *Sprint Planning 2*: (60 min per sprint week) the designing phase for the selected backlog (the "HOW"). The team discusses a solution for the selected stories and creates according task for each story.

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

    i. What have I done since yesterday?
    ii. What am I planning to do today?
    iii. What are my impediments?

4. *Sprint Review*: (up to 60 min per sprint week) used to present and review the work that was completed and not completed during a sprint. It should include a demonstration of the realized product increment.

5. *Sprint Retrospective*: (about 45 min per sprint week) a reflection on the past sprint used to make continuous process improvements. Two main questions are asked in the sprint retrospective:

    i. What went well during the sprint?
    ii. What could be improved in the next sprint?

6. *Backlog Refinement*: (max. 60 min) used to introduce and estimate new backlog items and to refine existing estimations as well as acceptance criteria. It is also used to break large stories into smaller ones.

6.5.1.3 Work-products of AgiFPL

The work-products are artifacts produced, modified or read by team's people or stakeholders when performing tasks and activities. Since AgiFPL approach has two tiers, each tier has its own work-products.

6.5.1.3.1 Work-products of Domain Engineering Process according to AgiFPL

Based on AgiFPL method, the *Domain Engineering (DE) Process* has the following activities and artifacts:

- *Domain Requirement Engineering (DRE)*: DRE aims to understand the problem space by studying the settings of the organizations related to the domain. By studying a set of organizations, the emphasis is put on understanding the motivations and rationales that underline the domain requirements. In fact, at this stage the reusable artifacts-to-be are described within its operational environment, along with relevant functions and qualities.

- *Domain Design (DD)*: At this stage, the scope of the domain and the architectures have to be determined, i.e., decide which the product line should cover products and, consequently, which features are relevant and should be implemented as reusable artifacts.

- *Features Backlog (FB)*: an ordered list of features that the team maintains for a product. At this stage, one should document requirements in 'user story' format. Anyone can edit the backlog, but "Domain Analysts" are ultimately responsible for ordering the user stories. Stories in the features backlog contain rough estimates of both business value and development effort.

- *Selected backlog (SB)*: 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 or features from the top of the product backlog.

- *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.

- *Task backlog (TB)*: 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 or 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.

- *User Story (US)*: a description of a certain feature or behavior, written strictly from the *Domain Experts'* point of view. Usually, the *"Domain Experts"* writes the user stories.

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

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

- *Parking lot*: for tasks, which the team cannot 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 deadlocks, where unfinished tasks block production lines.

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

6.5.1.3.2 Work-product of the Application Engineering Process according to AgiFPL

The Scrum-inspired Process of the *Application Engineering (AE)* defined by the AgiFPL method has the following activities and artifacts:

- *App Requirement Engineering (ARE)*: where the system-to-be (App-to-be) is described within its operational environment, along with relevant functions and qualities.

- *App Backlog*: an ordered list of requirements that the team maintains for an application. In AgiFPL approach, 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 App Backlog contain rough estimates of both business value and development effort.

- *Sprints*: work is performed in iterations or cycles of up to a calendar month called sprints. Sprints are time-boxed so they always have a fixed start and end date, and generally, they should all be of the same duration. A new sprint immediately follows the completion of the previous sprint.

- *Sprint Backlog*: a list of work the development team must address during the next sprint. The list is created by selecting user stories from the top of the product backlog until the development team feels it has enough work to fill the sprint, keeping in mind the velocity of its previous sprints. The stories or features are broken down into tasks by the development team. Often an accompanying task board is used to see and change the state of the tasks of the current sprint, like "To Do", "In Progress" and "Done".

- *Sprint Execution*: Once the team finishes sprint planning and agrees on the content of the next sprint, the development team, guided by the *Line i Scrum Master*'s coaching, performs all of the task-level work necessary to get the features done. "Done" means there is a high degree of confidence that all of the work necessary for producing *good-quality* features has been completed.

- *User story*: a description of a certain product/application feature or behavior, written strictly from the user's point of view. Usually, the *"App i Owner"* writes the user stories.

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

- *Impediment Backlog of Line i*: a list of current impediments maintained by the *Line i Scrum Master*.

- *Definition of Done (DOD)*: a checklist of activities required to declare the implementation of a story to be completed. The definition is determined at the beginning of the project but the team can change it at any time.

6.5.2 AgiFPL Framework

As described in Section 6.5.1, AgiFPL is a two a tier framework, where the first layer, modelled in Figure 6.12, is dedicated to the Domain Engineering (DE) and the second one, modelled in Figure 6.13, is dedicated to the Application Engineering (AE). Moreover, the framework considers two spaces i.e. Problem Space and Solution Space. Each tier has its start point, at the DE tier, the start point is the *"Software Vendor"* and at the AE tier, the start point is the *"App Owner"*.

6.5.2.1 AgiFPL's Domain Engineering (DE) sub-process

Hereafter the process of DE according to AgiFPL shown in Figure 6.12, is described in detail. First, the *Software Vendor* has a business strategy for the domain. This strategy built from market trends, potential investment opportunities and core-business improvements. The domain strategy constitutes the main base to define the scope of the domain and start the phase of *Domain requirement engineering (DRE)*.

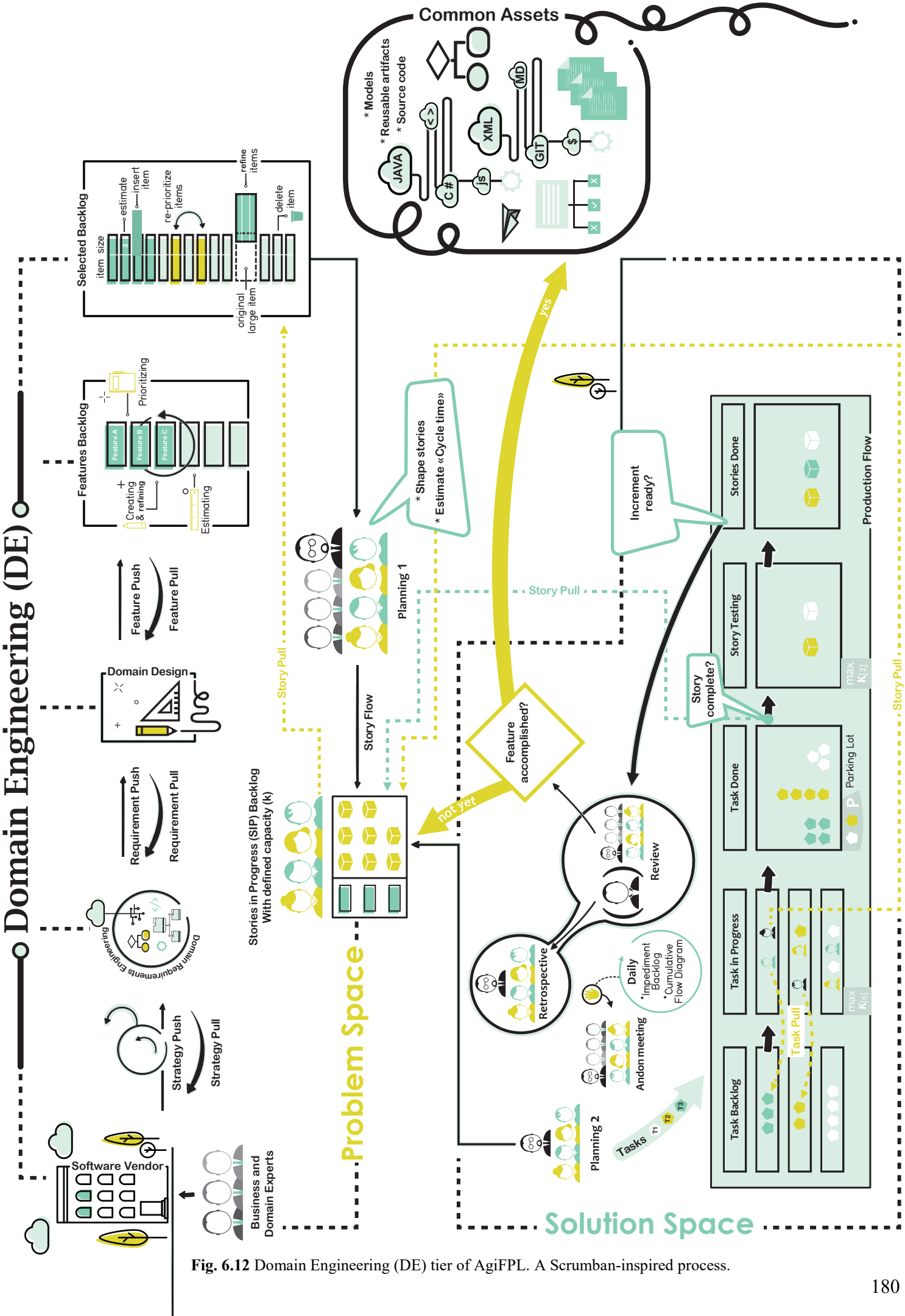

Fig. 6.12 Domain Engineering (DE) tier of AgiFPL. A Scrumban-inspired process.

The strategy pushes to start the *DRE phase*. The DRE phase is the first step in the problem space. DRE is a sub-process that aims to understand the problem space by studying the settings of the organizations related to the domain as well as the targeted stakeholders. By studying a set of organizations, the emphasis is put on understanding the motivations and rationales that underline the domain requirements. The roles involved in this sub-process are *Business Experts*, *Domain Experts*, *Domain Sensei* and *Development Team*. During this sub-process, the people involved model the target models according to specific approach designed for the RE of both tiers DE and AE. Chapter 8 presents this approach and gives the whole explanations about the proposed Requirements Engineering framework for Agile Product Lines [Haidar et al., 2018] [Haidar et al., 219]. During this stage, there are many interactions between the people involved and thus some elements could be identified during the analysis that could influence the strategy of the beginning. When the *"Software Vendor"* adopts a new or modified strategy, *"Business and Domain Experts"* **pull** this strategy and upgrade the *"Domain Requirements"* by restarting the DRE phase. That's why the ***"Strategy Pull"*** link exists.

After receiving all the documentations and models delivered at the end of the DRE phase the phase of *Domain Design (DD)* starts. The DD phase takes the reference GORE models as input and creates a reference architecture for the product line's platform, which gathers the common assets. At this phase, the scope of the domain and the architectures have to be determined, i.e., decide which products should be covered by the product line and, consequently, which features are relevant and should be implemented as reusable artifacts.

DD phase contains two primary tasks: *domain scoping and domain modeling*. Domain scoping is the process of deciding on the extent or range of a product line. During this step, domain engineers decide which of all possible requirements arising in a domain should be considered. Domain modeling captures and documents the commonalities and variabilities of the scoped domain, as the scope of a product line should be recorded. The results of DD phase are usually documented in a *feature model (FM)* (for more details, see Chapter 8). As well as in the DRE phase, the roles involved in the DD phase sub-process are *Business and Domain Experts*, *Domain Sensei* and *Domain Development Team*.

Once commonalities and variabilities are identified and feature models are generated at the DD phase, *Domain Experts* defines the *Features Backlog (FB)*. During this phase, the persons involved document features in *User Stories (US)* format. At this stage, stories contain rough estimates of both business value and development effort. At the end of this stage, the *Domain Experts* build a *Selected Backlog (SB)* with defined capacity from the FB. The SB represents a list of work the *Domain Development Team* must address next. It has a defined capacity limit. As soon as capacity is available, it is filled up with user stories or features from the top of the FB.

After building the SB, *Domain Experts* and *Domain Development Team* hold the *Planning 1* meeting in order to shape stories and estimate *Cycle Time*. The *Planning 1* is considered as the *"WAHT"*: Whenever the *Domain Experts* pulls new user stories into the SB. The *Domain Experts* have the power to select the next user stories to work on, explaining the user stories of features backlog and answering open questions. After this analysis, the *Domain*

Development Team should comprehend the different features. Therefore, the team is able to estimate the complexity of each user story.

Once the *Planning 1* is held, the *Domain Development Team* structures the *Stories in Progress Backlog (SIP Backlog)* with *defined capacity* ***(K[n])***. *SIP Backlog* is 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.

After structuring the SIP Backlog, *Domain Sensei* and *Domain Development Team* hold the *Planning 2* meeting. With the *Planning 2* starts the *Domain Implementation or Production of Common Assets* phase. The *Planning 2* is considered as 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 creates tasks for each user story accordingly. During the *Planning 2* meeting, the team establish the *Production Flow*. The ***Production Flow*** components are the following:

1. *Task Backlog*;
2. *Task in Progress* with defined capacity (**max *k[6]***). Once the "Task in Progress" is done, team members pull tasks from "Task Backlog".
3. *Task Done* with *"Parking Lot"*. Once the story is completed (*Story complete?*) *"Development Team"* members pull user stories from the "SIP Backlog".
4. *Story Testing* with defined capacity (**max *k[2]***);
5. *Stories Done*;

During the *Production Flow*, *Domain Sensei* and *Domain Development Team* hold a *Daily* meeting (15 min max). The *Daily* is a short, time-boxed meeting, taking place every day at the same time. Every team member answers three questions:

1. What have I done since yesterday?
2. What am I planning to do today?
3. What are my impediments? Note that if there are impediments, these impediments are maintained by the *Domain Sensei* in a list called *Impediment Backlog*.

Whenever the team ships an increment (Increment Ready?), the *Business and Domain Experts*, *Domain Sensei* and *Domain Development Team* hold a *Review* meeting in order to review the work accomplished. 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 increment. Once the features are implemented as *reusable artifacts (Models, Source Codes,…)*, these artifacts are placed in the *"Common Assets Warehouse"*.

After any *Review*, the Domain Sensei holds the *Retrospective* meeting to reflect on the past production cycle in order to ensure continuous process improvements. The *Domain Sensei* always asks two questions in the retrospective:

1. What went well during the last cycle?
2. What should improve in the next cycle?

During the *Production Flow*, whenever a problem occurs, the *Domain 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 *Domain Development Team* and the *Domain Experts* take place in this meeting and work on a solution to solve the open issue.

6.5.2.2 AgiFPL's Application Engineering (AE) sub-process

In this section, the process of AE according to AgiFPL shown in Figure 6.13, is described in detail. In *AE* tier, there are several product lines where the output is client applications (products). At this stage, *"App i Owners"* and *"App i Stakeholders"* are more involved in the development process. Hereafter, the AgiFPL proposition for the *"Product (App) i Line",* which deliver at the end the *"App i",* is described. In fact, the AgiFPL method proposes a simple agile process based on Scrum approach. Each line of the AE tier will follow this process.

First, an *"App i Owner"* comes with an idea of what he/she wants to create. The *App i Line* process starts and lunches the *"App Requirement Engineering" (ARE)* phase. This phase has two main missions:

1. The *"App i Owner"*, *"App i Stakeholders"*, *"Line i Scrum Master"*, and *"Line i Development Team"* study the *"Organization i"* in order to understand the motivations and rationales that underline the *"App i"* requirements. In fact, people involved model the target *"App i"* according to the RE approach presented in Chapter 8.
2. The *"App i Owner"* idea could be large, therefore, through an activity called *"Grooming"*; it is broken down into a set of features that are collected into a prioritized list called the *"App i backlog"*.

Note that, the requirement assets produced in Domain Engineering (DE) are a start, but they will not satisfy all "App i Owners" and "App i Stakeholders" requirements completely. The gap between what is available and what is required must be analyzed, and a trade-off decision made for each unsatisfied requirement. At the end of the ARE phase there are two type of features:

- Features that exist and could be selected from the *Common Assets Warehouse*. The set of this features called *"Selection of Features" (SoF)*.
- Features that are required for the development of the application and do not exist in the *Common Assets Warehouse*. Since these features are defined during the ARE phase, the set of these features called *"Definition of Features" (DoF)*.

Consequently, the building of the *"App i backlog"* is finished and it contain an ordered list of documented *Use Stories (US)* that the team maintains for an *"App i"*. Note that all new reusable artifacts issued from the AE process have to be classified in the *"Common Assets Warehouse"*.

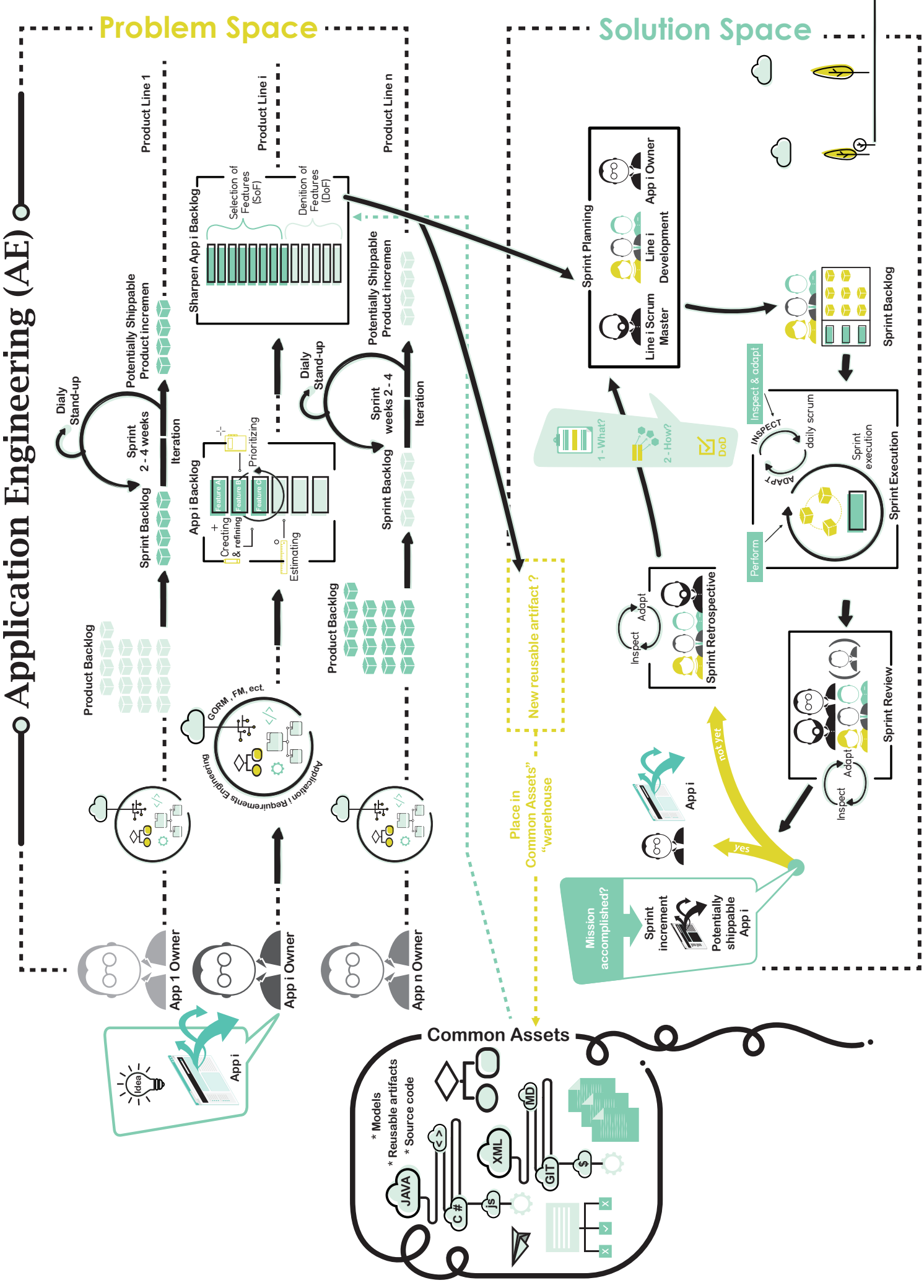

Fig. 6.13 Application Engineering (AE) tier of AgiFPL. A Scrum-inspired process.

Once the *"App i backlog"* is finished, the work is performed in iterations or cycles of up to a calendar month called *Sprints*. The work completed in each sprint should create something of tangible value to the customer or user. The *Sprint* starts with the *Sprint Planning*. The *"App i Owner"*, *"Line i Scrum Master"* and *"Line i Development Team"* perform the *"Sprint Planning"* in order to determine the most important subset of *"App i backlog"* items to build in the next sprint. During sprint planning, the *"App i Owner"* and *"Line i Development Team"* agree on a sprint goal that defines what the upcoming sprint is supposed to achieve. The *"Sprint Planning"* could be split in two meetings:

- *"Sprint Planning 1"*: (60 min per sprint week) is held to select the work to be done for the next sprint (the "WHAT"). The *"App i Owner"* explains the stories of the *"App backlog"* to the team (*"Line i Master"* and *"Development Team i"*) and answers their question. After the planning, the team should have understood the requirements and it commits the scope for the sprint.
- *"Sprint Planning 2"*: (60 min per sprint week) the designing phase for the selected backlog (the "HOW"). The team discusses a solution for the selected stories and creates according task for each story.

At the end of the *"Sprint Planning"*, the *"Sprint Backlog"* is defined and *"Definition of Done" (DoD)* list is established. In fact, DoD is a checklist of activities required to declare the implementation of a story to be completed.

Once the people involved in the "Line i" finish the*"Sprint Planning"* and agree on the content of the next sprint, the *"Line i Development Team"*, guided by the *"Line i Scrum Master"*, performs all of the task-level work necessary to get the tasks done. In fact, *"done"* means there is a high degree of confidence that all of the work necessary for producing good-quality features has been completed. This step is called *"Sprint Execution"*. Exactly what tasks the team performs depends of course on the nature of the work.

Each day of the sprint, ideally at the same time, the *"Line i Development Team"* members hold a time-boxed (about 15 minutes or less) *Daily* meeting. This *inspect-and-adapt* activity is sometimes referred to as the daily stand-up, because of the common practice of everyone standing up during the meeting to help promote brevity. A common approach to performing the daily Scrum has the *"Line i Scrum Master"* facilitating and each team member taking turns answering three questions for the benefit of the other team members:

1. What did I accomplish since the last *"Daily"*?
2. What do I plan to work on by the next *"Daily"*?
3. What are the obstacles or impediments that are preventing me from making progress?

The *Sprint* results are considered as a *"Potentially shippable App i increment"*, meaning that whatever the *"Line i Development Team"* agreed to do is essentially done according to its agreed-upon definition of done. This definition specifies the degree of confidence that the work completed is of good quality and is potentially shippable. At the end of the *"Sprint"*, there are two additional *inspect-and-adapt* activities:

1. *Sprint Review*: The goal of this activity is to inspect and adapt the product being built. Critical to this activity is the conversation that takes place amongst its participants, which include the *"App i Owner"*, *"App i stakeholders"*, *"Sponsors"*, *"Line i Scrum Master"*, *"Line i Development Team"*, and interested members of other teams. The conversation is focused on reviewing the just-completed features in the context of the overall development effort. Everyone in attendance gets a clear view into what is occurring and has an opportunity to help guide the forthcoming development to ensure that the most business-appropriate solution is created.

2. *Sprint Retrospective*: The second inspect-and-adapt activity at the end of the sprint is the sprint retrospective. This activity frequently occurs after the sprint review and before the next sprint planning. Whereas the sprint review is a time to inspect and adapt the product, the sprint retrospective is an opportunity to inspect and adapt the process. During the sprint retrospective the *"Line i Scrum Master"*, *"Line i Development Team"*, and *"App i Owner"* come together to discuss what is and is not working with *Sprint* and associated technical practices. The focus on the continuous process improvement is necessary to help a good *"Line i Development Team"* become great. At the end of a sprint retrospective, the *"Line i Development Team"* should have identified and committed to a practical number of process improvement actions that will be undertaken by the *team* in the next sprint.

After the *"Sprint Retrospective"* is completed, the whole cycle is repeated again—starting with the next sprint-planning session, held to determine the current highest value set of work for the team to focus on. At the end of the *Sprint* the *"Line i Scrum Master"*, *"Line i Development Team"*, and *"App i Owner"* perform a *Backlog refinement*: (max. 60 min) used to introduce and estimate new backlog items and to refine existing estimations as well as acceptance criteria. It is also used to break large stories into smaller ones.

After an appropriate number of sprints have been completed, the *"App i Owner's"* vision will be realized and the solution can be released.

6.5.3 AgiFPL critical processes: The Big Picture

The proposed AgiFPL approach has two main processes. The first process concerns the Domain Engineering (DE) tier and the second one concerns the Application Engineering (AE) tier. Each process is based on *iterative* and *incremental* development, as shown in Figure 6.14.

On the one hand, *iterative development* is a planned rework strategy. Multiple passes are used to improve what is being built, so that a good solution can be converged on. Iterative development is an excellent way to improve the "Domain" and the different "Application Lines" as they are being developed. The biggest downside to iterative development is that in the presence of uncertainty it can be difficult up front to determine (plan) how many improvement passes will be necessary.

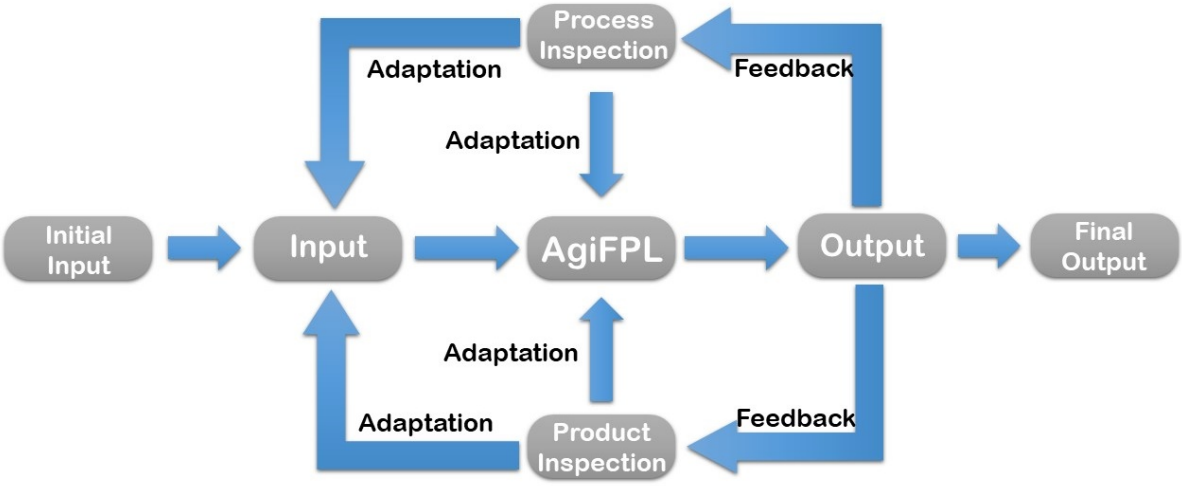

Fig. 6.15 AgiFPL process Model.

On the other hand, *incremental development* is based on the age-old principle of *"Build some of it before you build all of it"*. Team members avoid having one large, big-bang-style event at the end of development where all the pieces come together and the entire product is delivered. Instead, team members break the product into smaller pieces so that they can build some of it, learn how each piece is to survive in the environment in which it must exist, adapt based on what we learn, and then build more of it. Incremental development gives involved people important information that allows them to adapt our development effort and to change how we proceed. The biggest drawback to incremental development is that by building in pieces, we risk missing the big picture (we see the trees but not the forest).

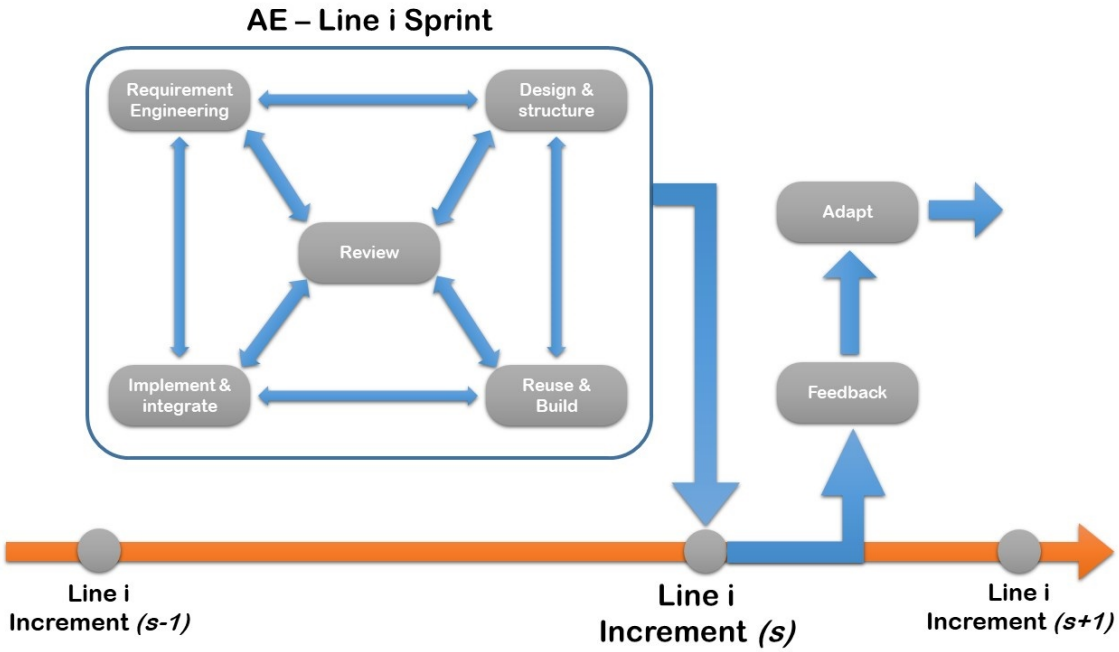

Fig. 6.14 AgiFPL method uses principles of iterative and incremental development.

Since the processes of AgiFPL are Scrum-inspired and Scrumban-inspired, the AgiFPL methodology leverages the benefits of both iterative and incremental development, while negating the disadvantages of using them individually. Therefore, AgiFPL does this by using both ideas in an adaptive series of time-boxed iterations called *"Production Flow"* in DE process and *"Sprints"* in AE process.

During each *time-box*, people involved in "Line i" perform all of the activities necessary to create a working product increment (some of the product, not all of it). Figure 6.15 shows that some analysis, design, build, integration, and test work is completed in each sprint. This *all-at-once* approach has the benefit of quickly validating the assumptions that are made when developing product features.

Based on AgiFPL method, the involved people do not work on a phase at a time. In fact, they work on a *feature* at a *time*. Therefore, by the end of a *time-box* they have created a valuable product increment (some but not all of the *Domain/Application* features). That increment includes or is integrated and tested with any previously developed features; otherwise, it is not considered done. At the end of the *time-box*, people involved in "Line i" can get feedback on the newly completed features within the context of already completed features. This helps them to view the *Domain or Application* from more of a *big-picture* perspective than they might otherwise have.

People involved in "Line i" receive feedback on the *time-box* results, which allows them to adapt. They can choose different features to work on in the next *time-box* or alter the process we will use to build the next set of features. In some cases, they might learn that the increment, though it technically fits the bill, is not as good as it could be. When that happens, they can schedule rework for a future *time-box* as part of their commitment to iterative development and continuous improvement. This helps overcome the issue of not knowing up front exactly how many improvements passes we will need. AgiFPL does not require that people involved in its different process predetermine a set number of iterations. The continuous stream of feedback will guide them to do the appropriate and economically sensible number of iterations while developing the product incrementally.

6.5.4 Some adopted Metrics for AgiFPL method

Since this research work tailor Scrumban and Scrum for DE and AE respectively, the AgiFPL method has adopted some metrics to evaluate the performance of teams and involved people. According to AgiFPL, there are metrics for the Scrumban-inspired process of DE, as well as metrics for the Scrum-inspired process of AE.

6.5.4.1 Metrics for AgiFPL's Domain Engineering

❖ **Kanban Board**

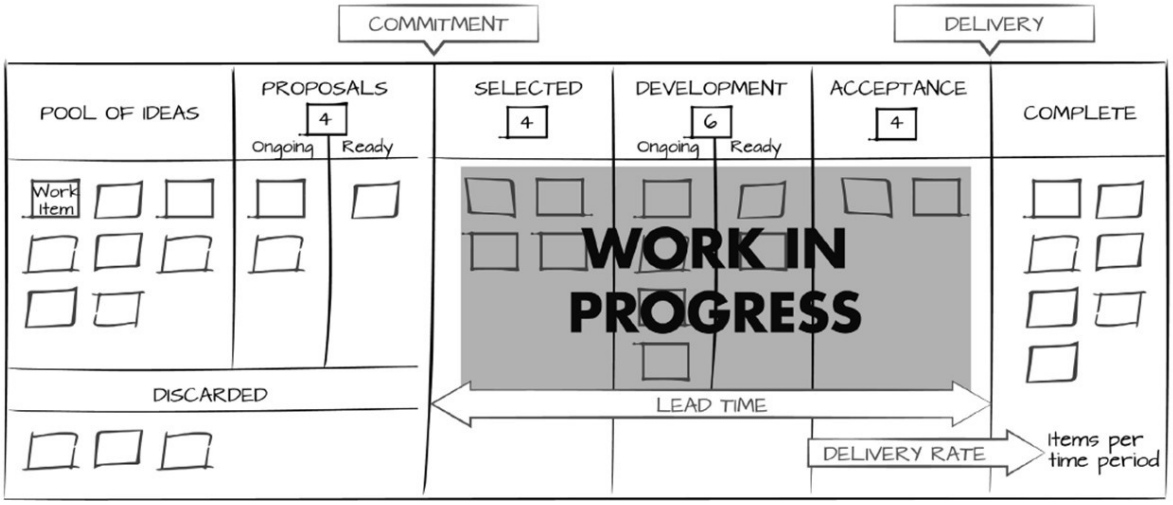

Fig. 6.16 A template of Kanban Board - Adopted from [Anderson and Carmichael, 2016].

According to [Anderson and Carmichael, 2016], Kanban is used to define, and improve systems that deliver services of value to the customer. As, in software industry, Kanban is applied to knowledge work, where the deliveries consist of information in various other forms rather than physical items, the process can be defined as a series of "knowledge-discovery" steps and their associated policies made visible in a *"Kanban board"*.

Figure 6.16 shows an example of Kanban Board. This board depicts a flow system in which work items flow through various stages of a process, ordered from left to right. Several conditions must be satisfied for this flow system in order to be a *Kanban system* for software development. These core conditions are:

1. There must be signals (usually visual) to limit work-in-progress (WIP). The signals are derived from the combination of the *"cards"*, the displayed *"WIP Limits"*, and the column that represents the *"activity"*.
2. Kanban systems must have identified *"commitment"* and *"delivery"* points.

❖ **Cumulative Flow Diagram (CFD) & Lead and Cycle time**

As shown in Chapter 3 (*Section 3.6.2.3*), the performance of work in the Scrumban-inspired process of DE would be evaluated via the *Cumulative flow diagram (CFD),* which is a publicly displayed chart showing a detailed view of the teams' past and present performance. The CFD allows identifying bottlenecks in the production flow. It also enables the product owner to predict the time a new requirement will most probable need to be completed.

Moreover, other important performance indicators are *Lead and Cycle Time* (LT and CT respectively). LT and CT define the time that it takes from the initial request to the completion of a task or the time from starting the task to its completion. They are used to estimate how long the iteration will last and what should be the backlog limit for the team.

The time that an item is in process between the *Commitment* and *Delivery points* is referred to as the item's *Lead Time* (or *System Lead Time*). *Customer Lead Time* may be different. In fact, it is the time a customer waits for an item – typically from request to receipt [Brechner, 2015].

The rate at which items are delivered is known as the *Delivery Rate*. This rate is obtained from the reciprocal of the time between the latest delivery and the previous one or, for an average Delivery Rate over a given period, by dividing the number of deliveries by the length of the period [Anderson, 2010].

$$Av(Delivery\ Rate) = \frac{Av(WIP)}{Av(Lead\ Time)}$$

Av (…) here denotes the arithmetic mean.

Furthermore, the term *Throughput* is used instead of *Delivery Rate* if the end of the process under consideration is not the *delivery point*.

$$Av(Troughput) = \frac{Av(WIP)}{Av(TIP)}$$

Here *TIP* means *Time in Process*; it is the period during which an item is in the process under consideration.

6.5.4.2 Metrics for AgiFPL's Application Engineering

- **Burndown chart**

According to the Scrum method, *Velocity* is a measure of the amount of work a team can tackle during a single Sprint and is the key metric in Scrum. Velocity is calculated at the end of the Sprint by totaling the Points for all fully completed User Stories (US) [Kenneth, 2013]. Points from partially-completed or incomplete stories should not be counted in calculating velocity. Velocity should be tracked throughout the Sprint on the Sprint Burndown Chart and be made visible to all Team members.

The Burndown chart is used to evaluate work progress. In fact, Burn down charts are publicly displayed charts showing the invested and remaining work. The team uses sprint burn down charts to visualize the progress within a sprint. Release burn down charts show the amount of work left to complete the target commitment for a product release.

The Kanban board could be used for the Scrum-inspired process of AE and considered as a visual tool that organizes the workflows within the processes.

6.5.5 How to implement AgiFPL

In order to put AgiFPL approach into practice, we have reviewed several transition models and experiments that were found in literature. Among the reviewed works, it was found that the transition approach, proposed by Gregg et al. [Gregg et al., 2016], has been effectively the work

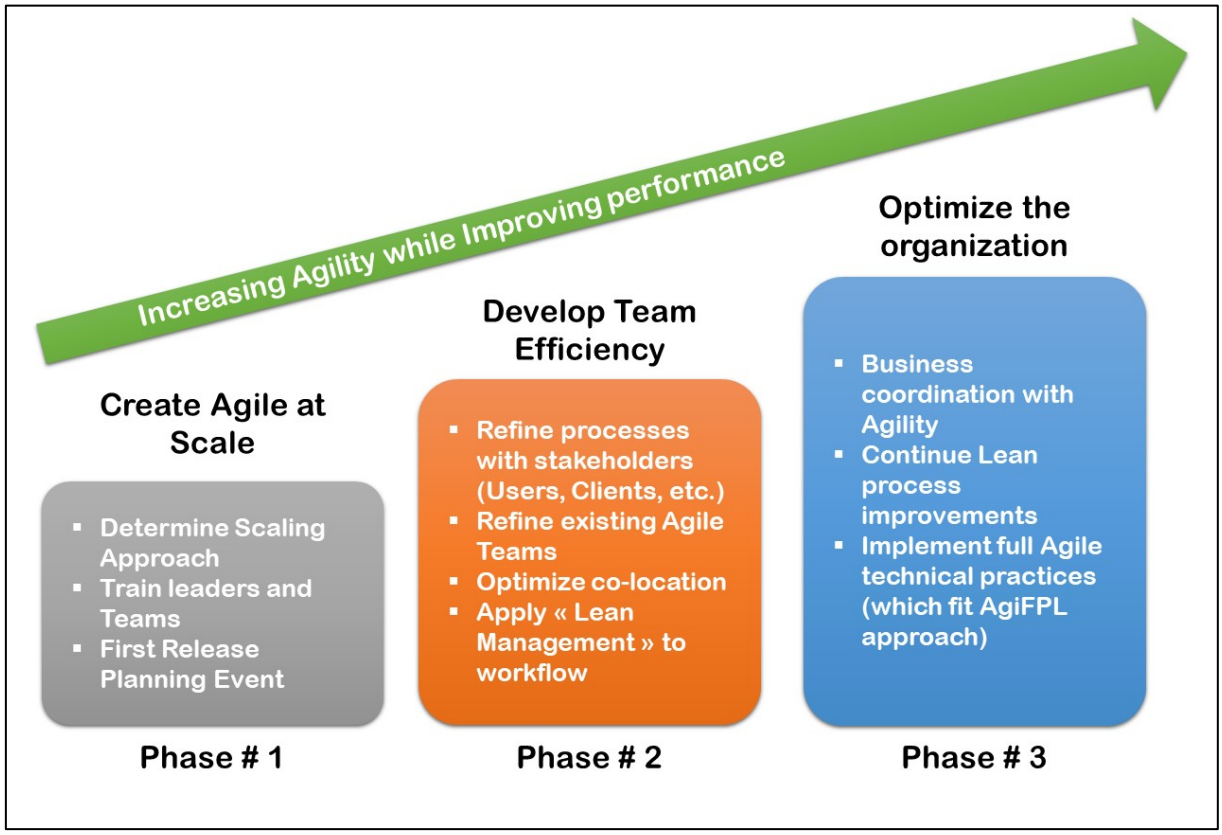

Fig. 6.17 Three-phases approach to incorporate AgiFPL. Adapted from [Gregg et al., 2016].

that meet the expectations of AgiFPL and fit its nature. In fact, Gregg et al. have proposed a practical case study that shows how Lockheed Martin has integrated Agile Software Development (ASD) within its Software Product Lines.

As shown in Figure 6.17, the AgiFPL approach has to be implemented *incrementally*. The period of implementation depends on the size of the Product Lines. The first phase of implementing AgiFPL concentrates on risk reduction, training, and release planning. The second phase continues to rollout and begins to optimize workflows and team organizations, that were put into place in the first phase. In addition, it also begins to re-define some of the customer interactions that need to change, such as adapting to finer-grained scheduling and budgeting and providing more embedded feedback. Finally, the third phase continues the rollout, incorporates business coordination, and transitions from introduction to optimization.

The time of each phase of the proposed approach depends on the type of the SPL Company (or organization). For example, for small-scale SPLs each phase could takes from 3 months to 6 months. However, for large-scale SPLs, each phase may take about 1 year. The structure of the SPL organization and its market environment are the main factors used in addition to other technical factors in order to define the time of each phase.

Following Gregg et al. [Gregg et al., 2016], in order to bridge gaps between classical SPL and small-scale short-iteration Agile Software Development (ASD), the following steps could launch the start:

1. **Small projects**: ASD, in general, is about the lean execution of development projects. With SPLs, projects are associated with component products. That is, the projects are already defined. The goal, then, became making those projects Agile. That in turn involved transforming the project teams into small, Agile teams working in short iterations to support small-scale tasks that, taken as a whole, sum up to the purpose and work of the original product teams.

2. **Small teams**: Small teams resulted from decomposing the products' large teams into AgiFPL teams of seven to ten members. The teams, considered together, do the same work as they did before, but the work has also been divided into smaller bite-sized chunks that match the team size and short iteration (sprint) schedule. The teams are self-organized by product, mostly around areas of domain expertise or functionality, or around specific elements in the architecture. So far so good, decomposing large teams into small ones, and large tasks into small tasks, is relatively straightforward.

3. **Small Iterations**: The work for each team follows Scrumban practices in DE and Scrum practices in AE, meaning that planned work is organized into two-week sprints. A certain number of sprints add up to a release. The work accomplished in each sprint is defined in terms of User Stories. A user story is a *"developer-sized"* task that can be implemented and tested in one or two days [Kenneth, 2013]. It constitutes the smallest capability that has business value. User stories may be decomposed into tasks, which generally take 2-8 (in the worst case, 16) hours to complete. Users stories are used to accomplish features. A feature is a sort of large-scale user story that often applies to an entire release. A feature is accomplished within a release cycle, and often spans teams in its scope. Features in turn constitute epics, a capability that spans multiple releases (perhaps even over multiple years) and often involves the addition of major capability to the product line.

Operationally, each product team takes a program feature and decomposes it into features that affect their product. These product features are then decomposed into user stories and tasks. These constitute the team's backlog. The work under this approach is by and large what it would have been under the "pure" SPL approach, except that now it is decomposed into assignments that can be agilely managed with greater precision and predictability, and allowing for a much finer-grained prioritization [Gregg et al., 2016].

6.6 Discussion

By implementing the AgiFPL method, short production flow for Domain Engineering and short sprints for Application Engineering were established. This has provided faster and more frequent feedback, direct communication channels, and opportunities for improvements. For example, some unseen variabilities could be identified and communicated from the first iterations.

The established agile development teams of AgiFPL could easily identify problems related to the incremental development of reusable artifacts and final products. Thus, some refactoring may occur in the future iterations.

The issues related to traceability were addressed implicitly through the practices of Scrumban and Scrum. In addition, these practices were completed by the proposed Requirements Engineering practices proposed in Chapter 8. The issues related to Product Line Architecture are partially covered by AgiFPL and could be completed by the approach proposed by Díaz et al. [Díaz et al., 2014], who have proposed a Flexible-Product Line architecture modeling framework that supports successful development and evolution of PLAs. The use of the approach proposed by [Díaz et al., 2014] is recommended as assistant process that complement AgiFPL in case of large-scale SPL organizations.

The processes provided by AgiFPL for DE and AE establish continuous feedbacks and adaptations through daily meetings, retrospectives, reviews and andons. In fact, AgiFPL has facilitated feedbacks sharing among all people involved in the projects. For example, technical problems regarding the implementation may be solved faster and precisely during the daily meetings. Moreover, the retrospectives meetings, established by AgiFPL, have ensured easy sharing of problems and solutions regarding communalities and variabilities for all products.

The AgiFPL method takes into account the maintenance activities. However, the established processes need to be completed by more explicit maintenance-oriented practices in future works.

6.7 Conclusion

Agile Product Line Engineering (APLE) is a new emerging paradigm that aims to reduce the upfront design, as currently practiced in classical Software Product Line Engineering (SPLE), and to transform the development of product-lines to more flexible, and adaptability in order to to be open to changes. Designing and implementing an APL method was not easy. However, studying the current APL methods and identifying their strengths and weaknesses have allowed to design a powerful APL method that has taken advantages from the previous APL methods and has overcome most of their weaknesses.

In this chapter, we have presented a feature-oriented APL method called “Agile Framework for managing evolving Software Product Lines: the AgiFPL method”. The different

processes of AgiFPL were presented as well as the roles and the practices defined by AgiFPL. In addition, an approach to implement AgiFPL method in practice was presented in this chapter.

The contribution of this chapter has been published in the following conference paper: 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.

7 Assessing the adoption level of agile development within Software Product Lines: the AgiPL-AM model

Abstract. *The AgiPL-AM model* is an assessment model for assessing 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, an assessment model was built called AgiPL-AM, allowing self-evaluations within the team in order to determine the current state of agile software development in combination with Software Product Lines. AgiPL-AM, the model for assessing agility attribute of Agile Product Line approaches, is comprised of six categories (five are related to agile principles and one to product-line architecture) and five levels of maturity. The assessment results demonstrate that AgiPL-AM has the ability to reveal and pinpoint agile product-line approach strengths and weaknesses. It makes recommendations to improve the status of "agility" and may give a guideline for this improvement.

7.1 Introduction

As stated earlier in this thesis, 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 to combine agile and product lines techniques [da Silva et al., 2011]. The goal was to make software product-line methodologies evolve from predictive to iterative, incremental, and agile.

With the adopted definition of APLE (See Definition 6.1), we try to propose a point of view in which, an organization becomes agile while preserving its Software Product Lines, or an agile organization boosts its agility by integrating SPL principles. Therefore, *an organization needs to know when it has become "agile".* The literature on APLE does not define any threshold (point) after which an organization has adopted enough practices to be called *"AGILE".*

At this stage, many questions could be asked about the conducted combinations, their results, and their effectiveness. Since "agility" is considered as a "quality attribute" of the development process, the crucial research question tackled is this chapter is the following question:

"How to assess the agility attribute of an agile product line method?"

This chapter proposes an assessment model to determine the current state of agile development in combination with software product lines. In fact, assessing the status of the development is a crucial step for a successful combination of agile methods and Software Product Lines. Thus, through this chapter, we propose an assessment model called AgiPL-AM, which allows self-assessments within the "Domain and Application teams" in order to determine the current state of Agile Software Development (ASD) in the context of Software Product Lines (SPL).

To develop the targeted assessment model a research process of three phases was followed. The first phase reviews the literature on maturity models that concern Software Product Line Engineering, Agile Product Lines, and Agile Software Development. The desired assessment model is built in the second phase. Finally, the third phase applies and evaluates the proposed model.

The obtained model (i.e. AgiPL-AM) is an agility assessment model for assessing agility of Agile Product Line approaches that comprises six categories (i.e. five categories are related to agile principles and one category is related to product line architecture) and five levels of maturity. To build the assessment model, an existing agile maturity model (SAMI model [Sidky et al., 2007]) was used as a basis.

7.2 Assessment models for software reuse strategies

Over the past years, different assessment models were proposed to assess software reuse. Hereafter, we present three of them. The CMMI-DEV model [TEAM, CMMI Product, 2010] provides a collection of the best practices that support organizations to improve their processes. It focuses on activities to develop products that meet needs of customers and a well-known standard that defines methods to evaluate complete process models and organizations.

Based on CMMI, Jasmine and Vasantha [Jasmine and Vasantha, 2010] have defined the Reuse Capability Maturity Model (RCMM). RCMM model focuses on a well-planned and controlled reuse-oriented software development. This model comprises maturity levels that denote an achieved level in the evolution to a mature reuse process.

The "VDA QMC Working Group" has proposed the Automotive SPICE model [VDA QMC, 2017]: a process assessment model that contains a set of indicators to be considered when interpreting the intent of the Automotive SPICE process reference model. These

indicators may also be used when implementing a process improvement program subsequent to an assessment.

7.3 Assessment models for agile development

In practice, organizations are unable to fully adopt agile development practices immediately or over a short period, asit requires a socio-technical transformation or migration process [Qumer and Henderson-Sellers, 2008]. Accordingly, maturity models can help and guide organizations in providing the directions concerning the practices and the manner that they can be introduced and established in the organization.

Schweigert et al. [Schweigert et al., 2013] have identified several maturity models for agile development. They use a set of assessment criteria to assess each identified maturity model in term of their fitness of purpose, completeness, definition of agile levels, objectivity, correctness, and consistency. With these assessment criteria, they surveyed the related issues (e.g. *whether the emphasis of the model is on assessing agile practices or not (i.e. Fitness of purpose)*). Following these criteria, they have concluded that the SAMI model (Sidky Agile Measurement Index) proposed by Sidky et al. [Sidky et al., 2007] has the highest scores among the studied models. SAMI consists of two components. The first one is an agile measurement index and the second one is a four-stage process. Together, these two components guide and assist the agile adoption efforts of organizations.

SAMI is structured into four main parts: agile levels, agile principles, agile practices and concepts, and indicators. Driven from the values and principles of the "Agile Manifesto" [ManifestoAgile, 2001], the model defines five agile levels:

- **Level 1:** is dedicated for the *"Collaboration which is one of the essential values and qualities of agile"*;
- **Level 2:** represents the objective of *"Developing software through an evolutionary approach"*;
- **Level 3:** represents the objective of *"Effectiveness and efficiency in developing high quality software"*;
- **Level 4:** is depicted for the objective of *"gaining the capability to respond to change through multiple levels of feedback"*;
- **Level 5:** represents the objective of *"Establishing a vibrant and all-encompassing environment to sustain agility"*.

In addition, the SAMI model has clustered the 12 agile principles into five categories that group the agile practices. These categories are:

1) Embracing change to deliver customer value;
2) Plan and deliver software frequently;
3) Human-centricity;
4) Technical excellence;

5) Customer collaboration.

In total, SAMI incorporates 40 agile practices. Organization should start adopting agile practices on lower levels first, because the agile practices on a higher level are dependent on the practices introduced at the lower levels [Sidky et al., 2007]. Moreover, Sidky et al. [Sidky et al., 2007] have proposed a range of indicators that are used to assess certain characteristics of an organization or project, such as people, culture, and environment, in order to ascertain the readiness of the organization or project to adopt an agile practice [Schweigert et al., 2013]. The SAMI model contains about 300 different indicators for the 40 agile practices [Sidky and Arthur, 2006].

7.4 Research approach

To reach the main target of this chapter, the research procedure was inspired by the work of Hevner et al. [Hevner et al., 2004]. Since the primary objective is to propose an assessment artifact that could be used to assess the adoption level of agile development within Agile Software Product Lines, the followed research method mainly involves the three phases. The first phase is dedicated to the definition of the problem and the objectives of the assessment artifact. The second phase is devoted to the design and the development of the targeted assessment artifact. The third phase illustrates the applicability of the proposed model.

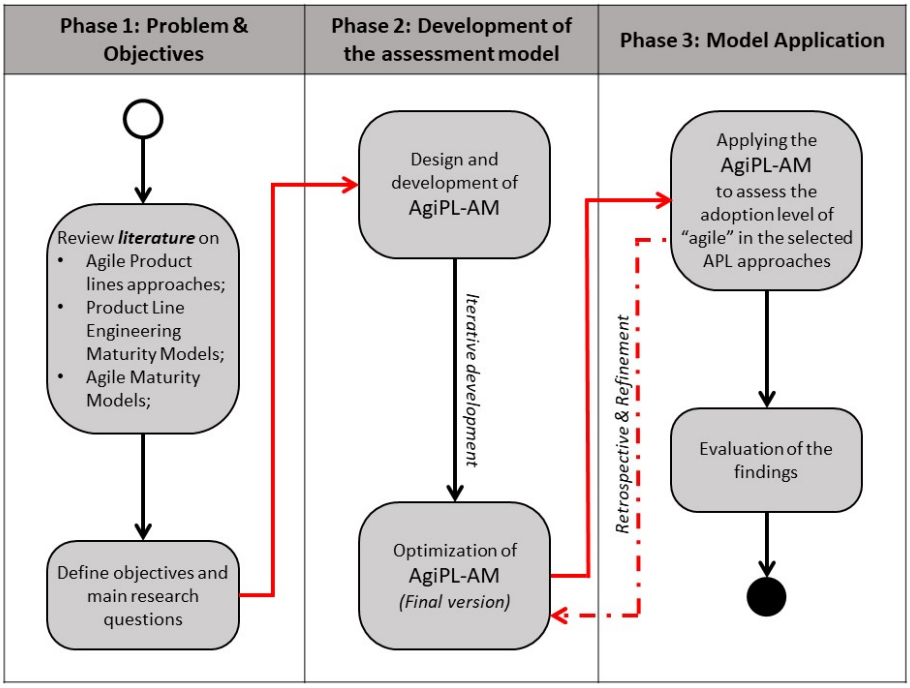

Fig. 7.1 Research procedure.

Figure 7.1 depicts a detailed view on the procedure followed in order to develop, apply, and evaluate the Agility Assessment Model (i.e. AgiPL-AM) proposed in this chapter. 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 AgiPL-AM was designed and developed in an iterative way. In the third phase, the model was applied and evaluated. At this stage, the model was reviewed and refined in order to optimize and finalize AgiPL-AM.

7.5 Assessment Model for Agile Product Lines: AgiPL-AM

In order to attempt the target of this thesis, an extensive review of agile product line approaches, relevant case studies, and software process oriented-maturity models was performed, with emphasis on the agile software development approaches and on the agile product line approaches. It was identified that seven important areas need to be considered in an assessment for the combination of agile development and software product lines. According to Hohl et al. [Hohl et al., 2018], these areas are the following:

1. *Product Line Architecture:* the objective of this area is to provide a suitable software architecture to enable the implementation of several software variants for different products with a high degree of software reuse;

2. *Domain Requirements Engineering:* the purpose of this area is to identify the reuse assets that should be developed in a software product line. This includes the identification of products and features that should be part of the product line and the definition of common and variable features;

3. *Agile Software Development:* the main target of this area is to react faster to customer needs and legal constraints to reduce the time-to-market for innovative feature upon a simultaneous increase of the quality of software;

4. *Continuous Execution:* the objective of this area is to continuously execute tasks that lead to a more stable, compliant, and better products;

5. *Continuous Model Improvement:* the purpose of this area is to continuously reflect on the assessment model and improve the interaction of assessment results and the suggested improvement for the software development process;

6. *Test Strategy:* the main purpose of this area is to provide an environment that allows for the verification of the correct behavior and ensures the software quality for various software variants that are developed in a fast development pace within the software development;

7. *Communication:* the objective of the Communication Area is to verify the communication of all participating roles to avoid knowledge silos and to react on customer requirements faster.

Considering these areas, the review has led us to take the SAMI agile maturity model [Sidky et al., 2007] [Sidky and Arthur, 2006] as a basis to develop the proposed model. Ozcan-Top and Demirörs [Ozcan-Top and Demirörs, 2013] have confirmed that the SAMI model is comprehensive and well-organized structure, an argument further confirmed in [Turetken et al., 2017]. By reviewing the agile practices offered by the SAMI model and evaluating their applicability, SAMI can be viewed as providing agile practices that the Agile Product Lines (APL) require at several levels. Therefore, these agile practices were adopted as the basis for the targeted maturity model to address the "APL team level practices". In fact, we have adopted the 40 original SAMI agile practices. Then we have adapted and extended the SAMI model in accordance with the Agile Product Line principles and practices defined in the main sources of Agile Product Line Engineering such as [da Silva et al., 2011], [Díaz et al., 2011], and [Farahani and Ramsin, 2014].

7.5.1 Development of AgiPL-AM

In addition to the 38 original SAMI agile practices, we[10] defined 49 Agile Product Line practices that are incorporated in the final version of the AgiPL-AM model. Precisely, both the SAMI agile practices and the APL practices underwent a review and refinement by using the phase two of the research approach. These refinements and changes were applied with respect to the original agile practices of the SAMI model.

Considering the areas presented above, it was identified that the SAMI model covers mainly the area 3 (i.e. Agile Software Development) and partially the area 2, area 4, area 5, and area 6. In the proposed model we have defined a new category called "Category 6 – Product line Architecture". Moreover, in our proposed assessment model we have conserved the five agile levels of SAMI model. Hereafter we presented the different added practices. Due to lack of space, we have presented the new defined agile practices for each category separately.

Table 7.1[11] presents the agile practices as well as the APL practices defined within the AgiPL-AM model, for each level and each category. In addition, it presents the code of each practice, as well as the description of each practice.

[10] The "we" here refers to the author and two co-authors that have participated in the research work.
[11] (Due to the size of the Table 7.1, it was placed on the next page).

Table 7.1 AgiPL-AM: Levels, Principles, and Practices.

Maturity Level	Principle	ID	Practice	Description of Practice
Level 1 Collaborative	Category 1	L1.C1.1	Reflect and tune process	Holding retrospectives at regular intervals of the development process. The objective of this practice is to overcome process challenges that have been faced thus far [Sidky, 2007].
	Category 2	L1.C2.1	Collaborative planning	Collaborative Planning encourages all stakeholders to come together during the planning phase. The collaborative efforts increase visibility, loyalty, and acceptance and buy in from all stakeholders. [Sidky, 2007]
	Category 3	L1.C3.1	Empowered and motivated teams	Supervisors must empower and equip the development teams with the authority to make decisions on their own. This authority helps motivate the team members to solve problems and tasks on their own. [Sidky, 2007]
		L1.C3.2	Collaborative Teams	Team members must communicate and cooperate with each other and other teams. [Sidky, 2007]
	Category 4	L1.C4.1	Product Backlog	The product backlog is a repository for all the upcoming work, which is anticipated to be delivered. The product backlog can be utilized at different levels of granularity, e.g. team, program and portfolio backlog [Turetken et al., 2017].
		L1.C4.2	Coding standards	Collaboration through code by creating a common language (coding standards) among developers. Coding standard increase the comprehension and ease of sharing code among team members. [Sidky, 2007]
		L1.C4.3	Knowledge sharing	Knowledge sharing tools enhance collaboration by helping to document and maintain the information and knowledge exchange among team members. Knowledge sharing tools can be electronics (e.g. wiki, blogs) or simple whiteboards and walls. [Sidky, 2007]
		L1.C4.4	Task volunteering	During the planning meeting the developers should volunteer for tasks, rather than tasks being assigned to them by managers. This practices increases motivation, job satisfaction and quality of performance from team members. If there are no volunteers, the team should take collective responsibility to complete the task. [Sidky, 2007]
		L1.C4.5	Continuous and automated tasks	The software development process is supported by continuous and automated tasks, where, supporting tasks are implemented. The structure has to ensure that supporting and repetitive tasks are comprised in the development process and an automated feedback is provided. [Hohl et al., 2018]
		L1.C4.6	Acceptance testing	Acceptance tests are functional tests that verify that the system implements the story as intended. To avoid large volume of manual tests they are automated wherever possible. [Turetken et al., 2017] [Leffingwell, 2011]

Maturity Level	Principle	ID	Practice	Description of Practice
	Category 5	L1.C5.1	Customer commitment to work with development team.	The customer has to commit to working with the development team in order to establish collaboration with development team [Sidky, 2007]. Moreover, involve all parties affected by the development and ensure that all involved parties are involved directly in the development process [Hohl et al., 2018].
		L1.C5.2	Destructed "Knowledge silos"	Break down knowledge silos and ensure that the development process is transparent for every participant to avoid knowledge silos. [Hohl et al., 2018]
		L1.C5.3	Fast feedback channels	Fast feedback loops are established and the structure ensures that fast feedback loops are established to provide a faster learning loop. This includes automatic generated feedback from test systems, feedback from customer, and feedback from all affected parties [Hohl et al., 2018].
		L1.C5.4	Common understanding for the SPL	Ensure that a common understanding for the SPL is developed and ensure that all involved parties are brought together as good as possible. In addition, ensure that all affected parties be aware of the development within the software product line. All affected parties are mutually responsible to identify impediments, dependencies, incompatibilities and similarities of software features. [Hohl et al., 2018]
	Category 6	L1.C6.1	Software Product Line architecture	Ensure that a software product-line architecture is defined. Ensure that a software product-line is used for development and that the software product line architecture is able to generate various software variants for different products and different markets. [Hohl et al., 2018]
		L1.C6.2	Reuse strategy	The reuse strategy enforces the creation of reusable artifacts in Domain Engineering and the reuse of these artifacts in Application Engineering [Pohl et al., 2005].
		L1.C6.3	Modular software architecture	The software architecture shall be modular. It should ensure that parts of the software could be replaced or reused without affecting other parts of the software, by means of loosely coupling between software units [Hohl et al., 2018].
Level 2 Evolutionary	Category 1	L2.C1.1	Evolutionary requirements	Requirements should be evolving iteratively rather than being fully developed in one major specification effort. The requirements in agile tend to evolve and change based on customer feedback and thus this agile practice is important [Sidky, 2007].
		L2.C1.2	Domain Requirements	Domain requirements encompass requirements that are common to all applications of the software product line as well as variable requirements that enable the derivation of customized requirements for different applications [Pohl et al., 2005].

Maturity Level	Principle	ID	Practice	Description of Practice
		L2.C1.3	Smaller, more frequent releases	Agile teams should deliver smaller and more frequent releases to customers and end users. This provides increased responsiveness and reduces risks to the enterprise. This practice does not specify how often an organization should schedule a release that is the purpose of a similar practice at Level 4. [Leffingwell, 2007]
		L2.C1.4	Requirements discovery	Agile practice which is used to better understand what needs to be build and why. There are variety of software requirements techniques which are recommended and can be used by teams e,g, Spikes, Use-Case modeling, User Experience Mock-ups, etc. [Leffingwell, 2011]
	Category 2	L2.C2.1	Continuous delivery	Delivering software in small iteration at regular intervals. These regular intervals ensure that the team has to divide the product for each iteration. At this level, the duration of the iteration is irrelevant but it is later addressed at level 4. [Sidky, 2007]
		L2.C2.2	Two-tier planning and tracking (i.e. Domain Engineering tier & Application Engineering tier)	Ensure that there are planning activities that concern the Domain Engineering process and the Application Engineering process. Ensure that the planning and analyzing processes of DE are sub-processes of the domain of a product line and of developing reusable artifacts (i.e. planning for the development of reusable artifacts). In addition, ensure that the planning and analyzing sub-processes of AE have the goal of developing a specific product foe the needs of a particular customer or other stakeholder (i.e. planning for development with reusable artifacts) [Apel et al., 2013].
		L2.C2.3	Two-level planning and tracking	This agile practice suggests development of iteration planning and release planning. Iterations plans are with short-term focus and are estimated with great precision and confidence. Releases have long-term focus and are more coarse-grained, less comprehensive and less precise. [Leffingwell, 2007]
		L2.C2.4	Agile Estimating and Velocity	Agile estimating is a practice for estimating the workload to be delivered and is critical to the productivity and reliability. Estimating is done using story points. The number of story points a team delivers per iteration defines the team velocity. This velocity estimate can then be used for estimating schedule and estimating cost. [Leffingwell, 2011] [Leffingwell, 2007]

Maturity Level	Principle	ID	Practice	Description of Practice
		L2.C2.5	Release planning	Release planning is a very high-level plan for multiple Sprints (e.g. three to twelve iteration). A guideline reflects expectations about which features will be implemented and when they are completed. It also serves as a base to monitor progress within the project. Releases can be intermediate deliveries done during the project or the final delivery at the end [Cohn, 2013].
		L2.C2.6	Work product list	Ensure that exit a "Work Product list" that consists of historical records of all changes made to an object (i.e. document, file, feature, software component, etc.) and which indicates: – the ability to record customer and process owner information, – the ability to record related system configuration information, – the ability to record information about problem or action plan, – the ability to record proposed resolution or action plan, – and the ability to provide management status information [Hohl et al., 2018].
	Category 3	L2.C3.1	The Define/Build/Test teams of each tier: Domain Engineering tier & Application Engineering tier	A cross functional group of individuals that has the ability and authority to Define (elaborate and prioritize requirements and design the solution), Build (write the code) and Test (run tests cases) – all in a short iteration time-box. The define/build/test team is self-organize, empowered and self-managing and thus depends on practices from lower level. [Cohn, 2013]
	Category 4	L2.C4.1	Software configuration management	SCM tools help control the different versions of the software being developed. [Sidky, 2007]
		L2.C4.2	Tracking iteration progress	Agile practice which is concerned with the team having the means by which they can measure the progress of the development effort within each iteration. This concept does not dictate a particular method to fulfill this tracking but its latter defined at level 4 (Daily progress tracking meetings). [Sidky, 2007]
		L2.C4.3	No big design up front	Agile practice which ensures that the product is being developed using an evolutionary approach. BDUF is where a "big" design is created before coding and testing takes place and is typical for waterfall development process. In agile design occurs throughout the development process. [Sidky, 2007]
		L2.C4.4	Automated testing	The individual priority of test cases is determined in a regular period by the test strategy and suitable test cases are automatically executed. [Hohl et al., 2018]

Maturity Level	Principle	ID	Practice	Description of Practice
		L2.C4.5	Bidirectional traceability record	The consistency and bidirectional traceability between software requirements, software units and software architecture is supported by a toolchain. Accordingly, manage the SPL by means of suitable tools that Ensure traceability and consistency between software requirements, software units and the software architecture by means of suitable tools [Hohl et al., 2018].
	Category 5	L2.C5.1	Customer contract reflective of evolutionary development	The customer understands the evolutionary nature of software development and the contract reflects this evolutionary approach. This practice prevents the contract to define the dates when milestones should be completed but it is rather reflecting of the evolutionary approach. [Sidky, 2007]
	Category 6	L2.C6.1	Distributed development	Software product-line architecture supports a distributed development, where only one software product line is used for the worldwide development of a product [Hohl et al., 2018].
		L2.C6.2	Modularity of software components	A high modularity of software components is achieved. The software architecture shall be modular. Ensure that parts of the software can be replaced or reused without affecting other parts of the software, by means of loosely coupling between software units. [Hohl et al., 2018]
		L2.C6.3	Reusable software units	Reusable software units shall be defined. Ensure that software parts, which can be reused, are identified and required new functionality is first checked against existing functionality to ascertain that the functionality is not implemented already in another software variant, by means of scoping meetings. A high amount of reusable software elements shall be achieved [Hohl et al., 2018].
Level 3 Effective	Category 1	L3.C1.1	Regular reflection and adaptation	Teams adopt regular reflection and adaptation at each iteration and release levels. These reflections are accompanied by quantitative assessment that measure iteration and release metrics. [Leffingwell, 2011]
		L3.C1.2	Customer feedback is accessed for new features and ideas.	Consider ideas and desires from customers. Ensure that the customer gets the possibility to enter new innovative features for the software development in an easy way. [Hohl et al., 2018]
	Category 2	L3.C2.1	Risk driven Iterations	Risk driven iterations help tackling risk elements as early as possible. Mitigating these risks early ensures that the project team does not spend a considerable amount of time building a system that they cannot complete. By catching these issues, the development becomes more effective. [Sidky, 2007]

Maturity Level	Principle	ID	Practice	Description of Practice
		L3.C2.2	Plan features not tasks	Customer expresses their needs in terms of features, so when the feature changes the impact on the related tasks is minimized. The planning should be done in terms of features in order to prepare the development process for Client Driven Iterations at Level 4. [Sidky, 2007]
		L3.C2.3	Mastering the iteration	The base construct of agile and iterative development is the iteration- the ability of a team to create working, tested software in a short time-boxed interval. The aim of this agile practice is effective production of an increment of working software at each iteration. [Turetken et al., 2017] [Leffingwell, 2007] In essence, this practice validates of the effectiveness of the practices established on lower levels.
		L3.C2.4	DoD after each software release	After each release, the DoD has to be defined. Conceptually the definition of done is a checklist of the types of work that the team is expected to successfully complete before it can declare its work to be potentially shippable. [Cohn, 2013] The specific items on the checklist will depend on a number of variables: – The nature of the product being built – The technologies being used to build it – The organization that is building it – The current impediments that affect what is possible
		L3.C2.5	Backlogs and Kanban Systems	The Kanban system describes four queues that an epic passes through on the way to implementation: Funnel, Backlog, Analysis, and Implementation [Leffingwell, 2011]. The Kanban Systems are used for visualizing workflow, limiting the work in progress, measuring and managing flow, making process policies explicit and using models to recognize improvements [Leffingwell, 2007].
	Category 3	L3.C3.1	Self-organizing teams	Team is empowered to make decisions without waiting for approval from management. Teams are cross-functional, roles, responsibilities of team members are not or loosely defined, and the whole team is responsible for delivery of working software. The importance of self-organizing teams is highlighted in the Agile Manifesto which states that the best architectures, requirements and designs emerge from self-organizing teams. [Sidky, 2007]
		L3.C3.2	Frequent face-to-face communication	For establishing efficient and effective development process frequent communication among team members is necessary. [Sidky, 2007]

Maturity Level	Principle	ID	Practice	Description of Practice
	Category 4	L3.C4.1	Continuous deployment	Ensure that continuous deployment approach is set up, in order to ensure that software is deployable (regarding all types of constraints) to the field and the infrastructure to deploy the software is available [Hohl et al., 2018].
		L3.C4.2	Continuous integrating	Continuous integration is an agile that encourages members of a team to integrate their work frequently. It is preferred that an automated build tool verifies each integration in order to detect any integration errors as quickly as possible. In addition, Continuous integration is applied and failures in integration are identified earlier in the development process, in order to ensure that software failures in all software variants are identified in the integration phases due to the use of timed and automated continuous integration [Sidky, 2007].
		L3.C4.3	Scalable and Continuous tests	Ensure that continuous software tests are applied and failures in the software are identified earlier in the development process. Ensure that software failures are identified by timed and automated software tests. Moreover, a scalable test strategy for different test hierarchies is implemented, including software variants, systems and developed applications is realized [Hohl et al., 2018].
		L3.C4.4	Continuous compliance	Continuous compliance is set up in order to ensure that each software release is (potentially) compliant. In addition, verify that the software development process ensures that all the related constraints are abided by verifying continuous compliance [Hohl et al., 2018].
		L3.C4.5	Continuous improvement (refactoring)	Refactoring is an essential practice to be adopted at level 3 because of the evolutionary development process assumed at Level 2. Refactoring involves rewriting the code to improve its structure while preserving the behavior. In general refactoring focuses on removing code duplication. [Sidky, 2007]
		L3.C4.6	30% of “level 2” and “level 3” people	Cockburn people levels are directly related to the amount of experience the developer has. Cockburn identified three levels of understanding when they approach new material and has argued that a person level of understanding is directly linked to his experience in the domain. At Level 3 the team needs developers who can handle new unexpected problems and that is why Cockburn Level 2 and Level 3 people are needed. [Sidky, 2007]
		L3.C4.7	Unit tests	Unit tests are code procedures used to validate that the individual units of source code are working properly. A unit of source code is the smallest testable part of an application. It provides a strict, written contract that the code must satisfy. It is recommended that unit tests be automated. [Sidky, 2007]

Maturity Level	Principle	ID	Practice	Description of Practice
	Category 5	L3.C5.1	Direct communication channels	Direct communication channels are established in order to overcome hierarchies and ensure that the communication is not blocked by the hierarchy and every participant can speak to every other directly [Hohl et al., 2018].
		L3.C5.2	Vertical commitment	Ensure that a vertical commitment in the company is established and all involved parties are involved in the development process [Pohl et al., 2005] [Hohl et al., 2018].
	Category 6	L3.C6.1	SPL architecture is open to changes and refactoring is possible	The software architecture shall be open to change. In fact, the assessor should ensure that the SPL architecture is not a monolithic architecture, by means of a modular architecture structure that is capable to manage loosely coupled software units [Hohl et al., 2018].
		L3.C6.2	Collaboration with supplier is improved	Ensure that a good and open-minded collaboration with the supplier is established and the supplier is included in the process [Hohl et al., 2018].
Level 4 Adaptive	Category 1	L4.C1.1	Client driven iterations	The client is in control and can request and prioritize features per iteration. The client steers the project, iteration by iteration, requesting the features that are of highest business value to them. [Sidky, 2007]
		L4.C1.2	Continuous customer satisfaction feedback	Continuous customer feedback is crucial to ensure the customer is satisfied with what is being developed. If customer feedback is only sought at the end of the project, then there is a significant risk that what has been developed ins not what the customer needed [Sidky, 2007].
		L4.C1.3	Lean requirements at scale	For larger organizations, it becomes increasing more difficult to make the whole team working toward a common purpose. There is a need to create a scalable requirements pattern consisting of vision, roadmap and just-in-time elaboration as a method that brings the benefits of agility to larger scale teams. [Leffingwell, 2007]
	Category 2	L4.C2.1	Smaller and more frequent releases	This encourages organizations to keep small timeframe for the releases and limit them to 8 weeks. Usually building a release will include multiple iterations. Having shorter releases helps the development process to embrace change. [Sidky, 2007]
		L4.C2.2	Adaptive planning	Adaptive planning delays developing the iteration details until immediately before the following iteration, and therefore incorporates all the feedback obtained including what is learned during previous iteration. Adaptive planning helps organizations embrace change because the focus shifts from following a plan to adaptive planning based on latest feedback. [Sidky, 2007]

Maturity Level	Principle	ID	Practice	Description of Practice
		L4.C2.3	Measuring business performance (Project measure; Quality measure; Risk measure; Delivery record)	Implementing a flexible, automate and meaningful BSC (balanced scorecard) for the enterprise that measures performance in terms of efficiency, value delivery, quality and agility [Leffingwell, 2007]. In addition, the efficiency of the development process would be continuously evaluated to monitor development efficiency. Moreover, assessor should ensure that metrics are selected to measure the efficiency and the implications on the process when suggestions for improvement are introduced [Hohl et al., 2018].
	Category 3	L4.C3.1	Managing highly distributed teams	Large corporates are distributed. They should be managed with proper communication and the necessary networking and tooling architecture. [Leffingwell, 2007] [Leffingwell, 2011]
	Category 4	L4.C4.1	Daily progress tracking meetings	This is a more specific version of tracking iteration progress practice which was introduced at Level 2. This practice emphasizes that the team should be informed on a daily bases regarding the status of the iteration. [Sidky, 2007]
		L4.C4.2	User stories	An agile practice where software requirements are formulated as one or two sentences in the every-day language of the user. Users stories are a quick way of handling customer requirements with minimal amount of documentation. [Sidky, 2007]
		L4.C4.3	Adaptive test strategy	The test strategy ensures that failures and bugs in the software units are identified as soon as possible in the development process. Ensure that always the best fitting test cases for one software variant are executed at the beginning of a test, as these foster a rapid error detection [Hohl et al., 2018].
		L4.C4.4	Improvement opportunity and plan	Ensure that exist procedures and applications for identifying improvement opportunities and that allow the planning of the improvement plans [Hohl et al., 2018]. *Improvement opportunity procedure that help to:* – Identify what the problem is – Identify what the cause of a problem is – Suggest what could be done to fix the problem – Identify the value (expected benefit) in performing the improvement – Identify the penalty for not making the improvement

Maturity Level	Principle	ID	Practice	Description of Practice
				Improvement plans presents: – Improvement objectives derived from organizational business goals – Organizational scope – Process scope, the processes to be improved – Key roles and responsibilities – Appropriate milestones, review points and reporting mechanisms – Activities to be performed to keep all those affected by the improvement program informed of progress
	Category 5	L4.C5.1	"CRACK" Customer immediately accessible	The customer is Collaborative, Representative, Authorized, Committed and Knowledgeable. At this level the concern is not with the location of the customer but rather how responsive they are. [Sidky, 2007]
		L4.C5.2	Customer contact revolve around commitment of collaboration	The customer agrees to contract the degree and amount of collaboration and not the requirements and features. This is one of the ultimate factors that enables organizations to embrace change. [Sidky, 2007]
		L4.C5.3	DevOps (Integrated Development and Operations)	An agile practice that is used to ensure a faster flow of value to the user by tighter integration of development and operations. This is accomplished by integration personnel from operations into the agile teams or by continuously maintaining deployment readiness. [Turetken et al., 2017]
		L4.C5.4	Vision, features	Describes the stakeholder's view of the solution to be developed in terms of stakeholder's needs and proposed features. It captures the essence of the envisaged solution in the form of high-level features, non-functional requirements and design constraints, and provides an overview of the system to be developed. [Turetken et al., 2017]
		L4.C5.5	Impact on customers and operations	Measuring the impact on sales, operations and customer due to the changes in the development model. [Turetken et al., 2017] [Leffingwell, 2011]
	Category 6	L4.C6.1	Fast changes in requirements	Fast changes in requirements are supported by the software architecture. The software architecture shall provide the possibility to include fast changing requirements. Ensure that the architecture of the software can deal with fast changing requirements, by means of standardized interfaces between software units and a modular architecture [Hohl et al., 2018].

Maturity Level	Principle	ID	Practice	Description of Practice
		L4.C6.2	Adequate scoping process	Ensure that an adequate scoping process is established that ensure that scoping meetings are implemented. They are implemented in a recurrent and defined timeframe to prioritize and categorize new required software parts. In the scoping meetings, it is determined, if the feature is part for the common baseline or a product/market specific feature [Hohl et al., 2018].
		L4.C6.3	Intentional architecture	Intentional architecture is a set of purposeful, planned architectural initiatives to enhance solution design, performance and usability and provides guidance for inter-team design and implementation synchronization [Leffingwell, 2007]. Architecture that has been planned to some extent, has been built incrementally, has emerged over the course of prior iterations or has evolved to adequately support the needs of the customers [Leffingwell, 2011].
Level 5 Encompassing	Category 1	L5.C1.1	Low process ceremony	Lower process ceremony allow for being responsive to changes. The changes don't have to be approved by a least 3 levels of management. Process ceremony is also the level of paperwork involved in the process. [Sidky, 2007]
	Category 2	L5.C2.1	Agile project estimation	Under the "Planning and Deliver Software Frequently" agile principles most practices are related to planning. Agile estimation is important since plans are only as good as the estimates they are based on. [Sidky, 2007]
		L5.C2.2	Audit activities	Activities that propose suggestions to improve the software development process. Audit activities that ensure that suggestions are available according to the evaluation and assessment results [Hohl et al., 2018].
	Category 3	L5.C3.1	Ideal agile physical setup	Ideal agile physical setup helps establish the right environment for the agile software development to thrive in. The key in this setup is that the team is co-located and knowledge can be shared among team members instantly [Sidky, 2007].
		L5.C3.2	Changing the organizations	Organizations must make substantive changes to achieve full benefits of agile at enterprise level. These changes can mean changes in engineering practices, managing the impact on operations, reorganizations, etc. The executive sponsor has a critical role and his dedication and commitment to the process, as well as leadership by example, is the key factor in unlicking great benefits. [Turetken et al., 2017]
	Category 4	L5.C4.1	Test driven development	Software development technique that involves repeatedly first writing a test case and then implementing only the code necessary to pass the test [Sidky, 2007].

Maturity Level	Principle	ID	Practice	Description of Practice
		L5.C4.2	no or minimal number of Cockburn Level "1B" or "-1"	Cockburn "Level 1B" and "Level -1" developers have the least experience and are not able or willing to collaborate which can hamper the transition to agility. This is why their presence is discouraged at Level 5. [Sidky, 2007]
	Category 5	L5.C5.1	Frequent Face-to-face interaction between develops and users	It is ideal to have not just the developers collocated but also have the customers and users in the same rooms. This ensures almost instant feedback and incredible communication. [Sidky, 2007]
		L5.C5.2	Concurrent testing	The practice of incorporating test-development, test automation, and test execution practices within the course of the iteration. Concurrent testing involves several Agile Testing strategies such as *Unit Testing, Acceptance Testing, Component Testing, System, Performance and Reliability testing*. This practice is listed at Level 5 since all the tests should be automated; run frequently and all these efforts take long time to master. [Turetken et al., 2017]
	Category 6	L5.C6.1	Standardized interfaces for software units	Ensure that standardized interfaces between software units are defined and ensure that these interfaces are standardized to enable the development to replace software parts easily [Hohl et al., 2018].
		L5.C6.2	Continuously evaluation of the architecture	Ensure that the efficiency of the SPL architecture and the development process are continuously evaluated. In order to "monitor development efficiency", it should be ensured that metrics are selected to measure the efficiency and the implications on the process when suggestions for improvement are introduced [Hohl et al., 2018].

For the *"Category 1 – Embrace change to deliver customer value"*, We have retained five practices from SAMI model and defined five new practices, namely L2.C1.2, L2.C1.3, L2.C1.4, L3.C1.1, L3.C1.2, and L4.C1.3. For example, the description of the practice "L1.C1.1 – Reflect and tune process" is the following:

"Holding retrospectives at regular intervals of the development process. The objective of this practice is to overcome process challenges that have been faced thus far".

In addition, the practice "L2.C1.2 – Domain requirements" is an APL practice and the description of this practice is the following:

"Domain requirements encompass requirements that are common to all applications of the software product line as well as variable requirements that enable the derivation of customized requirements for different applications".

For the "Category 2 – Plan and deliver software frequently", seven agile practices were retained from SAMI model and ten APL practices were adopted. The adopted practices are L2.C2.2, L2.C2.3, L2.C2.4, L2.C2.5, L2.C2.6, L3.C2.3, L3.C2.4, L3.C2.5, L4.C2.3, and L5.C2.2.

For the "Category 3 – Human centricity", five agile practices from SAMI model were adopted and three new APL practices, namely, L2.C3.1, L4.C3.2, and L5.C3.2, were defined.

For "Category 4 – Technical excellence", this category of the AgiPL-AM has twenty-four practices among these practices fourteen practices were retained from SAMI model. Moreover, ten new APL practices have been defined, namely, L1.C4.1, L1.C4.5, L1.C4.6, L2.C4.4, L2.C4.5, L3.C4.1, L3.C4.3, L3.C4.4, L4.C4.3, and L4.C4.4.

For "Category 5 – Customer collaboration", based on the proposed assessment model, eight APL practices were defined and seven practices were adopted from SAMI model. Precisely, L1.C5.2, L1.C5.3, L1.C5.4, L3.C5.1, L3.C5.2, L4.C5.3, L4.C5.4, L4.C5.5, and L5.C5.2.

Finally, for "Category 6 – Product-Line Architecture", which is a new category added to the 5 five categories of SAMI model, thirteen APL practices were defined.

7.5.2 The AgiPL-AM

Based on the iteration development and the retrospective of followed approach, several adjustments were made in order to optimize the AgiPL-AM model, which involves both agile practices adopted from the SAMI model and APL practices that were defined to address the main objective. The main changes that are performed on the agile practices adopted from the SAMI model are the following:

- The agile practices "Paired programming" and "Agile documentation" were removed as their purposes are covered by "L2.C3.1 – Define/Build/Test teams of each tier: Domain Engineering tier & Application Engineering tier" and "L2.C1.2 – Domain Requirements";
- The agile practices "Backlog" was renamed into "Product Backlog" in order to match the actual agile terminology more and thus provide a better representation;
- The practice "Product Backlog" was moved to the "level 1 – Collaborative", as it is considered to provide the basis for other practices at higher maturity levels.

When the agile/APL practices were defined and refined, a set of governing rules were applied in order to populate these practices in the appropriate maturity level and principle. These rules are the following:

- The *first rule* states that each practice has to contribute to the achievement of the maturity level objective in which it is positioned. For example, the practice "L1.C3.2 –

Collaborative Teams" should addresses directly the "collaboration" objective of maturity Level 1 (i.e. Collaborative);

- The *second rule* is followed to ensure the relevance of the practice with respect to the agile principle that it is associated. The practice L1.C3.2 is related to the principle for "Human-centricity";

- The *third rule* states that the relation between the practices in such a way that practices positioned at higher levels depend on the achievements of the practices at lower levels. For example, the APL practice of "L2.C2.2 – Two-tier planning and tracking (i.e. Domain Engineering tier & Application Engineering tier)" at level 2 depends on achieving some of "Level 1" practices, such as "L1.C3.2 – Collaborative Teams" and "L1.C2.1 – Collaborative planning".

The final version of the AgiPL-AM model was optimized and its adopted practices are presented above, in *Section 7.5.1*. AgiPL-AM is considered as a descriptive model (i.e. as opposed to prescriptive), asit describes only the essential practices that an organization adopting an APL approach should possess at a particular level of maturity. Table 7.2 summarizes the codes of the practices adopted by AgiPL-AM, as well as their Agile and APL principles, and Agile Levels. The codes in bold and italic are the codes of the new defined practices of AgiPL-AM.

Table 7.2 AgiPL-Assessment Model: Levels, Principles, and ID of Practices

		AGILE PRINCIPLES					**APL**
		Category 1	**Category 2**	**Category 3**	**Category 4**	**Category 5**	**Category 6**
AGILE LEVELS	**Level 1 Collaborative**	L1.C1.1	L1.C2.1	L1.C3.1 L1.C3.2	***L1.C4.1*** L1.C4.2 L1.C4.3 L1.C4.4 ***L1.C4.5*** ***L1.C4.6***	L1.C5.1 ***L1.C5.2*** ***L1.C5.3*** ***L1.C5.4***	***L1.C6.1*** ***L1.C6.2*** ***L1.C6.3***
	Level 2 Evolutionary	L2.C1.1 ***L2.C1.2*** ***L2.C1.3*** ***L2.C1.4***	L2.C2.1 ***L2.C2.2*** ***L2.C2.3*** ***L2.C2.4*** ***L2.C2.5*** ***L2.C2.6***	***L2.C3.1***	L2.C4.1 L2.C4.2 L2.C4.3 ***L2.C4.4*** ***L2.C4.5***	L2.C5.1	***L2.C6.1*** ***L2.C6.2*** ***L2.C6.3***
	Level 3 Effective	***L3.C1.1*** ***L3.C1.2***	L3.C2.1 L3.C2.2 ***L3.C2.3*** ***L3.C2.4*** ***L3.C2.5***	L3.C3.1 L3.C3.2	***L3.C4.1*** L3.C4.2 ***L3.C4.3*** ***L3.C4.4*** L3.C4.5 L3.C4.6 L3.C4.7	***L3.C5.1*** ***L3.C5.2***	***L3.C6.1*** ***L3.C6.2***
	Level 4 Adaptive	L4.C1.1 L4.C1.2 ***L4.C1.3***	L4.C2.1 L4.C2.2 ***L4.C2.3***	***L4.C3.1***	L4.C4.1 L4.C4.2 ***L4.C4.3*** ***L4.C4.4***	L4.C5.1 L4.C5.2 ***L4.C5.3*** ***L4.C5.4*** *L4.C5.5*	***L4.C6.1*** ***L4.C6.2*** ***L4.C6.3***
	Level 5 Encompassing	L5.C1.1	L5.C2.1 ***L5.C2.2***	L5.C3.1 ***L5.C3.2***	L5.C4.1 L5.C4.2	L5.C5.1 ***L5.C5.2***	***L5.C6.1*** ***L5.C6.2***

In the proposed model, on the one hand, the agile practices adopted from the SAMI model are assessed by using the original indicators of SAMI model. On the other hand, the APL practices are assessed by using AgiPL-AM indicators (i.e. as set of defined indicators related to the APL practices defined as part of AgiPL-AM). These indicators are not listed in this chapter, as listing these indicators does not fall within the scope of this thesis. For example, in order to assess the practice "L3.C2.3 – Mastering the iteration", the following indicator is used: ***"L3.C2.3.ind** – the development team has effective iterations consisting of sprint planning, tracking, execution, and retrospectives".*

Furthermore, based on the practices of AgiPL-AM, all the indicators are rated by using an achievement scale. From the ISO/IEC 15504 assessment standard [ISO/IEC, 2013], the rating scheme was adopted. This rating scheme is the following:

i. **(N) – "Not achieved (0% − 35%)"** represents little or no evidence of achievement of the practice;

ii. **(P) – "Partially achieved (35% − 65%)"** denotes some evidence of an approach to, and some achievement of, the practice. Some aspects of achievement may be unpredictable;

iii. **(L) – "Largely achieved (65% − 85%)"** indicates that there is evidence of a systematic approach to, and significant achievement of, the practice; despite some weaknesses;

iv. **(F) – "Fully achieved (85% – 100%)"** indicates strong evidence of a complete and systematic approach to, and full achievement of, the practice without any significant weaknesses.

The assessment process requires going through all practices and corresponding indicators to assess the entire set of practices in AgiPL-AM. In order to provide confirmation regarding the results of the assessment, an assessment report should be compiled to present the results to relevant parties.

7.6 Conclusion

The combination of agile software development and software product lines is a promising approach. The current status on the agile adoption within agile software product line approaches is hard to define [Hohl et al., 2017], thus, the need for a specific assessment model, to evaluate the situation of agile adoption within agile product line approaches, was identified.

The research objective of this chapter is to design an assessment model that can be used as a guideline by organizations to adopt agile product line methodologies and assess the success level of agile practices adoption. Through the review of the literature, it was identified that many of the known studied assessment models focus simultaneously on agile and APL practices in detail within APL approaches. Comparing to these approaches, AgiPL-AM is considered structured because it increases the chances of success in agile and APL practices within agile software product lines. In addition, AgiPL-AM serves as an evolutionary path that increases

organization's agile maturity. Also, the proposed model prioritizes the improvement actions in adopting agile and APL practices.

This assessment model would be used in the validation part in order to assess the proposed APL methodology in this thesis (i.e. AgiFPL). In fact, AgiPL-AM will be used to highlight the achievement level of each integrated agile practice or activity, as well as the usability of the different processes of AgiFPL. Moreover, AgiPL-AM will help us to identify the strengths and weaknesses of AgiFPL when comparing other existing APL methodologies. Furthermore, this assessment model will be considered as an add-on that help practitioners and researchers to highlight the main elements of the "Big Picture" of Agile Product Line Engineering scientific field.

The proposed work is an ongoing work. For future work, we plan to further evaluate the AgiPL-AM model in order to improve it. As next step, AgiPL-AM will be evaluated empirically and a number of members of companies will be involved in the evaluation of the applicability of AgiPL-AM; more precisely, in assessing the level of agility of their companies, and in evaluating the overall findings of the assessment model.

8 An Integrated Requirements Engineering Framework for Agile Software Product Lines

Abstract. *The Integrated Requirements Engineering framework for Agile Software Product Lines* is the part that provides the syntax and semantics used for expressing the products of an APL method. This part actually 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; they notably allow establishing the relevance of adopting or not the product line approach for software-intensive systems production. This chapter 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 a holistic perspective integrating human, organizational and agile aspects to better understand product lines dynamic business environments. It helps bridging the gap between product lines structures and requirements models, and proposes an integrated framework to all actors involved in the product line architecture. In this chapter we show how our proposed metamodel can be applied to the requirements engineering stage of an agile product line development mainly for feature-oriented agile product lines such as our own methodology AgiFPL.

8.1 Introduction

As shown in this thesis, the successful adoption and implementation of an Agile Product Line methodology (e.g. AgiFPL) requires a deep organizational mind shift as, in fact, all software engineering processes are affected, from *requirements* to *maintenance* and *evolution* activities. In Chapter 6, the AgiFPL method has organized agile activities and practices 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.

Generally, software development methodologies consist of two integral parts. The first one is a modeling language and the second one is a process [Asadi and Ramsin, 2008]. Thus, it is clear that any proposed methodology uses to handle, on the one hand, a requirements engineering perspective (including modeling and other related tasks) and, on the other hand, the dynamic perspectives such as development, implementation, or maintenance, etc. Since one of the major goals is to propose a complete methodology, we therefore take into account these two parts. The second part was addressed in Chapter 6, and in the present chapter, we handle the Requirements Engineering perspective.

Requirements engineering (RE) – more precisely here, Goal-Oriented Requirements Engineering (GORE) − and Feature Modeling including elicitation, analysis, specification, verification, and management [Pohl et al., 2005], plays a determinant role when defining a new Agile Product Lines methodology and making the decision to adopt (one of) them to meet the business goals of an organization. Compared with RE for a single custom-built system, RE for a software-intensive system family focuses more on systematic reuse, not only from the technical perspective, but from the organizational and process perspectives as well [Coplien et al., 1998]. Therefore, many crucial decisions have to be made during the requirements engineering stages that influence the structure, development, and implementation of an agile product line.

Managing product line requirements is non-trivial as they reflect stakeholders' diverse perspectives, have complex configuration dependencies (e.g., requires, uses, excludes, extends …), and are expressed in various forms (e.g., textual, goals) and at different granularity levels (e.g., features, qualities) [Alves et al., 2010]. This chapter proposes an integrated goal and feature-based metamodel for managing requirements phases of Agile Product Line. The aim was to allow analysts and developers to produce specifications that precisely capture the stakeholder's needs and intentions as well as to manage product line variabilities. In addition, our motivation was to understand and build an efficient structure of the requirements engineering of a feature-oriented product line in an agile context with the research question of building a RE approach for an agile product line to efficiently represent stakeholders' intentions and goals, as well as product line variabilities and commonalities.

This chapter has defined and formalized a specific metamodel for describing product lines, using the Z specification language [O'Regan, 2013]. Furthermore, it highlights how the proposed metamodel can be applied to requirements engineering stages of our agile product line methodology AgiFPL that has been introduced in Chapter 6. Note that the metamodel application process starts by (1) specifying the concerned requirements processes of the methodology; (2) detailing the integration of the metamodel with these processes; (3) validating on a real-world case study; and (4) comparing the proposal to existing approaches.

According to AgiFPL the Domain Engineering (DE) deals with all the aspects of managing reusable assets (artifacts), while the Application Engineering (AE) aims at developing a specific product for a particular stakeholder. Therefore, requirements engineering approaches have to cope with the different organizational levels and architectural complexity. Specifically for product lines, requirements engineering captures both commonality and variability among product line members [Borba and Silva, 2009]. Our proposed metamodel

follows a holistic approach that allows the modeling of the organizational and operational context of a product line within a flexible and rapidly evolving environment. It offers thus a better understanding of the representation of product lines requirements and their stakeholders' requirements.

8.2 Related work

Chapters 4 and 5 have demonstrated the difficulty of integrating agile methods with product line engineering, due to the plan-driven and sequential nature of product line approaches versus the iterative and flexible nature of agile frameworks. However, the chapters have highlighted that adding agility to product line engineering is not only possible but can also be highly beneficial [Boehm and Turner, 2003]. This chapter focuses on the requirements engineering part of this particular integration issue; we hereafter present how the requirements engineering stage has been considered in the existent agile product line approaches.

Generally speaking, both agile methods and software product line approaches recognize that changes to the requirements during the product development are going to occur inevitably. However, they handle different strategies to deal with the requirements and the occurred changes.

Table 8.1 Agile Methods versus Product Line Approaches

	Agile Methods	**Product Line Approaches**
Requirements	Emphasis on quick response to requirements' changes with short iterations and small increments for the application. Direct stakeholders collaboration; stakeholders (i.e. owner, etc.) have to participate in the whole software project lifecycle.	Domain requirements and application requirements are both engineered. Indirect stakeholders' collaboration using *well-trained* customer (and user) representatives.
Architecture	Minimal emphasis on the application architecture features beyond the immediate iteration	Domain and Application architecture are both engineered.
Reuse	– Optimistic use of COTS; – Applies streamlined domain engineering activity without emphasis on development of reusable artifacts.	– Special emphasis on maturity and reliability of COTS; – Foundation of the method is "re-using core assets (i.e. artifacts) defined in domain engineering tier".

Table 8.1 compares how agile methods and product line approaches take in charge "Requirements", "Architecture", and "Reuse". Change management strategies for Agile methods deal with occurred changes to requirements by focusing on incremental development and close interactions with stakeholders mainly through the product owner role [Noor et al., 2008]. These agile methods consider organizations as complex adaptive systems, in which requirements are emergent rather than pre-specifiable; therefore, project teams rarely perform comprehensive requirements' elicitation, specification, analysis, validation, or even

management activities [Schön et al., 2017]. On the contrary, the strategies to deal with requirements in product line approaches focus on finding solutions to meet stakeholders' needs by customizing the core assets [Clements and Northrop, 2001] [Highsmith and Cockburn, 2001] and concerned teams grant importance to the process of formal and written specification and documentation. In fact, they try to predict changes in the beginning of the process and maintain the variability models (such as Feature Models) as core assets. Therefore, requirements engineering in product line approaches is done by defining the product line scope [Apel et al., 2013]. In order to define the domain scope, first, the right products for the domain have to be targeted and, second, factors used to know similar systems (i.e. commonalities) and future market demands must be determined. Too large or small of a scope will deteriorate the capabilities of product line approaches to achieve variabilities and desired economies. Requirements and their changes in product line approaches are carried out through analysis, careful predications and smart selections [Pohl et al., 2005]. If a customer's desired product is out of the product line scope, then the cost associated with amending and revising the core assets to meet their needs will be higher than the cost of products within the scope. Thus, the clients have to either accept the high cost or modify their initial requirements in order to gain benefits, such as better maintainability and faster delivery [Tian and Cooper, 2006]. In addition, stakeholders' collaboration is managed through *"customer interface management practice"*. A number of customer representatives (e.g. domain experts, product managers, etc.) have explicitly assigned roles and responsibilities and act as a bridge that connects stakeholders and development teams [Northrop et al., 2012].

On their side, agile methods emphasize simplicity. They call for removing or not taking into account architectural features that are not of immediate interest for the current iteration [van der Linden et al., 2007]. However, product line approaches consider the *"mass production"* principle, which requires the definition and maintenance of a product line architecture in order to satisfy the general requirements of the product line and the individual requirements of products by explicitly recognizing a set of variation points required to support the family of products within the scope of the targeted domain [Kenneth, 2013].

Agile methods can make opportunistic reuse of existent artifacts and pre-developed software components in an application development. In fact, agile methods apply streamlined domain engineering activity without putting emphasis on development of reusable core assets [Wautelet et al., 2014]. However, as said, the Software Product Line Engineering paradigm separates two main processes, namely, Domain Engineering and Application Engineering. The Domain Engineering (DE) process is fundamentally dedicated for establishing the reusable artifacts and thus defining the commonality and the variability of the product line. The Application Engineering (AE) process is essentially dedicated for developing a single product from reusable artifacts created within the domain engineering process [Pohl et al., 2005].

After reviewing some foundations of agile methods and product line approaches (see Chapter 4 and Chapter 5), it appears that classical product line approaches have to deal with the increasingly rapid pace of changing requirements to satisfy market needs, as well as to be within the framework of existing and changing norms and standards; which lead to an increasing need for rapid development and adaptability [Broy, 2013].

Due to their benefits, agile methods could help product line teams and companies deal with the highlighted issue and thus being agile. In fact, if the known requirements, on the basis of which the product line development teams perform the project scoping are not sufficiently detailed for a thorough analysis, agile methods can be applied. Indeed, these allow eliciting requirements and/or managing changes in the already identified ones. Agile methods may also help these teams (in both domain engineering and application engineering tiers) to produce prototypes swiftly and modify them quickly according to any change in requirements. In addition, applying agile methods may gradually clarify the targeted scope [Klünder et al., 2018]. Therefore, integrating requirements engineering practices as well as development practices from agile methods could enhance product line approaches.

Several concrete methods and models that integrate agile methods and product line approaches are available. Each proposed method combines product line engineering with selected agile approaches and techniques [Klünder et al., 2018]. Since the main target of this chapter is to propose a requirements engineering framework for agile product lines, hereafter, the requirements engineering techniques are identified by surveying some relevant Agile Product Line approaches.

Babar et al. [Babar et al., 2009] have promoted the integration of agile software development and product line engineering as means of reducing time-to-market, increasing productivity, and improving quality. They establish a phase called “Exploration before agile product development” as requirements engineering process. This process uses the “Product Roadmaps” in order to perform a “Feature Analysis” and thus define the “Product Backlog” that contains “Feature Description”. Based on the “Product Backlog”, the “Sprint Backlog” for the implementation is structured.

Ghanam et al. [Ghanam et al., 2008b] have introduced an agile product line method (i.e. Test Driven Development (TDD)) that emphasizes on writing tests before writing code as a means of ensuring the satisfaction of customer requirements, and reinforcing good design habits. To satisfy the customer requirements, they have proposed “Acceptance Tests” in order to identify what features are to be delivered by the implemented product lines. Acceptance tests are considered as “executable specifications” and they are usually written in a format accessible to both technical and non-technical audiences; they can always be a reference of what was requested by the customer, what has been done so far and what is to be done next.

O’Leary et al. [O'Leary, 2012] have proposed an agile process model for deriving products in Software Product Line Engineering. The model was developed through industry-based case study research. The requirements engineering activities are done during the phase “Preparing for Derivation”. In this phase the requirements are determined, prioritized, and assigned to development iterations. The purpose of this phase is to achieve agreement among all the stakeholders on the product requirements.

Díaz et al. [Díaz et al., 2014] have introduced an agile product line approach called “Agile Product-Line Architecting (APLA)” that integrates a set of mechanisms in order to support agile architecting of the Product lines. The APLA process has been deployed in Scrum [Schwaber and Beedle, 2001] by the smooth integration of its mechanisms in this agile method.

According to the APLA process, the first task consists of capturing the requirements of the "Software Product Line Owner" from the product vision (i.e. features). These features can be decomposed into a list of "user stories (US)" and "tasks" known as "Software Product-Line Backlog". In addition, dos Santos Jr. and Lucena Jr. [dos Santos and Lucena, 2010] have presented the ScrumPL approach that supports iterative domain and application engineering based on Scrum.

Table 8.2 RE Tools/approach and activities, identified in studied agile product line approaches.

RE approach/tools	RE activities
– Structured text; – Use Cases; – Features; – Orthogonal variability models; – User Stories.	– Plan and Elicit; – Model and Analyze; – Communicate and Agree; – Realize and Evolve.

Table 8.2 summarizes the requirements engineering approaches/tools and activities used within the identified publications that propose agile product lines models or approaches. These works underline the need for a performance requirements engineering framework that ensure the agile transformation, while preserving existing software product lines. To answer that, we try here to propose a convenient framework that can be applied to requirements engineering stage of feature-oriented agile product lines.

8.3 Research approach

The main research question (RQ) tackled in this chapter aims to figure out the requested RE practices that fit the proposed APL method (i.e. AgiFPL) without harming the agile principles. The goal is to establish a trade-off between the required RE activities that satisfy the SPL structure and the need to minimize the upfront design as much as possible. Therefore, the research question of this chapter is the following:

What are the Requirements Engineering activities and practices that fit the Agile Software Product Lines principles? How they could be integrated to AgiFPL?

Figure 8.1 shows the research method followed in order to respond to the research question mentioned in this section. We have followed these three phases in order to define a framework that meets the main goal of this chapter. In fact, a framework for managing the requirements stage of Agile Product Lines should serve the analysis of software product lines processes, the definition of a (conceptual) process language, and the implementation of software processes using tools [Kuhrmann, and Tiessler, 2014]. In this section, before detailing our metamodel itself; we first present how it was crafted. However, the implication on Agile-Product-Lines' processes analysis, design and implementation will be discussed in future work.

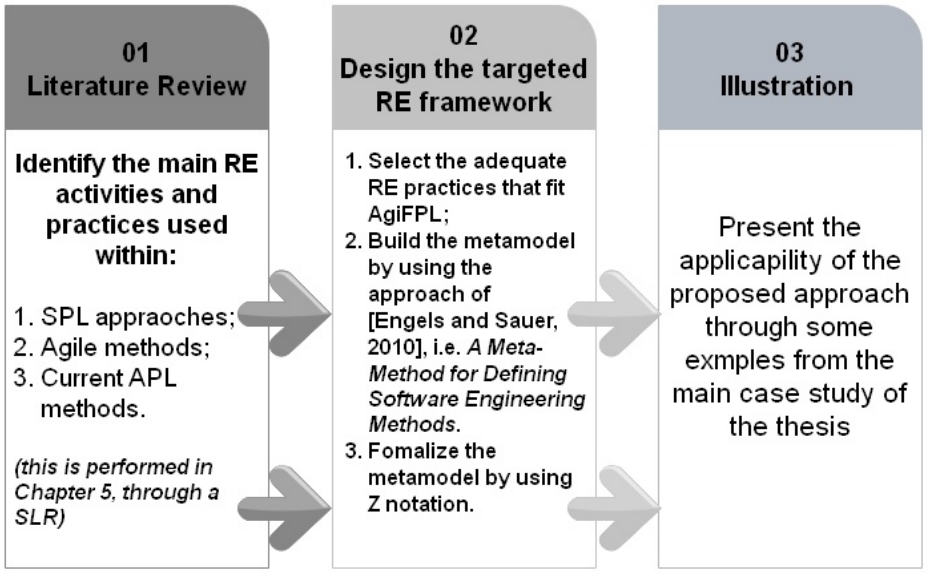

Fig. 8.1 Research process.

According to Engels and Sauer [Engels and Sauer, 2010], the procedure for constructing metamodels that support Life Cycle Management adopts the following steps:

- 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.

Based on the approach presented in [Engels and Sauer, 2010] and to target the research question mentioned above, the proposed metamodel is based on the comprehensive analysis of common data types, representation schemes and relationships that exist in Agile Product Lines approaches and then represented through a Unified Modeling Language (UML) class diagram [OMG, 2017]. Three major steps are taken (iteratively):

– *Step 1*: identification of the basic RE concepts of Agile Product Lines methodologies;

– *Step 2*: Analysis of the relationships between concepts. Four types of relationships are used, namely generalization, composition, aggregation and association. The existence of relationships between concepts needs to be identified and their types distinguished;

– *Step 3*: Formal expression of the metamodel according to the basic concepts and relationships gathered in the first steps, including UML class diagram graphical representation.

8.4 A Metamodel for Agile Product Lines

As stated above, our motivation is to understand and build an efficient structure of the requirements engineering of a feature-oriented product line in an agile context. This leads us to define a goal and feature-oriented specification to provide modeling constructs that permit:

- *Representations of stakeholder's intentions and goals;*
- *Variability, commonality and technical elements of the agile product line;*
- *Requirements artifacts and their relationships used by agile teams.*

The proposed metamodel defines two main perspectives. The first one is the product line engineering perspective itself, in which goal (i.e. Family Goal Model) and feature models provide different variability perspectives and the rationale of the variability. The second one is the agile development perspective, in which the agile requirements artifacts and goal models provide an exhaustive structure for the implementation of product line's features and products derivation.

Standard goal model frameworks like *i** [Yu et al., 2011] [Mouratidis et al., 2005] can represent intentional variability, but lack mechanisms for representing differences between intentional spaces of various systems (i.e., product line variability in the intentional space). Therefore, Asadi et al. [Asadi et al., 2014] have introduced the notion of family goal model to extend standard goal modeling techniques, which we apply in this paper to *iStar 2.0* [Dalpiaz et al., 2016], the second version of *i**.

Our metamodel connects *Family Goal Models (FGM)* and *Features Models (FM)* through mappings. They provide bidirectional relationships and traceability links between high-level stakeholders' business objectives, which are described by goal models and implementation units enclosed within features in feature models. In addition, we seek to support the stakeholders of a product line, especially in the application engineering tier, through *iStar 2.0* models, which provide a graphical and comprehensive vision of the stories and their relationships. Our proposed model connects Backlog items (i.e., user stories…), and family goal models by mappings performed through heuristics rules proposed in [Jaqueira et al., 2013] and [Highsmith and Cockburn, 2001].

Figure 8.2 introduces the main entities and relationships of our metamodel. We subdivide it into four sub-models:

- The *Organizational sub-model*, describing the members (i.e. actors, teams, …) of the product line, their organizational roles, responsibilities, capabilities and relationships;
- The *Goal-oriented sub-model*, describing the intentions of the product line stakeholders and generating a stakeholder's view of feature models;
- The *Feature-oriented sub-model*, illustrating the product line variability;
- The *Agile requirements artifacts sub-model* defining the requirements artifacts used by agile teams, as well as the relationships among these artifacts.

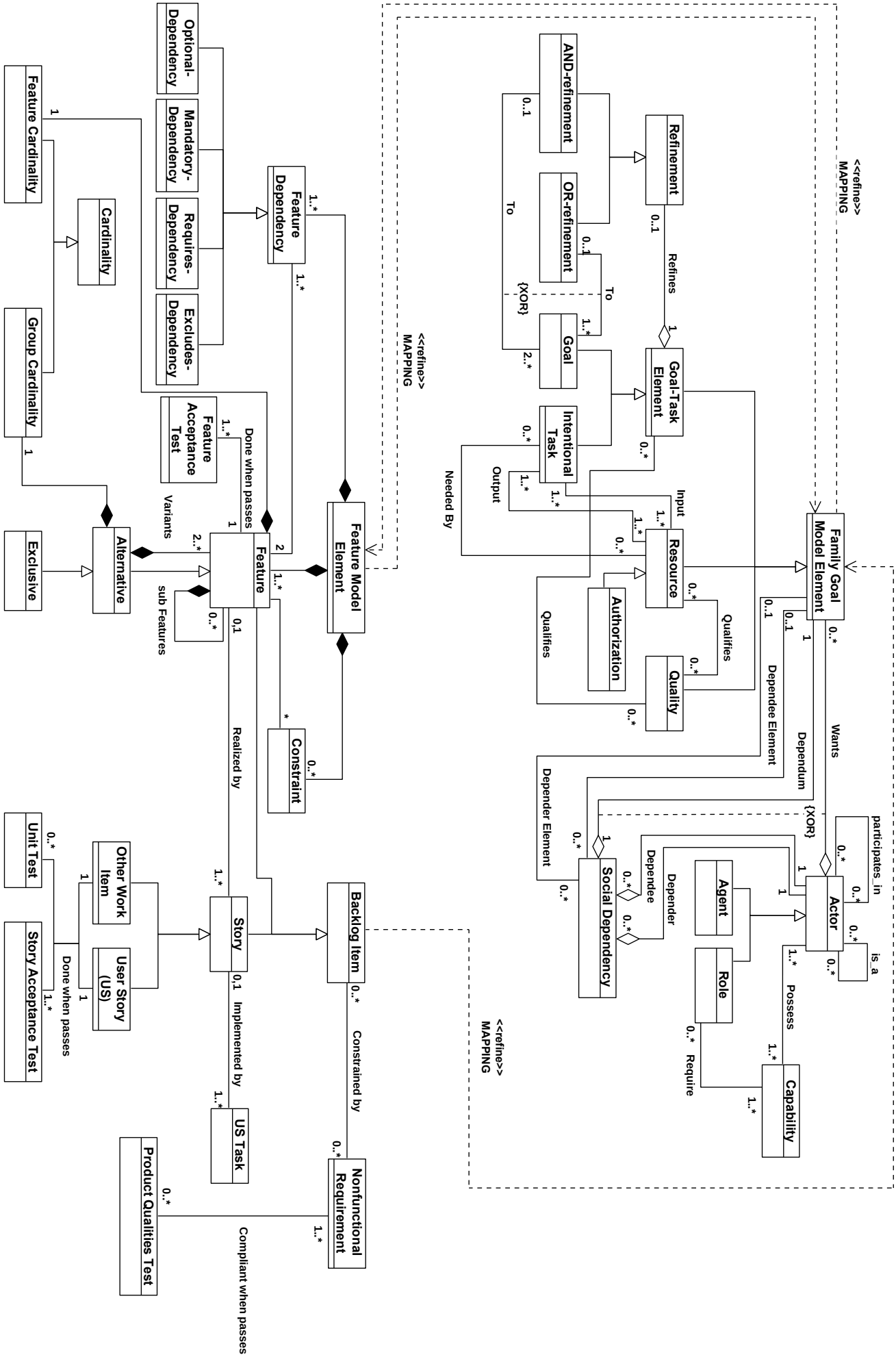

Fig. 8.2 Requirements-oriented Meta-model for Agile Product Lines.

The primitives of our framework are also of different types. We classify them as:

- *Meta-concepts*: Goal, Feature, Actor, User Story …
- *Meta-relationships*: Qualifies, Refines, Composition, Aggregation, Generalization …
- *Meta-attributes*: Power, Motivation ...
- *Meta-constraints*: implications between features located in different parts of the feature hierarchy.

All meta-concepts, meta-relationships and meta-constraints have the following mandatory meta-attributes: *Name* and *Description*. ***Name*** allows unambiguous reference to the instance of the meta-concept and ***Description*** provides a precise and unambiguous description of the corresponding instance of the meta-concept. The description should contain sufficient information for a formal specification to be derived for use in requirements specifications for a future product or application of the product line.

Figure 8.2 insists on meta-concepts and meta-relationships. Meta-attributes and meta-constraints are formalized with the *Z state-based* specification language [O'Regan, 2013]. We use Z since it provides sufficient modularity, abstraction and expressiveness to describe the requirements engineering aspects of agile product line and the wider context in which they are used in a consistent and structured way. In addition, Z offers a pragmatic approach to specifications by allowing a clear transition between specification and implementation of product lines' applications. Moreover, it is widely adopted in the software development industry and academia.

This chapter details next the organizational, goal-oriented, feature-oriented sub-models and their integration, and the user story concept. It also discusses their relevance for agile product lines requirements engineering.

8.4.1 Organizational Sub-Model

The main entities and relationships of the organizational sub-model defined in our metamodel are presented in the right corner of the bottom of Figure 8.2. The Organizational sub-model, describes the members (i.e. actors, teams …) of the product line, their organizational roles, responsibilities, capabilities and relationships. In fact, this sub-model identifies the relevant ***Actors*** of the product line, the ***Roles*** they occupy, the ***Capabilities*** they possess, and the ***Dependum*** for which Actors depend on one another.

8.4.1.1 Actor

Most of stakeholders are represented as actors. Actors can be human, organizations, technical systems (i.e. hardware, software), or any combination thereof. Actors are active, autonomous entities that aim at achieving their goals by exercising their *know-how* in collaboration with

other actors. According to the iStar 2.0 language, two types of actors can be distinguished [Dalpiaz et al., 2016] [Kolp et al., 2005]:

- **Role:** *an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor*.
- **Agent:** *an actor with concrete, physical manifestations, such as a human individual, an organization, or a department*.

Actor's intentionality is made explicit through the actor boundary, which is a graphical container for their intentional elements.

[Name]
[Informal_Defintion]
[Actor_Type] := Role | Agent
[Goal]

Actor

name : *Name*
description : *Informal_Definition*
is_a : *Actor_Type*
want : *set Goal*
own : *set Resource*
possess : *set Capability*

(want ≠ ø) ∧ (possess ≠ ø) **(c1)**

(∀ act : *Actor*) act.is_a = Agent ⇒ act.own ≠ ø **(c2)**

The ***Actor*** schema above shows the Z formal specification [O'Regan, 2013] of the Actor concept. The first part of the specification represents the definition of types. The Actor specification first defines the type Name (which represents the Name at-tribute) by writing [Name]. This declaration introduces the set of all names, without making assumptions about the type (i.e. whether the name is a string of characters and numbers, or only characters,…). The type [Actor_Type] is defined as being either a *Role* or an *Agent* or even just an *Actor*.

More complex and structured types are defined with specific schemata. For instance, the ***Actor*** schema is partitioned horizontally into two sections:

- The declaration section introduces a set of named, typed variable declarations;
- The predicate section provides predicates that constrain values of the variables. Identifiers e.g. "(c1)" are used to refer to predicates, i.e. *constraint (c1)* of the schema.

In essence an *Actor* of an agile product line *wants* to fulfill the product line *Goals*, as well his/her own *Goals*. In fact, an *Actor* possesses his/her specific *Capabilities* and owns a set of *Resources*. Each Actor applies plans that are part of his/her *Capabilities* and uses *Resources* in order to achieve the *Goal* that he/she *wants*. As the *Actor* is present in a rapid and flexible environment, he/she takes into account the changing *Intentional Elements* related to the product

line as well as the ones related to specific customers' needs, in order to adapt its behavior to environmental circumstances. Considering these changes are crucial when eliciting product line requirements, as well as stakeholders (i.e. product owner, etc.) requirements. Since an *Actor* can be also a *Role* or an *Agent*, two different types of actor links exist:

- ***is-a***: represents the concept of generalization or specification. Only *Roles* can be specialized into *Roles*, or general *Actors* into general *Actors*. However, *Agents* cannot be specialized via ***is-a***, as they are concrete instantiations;
- ***participates-in***: represents any kind of association, other than generalization or specialization, between two *Actors*. No restriction exists on the type of actors linked by this association. Note that every *Actor* can ***participates-in*** multiple other *Actors*.

Thus, a ***is-a*** relationship applies only between pairs of *Roles* or pairs of *Actors*. There should be no *is-a* cycles. In addition, there should be no *participate-in* cycles. A pair of Actors can be linked by at most one actor link. It is not possible to connect two actors via both *is-a* and *participates-in*. An *Actor* can (sometimes, has to) cooperate with another *Actor* to fulfil common *Goals* to the *Roles* that each of these *Actors* occupies.

8.4.1.2 Role

As stated above, an organizational *Role* of the product line is an abstract characterization of expected behavior of an *Actor* within some specified context of the product line. An *Actor* can occupy multiple *Roles* and multiple *Actors* can occupy a *Role*.

[Goal_control_Status]

Role

name : *Name*
description : *Informal_Definition*
require : *set Capability*
responsible : set *Goal*
control : *set* (*Goal*, *Goal_control_status*)
authoroty_on : *set Role*

(require ≠ ø) ∧ (responsible ≠ ø) **(c3)**

(∀ act : *Actor* ; r : *Role*) **(c4)**
r ∈ act.occupy ⇒ r.require ⊂ act.possess

(∀ act : *Actor* ; r : *Role*) **(c5)**
act.leave ⇒ r ∉ act.occupy

(∀ r : *Role* ; g : *Goal*) **(c6)**
g ∈ r.responsible ⇒ g.sec_is_a = Goal

(∀ r_1 , r_2 : *Role* ; g : *Goal* ; a_1 , a_2 : *Actor*) **(c7)**
(g.sec_is_a = Goal ∧ g ∈ r_1.responsible ∧ g ∈ r_2.control ∧ r_1 ≠ r_2 ∧ r_1 ∈ act.occupy
∧ r_2 ∈ act.occupy) ⇒ a_1 ≠ a_2

The ***Role*** schema above shows the Z formal specification of Role concept within a product line. Each Role requires a set of Capabilities to fulfill or contribute to Goals for which it is responsible. An Actor can occupy the Role only if she/he possesses the required Capabilities **(c4)**. Moreover, to entering Roles, Actors should be able to leave Roles at runtime **(c5)**.

Roles are responsible for *Goals* **(c6)** and can control their fulfillment. This control procedure requires that a single *Actor* can never occupy distinct *Roles* that are responsible for and control the fulfillment of the *Goal* **(c7)**. In addition, *Roles* can have different levels of authority. Consequently, a Role can have authority over other Roles. The authority on relationship specifies the hierarchical structure of the product line.

8.4.1.3 Capability

In this proposal, we consider that a *Capability* specifies the behavior that *Roles* should have in order to be responsible for or to control their *Goals*.

[Capability]
[Capability_Availability] := Available | Unavailable

Capability

name : *Name*
description : *Informal_Definition*
composed_of : set *Capability*
availability : *Capability_Availability*

(composed_of ≠ ø) **(c8)**

(∀ cap : *Capability*) **(c9)**
∃ act : *Actor* ; cap ∈ act.possess ⇒ cap.availability = available

In general, an Actor possesses *Capabilities*. The ***Capability*** schema above shows the formal specification of the *Capability*. In addition, it shows that a *Capability* can be structured as a set of other *Capabilities*. When exploring possible alternative product line members (i.e., new domain feature,...), newly identified *Roles* can require *Capabilities* that no Actor possesses. Thus, these new Capabilities have to be confronted to those available in the product line structure **(c9)**, in order to evaluate the proposed alternatives with respect to the current *Roles* and the way they use existing *Capabilities*. Moreover, the availability of a Capability is formally expressed through the availability attribute, as mentioned in the Capability schema.

8.4.1.4 Dependum

In social models, such as iStar 2.0, dependencies represent social relationships. A dependency is defined as a relationship with five arguments:

- *Depender* is the actor that depends on something (the dependum) to be provided;
- *DependerElmt* is the intentional element within the depender's actor boundary where the dependency stems from; which explains why the dependency exists;
- *Dependum* is an intentional element that is the object of the dependency;
- *Dependee* is the actor that should provide the dependum;
- *DependeeElmt* is the intentional element that explains how the dependee intends to provide the dependum.

Dependencies link the *dependerElmt* within the *depender* actor to the dependum, outside actor boundaries, to the *dependeeElmt* within the *dependee* actor. The type of the dependum specifies the semantics of the relationship:

- *Goal*: the dependee is expected to achieve the goal, and is free to choose how;
- *Quality:* the dependee is expected to sufficiently satisfy the quality, and is free to choose how;
- *Task:* the dependee is expected to execute the task in a prescribed way;
- *Resource:* the dependee is expected to make the resource available to the depender.

[Dependum_Type] := Goal | Quality | Task | Resource

Dependum

name : *Name*
description : *Informal_Definition*
type : *Dependum_Type*
depender : set *Role*
dependee : set *Role*

(type ≠ ø) ∧ (depender ≠ ø) ∧ (dependee ≠ ø) **(c10)**

(∀ d : *Dependency* ; dpd : *Dependum* ; r_1, r_2 : *Role*) **(c11)**
$r_1 \neq r_2 \wedge (d \equiv r_1 \text{ x dpd x } r_2) \Rightarrow$ (depender = $r_2 \wedge$ dependee = r_1)

(∀ d : *Dependency* ; dpd : *Dependum* ; r_1, r_2 : *Role*) **(c12)**
$r_1 \neq r_2 \wedge (d \equiv r_1 \text{ x dpd x } r_2) \wedge$ (dpd.type = authorization) $\Rightarrow r_1 \in r_2$.authoroty_on

(∀ res : *Resource* ; a_1, a_2 : *Actor* ; cap_1, cap_2 : *Capability* ;
t_1, t_2 : *Task* ; r_1, r_2 : *Role*) **(c13)**
($a_1 \neq a_2 \wedge cap_1 \neq cap_2 \wedge t_1 \neq t_2 \wedge (t_1 \in cap_1$.composed_of $\wedge cap_1 \in a_1$.possess) $\wedge$
($t_2 \in cap_2$.composed_of $\wedge cap_2 \in a_2$.possess) $\wedge$ res $\in t_1$.postcondition $\wedge$
res $\in t_2$.input $\wedge r_1 \in a_1$.occupy $\wedge r_2 \in a_2$.occupy $\wedge \{ r_1, r_2\} \notin \{ a_1$.occupy $\cap a_2$.occupy$\}$)
$\Leftrightarrow$ ($\exists$ dm: Dependum $\wedge$ dm.type = Resource $\wedge$ dm.name = res.name $\wedge$
dm.depender = $r_2 \wedge$ dm.dependee = r_1)

The ***Dependum*** schema above shows the formal specification of the *Dependum*. *Resource* dependency allows the represention of any specialization of the *Resource* concept as a *Dependum*. For example, a *Role (r_1)* might depend on another *Role (r_2)* for an *Authorization*. This has implication on the authority of relationship, as this dependency means that r_2 must have authority over r_1 (i.e. **c13**). In addition, the constraint **(c13)** demonstrates that the existence of a *Resource Dependum* among *Roles* has implications on the *Input* and *Postcondition* of *Tasks* accomplished by *Actors* that occupy these *Roles*.

8.4.2 Goal Sub-Model

Intentional elements are the actors' needs. As such, they model different kinds of requirements and are central to the proposal. The following elements are considered as *Intentional Elements (Family Goal Model Elements)* in this work:

- *Goal*: a state of affairs that the actor wants to achieve and that has clearly cut criteria of achievement;
- *Quality*: an attribute for which an actor desires some level of achievement;
- *Task*: an action that an actor wants to have executed, usually with the purpose of achieving some goal;
- *Resource*: a physical or informational entity that the actor requires in order to perform a task.

[Family_Goal_Element] := Goal | Quality | Task | Resource
[Goal_Type] := Requirement | Expectation
[Goal_Pattern] := Achieve | Cease | Maintain | Avoid
[Status] := Fulfilled | Unfulfilled
[Refinement_Alternative]

Family Goal Model

name : *Name*
description : *Informal_Definition*
intentional_elmt_is_a : *Family_Goal_Element*
goal_is_a : *Goal_Type*
pattern : *Goal_Pattern*
status : *Status*
refined_by : set *Refinement_Alternative*

(∀ g: *Goal* ; t: *Task*) g.intentional_elmt_is_a = Goal ∧
t.intentional_elmt_is_a = *Task* ⇒ (g.status ≠ ∅) ∧ (t.status ≠ ∅) **(c14)**
(∀ g: *Goal*) g.intentional_elmt_is_a = *Goal*

∧ ∃ tset = {t_1 , ... , t_2} ⇒ g.status = Fulfilled **(c15)**

(∀ g : *Goal* ; r : *Role* ; act : *Actor*) **(c16)**
(g.intentional_elmt_is_a = *Goal* ∧ r ∈ act.occupy ∧ g ∈ r.responsible ∧
act.isa = *Agent*) ⇒ g.goal_is_a = *Requirement*

(∀ g : *Goal* ; r : *Role* ; ac t: *Actor*) **(c17)**
(g.intentional_elmt_is_a = Goal ∧ r ∈ act.occupy ∧ g ∈ r.responsible ∧
act.isa = Role) ⇒ g.goal_is_a = Expectation

The ***Family Goal Model*** schema above highlights the formal specification of the *Family Goal Model* adopted in this proposal. Constraint **(c14)** states that Goals and Tasks must have a non-empty status. In addition, if there is a set of Tasks *(tset)*, such that the *Goal* is a subset of *tset*, then the *Goal is fulfilled* **(c15)**. Moreover, a Goal is a *Requirement* if there is some *Agent Actor act* which occupies a *Role* which in turn is responsible for the Goal **(c16)**. A Goal is an *Expectation*, if there is some specific *Role* that is responsible for the *Goal* **(c17)**. Several types of links exist in order to connect intentional elements. These links are: *refinement*, *needed-by*, *contribution* and *qualification*.

Refinement is an n-ary relationship relating one parent to one or more children. An intentional element can be the parent in at most one refinement relationship. Two types of refinement – applied to any kind of parent (i.e. Goal or Task) define the logical operator relating the parent with the children:

- ***AND-refinement*:** the fulfillment of all the n children ($n \geq 2$) makes the parent fulfilled.
- ***Inclusive OR*:** the fulfillment of at least one child makes the parent fulfilled.

The *Needed-By* relationship links a task with a resource and indicates that the actor needs the resource in order to execute the task. The *Contribution links* represent the effects of intentional elements on qualities, and are essential to assist analysts in the decision-making process among alternative goals or tasks. Contribution links lead to the accumulation of evidence for qualities. The *Qualification* relationship relates a quality to its subject (i.e. a task, goal, or resource).

In this proposal, the goal model called Family Goal Model (FGM) represents the intentional space of a domain for which the product line is developed. The adopted goal-oriented approach helps to build artifacts that represent stakeholders' objectives and strategies.

8.4.3 Feature sub-model

As stated above, this proposal offers feature-oriented design and implementation for which feature models are a standard visual representation. Feature models (FM) support a natural description of a wide range of variability schemata.

Features are a fundamental notion in modern software engineering [Apel et al., 2010] [Czarnecki et al., 2012]. In fact, features have already profoundly affected how software is engineered. They can substantially improve the communication among all stakeholders and will likely lead to new, more effective ways to modularize and develop software [Apel et al., 2013].

Features are of primary interest in product-line engineering, as they can be used to distinguish the products of a product line. The notion of a feature is inherently hard to define precisely as it captures, on the one hand, intentions of the stakeholders of a product line, including end users and, on the other hand, design and implementation-level concepts used to structure, vary, and reuse software artifacts [Batory and O'Malley, 1992]. Therefore, the notion

of feature has many definitions. Classen et al. [Classen et al., 2008] have ordered some of these definitions from abstract to technical as shown in Table 8.3 classification.

Table 8.3 Abstract and technical definitions of "feature" concept.

	Ref.	Definition of the concept of "Feature"
1	[Kang et al., 1990]	*"A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"*
2	[Kang et al., 1998]	*"A distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained"*
3	[Czarnecki and Eisenecker, 2000]	*"A distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept"*
4	[Bosch, 2000]	*"A logical unit of behavior specified by a set of functional and non-functional requirements"*
5	[Chen et al., 2005]	*"A product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements"*
6	[Batory et al., 2004]	*"a product characteristic that is used in distinguishing programs within a family of related programs"*
7	[Classen et al., 2008]	*"A triplet, `f = (R, W, S)`, where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification"*
8	[Zave, 2003]	*"An optional or incremental unit of functionality"*
9	[Batory and O'Malley, 2005]	*"An increment of program functionality"*
10	[Apel et al., 2013]	*"A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option"*

According to Apel et al. [Apel et al., 2010], the first seven definitions treat features mainly as a means to communicate between the different stakeholders of a product line (end users, managers, programmers, and so forth), in order to distinguish software products. The last three definitions treat features as design decisions and implementation-level concepts that are part of the software construction phase. These different views on features stem from the different use of features in the different phases of product-line engineering. Based on the essence and commonalities of prior usage, we endorse [Apel et al., 2010] to define features as follows:

Definition 8.1:

A "feature" 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 life cycle [Apel et al., 2010].

A feature model is a tree of which nodes are labeled with feature names. It also proposes various parent-child relationships between features and their constraints. In fact, if a feature ***f*** is a child of another feature ***p***, ***f*** can be selected only when ***p*** is also selected. Typically, a feature model includes mutual relations between features. In addition, *Mandatory* and *Optional* features are distinguished within the feature model. Note that in this proposal the focus lies on *Boolean features* identified by a name. In principle, *non-Boolean features* or *attributes of*

features may also be of interest in distinguishing applications of the product line. In this chapter, we cover essentially *Boolean* features; *non-Boolean* features will be considered in future work.

[Feature_Type] := Parent | Child | Abstract | Concrete
[Feature_Availability] := Available | Unavailable
[Feature_Constraint_Type] := Mandatory | Optional | Alternative | Or

Feature Model

name : Name
description : Informal_Definition
is_a : Feature_Type
availability : Feature_Availability
constraint_type : Feature_Constraint_Type

---

root (f) ≡ f **(c18)**

mandatory (p, f) ≡ f ⇔ p **(c19)**

optional (p, f) ≡ f ⇒ p **(c20)**

alternative (p, {f_1 , … , f_n}) ≡ ((f_1 ∨ … ∨ f_n) ⇔ p) ∧ ($\bigwedge_{i<j}$ ¬ (f_i ∧ f_j)) **(c21)**

Or (p, {f_1 , … , f_n}) ≡ (f_1 ∨ … ∨ f_n) ⇔ p **(c22)**

The ***Feature Model*** schema above formalizes the *Feature Model* concepts. All feature names from the set ***F*** of feature names are interpreted as propositional variables, ***p***, ***f*** and f_i represents members of ***F***. Each edge in the tree is defined by exactly one feature constraint, that is, by a declaration of one of the feature constraint types *mandatory*, *optional*, *alternative*, or "*or"*.

A *mandatory* feature definition between a parent feature and a child feature corresponds to a logical equivalence. That is, whenever the parent feature is selected, so must the child and vice-versa **(c19)**.

An *optional* feature corresponds to implication. The implication states that the parent feature may be chosen independently from the child feature, but the child feature can only be chosen if the parent feature is selected **(c20)**.

The *alternative* constraint defines a one-out-of-many choice. The definition of the constraint **(c21)** has the parent feature as the first parameter and a non-empty set of child features as the second parameter. This constraint is a disjunction, in which, at least, one child feature is selected when the parent is chosen. In addition, it was ensured that for each pair of child features no two child features are selected together.

An unrestricted *choice* or *"or"* defines some-out-of-many choice. Again, the constraint **(c22)** has a non-empty set of child features as second parameter. The selection of parent feature is equivalent to a disjunction of the child features.

Additionally, a set of cross-tree constraints may be defined in the Feature Model (FM). The corresponding propositional formula of the feature constraints and the cross-tree constraints are conjoined resulting in one logic formula that represents the semantics of the whole Feature Model.

8.4.4 User Story concept

The proposed metamodel focuses on agile perspectives; relevant agile requirements artifacts therefore play a core role within the proposal. This section details the user story concept, which is the proposed metamodel integrates.

User stories are considered here due to their wide use and to take profit from their effectiveness. Leffingwell [Leffingwell, 2011] and Cohn [Cohn, 2004], consider them as an increasingly popular textual notation to capture requirements in agile software development. User stories are statements that use a simple template such as

"*As a ⟨role⟩, I want ⟨goal⟩, [so that ⟨benefit⟩]*".

The ***User Story*** schema below formalizes the *User Story* (μ_i) concept. Let U = {μ_1, μ_2,…} a set of user stories in a project. A user story μ is a 4-tuple $\mu_i = \langle r_i, m_i, E_i, f_i \rangle$ where r is the role, m is the means, $E = \{e_1, e_2, ...\}$ is a set of ends, and f is the format. In addition, a means m is a 5-tuple $m = \langle s, av, do, io, adj \rangle$ where s is a *"subject"*, av is an *"action verb"*, do is a *"direct object"*, io is an *"indirect object"*, and adj is an *"adjective"* (io and adj may be null).

A user story μ_1 is an exact duplicate of another user story μ_2 when they are identical **(c23)**. The constraint **(c24)** indicates that a user story μ_1 duplicates the request of μ_2, while using a different text (i.e. Semantic Duplicate). **(c25)** denotes two or more user stories that have the same end, but achieve this using different means. **(c26)** represents the case in which two or more user stories use the same means to reach different ends. For the case where two or more user stories with different roles, but same means and/or ends we formalize the constraint **(c27)**. When there is a strong semantic relationship between two user stories, it is important to add *explicit dependencies* to the user stories, although this breaks the *independent* criterion **(c28)**.

Uniformity in the context of user stories means that a user story format is consistent with the one of the majority of user stories in the same set. Therefore, the format f_1 of an individual user story μ_1 is syntactically compared to the most common format f_{std} to determine whether it adheres to the uniformity criterion **(c29)**.

[User_Story_Element] := Format | Role | Means | Ends

[Mean] := Subject | Action_Verb | Direct_Object | Indirect_Object | Adjective

[End] := Clarification | Dependency | Quality

[Status] := To_Do | In_Progress | Testing | Done

User Story

identifier : *Identifier*
user_story_elmt_is_a : *User_Story_Element*
mean_is_a : *Mean*
end_is_a : *End*
status : *Status*

(∀ μ_1 , μ_2 : User_Story) **(c23)**
is_Full_Duplicate (μ_1, μ_2) ↔ $\mu_1 =_{syn} \mu_2$

(∀ μ_1 , μ_2 : *User_Story*) **(c24)**
is_Sem_Duplicate (μ_1 , μ_2) ↔ $\mu_1 = \mu_2 \wedge \mu_1 \neq_{syn} \mu_2$

(∀ μ_1 , μ_2 : *User_Story* ; m_1, m_2 : *Means* ; E_1, E_2 : *Ends*) **(c25)**
diff_Means_same_Ends (μ_1 , μ_2) ↔ $m_1 \neq m_2 \wedge E_1 \cap E_2 \neq \emptyset$

(∀ μ_1 , μ_2 : *User_Story* ; m_1, m_2 : *Means* ; E_1, E_2 : *Ends*) **(c26)**
same_Means_diff_Ends (μ_1 , μ_2) ↔ $m_1 = m_2 \wedge (E_1 \setminus E_2 \neq \emptyset \vee E_2 \setminus E_1 \neq \emptyset)$

(∀ μ_1 , μ_2 : *User_Story* ; m_1, m_2 : *Means* ; E_1, E_2 : *Ends* ; r_1, r_2 : *Role*) **(c27)**
same_Role_diff_Story (μ_1 , μ_2) ↔ $r_1 \neq r_2 \wedge (m_1 = m_2 \vee E_1 \cap E_2 \neq \emptyset)$

(∀ μ_1 , μ_2 : *User_Story* ; m_1, m_2 : *Means* ; E_1, E_2 : *Ends*) **(c28)**
purpose_is_Means (μ_1 , μ_2) ↔ $E_1 = \{m_2\}$

(∀ μ_1 : *User_Story* ; f_1, f_{std} : *Format*) **(c29)**
is_not_Uniform (μ_1, f_{std}) ↔ $f_1 \neq_{syn} f_{std}$

(∀ μ_1 : *User_Story* ; av_1, av_2 : *Action_Verb* ; do_1, do_2 : *Direct_Object*) **(c30)**
has_Dep (μ_1 , μ_2) ↔ depends (av_1, av_2) ∧ $do_1 = do_2$

(∀ μ_1 , $\mu_2 \in U$: *User_Story* ; do_1, do_2 : *Direct_Object*) **(c31)**
has_is_a_Dep (μ_1 , μ_2) ↔ ∃ $\mu_2 \in U$. is_a (do_1, do_2)

(∀ μ_1 , $\mu_2 \in U$: *User_Story* ; av_1, av_2 : *Action_Verb* ; do_1, do_2 : *Direct_Object*) **(c32)**
void_Dep (μ_1) ↔ depends (av_1, av_2) ∧ ∄ $\mu_2 \in U$. $do_1 = do_2$

In some cases, it is necessary that one user story μ_1 be completed before the developer can start on another story μ_2. Formally, the predicate *has-Dep(μ_1, μ_2)* holds when μ_1 causally

depends on μ_2 **(c30)**. Moreover, an object of one user story μ_1 can refer to multiple other objects of stories in *U*, indicating that the object of μ_1 is a parent or superclass of the other objects. Formally, predicate *has-is-a-Dep(μ_1, μ_2)* is true when μ_1 has a direct object superclass dependency based on the sub-class do_2 o do_1 **(c31)**.

Implementing a set of user stories *U* should lead to a feature-complete application. While user stories should not thrive to cover 100% of the application's functionality preemptively, crucial user stories should not be missed, for this may cause a show stopping feature-gap. The predicate *void-Dep(μ_1)* holds when no story μ_2 satisfies a dependency for μ_1's direct object **(c32)**.

In a nutshell, the proposed metamodel offers a better understanding of the representation of product lines requirements and their stakeholders' requirements. In addition, it takes inspiration from research in GORE frameworks, such as iStar 2.0 [Dalpiaz et al., 2016], and from feature-oriented modeling [Apel et al., 2013] related to agile requirements practices like user stories [Leffingwell, 2011] [Cohn, 2004] [Wautelet et al., 2017].

8.5 Applying the proposed RE approach to AgiFPL

This section illustrates how the proposed metamodel can be adopted for the requirements engineering stage of agile product lines, when using methodology called AgiFPL. As stated in Chapter 6, like classical agile product lines methodologies, AgiFPL is a feature-oriented approach involving two classical tiers of product line engineering, namely *Domain Engineering (DE)* and *Application Engineering (AE)*.

AgiFPL also considers the two, Problem and Solution, spaces. The problem space calls attention to the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products. The solution space represents the developers' and vendors' perspectives. The Solution Space is not emphasized in this work, as the proposed metamodel is designed essentially for the requirements engineering concerned by the Problem Space.

Integrating the proposed metamodel to AgiFPL allows modeling and managing intentions, goals, variabilities and commonalities of the product lines. For example, the *"Family Goal Models"* of the proposed metamodel will guide the development of variability of the product line in the domain engineering, while they are used for the configuration of products in the application engineering. Note that in this chapter we will not present all processes of AgiFPL (see Chapter 6 for the whole process). However, a detailed description of the processes concerning the requirements engineering in both tiers of the methodology (i.e. Domain requirements engineering process and Application requirements engineering process) will be given.

Figure 8.3 illustrates the problem space of the Domain Engineering tier. The figure depicts the main steps of the RE process followed in the domain engineering. Based on the

strategy of a software vendor, who decides to adopt AgiFPL, *Business Experts, Domain Experts* and *concerned teams* apply the proposed metamodel as follows:

1. Execute a sub-process for modeling the family goal models (FGMs). (i.e. goal-oriented requirements engineering);
2. Apply the stated practices and rules of the proposal in order to generate the correspondent feature models (FM) (specifies and design the desired domain – i.e. Domain Design & Feature Backlog);
3. Prioritize the identified features of the designed domain and then document the required user stories (US) (apply the correspondent agile requirements practices – i.e. Stories Backlog, tests, …).

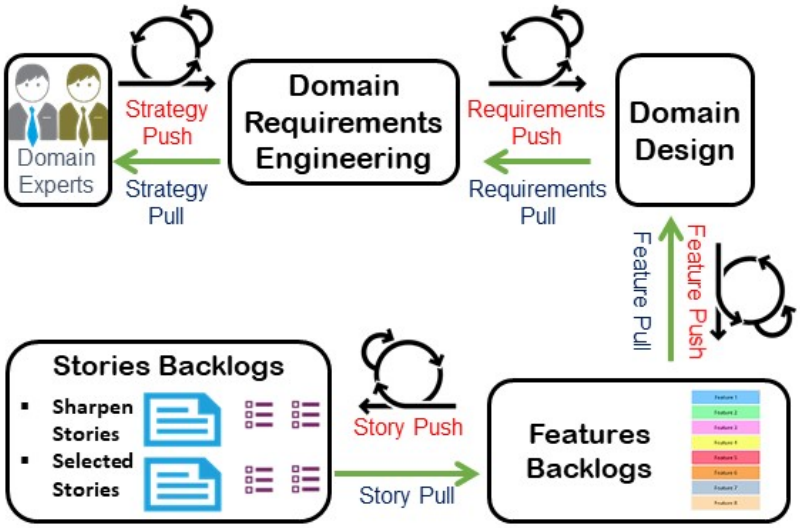

Fig. 8.3 Problem space of Domain Engineering in AgiFPL.

Figure 8.4 presents the problem space of the Application Engineering tier. The concerned requirements engineering process of this tier starts with the goals and the intentions of a specific product owner. These personal goals and intentions are studied, modeled and realized according to the proposed metamodel. For this stage, two optional methods are proposed. Based on the goals and intentions of the *"App i Owner"* and the context of the *"Product/App i Line"*, the *"Line i Development Team"* has to choose the method that best fits the context:

1. In the case where the *"Line i Development Team"* has to develop new reusable artifacts that do not exist within the common assets, the team applies the same process used for the domain-engineering phase;

2. In the case where some stakeholders' goals do not affect the product line, have not equivalent features in the common assets and concern a specific product, the *"Line i Development Team"* produces directly the User Stories and their Backlogs.

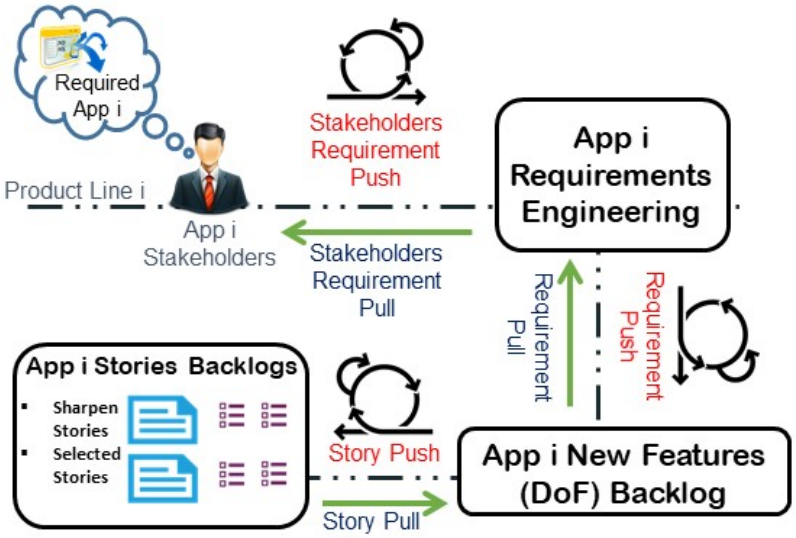

Fig. 8.4 Problem space of Application Engineering in AgiFPL - Product Line (i).

Making the relevant decisions for each tier allows analysts and developers of an agile product line to efficiently represent stakeholders' intentions and goals on the one hand and product line variabilities and commonalities on the other hand. AgiFPL is therefore, an agile methodology designed to improve the agility within the software product lines and effectively meet any newly emerged business expectations. Its main goal is to move teams from the classical approach to a more evolved APLE framework.

8.6 Applying the proposed metamodel – A concrete real-life example

A simple and short example related to an e-commerce product line is outlined below to describe and show the applicability of the proposed metamodel. The e-commerce case study is available in the SPLOT repository (Software Product Lines Online Tools[12]).

We first design the family goal model related to the case study and then follow the practices of the proposed metamodel to generate the correspondent feature model. Due to the limited space, only the application of Goals and Features sub-models is presented; the mapping from the goal model to its correspondent feature model and an example of user story.

Figure 8.5 shows a concrete *Family Goal Model (FGM)* of the *"Order Process"* related to the e-commerce case study (modeled using iStar 2.0). It represents the intentional elements and relations. For example, the goal **<Item_Available>** can be achieved by **<Prepare_and_Package_Item>**, by **<Obtain_From_Stock>**, and by **<Aquire_From_Supplier>**. In addition, satisfactions of the tasks **<Obtain_From_Stock>**, and **<Aquire_From_Supplier>** lead to satisfaction and dissatisfaction of the quality **<Avoid_Unsold_Stock>**. In fact, if the "sales department" adopts a "Make to Stock" strategy,

[12] http://www.splot-research.org/

it could result in unsold items. However, adopting the *"Make to Order"* strategy will help to avoid a stock of unsold items.

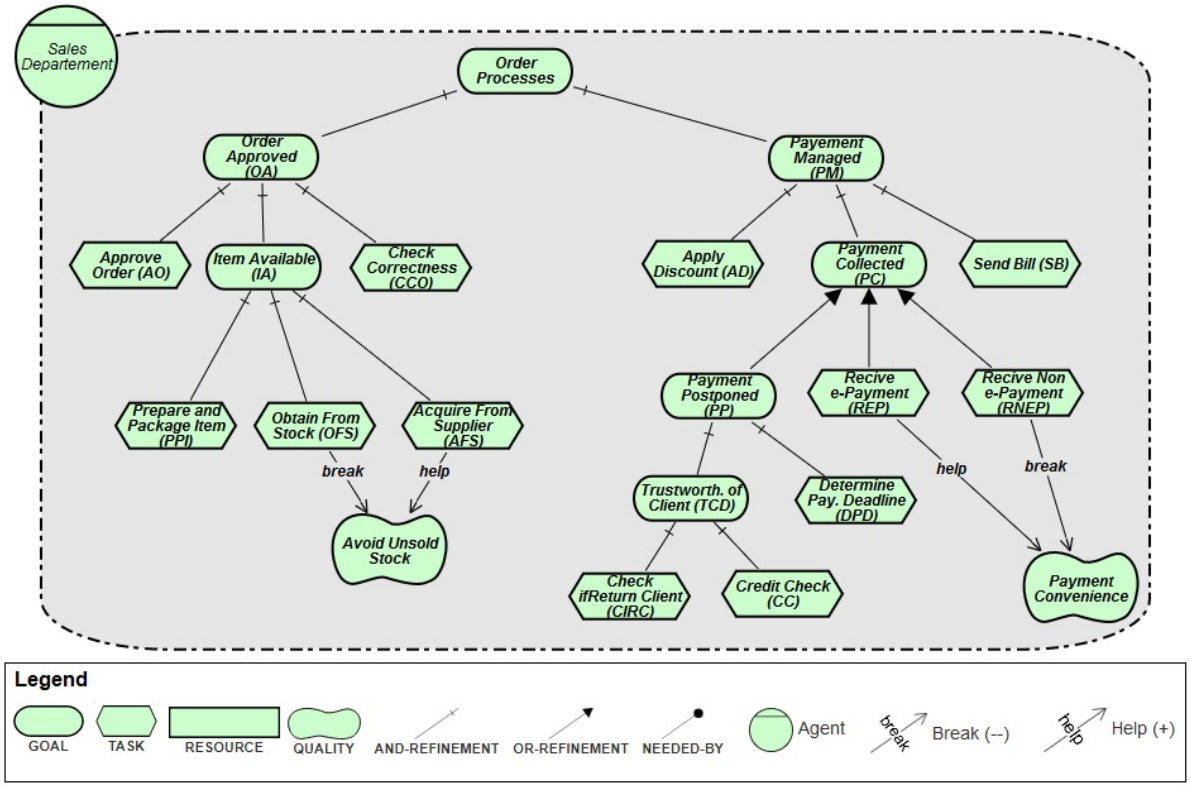

Fig. 8.5 A FGM for "Order Processes", modeled from the e-commerce case study.

According to the proposed framework, to represent a mapping, a mapping relation *(Φ)* should be developed for each mapped task. For example, the *Approve Order (AO)* task is mapped to the *Automatic Approval*, and *Manual Approval* features. Therefore, the mapping relation created is the following:

ΦAO (Approve Order, {Automatic Approval, Manual Approval}).

Moreover, the *Receive e-Payment (REP)* task is mapped to *Debit Card Payment*, *Credit Card Payment*, and *Payment Gateway*. Thus, the mapping relation created is the following:

ΦREP (Receive e-Payment, {Debit Card Payment, Credit Card Payment, Payment Gateway}).

Once the *"explicit mapping"* between tasks in the family goal model and features in feature model is executed, an *"implicit mapping"* between intermediate tasks, goals, and features can be started. The implicit mapping is performed between *"intentional relations"* in family goal models and *"feature relations"* in feature models. For example, following the proposed metamodel, it can be inferred that the goal *Payment Managed (PM)* in the family goal model (see Figure 8.5) is implicitly mapped to the feature *Payment Management (PMa)* (see Figure 8.6).

Note that, if a feature is mapped to more than one goal or/and task, then the corresponding feature appears in the mapping relations of all those goals or/and tasks.

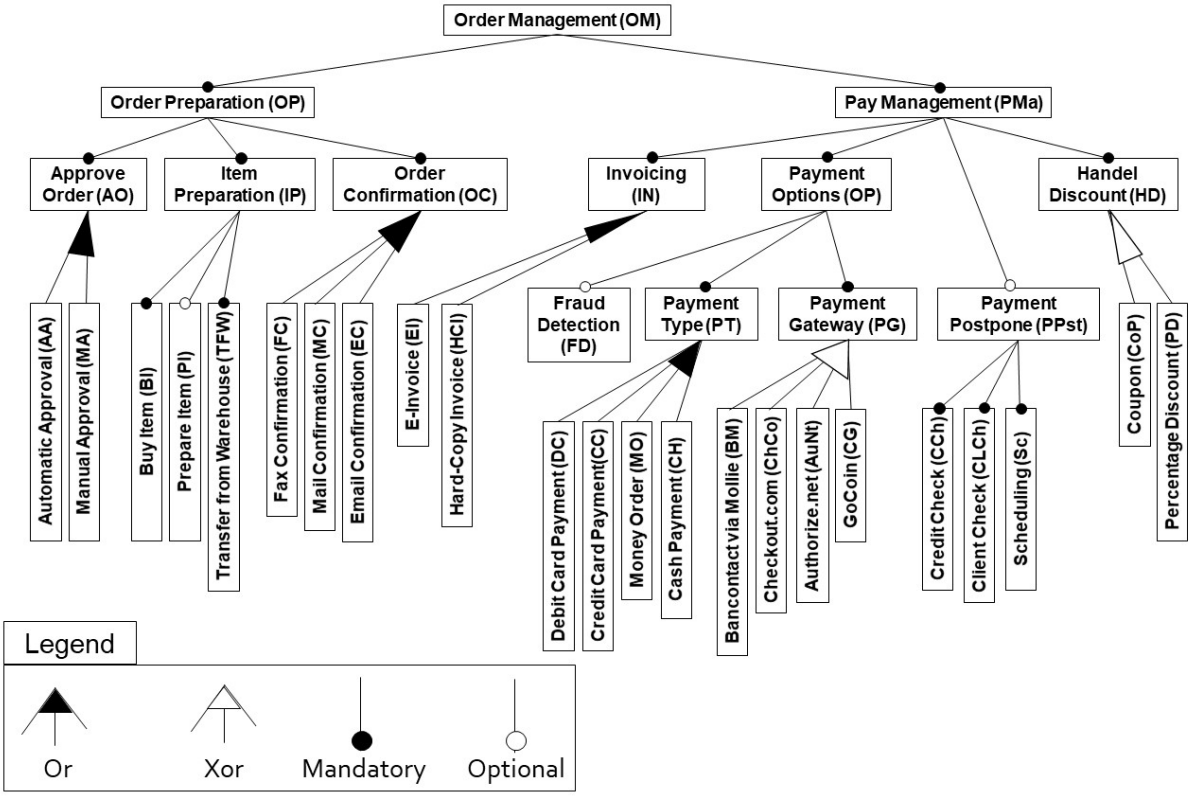

Fig. 8.6 Based on the FGM in Figure 8.5, the Correspondent Feature Model of "Order Process".

Figure 8.6 shows the corresponding feature model of the family goal model presented in Figure 4. The obtained feature model is represented using a tree graphical notation that could be translated into propositional logic. In addition, the feature model is generated according to the rules and practices of the proposed metamodel.

Based on the illustrated example, it was shown that the modeled family goal model of "Order Processes" (i.e. Figure 8.5) captures the intentional variability and describes the intentions behind existing features in the product line of e-commerce. Hereafter, we present some mappings as follows:

Order Processes (G-OP) = Order Management
Order Approved (G-OA) = Order Preparation
Payment Managed (G-PM) = Payment Management
Item Available (G-IA) = Item Preparation
Check Correctness of Order (T-CCO) = Order Confirmation (FC ∨ MC ∨ EC)
Obtain From Stock (T-OFS) = TFW
Acquire From Supplier (T-AFS) = BI
Apply Discount (T-AS) = (CoP ∨ PD)

Finally, as an illustration of user stories generated according to the proposed metamodel from the correspondent features, **<Invoicing>** could be realized by several user stories, such as:

As ⟨Accountant⟩, I want to ⟨Generate and Send Invoices⟩, so that ⟨the Invoice can be paid⟩

8.7 Conclusion

Being agile, while preserving a Software Product-Line structure, entails the adoption of an Agile Product Line approach that conforms to agile values and principles. Requirements engineering issues are the main elements to consider when integrating agility into software product lines. Collaborating with customers and responding quickly to changes are principles that have to be supported by any proposed (agile) process. Thus, any Agile Product Line method should assign a high priority to customers satisfaction through early and continuous delivery of valuable software from the start; and in addition accommodate any changing requirements, even until late in the process.

Following these guidelines, in this chapter, an integrated and consistent metamodel was proposed, for software analysts and developers who adopt Agile Product Line methods. The proposed metamodel allows capturing intentional variability and describing goals behind existing features in the agile product line. Therefore, by using family goal models it can be ensured that existing features and variability relations in feature models are aligned with intentional variability in the family goal models. In addition, it can trace back differences in products from differences in the intentions of stakeholders. Moreover, applying intentional elements within agile product lines not only facilitates identifying features in domain engineering, but also eases the selection of features based on stakeholder's intentions and needs in the application engineering.

This chapter compares how agile methods and product-line approaches take in charge "Requirements", "Architecture", and "Reuse". It also studies the requirements engineering approaches, tools or practices within the identified publications that propose agile product lines models or approaches. Identifying and studying RE techniques of relevant agile product lines approaches has helped to propose a powerful, convenient, and performant framework that can be applied to the requirements engineering stage of feature-oriented agile product lines. By summarizing the RE Tools/approach and activities identified in studied agile product line approaches, we could point out that each approach covers the agile values and principles related to RE only partially. However, in our framework, we aim to cover all the agile values and principles (i.e., values 2 and 3, and principles 1, 2, 7 and 8 of the agile manifesto) by modeling the organizational and operational context of the Domain and Application Engineering tiers within the environment of a product line. The framework allows capturing intentional variability and describing the intentions behind existing features in the agile product line.

Our future priorities aim to develop a procedure to discover inconsistencies in mapping results (i.e., generated goal models and/or generated feature models). We will aim at extending the validation technique to cover other properties, such as safe composition (i.e., for all goal

models modeled within the Application Engineering (AE) tier, they have at least one correspondent feature model and vice versa).