1

Introduction

AWAIS RASHID, JEAN-CLAUDE ROYER AND ANDREAS RUMMLER

> He sat, in defiance of municipal orders, astride the gun Zam-Zammah on her brick platform opposite the old Ajaib- Gher – the Wonder House, as the natives call the Lahore Museum. Who hold Zam-Zammah, that 'fire-breathing dragon', hold the Punjab.
>
> (Rudyard Kipling, *Kim*)

As the size and complexity of software systems grows, so does the need for effective modularity, abstraction and composition mechanisms to improve the reuse of software development assets during software systems engineering. This need for reusability is dictated by pressures to minimise costs and shorten the time to market. However, such reusability is only possible if these assets are variable enough to be usable in different products. Variability support has thus become an important attribute of modern software development practices. This is reflected by the increasing interest in mechanisms such as software product lines (Clements & Northrop, 2001) and generative programming (Czarnecki & Eisenecker 2000). Such mechanisms allow the automation of software development as opposed to the creation of custom 'one of a kind' software from scratch. By utilising variability techniques, highly reusable code libraries and components can be created, thus cutting costs and reducing the time to market.

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Core assets are produced and reused in a number of products that form a family. These core assets may be documents, models, etc., comprising product portfolios, requirements, project plans, architecture, design models and, of course, software components.

Thus, software product lines provide a systematic means to manage variability in a suite of products. Their potential benefits involve identification of variants

and expression of interrelationships between variants as well as their relationships with the core assets. Furthermore, the product line design allows for specific variations while leaving room for future extensions. This requires underpinning implementation technologies that support flexible yet modular implementations. Last, but not least, a product line approach supports the management of evolution and consistency in multi-product systems.

One view of product line development is as an up-front design and engineering investment that pays-off at some point after enough applications within the family are generated – the so-called *proactive product line engineering* (Clements & Krueger, 2002). An alternative view of a product line is to see it as an accumulation of investment in addition to an up-front investment. The more similar applications we build for a given domain, the more we tend to reuse the previously developed behaviour in the new applications. This view calls for support for generalisation of reusable assets out of concrete products via systematic refactoring – the so-called *reactive* or *extractive product line engineering* (Clements & Krueger, 2002).

Turning an up-front and/or accumulation of investment into a really profitable product line is not only a matter of discipline and architectural refactoring but also a matter of development technology support for doing so. There are three major barriers that stand in the way of reaping the full benefits of software product lines as noted above. First, there is the challenge of scale: a large number of variants may exist in a product line context and the number of interrelationships and dependencies can rise exponentially especially as one attempts to understand the interactions amongst variants as they are refined from requirements to implementation. Second, variations tend to be systemic by nature in that they affect the whole architecture of the software product line. Finally, software product lines often serve different business contexts, each having their own intricacies and complexities.

The AMPLE approach (short for aspect-oriented, model-driven, product line engineering), which is the focus of this book, aims to tackle the three major barriers described above by combining advances in aspect-oriented software development and model-driven engineering. The approach has been developed by a consortium of six leading research centres in the areas of software product lines, aspect-oriented software development and model-driven engineering, and three industrial organisations working with or seeking to deploy product line solutions. The efforts were co-funded by the European Commission FP6 funding programme over the period 2006–2009.

In this chapter, we first provide an overview of software product lines, model-driven engineering and aspect-oriented software development. This is followed by an overview of the AMPLE approach and its tool chain. The subsequent chapters in the book provide detailed discussions of the various facets of the approach and the tools.

1.1 Software product line engineering

Product line engineering (PLE) is a common principle in engineering disciplines other than software engineering. It has its origins in the need for individualised products instead of standardised products for the mass market.

Examples of product lines are manifold and can be found in different domains. A very good and illustrative one is the market for mobile phones. Most vendors of mobile phones cover a wide range of customer demands, ranging from simple entry models to feature-rich models targeted at the business-user segment of the market. In the context of PLE each model is called a *product*. Products are distinguished from each other by certain characteristics, but also share common characteristics. From an engineering perspective and also from an economic point of view it doesn't make sense to develop each product separately. This would mean setting up 20 different development teams for the creation of 20 different mobile phone models, which would prevent the utilisation of any synergies during the development. Instead, it is desirable to create a set of common components that can be reused during the development of all models. In the context of PLE such a set of components (no matter how tightly integrated they are) is called a *platform*. A platform consists of a set of core artefacts on top of which all products are built. The platform also contains base technologies which allow the *derivation* of products. The derivation of products describes the process of their creation. The mechanisms used in the derivation process are ideally powerful enough to execute this process automatically. However, in reality this process is only semi-automatic.

The products are differentiable from each other by features. A *feature* is a characteristic of a product that is visible to the end-user in some way. To pick the example of mobile phones, features may change both the hardware and software utilised by a particular model. Examples of features are the availability of a touch screen or a keyboard to enter telephone numbers, a built-in digital camera, a music player, a GPS or a radio receiver, or simply the possibility to connect to GSM and/or UMTS networks. Features may have relations among each other; or they may be independent from each other and built into a system in parallel (e.g. a camera and a music player). They may also exclude each other (e.g. a touch screen and a keyboard) or they may require the presence of each other (e.g. a firmware module for controlling a radio receiver requiring the presence of the receiver itself).

Obviously the concept of features is bound to (potentially physical) objects that can be reused during the development of products. *Reuse* plays an important role in PLE and is one of the key drivers in the engineering process. Reuse also implies that all artefacts, which serve as building blocks for products, need to

be managed somehow. We also introduced the concept of a platform. A platform combines and manages all available artefacts, which serve as the technical basis on top of which all products are built. In systems based on a layered architecture, a platform usually denotes one certain layer, which provides abstractions for a higher layer. However, in PLE a platform is more than this. In addition to technical building blocks (e.g. a digital camera module or a GPS receiver) it also captures artefacts from all development stages (including non-physical ones), ranging from requirements over architectural blueprints to test cases. It also includes supplemental elements such as documentation or even development methodologies. Overall, a platform in PLE comprises all elements that are common to all possible products and that are needed to create single products out of the product line.

1.1.1 Commonality and variability

Having established the concepts of a platform from which single products are derived, it is now possible to take a closer look at the relationship between the platform and the derived products. When examining products it becomes obvious that these products share certain characteristics while they differ in others. In the context of PLE they have properties *in common*, while they *vary* in others. These are key concepts in PLE; consequently PLE is basically about managing *commonality* and *variability*. Both concepts are handled in the platform itself. Common features are reused on an as-is basis in products, whilst variable elements need to be configured first and are used upon request only. Configuration in this context means choosing between several of options. In order to cope with variability in a systematic way, the concepts of *variation points* and *variants* have been developed. A variation point is a property exposed by the platform that can be altered in some way (i.e. set to a certain value). A variant is formed when a variation point is bound to a certain value. An example in the area of mobile phones is the digital camera module. The presence of such a module can be a variation point, which may offer a certain set of possible values, i.e. modules with 2, 3 and 5 megapixels. Choosing one particular module (i.e. the 3 megapixel model) resolves the variability and forms a variant.

To better illustrate these concepts, Figure 1.1 depicts the basic concepts of PLE graphically.

1.1.2 Benefits of product line engineering

The motivation behind PLE is manifold. First, of course, is the reduction in development costs. This reduction does not come per se, as there is a substantial up-front

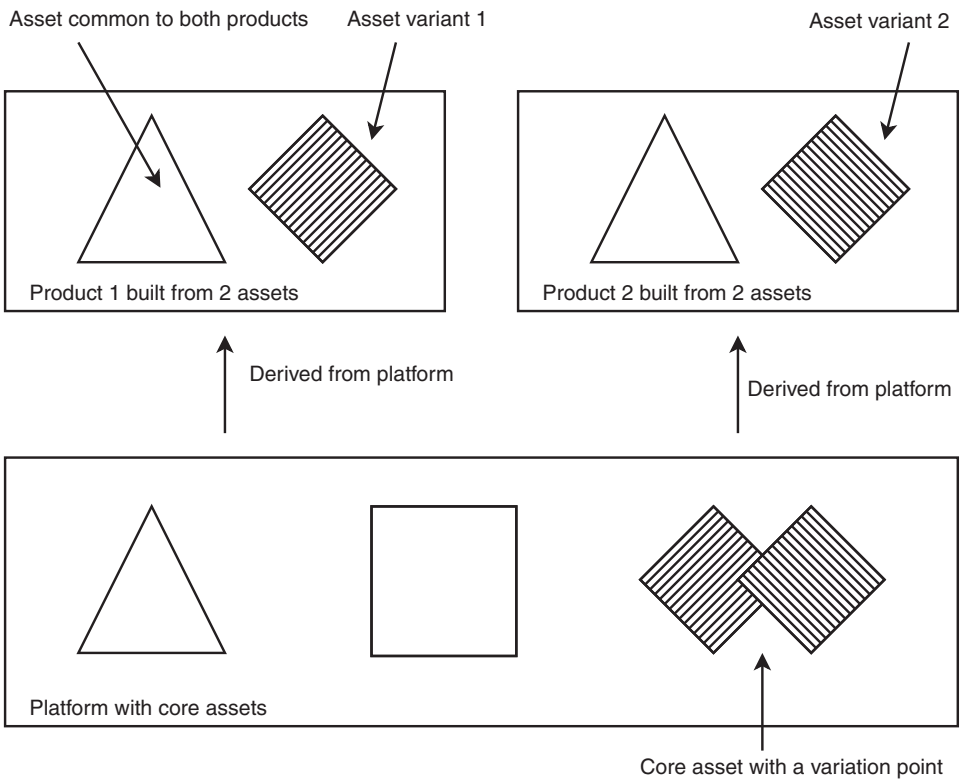

Figure 1.1 Basic concepts of product line engineering.

investment to be made. In single-system engineering the total costs for n individual systems is equal to n times the costs for each single system. In contrast, in PLE there are additional costs for creating the platform. There is also a significant overhead in creating the reusable artefacts that form the PL platform. However, deriving single products results in significantly lower costs as the major development effort has already been done before and does not need to be repeated. It is hard to give a general statement on the break-even point, where setting up a product line pays off. However, Weiss and Lai (1999) have shown that this point often lies between three to four systems.

Another benefit that is closely related to the one discussed above is the reduction of the time-to-market. Ideally a product can be created by only using existing artefacts from the platform – no product-specific development needs to be undertaken. While this is an ideal case, it is nevertheless not unrealistic. In general, the actual product creation (derivation) process can be semi-automated or even fully automated. As a large amount of development work does not need to be carried out,

the time to create a ready-to-ship product can be drastically reduced. The actual time required depends on the product derivation infrastructure and the amount of dedicated functionality (not available as platform asset(s)) that needs to included in the product. In addition, as the product derivation infrastructure exists and its characteristics are known, the estimation of costs when creating a new product can be much more accurate.

Another aspect which is also a major benefit for the application of product lines is the increase in product quality and the reduction of maintenance costs. Artefacts provided by the platform are used in different products in (potentially) different environments. This contributes to the process of stabilising the functioning of assets and increases the chance of detecting errors. In return, flaws can be fixed at a single point and the resulting changes can be forwarded to all products containing the affected asset.

1.1.3 Domain and application engineering

Following the basics outlined above, it becomes quite obvious that there is no straightforward sequential development process in PLE. Instead, there are two different processes that are, however, interrelated. Both the platform and the derived products need to be created. The process of establishing the platform is called *domain engineering* (DE), while the creation of products is referred to as *application engineering* (AE).

Domain engineering encompasses all activities in constructing the platform. In this process commonality and variability is defined and components capable of handling the variability are created. In addition, tools are prepared that can be used to resolve and bind the variability when it comes to engineering the derived products. As outlined above the platform not only consists of reusable components, it also consists of tools/methods for using and managing those components. The creation of all of those elements is part of DE. As a bottom line, in DE the product line owner defines what parts are variable and how this variability is exposed to the user (of the products that are derived, i.e. the product manager responsible for a single product). Therefore, DE defines the scope in which products can be constructed.

In application engineering actual products are derived by using the elements created in DE. This incorporates the application of methods and/or tools the platform provides, the reuse of existing components and the binding of variability according to specific application requirements (resulting in component variants). An important aspect that complements these activities is the creation of dedicated application-specific components used in only one product. This step may not be necessary in some cases; however, it is much more likely that a

product needs to be complemented with dedicated functionality, which is also part of AE and may be quite expensive depending on the complexity of the required functionality.

Both DE and AE processes run in parallel – completely decoupled development processes for both are impossible. Instead, both influence each other. The platform must be designed in such a way that it is profoundly beneficial for AE. Most of the foreseeable features should be provided by components of the platform in order to raise the level of reuse to the highest possible degree (which results in minimised development costs). On the other hand, a newly requested feature for one product might also be introduced into other products, which would make it a candidate for incorporation into the platform. A close collaboration between platform and product owners is necessary to prevent a drift in functionality between platform and products. Over time, this would result in a platform that doesn't serve its purpose of being a basis for products and creates an unnecessary cost overhead when the same functionality is implemented many times in different products. Therefore, the design of the platform must already be prepared in a way that it supports evolution over time and already captures the possible directions of future functionality. For this reason careful platform design is key to the successful application of PLE techniques.

1.1.4 Product line engineering for software

The introduction to product line engineering above was not given with a specialised focus on software engineering. But it is obvious that although there is no reason not to apply PLE concepts to the development of software, the process of creating software is different in some aspects from other engineering disciplines. Therefore, the following question needs to be discussed: what issues need to be solved when setting up a software product line engineering (SPLE) process?

The processes of domain engineering and application engineering are similar to each other and also similar to normal processes in software engineering. They go through all stages ranging from requirements engineering via design and implementation to testing, maintenance and evolution. The biggest difference is that in DE and AE different artefacts are created. While DE concentrates on creating reusable elements and templates or stubs that can be used later in AE, AE itself uses and completes these elements and templates to create actual products. These elements encompass all kinds known from software engineering such as requirements documents, documentation, architectural models, source code, test cases, and so on. Most elements are in some way 'componentised' in order to ease the task of composing and configuring them in AE.

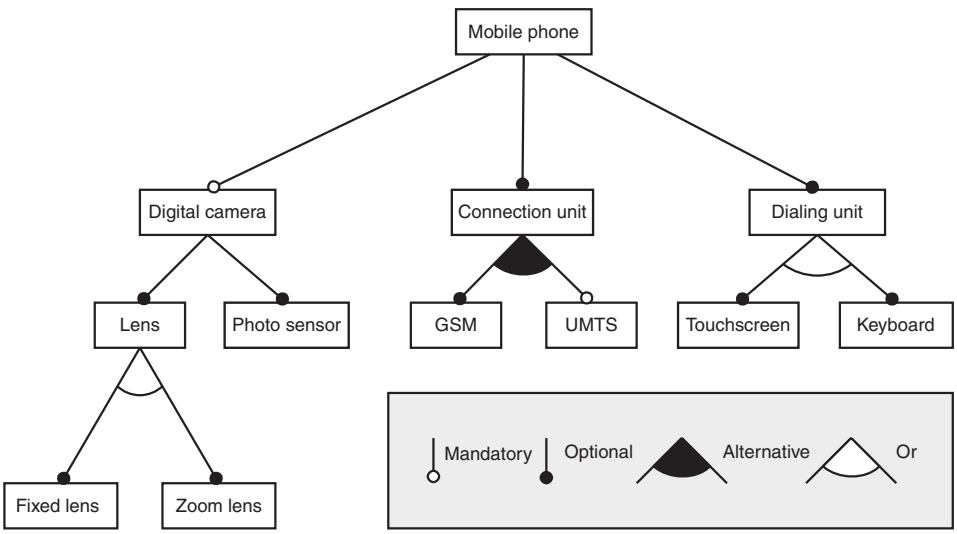

Figure 1.2 Example of a feature diagram for a product line of mobile phones.

Collecting components in some repository in the hope that they will be reused in an SPL is not sufficient. To ease the task of product derivation, which should be automated as far as possible, it is necessary to describe what components are available, what variable characteristics they have and how this variability is resolved. The way these descriptions are made and the way they are used and evaluated is the biggest difference between PLE for software engineering and PLE for other domains. An introduction to modelling and models is given in the next section; however, we anticipate a little at this point and give an introduction to very basic variability description mechanisms employed in SPLE.

We have already introduced the notion of a feature. However, we did not discuss the possible interrelations among features. We defined a feature as a characteristic of a system that is somehow visible to the user. Systems usually contain many different features that can be grouped together. By grouping features the capabilities of the system can be described in one (possibly large) tree, called a *feature tree*. Single features in this tree might have different attributes and relations. Some features may be mandatory in a product, some may be optional. Some may require other features to be included as well (dependency) or may interfere with the inclusion. Even alternatives to select from are possible. The grouping of all features including their interrelations is captured in a so-called *feature model*. These feature models were first introduced in Kang *et al*. (1990) and are now a widespread way of expressing the structure of a product line, and are used as a first-order input during product derivation. Feature models are often visualised in feature diagrams. An example of such a diagram is given in Figure 1.2.