Chapter 3

Aligning the economic modeling of software reuse with reuse practices

3.1 Introduction

Software reuse is the development of new software assets by reusing and extending existing software assets, such as libraries, frameworks, and components (Krueger, 1992). Potential benefits of software reuse include reduced development time and costs, shortened time-to-market, and improved software quality and maintainability. Schmid and Verlage (2002) argue that some organizations have been able to reduce development costs and time-to-market by a factor of ten or more by applying software reuse. Other estimates show that the required workforce can be reduced by a factor of four (Clements and Northrop, 2002). Software reuse, however, does not automatically result in cost reduction. Reuse processes and standards have to be implemented, variations among related applications have to be taken into account, and reusable assets have to be catalogued in a reuse library. The development of reusable software should therefore be considered an investment (Barnes and Bollinger, 1991).

Reuse economic models quantify the economic feasibility of this investment. The benefits of software reuse are generally defined in terms of the cost difference between the development of applications without reuse and the development of applications with reuse. Software reuse is considered to be economically feasible if the accumulated benefits exceed the total costs of developing software assets for reuse.

A state-of-the-art review leads to the assertion that existing reuse economic models are based on two assumptions that do not reflect software reuse practices very well. First, many assume that the cost of reusing a software asset depends on its size (measured in terms of source lines of code or another size-related metric), including Mili et al. (2001), Böckle et al. (2004), and Boehm et al. (2004). Size, however, is not always a good predictor of this cost. For example, "off-the-shelf" components are often adjusted to the context in which they are used through parameterization (Carney and Long, 2000; Ravichandran and Rothenberger, 2003). The cost of reusing such a component will therefore primarily be determined by the number of parameters that need to be configured, which depends on the commonality among applications in the target domain, rather than on its size. Second, existing reuse economic models obtain the cost of developing a software asset for reuse by multiplying the cost of developing the asset for one-time use by the "Relative Cost of Writing for Reuse" multiplier, see e.g. Lim (1996) and Wiles (1999). This approach reflects the case when reusable assets are developed from scratch, because "reusability" is added to every part of these assets (Wiles, 1999). In practice, however, software reuse is often applied recursively, i.e. applications are developed by reusing and extending one or more existing software assets that have been developed with reuse as well (Bosch, 2001; Jaaksi, 2002; Van Ommering, 2005). Existing reuse economic models cannot be used in this situation, as they do not account for the possibility that software assets are both developed for and with reuse.

The contribution of this chapter is that it provides modeling elements that are better aligned with software reuse practices. It obtains these elements by first distinguishing three different mechanisms for achieving software reuse–composition, black-box variation, and white-box variation–and then quantifying the associated costs. By using the modeling elements, a reuse economic model is constructed that can assist practitioners in deciding whether or not to apply software reuse recursively. The functioning of the model is illustrated in an example.

The remainder of this chapter is organized as follows. First, the three different mechanisms of software reuse are distinguished and the principle of recursive reuse is explained. Next, existing reuse economic models are reviewed and their assumptions are discussed. Subsequently, the modeling elements are derived and the reuse economic model is constructed. Finally, the chapter concludes with a brief summary and suggestions for further research.

3.2 Software reuse

When constructing a reuse economic model, two characteristics of software reuse have to be taken into account. First, many forms exist, such as software components, object-oriented frameworks, component frameworks, and product families. To be widely applicable, a reuse economic model should be an abstract representation of these forms of reuse. Van Ommering and Bosch (2002) provide such an abstraction. They show that each form is based on the mechanisms of composition and variation. Section 3.2.1 discusses these mechanisms. Second, large-scale software reuse is not always directly targeted at building applications. The functionality that is provided by a set of generic software assets is sometimes extended by adding functionality that is useful to a smaller group of applications. A hierarchical grouping of software assets can then be distinguished at software design-time, which is explained in Section 3.2.2.

3.2.1 Composition and variation

When software reuse is applied, a new software asset is created by reusing and extending one or more existing software assets. In general, two (often combined) mechanisms for achieving software reuse can be distinguished: composition and variation (Van Ommering and Bosch, 2002). Composition refers to the integration of two or more existing software assets that have been developed independently from each other (Sommerville, 2004; Törngren et al., 2005), whereas variation or tailoring refers to the ability to adjust a software asset to the context in which it is used (Bosch, 2000; Van Gurp et al., 2001; Sinnema et al., 2004). For example, in component-based software engineering, a new software system is obtained by integrating several preexisting components (Brown and Wallnau, 1996; Brereton and Budgen, 2000; Crnkovic et al., 2002). Software product line engineering, in contrast, is primarily targeted at variation: software systems are derived from a common set of core assets using variation points to implement the differences (Bosch, 2000; Clements and Northrop, 2002).

When applying composition, a new software asset is obtained by assembling two or more existing assets. Integration code is then required to compose the reused assets. An important form of composition is component-based software engineering (CBSE) (Brown and Wallnau, 1996; Brown and Wallnau, 1998; Szyperski, 1998; Brereton and Budgen, 2000; Crnkovic et al., 2002). In this approach, software components, i.e. "unit[s] of composition with contractually specified interfaces and explicit context dependencies only" (Szyperski, 1998), are assembled to create an application. When CBSE is applied systematically, components are integrated by using an architectural infrastructure, also referred to as component framework, that mediates and regulates component interaction (Brown and Wallnau, 1996; Brown and Wallnau, 1998; Crnkovic et al., 2002). Wrappers may be required to adapt the components to fit within the infrastructure and/or to reconcile the interfaces of incompatible components (Brown and Wallnau, 1996; Brereton and Budgen, 2000; Sommerville, 2004). Components can also be composed without the use of a component framework. Then, custom-developed code, often referred to as "glue" (Brown and Wallnau, 1996), is needed to handle the non-functional concerns that are otherwise managed by the framework.

The components being reused may not provide sufficient functionality to satisfy all the functional requirements (Brereton and Budgen, 2000). One or more new components are then created to realize this functionality. The development of new components can also be regarded as a kind of composition, because they are either obtained by assembling smaller-grained components or by composing lower-level platform functionality, such as numerical computation and file access (Krueger, 1992; Messerschmitt and Szyperski, 2003; Atkinson and Kühne, 2005). A software component that is written in Java, for example, will depend on a large set of predefined Java classes that comprise the Java platform (Flanagan, 2002).

When applying variation, the behavior of a reusable asset is configured to fit the requirements at hand. The places at which this behavior can be customized are generally referred to as variation points (Bosch, 2000; Van Gurp et al., 2001; Sinnema et al., 2004). Each variation point is associated with a number of alternative choices called variants. Figure 3.1 gives a schematic overview of a reusable asset Z (the central rectangle) and its variation points, which are numbered A, B, and C. The variants associated with A and B are shown as well (in the right and left ellipse, respectively). A variation point is bound by selecting a certain variant from the set of associated variants (Van Gurp et al., 2001; Sinnema et al., 2004), see also Figure 3.1. For example, suppose that variation point A is a parameter. To configure this parameter, a value has to be selected from the set {A1, A2, A3, A4}

A variation point is open if the set of variants can still be extended and closed otherwise (Sinnema et al., 2004). In Figure 3.1, open and closed variation points are represented by dashed and solid squares, respectively. Variation point A is closed, so the set {A1, A2, A3, A4} cannot be extended. In contrast, variation point B is open, so the set {B1, B2, B3, B4} can be extended by developing a new variant, say B5, thereby increasing the number of variants to select from during variation point binding.

Rather than being independent, new variants have to function in the context of a larger whole, i.e. the asset Z on which variation is applied. An un-

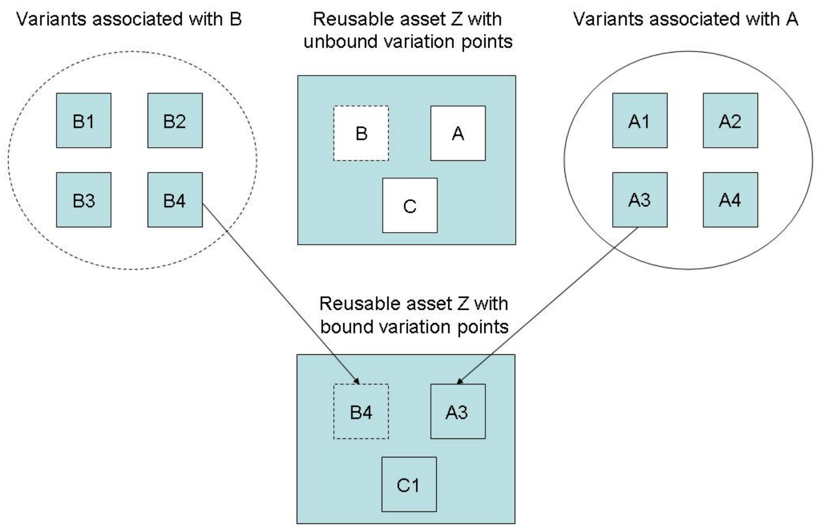
Variants associated with B
Reusable asset Z with unbound variation points
Variants associated with A
B1
B2
B3
B4
B
A
C
A1
A2
A3
A4
Reusable asset Z with bound variation points
B4
A3
C1

Figure 3.1: *A reusable asset with variation points, numbered A, B, and C, and associated variants.*

derstanding of this context is then required. New variant development can therefore be regarded as a form of white-box reuse: the use of "a software fragment, through its interfaces, while relying on the understanding gained from studying the actual implementation" (Szyperski, 1998). In this chapter, the term white-box variation is used to refer to the process of specializing Z by first developing one or more new variants, such as B5, and then configuring this asset by binding its variation points. In contrast, black-box reuse refers to "the concept of reusing implementations [such as Z] without relying on anything but their interfaces and specifications" (Szyperski, 1998). We use the term black-box variation to refer to the configuration of a reusable asset without developing new variants.

The specialization of an object-oriented framework is a well-known example of white-box variation. An object-oriented framework has certain classes that can be extended through subclassing. These classes constitute the framework's variation points. Each new subclass inherits the methods of the extended superclass and is regarded as a variant. Software developers cannot extend and/or override these methods without an understanding of the internal structure of this framework (Pree, 1994; Fayad and Schmidt, 1997). The development of a component plug-in within a product family is another example of white-box variation, since it requires software developers to understand the internal functioning of this product family (Bosch, 2000). The configuration of a commercial-of-the-shelf (COTS) component is a typical example of black-box variation (Ravichandran and Rothenberger, 2003).

Variation should not be confused with adaptation, which refers to the customization of a reused asset by rewriting part of its source code, see e.g. Bosch (2000). Many of the benefits that are associated with software reuse disappear when code changes are made. For example, a programmer must thoroughly understand this part of the source code, and unit testing of the adapted asset has to be repeated. As Krueger (1992) puts it:

> In the worst cases, a software developer spends more time locating, understanding, modifying and debugging a ... code fragment than the time required to develop the software from scratch.

Adaptation is therefore not considered in this chapter.

To summarize, three different mechanisms for achieving software reuse have been distinguished: composition, black-box variation, and white-box variation; see also Figure 3.2. The labels and multiplicities attached to the arrows indicate how these forms of reuse interact with one or more reusable assets to create a new software asset. The multiplicity 2..* indicates two or more.

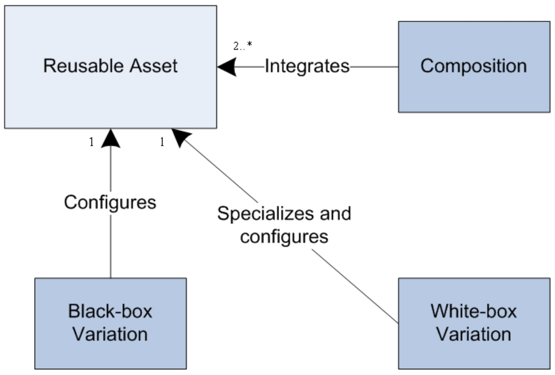

Figure 3.2: *Three different mechanisms for achieving software reuse and how they interact with one or more reusable assets.*

3.2.2 Applying software reuse recursively

In practice, software reuse is often applied recursively, in the sense that concrete applications are obtained by reusing and extending one or more existing software assets that have also been developed with reuse (Bosch, 2001; Jaaksi, 2002; Van Ommering, 2005). For example, Bosch (2001) argues that organizations that apply the product line approach sometimes develop specialized assets for a subset of the applications in the product line. Concrete applications are derived from these assets by adding application-specific functionality. The specialized product line assets, in turn, have been obtained by reusing and extending a set of generic product line assets.

Nokia is an example of an organization that has adopted this approach to develop browser products that allow mobile phone users to access services over wireless telecommunication networks (Jaaksi, 2002). A generic version of the browser is offered as a software product to external customers. A Nokia and a Symbian OS version of this browser have been obtained from the generic version of the browser by including platform-specific adaptations and interfaces. Finally, each Nokia mobile phone has its own specifications for the browser.

The way in which Philips deals with commonality among embedded software systems in consumer electronics is another example of recursion in software reuse (Van Ommering, 2005). Philips engages in a range of software intensive products such as televisions, video recorders, and audio systems. Each product group is organized as a separate product line. Philips has achieved reuse among product lines by introducing an extra layer of subsystems. Although embedded systems from different product lines differ significantly from each other, they can still be based on this common set of subsystems.

The examples illustrate that when software reuse is applied recursively, generic software assets are combined in multiple ways into domain-specific assets, which, in turn, are combined in multiple ways into applications. A hierarchical grouping of software assets can then be distinguished at design-time (Van Ommering, 2002). Each level of this hierarchy consists of software assets that have been obtained by reusing and extending one or more assets from the level below, except the ones at the lowest level which have been developed from scratch. Figure 3.3, which is derived from Van Ommering (2002), gives an example of such a hierarchy. The applications at the top level are derived from the domain-specific asset at the intermediate level by adding functionality that is specific to these applications: the first one is obtained by adding C1 and the second one is obtained by adding C2 and C3. The domain-specific asset, in turn, has been developed by reusing and extending three generic assets, i.e. A1, A3, and A5.

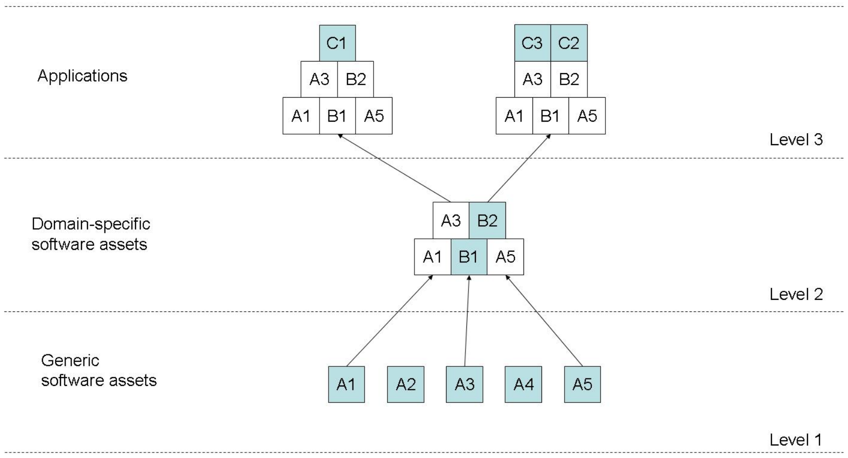

Figure 3.3: *A hierarchical grouping of software assets. Each asset in this hierarchy has been obtained by reusing and extending one or more assets from the level below.*

3.3 Economic analysis of software reuse

Reuse economic models quantify the trade-off between software development with reuse and software development without reuse. Many reuse economic models have been proposed in the literature, see e.g. Gaffney and Durek (1989), Barnes and Bollinger (1991), Gaffney and Cruickshank (1992), Poulin et al. (1993), Pfleeger and Bollinger (1994), Poulin (1997a), Favaro et al. (1998), Mili et al. (2001), Böckle et al. (2004), Boehm et al. (2004), Nazareth and Rothenberger (2004), and Ben Abdallah Ben Lamine et al. (2005). The benefits of software reuse are generally defined in terms of the cost difference between the development of applications without reuse and the development of applications with reuse. Software reuse is considered to be economically feasible if the accumulated benefits exceed the total costs of developing software assets for reuse. Thorough surveys have been conducted by Frakes and Terry (1996), Lim (1996), Poulin (1997b), and Wiles (1999). This section reviews the reuse economic models presented in Mili et al. (2001), Böckle et al. (2004), and Boehm et al. (2004), which are not included in these surveys, as they are more recent. The rationale for selecting Mili et al. (2001) is that this model encompasses seventeen earlier ones. In addition, the SoCoEMo-PLE 2 model (Ben Abdallah Ben Lamine et al., 2005) is based on it. The other two models are selected because they are representative among many other reuse economic models.

3.3.1 Overview

Mili et al. (2001) present a reuse economic model for component-based software engineering. The cost of developing a component for reuse is determined by multiplying the cost of developing this component for one-time use by the "Relative Cost of Writing for Reuse" multiplier. The cost of developing an application with reuse consists of: (i) the reuse adoption costs, (ii) the cost of developing the fraction of this application that consists of custom-developed software, and (iii) the cost of locating, instantiating, and integrating the reused components. The authors do not provide an expression for the cost of developing custom software, but argue that this cost can be derived by using traditional software cost estimation models. The same applies to the integration costs of the reused components. Furthermore, it is assumed that the reuse adoption costs, e.g. the costs of training, tool acquisition, and the operational impact of reuse processes, and the component retrieval and assessment costs can be estimated by expert judgment. Finally, the cost of instantiating a reused component is assumed to be a fixed fraction (either 0.20 or 0.67, depending on whether the component's source code is adapted) of the cost of developing this component from scratch.

Böckle et al. (2004) provide a reuse economic model for software product line engineering. They assume that the cost of establishing a software product line consists of three elements: the costs for an organization to adopt the product line approach, the cost of developing a reusable core asset base, and the cost of developing concrete applications by reusing the product line assets. The authors split an application developed with reuse into a part consisting of reused product line software and one consisting of custom-developed software. The cost of reusing product line software is assumed to be 10% of the cost of building equivalent software from scratch, whereas the cost of developing the product line assets is assumed to be 150% of the cost of developing this software for one-time use.

The Constructive Product Line Investment Model (COPLIMO) by Boehm et al. (2004) is an extension of the COCOMO II model for software cost estimation (Boehm et al., 1995). The authors divide an application that is developed with reuse into a product-specific fraction, an adapted-software fraction, and a reused-software fraction. Adapted software is existing software that has been adjusted to the requirements at hand by modifying part of its source code, whereas reused software is existing software that has been black-box reused. To establish the cost of developing an application with reuse, equivalent amounts of custom-developed software are determined for the adapted-software and the reused-software fractions of this application by applying the COCOMO II reuse model. The cost of developing a software asset for reuse is obtained by multiplying the cost of developing similar software for one-time use by the "Relative Cost of Writing for Reuse" multiplier, defined as the product of the COCOMO II cost drivers "Development for Reuse", "Required Reliability", and "Degree of Documentation".

3.3.2 Discussion

The commonality among the models reviewed above is the following. Each model is built around the "Relative Cost of Writing for Reuse" (RCWR) and the "Relative Cost of Reuse" (RCR) factors introduced by Poulin (1997b): RCWR represents the cost of developing software for reuse relative to the cost of developing this software for one-time use, while RCR represents the cost of reusing existing software relative to the cost of developing this software from scratch. The cost of developing software for reuse is obtained by multiplying the cost of developing this software for one-time use by the RCWR multiplier. Similarly, the cost of developing the fraction of an application that consists of reused software is obtained by multiplying the cost of custom-developing this fraction by the RCR factor. Mili et al. (2001) and Boehm et al. (2004) use different RCR factors for existing software that is reused without modification and existing software that is adapted by rewriting part of its source code. Böckle et al. (2004) do not make such a distinction.

By applying the RCR factor, the reuse economic models above assume that there is a correlation between the cost of reusing existing software and the cost of custom-developing this software. In general, such a correlation will exist if some other factor underlies both of these costs, causing them to vary together. The cost of developing custom software is primarily determined by the size of this software, measured in terms of source lines of code or another size-related metric; see e.g. Banker et al. (1994), Boehm et al. (1995), and Agarwal et al. (2001). For example, in the COCOMO II model, a widely used software cost estimation model, the effort required to develop a software system is expressed as a function of this system's size (in thousands of source lines of code), using an exponential factor to account for possible economies or diseconomies of scale (Boehm et al., 1995). A correlation between the cost of reusing existing software and the cost of custom-developing this software will therefore only exist if the cost of reusing existing software depends on its size.

In section 3.2.1, it has been argued that the complexity of reusing existing software depends on whether this software is black-box or white-box reused. Thus, in order to determine whether the cost of reusing existing software depends on its size, the distinction between black-box and white-box reuse has to be taken into account. White-box reuse refers to the use of an existing software asset, while relying on the understanding gained from studying its implementation. The effort required to study a reused asset's implementation is therefore included in the cost of white-box reuse. This effort will generally depend on the size of the reused asset. In the COCOMO II model, for example, the effort required to study a reused asset's implementation depends linearly on its size (Boehm et al., 1995). Size can therefore be used as a predictor of the cost of white-box reuse.

Black-box reuse, in contrast, refers to the use of an existing software asset without knowledge of its implementation. The cost of black-box reuse is therefore primarily determined by the effort required to configure the reused asset. This effort will generally depend on the number of variation points, which is determined by the degree of commonality among applications in the target domain (Atkinson et al., 2000; Lycett, 2001; Schmid, 2002). As a consequence, relatively large software assets can have few variation points, whereas relatively small software assets can have many variation points. Size can therefore not always be used as a predictor of the cost of black-box reuse.

To summarize, a correlation between the cost of reusing existing software and the cost of custom-developing this software will generally exist in the situation of white-box reuse. It is less likely, however, that this correlation exists when black-box reuse is applied. The RCR factor should therefore not be used to estimate the cost of black-box reuse.

The RCWR multiplier can be used to capture an important characteristic of software reuse: it is more expensive to develop a software asset for reuse than to develop a similar asset for one-time use (Boehm et al., 1995; Poulin, 1997b). The reuse economic models above obtain the cost of developing a software asset for reuse by multiplying the cost of developing the asset for one-time use by the RCWR multiplier. This approach reflects the case when reusable assets are developed from scratch, because "reusability" is added to every part of the asset (Wiles, 1999). In Section 3.2.2, however, it has been argued that software reuse is sometimes applied recursively in the sense that applications are developed by reusing and extending one or more existing software assets that have also been developed with reuse. A software asset that is both developed for and with reuse can be divided into a part that consists of custom-developed software and a part that consists of reused software. No additional effort is required to generalize the part of the asset that consists of reused software, as this part has already been developed for reuse. Thus, in order to estimate the cost of developing for reuse correctly, the RCWR multiplier should only be applied to the part of the asset that consists of custom-developed software. To conclude, existing reuse economic models cannot be used when software reuse is applied recursively, as they do not account for the possibility that software assets are both developed for and with reuse.

3.4 Modeling elements

The previous section showed that existing reuse economic models are based on two assumptions that do not reflect software reuse practices very well. In this section, a set of modeling elements is provided that is better aligned with these practices. In general, two interrelated trade-offs apply to the development of a software asset: (1) whether to develop this asset with or without reuse, and (2) whether to develop this asset for reuse or for one-time use. The first trade-off depends on whether the effort needed to tailor the reused assets to fit the requirements at hand is less than the effort needed to develop the required functionality from scratch (Ravichandran and Rothenberger, 2003). The costs of developing a software asset with reuse are quantified in Section 3.4.1. The second trade-off, also referred to as reuse infrastructure scoping (Schmid, 2002), depends on whether the expected benefits of reusing this asset offset the cost of making it reusable. The cost of developing a software asset for reuse is quantified in Section 3.4.2.

3.4.1 The cost of developing with reuse

In Section 3.2.1, three different mechanisms for achieving software reuse have been distinguished: composition, black-box variation, and white-box variation. Below, the associated costs are quantified.

When applying composition, a new software asset is created by assembling two or more existing assets. Integration code—either wrappers or glue, depending on whether the reused assets are inserted into a component framework—is then required to compose the reused assets. In addition, custom-developed code may be required to provide missing functionality. Let S_{in} denote the total amount of integration code, and let S_{new} denote the total amount of new functionality, both measured in terms of thousands of source lines of code (KSLOC). Similar to COCOMO II (Boehm et al., 1995), we formulate the cost of developing custom software as:

$$C_c = \alpha_c \cdot S_{custom}^{\beta_c}, \tag{3.1}$$

where α_c reflects the average cost of developing a thousand source lines of custom code, $S_{custom} = S_{in} + S_{new}$ the total amount of custom-developed code, and β_c an exponential factor to account for possible economies or diseconomies of scale in writing lines of code (Banker et al., 1994). If $\beta_c < 1$, doubling the amount of custom-developed software increases development costs by a factor of less than two, so writing lines of code exhibits economies of scale. In contrast, $\beta_c > 1$ implies that writing lines of code exhibits diseconomies of scale. Finally, if writing lines of code exhibits neither economies nor diseconomies of scale, $\beta_c = 1$.

When applying variation, the behavior of an existing software asset is configured by binding its variation points. Let V denote the number of variation points belonging to a reused asset. It is assumed that the accumulated costs of binding variation points can be expressed as:

$$C_b = \alpha_b \cdot V^{\beta_b}, \tag{3.2}$$

where α_b is the average cost of binding an individual variation point and β_b an exponential factor to account for possible dependencies among variation points. These occur when the selection of a certain variant at one variation point constraints the selection of variants at other variation points. For example, the binding of variant A1 to variation point A may exclude the binding of variant B1 to variation point B or may require the binding of variant C1 to variation point C, see e.g. Sinnema et al. (2004). If there are no dependencies among variation points, the accumulated costs of binding variation points are assumed to be equal to the sum of the costs of binding individual variation points, i.e. $\beta_b = 1$. On the other hand, the presence of dependencies increases the complexity of binding variation points, so $\beta_b > 1$.

The set of variants associated with an open variation point is sometimes extended by developing a new variant. In Section 3.2.1, it has been argued that new variant development is a form of white-box reuse, so an understanding of the reused asset's implementation is then required. Let S_{total} denote the overall size of a reused asset, measured in KSLOC. The cost of studying the asset's implementation is assumed to depend linearly on its size:

$$C_s = \alpha_s \cdot S_{total}, \tag{3.3}$$

where α_s is the average cost of studying a thousand source lines of code. Again, this assumption is in line with the COCOMO II model, which obtains this cost by multiplying the size of the asset by the "software understanding increment" (a factor that ranges from 0% to 50%, depending on (i) the structure, clarity, and self-descriptiveness of the reused asset, and (ii) the programmer's familiarity with it) (Boehm et al., 1995). In addition, custom-developed code is required to develop the variant itself. Equation (3.1) can be used to account for this cost by increasing S_{custom} by the size of the new variant (measured in KSLOC).

Equations (3.1), (3.2), and (3.3) quantify three different sources of costs associated with software reuse. Table 3.1 shows how these costs apply to the mechanisms of composition, black-box variation, and white-box variation as described in Section 3.2.1:

- Equation (3.1) covers the cost of connecting the reused assets when applying composition.
- When applying black-box variation, a reusable asset is configured by binding variation points. Equation (3.2) can be used to determine this cost.
- In the situation of white-box variation, a reusable asset is specialized by extending the set of variants associated with one or more of its open variation points. An understanding of the reused asset's internal structure is then required. Equation (3.3) covers the cost of studying the reused asset's implementation, whereas the cost of developing the new variants is covered by Equation (3.1). Finally, Equation (3.2) is needed to obtain the cost of configuring the reusable asset by binding its variation points.

Table 3.1: *The mapping of Equations (3.1)-(3.3) to the mechanisms of composition, black-box variation, and white-box variation. A cross in the (i, j)-th entry of the table indicates that Equation j applies to mechanism i.*

	Equation (3.1)	Equation (3.2)	Equation (3.3)
Composition	X		
Black-box variation		X	
White-box variation	X	X	X

3.4.2 The cost of developing for reuse

It is more expensive to develop a software asset for reuse than to develop similar software for one-time use. For example, domain analysis is required to identify variation points (Atkinson et al., 2000; Thiel and Hein, 2002) and software that is developed for reuse needs extra documentation and testing (Boehm et al., 2004; Sommerville, 2004). We assume that the "Relative Cost of Writing for Reuse" multiplier (Poulin, 1997b) can be used to account for these additional costs:

$$C_{fr} = \mathrm{RCWR} \cdot C_c, \tag{3.4}$$

where C_{fr} is the cost of generalizing the part of a software asset that consists of custom-developed software and RCWR the "Relative Cost of Writing for Reuse" multiplier.

In Section 3.3.2, it has been argued that existing reuse economic models obtain the cost of developing a software asset for reuse by multiplying the cost of developing the asset for one-time use by the RCWR multiplier. Equation (3.4) differs from this approach in the sense that the RCWR multiplier is only applied to the part of the asset that consists of custom-developed software. Both approaches yield the same result when reusable assets are developed from scratch. However, our approach gives lower estimates when software reuse is applied recursively.

3.5 The reuse economic model

In Section 3.2.2, it has been argued that when software reuse is applied recursively, a hierarchical grouping of software assets can be distinguished at design-time. Given such a hierarchy, a distinction can be made between the economic value of the software assets at a particular level of this hierarchy and the economic value of the total reuse investment, which includes each level of reusable assets. This section focuses on determining the economic value of the software assets at the intermediate level of a hierarchy of three levels. In particular, by using the modeling elements from the previous section, we develop a reuse economic model for determining the economic feasibility of having such an intermediate layer of reusable software assets. The practical relevance of the model is as follows. Suppose that a company has developed one or more generic software assets, i.e. reusable assets that apply to several application domains. Now, this company may decide to create applications by first deriving a set of domain-specific assets and then adding application-specific functionality. Alternatively, applications can be derived directly from the generic assets. Our model supports practitioners in making this decision.

Let us assume that the sets $L_1 = \{a_{1,1}, \ldots, a_{1,n_1}\}$, $L_2 = \{a_{2,1}, \ldots, a_{2,n_2}\}$, and $L_3 = \{a_{3,1}, \ldots, a_{3,n_3}\}$ represent the three levels of the design-time hierarchy. The bottom level, the intermediate level, and the top level are indexed as 1,2, and 3, respectively. The notation employed in the model is as follows:

n_k	Number of software assets at level k
β_c	Exponential factor to account for possible economies and diseconomies of scale in writing custom code
β_b	Exponential factor to account for possible dependencies among variation points
α_c	Average cost of developing a thousand source lines of custom code
α_b	Average cost of binding a variation point
α_s	Average cost of studying a thousand source lines of custom code
RCWR	Relative cost of writing for reuse
$S_{total}(a_{j,t})$	Overall size of asset $a_{j,t}$, measured in thousands of source lines of code
$S_{custom}(a_{j,t})$	Size of custom-developed software for asset $a_{j,t}$, measured in thousands of source lines of code
$V(a_{j,t})$	Number of variation points belonging to asset $a_{j,t}$
$I_r(a_{i,s}, a_{j,t})$	Binary variable which equals 1 if asset $a_{i,s}$ is reused when asset $a_{j,t}$ is developed and 0 if otherwise
$I_{wb}(a_{i,s}, a_{j,t})$	Binary variable which equals 1 if asset $a_{i,s}$ is white-box reused when asset $a_{j,t}$ is developed and 0 if otherwise

$C(a_{j,t}, k)$	Cost of developing asset $a_{j,t}$ by reusing and extending one or more assets from level k
B	Accumulated benefits of reusing the assets from the intermediate level
TC	Total costs of developing the assets at the intermediate level

Software assets at the intermediate level have both been developed for reuse and with reuse. By combining Equations (3.1)-(3.4), the cost of developing software asset $a_{2,t}$ can be written as:

$$\begin{aligned} C(a_{2,t}, 1) = \{\alpha_c \cdot (S_{custom}(a_{2,t}))^{\beta_c}\} \cdot \text{RCWR} \\ + \alpha_b \cdot \sum_{s=1}^{n_1} V(a_{1,s})^{\beta_b} \cdot I_r(a_{1,s}, a_{2,t}) \\ +\alpha_s \cdot \sum_{s=1}^{n_1} S_{total}(a_{1,s}) \cdot I_{wb}(a_{1,s}, a_{2,t}). \end{aligned} \tag{3.5}$$

The variable $I_r(a_{1,s}, a_{2,t})$ is equal to 1 if asset $a_{1,s}$ is reused when asset $a_{2,t}$ is developed and 0 if otherwise. Similarly, the variable $I_{wb}(a_{1,s}, a_{2,t})$ is equal to 1 if asset $a_{1,s}$ is white-box reused when asset $a_{2,t}$ is developed and 0 if otherwise. Because white-box reuse is a special case of reuse, it holds that $I_{wb}(a_{1,s}, a_{2,t}) \leq I_r(a_{1,s}, a_{2,t})$. If asset $a_{1,s}$ is black-box reused when asset $a_{2,t}$ is developed, $I_r(a_{1,s}, a_{2,t}) = 1$ and $I_{wb}(a_{1,s}, a_{2,t}) = 0$. The cost of reusing this asset is then equal to the cost of binding its variation points. Similarly, if asset $a_{1,s}$ is white-box reused when asset $a_{2,t}$ is developed, $I_r(a_{1,s}, a_{2,t}) = I_{wb}(a_{1,s}, a_{2,t}) = 1$. Then, an additional cost element is taken into account: the cost of studying the reused asset's implementation. Recall that the cost of developing one or more new variants is accounted for by increasing $S_{custom}(a_{2,t})$ by the size of these variants.

The total costs of developing the assets at the intermediate level are equal to the sum of the costs of developing the individual assets at this level:

$$TC = \sum_{t=1}^{n_2} C(a_{2,t}, 1). \tag{3.6}$$

The accumulated benefits of reusing the assets from the intermediate level are obtained by subtracting the costs of developing applications by reusing and extending the assets from the intermediate level from the costs of developing these applications by reusing and extending the assets from the bottom level. The cost of developing application $a_{3,t}$ by reusing and extending the assets from level k can be formulated as:

$$\begin{aligned} C(a_{3,t}, k) = \alpha_c \cdot (S_{custom}(a_{3,t}))^{\beta_c} \\ + \alpha_b \cdot \sum_{s=1}^{n_k} V(a_{k,s})^{\beta_b} \cdot I_r(a_{k,s}, a_{3,t}) \\ + \alpha_s \cdot \sum_{s=1}^{n_k} S_{total}(a_{k,s}) \cdot I_{wb}(a_{k,s}, a_{3,t}). \end{aligned} \tag{3.7}$$

The accumulated benefits of reusing the assets from the intermediate level will then be equal to:

$$B = \sum_{t=1}^{n_3} (C(a_{3,t}, 1) - C(a_{3,t}, 2)). \tag{3.8}$$

If $B \geq TC$, the accumulated benefits of reusing the assets from the intermediate level of the hierarchy offset the costs of developing these assets, so the intermediate level is economically feasible. In contrast, $B < TC$ implies that developing the assets at the intermediate level of the hierarchy increases software development costs.

The additive forms of Equations (3.5) and (3.7) are based on the assumption that the effort required to understand and configure one of the reusable assets does not affect the effort required to understand and configure the other reusable assets. This assumption is likely to hold, because reusable assets have been developed independently from each other. Similarly, the additive forms of Equations (3.6) and (3.8) suggest that developing one of the intermediate assets (applications) does not affect the effort required to develop the other intermediate assets (applications). This assumption is plausible if each of these assets is developed by a different programmer. However, when the same programmer develops several of them, interactions may come in play. For example, suppose that this programmer reuses a particular asset from the bottom level repeatedly in developing the intermediate assets. Then, the effort required to study this asset's implementation is likely to decrease every time the asset is reused. Equations (3.6) and (3.8) ignore these interactions.

3.6 Example

The following example illustrates the functioning of our model. The example is based on a real-life situation, yet simplified. Company A provides information systems to a large number of customers spread across several market segments. Within a market segment, information systems are derived from a generic framework called branch model. Although branch models from different market segments differ significantly from each other, they can still be based on a common set of business objects. Examples of business objects include ′contract′, ′order′, ′customer′, and ′product′. Over time, the number of branch models has been extended to address new market segments. Currently, management faces the decision whether to develop a branch model for the healthcare domain. To account for the differences among customers, management decides to offer 3 variants of the healthcare information system.

To determine whether it is economically feasible to develop a branch model for the healthcare domain, it is assumed that business objects and branch models are black-box reused. In addition, it is assumed that software development exhibits constant returns to scale and that there are no dependencies among variation points, i.e. $\beta_b = \beta_c = 1$.

Management estimates that the size of custom-developed software for the healthcare branch model will be 30 KSLOC and that 15 business objects will be reused. A business object has on average 10 variation points. The average cost of developing a thousand source lines of custom code has been normalized to 1. Suppose that the average cost of binding a variation point is equal to 0.02 and that the relative cost of writing for reuse is equal to 1.5. Then, the cost of developing the healthcare branch model will be equal to (Equation (3.6)):

$$30 \cdot 1.5 + 0.02 \cdot 15 \cdot 10 = 48.$$

The healthcare branch model will have 50 variation points. Suppose that 5 KSLOC is custom-developed if a variant of the healthcare information system is obtained by reusing and extending the healthcare branch model and that 15 KSLOC is custom-developed if this variant is built directly on top of the business objects. Finally, assume that 12 business objects will be reused if an information system variant is obtained by composing business objects. For 3 variants of the healthcare information system, the accumulated benefits of reusing the healthcare branch model will then be equal to (Equation (3.8)):

$$3 \cdot ((15 + 0.02 \cdot 12 \cdot 10) - (5 + 0.02 \cdot 50)) = 3 \cdot 11.4 = 34.2.$$

The accumulated benefits of reusing the healthcare branch model do not offset its development costs, so management decides to build the 3 variants of the healthcare information system directly on top of the business objects. In this example, developing the healthcare branch model pays off at 5 variants.

3.7 Conclusions and future research

Existing reuse economic models rely on two assumptions that do not reflect software reuse practices very well. First, they assume that there is a correlation between the cost of reusing existing software and the cost of custom-developing this software. Although such a correlation will generally exist in the situation of white-box reuse, we have argued that it is less likely to exist when black-box reuse is applied. Second, existing reuse economic models obtain the cost of developing a software asset for reuse by multiplying the cost of developing the asset for one-time use by the "Relative Cost of Writing for Reuse" multiplier. This approach reflects the case when reusable assets are developed from scratch. In practice, however, software reuse is often applied recursively in the sense that applications are developed by reusing and extending one or more existing software assets that have been developed with reuse as well. A different approach is then required to estimate the costs that are associated with developing for reuse.

This chapter has provided modeling elements that are better aligned with software reuse practices by quantifying the costs that are associated with three different mechanisms for achieving software reuse: composition, black-box variation, and white-box variation. It has also shown that when software reuse is applied recursively, a hierarchical grouping of software assets can be distinguished at design-time. Each level of this hierarchy consists of software assets that reuse and extend one or more assets from the level below. By using the modeling elements, a reuse economic model has been constructed that allows one to determine whether the intermediate level of such a hierarchy is economically feasible. An example has shown how our model may serve as a tool for practitioners to support decisions such as whether or not to develop one or more domain-specific software assets by reusing and extending a set of generic software assets.

Future research will primarily involve empirical validation of the assumptions that underlie the modeling elements. In particular, we are interested in testing the proposed correlation between the cost of black-box variation and the number of variation points. Effort will also be directed at extending the reuse economic model. Typically, the life cycle of a software asset consists of two phases: the development phase, in which the first version of this asset is created, and the evolution phase, in which upgrades are produced. In its current form, the model applies to the development phase only. A natural way to extend the model is therefore to include annual maintenance costs. In addition, revenue streams from selling applications to customers can be taken into account, so that the profitability of alternative software development strategies can be compared. Finally, it has been assumed that reusable assets are developed internally. In practice, third-party components may be reused as well. In-house development costs should then be replaced by license fees and royalties.