9

REUSE AND DOMAIN ENGINEERING

Before software can be reusable it first has to be usable.
—Ralph Johnson

9.1 GENERAL IDEA

Software reuse continues to be practiced since the early days of programming. Separate compilation of Fortran II subroutines and increased use of Fortran subroutine libraries are examples of reuse practiced in the early days of programming [1]. Formal software reuse was first introduced by Dough McIlroy. He envisioned an industry for the development of reusable components and the industrialization of the production of application software from off-the-shelf components [2]. Other examples of early development of reuse include the concept of program families introduced by David Parnas [3] and the concepts of domain and domain analysis introduced by Jim Neighbors [4]. A set of programs with several common attributes and features is known as a program family. On the other hand, domain analysis means finding objects and operations of a set of similar software systems in a specific problem domain. A domain is a field of expertise or a specialized body of knowledge. The concept of program families is related to the concept of domain analysis [3, 5, 6]. In this context, software reuse involves two main activities: software development *with* reuse and software development *for* reuse.

Software Evolution and Maintenance: A Practitioner's Approach, First Edition.
Priyadarshi Tripathy and Kshirasagar Naik.

Intuitively, software reuse means using existing assets in the development of a new system. Reusable assets can be both reusable artifacts and software knowledge. There are four types of reusable artifacts as follows [7]:

- *Data reuse:* It involves a standardization of data formats. A standard data interchange format is necessary to design reusable functions.
- *Architectural reuse:* This means developing: (i) a set of generic design styles about the logical structure of software; and (ii) a set of functional elements and reuse those elements in new systems.
- *Design reuse:* This deals with the reuse of abstract design. A selected abstract design is custom implemented to meet the application requirements.
- *Program reuse:* This means reusing executable code. For example, one may reuse a pattern-matching system, that was developed as part of a text processing tool, in a database management system.

The reusability property of a software asset indicates the degree to which the asset can be reused in another project. For a software component to be reusable, it needs to exhibit the following properties that directly encourage its use in similar situations.

1. *Environmental independence:* This means that a component performs a task, and it makes minimal interactions with other components. Dependencies arise from the referencing of global data and assumptions made about the operational usage. The components with the environmental independence characteristic can be reused irrespective of the environment from which they were originally captured.
2. *High cohesion:* A component is said to have high cohesion, if its subsystems cooperate with each other to achieve a single objective. Components with high cohesion allow easy interpretation of their functionality.
3. *Low coupling:* If two components either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, there is said to be low coupling. Low coupling components are excellent candidates for reuse. If a component or function performs one specific task and has few or no dependencies, it will be much more reusable than if it performs multiple tasks, has many side effects, and needs other components.
4. *Adaptability:* Adaptability means being easily changed to run in a new environment.
5. *Understandability:* If a program component is easily comprehended, programmers can quickly make decisions about its reuse potential.
6. *Reliability:* It is the ability of a software system to consistently perform its intended function without degradation or failure.
7. *Portability:* It is the usability of the same software in different environment. Therefore, components that are portable are excellent candidates for reuse.

9.1.1 Benefits of Reuse

One benefits in several ways from software reuse. The most obvious advantage of reuse is economic benefit. Projects become more cost-effective if some existing assets are reused. Other tangible benefits of reuse are as follows [8, 9]:

1. *Increased reliability:* Reusable components tend to reveal more failures because of the extra efforts put in their design and their extensive usage.
2. *Reduced process risk:* Reusing components can reduce risks inherent in software projects, because a working, presumed fault-free component is being reused, thereby reducing much uncertainty.
3. *Increase productivity:* By reusing artifacts and processes, software engineers become more productive because their activities, namely, specification, design, implementation, and testing, consume less time and effort. For reuse to be justified, the time to find the reuse component and adapt it must be much less than the time needed to develop the component anew. A reuse library stores reuse assets and provides an interface to search the repository [10, 11].
4. *Compliance with standards:* Reusing code in a software system being developed can improve its compliance with standards, thereby raising its quality. For example, by reusing familiar user interfaces, the usability of a system is increased.
5. *Accelerated development:* Software development time can be reduced by reusing assets.
6. *Improved maintainability:* Reusable components generally possess some desired characteristics: modularity, compliance with standards, low coupling and high cohesion, and consistent programming style. Not surprisingly, these characteristics are very much sought in maintainable programs. Therefore, any attempt to raise the reusability of an asset will contribute to better maintainability.
7. *Less maintenance effort and time:* Reusable components are easy to understand and change during a software modification. A significant benefit is obtained while executing perfective maintenance for two reasons: (i) new components needed during perfective maintenance can be obtained from reuse libraries and (ii) almost 50% of maintenance cost is attributed to perfective maintenance [12].

9.1.2 Reuse Models

Development of assets with the potential to be reused requires additional capital investment. The organization can select one or more reuse models that best meet their business objectives, engineering realities, and management styles [13]. Reuse models are classified as *proactive*, *reactive*, and *extractive*.

- *Proactive approaches:* In proactive approaches to developing reusable components, the system is designed and implemented for all conceivable variations; this includes design of reusable assets. A proactive approach is

to product lines what the Waterfall model is to conventional software. The term product line development, which is also known as domain engineering, refers to a "development-for-reuse" process to create reusable software assets (RSA). This approach might be adopted by organizations that can accurately estimate the long-term requirements for their product line. However, this approach faces an investment risk if the future product requirements are not aligned with the projected requirements.

- *Reactive approaches:* In this approach, while developing products, reusable assets are developed if a reuse opportunity arises [14]. This approach works, if (i) it is difficult to perform long-term predictions of requirements for product variations; or (ii) an organization needs to maintain an aggressive production schedule with not much resources to develop reusable assets. The cost to develop assets can be amortized over several products. However, in the absence of a common, solid product architecture in a domain, continuous reengineering of products can render this approach more expensive.
- *Extractive approaches:* The extractive approaches [15] fall in between the proactive approaches and the reactive ones. To make a domain engineering's initial baseline, an extractive approach reuses some operational software products. Therefore, this approach applies to organizations that have accumulated both artifacts and experiences in a domain, and want to rapidly move from traditional to domain engineering.

9.1.3 Factors Influencing Reuse

Reuse factors are the practices that can be applied to increase the reuse of artifacts. There are several factors that influence software reuse. Frakes and Gandel [16] identified four major factors for systematic software reuse: *managerial*, *legal*, *economic, and technical*.

- *Managerial:* Systematic reuse requires upper management support, because: (i) it may need years of investment before it pays off; and (ii) it involves changes in organization funding and management structure that can only be implemented with executive management support. The management activities that directly support a reuse program are creating and managing incentives, educating personnel, and effecting a change in culture [17].
- *Legal:* This factor is linked with cultural, social, and political factors, and it presents very difficult problems. Potential problems include proprietary and copyright issues, liabilities and responsibilities of reusable software, and contractual requirements involving reuse. Use of third-party software makes this issue more important [18].
- *Economic:* Software reuse will succeed only if it provides economic benefits. A study by Favaro [19] found that some artifacts need to be reused more than 13 times to recoup the extra cost of developing reusable components.
- *Technical:* This factor has received much attention from the researchers actively engaged in library development [11], object-oriented development paradigm

[20], and domain engineering [21]. A reuse library stores reusable assets and provides an interface to search the repository. One can collect library assets in a number of ways: (i) reengineer the existing system components; (ii) design and build new assets; and (iii) purchase assets from other sources. Next, those reusable components are put through a *certification* process, including testing and verification, to assure that the components have the desired attributes [22]. Finally, the components are put into various categories for effective searching.

9.1.4 Success Factors of Reuse

An organization that attempts to implement a reuse program needs to address a broad spectrum of technical and nontechnical problems. It is difficult and risky to execute a successful reuse program. There is no common solution that all organizations can follow. Rather, each organization must analyze its own requirements, implement a process to measure the effectiveness of reuse, define the expected benefits, discover and eliminate impediments, and manage risks. The following steps aid organizations in running a successful reuse program [20]:

- Develop software with the product line approach.
- Develop software architectures to standardize data formats and product interfaces.
- Develop generic software architectures for product lines.
- Incorporate off-the-shelf components.
- Perform domain modeling of reusable components.
- Follow a software reuse methodology and measurement process.
- Ensure that management understands reuse issues at technical and nontechnical levels.
- Support reuse by means of tools and methods.
- Support reuse by placing reuse advocates in senior management.
- Practice reusing requirements and design in addition to reusing code.

9.2 DOMAIN ENGINEERING

The term domain engineering refers to a "development-for-reuse" process to create RSA. It is also referred to as product line development. Domain engineering is the set of activities that are executed to create RSA to be used in specific software projects. For a software product family, the requirements of the family are identified and a reusable, generic software structure is designed to develop members of the family. In the following, we explain analysis, design, and implementation activities of domain engineering.

- *Domain analysis* comprises three main steps: (i) identify the family of products to be constructed; (ii) determine the variable and common features in the family of products; and (iii) develop the specifications of the product family.

The feature-oriented domain analysis (FODA) method developed at the Software Engineering Institute [23] is a well-known method for domain analysis. The FODA method describes a process for domain analysis to discover, analyze, and document commonality and differences within a domain.

- *Domain design* comprises two main steps. (i) Develop a generic software architecture for the family of products under consideration; and (ii) develop a plan to create individual systems based on reusable assets. The design activity emphasizes a common architecture of related systems. The common architecture becomes the basis for system construction and incremental growth. The design activities are supported by architecture description languages (ADLs), namely, Acme [24], and interface definition languages (IDLs) [25], such as Facebook's Thrift.
- *Domain implementation* involves the following broad activities: (i) identify reusable components based on the outcome of domain analysis; (ii) acquire and create reusable assets by applying the domain knowledge acquired in the process of domain analysis and the generic software architecture constructed in the domain design phase; (iii) catalogue the reusable assets into a component library. Development, management, and maintenance of a repository of reusable assets make up the core of domain implementation.

Application engineering (a.k.a. product development) is complementary to *domain engineering*. It refers to a "development-with-reuse" process to create specific systems by using the fabricated assets defined in domain engineering [26]. Application engineering composes specific application systems by: (i) reusing existing assets; (ii) developing any new components that are needed; (iii) reengineering some existent software; and (iv) testing the overall system. Similar to the standard practices in software engineering [27], it begins by eliciting requirements, analyzing the requirements, and writing a specification. However, the requirements are specified in incremental chunks, called delta, from a set of generic system requirements developed during domain engineering.

Both domain and application engineering processes feed on each other, as illustrated in Figure 9.1. Application engineering is fed with reusable assets from domain engineering, whereas domain engineering is fed with new requirements from application engineering. A closed loop occurs between the two engineering processes because application engineers may discover that the present reusable assets do not cover some requirements. Therefore, some application-specific development and/or

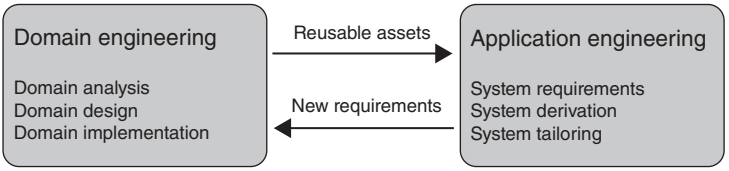

FIGURE 9.1 Feedback between domain and application engineering

tailoring is required [28]. To make the reusable assets consistent with the needs of the product, the new requirements are fed back into domain engineering.

In the following sections, we briefly discuss *nine* domain engineering approaches reported in the literature [13, 29, 30]: Draco, domain analysis and reuse environment (DARE), family-oriented abstraction, specification, and tTransportation (FAST), feature-oriented reuse method (FORM), "Komponentbasierte Anwendungsentwicklung" (KobrA), product line UML-based software engineering (PLUS), product line software engineering (PuLSE), Koala, and reuse-driven software engineering business (RSEB).

9.2.1 Draco

Based on transformation technology, Draco is the first prototype of the domain engineering approach proposed by Neighbors [5]. Neighbor's work provided an important contribution in domain engineering in the form of generative programming and transformation systems. However, it is very complex to apply in production environment.

9.2.2 DARE

A method and a toolset for performing domain engineering constitute DARE [31]. To make domain models, the DARE process utilizes three information sources: code, documents, and expert knowledge. The generic architectures, feature tables, and facet tables, and all models and information are stored in a domain book. Successful industry applications of DARE include building of database at Oracle [32].

9.2.3 FAST

The former Lucent Technologies (now Alcatel-Lucent) introduced the FAST method in 1999 [33] to develop telecommunication infrastructure. Applied in product line engineering, FAST defines a pattern of engineering processes. Three sub-processes constitute FAST: domain qualification (DQ), domain engineering (DE), and application engineering (AE). A product line that is worthy of investment is identified with domain qualification. Domain engineering enables the development of product line environments and assets. By using product line assets, application engineering develops products rapidly.

9.2.4 FORM

FORM, developed at Pohang University of Science and Technology [34, 35], is basically an extension of the FODA method [23]. FORM finds commonalities and differences in a product line in terms of features, and uses those findings to develop architectures and components for product lines. A feature model captures the commonalities and variabilities to support: (i) development of assets for reusable product lines; and (ii) development of products using the reusable assets. Therefore, two processes are key to FORM: asset development and product development.

9.2.5 KobrA

KobrA is a method for component-based application development [30]. The two main activities in KobrA are: *framework engineering* and *application engineering*. By means of framework engineering, one makes a common framework that manifests all variations in products making up the family. A set of components are organized in the form of a tree in a framework, and each component is described at the specification level and realization level. The specification level of abstraction described the components externally visible properties and behavior. The realization level of abstraction describes how the component satisfies the contracts with other components. Next, application engineering is applied on the framework to build specific applications.

9.2.6 PLUS

The UML-based modeling method to develop single systems has been extended in PLUS to support product lines [36]. In PLUS, (i) for requirements analysis activities, use-case modeling and feature modeling are provided; (ii) there are mechanisms to model the static aspects, dynamic interactions, state machines, and class dependency for product lines; and (iii) component-based software design and software architecture patterns are supported for product line design activity.

9.2.7 PuLSE

Product line software engineering (PuLSE) methodology was developed to enable the conceptualization and deployment of software product lines for large enterprises [37]. The PuLSE methodology comprises three key elements: (i) the deployment phases; (ii) the technical components; and (iii) the support components. The deployment phases describe activities for initialization, construction of infrastructure, usage of infrastructure, and management and evolution of product lines. The technical components describe how to operationalize the development of the product line. The support components are guidelines enabling better evolution, adaptation, and deployment of the product line.

9.2.8 Koala

Koala is a language to describe architectures for product lines for embedded software to be used in home appliances [38]. It was originally developed at Philips Corporation. To support product variations, diversity interfaces and switches are provided in Koala. Internal diversity of components are handled by means of diversity interfaces and the switches are used to route connections between interfaces. The extra functions of a component are treated as optional interfaces.

9.2.9 RSEB

RSEB is a use-case-driven reuse method based on UML [39]. It was designed to facilitate both asset reuse and the development of reusable software. RSEB supports both domain engineering and application engineering. The former comprises two processes: application family engineering and component system engineering. The

former is used to develop and maintain the overall layered system architecture. On the other hand, the latter is used to develop component systems.

9.3 REUSE CAPABILITY

Reuse capability concerns gaining a comprehensive understanding of the development process of an organization with respect to reusing assets and establishing priorities for improving the extent of reuse. It is defined by Ted Davis [40] as "the range of expected results in reuse proficiency, efficiency, and effectiveness that can be achieved by an organizations process" (p. 128). The concept of reuse *opportunities* is used as a basis to define reuse *efficiency* and reuse *proficiency*. An asset provides a reuse opportunity when the asset—to be developed or existing—satisfies an anticipated or current need. There are two broad kinds of reuse opportunities: *targeted* and *potential*. Targeted reuse opportunities are those reuse opportunities on which the organization explicitly spends much efforts. On the other hand, potential reuse opportunities are those reuse opportunities which will turn into actual reuse, if exploited. Not always a targeted opportunity turns into a potential opportunity.

Now we define reuse *proficiency* and reuse *efficiency* by means of

$$R_A, R_P, \text{ and } R_T,$$

where:

R_A counts the actual reuse opportunities exploited;
R_P counts the potential opportunities for reuse; and
R_T counts the targeted opportunities for reuse.

Reuse proficiency is the ratio R_A/R_P, and reuse efficiency is the ratio R_A/R_T. The benefits and costs of reusing assets in a project are succinctly captured by the concept of reuse *effectiveness*, which is represented as:

$$N(C_{NR} - C_R)/C_D,$$

where:

N = number of products, systems, or versions developed with the reusable assets;
C_{NR} = cost of developing new assets without using reusable assets;
C_R = cost of utilizing, that is, identifying, assessing, and adapting reusable assets;
C_D = cost of domain engineering, that is, developing assets for reuse and building a reuse infrastructure.

The concepts of reuse efficiency and reuse proficiency are used to measure the impacts of reuse capability. The linkage between the two concepts and reuse capability has been illustrated in Figure 9.2. The ovals denote the sets of potential, target, and actual reuse opportunities. For simplicity, assume that the areas of the ovals denote the counts of the assets corresponding to those opportunities. Now, in terms of the elements of the figure, reuse efficiency is calculated by dividing the area of the actual reuse oval by the area of the target oval. Similarly, reuse proficiency is calculated by

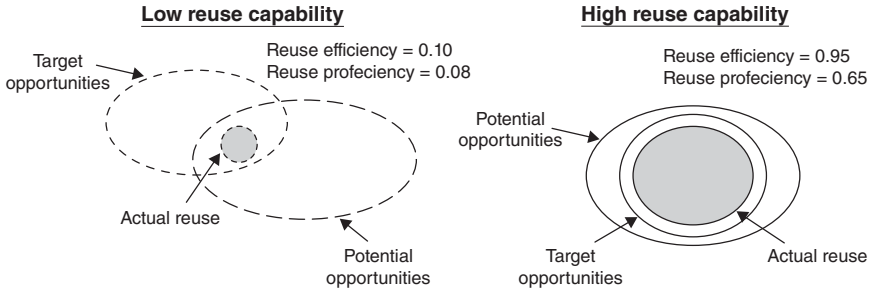

FIGURE 9.2 Reuse capability. From Reference 40. © 1993 IEEE

dividing the area of the actual reuse oval by the area of the oval representing potential reuse.

Two possible scenarios of low reuse capability and high reuse concerning the extent of reuse capability can be found in Figure 9.2. An informal approach to reuse results in low reuse, because potential opportunities—shown as dashes—are overlooked. In Figure 9.2, potential opportunities may exclude target opportunities, because: (i) the potential opportunities are unknown; and (ii) target opportunities may not be planned explicitly. Therefore, with many target opportunities falling outside the potential opportunities, the resulting reuse efficiency and proficiency become low. In addition, with resources being spent on many opportunities not resulting in actual reuse, reuse effectiveness becomes low.

9.4 MATURITY MODELS

A reuse maturity model (RMM) is an aid for performing planning and self-assessment to improve an organization's capability to reuse existing software artifacts. Such a model helps the organization's understanding of their existing and future goals for reuse activities. In other words, such a model can be used in planning systematic reuse. Organizations developing and maintaining multiple products, software systems, and versions are the primary stakeholders of the maturity models. In this section we discuss briefly three maturity models: reuse maturity model [41], reuse capability model [40], and RiSE maturity model [42].

9.4.1 Reuse Maturity Model

In circa 1991, Koltun and Hudson [41] presented the first RMM. The model provides a concise form of obtaining information on reuse practices in organizations. The model comprises 5 levels and 10 dimensions of reuse maturity as shown in Table 9.1. The columns of the table indicate the different levels of reuse maturity. Maturity improves on a scale from 1 to 5, where level 1 corresponds to Initial/Chaotic state and level 5 corresponds to the Ingrained state. Much automated tool support and measurement of reuse to track progress are the hallmarks of level 5. The rows of the table correspond to different dimensions of reuse maturity, namely, Planning to reuse

TABLE 9.1 Reuse Maturity Model

	Reuse Maturity Levels				
Dimension of Reuse	1. Initial/Chaotic	2. Monitored	3. Coordinated	4. Planned	5. Ingrained
Motivation/Culture	Reuse discouraged	Reuse encouraged	Reuse incentivized reinforced rewarded	Reuse indoctrinated	Reuse in the way we do business
Planning to reuse	None	Grassroots activity	Targets of opportunity	Business imperative	Part of strategic plan
Breadth of reuse	Individual	Work group	Department	Division	Enterprise wide
Responsible for making reuse happen	Individual initiative	Shared initiative	Dedicated individual	Dedicated group	Corporate group with division liaisons
Process by which reuse is leveraged	Reuse process chaotic; unclear how reuse comes in	Reuse questions raised at design reviews (after the fact)	Design emphasis placed on off-the-shelf parts	Focus on developing families of products	All software products are genericized for future reuse
Reuse assets	Salvage yard (no apparent structure to collection)	Catalog identifies language- and platform-specific parts	Catalog organized along application specific lines	Catalog includes generic data processing functions	Planned activity to acquire or develop missing pieces in catalog
Classification activity	Informal, individualized	Multiple independent schemes for classifying parts	Single scheme catalog published periodically	Some domain analyses done to determine categories	Formal, complete consistent timely classification
Technology support	Personal tools, if any	Many tools, but not specialized for reuse	Classification aids and synthesis aids	Electronic library separate from development environment	Automated support integrated with development environment
Metrics	No metrics on reuse level, payoff, or costs	Number of lines of code used in cost models	Maturity tracking of reuse occurrences of catalog parts	Analyses done to identify expected payoffs from developing reusable parts	All system utilities, software tools and accounting mechanisms instrumented to track reuse
Legal, contractual accounting considerations	Inhibition to getting started	Internal accounting scheme for sharing costs and allocating benefits	Data rights and compensation issues resolved with customer	Royalty scheme for all suppliers and customers	Software treated as key capital asset

Source: From Reference 43. © 1996 ACM.

and Motivation/Culture. For each row, the amount of organizational commitment increases as the organization moves from level 1 to level 5.

To benefit from this model, an organization evaluates its current maturity level before embarking on a reuse improvement program. Essentially, the organization identifies its placement on each of the 10 dimensions. Next, activities are performed to raise the level of reuse maturity. Upon reaching the highest level of reuse maturity, reuse activities become part of the business practice. The main obstacles to reaching the highest level of reuse are: technical, cultural, financial, institutional, and legal. This model was not applied in real case studies, but are considered as the key insights for the reuse capability model developed by Ted Davis [40].

9.4.2 Reuse Capability Model

The reuse capability model (RCM) [40] is used together with the reuse adoption process defined by the Software Production Consortium (SPC) [44]. The adoption process for reuse is a solution to implement a program for reuse, and it is based on the implementation model defined by Rubén Prieto-Diaz [14]. RCM comprises two models, namely, an assessment model and an implementation model. An organization can use the assessment model to: (i) understand its current capability to reuse artifacts; and (ii) discover opportunities to improve its reuse capability. A set of critical success factors are at the core of the assessment model. The success factors are described as goals that an organization uses to evaluate the present state of their reuse practice. The organization will get to know its strengths and learn opportunities for improvement. On the other hand, the organization can apply the implementation model in prioritizing the critical factor goals by grouping them into stages.

Assessment model: The success factors in the assessment model are grouped into four categories: application development, asset development, management, and process and technology. The critical success factors have been listed in Table 9.2. The various groups of critical success factors reflect the different views of reuse in an organization.

Now we briefly explain the four groups of factors one-by-one.

- Application Development Factors: The application development factors concern issues which are critical to the successful reuse of assets in the development of products. These factors relate to all the activities in the processes followed by the organization to develop software products.
- Asset Development Factors: The asset development factors concern issues which are critical to the successful development or acquisition of assets for reuse. These factors relate to all types of reusable assets, namely, requirements, architectures, design, source code, and test cases.
- Management Factors: The management factors concern issues which are critical to management's role in enabling reuse. These factors are quite broad, relating to such diverse aspects as products, business of the organization, finance, marketing, and product management.
- Process and Technology Factors: This group of factors concern general process issues across asset development, application development, and the management.

TABLE 9.2 Critical Success Factors

Application Development Factors	Asset Development Factors	Management Factors	Process and Technology Factors
Asset awareness and accessibility	Needs identification	Organizational commitment	Process definition and integration
Asset identification	Asset interface and architecture definition	Planning and direction	Measurement
Asset evaluation and verification	Needs and solution relationships	Cost and pricing	Continuous process improvement
Application integrability	Commonality and variability definition	Legal and contractual constraints	Training
	Asset value determination		Tool support
	Asset reusability		Technology innovation
	Asset quality		

Each critical success factor is defined in terms of one or more goals. The critical success factors and the corresponding goals have been described in the guidebook for adopting reuse [44]. The goals describe *what* is to be achieved—and not *how* those goals can be realized. Therefore, there is much flexibility in achieving those goals. As an example, the *needs identification* factor has the following goals:

- Identify the current needs for solutions of the developer.
- Identify the anticipated needs for solutions of the developer.
- Identify the current needs for solutions of the customer.
- Identify the anticipated needs for solutions of the customer.
- Use the identified needs as a reference to develop or acquire reusable assets to meet the specified needs.

Implementation model: Establishing goals and making strategies to achieve those goals are complex tasks. The goals are divided into four stages, namely, opportunistic, integrated, leveraged, and anticipating. In this section, brief descriptions of the four stages will be provided by focusing on the key characteristics of the stages. Specifically, each characteristic relates to a goal in the implementation model. More details of the implementation model can be found in the reuse adoption guidebook [44].

- **Opportunistic**: A common reuse strategy does not fit all projects so each project develops its own strategy to reuse artifacts. The strategy includes: (i) defining reuse activities in the project plan; (ii) using tools to support the reuse activities; (iii) identifying the needs of the developers and developing or acquiring reusable

artifacts; and (iv) identifying reusable artifacts throughout the life cycle of the project.

- **Integrated**: The organization defines a reuse process and integrates it with its development process. It is important for the organization to support the reuse process by means of policies, procedures, resource allocation, and organizational structure.
- **Leveraged**: To extract the maximum benefits from reuse in groups of related products, a strategy for reuse in product lines is developed.
- **Anticipating**: Reusable assets are acquired or developed based on anticipated customer needs.

9.4.3 RiSE Maturity Model

The RiSE maturity model [42] was developed during the RiSE project through discussions with industry partners [45]. The model structure intends to be flexible, modular, and adaptable to the needs of the organization that will use them. Thus, the model was based on two principles: modularity and responsibility. The term modularity is used in the sense of processes with less coupling and maximum cohesion, and the term responsibility is used in the sense of the possibility to establish one or more team responsible for each process, or activity. The RiSE maturity model includes: (i) reuse practices grouped by perspectives and in organized levels representing different degrees of software reuse achieved; and (ii) reuse elements describing fundamental parts of reuse technology, such as assets, documentation, tools, and environments. Reuse elements and reuse assets are synonymous.

Maturity levels, as listed below, indicate the: (i) reuse elements that are expected to be used in the organization; and (ii) the reuse practices that are expected to be implemented and followed in the organization. The activities at level n and higher are based on the reuse elements implemented and practices followed at level $n - 1$, such that $1 < n \leq 5$. The five maturity levels are as follows:

1. **Level 1:** Ad hoc Reuse
2. **Level 2:** Basic Reuse
3. **Level 3:** Initial Reuse
4. **Level 4:** Organized Reuse
5. **Level 5:** Systematic Reuse

Associated with each level of maturity are a list of reuse practices and a list of reuse elements. Each maturity level is characterized by its own goals to be achieved. The maturity levels are discussed next.

Level 1: Ad hoc Reuse At Level 1: (i) software is developed in a traditional way, without reusing assets; (ii) management does not support reuse; and (iii) reuse practices might be performed as an individual initiative by developers. By default, all organizations making no reuse efforts fall in Level 1.

Level 2: Basic Reuse At Level 2, simple tools are used to develop reusable technical assets, namely, documents, software design, and code. Those technical assets consider all the requirements of the system without making any distinction between domain-specific aspects and business aspects. Developers manually modify the reusable assets to complete an implementation. Organizations benefit by acquiring experience in system design and software reuse. The following goals are defined for Level 2.

- Goal 1: Best practices in reuse are adopted in the design and implementation of the software product.
- Goal 2: Use the technical assets, namely, code and documentation, to build software.

Level 3: Initial Reuse At Level 3, organizations make a distinction between business assets and domain-related assets, because it is useful to maintain the business issues independent from the implementation issues. The advantage of the aforementioned decoupling is that the same implementation asset can be reused in different projects having slightly different business requirements. Separating business assets from domain-related assets is important for system families. Next, reuse practices are institutionalized. Moreover, to automate the engineering process, efforts are made. An asset manager, which is basically a reuse repository, stores the engineering process knowledge [41]. To evaluate the effectiveness of reuse activities and practices, metrics are defined. For example, a count of the times a certain module is reused is a measure of the reuse of that module. At Level 3, the following goals are defined:

- Goal 1: Separate business-specific aspects from domain-related aspects.
- Goal 2: Use defined process.

Level 4: Organized Reuse Level 4 is characterized by a better integration of all reuse abstraction levels. Reuse is introduced at the highest abstraction level. Staff members know the reuse vocabulary and have reuse expertise. Reuse occurs across all functional areas. At Level 4, domain engineering is performed. Reuse-based processes are in place to support and encourage reuse, and the organization has a focus on developing families of products. The organization has all the data needed to decide which assets to build or what to acquire, because it has a reuse inventory organized along application-specific lines. The reuse practices within the projects start to be quantitatively managed, and quality metrics are collected and analyzed. All costs associated with assets development and all savings from its reuse are reported and shared. Projects are statistically controlled through standardized metrics that lead to a better control of the objectives of the project. For Level 4, the goals and practices are as follows:

- Goal 1: Enhance the organization's competitive advantage.
- Goal 2: Integrate reuse activities in the whole software development process.
- Goal 3: Ensure efficient reuse performance.

Level 5: Systematic Reuse At Level 5, the organization's knowledge base is planned, organized, stored, and maintained in a reuse inventory (a.k.a. asset manager) and is used with focus on the software development process. By the time an organization reaches this level, all major obstacles to reuse have been removed. All definitions, guidelines, and standards are in place enterprise wide. Domain engineering practices are put in place. Knowledge and reusable assets are regularly validated to make strategic assets reusable. For future reuse, all software products are generalized. Domain analysis is performed across all product lines. All system utilities, tools, and accounting mechanisms are instrumented to track reuse. From the reuse inventory perspective, the development process supports a planned activity to acquire or develop missing pieces in the catalog. From the technological perspective, the organization has automated reuse support integrated with their development process. From the business perspective, the organization benefits from having implemented a software reuse approach. The organization expresses its system development capability in the form of reusable assets. The capability to reuse assets accelerates the time to market. All costs are associated to a product line or a particular asset and all savings from its reuse are reported and shared. At Level 5, the goals and practices are as follows.

- Goal 1: Reuse is "the way we do business."
- Goal 2: Establish and maintain complete reuse-centric development.

Perspective and Factors In the RiSE maturity model, 15 factors were considered, and those are divided into 4 perspectives: organizational, business, technological, and processes. The organizational perspective concerns activities that are closely related to management decisions required to organize and manage a reuse project. Table 9.3 shows the factors related to the organizational perspective and their distribution across the RiSE maturity model levels. The business perspective addresses issues related to the business domain and market decisions for the organization. Table 9.4 shows the factors related to the business perspective. The technological perspective covers development activities in the software reuse engineering discipline and factors related to the infrastructure and technological environment. Table 9.5 shows the factors related to the technological perspective. Finally, the processes perspective includes only those activities which support the implementation of the engineering and the project management practices. Table 9.6 shows the factors related to the processes perspective.

9.5 ECONOMIC MODELS OF SOFTWARE REUSE

Project managers and financial managers can use the general economics model of software reuse in their planning for investments in software reuse. The project manager needs to estimate the costs and potential payoffs to justify systematic reuse. Increased productivity is an example of payoff of reuse. In the following section, we discuss cost model of Gaffney and Durek [46], application system cost model of Gaffney and Cruickshank [47], and business model of Poulin and Caruso [48].

TABLE 9.3 RiSE Maturity Model Levels: Organizational Factors [42]

	Levels				
Factors of Influence	1. Ad hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Planning for reuse	– Non-existent	– Grassroot activity – Reuse is viewed as single-point opportunities – Individual achievements are rewarded	– Targets of opportunity – Organization responsible for reuse – A key business strategy	– Business imperative – Reuse occurs across all functional areas	– Part of a strategic plan – Discriminator in business success
Software reuse education	– Lack of expertise by staff members – Frequent resistance to reuse	– Basic definitions of reuse are agreed upon	– The staff has the expertise and know-how to obtain benefits with reuse	– The staff members know the reuse vocabulary and have reuse expertise	– All definitions, guidelines, and standards are in place, enterprise wide
Legal, contractual, accounting considerations	– Inhibition to getting started	– Internal accounting scheme for sharing costs allocating benefits	– Data rights and compensation issues resolved with customer	– Royalty scheme for all suppliers and customers	– Software treated as key capital asset

(*continued*)

TABLE 9.3 (*continued*)

	Levels				
Factors of Influence	1. Ad hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Funding, costs, and financial features	– Costs of reuse are unknown	– Costs of reuse are "feared"	– Payoff of reuse is "known" and understand for a given domain – Investments made in reuse, payoffs expected – Costs of reuse are "known"	– All costs associated with an asset's development and all savings from its reuse are reported and shared	– All costs associated to a product line or a particular asset and all saving from its reuse are reported and shared
Rewards and incentives	– Reuse is discouraged by management	– Reuse is encouraged	– Reuse is motivated reinforced, rewarded	– Reuse is indoctrinated	– Reuse is "the way we do business"
Independent reusable assess development team	– Individual initiative (personal goal as time allows)	– Shared initiative	– Dedicated individual	– Dedicated group	– Corporate group (for visibility not control) with division liaisons

TABLE 9.4 RiSE Maturity Model Levels: Business Factors [42]

	Levels				
Factors of Influence	1. Ad hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Product family approach	– Isolated products – No family product approach	– Common features and requirements across the products – Commonalities and reuse possibilities were identified	– Product line domain analyses performed	– Focus on developing families of products – Domain engineering performed	– Domain analysis performed across all product lines – Product family approach
Software reuse education	– Chaotic development process unclear where reuse comes in	– Reuse questions raised at design reviews (after the fact) – Development process defined (some reuse activity indications)	– Design emphasis placed on reuse of off-the-shelf parts – Product line domain analyses performed – Shared understanding of all the activities needed to support reuse	– Focus on developing families of products – Reuse-based processes are in place to support and encourage reuse – Domain engineering performed	– All software products generated for future reuse – Domain analyses performed across all product lines – Product family approach

TABLE 9.5 RiSE Maturity Model Levels: Technological Factors [42]

	Levels				
Factors of Influence	1. Ad hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Repository systems usage	– Salvage yard (No apparent structure to collection)	– Catalog identifies language-and platform-specific parts – Simple structure like concurrent versions systems – Considered mainly source code	– Catalog includes generic data processing functions – Considered software components, reports and document models	– Catalog organized along application specifications – Have all data needed decide which assets to build/acquire – Considered screen generators database elements and test cases	– Planned activity to acquire or develop missing pieces in catalog – Considered all artifacts of software development life cycle
Technology support	– Personal tools, if any	– A collection of tools, e.g., CM, but not specialized to reuse – General-purpose analyzers combined to assess reuse levels	– Classification aids and synthesis aids – Standardization on components and architecture – Tools customized to support reuse	– Digital library separate from development environment	– Automated support integrated with development system – Fully integrated with development and reporting systems

TABLE 9.6 RiSE Maturity Model Levels: Processes Factors [42]

	Levels				
Factors of Influence	1. Ad hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Quality models usage	– No quality model adoption	– Some quality activities were incorporated in the software development process	– Software development process guided by a quality model	– High quality model usage in the engineering department	– Quality model completely adopted in the organization activities
Software reuse measurement	– No metrics on level of reuse, payoff, or cost of reuse	– Number of lines of reused code factored into cost models	– Manual tracking of reuse occurrences of catalog parts	– Analyses performed to identify expected payoffs from developing reusable parts	– All system utilities, software tools, and accounting mechanisms instrumented to track reuse
Systematic reuse process	– No reused-based process	– Some reuse activities were adopted in the development process – Planning to adapt the software development process of the organization for a reuse-based process	– Development process of the organization is adapted to reuse concepts	– Reuse benefits and concepts are clear for the engineering team – Development process is reused based	– Systematic reuse process is enterprise wide
Origin of the reused assets	– No reuse assets	– Build from scratch, sometimes indirectly	– Build from existent products; adapting existing products	– Build from existing products; extracted through a reengineering process	– Planning the design and building of reusable assets according to product family
Previous development of reusable assets	– No development of reusable assets	– Parallel with development	– Before development	– Before development	– Before development

9.5.1 Cost Model of Gaffney and Durek

The two cost and productivity models proposed by Gaffney and Durek [46] for software reuse are: *first order reuse cost model* and *higher order cost model*. The cost of reusing software components has been modeled in the *first order reuse cost model*, whereas the *higher order cost model* considers the cost of developing reusable assets.

First Order Reuse Cost Model In this model, we assume the following conditions [49].

1. The reused software satisfies the black-box requirements in the sense that it is stable and reliable. If the behavior of the component becomes unreliable, unstable, or inconsiderably different than the project's needs, the component becomes a white box because it needs to be fixed.
2. Users of the reusable components have adequate expertise in the context of reuse.
3. There is adequate documentation of the components to be reused.
4. The cost of reusing the components is negligible.

As explained in what follows, three broad categories of program code are used in a project.

- S_n*:* It represents the new code added to the system.
- S_o*:* It represents the original source code from the preexisting system. S_o includes both *lifted* code and modified code. *Lifted* code means unchanged, original code taken from past releases of a product. The source code from modified (partial) parts are not considered as reused code.
- S_r*:* It represents the reuse source code that are not developed or maintained by the organization. The reuse code is obtained from completely unmodified components normally located in a reuse library.

The *effective size*, denoted by S_e, is an adjusted combination of the modified source code and the new source code, as given in the following equation.

$$S_e = S_n + S_o(A_d \times F_d + A_i \times F_i + A_t \times F_t)$$

where:

A_d = is a normalized measure of design activity,
A_i = is a normalized measure of integration activity,
A_t = is a normalized measure of testing activity, and
$A_d + A_i + A_t = 1$.

The expression within the parentheses, namely, $(A_d \times F_d + A_i \times F_i + A_t \times F_t)$, is a weighted sum of relative efforts from design, implementation, and test; F_d, F_i, and F_t represent relative efforts from design, implementation, and test, respectively.

One or more reusable components comprise the rest of the system. Reusable components are assumed to be black boxes so their sizes are assumed to be not available. Therefore, the relative sizes of reusable components can be obtained by estimating the size of the reusable components with the assumption that those are built from the scratch. Letting S_r denote the estimated size of reusable components, the relative sizes of reusable components is given by R, where R is expressed as follows:

$$R = \frac{S_r}{S_e + S_r}.$$

Let C be the cost of software development for a given product relative to that for all new code (for which $C = 1$). Let R be the proportion of reused code in the product as defined earlier ($R \leq 1$). Let b be the cost, relative to that for all new code, of incorporating the reused code into the new product. Note that $b = 1$ for all new code. The relative cost for software development is:

[(relative cost of all new code) $*$ (proportion of new code)]
+
[(relative cost of reused software) $*$ (proportion of reused software)].

Therefore:

$$C = (1)(1 - R) + (b)(R) = (b - 1)R + 1$$

and the associated relative productivity is:

$$P = \frac{1}{C} = \frac{1}{(b - 1)R + 1}.$$

b must be <1 for reuse to be cost-effective. The size of b depends on the life cycle phase—requirements, design, implementation, or testing—of the reusable components. As an example, let us consider the relative cost of the development activities as shown in Table 9.7. Table 9.8 shows example relative activity costs to calculate the relative component reuse cost b. For instance, if we want to reuse a requirements component, then the accompanying costs are in the form of design, implementation, and test activities, which are 0.37, 0.22, and 0.33, respectively. Therefore, the relative cost to reuse requirements is $b = (0.37 + 0.22 + 0.33) = 0.92$. On the other hand, if code is reused, then the additional tasks will involve requirements and testing. Therefore, the relative cost to reuse code is $b = (0.08 + 0.33) = 0.41$.

TABLE 9.7 Relative Costs of Development Activities

Activity	Activity Code	Activity Cost
Requirements	Req	0.08
Design	Des	0.37
Implementation	Imp	0.22
Test	Test	0.33

TABLE 9.8 Relative Reuse Cost (*b*)

Component Type	Activities to be Completed	Relative Reuse Cost (*b*)
Requirements	Des, Imp, Test	0.92
Design	Req, Imp, Test	0.63
Code	Req, Test	0.41
Requirements, Design, Code,	Test	0.33

Higher Order Reuse Cost Model Estimating the cost of developing reusable components is key to formulating a reuse cost model. By combining the development cost of the reusable components into the economic model, we have

$$C = (1 - R) \times 1 + \left(b + \frac{a}{n}\right) R,$$

where a is the cost of developing reusable components relative to the cost of building new non-reusable components from the scratch, and n is the number of uses over which the cost of reusable components is amortized. Now, the model can be rewritten as

$$C = \left(b + \frac{a}{n} - 1\right) R + 1.$$

9.5.2 Application System Cost Model of Gaffney and Cruickshank

An application system cost model based on domain engineering and application engineering was proposed by Gaffney and Cruickshank. The cost of an application system is expressed as the sum of two component costs: (i) the investment in domain engineering apportioned over N application systems; and (ii) the cost of application engineering to develop a specific system. Therefore, the cost of an application system, C_s, is equal to the prorated cost of domain engineering plus the cost of application engineering. In addition, let the cost of application engineering be the cost of the new code plus the cost of the reused code in the new application system, and let R denote the fraction of code that is reused code.

Now, we have

$$C_s = C_{dp} + C_a$$
$$C_s = C_d/N + C_n + C_r,$$

where:

$C_{dp} = C_d/N$ and $C_a = C_n + C_r$;
C_s = the total cost of the application system;
C_d = the total cost of domain engineering;

C_{dp} = the prorated portion of C_d shared by each of the N application systems;
C_a = the cost of an application system;
C_n = the cost of the new code in the application system;
C_r = the cost of the reused code in the application system.

Each of the costs, C_d, C_n, and C_r, is taken to be the product of a unit cost (LM/KSLOC) and an amount of code (KSLOC), where LM/KSLOC stands for labor months/1000 source lines of code (LOC). Hence,

$$C_d = C_{de} * S_t, C_n = C_{vn} * S_n,$$

and

$$C_r = C_{vr} * S_r.$$

The equation for reuse cost is

$$C_s = C_{us}S_s = \frac{C_{de}S_t}{N} + C_{vn}S_n + C_{vr}S_r,$$

where:

C_{us} = unit cost of the application system;
C_{de} = unit cost of domain engineering;
C_{vn} = unit cost of new code developed for this application system;
C_{vr} = unit cost of reusing code in this application system;
S_t = expected value of the unduplicated size of the reuse library, measured in source statements;
S_n = amount of new code in terms of source statements developed for this application system;
S_r = amount of reused code incorporated into this application system in source statement;
S_s = total size of the application system in source statement.

Let $S_n/S_s = l - R$ and $S_r/S_s = R$, where R is the proportion of reuse. The reuse cost equation can be rewritten as

$$C_{us} = \frac{C_{de}}{N}\frac{S_t}{S_s} + C_{vn}(1 - R) + C_{vr}R.$$

Now let $S_t/S_s = K$, the library relative capacity. Thus, the basic reuse cost equation is

$$C_{us} = \frac{C_{de}}{N}K + C_{vn} - (C_{vn} - C_{vr})R.$$

The basic reuse cost equation assumes a single reuse of S_r units (SLOC, KSLOC) in each of the "N application" systems. Thus, this expression is applicable to systematic reuse of units of code relatively dense in functionality.

9.5.3 Business Model of Poulin and Caruso

Poulin and Caruso [48] developed a model at IBM to improve measurement and reporting software reuse. The authors present a measure for reuse level, the financial and the productivity benefit of reuse. They consider potential benefits of reuse against the cost of resources expended to identify and integrate reusable assets into a system. Their results are based on a set of data points as follows:

- *Shipped source instruction* (SSI): SSI is the total count of executable code lines in the source files of a product. Calling a reused part is counted as one SSI.
- *Changed source instruction* (CSI): CSI is the total count of executable code lines that are new, added, or modified in a new release of a product. Note that CSI does not include RSI and unchanged instructions from past releases of the system. Calling a reused part is counted as one CSI.
- *Reused source instruction* (RSI): RSI is the total source instructions shipped but not developed or maintained by the reporting organization. RSI does not include base instructions from past releases of a product. In addition, RSI does not include source instructions from partly changed components. Independent of the number of times the component is called or expanded, source instructions from a reused component are counted just once.
- *Source instruction reused by others* (SIRBO): SIRBO is the total lines of source instructions of an organization reused by others. SIRBO is a measure of the contributions of an organization to reuse, and it is calculated as follows:

$$\begin{aligned}\text{SIRBO} = &(\text{Source instructions per part}) \\ &\times (\text{The number of organizations using the part}).\end{aligned}$$

- *Software development cost* (Cost per LOC): This metric concerns the development of new software, and it is calculated in two steps. (i) Let S denote the total cost of the organization, including overhead; and (ii) divide S by the total outputs of the organization in number of LOC. It may be noted that one needs to know the cost of developing the software without reuse so as to understand the financial benefit of reuse.
- *Software development error rate* (Error rate): It is a historical average number of errors uncovered in the products. To estimate the cost of avoiding maintenance, a historical average value is used.
- *Cost per error:* To quantify the advantage of better quality reusable assets, the historical mean cost of maintaining components with traditional development methods is used as a base line. Now, the cost per error metric is calculated in two steps: (i) let S denote the sum of *all* costs; and (ii) divide S by the number of errors repaired.

The aforementioned metrics are combined to form three derived metrics: reuse percent, reuse cost avoidance, and reuse value added. The reuse percent and reuse cost avoidance metrics give indications of an organization's reuse activities. The reuse value added metric includes recognition for writing reusable code.

1. *Reuse Percent* (RP): It reflects the extent of reuse in a product. *RP* is analogous to *R* in the model of Gaffney and Durek. Poulin and Caruso make a distinction between the reuse percent of a product release and the reuse percent of a product:

$$\textit{Reuse percent of a product} = \frac{RSI}{RSI + SSI} \times 100\%.$$

$$\textit{Reuse percent of a product release} = \frac{RSI}{RSI + CSI} \times 100\%.$$

2. *Reuse cost avoidance* (RCA): The purpose of this metric is to measure reduced total product costs as a result of reuse. One must retrieve and evaluate the reusable assets to choose the appropriate ones to be integrated into the system being developed. According to studies performed by Poulin et al. [50], there are significant financial benefits due to reusing assets. For example, the cost of integrating a reusable software element is 20% of the cost of developing the same element anew. The financial benefit due to adopting reuse in the development phase of a project is calculated as follows:

$$\textit{Development cost avoidance} = RSI \times (1 - 0.2) \times (\textit{new code cost}).$$

In addition, saving in maintenance cost attributed to reuse is much more than those during software development, because of the fewer defects in reused components [51]. The saving is

$$\textit{Service cost avoidance} = RSI \times (\textit{error rate}) \times (\textit{cost per error}).$$

The total reuse cost avoidance is calculated as the sum of cost avoidance in the development and maintenance activities, which is

$$\begin{aligned}\textit{Reuse cost avoidance} = {} & \textit{Development cost avoidance} \\ & + \textit{Service cost avoidance}.\end{aligned}$$

3. *Reuse value added* (RVA): The main idea behind RVA is to provide a metric to reward an organization that reuses software components and help other organizations by developing reusable components. RVA is derived from SSI, RSI, and SIRBO:

$$\textit{Reuse value added} = \frac{(SSI + RSI) + SIRBO}{SSI}.$$

Organizations with no involvement in reuse have an RVA = 1. An RVA = 2 indicates that the organization is twice as effective as it would be without reuse. In this case the organization was able to double its productivity either directly by reusing it or indirectly by maintaining software that other organizations are using.

9.6 SUMMARY

We explained the concepts of reuse including program families and domain analysis. In addition, we discussed four major factors that influence software reuse: managerial, legal, economic, and technical.

Next we examined two concepts: domain engineering and application engineering. Domain engineering is also known as product line development, while application engineering being known as product development. Domain engineering is a set of activities that produce RSA to be used in several software projects. It includes analysis, design, and implementation activities. On the other hand, application engineering refers to a "development-with-reuse" process to create specific systems. Then we introduced nine domain engineering approaches reported in the literature: Darco, DARE, FAST, FORM, KobrA, PLUS, PuLSE, Koala, and RSEB.

Next, we discussed three maturity models: RMM, RCM, and RiSE maturity model. We concluded the chapter with a discussion of economic model of software reuse to justify. Finally, we discussed three models: the cost model of Gaffney and Durek, the application system cost model of Gaffney and Cruickshank, and the business model of Poulin and Caruso.

LITERATURE REVIEW

Numerous books and articles have been written about reuse. There are two good reuse research articles by Frakes and Kang [13] and Mili et al. [52]. A two-part book about reuse has been authored by Stanislaw Jarzabek (*Effective Software Maintenance and Evolution: A Reuse-Based Approach*, Auerbach Publications, Boca Raton, FL, 2007). The first part focuses on traditional methods for software maintenance. The second part describes a novel approach called "mixed strategy." In mixed-strategy program representation, code contains information about program design and modifications occurring over the lifetime of the system. A good discussion of integration reuse with an object-oriented development process can be found in "*Software Reuse: Architecture, Process and Organization for Business Success*" (I. Jacobson, M. Griss and P. Jonsson, Addison-Wesley, 1997).

A thorough treatment of a framework for practicing reuse within the software life cycle can be found in IEEE Std 1517-1999, "IEEE Standard for Information Technology – Software Life Cycle Processes – Reuse Processes." The standard provides a framework to extend the software life cycle processes to include software reuse. Moreover, the reuse adoption process [44], defined by the Software Productivity Consortium (SPC), provides a solution to implement a reuse program, and it is based on the implementation of the model proposed by Prieto-Diaz [14]. RCM [40] is used together with the reuse adoption process defined by SPC.

The article by Parastoo Mohagheghi and Reidar Conradi, "An empirical investigation of software reuse benefits in a large telecom roduct." *ACM Transactions on Software Engineering and Methodology*, 17(3), 2008, pp. 13:1–31, based on an empirical study, describes the benefits of software reuse in a large telecommunication

product. The key observation was that lower fault density and less modified code occurred between successive releases of the reused components.

Software product lines enjoy increasingly wide adoption in the software industry. The Software Engineering Institute defines basic concepts, activities, and practices that ensure success. In the article "SEI's software product line tenets," *IEEE Software*, July/August, 2002, pp. 32–40, the author Linda M. Northrop described the success stories and lessons learned while defining and applying this approach. Readers who are interested in relationships between software evolution and software product line process are recommended to study the following two articles.

Samuel A. Ajila and Ali B. Koba, Evolution support mechanisms for software product lines process. *Journal of Systems and Software*, 81(10), 2008, pp.1784–1801.

Stephen R. Schach and Amir Tomer, Development/maintenance/reuse: Software evolution in product lines. *Proceedings of the first Conference on Software Product Lines: Experience and Research Directions*, Denver, Colorado, Kluwer Academic, Norwell, MA, 2000, pp. 437–450.

The first article describes mechanisms to support evolution of software product lines. In the second article, the authors extend the *evolution tree model* and *propagation graph model* to describe the evolution of a software product line.

Readers interested in product-line development may read the article by Capretz, Ahmed, Al-Maati, and Al Aghbari (COTS-based software product line development. International Journal of Web Information Systems, Emerald Group Publishing, 4(2), 2008, pp. 165–180). The authors discussed COTS-based software product line development methodology based on Y-model [53]. The methodology integrates the concept of software product line with COTS.

REFERENCES

[1] J. Backus. 1998. The history of fortran i, ii, iii. *IEEE Annals of the History of Computing*, 20(4), 68–78.

[2] M. D. McIlroy. 1969. *Mass Produced Software Components*. Proceedings of Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering (Eds P. Naur, B. Randell, J. N.), pp. 138–155, Available through Petrocelli-Charter, New York.

[3] D. L. Parnas. 1976. On the design and development of program families. *IEEE Transactions of Software Engineering*, 2(1), 1–9.

[4] J. M. Neighbor. 1980. Software construction using components. Technical Report 160, Department of Information and Computer Sciences, University of California, Irvine, CA.

[5] J. M. Neighbors. 1984. The draco approach to constructing from software reusable components. *IEEE Transactions of Software Engineering*, 10(5), 564–574.

[6] R. Prieto-Diaz. 1990. A domain analysis: An introduction. *ACM Software Engineering Notes*, 15(2), 47–54.

[7] T. C. Jones. 1984. Reusability in programming: A survey of the state of the art. *IEEE Transactions of Software Engineering*, 10(5), 488–494.

[8] P. Grubb and A. Takang. 2003. *Software Maintenance Concepts and Practice*, 2nd edition. World Scientific, Singapore.

[9] I Sommervilles. 2001. *Software Engineering*, 6th edition. Pearson Education Limited, Harlow, England.

[10] R. Prieto-Diaz. 1991. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5), 88–97.

[11] A. Mili, R. Mili, and R. T. Mittermeir. 1998. A survey of software reuse libraries. *Annals of Software Engineering*, 5, 349–414.

[12] B. P. Lientz and E. B. Swanson. 1980. *Software Maintenance Management*. Addison-Wesley, Reading, MA.

[13] W. Frakes and Kyo Kang. 2005. Software reuse research: Status and future. *IEEE Transactions of Software Engineering*, 31(7), 529–536.

[14] R. Prieto-Diaz. 1991. Making software reuse work: An implementation model. *ACM Software Engineering Notes*, 16(3), 61–68.

[15] C. Krueger. 2002. Eliminating the adoption barrier. *IEEE Software*, 19(4), 29–31.

[16] W. Frakes and P. Gandel. 1990. Representing reusable software. *Information and Software Technology*, 32(10), Butterworth-Heinemann Ltd, 1990, pp. 653–664.

[17] W. Frakes and S. Isoda. 1994. Success factors of systematic reuse. *IEEE Software*, 11(5), 14–19.

[18] M. L. Griss. 1993. Software reuse: From library to factory. *IBM Systems Journal*, 32(4), 548–564.

[19] J. Favaro. 1991. What price reusability? a case study. *Ada Letter*, 11(3), 115–124.

[20] D. C. Rine. 1993. Supporting reuse with object technology. *IEEE Software*, 30(10), 43–45.

[21] J. L. Diaz-Herrera. 2001. Domain engineering. In: *Handbook of Software Engineering and Knowledge Engineering* (Ed. S. K. Change), Vol. 1, pp. 305–328. World Scientific Publishing Co., Singapore.

[22] S. Beydeda and V. Gruhn (Eds.). 2005. *Testing Commerical-off-the-Shelf Components and Systems*. Springer, Berlin, Germany.

[23] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and S. A. Peterson. 1990. Feature-oriented domain analysis (foda) feasibility study. CMU/SEI-90-TR-21, ADA 235785, Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University.

[24] B. Schmerl and D. Garlan. 2004. *Acmestudio: Supporting Style-centered Architecture Development (Research Demonstration)*. Proceedings of the 26th International Conference on Software Engineering, May, 2004, IEEE Computer Society Press, Los Alamitos, CA, pp. 23–28.

[25] J. Nester, J. M. Newcomer, P. Giannini, and D. Stone. 1990. *IDL: The Language and its Implementation*. Prentice-Hall, Upper Saddle River, NJ.

[26] L. M. Northrop. 2002. SEI's software product line tenets. *IEEE Software*, 19(4), 32–40.

[27] R. H. Thayer. 2002. Software system engineering: A tutorial. *IEEE Computer*, 2002, pp. 68–73.

[28] R. R. Macala, L. D. Stuckey, and D. C. Gross. 1996. Managing domain-specific product-line development. *IEEE Software*, 13(3), 57–67.

[29] E. S. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, and S. R. L. Meira. 2005. A survey on software reuse processes. IEEE International Conference on Information Reuse and Integration (IRI), August, 2005, Las Vegas, Nevada, IEEE Computer Society Press, Los Alamitos, CA, pp. 66–71.

[30] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laintenberger, R. Laqua, D. Muthiq, B. Paech, J. Wust, and J. Zettel. 2002. *Component-based Product Line Engineering with UML*. Addison-Wesley, Reading, MA.

[31] W. Frakes, R. Prieto-Diaz, and C. Fox. 1998. Dare: Domain analysis and reuse environment. *Annals of Software Engineering*, 5, 125–141.

[32] O. Alonso. 2003. Generating text search application for databases. *IEEE Software*, 20(3), 98–105.

[33] D. M. Wiess and C. T. R. Lai. 1999. *Software-line Engineering: A Family-based Software Development Process*. Addison-Wesley, Reading, MA.

[34] K. C. Kang, J. Lee, and P. Donohoe. 2002. Feature-oriented product line engineering. *IEEE Software*, 19(4), 58–65.

[35] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. 1998. Form: A feature-oriented reuse method and domain-specific reference architecture. *Annals of Software Engineering*, 5, 143–168.

[36] H. Gomaa. 2004. *Designing Software Product Lines with UML: From Use Case to Pattern-Based Software Architectures*. Addison-Wesley, Reading, MA.

[37] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J-M. Debaud. 1999. *Pulse: A Methodology to Develop Software Product Lines*. Proceedings of the 1999 Symposium on Software Reusability, May, 1999, Los Angeles, CA, ACM Press, New York, pp. 122–131.

[38] R Ommering, F. Linden, J. Kramer, and J. Magee. 2000. The koala component model for consumer electronics software. *IEEE Software*, 78–85. doi: 10.1109/2.825699

[39] I. Jacobson, M. L. Gries, and P. Jonsson, 1997. *Reuse-driven Software Engineering Business (RSEB)*. Addison-Wesley, Reading, MA.

[40] T. Davis. 1993. *The Reuse Capability Model: A Basis for Improving an Organization's Reuse Capability*. Proceedings of 2nd IEEEACM International Workshop on Software Reusability, 1993, Herndon, VA, IEEE Computer Society Press/ACM Press, Los Alamitos, CA, pp. 126–133.

[41] P. Koltun and A. Hudson. 1991. A reuse maturity model. In 4th Annual Workshop on Software Reuse, Herndon, VA, IEEE Computer Society Press/ACM Press, Los Alamitos, CA.

[42] V. C. Garcia, D. Lucrédio, A. Alvaro, E. S. Almeida, R. P. M. Fortes, and S. R. L. Meira. 2007. Towards maturity model for a reuse incremental adoption. Brazilian Symposium on Software Component, Architectures and Reuse (SBCARS), August, 2007, Campinas, São Paulo, Brazil, pp. 61–73.

[43] W. Frakes and C. Terry. 1996. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2), 415–435.

[44] SPC. 1993. Reuse adoption guidebook. Version 02.00.05., Technical Report SPC-92051-CMC, Software Productivity Consortium, 279.

[45] E. S. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, and S. R. L. Meira. 2004. *Rise Project: Towards a Robust Framework for Software Reuse*. IEEE International Conference on Information Reuse and Integration (IRI), August, 2004, Las Vegas, Nevada, IEEE Computer Society Press, Los Alamitos, CA, pp. 48–53.

[46] J. E. Gaffney and T. A. Durek. 1989. Software reuse - key to enhanced productivity: Some quantitative models. *Information and Software Technology*, 31(5), 258–267.

[47] Jr. J. E. Gaffney and R. D. Cruickshank. 1992. *A General Economics Model of Software Reuse*. Proceedings of the 14th International Conference on Software Engineering (ICSE), May, 1992, Melbourne, Australia, ACM Press, New York, pp. 327–337.

[48] J. S. Poulin and J. M. Caruso. 1993. *A Reuse Measurement and Return on Investment Model*. Second International Workshop on Software Reusability, March, 1993, Lucca, Italy, IEEE Computer Society Press, Los Alamitos, CA, pp. 152–166.

[49] R. W. Jensen. 2004. An economic analysis of software reuse. *CrossTalk A Journal of Defense Software Engineering*, 4–8.

[50] J. S. Paulin, J. M. Caruso, and D. R. Hancock. 1993. The business case for software reuse. *IBM Systems Journal*, 32(4), 567–594.

[51] S. R. Schach. 1994. The economic impact of software reuse on maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 6(4), 185–196.

[52] H. Mili, F. Mili, and A. Mili. 1995. Reusing software: Issues and research directions. *IEEE Transactions of Software Engineering*, 21(6), 528–562.

[53] L. F. Capretz. 2005. Y: A new component-based software life cycle model. *Journal of Computer Science*, 1(1), 76–82.

EXERCISES

1. Explain the terms reuse and reusability with respect to software development and maintenance.

2. Explain why it is important to reuse software instead of writing it from the scratch.

3. Explain the difference between reverse engineering and reuse engineering. What is software reclamation with regard to reuse engineering?

4. Compare and contrast the different approaches to reuse: proactive, reactive, and extractive.

5. Explain the differences between: (i) vertical and horizontal reuse; and (ii) generative and compositional reuse.

6. Explain the concepts of "development-for-reuse" and "development-with-reuse."

7. Briefly explain the five maturity levels in the RiSE maturity model.

8. Briefly explain the two components (assessment and implementation) of reuse capability model.

9. A programming team develops and maintains 80K SSI and the team additionally uses 20K RSI from a reuse library. Calculate the reuse percentage for the team.

10. An organization has a historical new code development cost of $200 per line, an error rate of 1.5/KLOC, and cost of $43K to fix an error. Calculate the value of RCA for integrating 20K RSI into the product.

11. A programming team maintains 80KLOC and uses 20KLOC from a reuse library. In addition, five other departments reuse a 10KLOC module the programming team contributed to the organizational reuse library. What is the value of RVA of the programming team?

12. Suppose that you joined a new company that has no reuse program. The first task for you is to implement the reuse program.

(a) What is the first step that you would take?

(b) Outline the technical, managerial, economical, organizational and legal steps that you would go through.

(c) What difficulties do you anticipate and how would you mitigate them?