Software Engineering

Lecture 12

Software Reuse

Dr. Obuhuma James

Description

This is the last topic in the series of 12 topics on software engineering. The topic covers software reuse as an approach that facilitates development of better software, more quickly, and at a lower cost. The various types or categories of reuse-based engineering are covered. The concepts of COTS product and ERP system reuse are also covered as examples of approaches that support component reuse. Finally, Component-Based Software Engineering is discussed.

Learning Outcomes

By the end of this topic, you will be able to:

- Discuss the concept of software reuse as applied in Software Engineering.
- Differentiate among the three types of software reuse.
- Apply software reuse in real life.

Overview of Software Reuse

Software reuse entails the development of software in a manner that utilizes existing software components. Such components include source code, designs and interfaces, software requirement specifications, user manuals and system documentations, among others. This in return leads to reduction in development effort, time, and cost, utilization of fewer resources, and probably development of better quality software products.

Software Reuse

In most engineering disciplines, systems are designed by putting together different existing components borrowed from other systems. Software engineering has for a long time been more inclined towards originality in development [1]. However, for close to a decade now, there has been a shift towards trying to embrace reuse-based development in software engineering. The major driving factors include the high likelihood of achieving better software, more quickly and at lower costs.

Software reuse can be embraced at different levels of software development depending on the desired benefit. Thus, the various broad types or categorisations of reuse-based engineering as will be seen in the subsequent sections.

Types of Software Reuse

Reuse-based software engineering may be accomplished using either application system reuse, component reuse, or object and function reuse. Each of the three varies depending on the extent of the reuse being employed.

1. Application system reuse entails a reuse of the whole application system either by incorporating it without making any changes in the system or by developing application families. In case of the former approach, then this is like the case of COTS reuse.
2. Component reuse entails reuse of independent components of an application from sub-systems to single objects.
3. Object and function reuse focuses on slightly smaller units of a system, which are the objects and functions. This type of reuse thus implements a reuse of a single well-defined object or function.

It is worth noting that an object represents a real-world thing. In this case, objects and functions represent a block of code bundled together to perform a specific task. On the other hand, a component is made up of a set of objects and functions that make up a module or subsystem of the main system. Thus, an application system represents the main system that incorporates many different components put together to achieve a global system purpose or goal. The three different aspects of a system hence determine the type of reuse to be undertaken.

Figure 1 shows the reuse landscape, expressed in terms of approaches that support software reuse. These approaches include design patterns, architectural patterns, ERP systems, application frameworks, COTS integration, program libraries, among others.

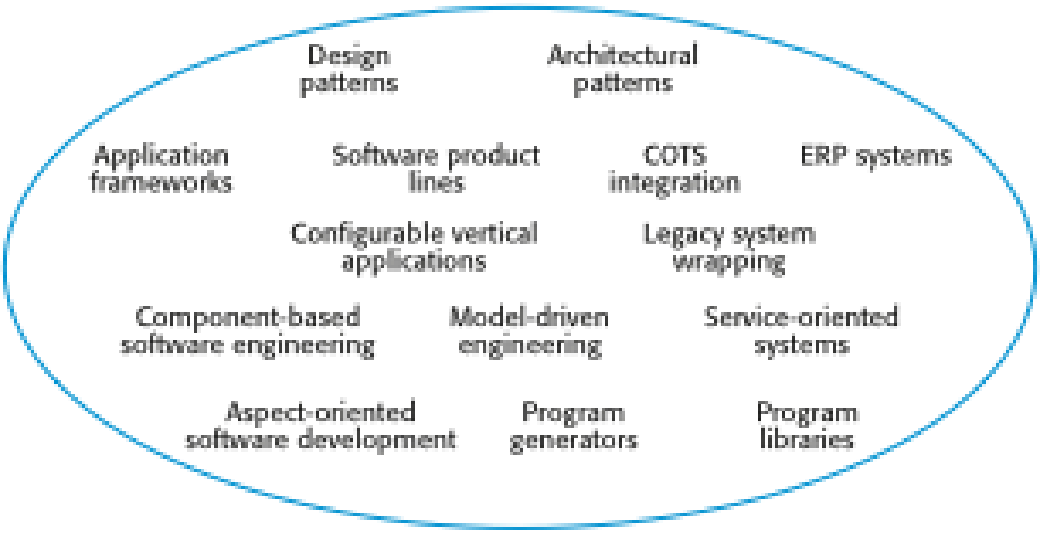

Figure 1. Software Reuse Landscape [1]

COTS Product Reuse

Commercial-off-the-shelf (COTS) software can be adapted for different customers without changing the source code of the system [1]. This is because COTS possess generic features that allow for reuse in different environments. Standard built-in mechanisms give room for tailoring the system to specific customer needs. Thus, COTS reuse leads to the following benefits

1. Rapid deployment of reliable systems.
2. System functionality can be visualized earlier for better decision making on whether to adopt them or not.
3. The reuse helps to avoid some risks that could be incurred in fresh system development.
4. The reuse could help businesses to cut resource costs by focusing on core activities rather than system development.
5. The customer is shielded from technological evolutions since updates remain a sole responsibility of COTS product vendors.

COTS reuse may however face some challenges, including

1. In some cases, system requirements must be adapted to reflect the functionality and mode of operation of the COTS.
2. Some COTS products may be based on practically impossible assumptions to change.

3. The lack of proper COTS products documentation makes decisions on choices of the right COTS system for an enterprise quite difficult.
4. In some cases, there may be no local expertise to support system development.
5. No or lack of COTS product vendor controls system support and evolution.

ERP Systems Reuse

Enterprise Resource Planning (ERP) systems are generic systems that support business processes that are common in a given organizational domain. Such processes include ordering and invoicing, manufacturing, etc. ERPs are very widely used in large companies, making them among the most common forms of software reuse. ERPs generic core is adapted by including modules and by incorporating knowledge of business processes and rules [1]. Figure 2 shows the architecture of an ERP system.

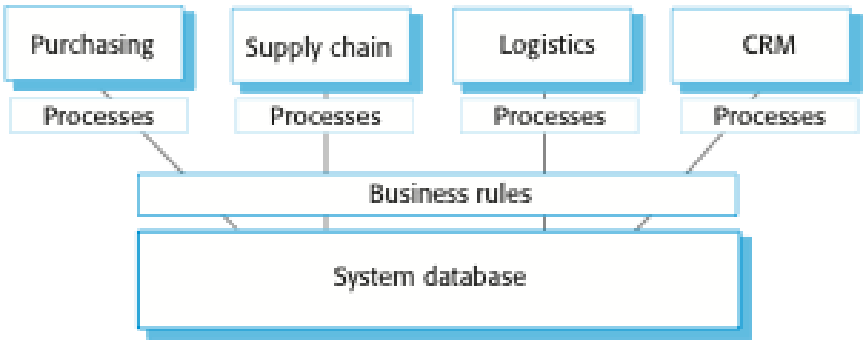

Figure 2. Architecture of an ERP System [1]

According to the architecture, several modules support different business functions. A defined set of business processes is associated with each module, and it maps to the activities in the respective module. The architecture has a common database that maintains information about all related business functions with a set of business rules that apply to all data in the database. The benefits and challenges associated with ERP reuse relate with those discussed under COTS reuse.

According to Sommerville [1], planning for software reuse is affected by a number of factors including

1. The software development schedule
2. The expect software lifetime
3. The development team's background, skills and experience
4. The nonfunctional requirements and criticality of the software

5. The domain of the application
6. The software's execution platform

Benefits and Challenges of Software Reuse

According to Sommerville [1], embracing software reuse leads guarantees the following benefits:

1. Increased dependability – an object, function, or component tried and tested in working system should be more dependable than a completely new object, function, or component since their design and implementation faults should already have been found and fixed.
2. Reduced process risk – the cost of using existing software can be easily estimated as opposed to that of developing a new system which is always a matter of judgement.
3. Effective use of specialists – software developers can develop reusable software that ends up to be a better utilization of their expertise.
4. Standards compliance – some standards like user interface standards can be easily implemented as a set of reusable components. Such standards results to improved dependability as a result of fewer mistakes made by user when using familiar interfaces.
5. Accelerated development – reusing existing system result to early introduction of the system to the market since it speeds up production since both development and validation times are reduced immensely.

Despite the benefits, a number of challenges are associated with software reuse. These includes [1]:

1. Increased maintenance costs – maintenance cost is likely to get high in case the source code of the reused software system or component is not availed.
2. Lack of tool support
3. Not-invented-here syndrome
4. Creating, maintaining, and using component libraries
5. Finding, understanding, and adapting reusable components

Component-Based Software Engineering

Component-based software engineering (CBSE) is a software development approach that advocates for reuse of software entities called software components. According to Sommerville [1], software components are more abstract than object classes and can be considered to be stand-alone service providers that can exist on their own. Components provide a service regardless of where the component is executing or its programming language. Thus, a component is an independent executable entity that can be made up of one or more executable objects. Component interfaces are normally published with all interactions being facilitated through the published interfaces. Part of the characteristics of any given component include standardized, independent, composable, deployable, and documented.

CBSE is based on sound software engineering design principles, in addition to the benefits of reuse. These principles include

1. Components are independent, making them not to interfere with each other.
2. Component implementations are hidden from user's view.
3. Any communication with the component is through well-defined interfaces as shown in Figure 3.
4. Component platforms are shared and reduce development costs.

Figure 3. Component Interface Structure [1]

Generally, standards are required to facilitate communication and interoperation among components. Unfortunately, the existence of many competing standards like Sun's Enterprise Java Beans, Microsoft's COM and .NET, and COBRA's COM, is a hindrance to the uptake of CBSE. In the current state, it is practically impossible for components developed using different approaches to integrate [1]. This becomes one of the main challenges of CBSE. Other challenges include

1. Component trustworthiness with respect to components with no available source code.
2. Component certification with respect to the person or entity responsible for certification.
3. Emergent component property prediction.
4. Trade-offs of requirements analysis between features of two or more components.

Figure 4 shows the basic elements of a component model, that is made up of interfaces, usage information, and deployment.

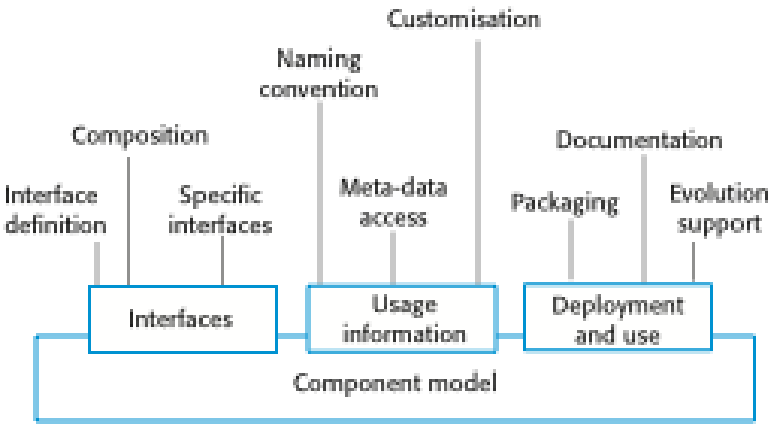

Figure 4. Basic Elements of a Component Model [1]

Components are defined by specifying their interfaces, where the model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions, that should be included in the interface definition. In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. This has to be globally unique and provide component usage information. Finally, part of the component model is a specification of how components should be packaged for deployment and use as independent, executable entities.

Summary

The topic has explored software reuse as one of the concepts applied in Software Engineering. The three types of component reuse have been discussed together with the reuse landscape, expressed in terms of approaches that support software reuse. COTS product and ERP system reuse have been discussed as examples approaches that support reuse. The benefits and

challenges of software reuse have also been covered. Finally, the concept of Component-Based Software Engineering (CBSE) has been briefly covered.

Check Points

1. Discuss the concept of software reuse as applied in Software Engineering
2. State and explain the differences among the three types of software reuse
3. Describe the concept of Component-Based Software Engineering
4. Demonstrate how COTS and ERP systems fit in the software reuse landscape
5. State and explain any three benefits and challenges of software reuse
6. With the aid of a diagram, discuss the basic elements of a component model

Core Textbooks

1. Sommerville, I., Software Engineering, 10th Edition, Addison Wesley, 2016.

Other Resources

References

[1] Sommerville, I., Software Engineering, 10th Edition, Addison Wesley, 2016.

[2] Kendall, K. E. and Kendall, J. E., Systems Analysis and Design, 8th Edition, Pearson, 2011.

[3] Pries, K. H., & Quigley, J. M. (2010). Scrum project management. CRC press.

[4] Schwaber, K. (2004). Agile project management with Scrum. Microsoft press.

[5] Gotterbarn, D., Miller, K., & Rogerson, S. (1997). Software engineering code of ethics. Communications of the ACM, 40(11), 110-118.

[6] Pressman, R. S. (2005). Software engineering: a practitioner's approach. Palgrave macmillan.