Module 8

Software Reuse; Component-Based Software Engineering

Objectives:

1. Understand the pros and cons of software reuse.
2. Understand what aspects of a software engineering project are suitable for reuse.
3. Know what COTS products are and how they are useful.
4. Know what Component-Based Software Engineering (CBSE) is, and know what a component is in the context of CBSE.
5. Understand some of the problems associated with building and assembling components.

Reading:

Sommerville, chapters 15 & 16.

Assignment:

Participate in the group discussion for Module 8.

Quiz:

Take the quiz for Module 8.

Reusable Components

Why waste time reinventing the wheel? If you have a piece of code that performs certain functionality, and you find that you need that functionality for another project, why not just reuse it? The notion of software reusability has been around for a while, with the vision of having a huge repository of components that can be assembled and "glued" together with small amounts of code written from scratch. If you've ever used a library component, such as a pre-built data structure, you have practiced software reusability.

Component reuse offers several benefits:

1. It can save time and consequently, money, by reducing the amount of code that must be written from scratch.

2. Components that are reused often tend to be more reliable than freshly written components. The reason is that when a component is first written and tested, an arbitrary number of faults are found and corrected. When the component is reused for a different project it will be tested again, probably by different engineers using different test cases. A few more faults may be found and corrected. Every time the component is reused additional faults may be discovered and removed, pushing the component closer to fault-free status. At some point a plateau will be reached where few, if any, faults are found, although unless exhaustive testing can be performed you can never really be sure 100% of the faults have been found. But the point is, the more a component is reused the more reliable it will b come.

3. It can improve consistency among aspects or features that are common, either throughout a single application or throughout a family of related applications. You are all probably aware of the common "look and feel" shared by the various Microsoft Office applications. Menus are laid out more or less the same way, the same shortcut keys are used, and so on. Within a single application, there may be several situations where an "Open File" dialog box is used. It makes sense that this dialog box should look the same in every case. This kind of consistency makes software easier to use, and it shortens the learning curve for new applications that share common features.

4. Over time, large libraries of reusable components can be assembled, which can further reduce the effort required to build new projects.

Of course, there can be drawbacks to reusing components:

1. If a component doesn't precisely fit the new specifications, it must be adapted. Often more time can be spent adapting a component than would be spent writing it from scratch.

2. The reliability of a pre-built component is predicated on thorough testing. If the component has only been tested superficially, it may contain more faults than anticipated and require more debugging.

3. Effective reuse requires that a certain amount of infrastructure be in place. Someone must be in charge of maintaining the component library, and manage the different component versions that are created over time. Configuration management systems can help with this, but some human oversight is still necessary.

Effort can be wasted by attempting to incorporate reusability into the design of every component. It takes time to make a component reusable, since you must ensure it is versatile enough to be reused in a wide variety of situations. Some components, such as data structures, lend themselves very well to reuse. The expectations of these components are not likely to change from one project to another, and if they are likely to be needed in many projects, it makes sense to invest some time designing them to be reusable. Some components, however, may have functionality that is so specific to a particular project that they will rarely, be used again. It probably does not make sense to spend a lot of time making these components reusable.

Developers may be unwilling to place trust in components written by someone else. They often feel more comfortable when the work has all been done in-house. This is such a prevalent phenomenon that it has been given a name: **NIH Syndrome**. The letters, "NIH", stand for "Not Invented Here".

Although I have focused the discussion so far on reusing the source code of library components. or the compiled components themselves, entire applications or systems can be reused. You've probably all seen examples of full-fledged applications written using Microsoft Excel, not because Excel was the best option, but because it was a relatively cheap, convenient option that can harness the power of Visual Basic. The enterprise resource planning (ERP) system known as Banner, which all of you use to register for classes and check final course grades, is used at many other universities besides the University of Illinois.

It is also important to recognize that code is not the only product generated during the software engineering process that can be reused. We've already discussed design patterns, which are reusable designs for solving certain types of recurring problems in software engineering. Test cases and test suites can be reused during regression testing. Depending on the situation, it is possible for portions of a software specification to be reused—for example, if a set of applications is being built to share a similar look and feel, or interact with each other. Even certain management products such as risk analyses and feasibility analyses may be able to be reused to some extent.

Key Factors to Consider During Reuse Planning

Sommerville talks about six major factors that should be considered when planning for software reuse. Let's look at those here, as well as one more I would like to add to the list.

1. <u>Development Schedule</u>

   If time is limited, you might want to give priority to searching for a suitable application or set of components that can be reused, rather than develop everything from scratch.

2. <u>Expected Software Lifetime</u>

   If the software you are building is expected to be around for some time, and undergo some upgrades or extension in the future, it is wisest to make sure you have the source code for all the components readily available. If a planned upgrade involves code that is proprietary to another party, there is no way to reliably ensure the upgrade can be done successfully. Third party code can change, too, and the newest version of the component may break software that worked well with an older version.

3. <u>Developer Background and Experience</u>

   In order to effectively incorporate reusable components into a system, there must be enough engineers on the team who have the ability to quickly analyze the components planned for reuse, and adapt them as needed to work with the current project. A balance must be struck between effort spent analyzing and adapting reusable components, and effort spent writing and testing new components.

4. <u>Criticality of the Software</u>

   A critical system is a system on which something of great importance depends, such as human lives, an organization's ability to do business, or money. Because of safety and security issues, it is probably best to have full access to all the source code for the system. Otherwise, it will be impossible to make any guarantees about the system's dependability. In some cases, it may be necessary to purchase the rights to the source code for certain critical components if they cannot be developed in-house.

5. <u>Application Domain</u>

   For certain application domains, there may already exist one or more ready-to-use systems that can be configured as needed to work in a variety of similar, yet different situations. The

Banner ERP system I mentioned earlier is an example of such a system. That system lies in the application domain of educational information management, and different functionalities can be added as "modules" as they are needed. Another example of an application domain not mentioned in the book is the domain of ecommerce shopping cart systems. If you've ever wondered why a lot of the websites that sell merchandise seem to look nearly identical except for the items being sold, odds are the sites were developed by an ecommerce shopping cart system. These systems typically include a number of templates for generating cookie cutter websites for displaying merchandise, as well as subsystems for handling consumer payment. One example of this type of system is XCart.

6. Target Platform

   Some component libraries have been developed to work only with certain hardware and operating systems. The target platform on which the finished software will run will therefore determine which component libraries will be available for use during project development. One of proclaimed advantages of using Java technology is that software written in Java will work on any platform for which a Java Runtime Environment has been created.

7. Customer Requirements

   Although Sommerville doesn't mention it as a separate factor (he sort of mentions it in with the criticality of the software), the customer's requirements will influence the decision about whether or not to employ reuse technologies. If there are no pre-built components that will satisfy the customer's requirements, everything must be developed from scratch. Non-functional requirements can also influence this decision.

Commercial-Off-The-Shelf Components

A Commercial-Off-The-Shelf (COTS) product is defined as a pre-built software system or application that can be used to satisfy a variety of different needs without making any changes to the source code; in other words, it can be used as is. In order to maximize their flexibility, COTS products tend to be heavily customizable, allowing users to alter the look and feel of the user interface, as well as control which functionality is present. The Banner ERP mentioned earlier is a good example of a COTS product. Other examples would be the software usually seen controlling self-checkout lanes at many stores, and the software used in the medical profession to gather and report medical data.

Although COTS products are flexible, there is still no guarantee they will offer the precise functionality required for a particular solution. There are a number of ways to deal with this situation. There may be more than one company that produces COTS products for the application domain of interest, so it won't always be the case that there is only a single option available. In some cases, COTS products are integrated with other COTS products, or even with legacy systems, in order to get the necessary functionality. COTS product integration can be tricky, especially when legacy systems are involved, since many legacy systems may themselves be held together by a bunch of jury-rigged patches. If there is no way to adapt the needs of the problem to any available COTS product, it may be possible to acquire a license that allows limited, non-redistributable modifications to the source code to make the capabilities of the COTS product better meet the customer requirements.

Component-Based Software Engineering (CBSE)

Although the notion of reusable components has been around for a long time, it did not immediately gain wide acceptance in the software engineering community. Many reusable components were made, using a variety of programming languages, but they often lacked the desired versatility. Initially, too much understanding of a component's implementation was needed in order to reuse the component. Additionally, there were several standards used for development of reusable components, and most of the time components built under one standard could not interoperate with components built using another standard. Not until recently has the software engineering community begun to fully embrace the concept of building an entire system out of truly reusable components, possibly components built using different technologies. The result has been the creation of **Component-Based Software Engineering**, or **CBSE** for short.

There are two cardinal rules that all components developed using CBSE should adhere to:

1. Components must operate independently of each other and not interfere with the operation of other components. This is accomplished by using the Principle of Information Hiding to minimize coupling between components.

2. Components must communicate with each other through well-defined interfaces. This is a consequence of the first rule. Implementation details are hidden to decouple components from one another, but there cannot be absolute decoupling, or the components could only operate in isolation. The compromise is to hide the implementation details, but provide a clear, well-defined public interface that controls how components can talk to each other.

If the above rules are followed, component infrastructures can be built to provide a wide variety of services that can be used by other software systems. These infrastructures are similar in nature to standard component libraries, but are designed to be capable of being used by software running on different platforms, in particular distributed systems. The computers that comprise a distributed system may have widely different hardware configurations, and may run several different operating systems. There may also be multiple versions of a given application that needs to use the component infrastructure. The goal of CBSE is to provide all these capabilities with a minimal requirement for writing new code.

The actual definition of the term "component" in CBSE is controversial. Sommerville points out five key characteristics that should be met to distinguish a CBSE component from a regular, run of the mill component:

1. It must be **standardized**; i.e., it must conform to some standard component model.

2. Ideally, the component should be **independent**. It should be able to be created and deployed on its own, with no dependence on other components.

3. It must carry out its functionality through a publicly defined interface, and this interface must be made available to any components that need to use that functionality. Components that meet this requirement are said to be **composable**.

4. It should be self-contained, and able to function as a stand-alone entity; i.e., it should be **deployable** on a platform running an implementation of the component's component model.

5. The component should be fully **documented**, so that developers can understand what the component's capabilities are and how to access them, and to determine whether or not the component meets all the developers' requirements.

The Difference Between Components and Objects

The notion of a component sounds very similar to the concept of an object in object-oriented programming. They are not the same thing, however. Both components and objects encapsulate data and related operations in a single structure, and the functionality of both components and objects can be accessed through a public interface. Components often are developed using the object-oriented approach, but that is about where the similarity ends. The key differences between components and objects, as quoted from Sommerville, are:

1. Components are deployable entities. That is, they are not compiled into an application program but are installed directly on an execution platform. The methods and attributes defined in their interfaces can then be accessed by other components.

2. Components do not define types. A class definition defines an abstract data type and objects are instances of that type. A component is an instance, not a template that is used to define an instance.

3. Component implementations are opaque. Components are, in principle at least, completely defined by their interface specification. The implementation is hidden from component users. Components are often delivered as binary units so the buyer of the component does not have access to the implementation.

4. Components are language-independent. Object classes have to follow the rules of a particular object-oriented programming language and, generally, can only interoperate with other classes in that language. Although components are usually implemented using object-oriented languages, such as Java or C#, you can implement them in non-object-oriented programming languages.

5. Components are standardized. Unlike object classes that you can implement in any way, components must conform to some component model that constrains their implementation.

Component Models in CBSE

A component model, as mentioned above, shares in its definition most of the characteristics of a component in CBSE, but rather than defining particular components, a component model defines how components should be created, and how they should interact with other components. The component model must define the public interface through which the components in the model can be accessed. This includes method/function names, as well as identification of the numbers and types of parameters. A consistent naming convention should be used throughout the component model. Usage of the components in the model usually requires the ability to determine, at runtime, what features are present in a component, and how they can be accessed. This information is typically stored as metadata (i.e., data about a component's data), and can usually be accessed through some defined mechanism provided by the component model. For remote interactions, unique identifiers must be given to components to allow them to be accessed from any location. The component model also defines how a particular component should be deployed, as well as any dependencies for the component. As the components evolve, the deployment process can become somewhat complex, as certain versions of some components may not function with certain versions of other components. Component compatibility must therefore be continually tracked and maintained, since there is no guarantee all systems using the component model will contain the same versions of components.

With regards to specific component models, there are many in existence. Sommerville posits the most important models are currently: the WebServices model, Enterprise Java Beans (EJB), and .NET. A key problem of there being so many component models is the lack of interoperability between components developed using different models. The current solution to this problem is to develop components as stand-alone entities, so that reliance on the installation of the component model is minimized. This type of architecture is known as **Service Oriented Architecture**, and the related development process is called **Service Oriented Software Engineering**. We'll cover these in more detail in the next module.