
Thakur Educational Trust's (Regd.)

THAKUR RAMNARAYAN COLLEGE OF ARTS & COMMERCE

ISO 21001:2018 Certified



# PROGRAMME: B.Sc (I.T)

# CLASS: S.Y.B.Sc (I.T)

# SUBJECT NAME: SOFTWARE ENGINEERING

# SEMESTER: IV

# FACULTY NAME: Ms. SMRITI DUBEY

# UNIT V

# Chapter 3 – Software Reuse



## Concepts:

Introduction

The Reuse Landscape

Application Frameworks

Software Product Lines

Cost Product Reuse

## Introduction

Reuse-based software engineering is a software engineering strategy where the development process is geared to reusing existing software.

In most engineering disciplines, systems are designed by composing existing components that have been used in other systems. Software engineering has been more focused on original development but it is now recognized that to achieve better software, more quickly and at lower cost, we need a design process that is based on systematic software reuse. There has been a major switch to reuse-based development over the past 10 years.

**Scale** of software reuse:

- **System reuse:** Complete systems, which may include several application programs.

- **Application reuse:** An application may be reused either by incorporating it without change into other or by developing application families.
- **Component reuse:** Components of an application from sub-systems to single objects may be reused.
- **Object and function reuse:** Small-scale software components that implement a single well-defined object or function may be reused.

**Benefits** of software reuse:

- **Accelerated development:** Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.
- **Effective use of specialists:** Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
- **Increased dependability:** Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.
- **Lower development costs:** Development costs are proportional to the size of the software being developed. Reusing software means that fewer lines of code have to be written.
- **Reduced process risk:** The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.
- **Standards compliance:** Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.

**Problems** with software reuse:

- **Creating, maintaining, and using a component library:** Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
- **Finding, understanding, and adapting reusable components:** Software components have to be discovered in a library, understood and, sometimes, adapted

to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.

- **Increased maintenance costs:** If the source code of a reused software system or component is not available then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.
- **Lack of tool support:** Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is particularly true for tools that support embedded systems engineering, less so for object-oriented development tools.
- **Not-invented-here syndrome:** Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.

**The Reuse Landscape**

Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used. Reuse is possible at a range of levels from simple functions to complete application systems. The reuse landscape covers the range of possible reuse techniques.

Key factors that you should consider when planning reuse are:

**1. The development schedule for the software** - If the software has to be developed quickly, you should try to reuse off-the-shelf systems rather than individual components. These are large-grain reusable assets. Although the fit to requirements may be imperfect, this approach minimizes the amount of development required.

**2. The expected software lifetime** - If you are developing a long-lifetime system, you should focus on the maintainability of the system. You should not just think about the immediate benefits of reuse but also of the long-term implications. Over its lifetime, you will have to adapt the system to new requirements, which will mean making changes to parts of the system. If you do not have access to the source code, you may prefer to avoid off-the-shelf components and systems from external suppliers; suppliers may not be able to continue support for the reused software.

**3. The background, skills, and experience of the development team** - All reuse technologies are fairly complex and you need quite a lot of time to understand and use them effectively. Therefore, if the development team has skills in a particular area, this is probably where you should focus.

**4. The criticality of the software and its non-functional requirements** - for a critical system that has to be certified by an external regulator, you may have to create a dependability case for the system . This is difficult if you don't have access to the source code of the software. If your software has stringent performance requirements, it may be impossible to use strategies such as generator-based reuse, where you generate the code from a reusable domain specific representation of a system. These systems often generate relatively inefficient code.

**5. The application domain** - In some application domains, such as manufacturing and medical information systems, there are several generic products that may be reused by configuring them to a local situation. If you are working in such a domain, you should always consider these as an option.

**6. The platform on which the system will run** - Some components models, such as .NET, are specific to Microsoft platforms. Similarly, generic application systems may be platform-specific and you may only be able to reuse these if your system is designed for the same platform.

Figure sets out a number of possible ways of implementing software reuse,

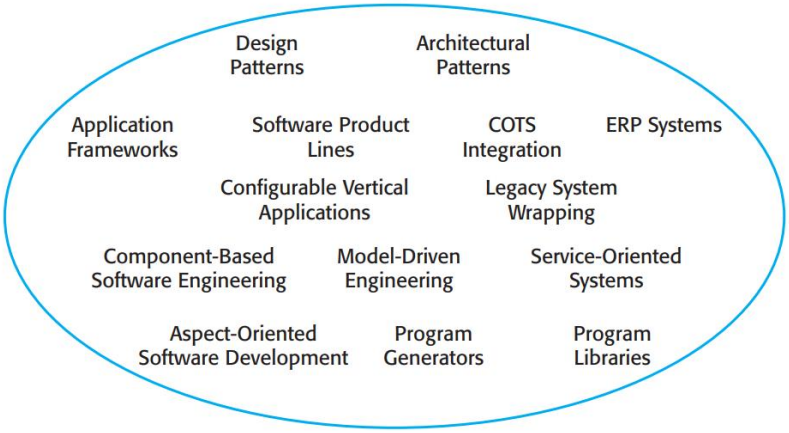



**Figure 16.3** The reuse landscape

| Approach | Description |
|---|---|
| Architectural patterns | Standard software architectures that support common types of application systems are used as the basis of applications. Described in Chapters 6, 13, and Chapter 20. |
| Design patterns | Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions. Described in Chapter 7. |
| Component-based development | Systems are developed by integrating components (collections of objects) that conform to component-model standards. Described in Chapter 17. |
| Application frameworks | Collections of abstract and concrete classes are adapted and extended to create application systems. |
| Legacy system wrapping | Legacy systems (see Chapter 9) are 'wrapped' by defining a set of interfaces and providing access to these legacy systems through these interfaces. |
| Service-oriented systems | Systems are developed by linking shared services, which may be externally provided. Described in Chapter 19. |
| Software product lines | An application type is generalized around a common architecture so that it can be adapted for different customers. |
| COTS product reuse | Systems are developed by configuring and integrating existing application systems. |
| ERP systems | Large-scale systems that encapsulate generic business functionality and rules are configured for an organization. |
| Configurable vertical applications | Generic systems are designed so that they can be configured to the needs of specific system customers. |
| Program libraries | Class and function libraries that implement commonly used abstractions are available for reuse. |
| Model-driven engineering | Software is represented as domain models and implementation independent models and code is generated from these models. Described in Chapter 5. |
| Program generators | A generator system embeds knowledge of a type of application and is used to generate systems in that domain from a user-supplied system model. |
| Aspect-oriented software development | Shared components are woven into an application at different places when the program is compiled. Described in Chapter 21. |



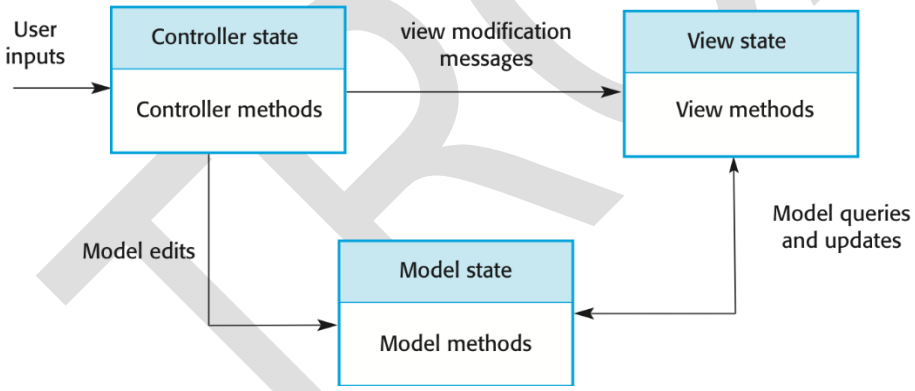
**Figure - Approaches that support software reuse**

**Application Framework**

Frameworks are moderately large entities that can be reused. They are somewhere between system and component reuse. Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them. The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework.

Application frameworks are moderately large entities that can be reused. They are somewhere between system and component reuse. Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them. The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework.

Web application frameworks (WAF) support the construction of dynamic websites as a front-end for web applications. WAFs are now available for all of the commonly used web programming languages e.g., Java, Python, Ruby, etc. Interaction model is based on the Model-View-Controller composite design pattern. An MVC framework supports the presentation of data in different ways and allows interaction with each of these presentations. When the data is modified through one of the presentations, the system model is changed and the controllers associated with each view update their presentation.




WAF features:

- **Security:** WAFs may include classes to help implement user authentication (login) and access.
- **Dynamic web pages:** Classes are provided to help you define web page templates and to populate these dynamically from the system database.
- **Database support:** The framework may provide classes that provide an abstract interface to different databases.

- **Session management:** Classes to create and manage sessions (a number of interactions with the system by a user) are usually part of a WAF.
- **User interaction:** Most web frameworks now provide AJAX support, which allows more interactive web pages to be created.

Frameworks are generic and are extended to create a more specific application or sub-system. They provide a skeleton architecture for the system. Extending the framework involves Adding concrete classes that inherit operations from abstract classes in the framework; Adding methods that are called in response to events that are recognized by the framework. Problem with frameworks is their complexity which means that it takes a long time to use them effectively.

**Software Product Lines**

Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context. A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements. Examples: a mobile operating system that works on different hardware models, a software line for a family of printers with varying features. Adaptation of a software line may involve:

- Component and system configuration;
- Adding new components to the system;
- Selecting from a library of existing components;
- Modifying components to meet new requirements.

The **base application** of a software product line includes:

**Core components** that provide infrastructure support. These are not usually modified when developing a new instance of the product line.

**Configurable components** that may be modified and configured to specialize them to a new application. Sometimes, it is possible to reconfigure these components without changing their code by using a built-in component configuration language.

**Specialized, domain-specific components** some or all of which may be replaced when a new instance of a product line is created.

**Application frameworks vs product lines:**

- Application frameworks rely on object-oriented features such as polymorphism to implement extensions. Product lines need not be object-oriented (e.g. embedded software for a mobile phone)
- Application frameworks focus on providing technical rather than domain-specific support. Product lines embed domain and platform information.
- Product lines often control applications for equipment.
- Software product lines are made up of a family of applications, usually owned by the same organization.

Various types of specialization of a software product line may be developed:

- Different versions of the application are developed for different **platforms**.
- Different versions of the application are created to handle different operating **environments** e.g., different types of communication equipment.
- Different versions of the application are created for customers with different **functional** requirements.
- Different versions of the application are created to support different business **processes**.

**COTS Product Reuse**

A commercial-off-the-shelf (COTS) product is a software system that can be adapted to the needs of different customers without changing the source code of the system. Virtually all desktop software and a wide variety of server products are COTS software.

Because this software is designed for general use, it usually includes many features and functions. It therefore has the potential to be reused in different environments and as part of different applications.

Torchiano and Morisio (2004) also discovered that using open-source products were often used as COTS products. That is, the open-source systems were used without change and without looking at the source code. COTS products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs.

For example, in a hospital patient record system, separate input forms and output reports might be defined for different types of patients. Other configuration features may allow the system to accept plug-ins that extend functionality or check user inputs to ensure that they are valid.

This approach to software reuse has been very widely adopted by large companies over the last 15 or so years, as it **offers significant benefits over customized software development:**

1. As with other types of reuse, more rapid deployment of a reliable system may be possible.

2. It is possible to see what functionality is provided by the applications and so it is easier to judge whether or not they are likely to be suitable. Other companies may already use the applications so experience of the systems is available.

3. Some development risks are avoided by using existing software. However, this approach has its own risks, as I discuss below.

4. Businesses can focus on their core activity without having to devote a lot of resources to IT systems development.

5. As operating platforms evolve; technology updates may be simplified as these are the responsibility of the COTS product vendor rather than the customer.

**Of course, this approach to software engineering has its own problems**:

1. Requirements usually have to be adapted to reflect the functionality and mode of operation of the COTS product. This can lead to disruptive changes to existing business processes.

2. The COTS product may be based on assumptions that are practically impossible to change. The customer must therefore adapt their business to reflect these assumptions.

3. Choosing the right COTS system for an enterprise can be a difficult process, especially as many COTS products are not well documented. Making the wrong choice could be disastrous as it may be impossible to make the new system work as required.

4. There may be a lack of local expertise to support systems development. Consequently, the customer has to rely on the vendor and external consultants for development advice. This advice may be biased and geared to selling products and services, rather than meeting the real needs of the customer.

5. The COTS product vendor controls system support and evolution. They may go out of business, be taken over, or may make changes that cause difficulties for customers