# Objectives

- To explain the benefits of software reuse and some reuse problems
- To discuss several different ways to implement software reuse
- To explain how reusable concepts can be represented as patterns or embedded in program generators
- To discuss COTS reuse
- To describe the development of software product lines

## Topics covered

- The reuse landscape
- Design patterns
- Generator based reuse
- Application frameworks
- Application system reuse

## Software reuse

- 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 recognised that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on **systematic software reuse**.
Reuse-based software engineering

- Application system reuse
  - The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families.
- Component reuse
  - Components of an application from sub-systems to single objects may be reused. Covered in Chapter 19.
- Object and function reuse
  - Software components that implement a single well-defined object or function may be reused.

Reuse benefits 1

| Increased dependability | Reused software, that has been tried and tested in working systems, should be more dependable than new software. The initial use of the software reveals any design and implementation faults. These are then fixed, thus reducing the number of failures when the software is reused. |
| Reduced process risk     | If software exists, there is less uncertainty in the costs of reusing that software than in the costs of development. This is an important factor for project management as it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as sub-systems are reused. |
| Effective use of specialists | Instead of application specialists doing the same work on different projects, these specialists can develop reusable software that encapsulate their knowledge. |

Reuse benefits 2

| Standards compliance | Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interfaces are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability as users are less likely to make mistakes when presented with a familiar interface. |
| 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 should be reduced. |

Reuse problems 1

| Increased maintenance costs | If the source code of a reused software system or component is not available then maintenance costs may be increased as the reused elements of the system may become increasingly incompatible with system changes. |
| Lack of tool support | CASE toolsets may 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. |
| Not-invented-here syndrome | Some software engineers sometimes prefer to re-write components as they believe that they can improve on the reusable component. 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. |
Reusing approaches 1

- **Design patterns**: Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions.

- **Component-based development**: Systems are developed by integrating components (collections of objects) that conform to component-model standards. This is covered in Chapter 19.

- **Application frameworks**: Collections of abstract and concrete classes that can be adapted and extended to create application systems.

- **Legacy system wrapping**: Legacy systems (see Chapter 2) that can be "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 that may be externally provided.

---

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.

---

The reuse landscape

- Creating and maintaining a component library
- Finding, understanding and adapting reusable components

Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.

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 will make routinely include a component search as part of their normal development process.
Reuse approaches 2

<table>
<thead>
<tr>
<th>Application product lines</th>
<th>An application type is generalised around a common architecture so that it can be adapted in different ways for different customers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COTS integration</td>
<td>Systems are developed by integrating existing application systems.</td>
</tr>
<tr>
<td>Configurable vertical applications</td>
<td>A generic system is designed so that it can be configured to the needs of specific system customers.</td>
</tr>
<tr>
<td>Program libraries</td>
<td>Class and function libraries implementing commonly-used abstractions are available for reuse.</td>
</tr>
<tr>
<td>Program generators</td>
<td>A generator system embeds knowledge of a particular type of application and can generate systems or system fragments in that domain.</td>
</tr>
<tr>
<td>Aspect-oriented software development</td>
<td>Shared components are woven into an application at different places when the program is compiled.</td>
</tr>
</tbody>
</table>

Reuse planning factors

- The development schedule for the software.
- The expected software lifetime.
- The background, skills and experience of the development team.
- The criticality of the software and its non-functional requirements.
- The application domain.
- The execution platform for the software.

Concept reuse

- When you reuse program or design components, you have to follow the design decisions made by the original developer of the component.
- This may limit the opportunities for reuse.
- However, a more abstract form of reuse is concept reuse when a particular approach is described in an implementation independent way and an implementation is then developed.
- The two main approaches to concept reuse are:
  - Design patterns;
  - Generative programming.

Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Patterns often rely on object characteristics such as inheritance and polymorphism.
Pattern elements

• Name
  – A meaningful pattern identifier.
• Problem description.
• Solution description.
  – Not a concrete design but a template for a design solution that can be instantiated in different ways.
• Consequences
  – The results and trade-offs of applying the pattern.

The Observer pattern

• Name
  – Observer.
• Description
  – Separates the display of object state from the object itself.
• Problem description
  – Used when multiple displays of state are needed.
• Solution description
  – See slide with UML description.
• Consequences
  – Optimisations to enhance display performance are impractical.
Application frameworks

- 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.
- Frameworks are moderately large entities that can be reused.

Framework classes

- System infrastructure frameworks
  - Support the development of system infrastructures such as communications, user interfaces and compilers.
- Middleware integration frameworks
  - Standards and classes that support component communication and information exchange.
- Enterprise application frameworks
  - Support the development of specific types of application such as telecommunications or financial systems.

Extending frameworks

- Frameworks are generic and are extended to create a more specific application or sub-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 recognised by the framework.
- Problem with frameworks is their complexity which means that it takes a long time to use them effectively.

Model-view controller

- System infrastructure framework for GUI design.
- Allows for multiple presentations of an object and separate interactions with these presentations.
- MVC framework involves the instantiation of a number of patterns (as discussed earlier under concept reuse).
Application system reuse

- Involves the reuse of entire application systems either by configuring a system for an environment or by integrating two or more systems to create a new application.
- Two approaches covered here:
  - COTS product integration;
  - Product line development.