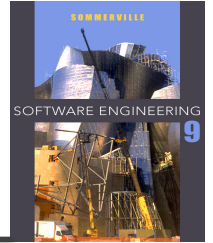


# Chapter 16 – Software Reuse

## Lecture 1

# Topics covered

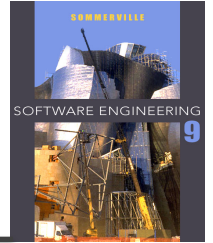
---



- ✧ The reuse landscape (benefits and problems)
- ✧ Application frameworks
- ✧ Software product lines
- ✧ COTS product reuse

# Software reuse

---

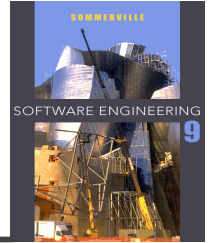


- ✧ In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.
- ✧ Software engineering has been more focused on original development but it is now recognised that to achieve **better** software, more **quickly** and at **lower cost**, we need 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.

Although reuse was proposed as a development strategy more than 40 years ago (McIlroy, 1968), it is only since **2000** that '**development with reuse**' has become the **norm for new business systems**.

# Reuse-based software engineering

---



## ✧ (1) 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.

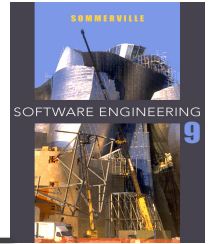
## ✧ (2) Component reuse

- Components of an application from **sub-systems** to **single objects** may be reused. Covered in Chapter 17.

## ✧ (3) Object and function reuse

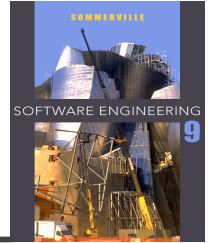
- Software components that implement a **single well-defined object** or **function** may be reused.

# Benefits of software reuse



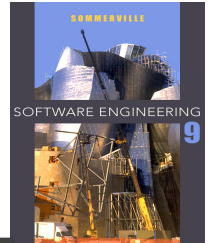
Benefit	Explanation
Increased dependability	Reused software, which has been tried and tested in working systems, should be <b>more dependable than new software</b> . Its design and implementation faults should have been found and fixed.
Reduced process risk	The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it <b>reduces the margin of error</b> in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.
Effective use of specialists	Instead of doing the same work over and over again, <b>application specialists</b> can develop reusable software that <b>encapsulates their knowledge</b> .

# Benefits of software reuse



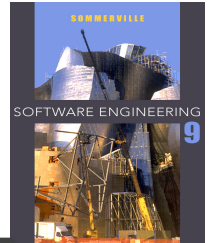
Benefit	Explanation
Standards compliance	Some standards, such as <a href="#">user interface standards</a> , can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.
Accelerated development	<a href="#">Bringing a system to market as early as possible</a> is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.

# Problems with reuse



Problem	Explanation
Increased maintenance costs	If the <b>source code</b> of a reused software system or component is <b>not available</b> then maintenance costs may be higher because the reused elements of the system may become increasingly <b>incompatible with system changes</b> .
Lack of tool support	Some software <b>tools do not support</b> development with <b>reuse</b> . 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 <b>embedded systems engineering</b> , less so for object-oriented development tools.
Not-invented-here syndrome	Some software engineers prefer to rewrite components because they believe they can <b>improve</b> on them. This is partly to do with <b>trust</b> and partly to do with the fact that writing original software is seen as <b>more challenging</b> than reusing other people's software.

# Problems with reuse



Problem	Explanation
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.

- Software development processes have to be adapted to take reuse into account;
- Software reuse is most effective when it is planned as part of an organization-wide reuse program.
  - Importance of reuse planning in Japan: Matsumoto, 1984, Japanese 'factory' approach: Cusamano, 1989;
  - Company: Hewlett-Packard(HP); Programs: Griss and Wosser, 1995

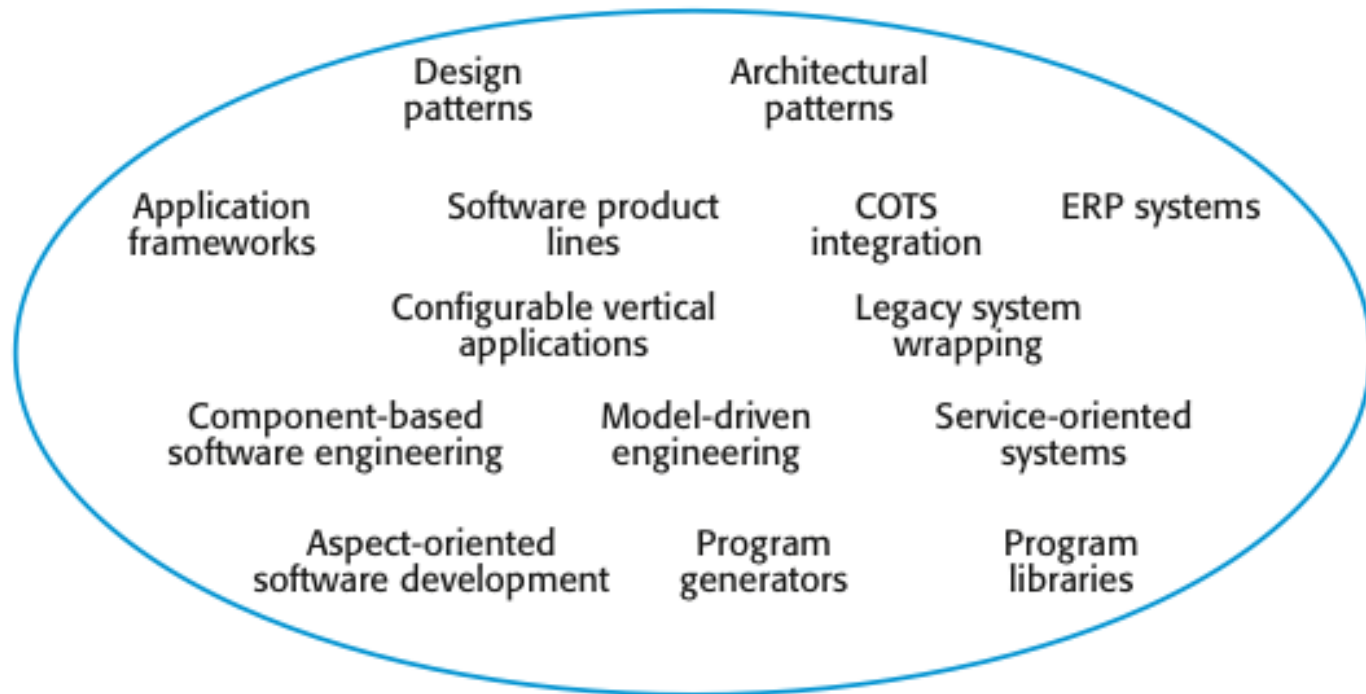
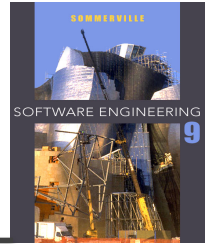


## 16.1 The reuse landscape

---

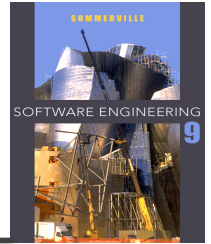
- ✧ Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used.
- ✧ Reuse is possible at a **range of levels** from simple functions to complete application systems.
- ✧ The reuse landscape covers the range of possible reuse techniques.
- ✧ Reuse techniques exploit the facts that **systems in the same application domain are similar and have potential for reuse.**

# The reuse landscape



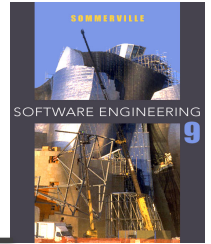
These reuse approaches are described briefly in the next three slides.

# Approaches that support software reuse



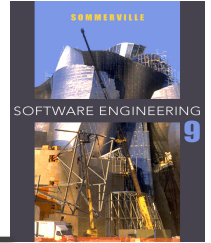
Approach	Description
Architectural patterns	Standard software architectures that support common types of application systems are used as the basis of applications. Described in Chapters 6, 13, and 20.
Design patterns	Generic abstractions that occur across applications are represented as design patterns 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.

# Approaches that support software reuse



Approach	Description
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.

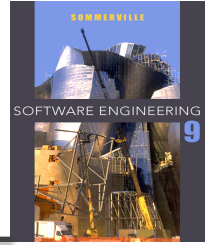
# Approaches that support software reuse



Approach	Description
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.

# Reuse planning factors

---



## ✧ Six Key factors:

- ✧ (1) The development schedule for the software.
  - ✧ (2) The expected software lifetime.
  - ✧ (3) The background, skills and experience of the development team.
  - ✧ (4) The criticality of the software and its non-functional requirements.
  - ✧ (5) The application domain.
  - ✧ (6) The execution platform for the software. E.g. .NET is specific to Microsoft platforms.
- ✧ Whether or not reuse is achieved is often a **managerial** rather than a **technical** issue. Managers may be unwilling to compromise their requirements to allow reusable components to be used.
- ✧ Although the risks of new software development may be higher, some managers may **prefer known** to unknown **risks**.

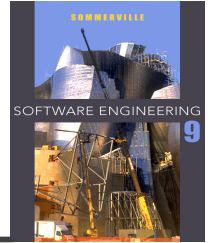
## 16.2 Application frameworks

---

- ✧ 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.

# Framework classes

---



## ✧ (1) System infrastructure frameworks

- Support the development of system infrastructures such as communications, user interfaces and compilers.

## ✧ (2) Middleware integration frameworks

- Standards and classes that support component communication and information exchange(e.g. ,MS. :.NET, Java: EJB).

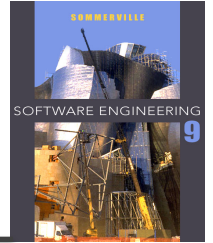
## ✧ (3) Enterprise application frameworks

- Support the development of specific types of application such as telecommunications or financial systems.



# Web application frameworks

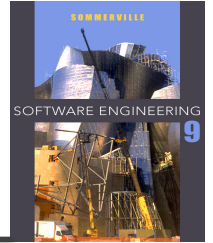
---



- ✧ 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(The architecture of a WAF) is based on the **M**odel-**V**iew-**C**ontroller composite pattern.

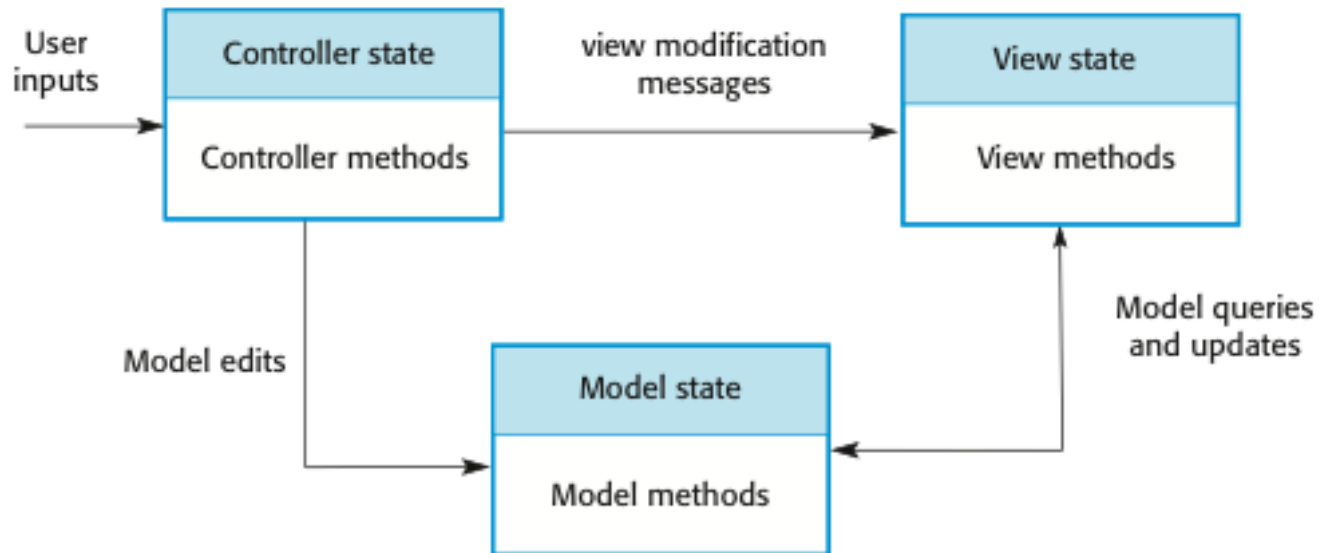
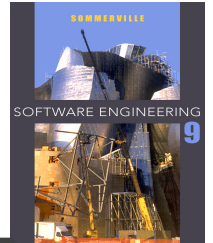
# Model-view controller

---



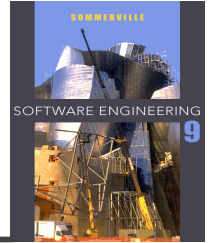
- ✧ The **MVC pattern** was originally proposed in the 1980s as an approach to **GUI design** that allowed for multiple presentations of an object and **separate styles of interaction with** each of these **presentations**.
- ✧ A **MVC framework** supports the presentation of data in different ways and allows interaction with each of these presentations.
- ✧ MVC framework involves the **instantiation** of a number of patterns (as discussed in Chapter 7), including: **the Observer pattern, Strategy pattern, the Composition pattern, etc.**

# The Model-View-Controller pattern



# WAF features

---

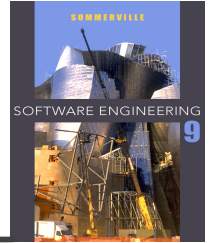


Most Web application frameworks support the following features:

- ✧ (1) *Security*. WAFs may include classes to help implement user authentication (login) and access.
- ✧ (2) *Dynamic web pages*. Classes are provided to help you define web page templates and to populate these dynamically with specific data from the system database.
- ✧ (3) *Database support*. The framework may provide classes that provide an abstract interface to different databases.
- ✧ (4) *Session management*. Classes to create and manage sessions (a number of interactions with the system by a user) are usually part of a WAF.
- ✧ (5) *User interaction*. Most web frameworks now provide AJAX support (Holdener, 2008), which allows more interactive web pages to be created.

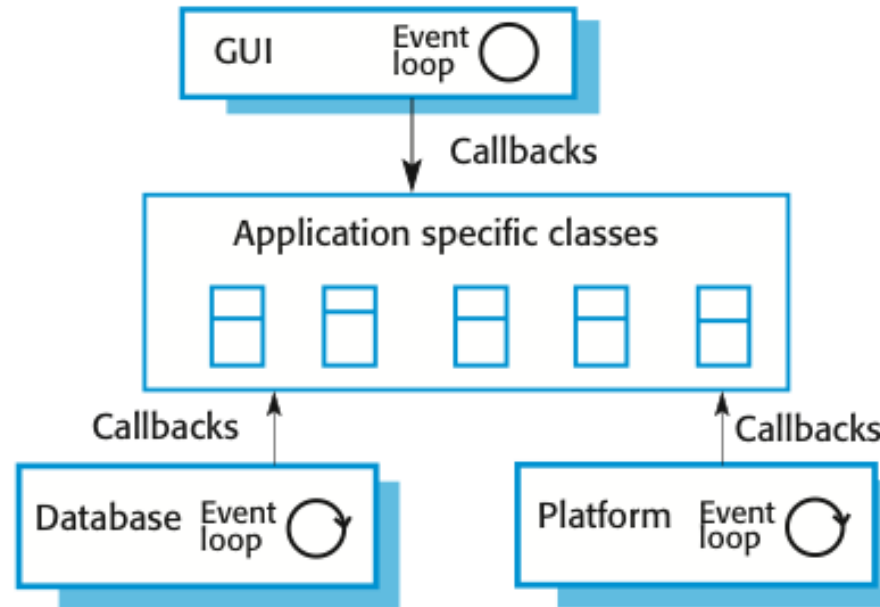
# Extending frameworks

---



- ✧ 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;
  - Defining callback 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.

# 'Inversion of control' in frameworks

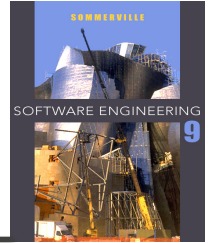


- 'Callbacks' are methods that are called in response to events recognized by the framework.
- The framework objects are responsible for control in the system, they invoke 'hook methods'.
- The application-specific functionality responds to the event in an appropriate way.

*Frameworks are usually more general than software product lines, which focus on a specific family of application system.*

# Key points

---



- ✧ Most new business software systems are now developed by reusing knowledge and code from previously implemented systems.
- ✧ There are many different ways to reuse software. These range from the reuse of classes and methods in libraries to the reuse of complete application systems.
- ✧ The advantages of software reuse are lower costs, faster software development and lower risks. System dependability is increased. Specialists can be used more effectively by concentrating their expertise on the design of reusable components.
- ✧ Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialization and the addition of new objects. They usually incorporate good design practice through design patterns.

# Chapter 16 – Software Reuse

## Lecture 2



## 16.3 Software product lines

---

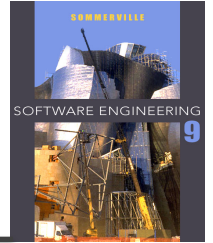
- ✧ **Software product lines** or **application families** are applications with generic functionality that can be adapted and configured for use in a specific context.
- ✧ A software product line is **a set of applications** with a common architecture and shared components, with each application specialized to reflect different requirements.
- ✧ **Adaptation** 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.

# Application frameworks and 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.

# Product line specialisation



## ✧ Platform specialization

- Different versions of the application are developed for different **platforms**. E.g., versions of the application may exist for Windows, Mac OS, and Linux platforms.

## ✧ Environment specialization

- Different versions of the application are created to handle different **operating environments**, e.g. different types of communication equipment. E.g., a system for the emergency services may exist in different versions, depending on the vehicle communications system.

## ✧ Functional specialization

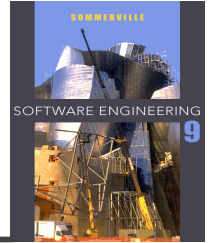
- Different versions of the application are created for customers with different **requirements**. E.g., a library automation system may be modified depending on whether it is used in a *public* library, a *reference* library, or a *university* library.

## ✧ Process specialization

- Different versions of the application are created to support different **business processes**. E.g., an ordering system may be centralized or distributed.

# Product line architectures

---



- ✧ Architectures must be structured in such a way to **separate different sub-systems** and to allow them to be modified.
- ✧ The architecture should also separate entities and their descriptions, and the **higher levels** in the system **access entities through descriptions** rather than directly.

# The architecture of a resource allocation system

## Interaction

User interface

## I/O management

User  
authentication

Resource  
delivery

Query  
management

## Resource management

Resource  
tracking

Resource policy  
control

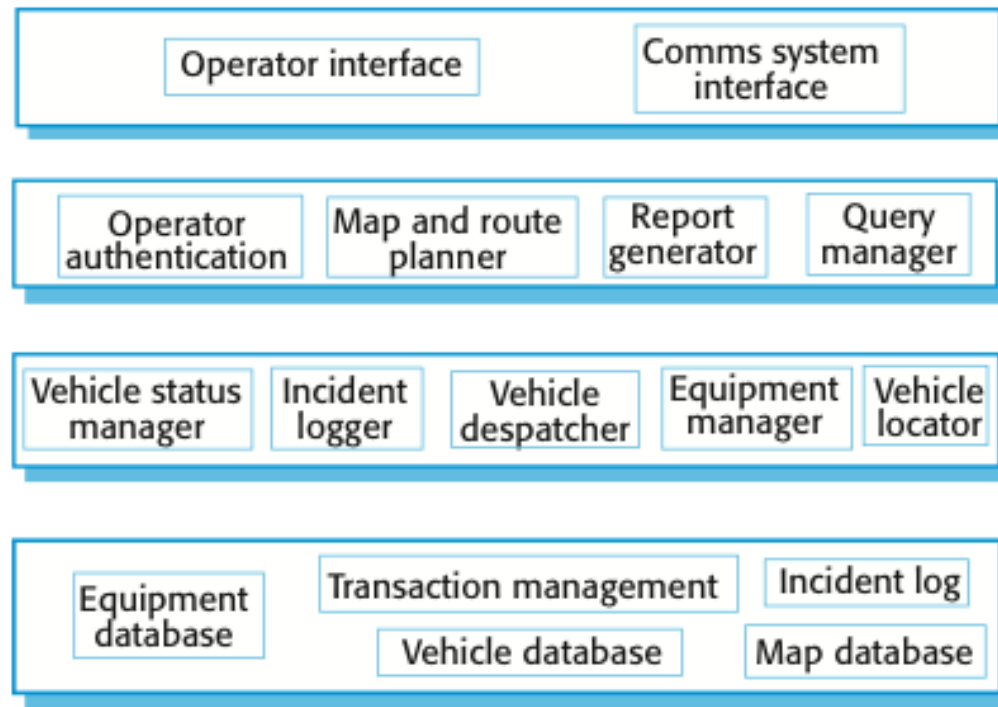
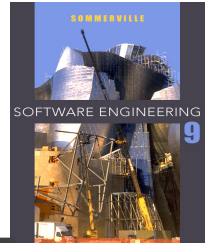
Resource  
allocation

## Database management

Transaction management  
Resource database

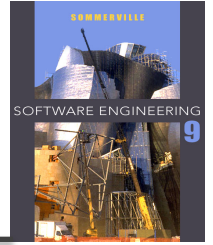
- This four-layer structure is [instantiated](#) in next slide.

# The product line architecture of a vehicle dispatcher



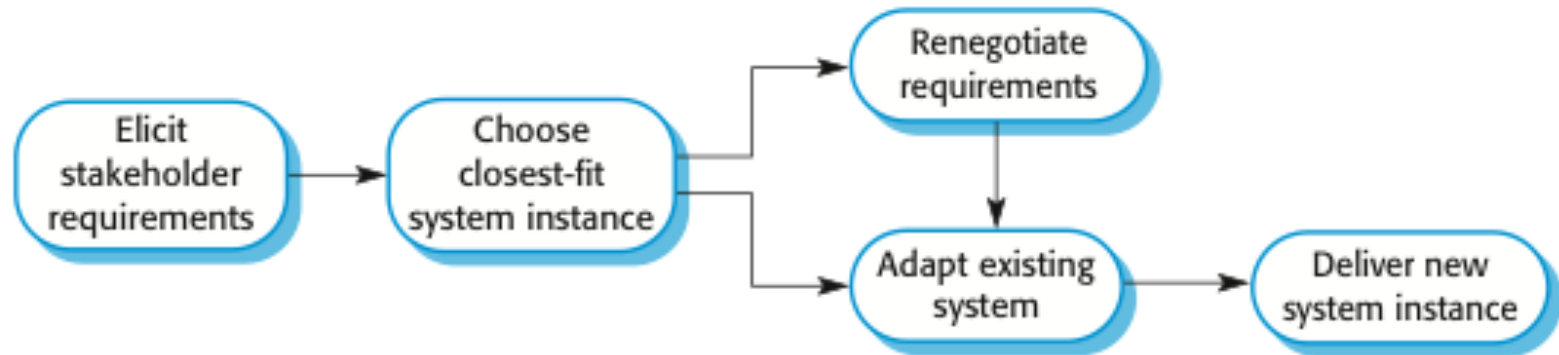
- This Figure shows the **modules** that might be included in a vehicle despatching system product line.

# Vehicle dispatching



- ✧ A specialised resource management system where the aim is to allocate **resources** (**vehicles**) to handle incidents.
- ✧ **Adaptations**(The components at each level) include:
  - At the **UI** level, there are components for operator display and communications;
  - At the **I/O management** level, there are components that handle authentication, reporting and route planning;
  - At the **resource management** level, there are components for vehicle location and despatch, managing vehicle status and incident logging;
  - At the **database** level, as well as the usual transaction management support, there are separate databases of equipment, vehicles and maps.

# Product instance development



The steps involved in [extending a software product line](#) to create a new application, more details in next slide.



# Product instance development

---

- ✧ (1) Elicit stakeholder requirements
  - Use existing family member as a prototype.
- ✧ (2) Choose closest-fit family member
  - Find the family member that best meets the requirements.
- ✧ (3) Re-negotiate requirements
  - Adapt requirements as necessary to capabilities of the software.
- ✧ (4) Adapt existing system
  - Develop new modules and make changes for family member.
- ✧ (5) Deliver new family member
  - The new instance of the product line is delivered to the customer. You should document key features for further member development.

# Product line configuration

---

Software product lines are designed to be reconfigured and this reconfiguration may involve **adding or removing components** from the system, **defining parameters and constraints** for system components, and **including knowledge of business processes**.

This configuration may occur at different **stages** in the development process:

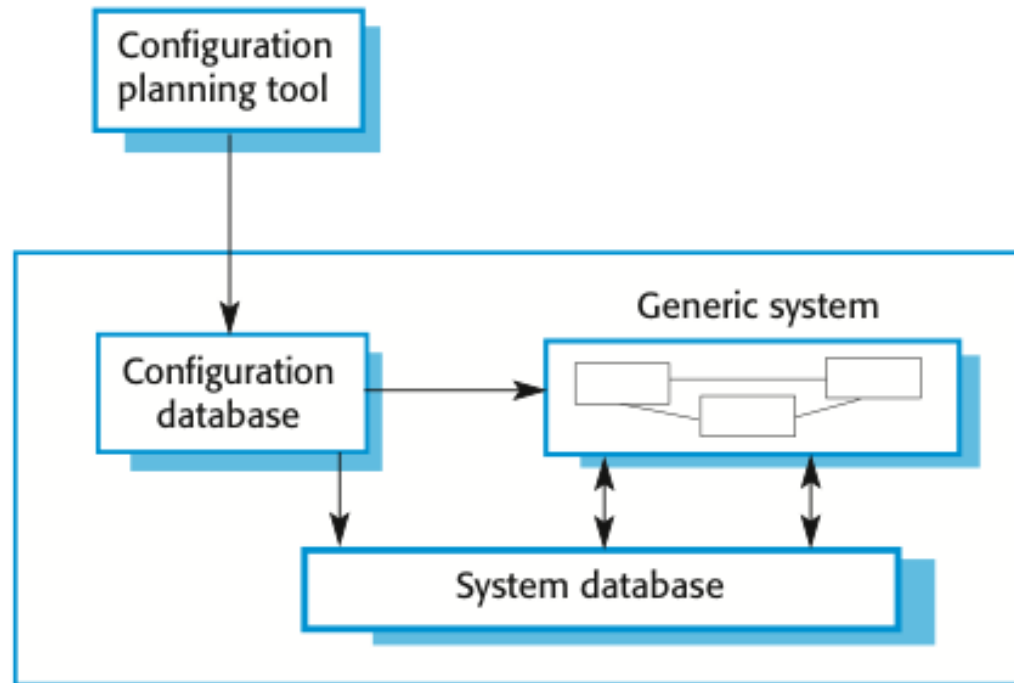
## ✧ (1) **Design time** configuration

- The product line is adapted and changed according to the requirements of particular customers.

## ✧ (2) **Deployment time** configuration

- The product line is configured by **embedding knowledge** of the customer's requirements and business processes. The software source code itself is not changed.

# Deployment-time configuration



- Deployment-time configuration involves using a configuration **tool** to create a specific system configuration that is recorded in a configuration **database** or as a set of configuration files.
- The **executing system** consults this database when executing so that its functionality may be specialized to its execution context.

# Levels of deployment time configuration

---

There are several **levels** of deployment-time configuration that may be provided in a system:

- ✧ (1) **Component selection**, where you select the modules in a system that provide the required functionality.
- ✧ (2) **Workflow and rule definition**, where you define workflows (how information is processed, stage-by-stage) and validation rules that should apply to information entered by users or generated by the system.
- ✧ (3) **Parameter definition**, where you specify the values of specific system parameters that reflect the instance of the application that you are creating.

## 16.4 COTS product reuse

- ✧ A commercial-off-the-shelf (**COTS**) product is a software system that can be adapted for different customers **without changing the source code** of the system.
- ✧ COTS systems have **generic features** and so can be used/reused in different environments.
- ✧ 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 patient.

# Benefits of COTS reuse

---

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.
- ✧ (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.

# Problems of COTS reuse

---

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**.
- ✧ (4) There may be a **lack of local expertise** to support systems development.
- ✧ (5) The COTS product vendor controls system support and evolution.

# COTS-solution and COTS-integrated systems

There are two types of COTS product reuse, namely **COTS-solution systems** and **COTS-integrated systems**:

<b>COTS-solution systems</b>	<b>COTS-integrated systems</b>
Single product that provides the functionality required by a customer	Several heterogeneous system products are integrated to provide customized functionality
Based around a generic solution and standardized processes	Flexible solutions may be developed for customer processes
Development focus is on system configuration	Development focus is on system integration
System vendor is responsible for maintenance	System owner is responsible for maintenance
System vendor provides the platform for the system	System owner provides the platform for the system

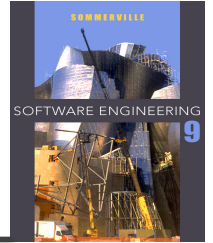


## 16.4.1 COTS solution systems

- ✧ COTS-solution systems are **generic** application systems that may be designed to **support** a **particular business type**, business activity or, sometimes, a complete business enterprise.
  - For example, a COTS-solution system may be produced for **dentists** that handles appointments, dental records, patient recall, etc.
- ✧ **Domain-specific** COTS-solution systems, such as systems to support a business function (e.g. document management) provide functionality that is likely to be required by a range of potential users.
  - For example, a system to support **student registration** in a university may assume that students will be registered for one degree at one university. However, if universities collaborate to offer joint degrees, then it may be practically impossible to represent this in the system.

# ERP systems

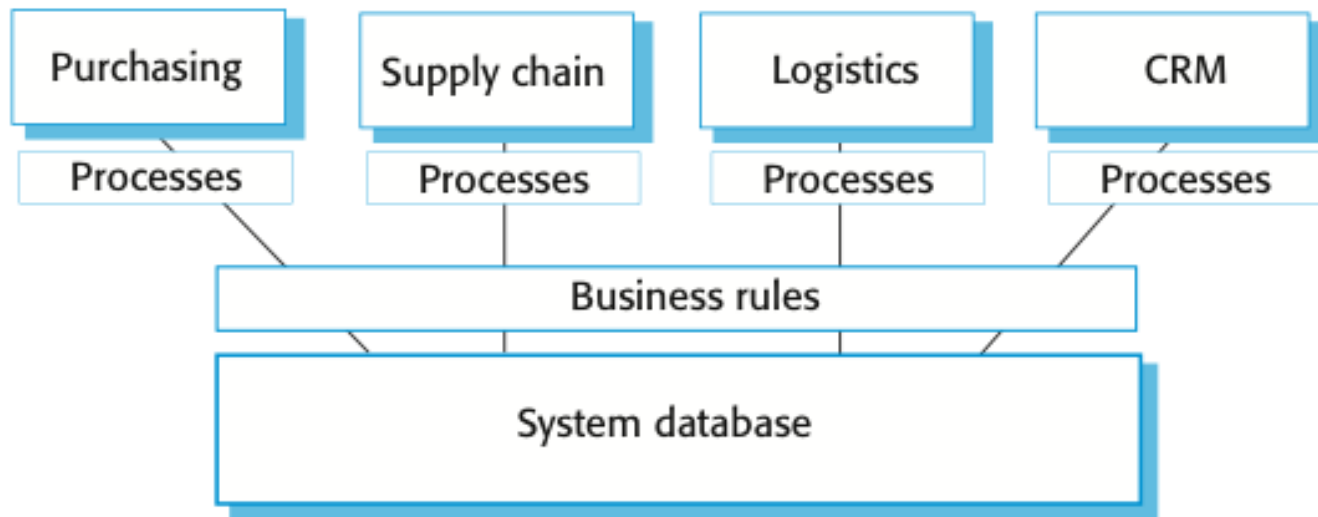
---



- ✧ An Enterprise Resource Planning (**ERP**) system is a **generic system** that supports common business processes such as ordering and invoicing, inventory management, and manufacturing scheduling, etc.
- ✧ These are very widely used in **large companies** - they represent probably the most common form of software reuse.
- ✧ The **generic core** is adapted by **including modules** and by **incorporating knowledge** of business processes and rules.

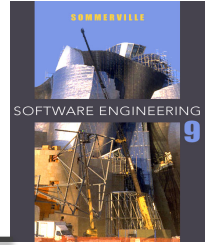
# The architecture of an ERP system

- A model of the overall architecture of an ERP system that supports a range of business functions is shown in the following:



# ERP architecture

---

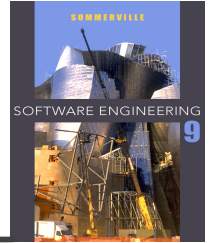


The **key features** of the above ERP architecture are:

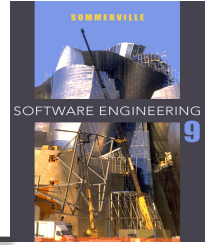
- ✧ (1) A number of **modules** to support different business functions.
- ✧ (2) A defined set of **business processes**, associated with each module, which relate to activities in that module.
- ✧ (3) A common **database** that maintains information about all related business functions.
- ✧ (4) A set of **business rules** that apply to all data in the database.

# ERP configuration

---



- ✧ Selecting the required functionality from the system.
- ✧ Establishing a data model that defines how the organization's data will be structured in the system database.
- ✧ Defining business rules that apply to that data.
- ✧ Defining the expected interactions with external systems.
- ✧ Designing the input forms and the output reports generated by the system.
- ✧ Designing new business processes that conform to the underlying process model supported by the system.
- ✧ Setting parameters that define how the system is deployed on its underlying platform.



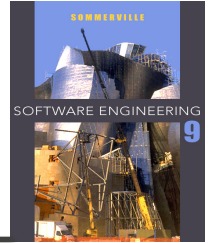
## 16.4.2 COTS integrated systems

---

- ✧ COTS-integrated systems are applications that include **two or more COTS** products and/or legacy application systems.
- ✧ You may use this approach when there is no single COTS system that meets all of your needs or when you wish to integrate a new COTS product with systems that you already use.

# Design choices

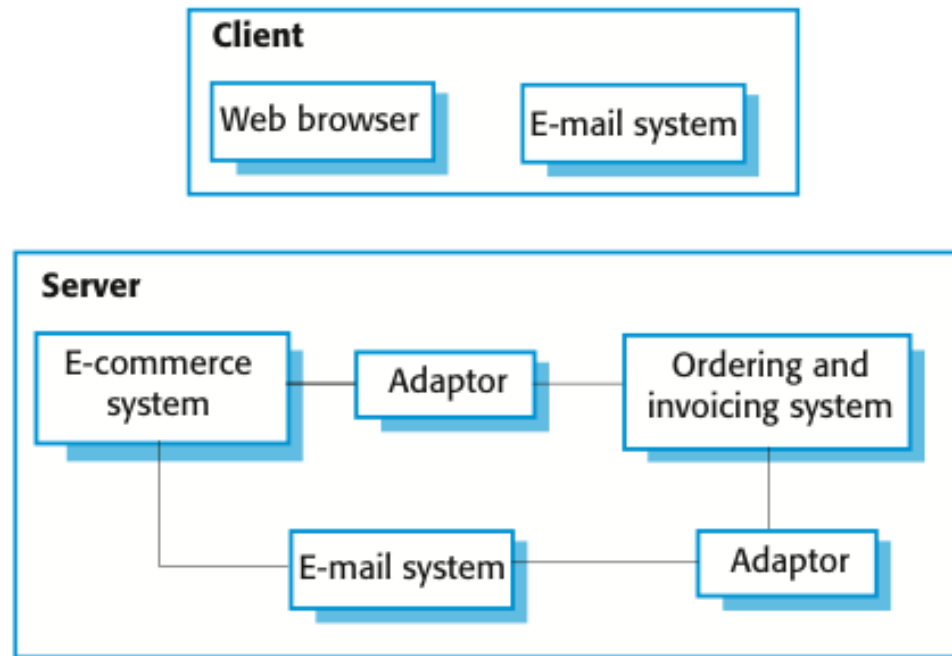
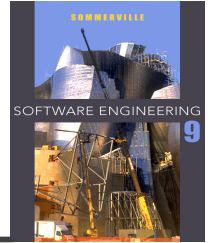
---



If possible, you should also **deny access to unused functionality** because this can interfere with normal system operation. Ariane 5 rocket, 1997)

The failure of **the first flight of the Ariane 5 rocket**(Nuseibeh, 1997) was a consequence of a **failure** in an inertial navigation system that was **reused from the Ariane 4 system**. However, the functionality that **failed** was **not** actually **required** in Ariane 5.

# A COTS-integrated procurement system

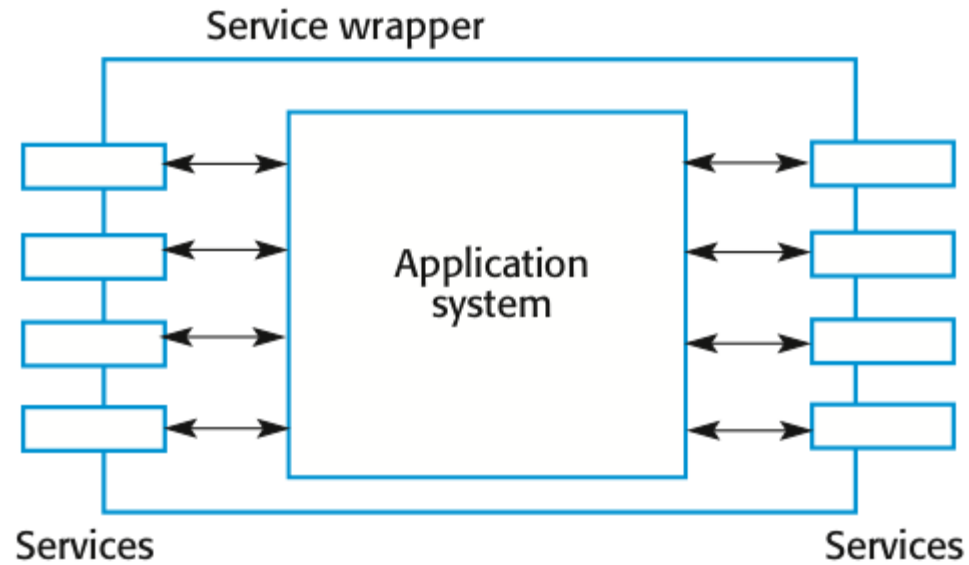
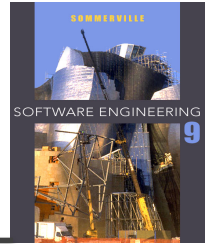




# Service-oriented COTS interfaces

- ✧ COTS integration can be simplified if a service-oriented approach is used.
- ✧ A service-oriented approach means allowing **access** to the application system's functionality **through a standard service interface**, with a service for each discrete unit of functionality.
- ✧ Some applications may offer a service interface but, sometimes, this **service interface** has to be implemented **by** the system **integrator**. You have to program a **wrapper** that hides the application and provides externally visible services(Figure in next slide).

# Application wrapping



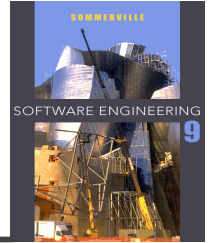
# COTS system integration problems

---

Four important COTS system integration problems:

- ✧ (1) **Lack of control over functionality and performance**
  - COTS systems may be less effective than they appear
- ✧ (2) **Problems with COTS system inter-operability**
  - Different COTS systems may make different assumptions that means integration is difficult
- ✧ (3) **No control over system evolution**
  - COTS vendors not system users control evolution
- ✧ (4) **Support from COTS vendors**
  - COTS vendors may not offer support over the lifetime of the product

# Key points



- ✧ **Software product lines** are related applications that are developed from a **common base**. This generic system is adapted to meet specific requirements for functionality, target platform or operational configuration.
- ✧ **COTS product reuse** is concerned with the reuse of large-scale, off-the-shelf systems. These provide a lot of functionality and their reuse can radically reduce costs and development time. Systems may be developed by configuring a single, generic COTS product or by integrating two or more COTS products.
- ✧ **Enterprise Resource Planning** systems are examples of large-scale COTS reuse. You create an instance of an ERP system by configuring a generic system with information about the customer's business processes and rules.
- ✧ **Potential problems** with COTS-based reuse include lack of control over **functionality** and **performance**, lack of control over system **evolution**, the need for support from external vendors and difficulties in ensuring that systems can inter-operate.