SSW-540: Fundamentals of Software Engineering

Software Reuse

Roberta (Robbie) Cohen, Ph.D.
Industry Professor
School of Systems and Enterprises

Software reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.
- Software engineering historically was more focused on original development but now recognises that to achieve better software, more quickly and at lower cost, it needs a design processes that are based on systematic software reuse.
- There has been a major switch to reuse-based development over the past several decades.
  - System reuse
  - Application reuse – intact or developing application families
  - Component reuse – sub-systems to single objects
  - Object and function reuse

Software Reuse

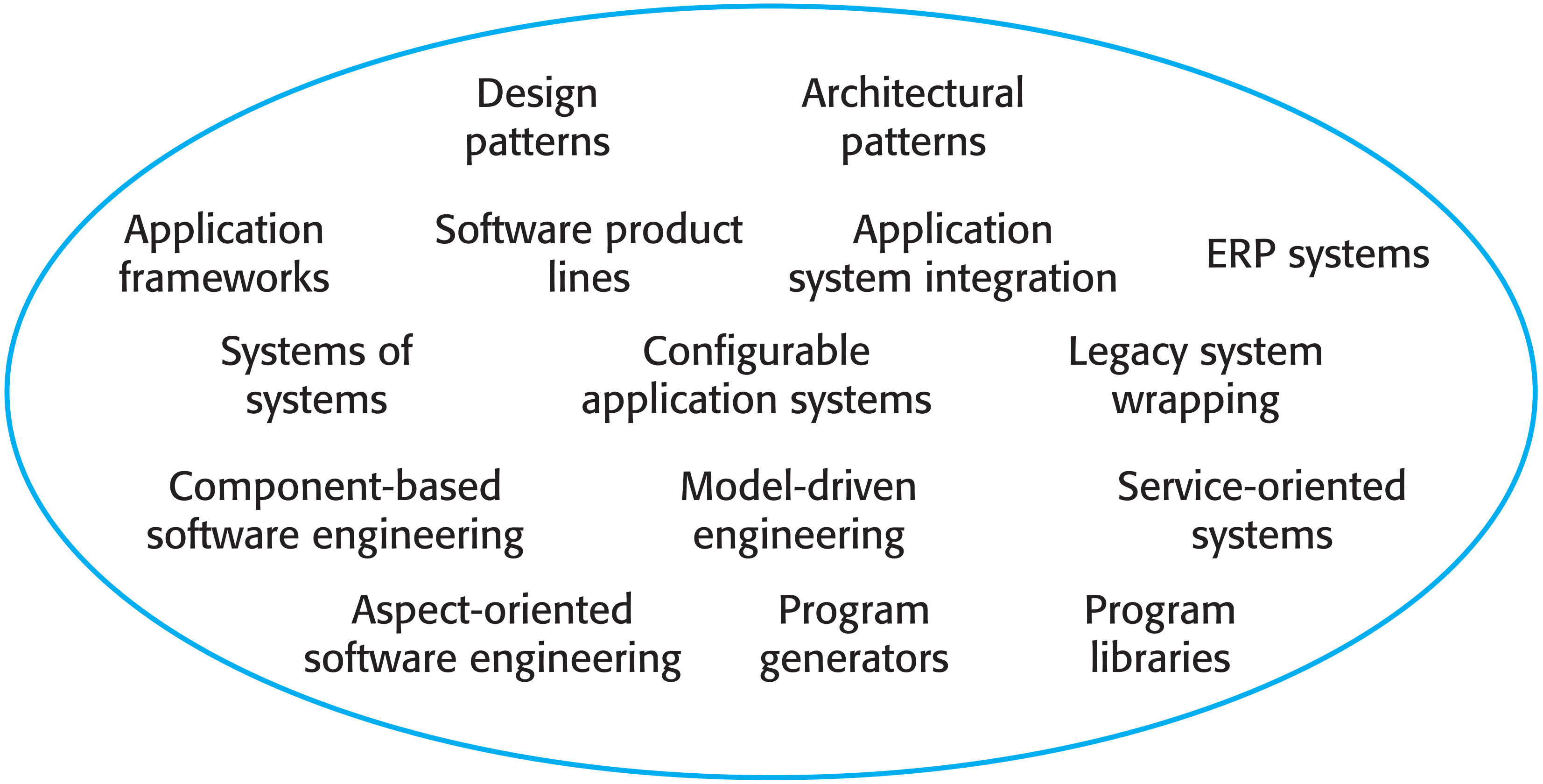
Benefits of

- Accelerated development
- Effective use of specialists
- Increased dependability
- Lower development costs
- Reduced process risk
- Standards compliance

Problems with

- Creating, maintaining and using a component library
- Finding, understanding and adapting reusable components
- Increased maintenance costs
- Lack of tool support
- Not-invented-here (NIH) syndrome

The reuse landscape



Far more than just reuse of software components--
Ranging simple functions to entire application systems!

Reuse planning factors: To re-use or not to re-use…

- 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.

One type of reuse: Application system reuse

- An application system product is a software system that can be adapted for different customers without changing the source code of the system.
- Application systems have generic features and so can be used/reused in different environments.
- Application system products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs.
  - COTS (Commercial Off The Shelf) systems are often examples of this reuse.
    - Configurable application systems are generic application systems that may be designed to support a particular business type, business activity or a complete business enterprise.
    - Domain-specific systems, such as systems to support a business function, provide functionality that is likely to be required by a range of potential users.

Application System Reuse

Benefits

- More rapid deployment of a reliable system may be possible.
- The functionality provided is visible, so it is easier to judge whether it is suitable.
- Some development risks are avoided by using existing software.
- Businesses can focus on their core activity.
- As operating platforms evolve, technology updates may be simplified as these are the responsibility of the COTS product vendor rather than the customer.

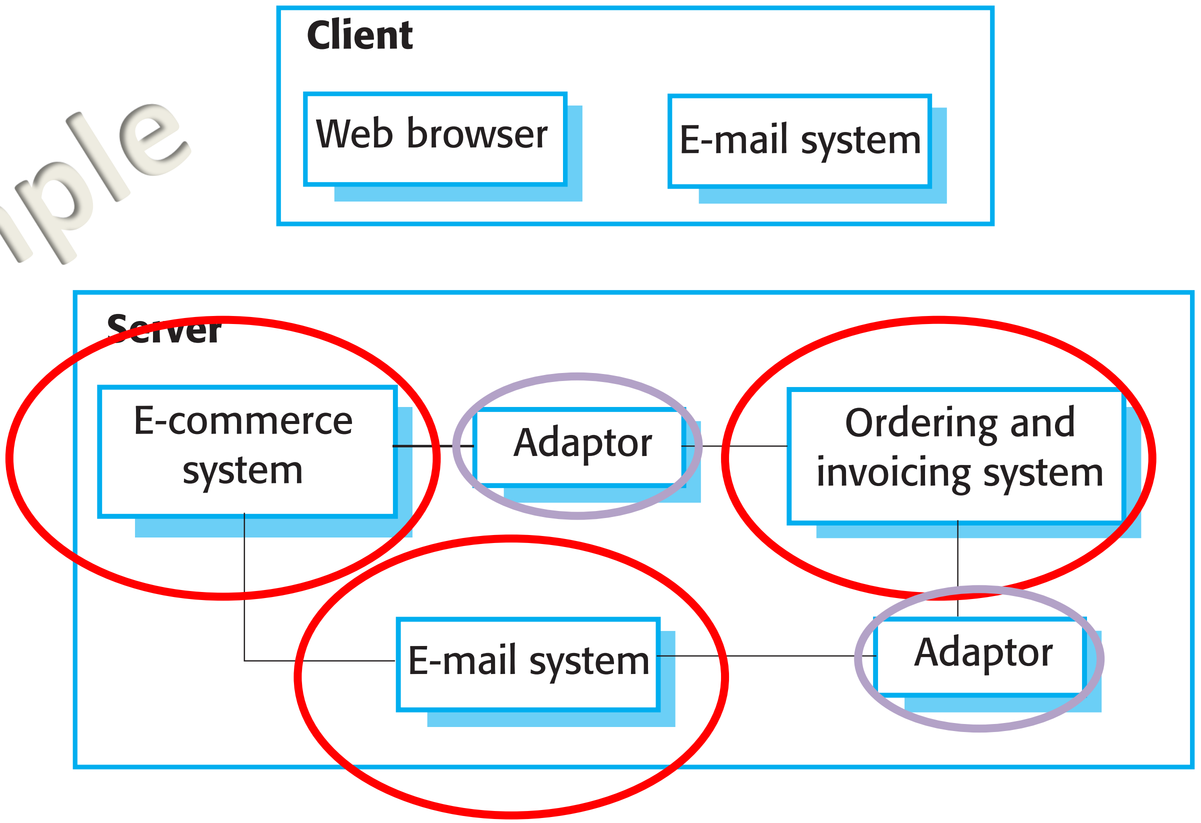
Problems

- Requirements may have to be adapted to reflect the functionality and mode of operation of the COTS product.
- The COTS product may be based on assumptions that are practically impossible to change.
- Many COTS products are not well documented so choosing the right system can be difficult.
- Organization may lack local expertise to support systems development.
- The COTS product vendor controls system support and evolution.

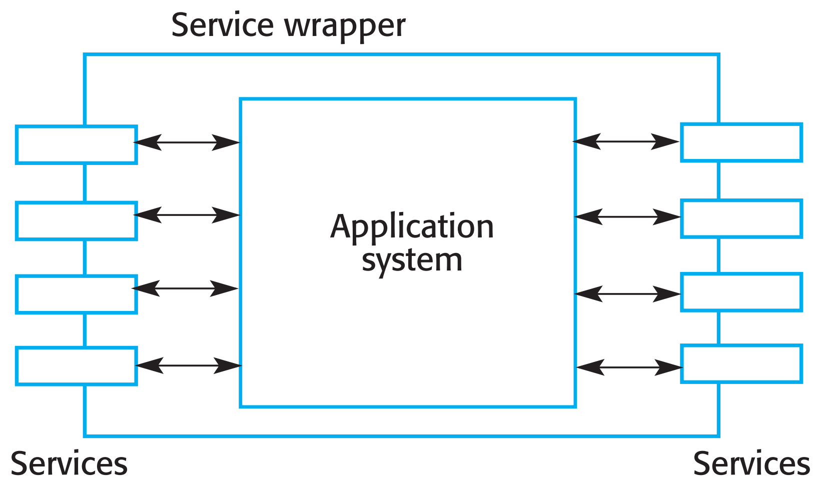
Integrated application systems

- Integrated application systems are applications that include two or more application system products and/or legacy application systems.
- You use this approach when there is no single application system that meets all of your needs or when you wish to integrate a new application system with systems that you already use.
- Design choices when integrating application systems:
  - **Which** individual application **systems** offer the most appropriate functionality?
    - Several products could be combined in various ways.
  - How will **data** be **exchanged**?
    - Different products will use unique data structures and formats.
  - **What features** of a product will actually be used?
    - Products may include more functionality than needed.

An integrated procurement system



Service-oriented interfaces



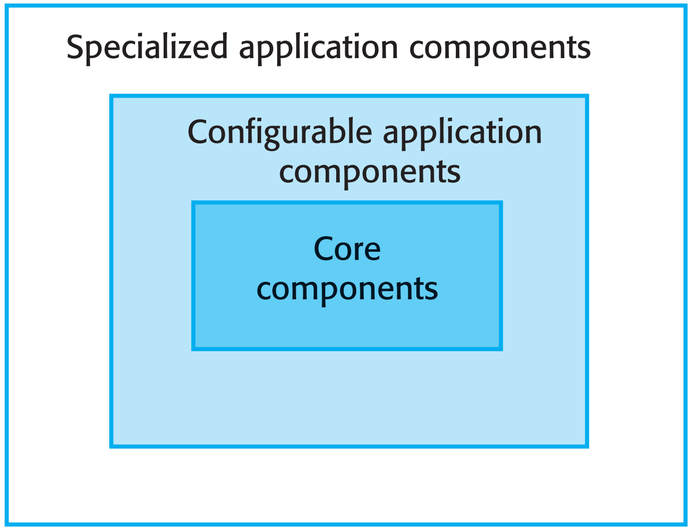
- Application system integration can be simplified if a service-oriented approach is used.
- A service-oriented approach means allowing access to the application system's functionality through a standard service interface with a service for each discrete unit of functionality.
- Some applications may offer a service interface but other times it must be implemented by a system integrator who programs a wrapper that hides the application and provides externally visible services.

Application system integration issues

- **Lack of control** over functionality and performance
  - Application systems may be less effective than they appear
- Problems with application system **inter-operability**
  - Different application systems may make different assumptions that means integration is difficult
- No control over system **evolution**
  - Application system vendors not system users control evolution
  - But you might benefit from new features you hadn't thought of
- **Support** from system vendors
  - Application system vendors may not offer support over the lifetime of the product

Another type of reuse: Software product lines

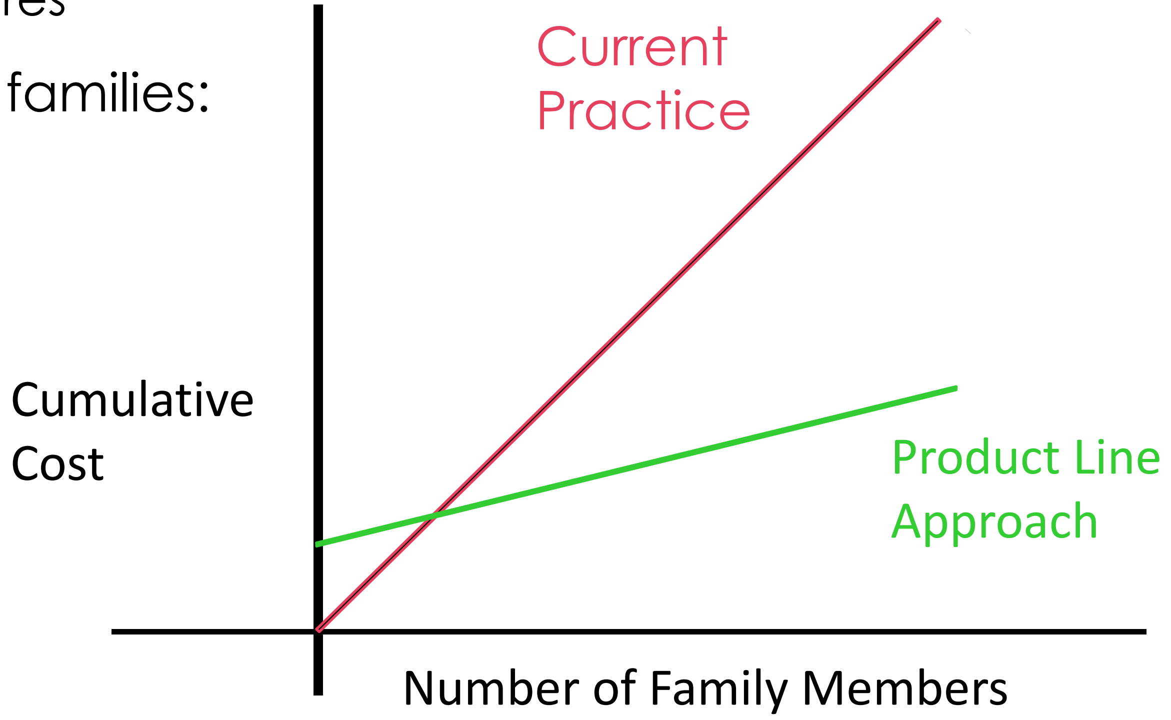
- Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context.
- A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect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


Specialized application components
Configurable application components
Core components

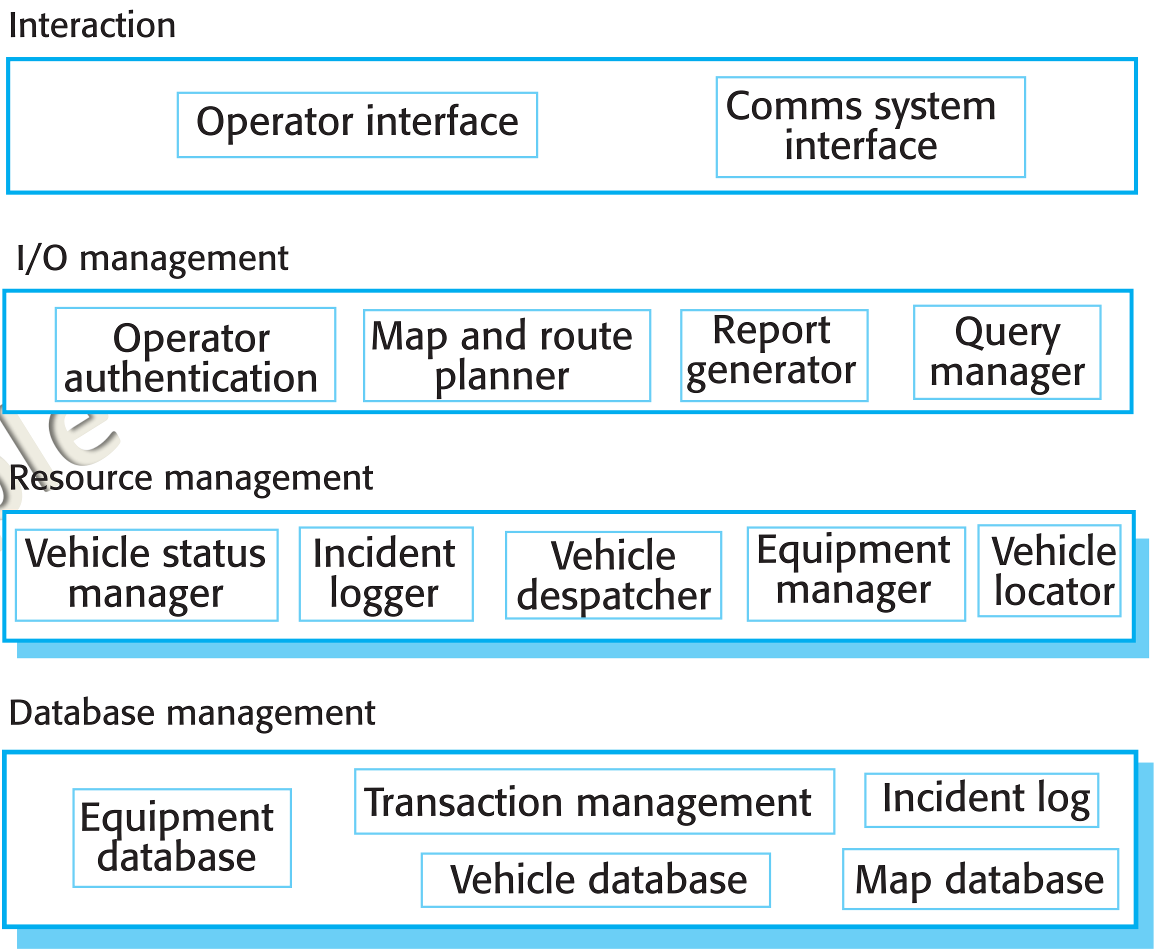
Base system for a software product line

Product line approach

- Software development organized to evolve a family of systems rather than a single system
  - Investment is in the product family "infrastructure" and application generators for "features"
- Economics of product families:



The product line architecture of a vehicle dispatcher



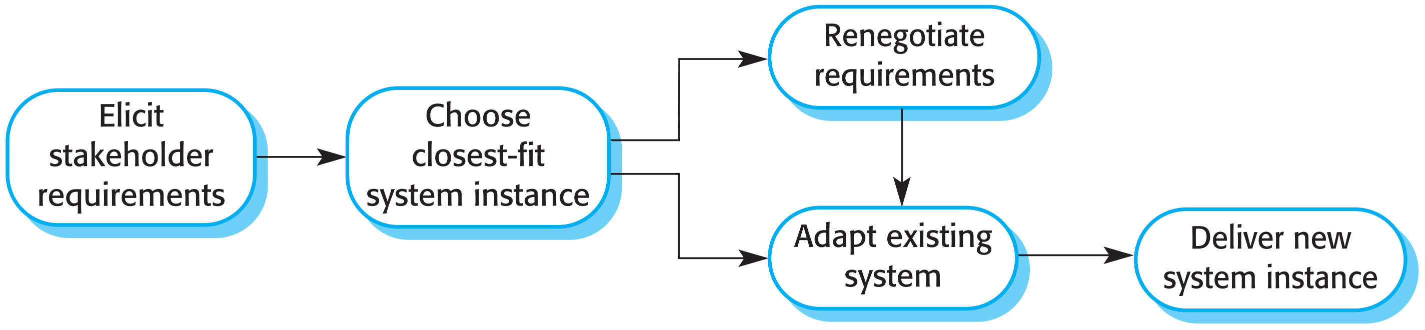
Vehicle dispatching example

- A specialized resource management system where the aim is to allocate resources (vehicles) to handle incidents.
- Adaptations 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 dispatch, managing vehicle status and incident logging;
  - The database includes equipment, vehicle and map databases.

Product line specialization

- Platform specialization
  - Different versions of the application are developed for different platforms.
- Environment specialization
  - Different versions of the application are created to handle different operating environments e.g. different types of communication equipment.
- Functional specialization
  - Different versions of the application are created for customers with different requirements.
- Process specialization
  - Different versions of the application are created to support different business processes.

Product line instance development

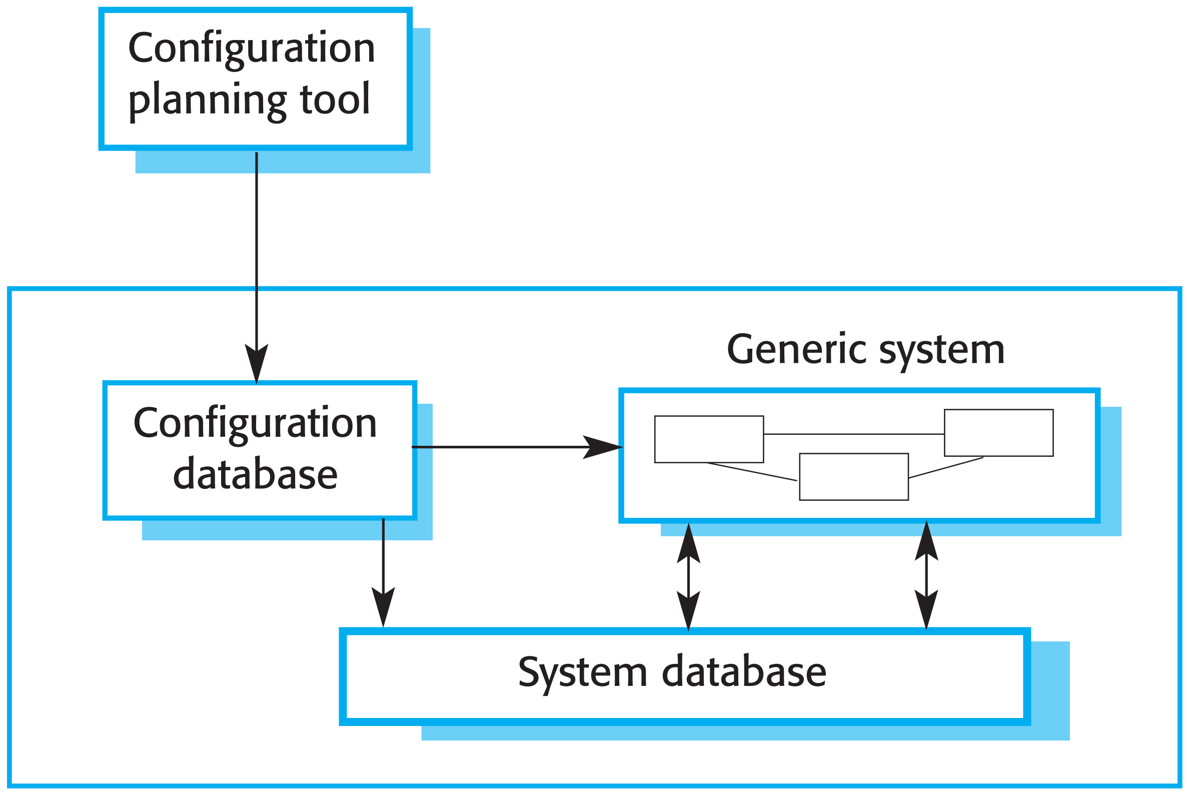

Elicit stakeholder requirements
Choose closest-fit system instance
Renegotiate requirements
Adapt existing system
Deliver new system instance

- Elicit stakeholder requirements - use existing family member as a prototype
- Choose closest-fit family member – find the family member that best meets the requirements
- Re-negotiate requirements - adapt requirements as necessary to capabilities of the software
- Adapt existing system - develop new modules and make changes for family member
- Deliver new family member - document key features for further member development

Product line configuration

- Design time configuration
  - The organization that is developing the software modifies a common product line core by developing, selecting or adapting components to create a new system for a customer.
- Deployment time configuration
  - A generic system is designed for configuration by a customer or consultants working with the customer. Knowledge of the customer's specific requirements and the system's operating environment is embedded in configuration data that are used by the generic system.
    - Component selection, where you select the modules in a system that provide the required functionality.
    - Workflow and rule definition, where you define workflows and validation rules that should apply to information entered by users or generated by the system.
    - Parameter definition, where you specify the values of specific system parameters that reflect the instance of the application that you are creating

Configuration at deployment



Reuse happens all levels of development

- System reuse
  - Commercial Off-the-Shelf systems
- Application reuse
  - COTS and custom software applications, used stand-alone or integrated together
- Object and function reuse
  - Usually the use of functions found in publicly- available libraries
- Component reuse
  - From sub-systems to single objects, software components **designed for** reuse

Component-based reuse

- Component-based software engineering (CBSE) is an approach to software development that relies on the reuse of entities called software components.
- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific.
- Components are more abstract than object classes and can be developed and used as stand-alone service providers.
- Essential aspects of CBSE:
  - Independent components specified by their interfaces.
  - Component standards to facilitate component integration.
  - Middleware that provides support for component inter-operability.
  - A development process that is geared to reuse.

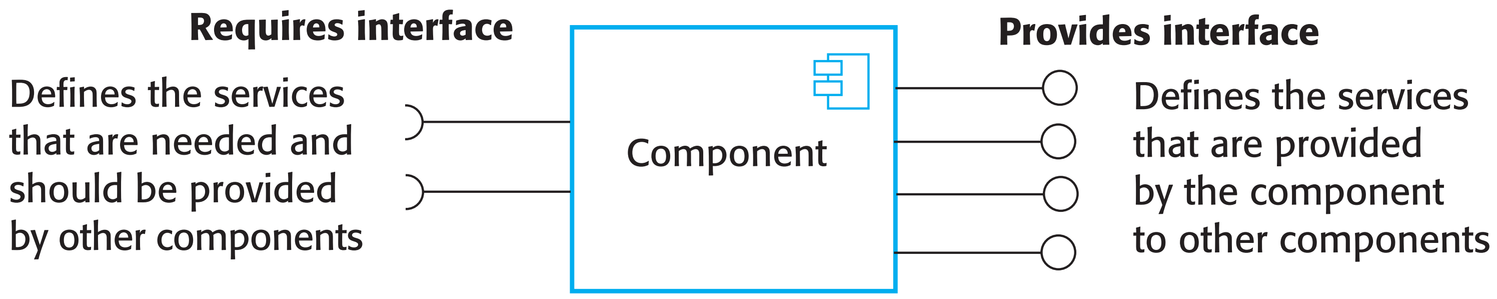
CBSE and design principles

- Apart from the benefits of reuse, CBSE is based on sound software engineering design principles:
  - Components are independent so do not interfere with each other;
  - Component implementations are hidden;
  - Communication is through well-defined interfaces;
  - One component can be replaced by another if its interface is maintained;
  - Component infrastructures offer a range of standard services.
- Standards need to be established so that components can communicate with each other and inter-operate.
  - Unfortunately, several competing component standards were established (Sun's *Enterprise Java Beans*, Microsoft's COM and *.NET*, CORBA's *CCM*)
  - These multiple standards have hindered the uptake of CBSE. Components developed using different approaches do <u>not</u> work together.

Independent software components

- Components provide a service without regard to where the component is executing (maybe the cloud!) or its programming language
  - A component is an independent executable entity that can be made up of one or more executable objects;
  - The component interface is published and all interactions are through the published interface;
- Component characteristics:
  - Composable – all external interactions use publicly defined interfaces
  - Deployable – self-contained; stand-alone operation (on a model platform)
  - Documented – function and interface syntax & semantics fully specified
  - Independent- composed or deployed without needing other components
  - Standardized – conforms to a standard component model

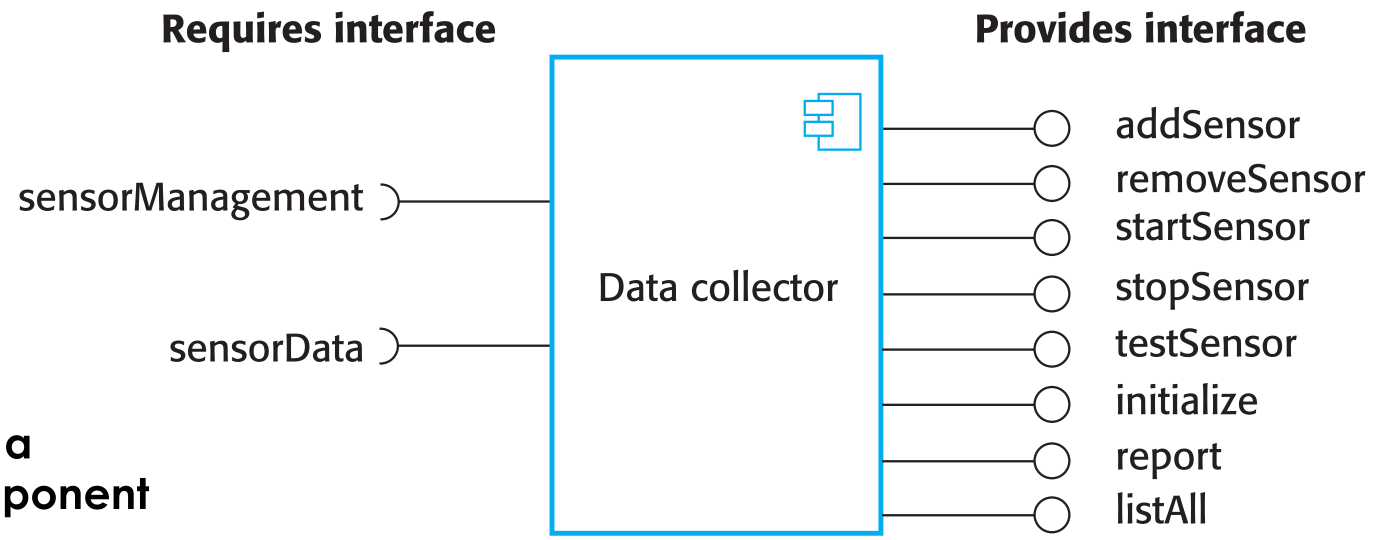
Component interfaces


Requires interface
Defines the services that are needed and should be provided by other components
Component
Provides interface
Defines the services that are provided by the component to other components

- Provides interface
  - This interface, essentially, is the component API. It defines the methods that can be called by a user of the component.
- Requires interface
  - This does not compromise the independence or deployability of a component because the ‘requires’ interface does not define how these services should be provided.

Component access

- Components are accessed using remote procedure calls (RPCs).
- Each component has a unique identifier (usually a URL) and can be referenced from any networked computer.
- Therefore it can be called in a similar way as a procedure or method running on a local computer.



Example modeling a data collector component

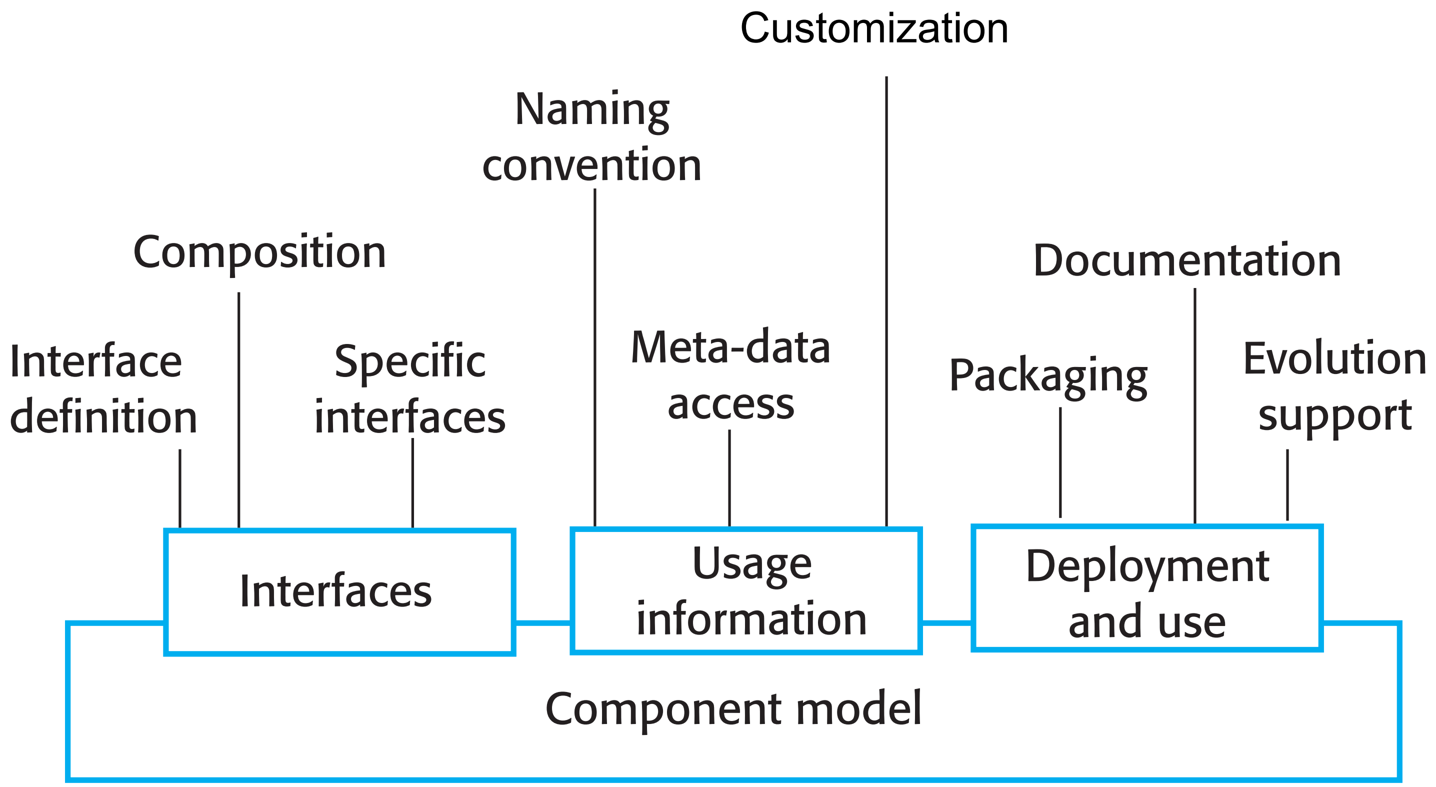
Service-oriented software engineering

- An executable service is a type of independent component. It has a ‘provides’ interface but not a ‘requires’ interface.
    - The services are made available through an interface and all component interactions occur through that interface.
    - The component interface is expressed in terms of parameterized operations and its internal state is never exposed.
- From the outset, services have been based around standards so there are no problems in communicating between services offered by different vendors.
- System performance may be slower with services but this approach is replacing CBSE in many systems.

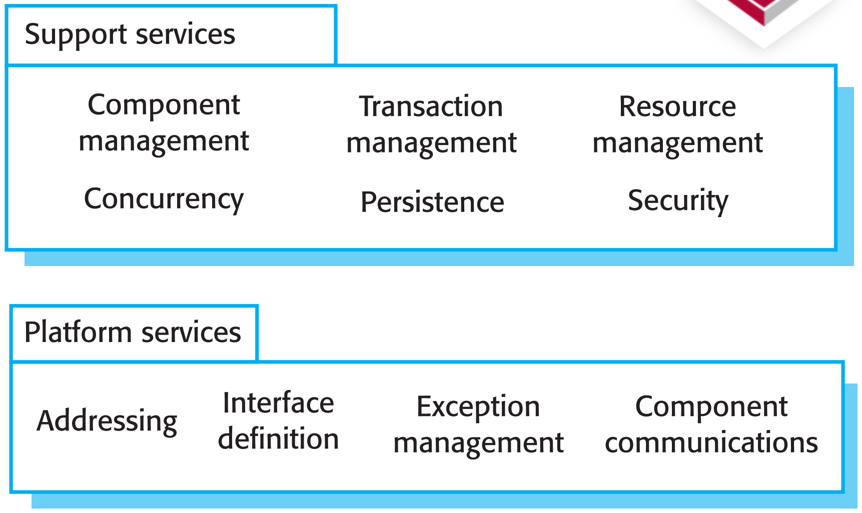
Component models

- A component model is a definition of standards for component implementation, documentation and deployment.
- Examples of (competing) component models
  - EJB model (Enterprise Java Beans)
  - COM+ model (.NET model)
  - Corba Component Model
- The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.

Basic elements of a component model

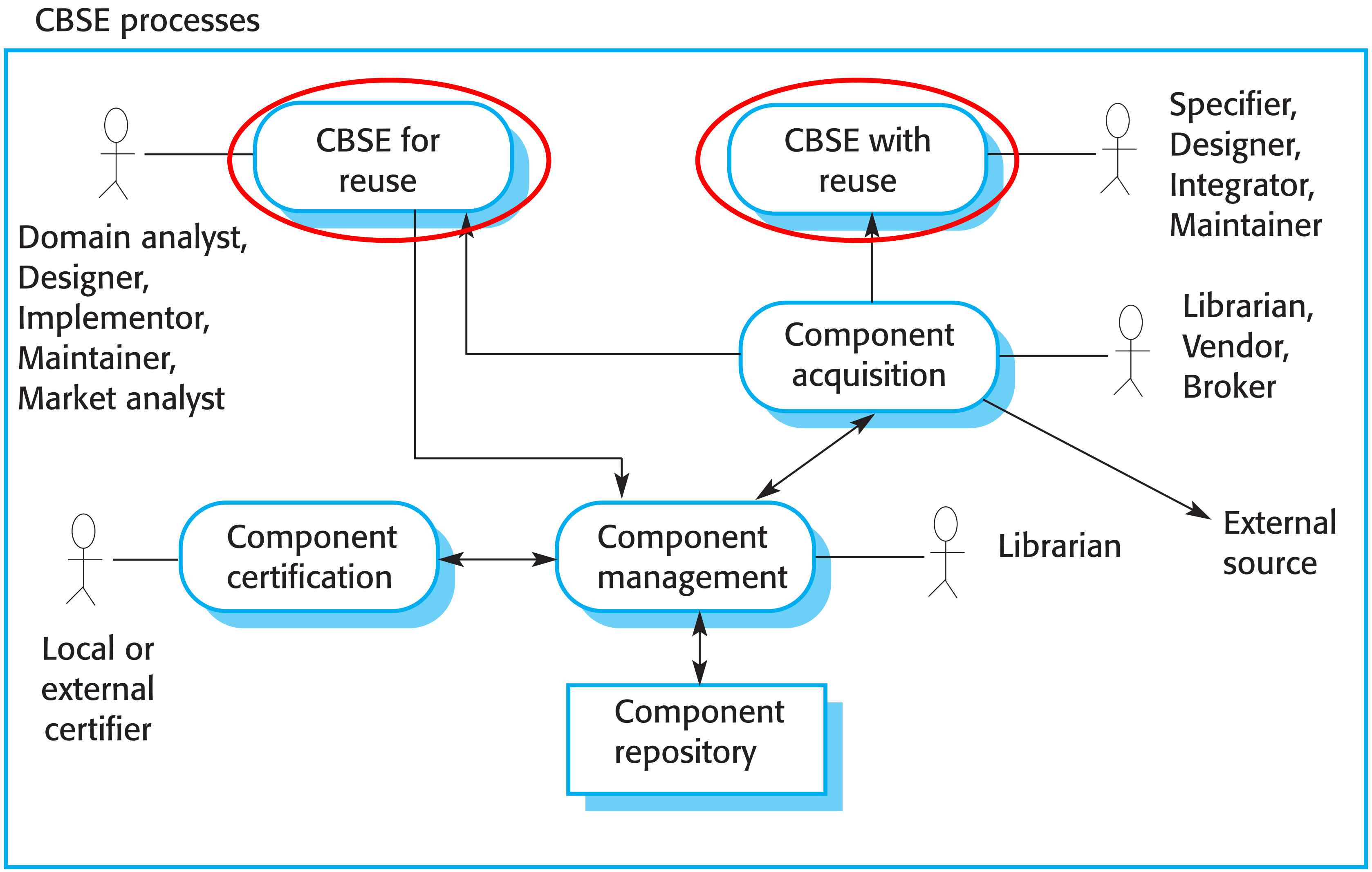


Middleware support



- Component models are the basis for middleware that provides support for executing components.
- Component model implementations provide:
  - Platform services that allow components written according to the model to communicate;
  - Support services that are application-independent services used by different components.
- To use services provided by a model, components are deployed in a container. This is a set of interfaces used to access the service implementations.

CBSE Processes


CBSE processes
CBSE for reuse
CBSE with reuse
Specifier, Designer, Integrator, Maintainer
Domain analyst, Designer, Implementor, Maintainer, Market analyst
Component acquisition
Librarian, Vendor, Broker
External source
Component certification
Component management
Librarian
Local or external certifier
Component repository

CBSE for reuse

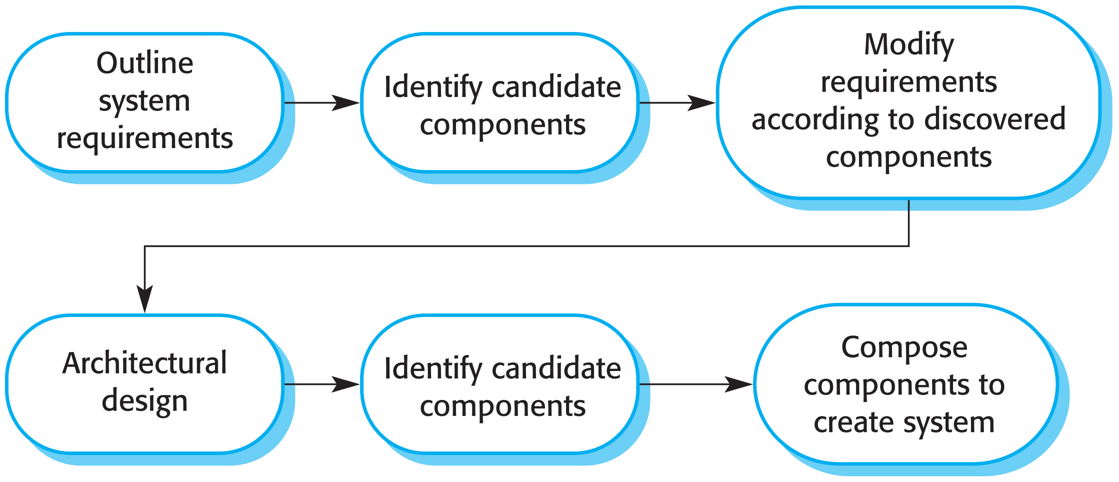
- CBSE for reuse focuses on component development.
- Components developed for a specific application usually have to be generalized to make them reusable.
- Component reusability
  - Should reflect stable domain abstractions (e.g., a business object);
  - Should hide state representation;
  - Should be as independent as possible;
  - Should publish exceptions through the component interface.
- There is a trade-off between reusability and usability
  - The more general the interface, the greater the reusability but it is then more complex and hence less usable.

Changes for reusability

- Remove application-specific methods.
- Change names to make them general.
- Add methods to broaden coverage.
- Add a configuration interface for component adaptation.
- Integrate required components to reduce dependencies.
- Make exception handling consistent.
  - Components should not handle exceptions themselves (they should define and publish exceptions as part of their interface), because each application will have its own requirements for exception handling.
  - In practice, however, there are two problems with this:
    - Publishing all exceptions leads to bloated interfaces that are harder to understand.
    - The operation of the component may depend on local exception handling, and changing this may have serious implications for the functionality of the component.

Issues with development for reuse

- Generic components may be less space-efficient and may have longer execution times than their specific equivalents.
- The development cost of reusable components may be higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organizational rather than a project cost.
- Component management must classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component and keeping track of different component versions. Also not a project cost.
- A company with a reuse program may carry out some form of component certification before the component is made available for reuse.
  - Certification means that someone apart from the developer checks the quality of the component.


Outline system requirements
Identify candidate components
Modify requirements according to discovered components
Architectural design
Identify candidate components
Compose components to create system

CBSE with reuse

- CBSE with reuse process has to find and integrate reusable components.
- When reusing components, it is essential to make trade-offs between ideal requirements and the services actually provided by available components.
- This involves:
  - Developing outline requirements;
  - Searching for components then modifying requirements according to available functionality.
  - Searching again to find if there are better components that meet the revised requirements.
  - Composing components to create the system.

Component reuse issues

- Trust. You need to be able to trust the supplier of a component. At best, an untrusted component may not operate as advertised; at worst, it can breach your security.
- Requirements. Different groups of components will satisfy different requirements.
- Validation.
  - The component specification may not be detailed enough to allow comprehensive tests to be developed.
  - Components may have unwanted functionality. How can you test this will not interfere with your application?
  - As well as testing that a component for reuse does what you require, you may also have to check that the component does not include any malicious code.

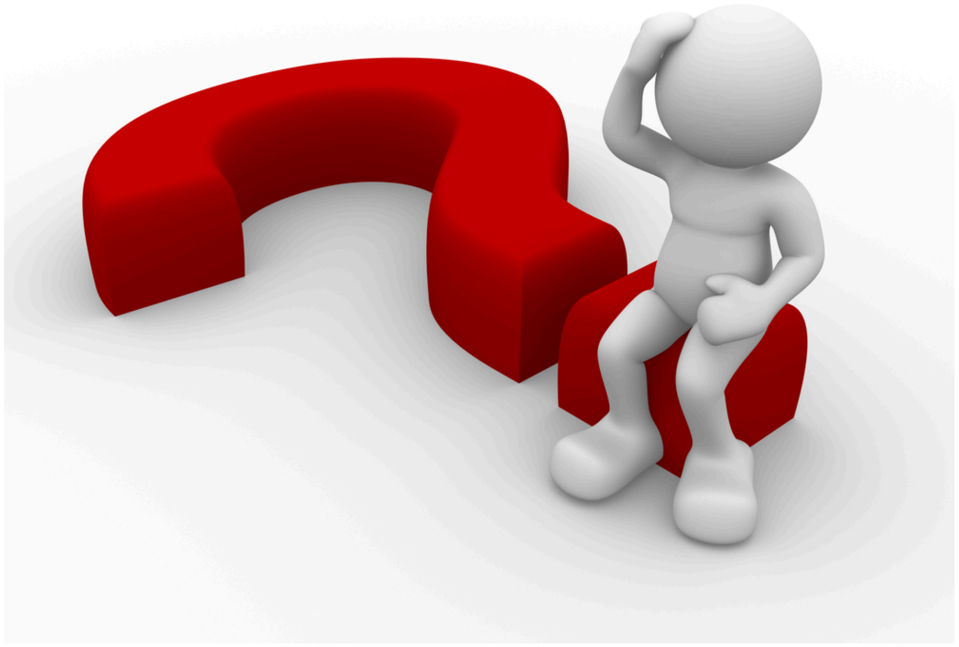
Composition trade-offs

- When composing components, you may find conflicts between functional and non-functional requirements, and conflicts between the need for rapid delivery and system evolution.
- You need to make decisions such as:
    - What composition of components is effective for delivering the functional requirements?
    - What composition of components allows for future change?
    - What will be the emergent properties of the composed system?

Ariane launch failure

- In 1996, the 1st test flight of the Ariane 5 rocket ended in disaster when the rocket went out of control 37 seconds after take off.
- The problem was due to a reused component from a previous version of the rocket (the Inertial Navigation System) that failed because assumptions made when that component was developed did not hold for Ariane 5.
- The functionality that failed in this component was not needed in Ariane 5.
- Video of the failure available on Canvas (less than 2 minutes long).

Questions?



Approaches that support software reuse

Approach	Description
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Application system integration	Two or more application systems are integrated to provide extended functionality
Architectural patterns	Standard software architectures that support common types of application system are used as the basis of applications. Described in Chapters 6, 11 and 17.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled. Described in web chapter 31.
Component-based software engineering	Systems are developed by integrating components (collections of objects) that conform to component-model standards. Described in Chapter 16.

Approaches that support software reuse

Approach	Description
Configurable application systems	Domain-specific systems are designed so that they can be configured to the needs of specific system customers.
Design patterns	Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions. Described in Chapter 7.
ERP systems	Large-scale systems that encapsulate generic business functionality and rules are configured for an organization.
Legacy system wrapping	Legacy systems (Chapter 9) are ‘wrapped’ by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Model-driven engineering	Software is represented as domain models and implementation independent models and code is generated from these models. Described in Chapter 5.

Approaches that support software reuse

Approach	Description
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.
Program libraries	Class and function libraries that implement commonly used abstractions are available for reuse.
Service-oriented systems	Systems are developed by linking shared services, which may be externally provided. Described in Chapter 18.
Software product lines	An application type is generalized around a common architecture so that it can be adapted for different customers.
Systems of systems	Two or more distributed systems are integrated to create a new system. Described in Chapter 20.