SSW-540: Fundamentals of Software Engineering

Software Reuse and Component-Based Software Engineering

Dr. Richard Ens
School of Systems and Enterprises

A few favorite things (for Python)

Python String Format Cookbook - - String formatting cheat sheet
- http://mkaz.tech/python-string-format.html

Debuggex - Create and debug regular expressions
- http://www.debuggex.com/?flavor=python

PyPI – 79K Python packages found at http://pypi.python.org/pypi
- NumPy/SciPy – scietific computing
- SciKit Learn – machine learning
- SQLAlchemy – database management
- Django – web framework
- Requests – HTML/web scraping
- Simplejson – JSON encoder/decoder
- Pip – Python package management

PyCharm – IDE for hardcore Python developers

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

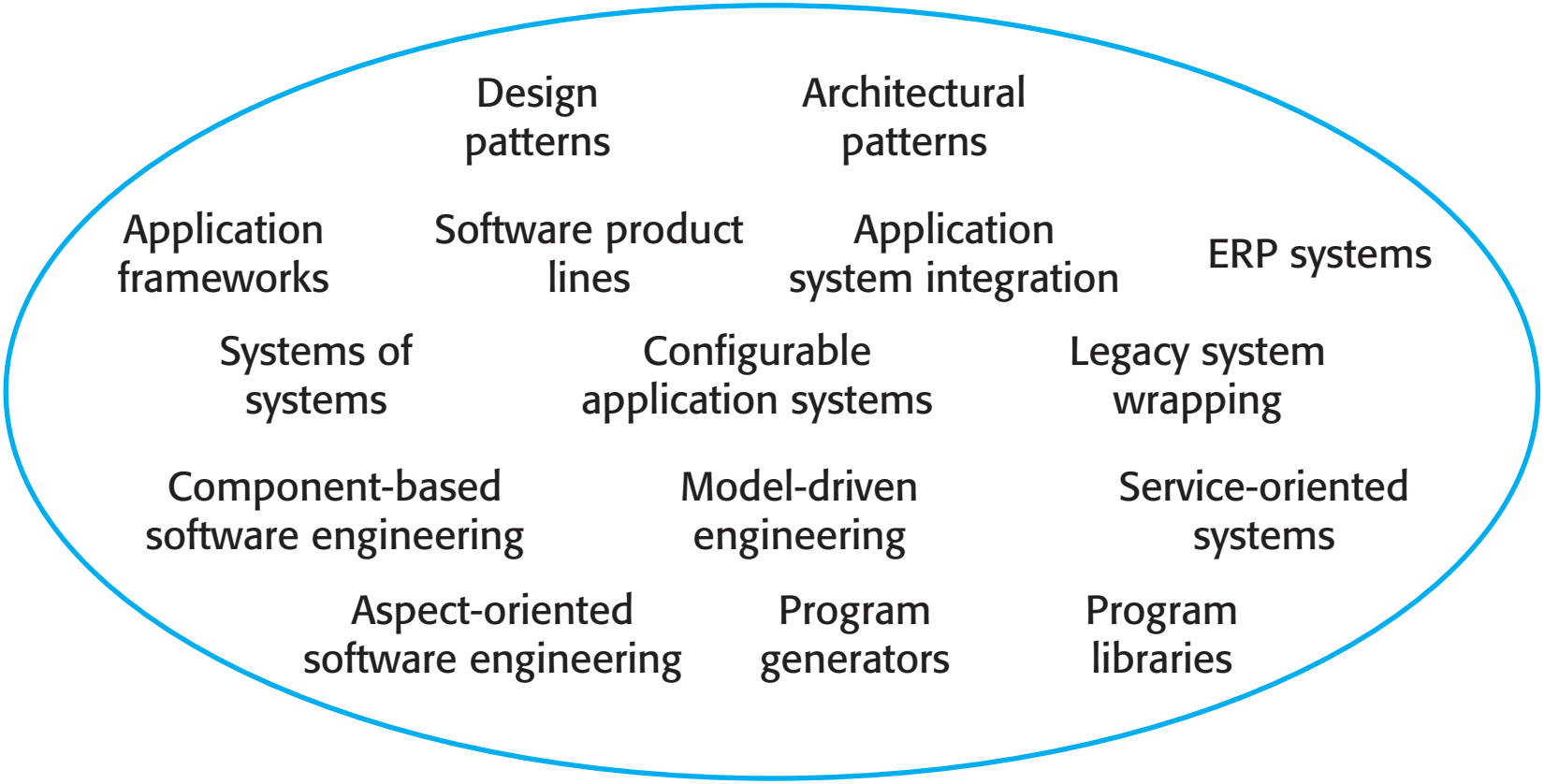
Benefits of

- 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



See Sommerville text for definitions of these.

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.

Application frameworks

"..an integrated set of software artefacts (such as classes, objects and components) that collaborate to provide a reusable architecture for a family of related applications."

- 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.
- The most widely used application frameworks are Web Application Frameworks (WAFs) and middleware integration frameworks such as Microsoft's .NET and Enterprise Java Beans (EJB)*

* EJB originated at IBM and was later developed by Sun Microsystems and by Java open source projects.

Extending frameworks

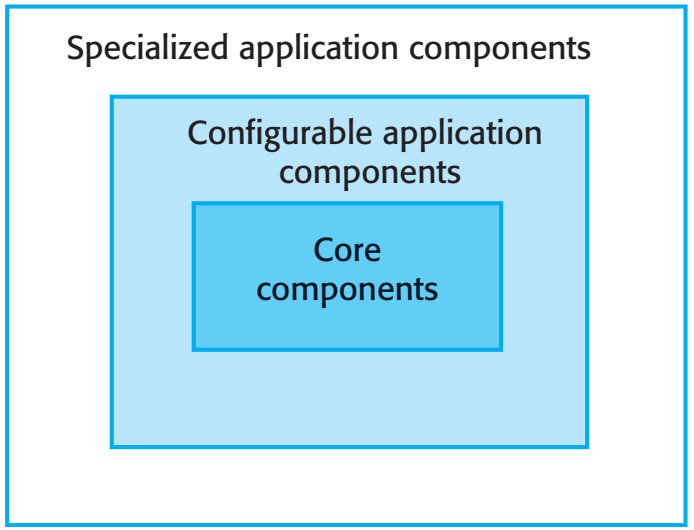
- Frameworks are generic and are extended to create a more specific application or sub-system. They provide a skeleton architecture for the system.
- Extending the framework involves
  - Adding concrete classes that inherit operations from abstract classes in the framework; these customize the framework to the domain
  - Adding methods that are called in response to events that are recognised by the framework.
- App frameworks save developer time by avoiding the rewriting of functionality, and they provide consistency across application providers that improves user acceptance
- The problem with frameworks is their complexity, which means that it takes a long time to use them effectively.

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.
- Web application frameworks
  - Support the construction of dynamic websites, usually based on the Model-View-Controller (MVC) pattern

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.



Base system for a software product line

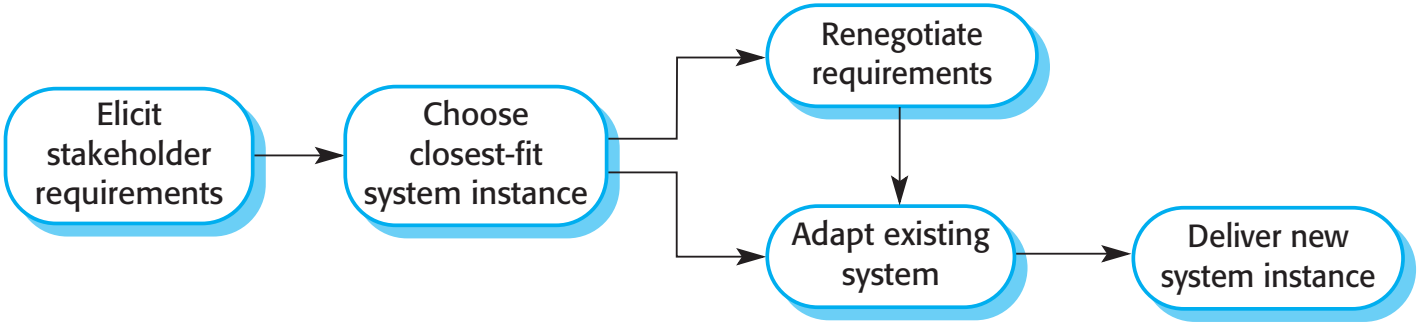
Application frameworks and product lines

- 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 specialization

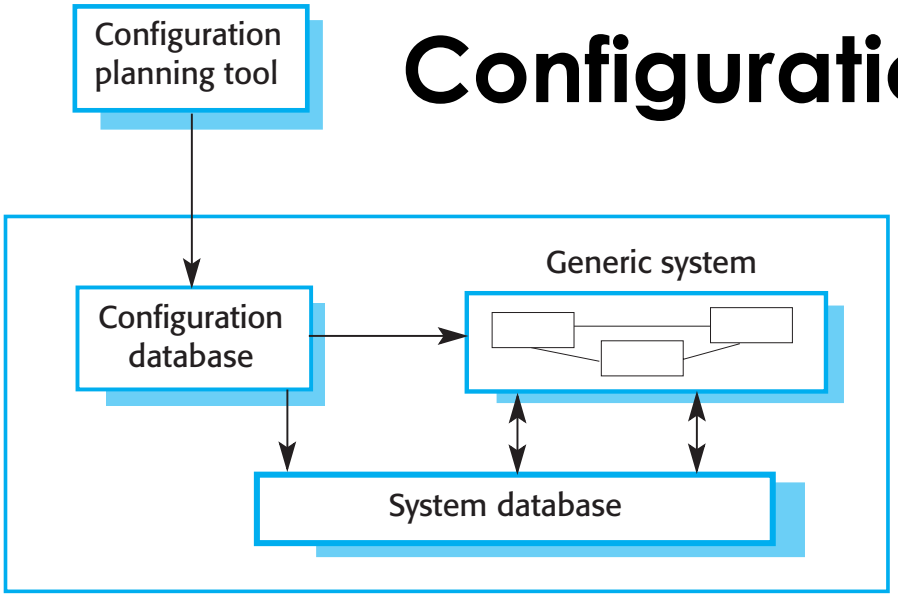
- Platform specialization
  - Different versions of the application are developed for different platforms.
- Environment specialization
  - Different versions of the application are created to handle different operating environments e.g. different types of communication equipment.
- Functional specialization
  - 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 instance development



- 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

Configuration at deployment



- Component selection, where you select the modules in a system that provide the required functionality.
- 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.
- Parameter definition, where you specify the values of specific system parameters that reflect the instance of the application that you are creating.

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.

COTS-solution and COTS-integrated systems

Configurable application systems	Application system integration
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

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 or not it is likely to be 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.

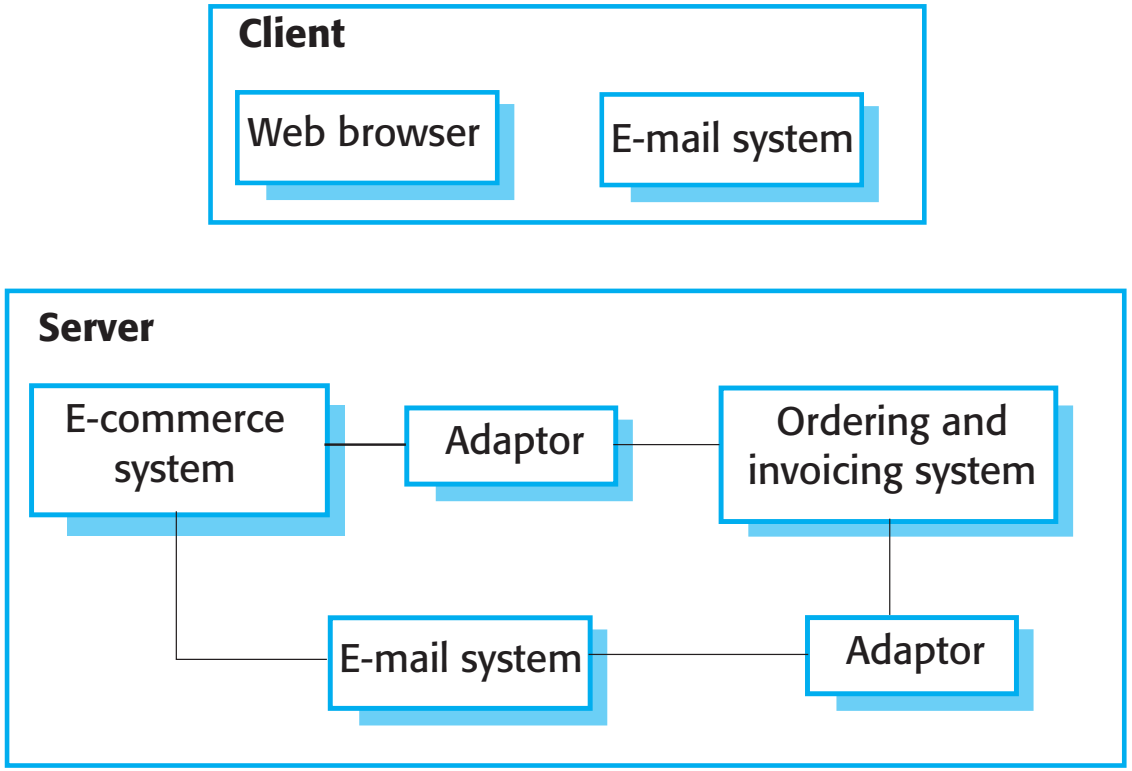
Problems

- 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.

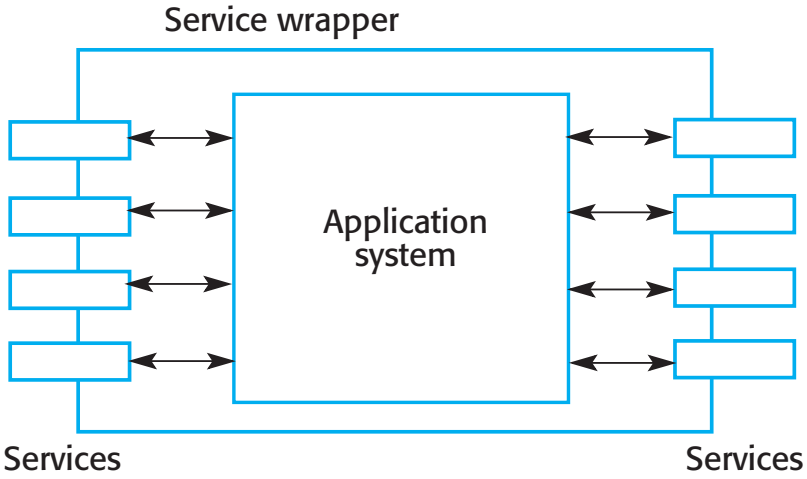
Integrated application systems

- Integrated application systems are applications that include two or more application system products and/or legacy application systems.
- You may 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, 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.

Application system integration problems

- 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

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 considered to be stand-alone service providers. They can exist as stand-alone entities. E.g., Docker
- 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 components 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 not work together.

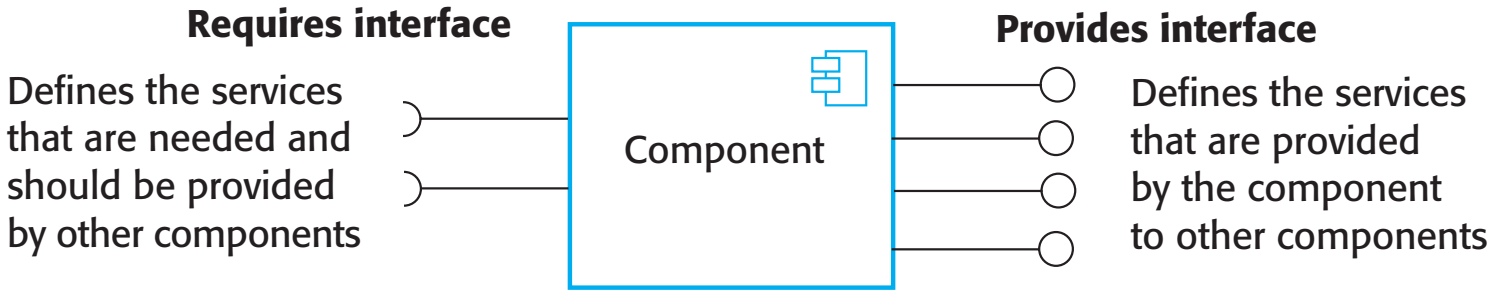
Software components

- 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

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.
- We will discuss service-oriented software engineering in detail next week.

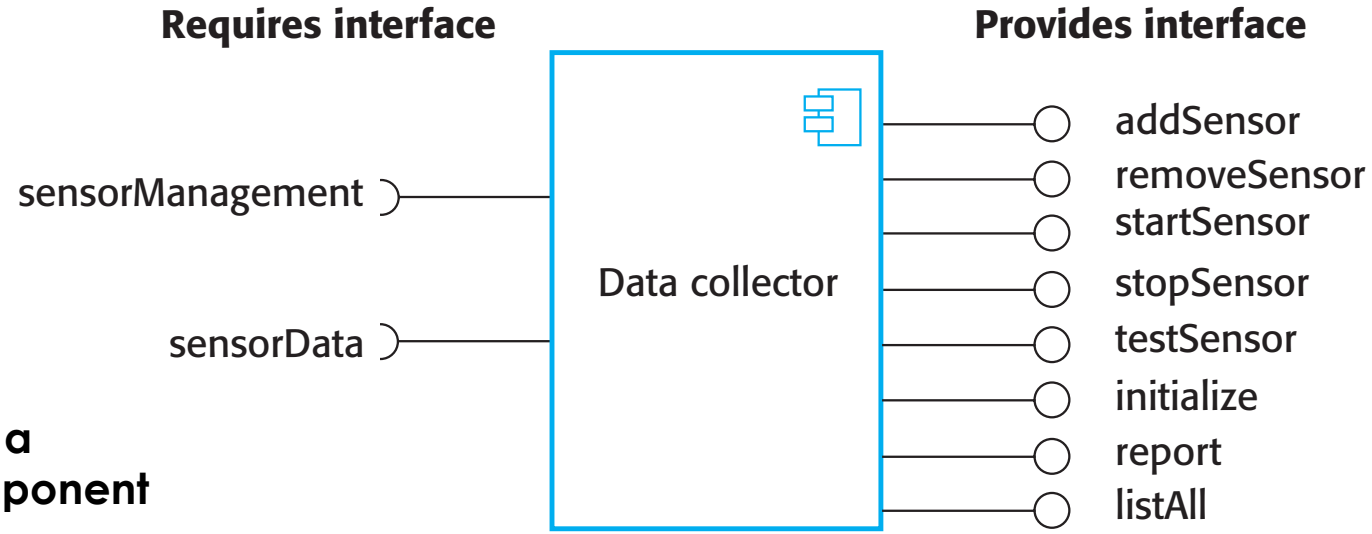
Component interfaces



- 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.

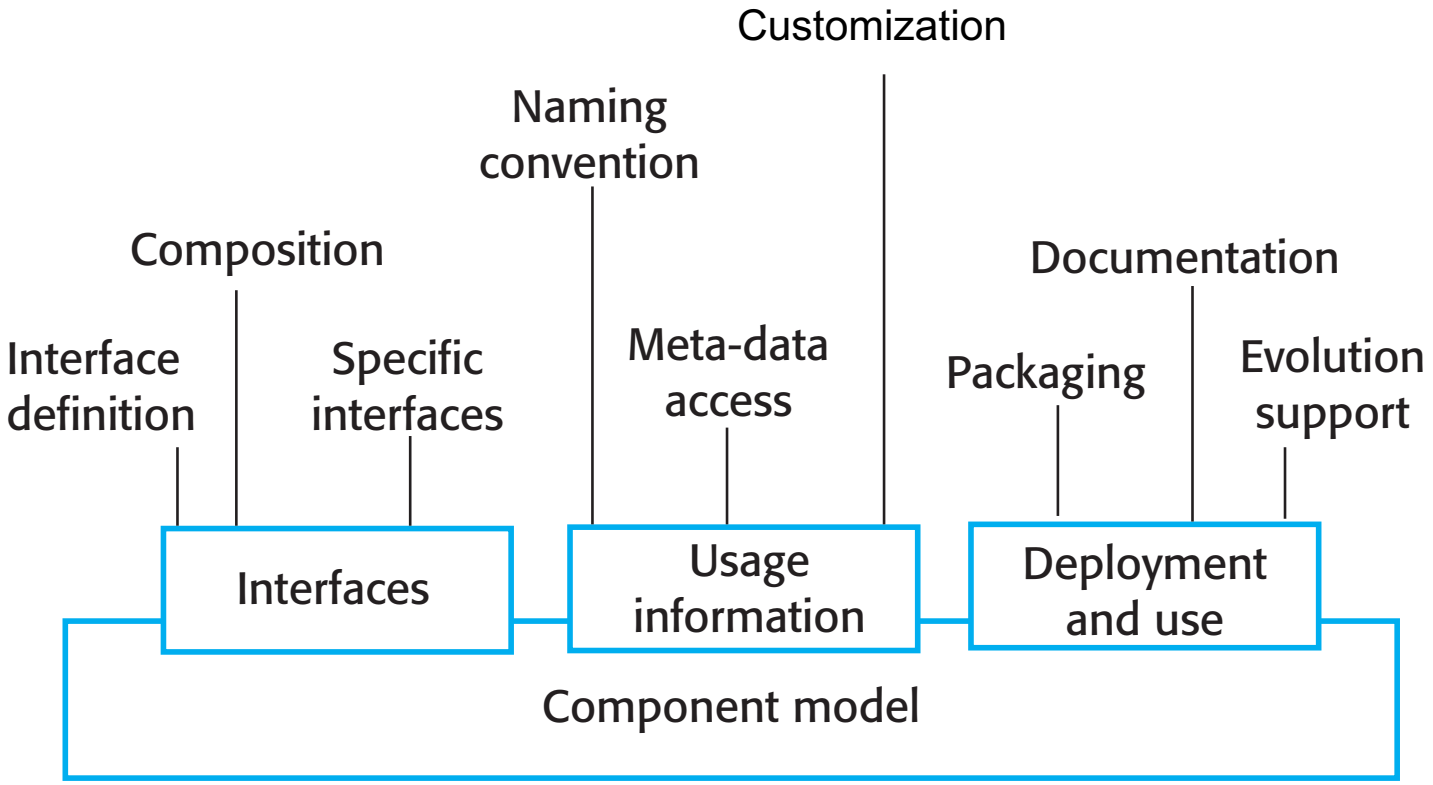


Example modeling a data collector component

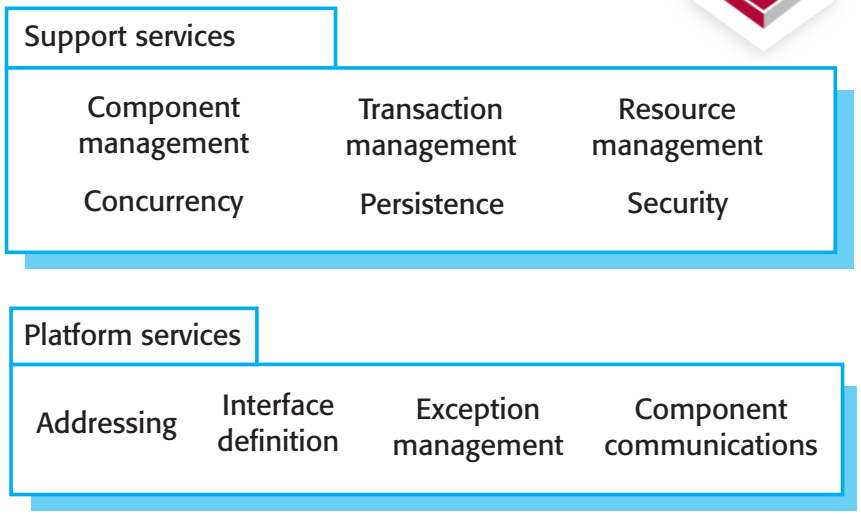
Component models

- 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

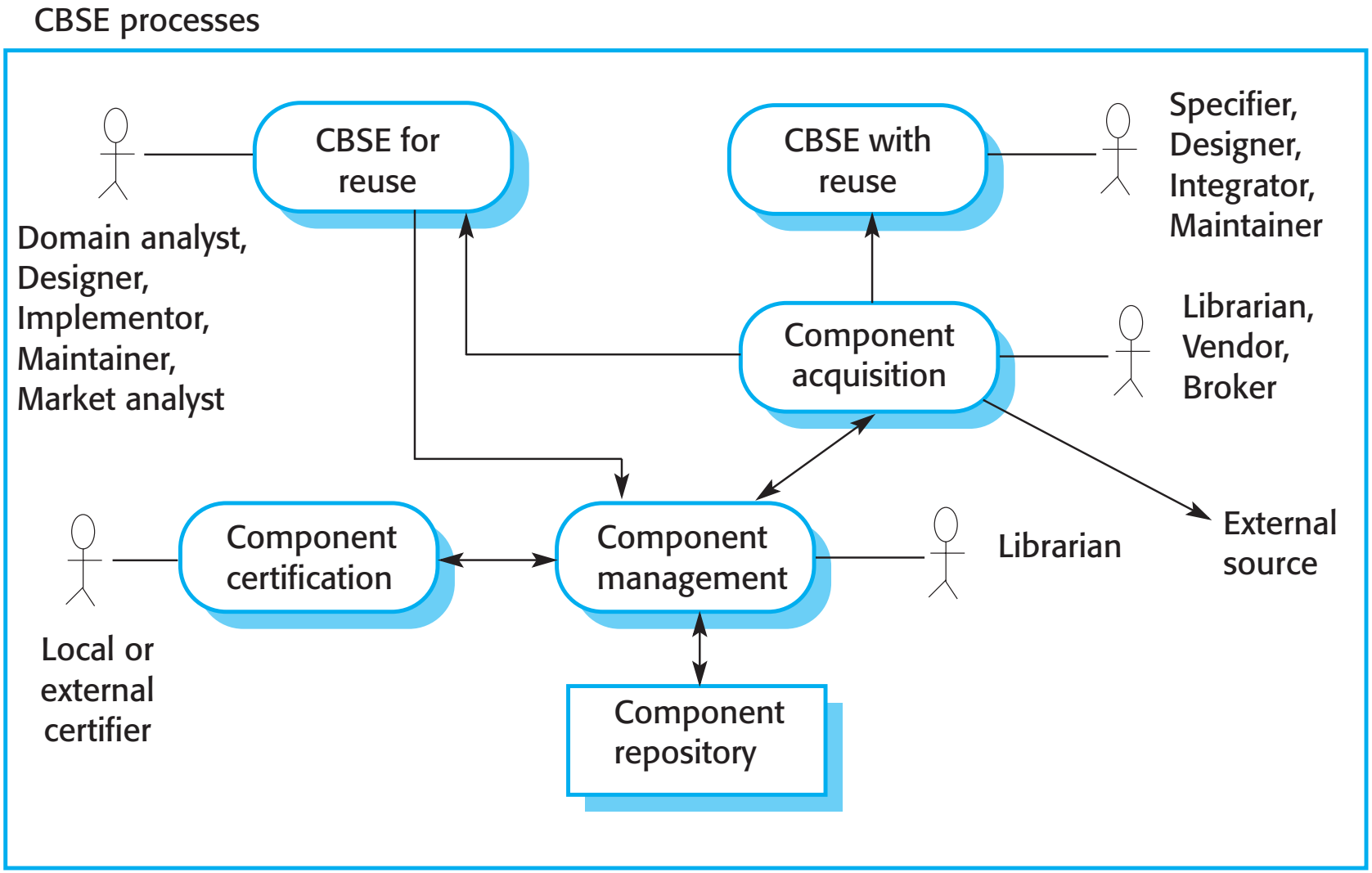


- Component models are the basis for middleware that provides support for executing components.
- Component model implementations provide:
  - Platform services that allow components written according to the model to communicate;
  - Support services that are application-independent services used by 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 are software processes that support component-based software engineering.
  - They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components.
- Development **for reuse**
  - This process is concerned with developing (from scratch) components or services that will be reused in other applications. It also may involve generalizing existing components.
- Development **with reuse**
  - This process is the process of developing new applications using existing components and services.

CBSE Processes graphically



CBSE for reuse

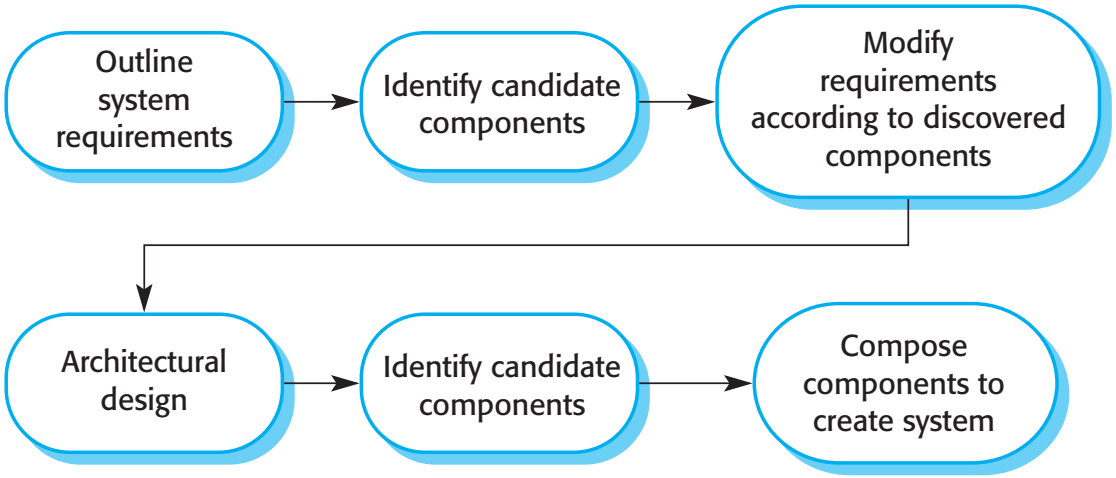
- CBSE for reuse focuses on component development.
- Components developed for a specific application usually have to be generalized to make them reusable.
- Component reusability
  - Should reflect stable domain abstractions (e.g., a business object);
  - Should hide state representation;
  - Should 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.



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

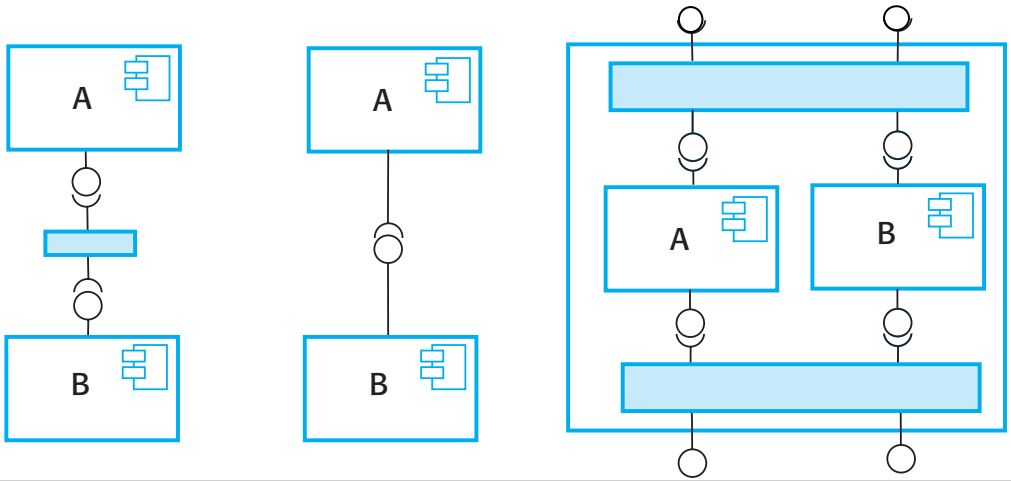
- Trust. You need to be able to trust the supplier of a component. At best, an untrusted component may not operate as advertised; at worst, it can breach your security.
- Requirements. Different groups of components will satisfy different requirements.
- Validation.
  - The component specification may 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.

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).

Component composition

- Composition involves integrating components with each other and with the component infrastructure to create a system.
- Normally you have to write 'glue code' to integrate components by resolving interface incompatibilities
- Types of composition:
  - Sequential, where the composed components are executed in sequence.
  - Hierarchical, where one component calls on the services of another.
  - Additive, where the interfaces of two components are put together to create a new component.



Interfacing when composing

- Interface incompatibilities
  - Parameter incompatibility where operations have the same name but are of different types.
  - Operation incompatibility where the names of operations in the composed interfaces are different.
  - Operation incompleteness where the provides interface of one component is a subset of the requires interface of another.
- Various adaptors, depending on the type of composition, are used to address component incompatibility.
- Component documentation is needed to decide if interfaces are actually, and not just syntactically compatible

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?

Current trends in reuse

- Reuse is having considerable impact among new software technologies such as cloud-based computing and very large data projects *a la* Hadoop.
- Application frameworks created for these new technologies are allowing companies to use the technology without having to deal with the details of the environment.
  - E.g., containerization capabilities from Docker and Cask
- Many of these frameworks are open source, but supported by commercial software vendors making them attractive for use by many companies.
- Open source, in general, has been very beneficial for reuse, making libraries of functions and components readily available, as well as giving impetus to design and application frameworks.

Classroom Activity – within your EMSS project teams

Activity:

Using your EMSS "architecture" developed in Week 6, identify the software components of your system and note the responsibilities that each takes on in your solution. Identify whether any of these components are likely to have been previously developed and made available (e.g., as COTS or via open source) and identify those that appear to be likely candidates for future reuse.

Next week, each team will do a 15-20 minute presentation to the class summarizing your EMSS solution. See Assignment EMSS Week 12 for more detail.

Submission to complete the Canvas Assignment EMSS Week 12:

Submit a brief paper on your components showing your findings and the reasoning used to arrive at those findings.

The submission of your presentation will be under the EMSS Presentation Assignment and due one hour before the next class.