2008 Fall

Software Modeling & Analysis

Part 4. Development

- Rapid Software Development
- Software Reuse
- Component-Based Software Engineering

Lecturer: JUNBEOM YOO
jbyoo@konkuk.ac.kr

※ This lecture note is based on materials from Ian Sommerville 2006.

Chapter 17.
Rapid Software Development

Objectives

- To explain how an iterative and incremental development process leads to faster delivery of more useful software
- To discuss the essence of agile development methods
- To explain the principles and practices of extreme programming
- To explain the roles of prototyping in the software process

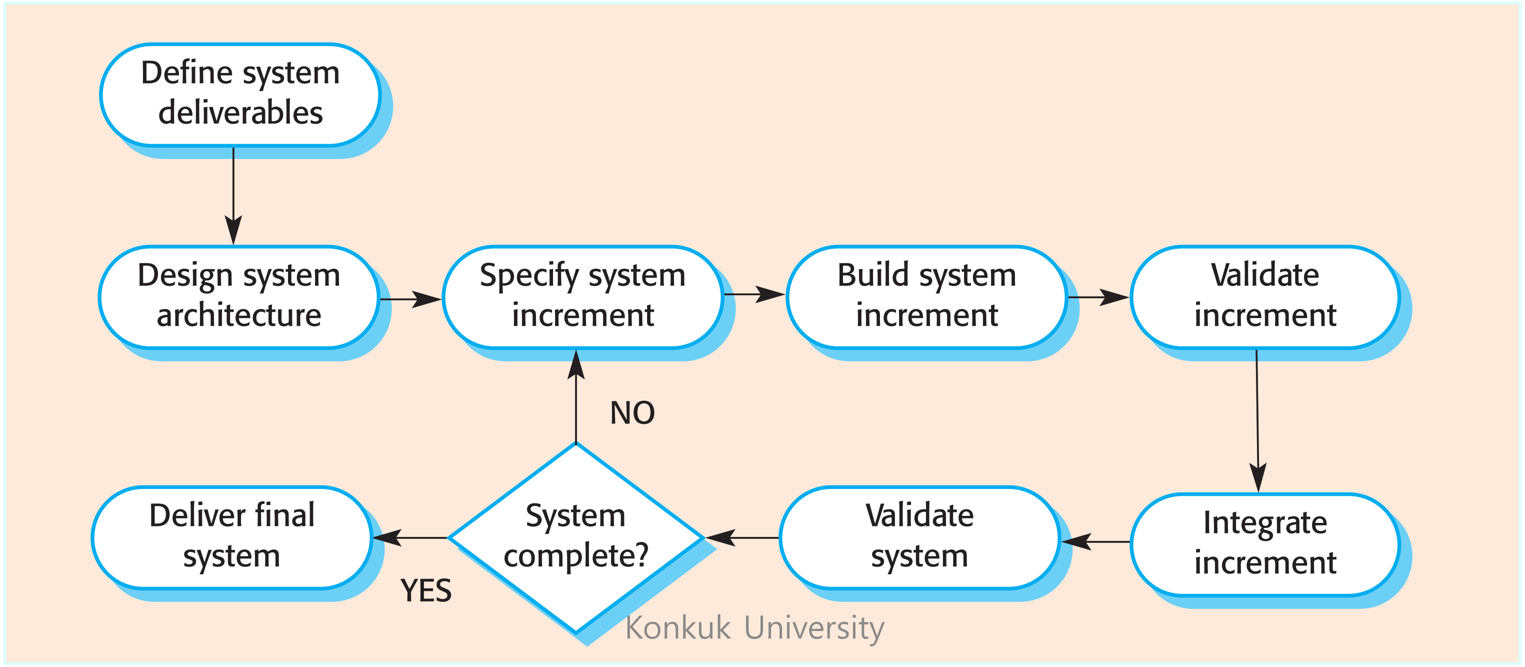
Rapid Software Development

- Because of rapidly changing business environments, businesses have to respond to new opportunities and competition.
- This requires rapid software development, and delivery is not often the most critical requirement for software systems.
- Businesses may be willing to accept lower quality software if rapid delivery of essential functionality is possible.

- Because of the changing environment, it is often impossible to arrive at a stable, consistent set of system requirements.
- Therefore a waterfall model of development is impractical
- Approach to development based on iterative specification and delivery is the only way to deliver software quickly.

Characteristics of R.S.D. Process

- Processes of specification, design and implementation are concurrent.
- No detailed specification and design documentation is minimized.
- The system is developed in <u>a series of increments</u>. End users evaluate each increment and make proposals for later increments.
- System user interfaces are usually developed using an interactive development system.



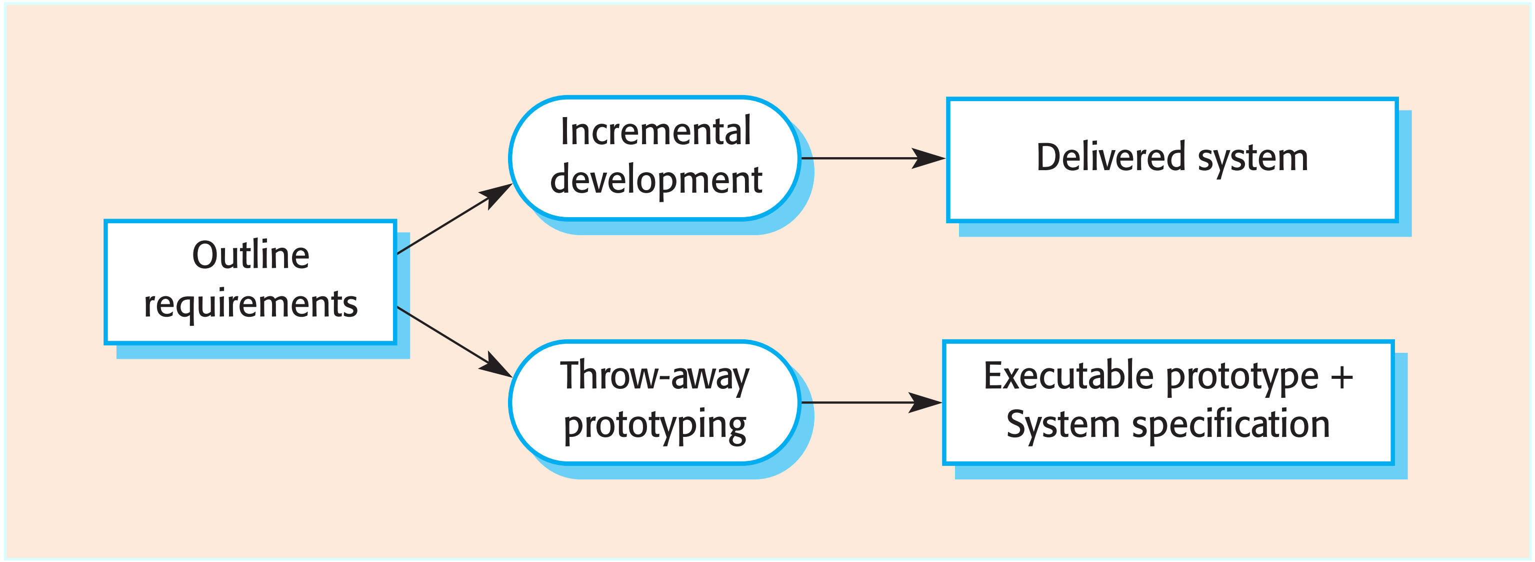
Characteristics of Incremental Development

- Advantages:
  - Accelerated delivery of customer services : Each increment delivers the highest priority functionality to the customer.
  - User engagement with the system : Users have to be involved in the development which means the system is more likely to meet their requirements and the users are more committed to the system.

- Problems:
  - Management problems : Progress can be hard to judge and problems hard to find because there is no documentation to demonstrate what has been done.
  - Contractual problems : The normal contract may include a specification; without a specification, different forms of contract have to be used.
  - Validation problems : Without a specification, what is the system being tested against?
  - Maintenance problems : Continual change tends to corrupt software structure making it more expensive to change and evolve to meet new requirements.

Prototyping

- For some large systems, incremental iterative development and delivery may be impractical. This is especially true when multiple teams are working on different sites.
- Prototyping, where an experimental system is developed as a basis for formulating the requirements, may be used. This system is thrown away when the system specification has been agreed.

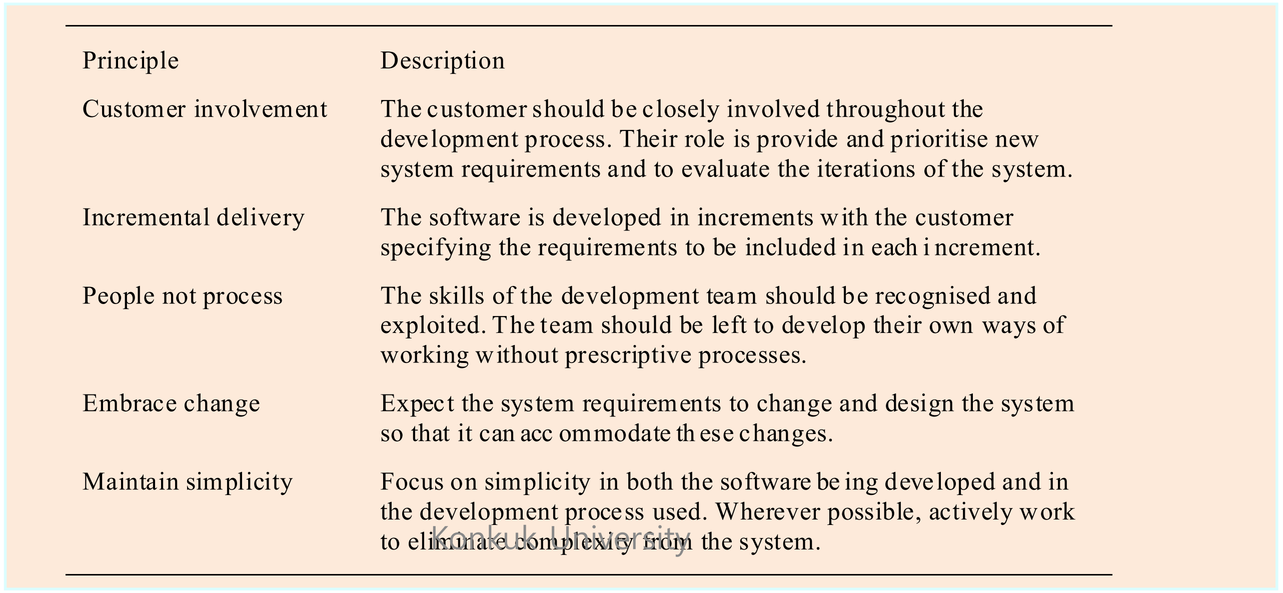


Conflicting Objectives

- The objective of incremental development is to deliver a working system to end-users. The development starts with those requirements which are best understood.
- The objective of throw-away prototyping is to validate or derive the system requirements. The prototyping process starts with those requirements which are poorly understood.

Agile Method

- Dissatisfaction with the overheads involved in design methods led to the creation of agile methods.
  - Focus on the code rather than the design
  - Are based on an iterative approach to software development
  - Are intended to deliver working software quickly and evolve this quickly to m eet changing requirements
- Agile methods are probably best suited to small/medium-sized business systems or PC products.



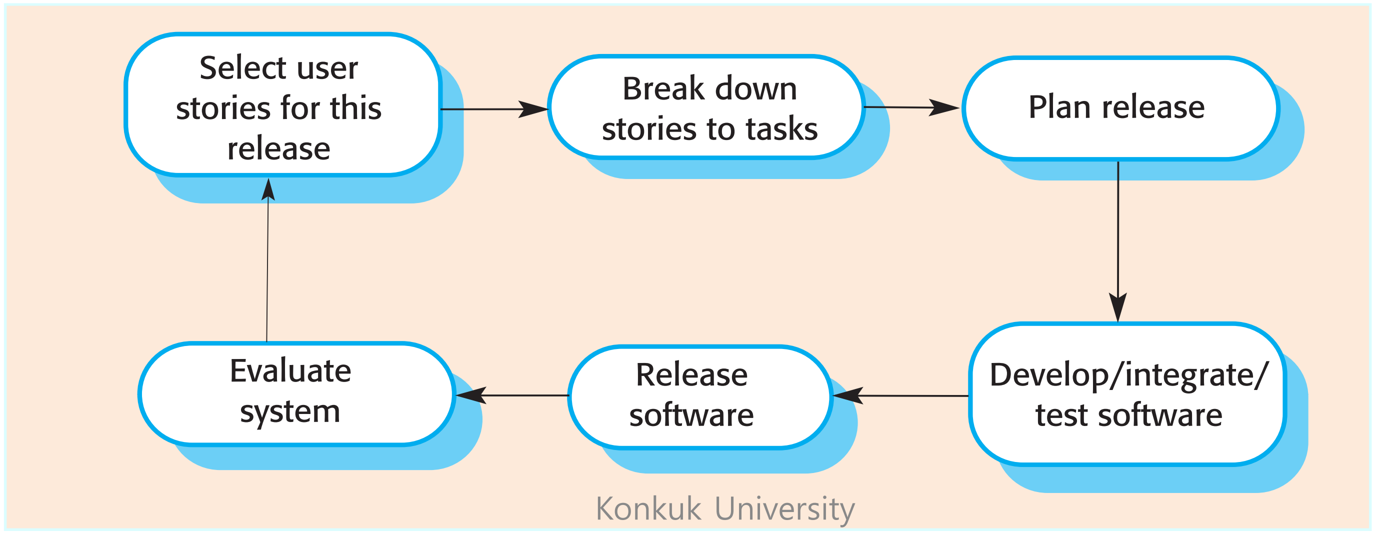
Principle	Description
Customer involvement	The customer should be closely involved throughout the development process. Their role is provide and prioritise new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each i ncrement.
People not process	The skills of the development team should be recognised and exploited. The team should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and design the system so that it can acc ommodate th ese changes.
Maintain simplicity	Focus on simplicity in both the software be ing developed and in the development process used. Wherever possible, actively work to eliminate complexity from the system.

Problems with Agile Method

- It can be difficult to keep the interest of customers who are involved in the process.
- Team members may be unsuited to the intense involvement that characterizes agile methods.
- Prioritizing changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.

Extreme Programming

- Perhaps the best-known and most widely used agile method.
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
  - New versions may be built several times per day.
  - Increments are delivered to customers every 2 weeks.
  - All tests must be run for every build and the build is only accepted if tests run successfully.
- XP release cycle:



Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/ test software
Release software
Evaluate system

Testing in XP

- SP is a test-first development.
- Incremental test development from scenarios
- Users are involved in test development and validation.
- Automated test harnesses are used to run all component tests each time that a new release is built.

- Test-first development:
  - Writing tests before code clarifies the requirements to be implemented.
  - Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
  - All previous and new tests are automatically run when new functionality is added. Thus checking that the new functionality has not introduced errors.

Test-First Development

- Writing tests before code clarifies the requirements to be implemented.
- Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
- All previous and new tests are automatically run when new functionality is added. Thus checking that the new functionality has not introduced errors.

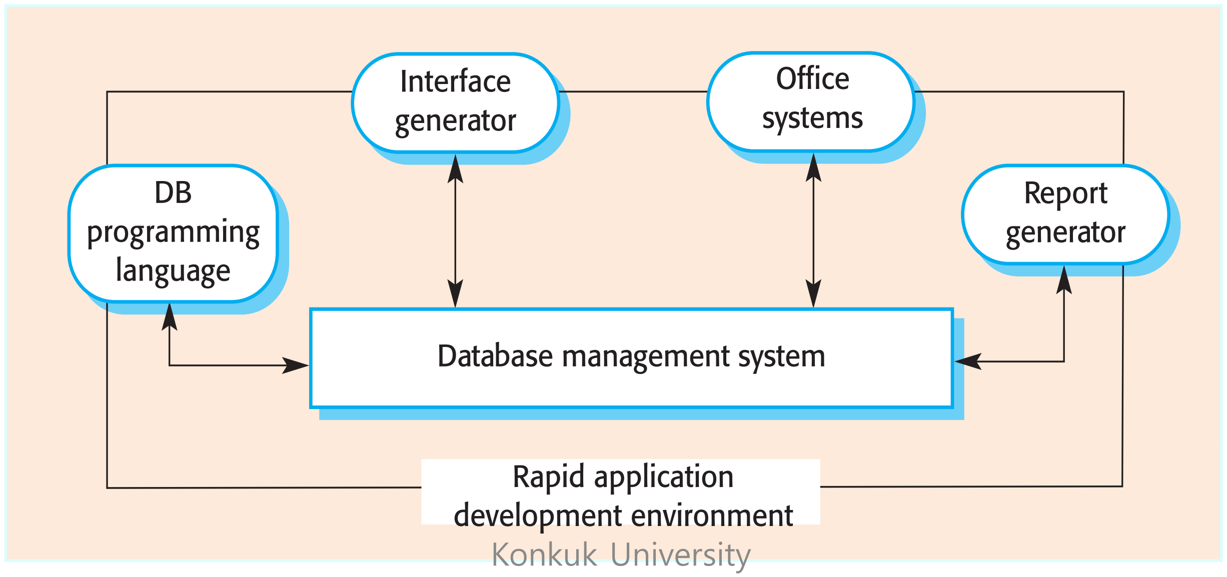
Pair Programming in XP

- In XP, programmers work in pairs, sitting together to develop code.
- This helps develop common ownership of code and spreads knowledge across the team.
- It serves as an informal review process as each line of code is looked at by more than 1 person.
- It encourages refactoring as the whole team can benefit from this.

- Measurements suggest that development productivity with pair programming is similar to that of two people working independently.

RAD (Rapid Application Development)

- Agile methods have received a lot of attention but other approaches to rapid application development have been used for many years.
- These are designed to develop data-intensive business applications and rely on programming and presenting information from a database.
- RAD environment:
  - Database programming language - Interface generator
  - Links to office applications - Report generators

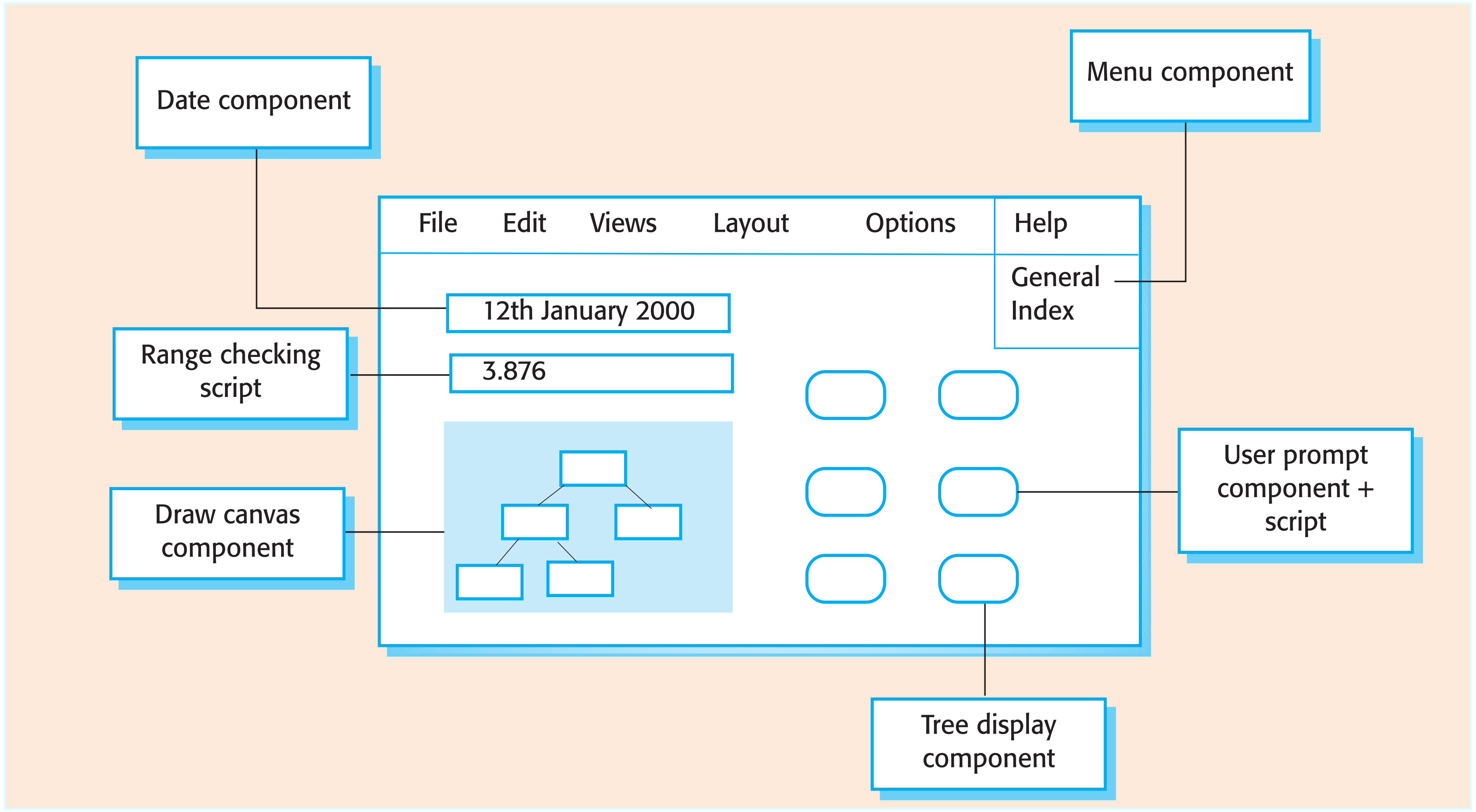


Interface Generation

- Many applications are based on complex forms and developing these forms manually is a time-consuming activity.
- RAD environments include support for screen generation including:
  - Interactive form definition using drag and drop techniques
  - Form linking where the sequence of forms to be presented is specified
  - Form verification where allowed ranges in form fields is defined

- Visual Programming:
  - Scripting languages such as Visual Basic support visual programming where the prototype is developed by creating a user interface from standard items and associating components with these items
  - A large library of components exists to support this type of development
  - These may be tailored to suit the specific application requirements

Visual Programming with Reuse



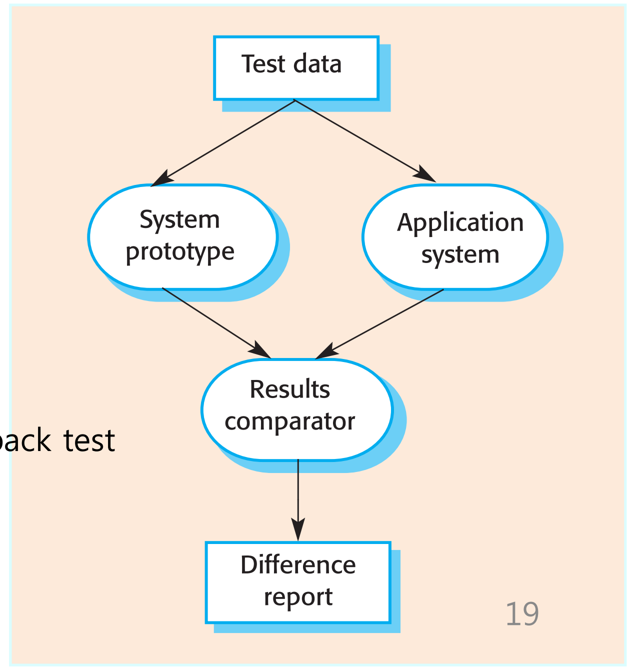
COTS Reuse

- An effective approach to rapid development is to configure and link existing off the shelf systems.
- For example, a requirements management system could be built by using:
  - A database to store requirements
  - A word processor to capture requirements and format reports
  - A spreadsheet for traceability management

Software Prototyping

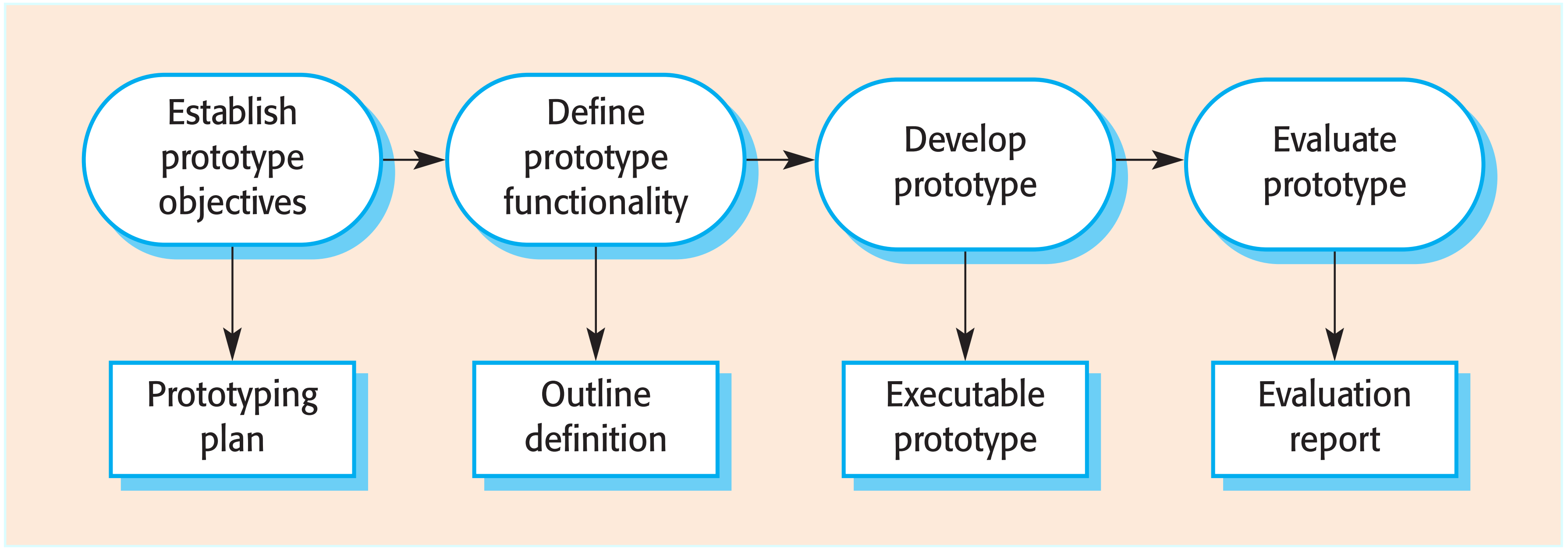
- A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- A prototype can be used in:
  - The requirements engineering process to help with requirements elicitation and validation
  - In design processes to explore options and develop a UI design
  - In the testing process to run back-to-back tests

- Benefits of prototyping
  - Improved system usability
  - A closer match to users' real needs
  - Improved design quality
  - Improved maintainability
  - Reduced development effort



Back-to-back test

Prototyping Process



Summary

- An iterative approach to software development leads to faster delivery of software.
- Agile methods are iterative development methods that aim to reduce development overhead and so produce software faster.
- Extreme programming includes practices such as systematic testing, continuous improvement and customer involvement.
- Testing approach in XP is a particular strength where executable tests are developed before the code is written.
- Rapid application development (RAP) environments include database programming languages, form generation tools and links to office applications.
- A throw-away prototype is used to explore requirements and design options.
- When implementing a throw-away prototype, start with the requirements you least understand, on the other hands, in incremental development, start with the best-understood requirements.

Chapter 18.
Software Reuse

Objectives

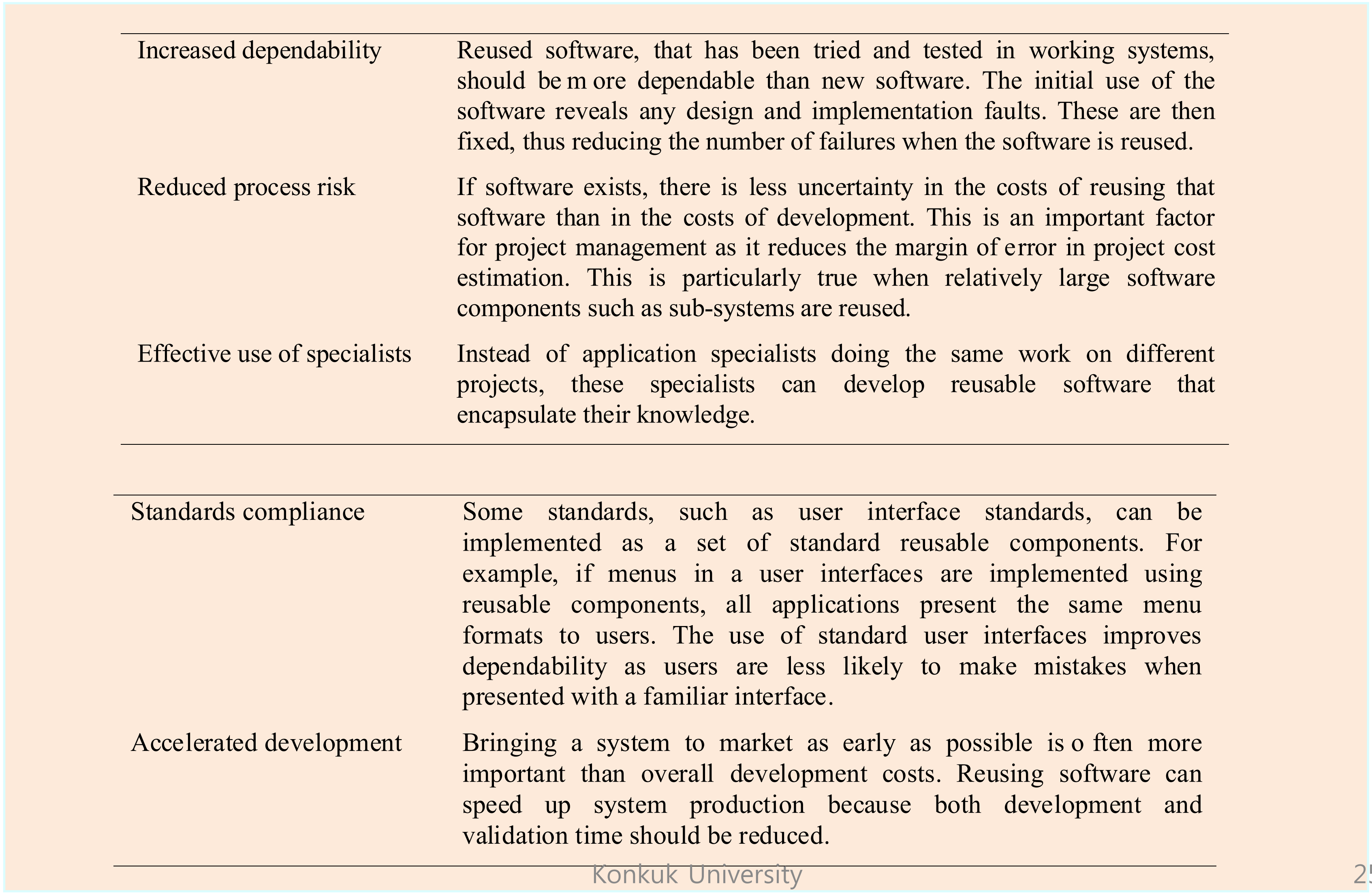
- To explain benefits of software reuse and some reuse problems
- To discuss several different ways to implement software reuse
- To explain how reusable concepts can be represented as patterns or embedded in program generators
- To discuss COTS reuse
- To describe the development of software product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

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 <u>systematic software reuse</u>.

- 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.

Benefits of Reuse



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.

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.

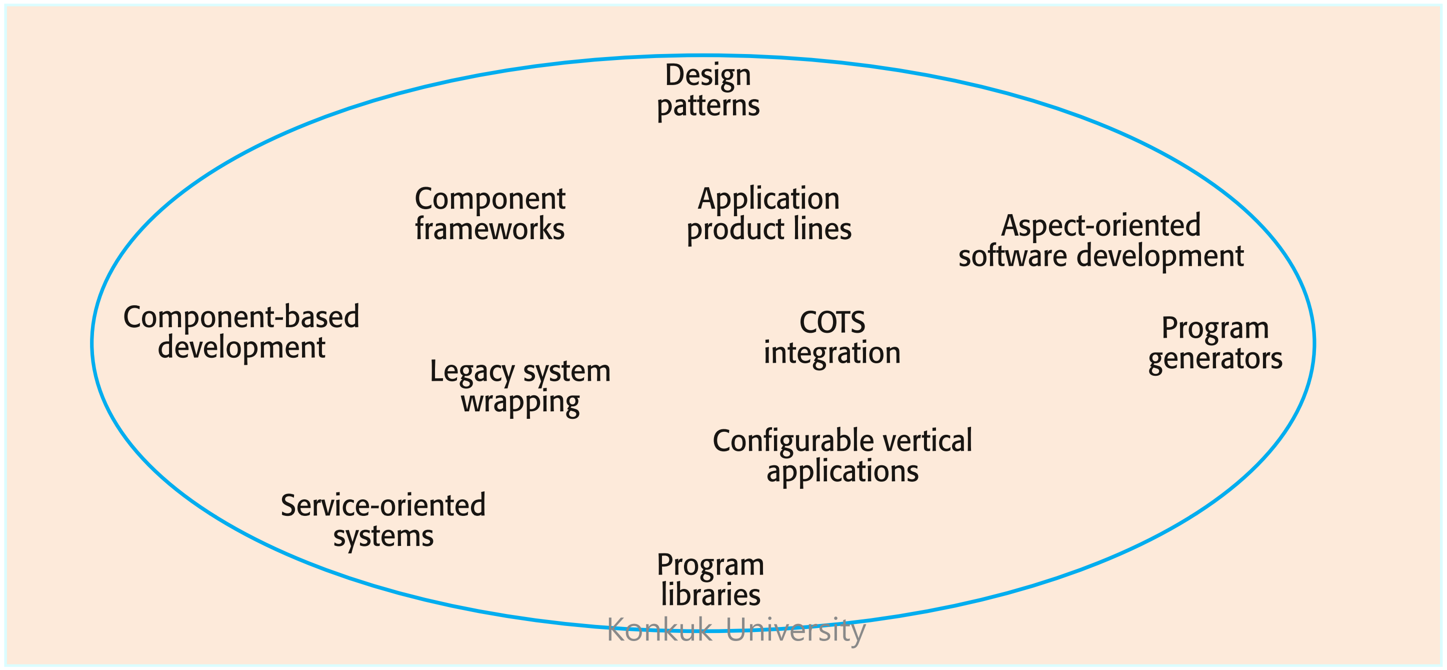
Problems in Reuse

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 peopleÕs software.

Creating and maintaining a component library	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.
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 will make routinely include a component search as part of their normal development process.

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 Approaches

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

Reuse: 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.

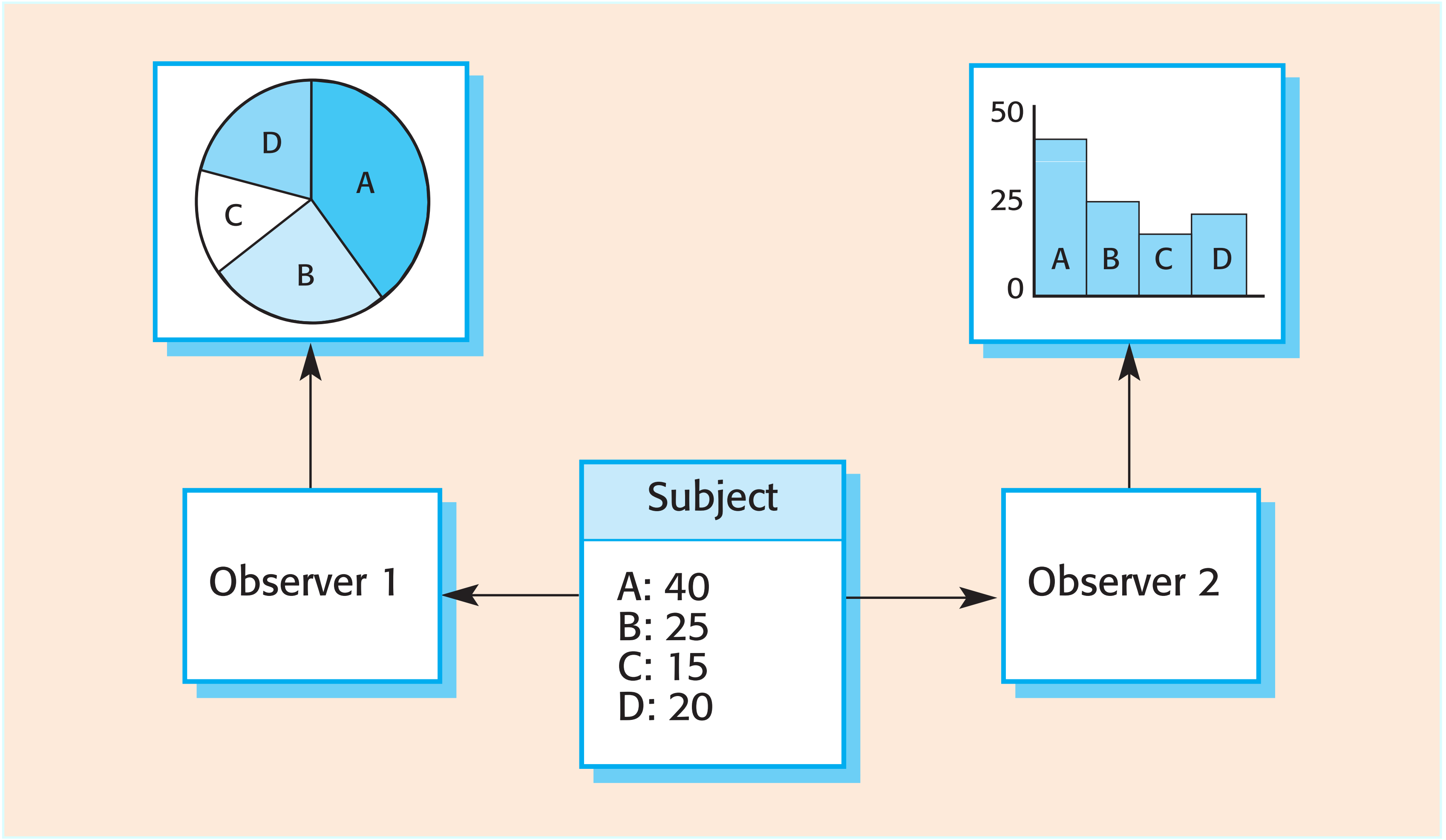
- Two main approaches to concept reuse are:
  - Design patterns
  - Generative programming (Program generator)

Design Pattern

- 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.

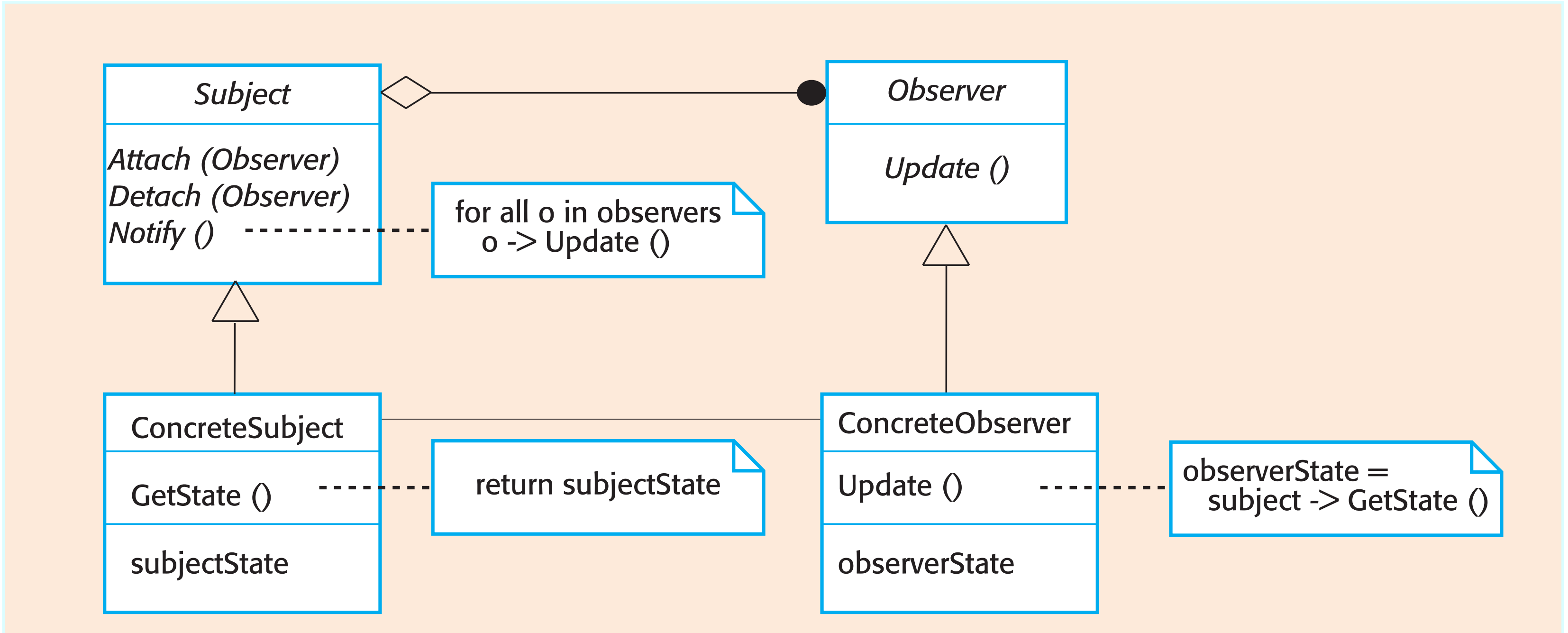
- Pattern element
  - Name : Meaningful pattern identifier.
  - Problem description
  - Solution description : Not a concrete design but a template for a design solution that can be instantiated in different ways.
  - Consequences : Results and trade-offs of applying the pattern.

Design Pattern Example: Multiple Displays



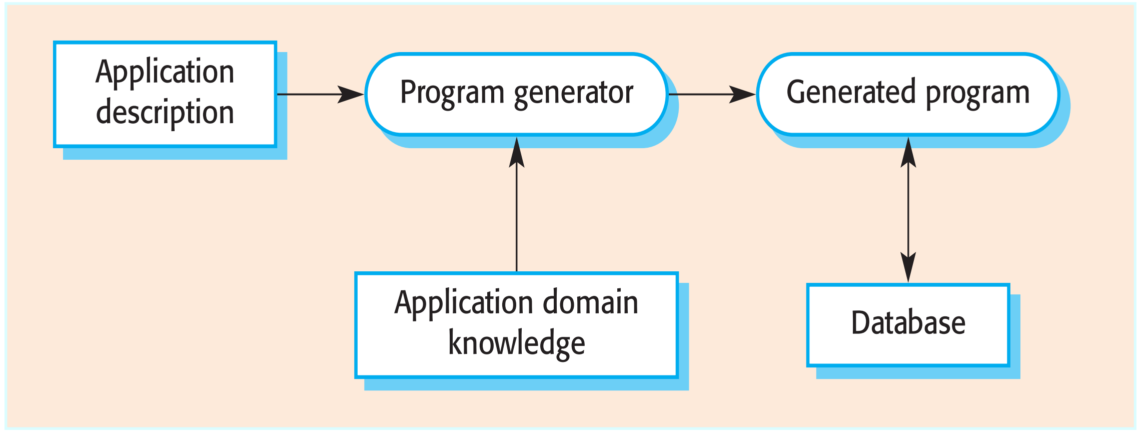
Observer Pattern

- 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.



Generator-Based Reuse

- Program generators involve the reuse of standard patterns and algorithms.
- These are embedded in the generator and parameterised by user commands. A program is then automatically generated.
- Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified.
- A domain specific language is used to compose and control these abstractions.

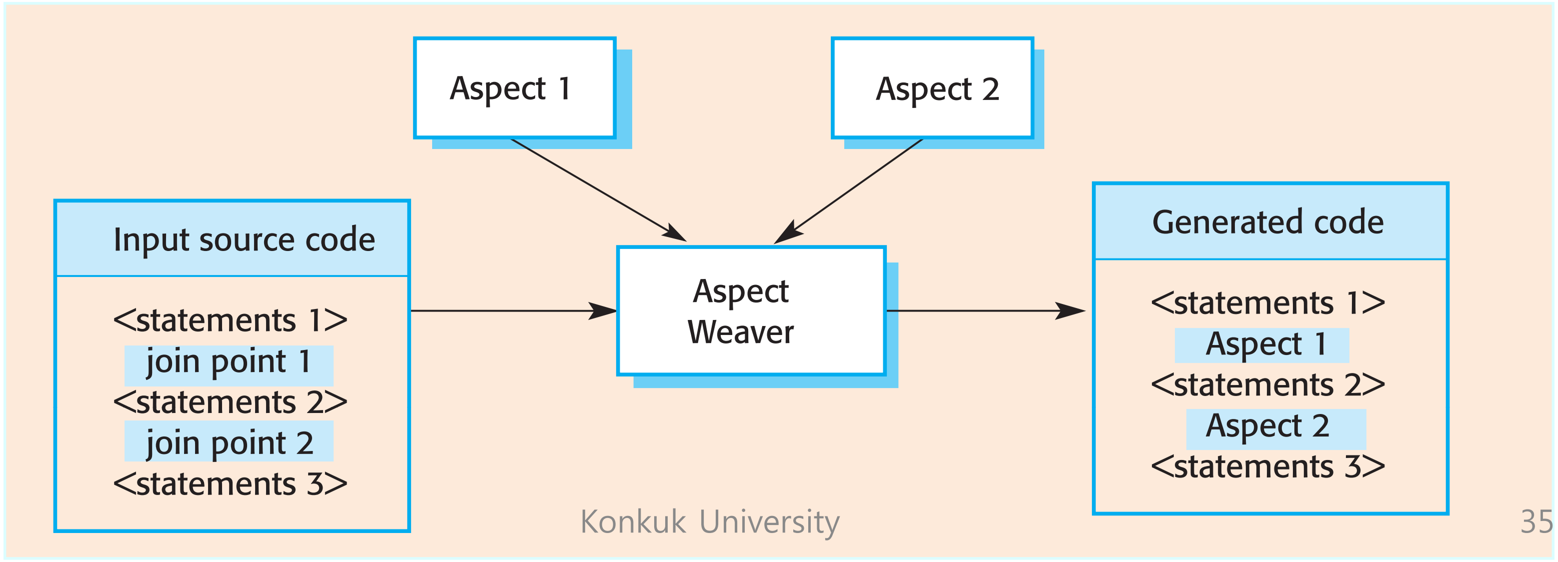


Types of Program Generator

- Types of program generator
  - Application generators for business data processing
  - Parser and lexical analyser generators for language processing
  - Code generators in CASE tools
- Generator-based reuse is very cost-effective but its applicability is limited to a relatively small number of application domains.
- It is easier for end-users to develop programs using generators compared to other component-based approaches to reuse.

Reuse: Aspect-Oriented Development

- Aspect-oriented development addresses a major software engineering problem - the separation of concerns.
- Concerns are often not simply associated with application functionality but are cross-cutting - e.g. all components may monitor their own operation, all components may have to maintain security, etc.
- Cross-cutting concerns are implemented as aspects and are dynamically woven into a program. The concern code is reused and the new system is generated by the aspect weaver.



Reuse: 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.

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

COTS Product Reuse

- COTS - Commercial Off-The-Shelf

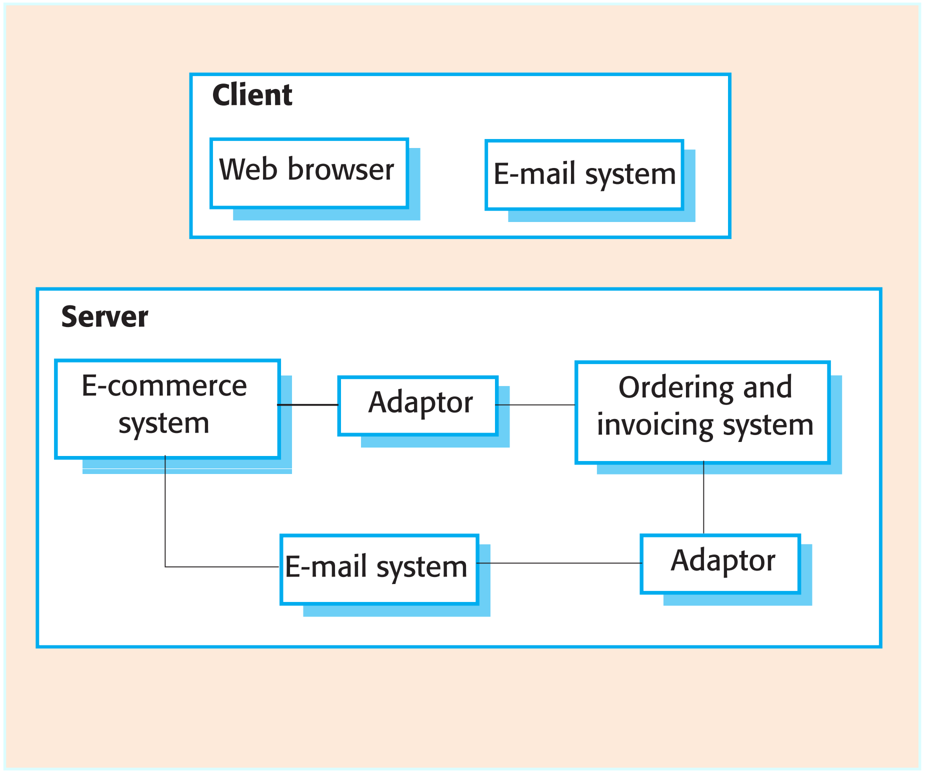
- COTS systems are usually complete application systems that offer an API (Application Programming Interface).
- Building large systems by integrating COTS systems is now a viable development strategy for some types of system such as E-commerce systems.
- The key benefit is faster application development and, usually, lower development costs.

COTS Design Choices

- Which COTS products offer the most appropriate functionality?
  - There may be several similar products that may be used.
- How will data be exchanged?
  - Individual products use their own data structures and formats.
- What features of the product will actually be used?
  - Most products have more functionality than is needed.
  - You should try to deny access to unused functionality.

- COTS system integration problems:
  - Lack of control over functionality and performance
    - COTS systems may be less effective than they appear
  - Problems with inter-operability
    - Different COTS systems may make different assumptions that means integration is difficult
  - No control over system evolution
    - COTS vendors do not control system evolution
  - Support from COTS vendors
    - COTS vendors may not offer support over the lifetime of the product

Example: E-Procurement System



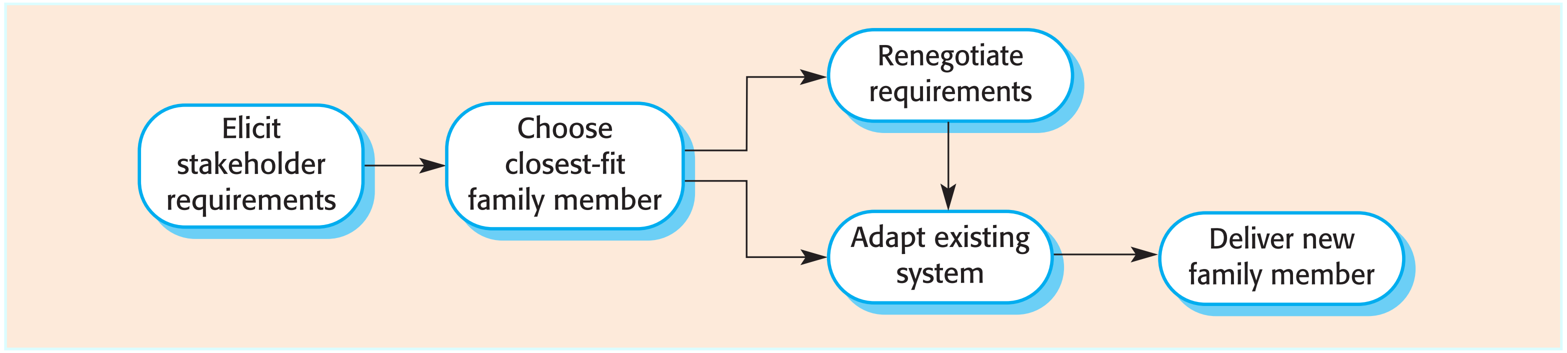
- On the client, standard e-mail and web browsing programs are used.
- On the server, an e-commerce platform has to be integrated with an existing ordering system.
  - This involves writing an adaptor so that they can exchange data.
  - An e-mail system is also integrated to generate e-mail for clients. This also requires an adaptor to receive data from the ordering and invoicing system.

Software Product Line

- Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context.

- 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

Product Instance Development



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

- 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

Summary

- Advantages of reuse are lower costs, faster software development and lower risks.
- Design patterns are high-level abstractions that document successful design solutions.
- Program generators are also concerned with software reuse - the reusable concepts are embedded in a generator system.
- Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialisation.
- COTS product reuse is concerned with the reuse of large, off-the-shelf systems.
- Problems with COTS reuse include lack of control over functionality, performance, and evolution and problems with inter-operation.
- Software product lines are related applications developed around a common core of shared functionality.

Chapter 19.
Component-Based Software Engineering

Objectives

- To explain that CBSE is concerned with developing standardized components and composing these into applications
- To describe components and component models
- To show principal activities in CBSE process
- To discuss approaches to component composition and problems that may arise

Component-Based Development

- Component-based software engineering (CBSE) is an approach to software development that relies on software reuse.
- 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.

CBSE Essentials

- 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

- 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.
  - Component platforms are shared and reduce development costs.

CBSE Problems

- Component trustworthiness
  - How can a component with no available source code be trusted?
- Component certification
  - Who will certify quality of the components?
- Emergent property prediction
  - How can the emergent properties of component compositions be predicted?
- Requirements trade-offs
  - How do we do trade-off analysis between the features of one component and another?

Components

- Components provide a service without regard to where the component is executing 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.

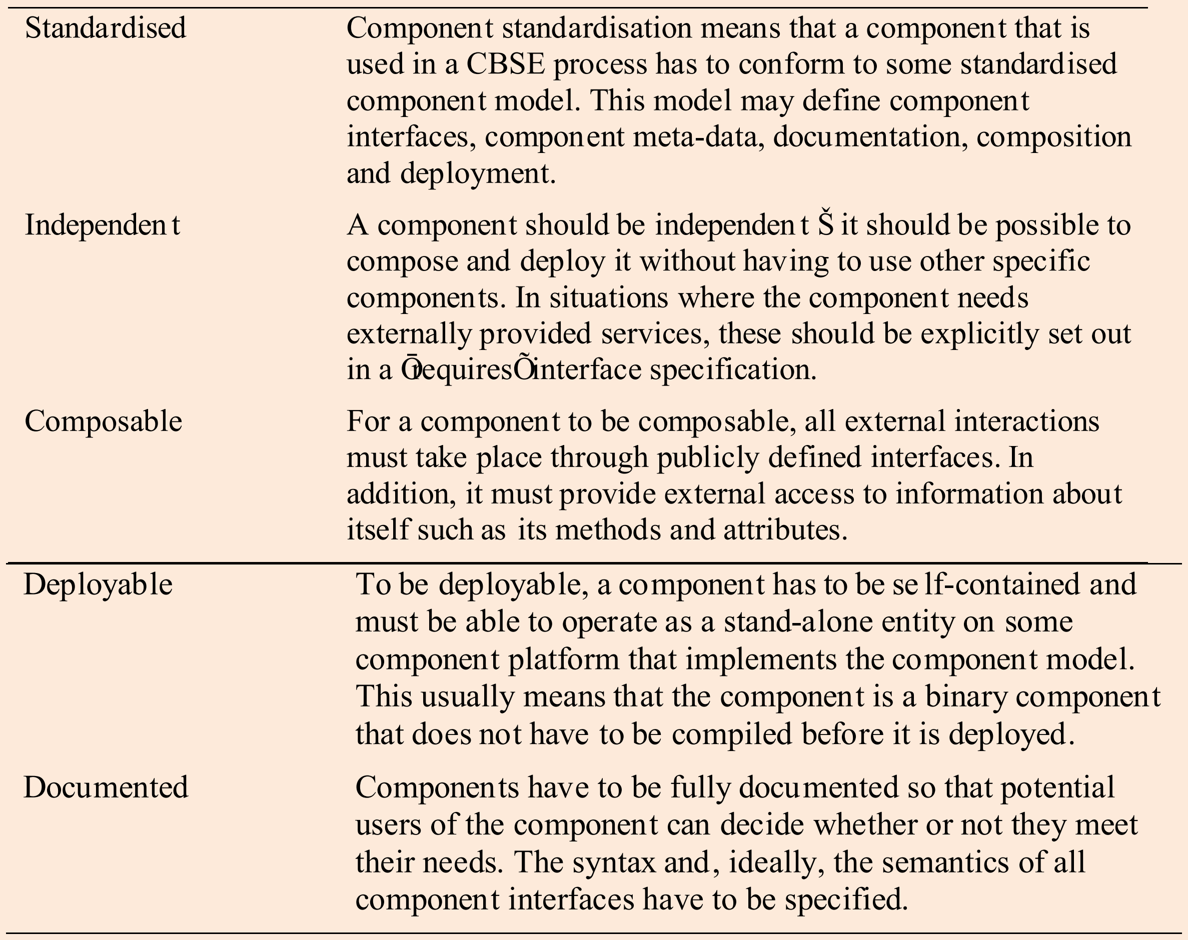
Councill and Heinmann:

A software component is a software element that conforms to a component model and can be independently deploye d and composed without modification according to a com position standard.

Szyperski:

A software component is a unit of composition with contra ctually specified interfaces and explicit context dependenci es only. A software component can be deployed independe ntly and is subject to composition by third-parties.

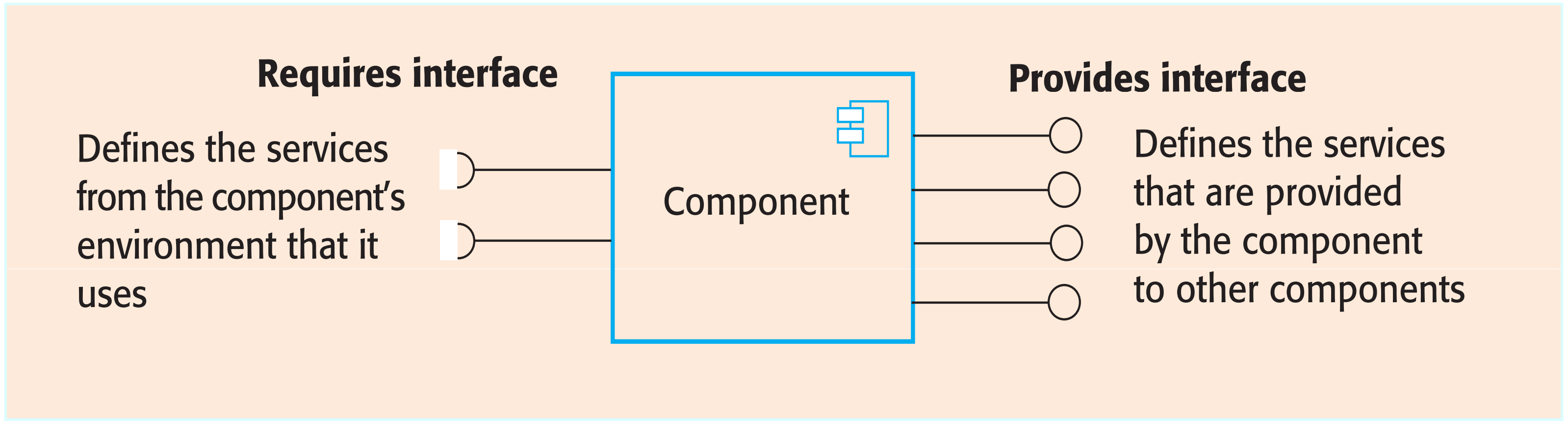
Characteristics of Components



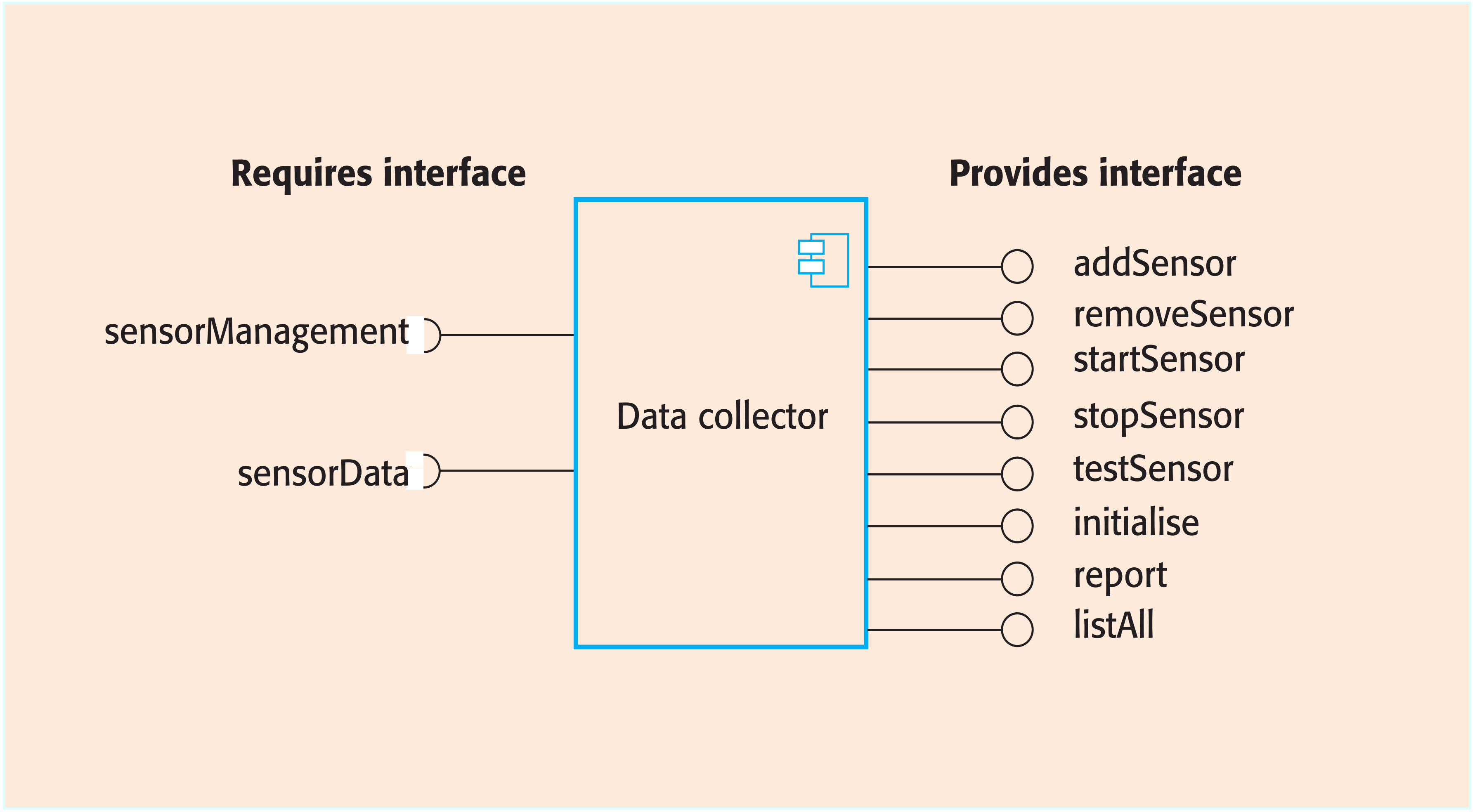
Standardised	Component standardisation means that a component that is used in a CBSE process has to conform to some standardised component model. This model may define component interfaces, component meta-data, documentation, composition and deployment.
Independent	A component should be independent Š it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a ŌrequiresÕ interface specification.
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity on some component platform that implements the component model. This usually means that the component is a binary component that does not have to be compiled before it is deployed.
Documented	Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified.

Component Interface

- Provides interface
  - Defines the services that are provided by the component to other components.
- Requires interface
  - Defines the services that specifies what services must be made available for the component to execute as specified.


Requires interface
Defines the services from the component's environment that it uses
Component
Provides interface
Defines the services that are provided by the component to other components

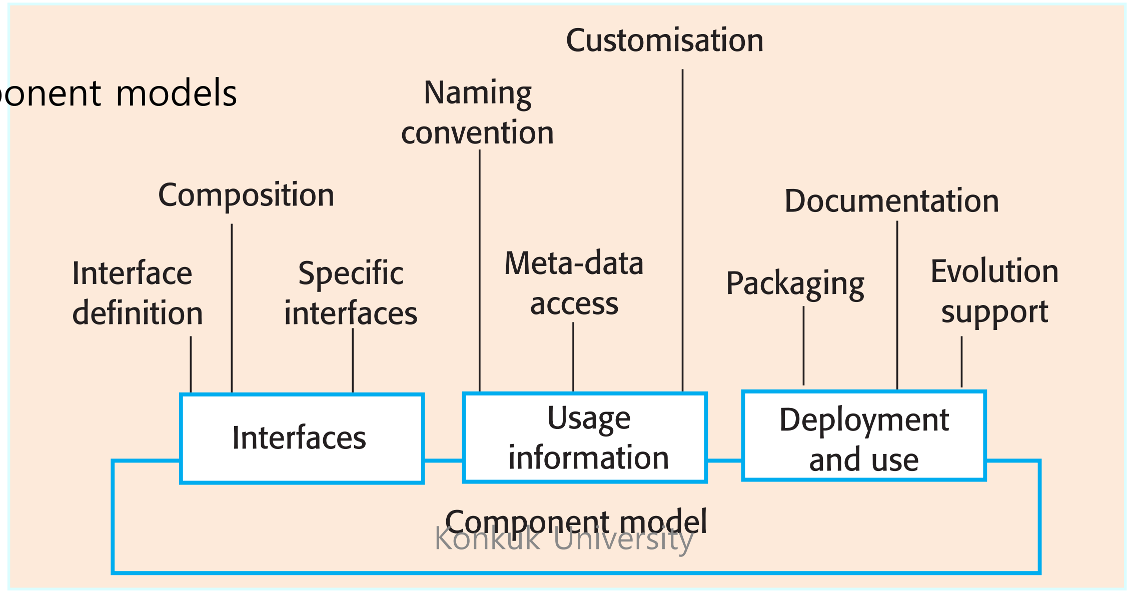
Example: A Data Collector Component Interface



Component Model

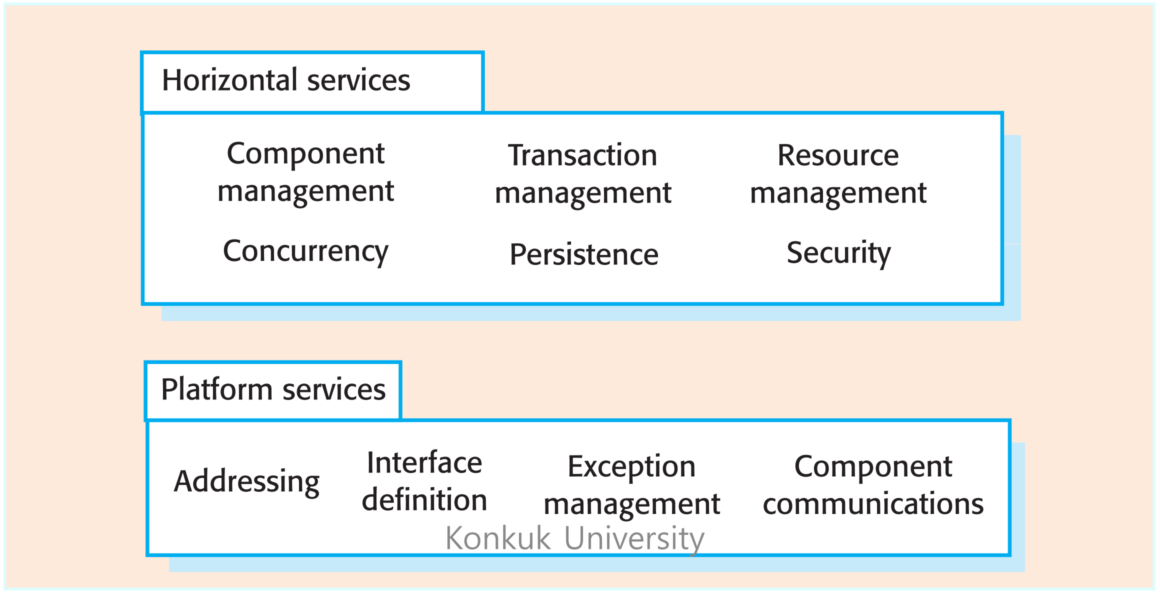
- A component model is a definition of standards for component implementation, documentation and deployment.
- Examples of component models
  - EJB model (Enterprise Java Beans)
  - COM+ model (.NET model)
  - CORBA Component Model
- Component model specifies how interfaces should be defined and the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

Elements of component models



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
  - Horizontal 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.



Component Development for Reuse

- Components developed for a specific application usually have to be generalized to make them reusable.
- A component is most likely to be reusable if it associated with a stable domain abstraction (business object).
  - In a hospital, stable domain abstractions are associated with the fundamental purpose - nurses, patients, treatments, etc.

- Component reusability
  - Should reflect stable domain abstractions
  - 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.

Legacy System Components

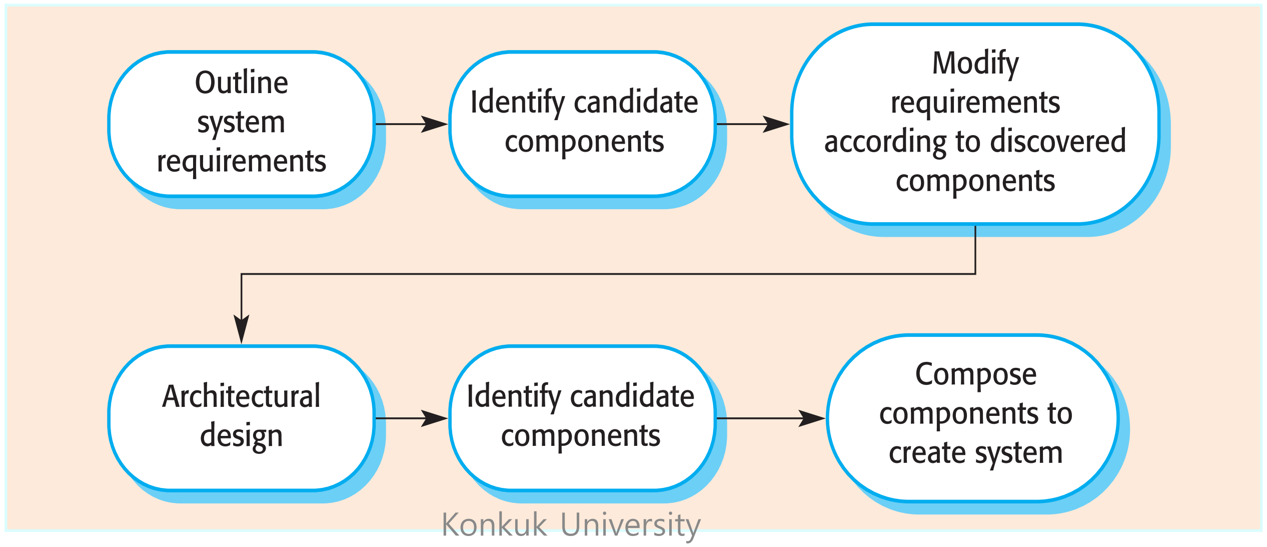
- Existing legacy systems that fulfill a useful business function can be re-packaged as components for reuse.
- This involves writing a wrapper component that implements provides and requires interfaces then accesses the legacy system.
- Although costly, this can be much less expensive than rewriting the legacy system.

Cost of Reusable Component

- The development cost of reusable components may be higher than the c
  ost of specific equivalents.
- This extra reusability enhancement cost should be an organization rather than a project cost.
- Generic components may be less space-efficient and may have longer ex
  ecution times than their specific equivalents.

CBSE Process

- 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


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

Component Identification Issues

- Trust
  - You need to be able to trust the supplier of a component. At best, an un-trusted 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?

Example: Ariane Launcher Failure

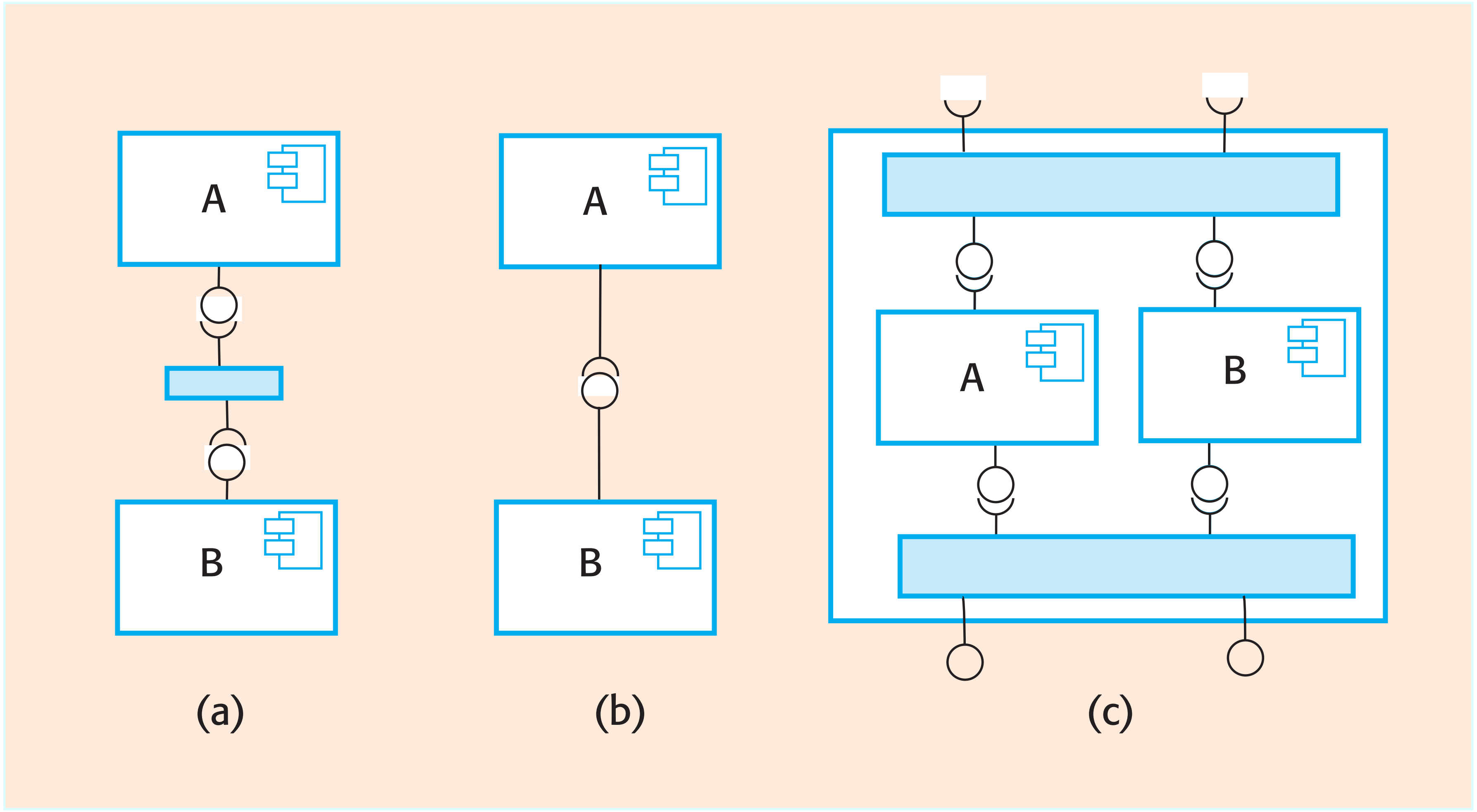
- In 1996, the 1st test flight of the Ariane 5 rocket ended in disaster when the launcher went out of control 37 seconds after take off.
- The problem was due to a reused component from a previous version of the launcher (the Inertial Navigation System) that failed because assumptions made when that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

Component Composition

- Process of assembling components to create a system
- Composition involves integrating components with each other and with the component infrastructure.
- Normally you have to write 'glue code' to integrate components.

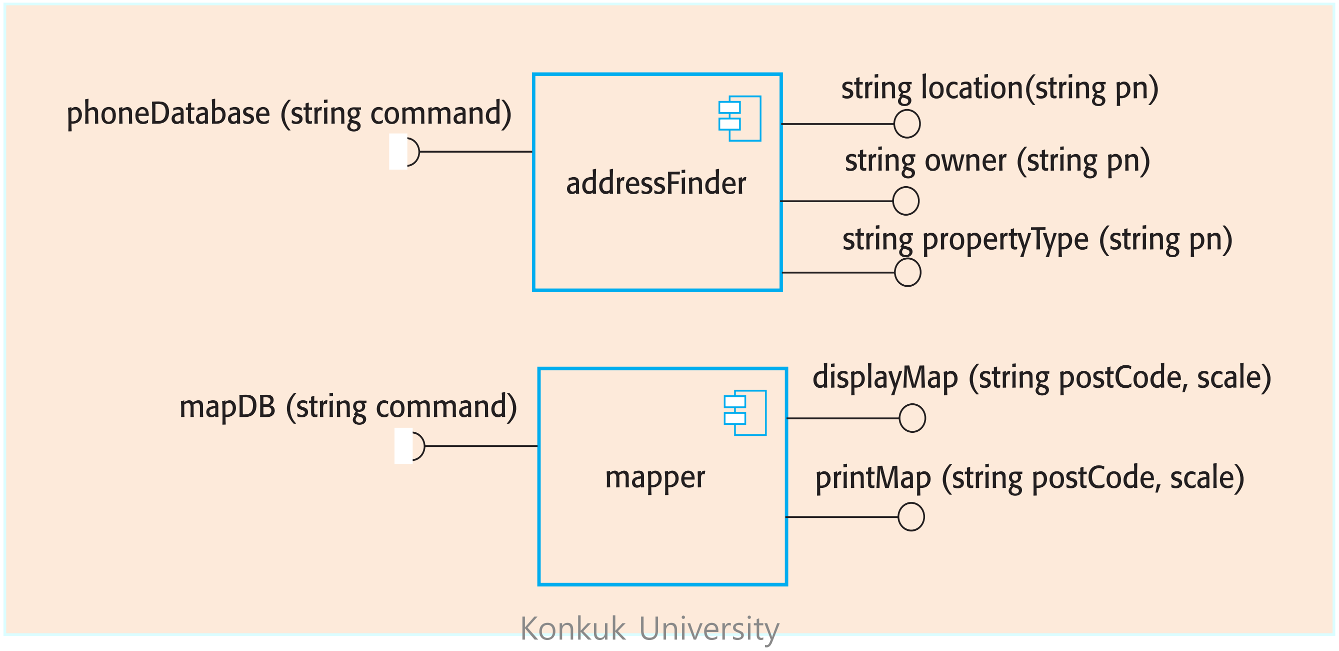
- Types of composition
  - Sequential composition where the composed components are executed in sequence. This involves composing the provides interfaces of each component.
  - Hierarchical composition where one component calls on the services of another. The provides interface of one component is composed with the requires interface of another.
  - Additive composition where the interfaces of two components are put together to create a new component.

Types of Composition



Interface Incompatibility

- 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.



phoneDatabase (string command)
addressFinder
string location(string pn)
string owner (string pn)
string propertyType (string pn)
mapDB (string command)
mapper
displayMap (string postCode, scale)
printMap (string postCode, scale)

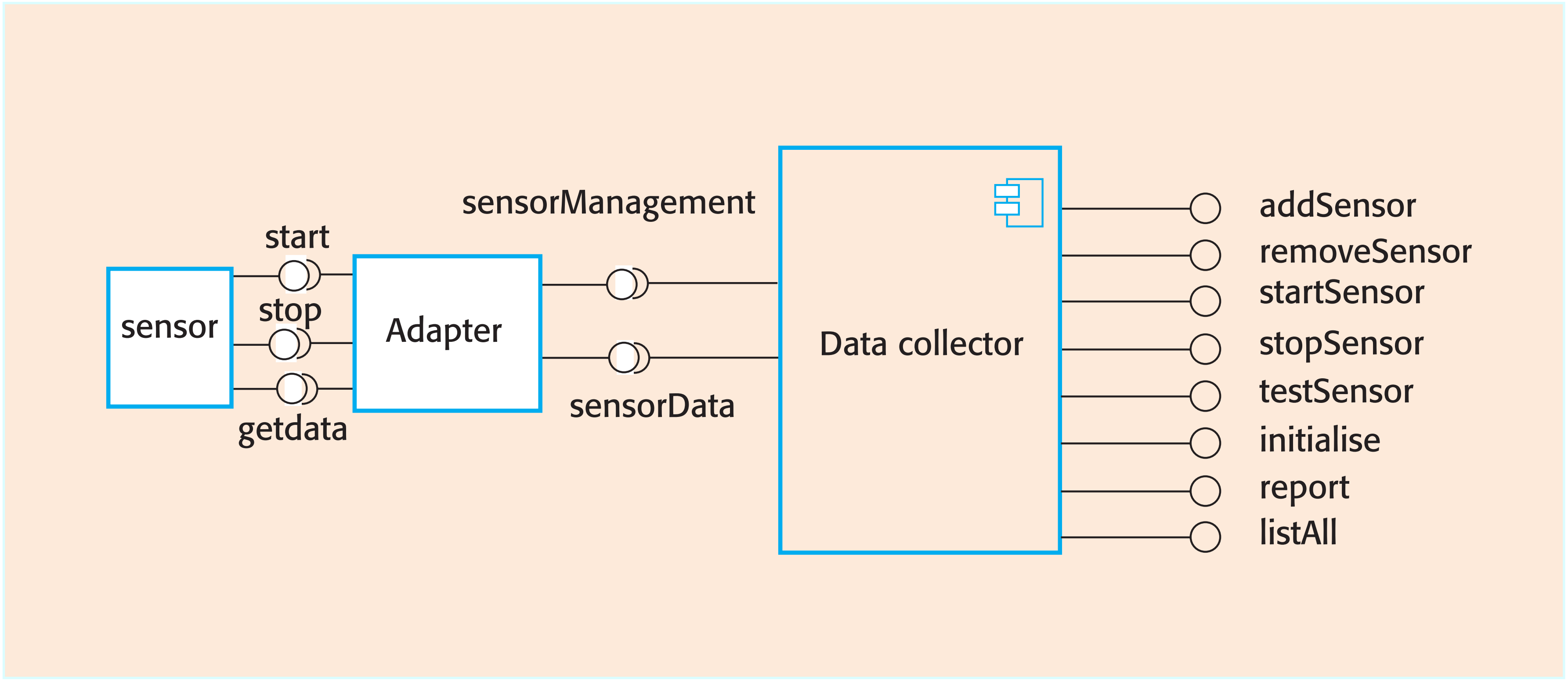
Adaptor Component

- Address the problem of component incompatibility by reconciling the interfaces of the components that are composed.
- Different types of adaptor are required depending on the type of composition.

- An *addressFinder* and a *mapper* component may be composed through an adaptor that strips the postal code from an address and passes this to the mapper component.

```
        address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000)
```

Adaptor for Data Collector Component



Interface Semantics

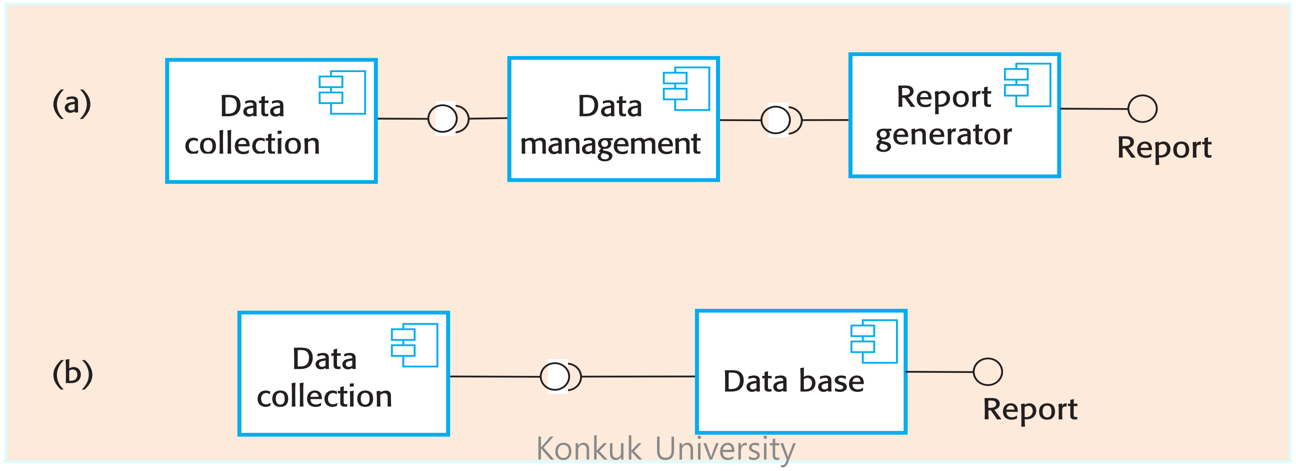
- You have to rely on component documentation to decide if interfaces that are syntactically compatible are actually compatible.

- Object Constraint Language (OCL)
  - OCL has been designed to define constraints that are associated with UML models.
  - It is based around the notion of pre and post condition specification - similar to the approach used in Z.

```
-- The context keyword names the component to which the conditions apply
context addItem

-- The preconditions specify what must be true before execution of addItem
pre:    PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

-- The postconditions specify what is true after execution
post:   libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

context delete

pre: PhotoLibrary.retrieve(pid) <> null ;

post: PhotoLibrary.retrieve(pid) = null
       PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
       PhotoLibrary.libSize() = libSize()@pre - 1
```

Trade-Offs in Composition

- 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?



Summary

- CBSE is a reuse-based approach to defining and implementing loosely coupled components into systems.
- A component is a software unit whose functionality and dependencies are completely defined by its interfaces.
- A component model defines a set of standards that component providers and composers should follow.
- During the CBSE process, the processes of requirements engineering and system design are interleaved.
- Component composition is the process of ‘wiring’ components together to create a system.
- When composing reusable components, you normally have to write adaptors to reconcile different component interfaces.
- When choosing compositions, you have to consider required functionality, non-functional requirements and system evolution.