Lecture 5 – System Design (Sommerville Ch. 14)

Karl R. Wilcox
K.R.Wilcox@reading.ac.uk

Objectives

- Design “from scratch” is relatively straight-forward, but time consuming – the real challenge is design with reuse
- To explain the benefits of software reuse and some reuse problems
- To describe different types of reusable component and processes for reuse
- To describe design patterns as high-level abstractions that promote reuse

Software reuse

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

Reuse-based software engineering

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

Reuse practice

- Application system reuse
  - Widely practised as software systems are implemented as application families. COTS reuse is becoming increasingly common
- Component reuse
  - Now seen as the key to effective and widespread reuse through component-based software engineering. However, it is still relatively immature
- Function reuse
  - Common in some application domains (e.g. engineering) where domain-specific libraries of reusable functions have been established

Benefits of reuse

- Increased reliability
  - Components exercised in working systems
- Reduced process risk
  - Less uncertainty in development costs
- Effective use of specialists
  - Reuse components instead of people
- Standards compliance
  - Embed standards in reusable components
- Accelerated development
  - Avoid original development and hence speed-up production

Requirements for design with reuse

- It must be possible to find appropriate reusable components
- The reuser of the component must be confident that the components will be reliable and will behave as specified
- The components must be documented so that they can be understood and, where appropriate, modified

Reuse problems

- Increased maintenance costs
- Lack of tool support
- Not-invented-here syndrome
- Maintaining a component library
- Finding and adapting reusable components

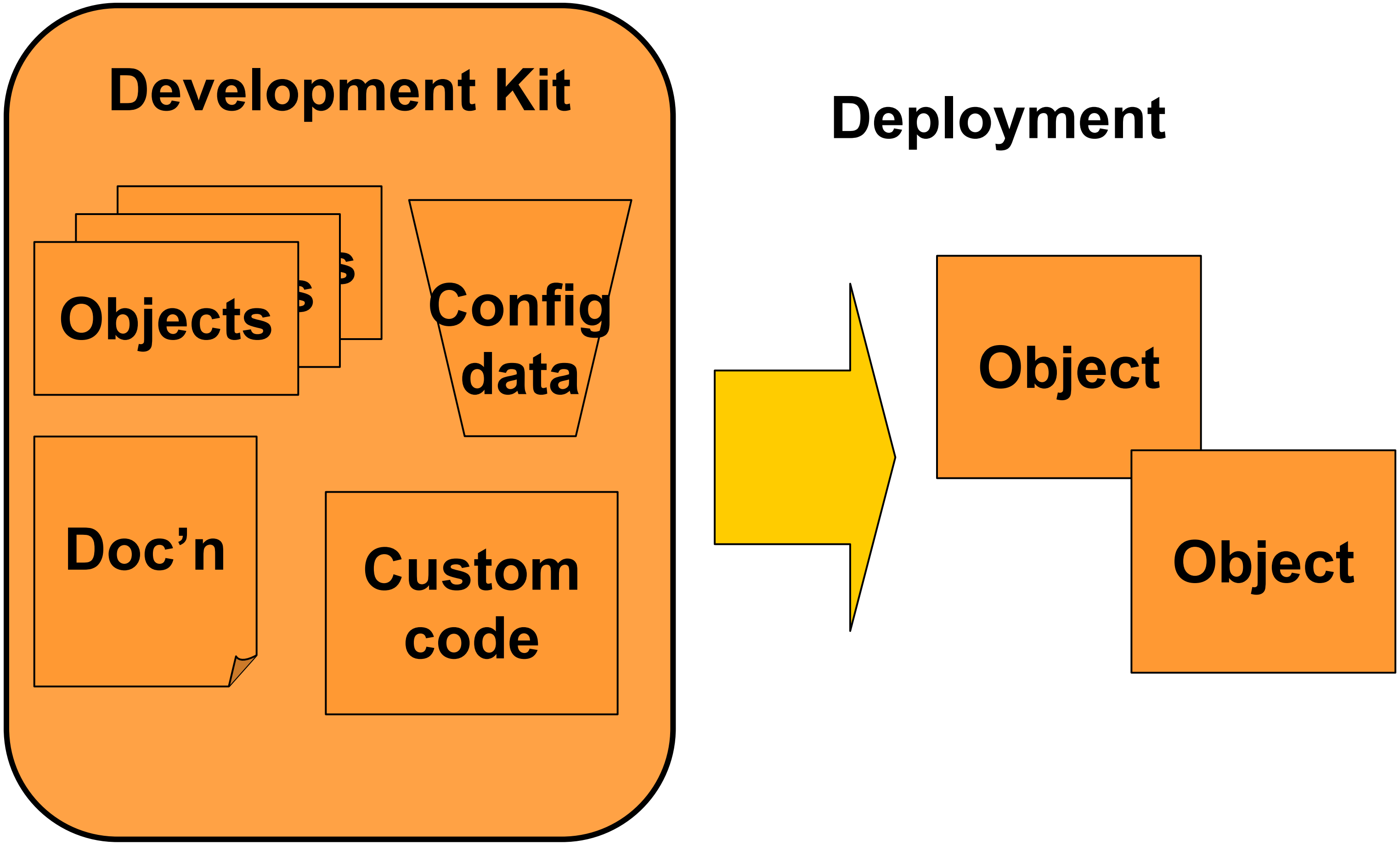
Component-based development

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

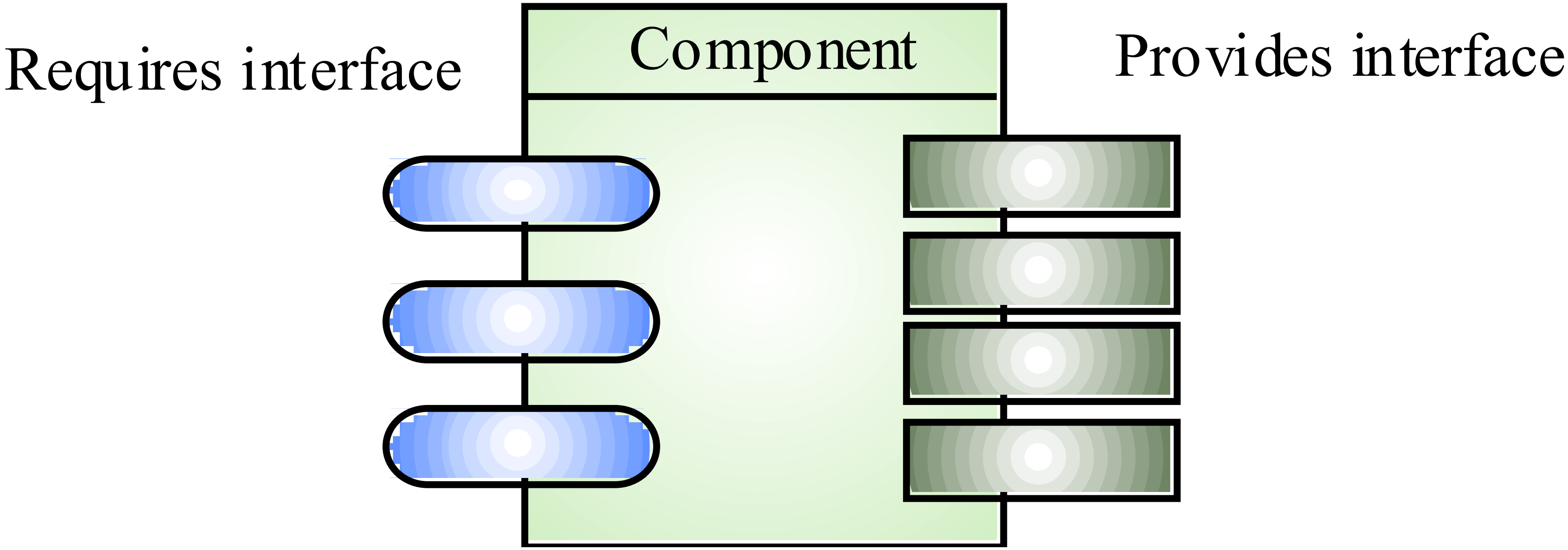
Components

- 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
- Components can range in size from simple functions to entire application systems

Components


Development Kit
Objects
Config data
Doc'n
Custom code
Deployment
Object
Object

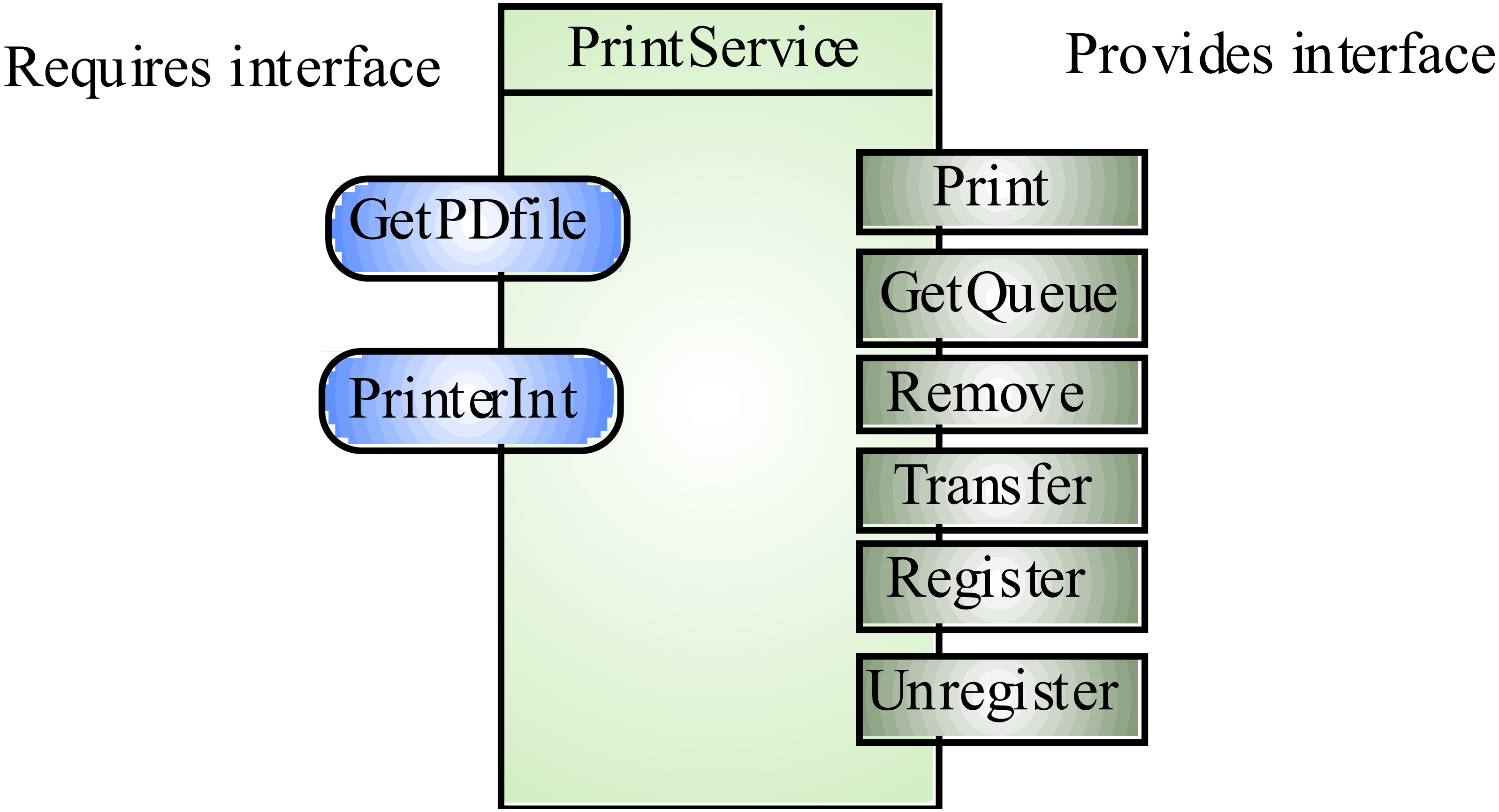
Component interfaces



Component interfaces

- **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**

Printing services component



Component abstractions

- ***Functional abstraction***
  - The component implements a single function such as a mathematical function
- ***Casual groupings***
  - The component is a collection of loosely related entities that might be data declarations, functions, etc.
- ***Data abstractions***
  - The component represents a data abstraction or class in an object-oriented language
- ***Cluster abstractions***
  - The component is a group of related classes that work together
- ***System abstraction***
  - The component is an entire self-contained system

Application frameworks

- Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them
- The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework
- Frameworks are moderately large entities that can be reused

Framework classes

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

Extending frameworks

- Frameworks are generic and are extended to create a more specific application or sub-system
- Extending the framework involves
  - Adding concrete classes that inherit operations from abstract classes in the framework
  - Adding methods that are called in response to events that are recognised by the framework
- Problem with frameworks is their complexity and the time it takes to use them effectively

COTS product reuse

- COTS - Commercial Off-The-Shelf systems
- 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

COTS system integration problems

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

Component development for reuse

- Components for reuse may be specially constructed by generalising existing components
- 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

Reusable components

- The development cost of reusable components is higher than the cost 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 execution times than their specific equivalents

Levels of Reusability

Applications (APIs, automation languages)

Application Frameworks (component groups)

Components (configurable objects)

Objects (properties, methods)

Function Libraries (sqrt(), rand())

Programming Language (‘C’, Pascal etc.)

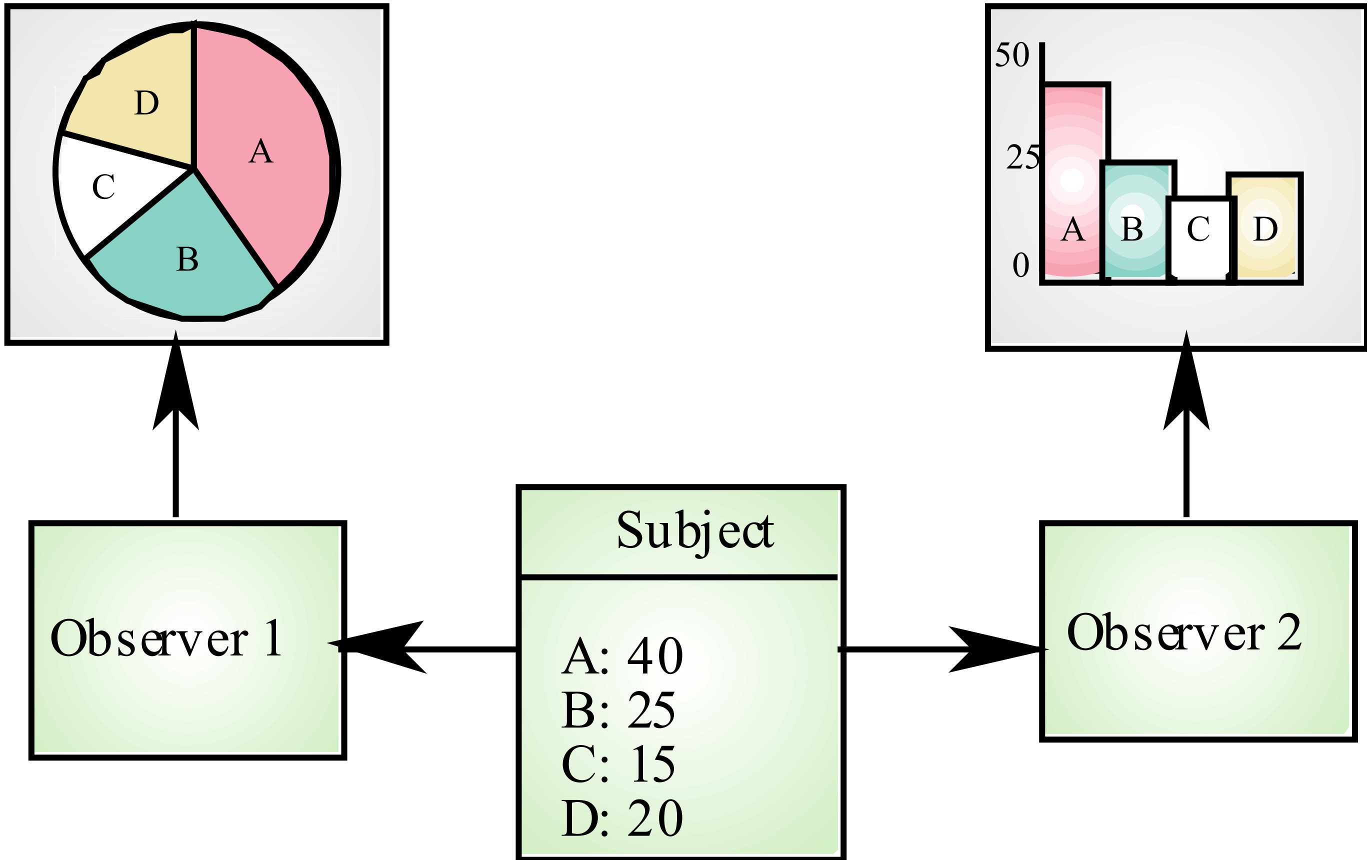
Design patterns

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

Pattern elements

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

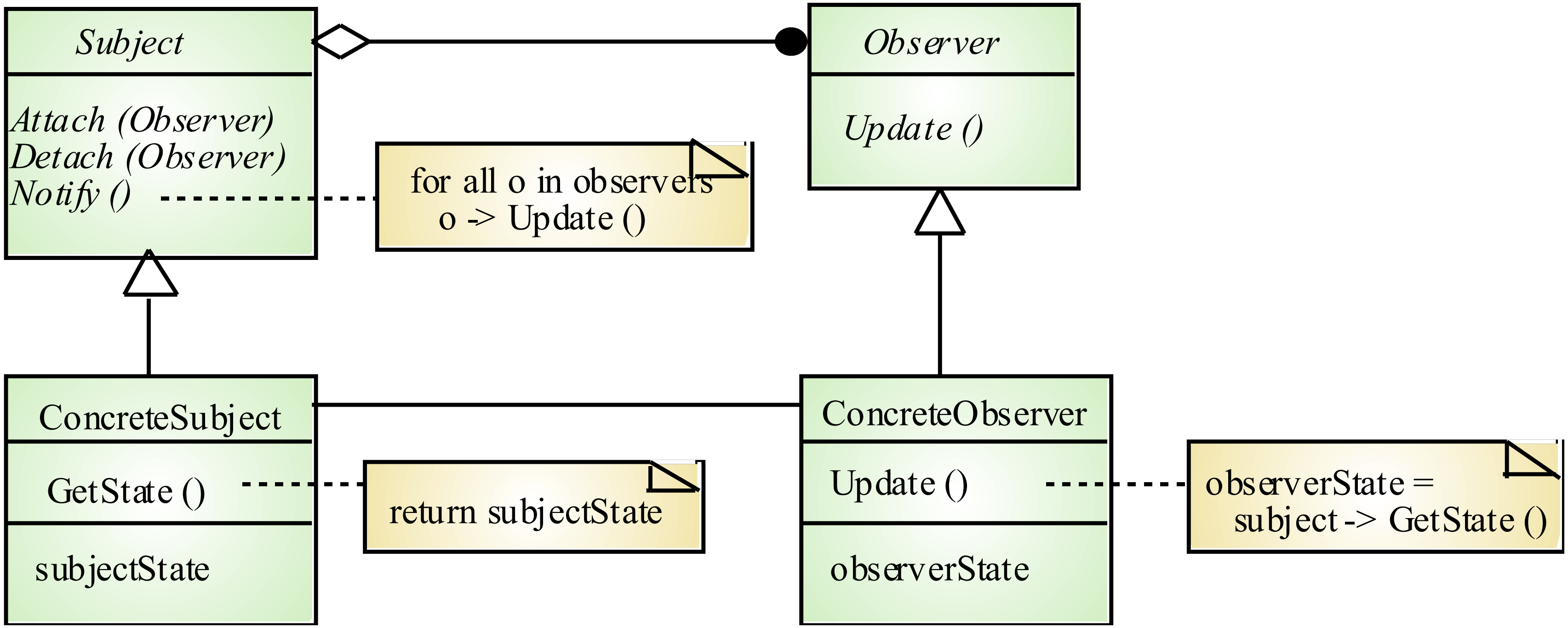
Multiple displays



The Observer pattern

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

The Observer pattern



Key points

- Design with reuse involves designing software around good design and existing components
- Advantages are lower costs, faster software development and lower risks
- Component-based software engineering relies on black-box components with defined requires and provides interfaces
- COTS product reuse is concerned with the reuse of large, off-the-shelf systems

Key points

- Software components for reuse should be independent, should reflect stable domain abstractions and should provide access to state through interface operations
- Design patterns are high-level abstractions that document successful design solutions