CITS4401 Software Requirements and Design
System Design Process

Lecturer: Arran Stewart

Design process cont'd

System design process

- We've said previously that the design process helps us go from a *requirements* model of our system, to *implementation*, in a managed way.
- It is a creative process concerned with how the system will be implemented, and its activities include *architecture design* (deciding on high-level structure and behaviour), *interface design* and *class design*.
- The design we end up with should describe the *architecture* or *structure* of the system, its behaviour, and the classes and algorithms needed to implement the system requirements.

System design process

In general, design is more "messy" a process than analysis.

Why is design difficult?

- Analysis: Focuses on the application domain
- Design: Focuses on the solution domain
  - Design knowledge is a moving target
  - The reasons for design decisions are changing very rapidly

System design process

- Analysis depends on the problem domain.
- When doing design, we add the implementation domain (i.e. hardware, aspects of the language being used, the computers being run on e3tc)
- We worry about how to map the application domain into the existing hardware.

System design process

If someone asks us, "What *is* the design of the system?" – it is the decisions (hopefully documented) that we have made about:

- what the subsystems are
- how they are to be implemented (in hardware vs software, using off-the-shelf components), etc
- how data is managed
- what the access control policy is
- what the system assumes about external systems/users/boundaries.

Design models

- We can think of the design as consisting of a set of increasingly refined/detailed *design models* of the system, that record the decisions we have made.
- Different parts of the design may be specified texctually, or using graphical notations such as UML diagrams (class diagrams, sequence diagrams, state charts, etc).
- Program description languages or pseudocode may be employed to define the algorithms and data structures used.

Design models

Many of the same tools we used in analysis, will be useful in design – but now we are focused on constructing a *solution* to a problem (not just modeling the problem).

By way of example:

- We used class diagrams to represent analysis entities and the relationship between them
- We can use class diagrams again in our design; except now, we aim to represent classes that we will actually *implement*. These will likely be different from the classes we came up with in analysis – additional classes are usually needed.

System design models

- These design models describe the *architecture* or *structure* of the system, its behaviour, and the classes and algorithms required to implement the system requirements.

Making uses of our analysis

- Nonfunctional requirements →
  - Design Goals Definition
- Functional model →
  - System decomposition (Selection of subsystems based on functional requirements, cohesion, and coupling)
- Object model →
  - Hardware/software mapping
  - Persistent data management
- Dynamic model →
  - Concurrency
  - Global resource handling
  - Software control
- Subsystem Decomposition
  - Boundary conditions

Design patterns

- One way we manage complexity in design is by looking for opportunities to re-use *design patterns* – the next topic we look at.

Design patterns

Outline

- What are Design Patterns?
  - A design pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the this solution a million times over, without ever doing it the same twice
- Design Patterns
  - Usefulness of design patterns
  - Design Pattern Categories
- Patterns covered in this lecture
  - Facade: Unifying the interface to a subsystem.
  - Adapter: Interfacing to existing systems (legacy systems)
  - Bridge: Interfacing to existing and future systems

Why Use Design Patterns

- Reuse: Once a design pattern has been verified, it can be used in any number of ways in a given context.
- Common Vocabulary: Design patterns give software designers a common vocabulary that concisely encapsulates solutions to design problems.
- Easy to modify: Designs patterns are easy to modify to apply to a particular problem. The solutions can also be modified to give flexibility with minimal risk.

Elements of a Pattern

- The Pattern Name encapsulates a well known solution to a design problem, and increases our design vocabulary.
- The Problem describes when to apply the pattern. It gives the context of the pattern, and possibly some pre-conditions to ensure the pattern will be effective.
- The Solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution is a template, that can be modified to apply to range of situations.
- The Consequences are the results and trade-offs of applying the pattern. These are critical for evaluating the costs and benefits of applying a pattern.
- [Gamma et al 95]

Reuse

- Main goal:
  - Reuse knowledge from previous experience to current problem
  - Reuse functionality which is already available
- Composition (also called Black Box Reuse)
  - New functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing components
- Inheritance (also called White-Box Reuse)
  - New functionality is obtained by inheritance.
- Three ways to get new functionality:
  - Implementation inheritance
  - Interface inheritance
  - Delegation

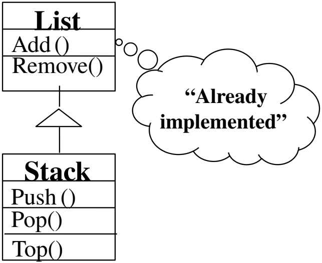
Implementation Inheritance vs Interface Inheritance

- Implementation inheritance
  - Also called class inheritance
  - Goal: Extend an applications' functionality by reusing functionality in parent class
  - Inherit from an existing class with some or all operations already implemented
- Interface inheritance
  - Also called subtyping
  - Inherit from an abstract class with all operations specified, but not yet implemented

Implementation Inheritance

A very similar class is already implemented that does almost the same as the desired class implementation.

- Example: I have a **List** class, I need a **Stack** class. How about sub-classing the **Stack** class from the **List** class and providing three methods, **Push()** and **Pop(), Top()**?



- Problem with implementation inheritance:

  Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?

Delegation

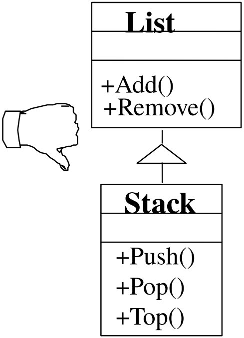
- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow the client to misuse the delegate object

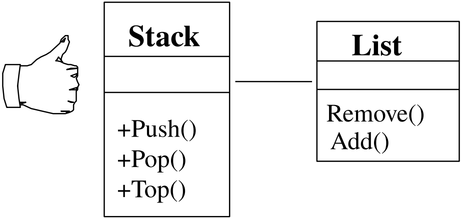
Delegation or Inheritance?

- Delegation
  - Pro:
    - Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  - Con:
    - Inefficiency: Objects are encapsulated.
- Inheritance
  - Pro:
    - Straightforward to use
    - Supported by many programming languages
    - Easy to implement new functionality
  - Con:
    - Inheritance exposes the details of a parent class to its subclasses
    - Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)

Delegation instead of Inheritance

Delegation: Catching an operation and sending it to another object.





Towards a Pattern Taxonomy

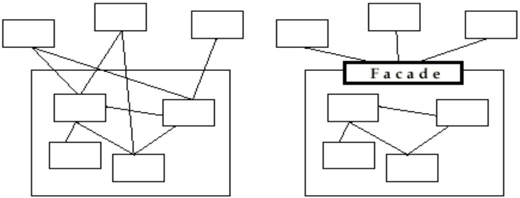
- Structural Patterns
  - Adapters, Bridges, Façades and Proxies are variations on a single theme:
    - They reduce the coupling between two or more classes
    - They introduce an abstract class to enable future extensions
    - Encapsulate complex structures
- Behavioural Patterns
  - Concerned with algorithms and the assignment of responsibilities between objects: Who does what?
  - Characterize complex control flow that is difficult to follow at runtime.
- Creational Patterns
  - Abstract the instantiation process.
  - Make a system independent from the way its objects are created, composed and represented.

Structural patterns: Façade, Adapter, Bridge

- A subsystem consists of
  - an interface object
  - a set of application domain objects (entity objects) modeling real entities or existing systems
    - Some of the application domain objects are interfaces to existing systems
  - one or more control objects
- Realization of Interface Object: Facade
  - Provides the interface to the subsystem
- Interface to existing systems: Adapter or Bridge
  - Provides the interface to existing system (legacy system)
  - The existing system is not necessarily object-oriented!

Facade Pattern

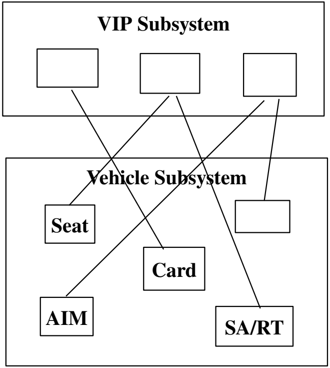
- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



Open vs Closed Architecture

- Open architecture:
  - Any client can see into the vehicle subsystem and call on any component or class operation at will.
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can't expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
  - We can be assured that the subsystem will be misused, leading to non-portable code

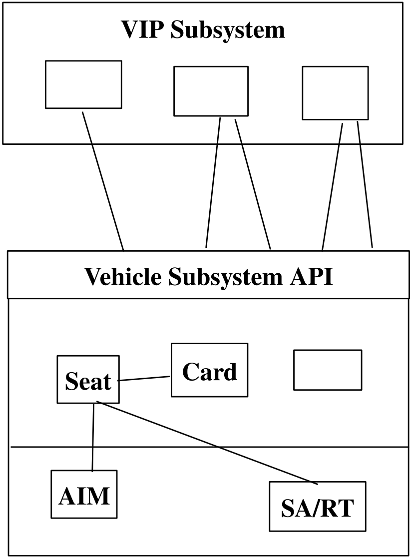
Open vs Closed Architecture



Realizing a Closed Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- The subsystem components can still be accessed directly.
- If a façade is used the subsystem can be used in an early integration test
  - We need to write only a driver

Realizing a Closed Architecture with a Facade

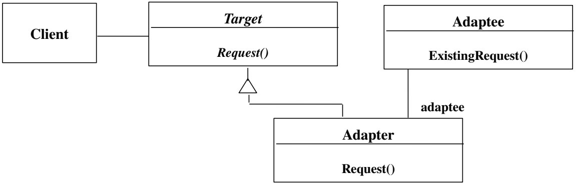


Adapter Pattern

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper
- Two adapter patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter:
    - Uses single inheritance and delegation
- We will mostly use object adapters and call them simply adapters

Adapter pattern

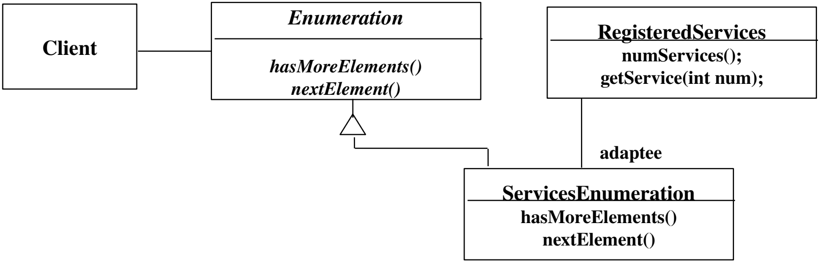
- Delegation is used to bind an Adapter and an Adaptee
- Interface inheritance is use to specify the interface of the Adapter class.
- Target and Adaptee (usually called legacy system) pre-exist the Adapter.



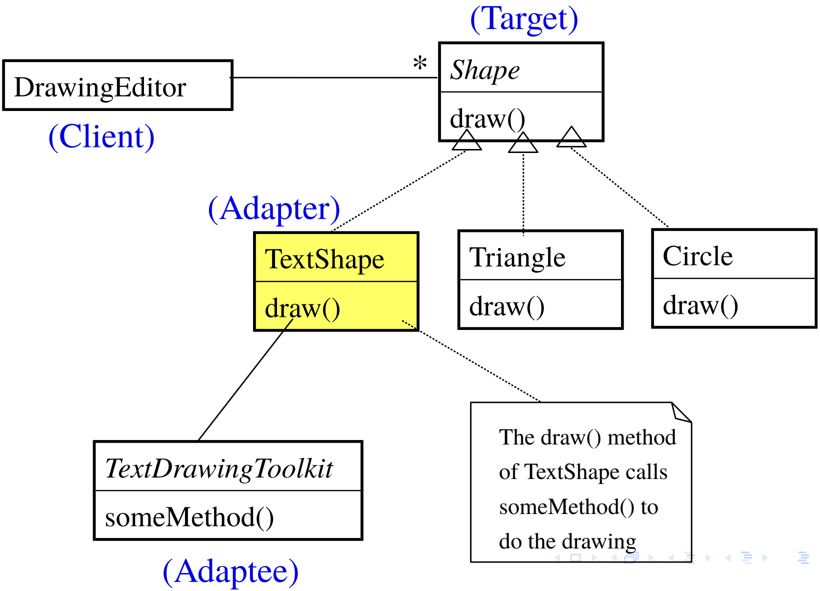
Adapter pattern example

```
public class ServicesEnumeration implements Enumeration {
  private RegisteredServices adaptee;
  public boolean hasMoreElements() {
    return this.currentServiceIdx <= adaptee.numServices();
  }
  public Object nextElement() {
    if (!this.hasMoreElements()) {
    throw new NoSuchElementException();
    }
    return adaptee.getService(this.currentSerrviceIdx++);
  }
  //...
```

Adapter pattern example



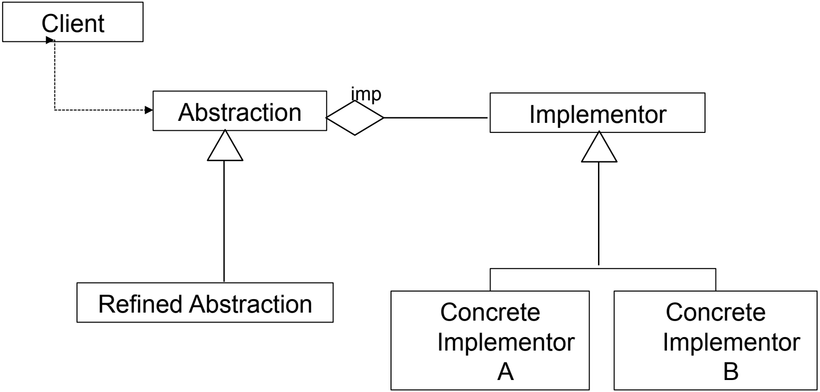
Adapter pattern example



Adapter vs Bridge

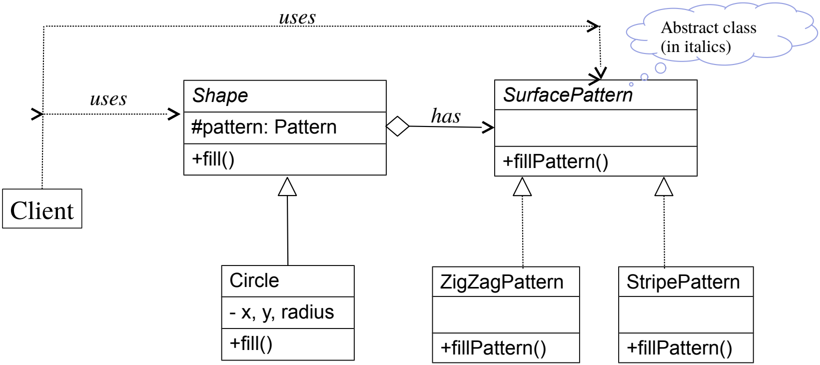
- Similarities:
  - Both used to hide the details of the underlying implementation.
- Difference:
  - The adapter pattern is geared towards making unrelated components work together
    - Applied to systems after they're designed (reengineering, interface engineering).
  - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
    - Green field engineering of an "extensible system"
    - New "beasts" can be added to the "object zoo", even if these are not known at analysis or system design time.

Bridge Pattern



Bridge Pattern – reference B&D Appendix A.3

Bridge Pattern – Example



Note: means “extends” means “implements”

Design Patterns encourage good Design Practice

- A facade pattern should be used by all subsystems in a software system. The façade defines all the services of the subsystem.
  - The facade will delegate requests to the appropriate components within the subsystem.
- Adapters should be used to interface to any existing proprietary components.
  - For example, a smart card software system should provide an adapter for a particular smart card reader and other hardware that it controls and queries.
- Bridges should be used to interface to a set of objects where the full set is not completely known at analysis or design time.
  - Bridges should be used when the subsystem must be extended later (extensibility).

Why are modifiable designs important?

- A modifiable design. . .
- . . . enables an iterative and incremental development cycle
  - concurrent development
  - risk management
  - flexibility to change
- . . . minimizes the introduction of new problems when fixing old ones
- . . . enables ability to deliver more functionality after initial delivery

What makes a design modifiable?

- Low coupling and high coherence
- Clear dependencies
- Explicit assumptions
- How do design patterns help?
- They are generalized from existing systems
- They provide a shared vocabulary to designers
- They provide examples of modifiable designs
  - Abstract classes
  - Delegation

More Design Patterns!

- Structural pattern
  - Façade, Adapter, Bridge
  - Proxy – creates a stand-in for an object that is costly to access.
- Behavioral pattern
  - Observer – coordinates several views of a single object.
- Creational Patterns
  - Abstract Factory – initializes objects independently from the client.