Software Design
Software Architecture

Outline

- What is design
- What is a software architecture
- Design in the small
  - modules
  - mechanisms
- Recurrent schemes (design patterns)

Part 1

- Components-off-the-shelf (COTS) and component models
- Design styles
- Abstract design structures: components and connectors
- Design examples
- Distributed architectures and middleware

Part 2

Design and architecture

- Sometimes used as synonyms
  - The design activity produces the software architecture (or software design)
- Sometimes architecture used as a higher level concept than design
- Although of course there is a continuum of abstraction/detail

A possible distinction

- Design
  - decomposition of system into modules
  - relations/interactions among modules
    - which uses which
    - which inherits from which
- Architecture
  - higher level description of logical components and how they interact
    - *The architecture of a software system defines the system in terms of computational components and interactions among those components. (Garlan&Shaw1996)*

Components and interactions

- Can be defined at different levels of abstractions
- We identify three:
  - level 1: mechanisms
    - what are the constituents and how are they aggregated and related?
  - level 2: recurrent schemes
    - design patterns
  - level 3: styles (in Part 2)

Mechanisms

- What are the modules?
- What is their interface?
- Which are the useful relations among modules?

Method issue
  - What are the criteria to decompose systems into modules?

Language issue
  - How to represent/implement?

Documentation
  - How to document?

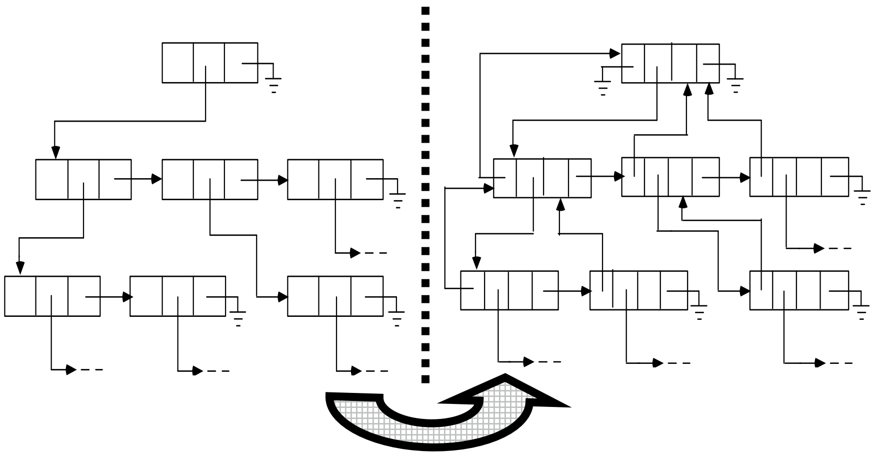
Design criteria

- Design for change (Parnas)
  - designers tend to concentrate on current needs
  - special effort needed to anticipate likely changes
- Product families (Parnas)
  - think of the current system under design as a member of a program family

Sample likely changes? (1)

- Algorithms
  - e.g., replace inefficient sorting algorithm with a more efficient one
- Change of data representation
  - e.g., from binary tree to a threaded tree (see example)
  - ≈17% of maintenance costs attributed to data representation changes (Lientz and Swanson, 1980)

Example



9

Sample likely changes? (2)

- Change of underlying abstract machine
  - new release of operating system
  - new optimizing compiler
  - new version of DBMS
  - …
- Change of peripheral devices
- Change of "social" environment
  - new tax regime
  - EURO vs national currency in EU
- Change due to development process (transform prototype into product)

10

Product families

- Different versions of the same system
  - e.g. a family of mobile phones
    - members of the family may differ in network standards, end-user interaction languages, …
  - e.g. a facility reservation system
    - for hotels: reserve rooms, restaurant, conference space, …, equipment (video beamers, overhead projectors, …)
    - for a university
      - many functionalities are similar, some are different (e.g., facilities may be free of charge or not)

11

Design goal for family

- Design the whole family as one system, not each individual member of the family separately

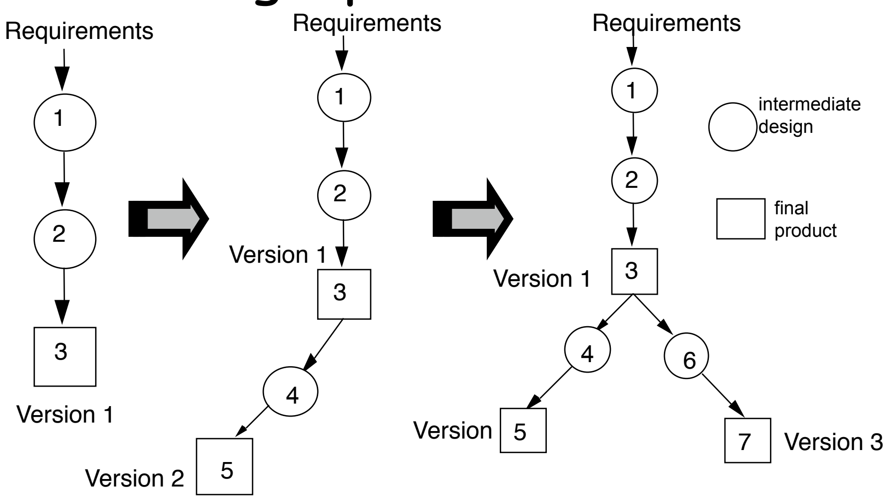
12

Sequential completion: the wrong way

- Design first member of product family
- Modify existing software to get next member products

13

Sequential completion: a graphical view



14

How to do better

- Anticipate definition of all family members
- Identify what is common to all family members, delay decisions that differentiate among different members

15

Language issues

- Clearly separate interface/specification from body/implementation
  - stable vs. changeable part
- Factorize common behaviors and use inheritance
- *These are the main features of object-oriented languages*

Documentation

- Modules
  - pre, post-conditions, invariants
    - may be part of the language
- Associations
  - use relation
  - inheritance
    - UML may be used as a notation

Advanced language issues

- All languages support some level of grouping and encapsulation of concerns into separate, independent entities by providing abstractions (procedures, modules, classes, methods) that can be used to represent these concerns
- Sometimes traditional modularization does not work
- Cross-cutting concerns defy these forms of encapsulation
  - Logging is the archetypal example of a crosscutting concern because a logging strategy necessarily affects every single logged part of the system. Logging thereby crosscuts all logged classes and methods

Aspect-oriented programming

- AOP supports encapsulation of each concern in one place (aspect)
- AspectJ does this for Java
- An aspect can alter the behavior of the base code (the non-aspect part of a program) by applying **advice** (additional behavior) at various **join points** (points in a program) specified in a quantification or query called a **pointcut** (that detects whether a given join point matches)

Recurrent schemes

Design patterns

What are they?

- Design patterns represent OO programming knowledge
- They are reusable
- They define a lexicon

How can their goals be defined?

- By providing answers to questions regarding:
  - Intent
    - **what** do you want to achieve by using it?
  - Motivation
    - **why** is it useful?
  - Applicability
    - **when** is it useful? **how** to use it?

Design patterns in software design

- They support design **in the small**
- They make the design of individual modules and module interactions systematic

Where do they come from?

- Patterns originated as an architectural concept by C. Alexander
- Design patterns gained popularity after the book "Design Patterns: Elements of Reusable Object-Oriented Software" (1995) by Erich. Gamma, R. Helm, Ralph. Johnson and Joh Vlissides, known as the "Gang Of Four"

Potential benefits

- Speed up the development process by reusing tested, proven development paradigms (knowledge consolidation)
- Provide a common language to developers to communicate
- Provide a baseline for improvement over time, making them more robust than ad-hoc designs

Pattern categories

- **Creational**
  - provide mechanisms for creating objects
- **Structural**
  - provide simple ways to realize relationships between entities and compositions
- **Behavioral**
  - provide common communication patterns between objects

Catalogue of design pattern

Purpose		
Creational	**Structural**	**Behavioral**
Factory	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Façade	Iterator
	Flyweight	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

Creational patterns

Deal with the problem of creating objects (products) without specifying the exact class of object that will be created

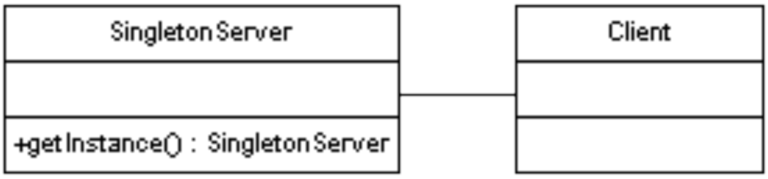
Singleton pattern: goals

- Intent
  - ensure that a class only has one instance
- Motivation
  - through it we deny direct access to contructor by providing a global point of access to instance
- Applicability
  - single instance property is part of the requirements
  - singleton encapsulates the single instance

Examples

- Java Security manager
  - All parts of a program must access the same security manager
  - Once set a security manager cannot be changed in a program
- A logger for the activity of a server
  - All parts of the server should use the same instance of the logging system
  - The server should not be able to change the instance of the logging system after it has been set

Singleton pattern



In Java

make constructor private

getInstance static method

Java implementation

```
public class Singleton {
   private static Singleton INSTANCE = null;
   // constructor not visible
   private Singleton() {}
   //synchronized creator to defend against
   //multi-threading issues
   private synchronized static void createInstance() {
   //check here to avoid multiple instantiation
     if (INSTANCE == null) {
         INSTANCE = new Singleton();
      }
   }
   public static Singleton getInstance() {
      if (INSTANCE == null) createInstance();
      return INSTANCE;
   }
}
```

the only exported method

A more concise Java implementation

```
public class Singleton {
        private Singleton() {...}
        final private static Singleton instance
                = new Singleton();
        public static Singleton getInstance()
                { return instance; }
}
```

final static field

Java semantics guarantees that object (instance) is created before first use

Factory method pattern: goals

- Easily recognized and frequently used
- Defines objects that are responsible for delivering other objects
  - client ignores the exact class of object that will be delivered
    - this is why we can't use the constructor
- Class of object to deliver may be unknown at compile-time: only interface is known (class becomes known at run-time)

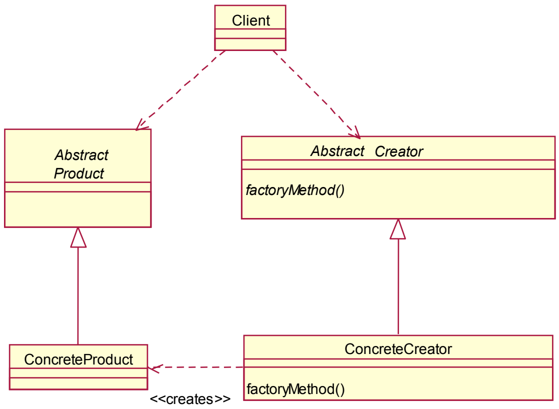
Possible uses

- We enter a bar and we ask for a beer
  - we do not know which kind of beer will be given to us
  - we only care about the abstract features
- We need a visual device to provide an input
  - if boolean, a checkbox
  - if text, a textbox
  - if small integer, a multiple choice list

Factory method pattern: the solution

- Creates an object that conforms to an abstract interface, separating the responsibility of creating the object from the client which uses the object
- The factory implementation decides which class has to be instantiated
- This is achieved by exploiting polymorphism

Factory method: class diagram



Factory: participants

- **Product**
  - defines the interface of objects created by factory method
- **ConcreteProduct**
  - implements the Product interface
- **Creator** (abstract class)
  - declares the factory method, which yields a Product
  - may define a default implementation of the factory method which yields a default ConcreteProduct
- **ConcreteCreator**
  - overrides the factory method to yield an instance of a specific ConcreteProduct

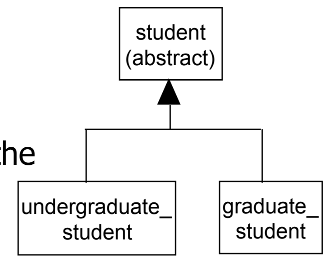
Example 1

- The concrete factory pre-allocates a pool of objects (concrete products)
- The client invokes the factory to get an instance
  - The factory has its own policy to decide what to do if all instances have been allocated

Example 2

There is a hierarchy of products

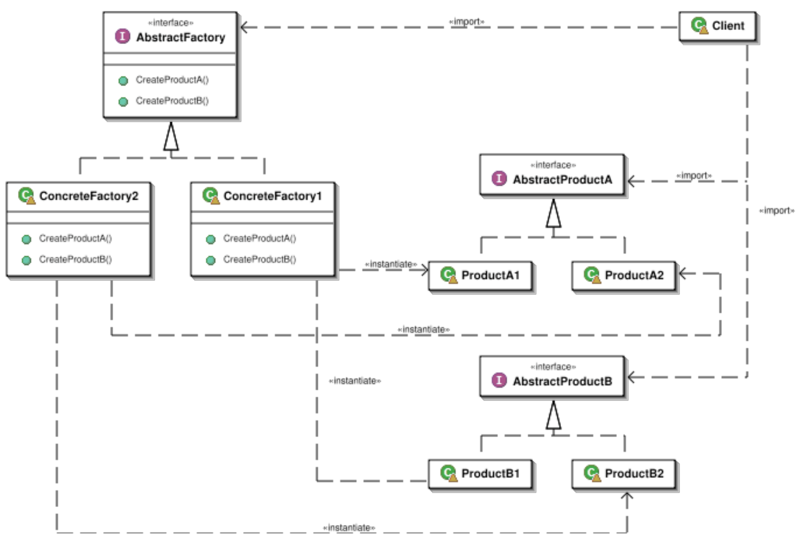
- a parameter allows the Factory method to determine which of the concrete products should be created
  - e.g. parameter is student ID number, and depending on it either an undergraduate or a graduate student descriptor is returned



Abstract Factory Pattern

- Encapsulates a group of individual factories that have a common theme
  - abstract factory class *DocumentCreator* provides interfaces to create a number of products
    - *createLetter()*
    - *createResume()*)
  - there are any number of derived concrete versions of *DocumentCreator*
    - *FancyDocumentCreator*
    - *ModernDocumentCreator*

  each with a different implementation of *createLetter()* and *createResume()* to create a *FancyLetter* or a *ModernResume*



Structural patterns

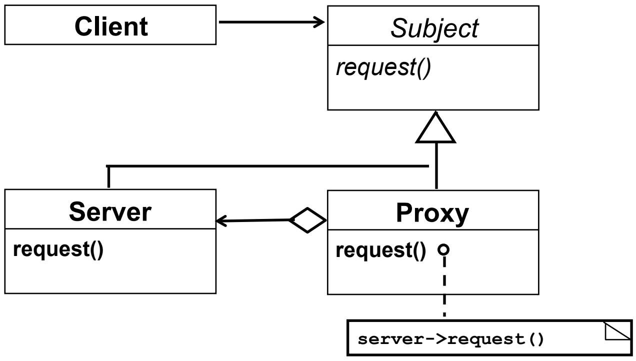
Proxy, Decorator, Façade, Adapter

Proxy

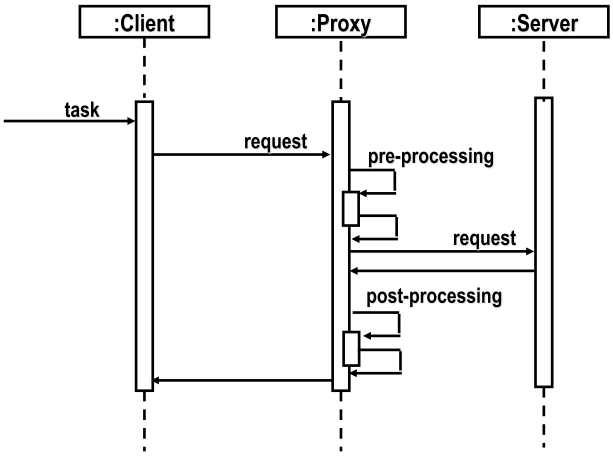
- Intent
  - multiple clients access the services offered by a server but not through direct access
- Motivation
  - avoid direct access, achieve location transparency, guarantee security and efficiency
- Applicability
  - an intermediary which may perform some pre- e post-processing (access control, caching, …)

in its most general form, a class functioning as an interface to another thing

Proxy pattern: class diagram



Proxy: dynamic behavior



Proxy examples

- Remote Proxy
  - Provides a reference to an object located in a different address space on the same or different machine
- Cache Proxy
  - Provides temporary storage of the results of expensive target operations so that multiple clients can share the results
- Protection (Access) Proxy
  - Provides different clients with different levels of access to a target object
- Smart Reference Proxy
  - Provides additional actions whenever a target object is referenced such as counting the number of references to the object
- Virtual Proxy
  - Buffers sequences of updates (e.g., insert/delete) without actually changing the target object
- Lazy-clone Proxy
  - Defers copying (cloning) a target object until required by client actions

Proxy examples

- OMG-CORBA, Java RMI
  - remote object communication based on "proxy"
- HTTP Proxy
  - intermediary between web browser and http server
  - frequently accessed documents are cached
- Access control (firewall)

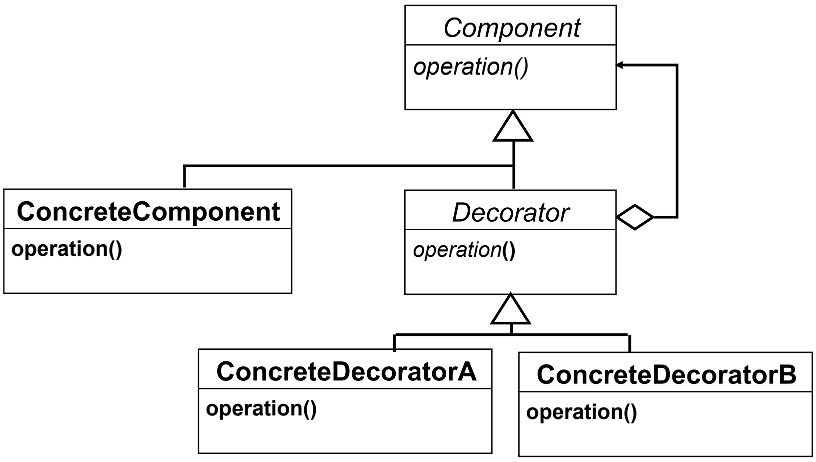
Decorator

- Intent
  - extend functionality of an object, attach additional responsibilities dynamically
    - example: a text box also has a scrollbar
- Motivation
  - we wish this extension to be defined dynamically, during execution, for each individual object
- Applicability
  - encapsulate the main object (e.g., the text box) into another called decorator (e.g., the scrollbar) which adds the additional responsibilities/capabilities

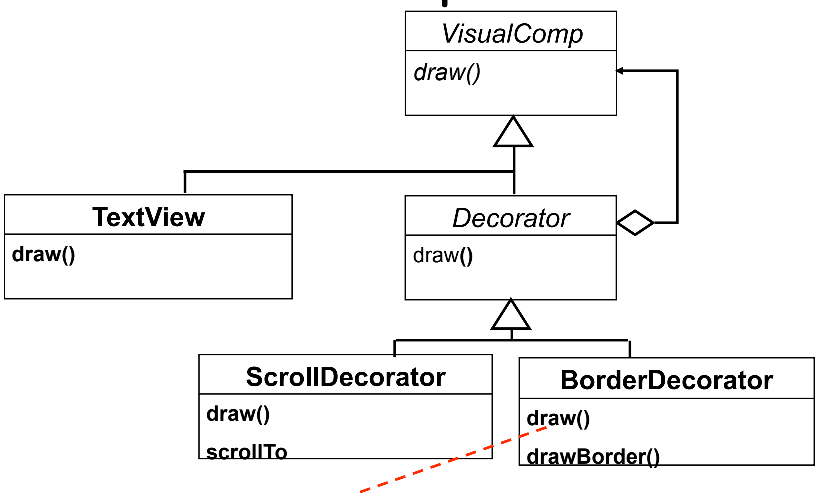
Why not a subclass?

- Subclasses add behavior at compile time whereas decorators provide a new behavior at runtime
  - typically achieved by passing the original object as a parameter to the constructor of the decorator, with the decorator implementing the new functionality

Decorator: class diagram

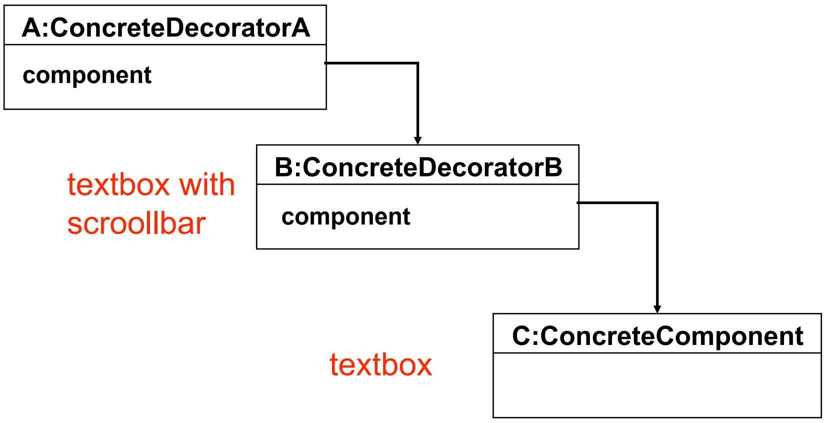


Example



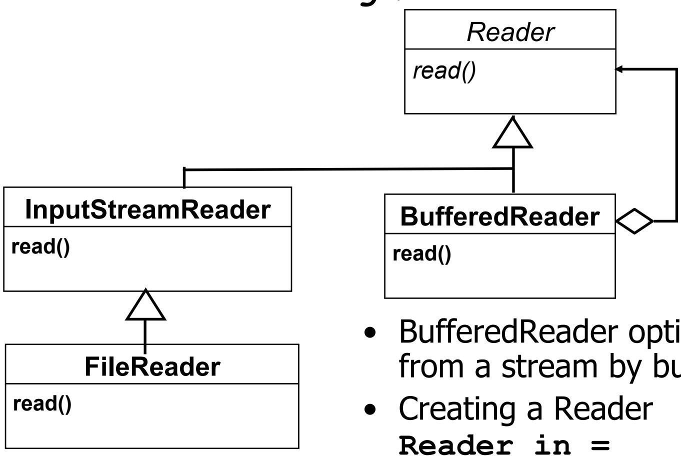
The draw of a BorderDecorator draws the internal visual component (which may be a text) and then the border

Decorator: object diagram



Decorator example in Java:

reading from a stream



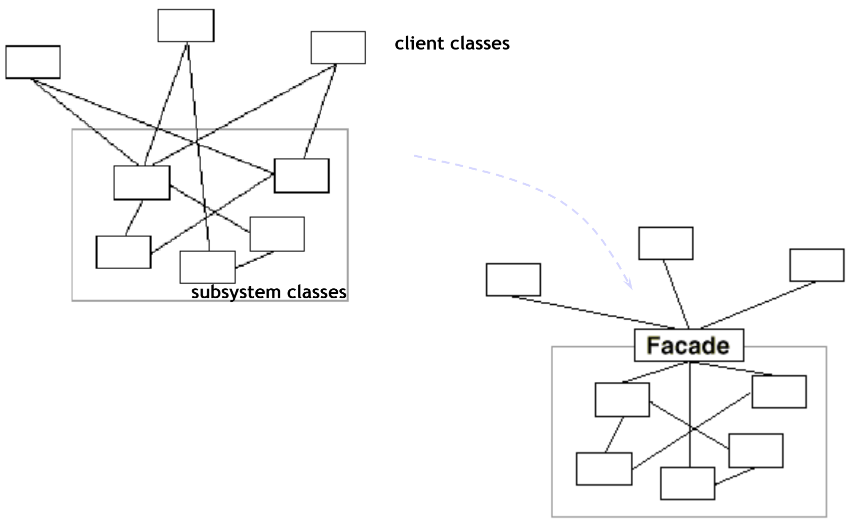
- BufferedReader optimizes input from a stream by buffering data
- Creating a Reader

```
Reader in =
    new BufferedReader(
      new
FileReader("foo.in"));
```

Façade

- Intent
  - Provide a simple interface to complex composites of modules
- Motivation
  - Decouple from the complexity of the composite
  - Offer a layering mechanism
- Applicability
  - Group together components through a class that represents the external "façade"

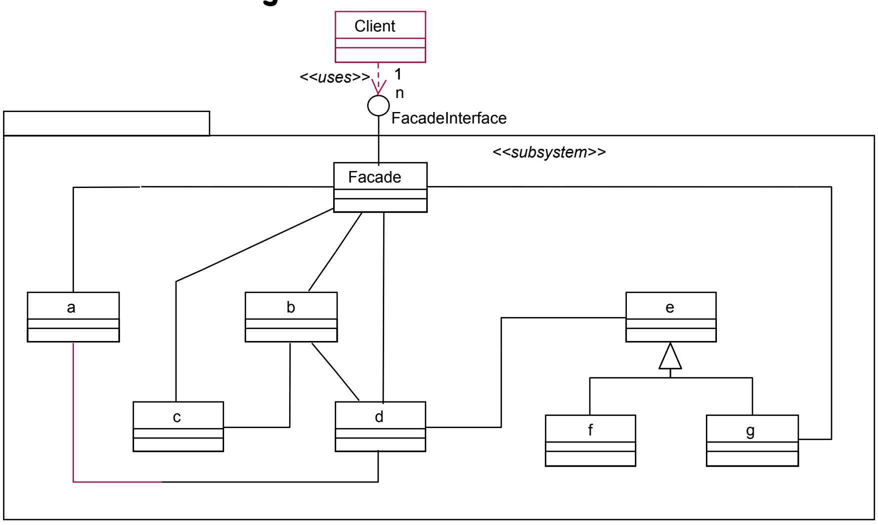
Façade: example



Façade : concrete example

- Suppose there are modules that
  - parse different languages
  - generate code from different sources
  - perform code optimization
  - support execution in debugging mode
- Instead of accessing the individual modules, a façade activates the modules based on a command string that defines what is needed by the user

Façade: structure



Adapter: why?

- Sometimes clients and services have incompatible interfaces, but need to work together
  - service offers an interface, client expects another
  - thus a class that was designed to be reused in practice cannot because its interface is not compatible with the interface required by the client
- Adapter converts the interface the service offers to the interface the client expects

Adapter: why?

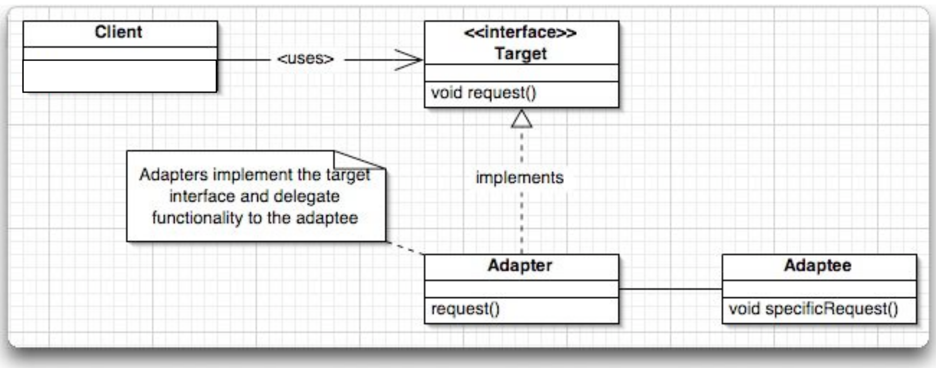
- It may be useful to adapt an interface to another, to obtain a uniform abstraction of possibly different interfaces
  - Allows different classes that have incompatible interfaces to work together
- It may be useful to adapt to several subclasses by providing an adaptor for the common ancestor

Adapter: how?

- The solution consists of *wrapping* the adaptee through a new interface
  - (differs from proxy because here the interface is different)

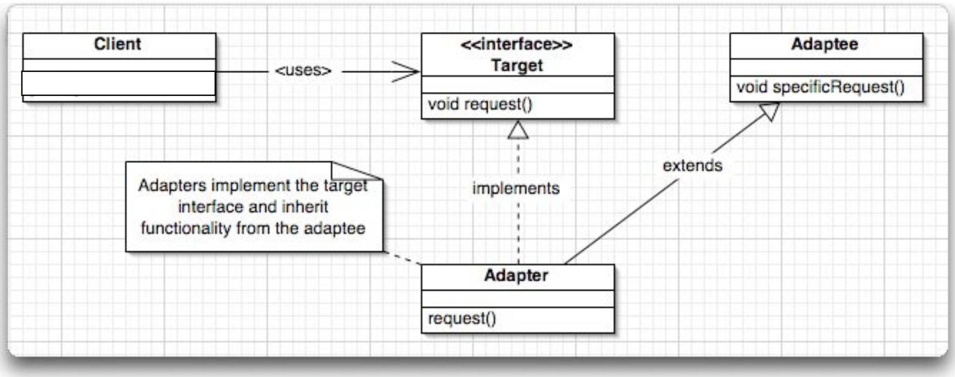
Adapter: class diagram (1)

exploits *delegation*
adapter contains an
instance of the class it wraps



Adapter: class diagram (2)

exploits *inheritance*
adapter inherits from
adaptee



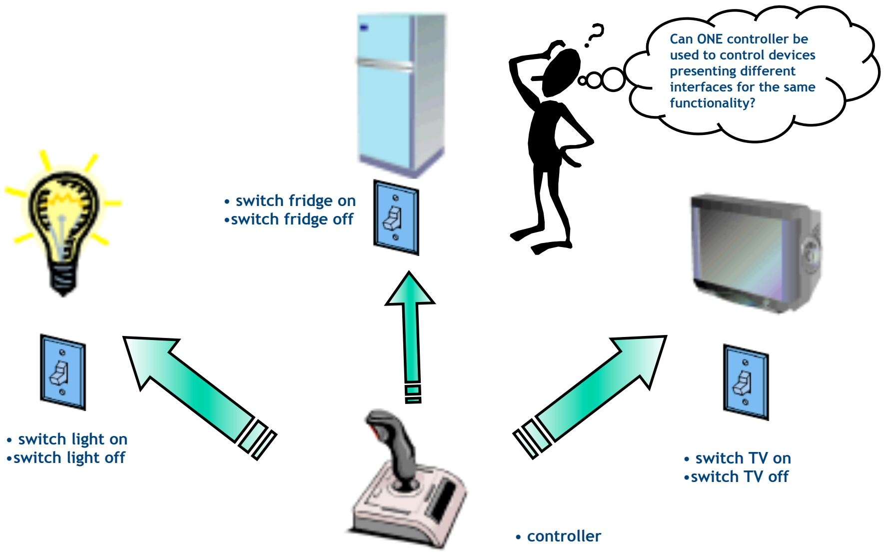
Adapter: Java implementation

```
interface Target {
  public void request();
}
public class Adaptee {
  public void specificRequest() {...}
}
public class InheritanceAdapter extends Adaptee
  implements Target {
  public void request() {
     specificRequest();
  }
}
public class Adapter implements Target {
  private Adaptee adaptee;
  Adapter(Adaptee a) {adaptee = a;}
  public void request() {
    adaptee.specificRequest();
  }
}
```

Adapter: comparison of approaches

- Inheritance:
  - Adaptee is adapted to Target through a concrete class InheritanceAdapter
    - Not possible to adapt all subclasses of a class if the language does not support multiple inheritance
  - Adapter can override the behavior of Adaptee, since it is a subclass
  - Only one object is introduced + no other indirect references needed to access the adapted object
- Delegation:
  - a single Adapter can work with an Adaptee and all its subclasses
    - Adapter may add functionality to all Adaptees
  - one more object is introduced

Problem: controlling heterogeneous devices



Exercise

How to use the adapter pattern to solve the control problem for heterogeneous devices

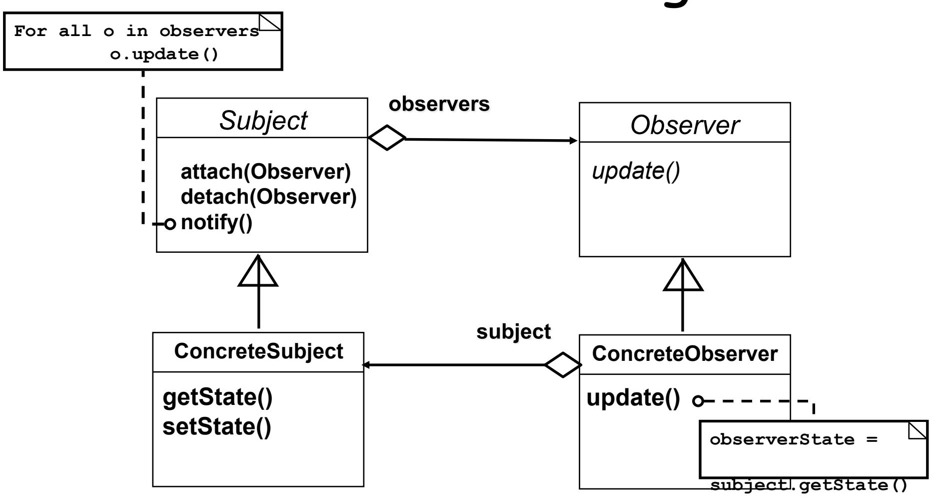
Behavioral patterns

Observer

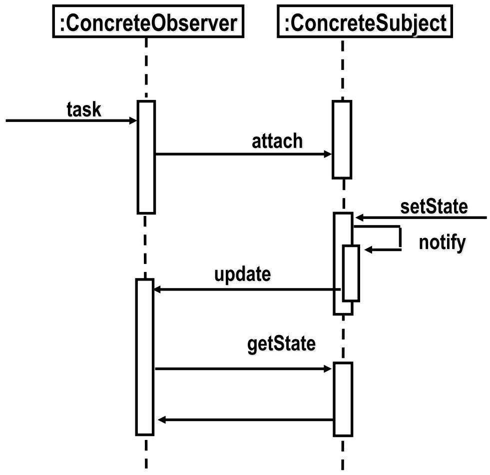
Observer

- ROLES
  - subject (which is observed)
  - observers (or listeners)
- Several objects interested in observing how a subject changes its state
- Subject does not depend on number and kind of observers
  - new observers may be added during execution
  - they declare their interest in state changes

Observer: class diagram



Observer: dynamic behavior



When should you use Observer

- When the subject's changes must be broadcasted to several observers (listeners)
- When subjects just notify observers through a callback, but ignore all of them

☛**JavaBeans** are reusable components implemented via this pattern

Conclusions

- Patterns offer a catalogue of typical "in-the-small" recurring idioms
- By collecting and systematizing them we can expect
  - a common design vocabulary
  - a documentation aid
  - a support in learning object-oriented techniques
  - a guideline in re-engineering existing software

Caveat

- They often introduce redundancies and indirections, which may lead to inefficiencies
- … because to be reusable they add a certain abstraction level
- Therefore they might be avoided in some cases where efficiency is a key attribute

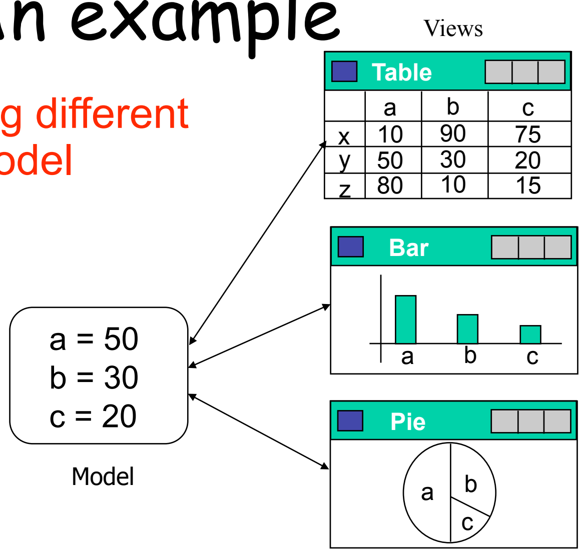
Towards architectural patterns

- Recurring architectural schemes, such as MVC (Model-View-Controller)
- They are at a higher-level than design patterns, which are closer to programming
- MVC separates an *application's data model*, *user interface*, and *control logic* into three distinct parts so that modifications to one part can be made with minimal impact to the others

An example

Synchronizing different views of a model

There are several views on the same raw data. One would like to manipulate each view, and yet keep the model synchronized with the different changes on the different views.



MVC

- Model
  - domain-specific representation of the information on which the application operates
  - domain operations add meaning to raw data e.g. calculating the totals, taxes and shipping charges for shopping cart items
- View
  - Renders the model into a form suitable for interaction, typically a user interface element
    - e.g., the html page and the code which gathers dynamic data for the page
- Controller
  - Responds to events, typically user actions, and invokes changes on the model (but not the view)

Typical control flow

1. User interacts with the GUI in some way (e.g., presses a button)
2. Controller contains input part of the GUI
3. Controller accesses the model, updating it as appropriate to the user's action (e.g., controller updates user's shopping cart)
   Complex controllers often structured using the command pattern to encapsulate actions and simplify extension
4. View uses the model to generate an appropriate user interface (e.g., a screen listing the shopping cart contents)
   observer pattern can be used to provide interaction between model and view, allowing the model to notify interested parties of a change
   view object can register itself with the model and listen for changes but the model itself remains view-agnostic
   controller does not pass domain objects (the model) to the view although it might issue a command telling the view to update itself.
5. User interface waits for further user interactions, which begins the cycle anew

MVC: the strategy

- MVC first described in 1979 in the context of Smalltalk
- Strategy
  - decouple views and models, so that views can be modified independent of model
- How synchronization is achieved
  - via events and listeners (observer pattern)

MVC in Smalltalk

- There are differences from what we said
- In the so-called "passive model", the controller invokes operation on view which, in turn, invokes getState on model to get state changes
- You may check at this link www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

Generalization

- The architectural pattern isolates
  - business logic
  - user interface
  - information (application data)
- 3 tiers

Bibliography

- E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design patterns - Elements of reusable object oriented software*. Addison-Wesley, 1995
- F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A system of patterns - Pattern oriented software architecture*. John Wiley, 1996
- Freely available on the web
  - Bruce Eckel, Thinking in patterns, http://www.mindview.net/Books/TIPatterns/.