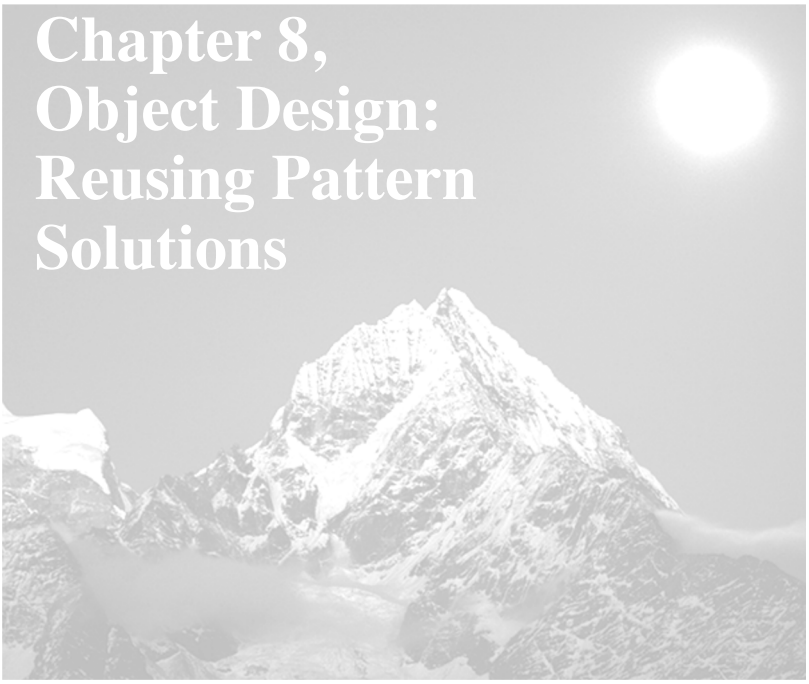
**Object-Oriented Software Engineering**
Using UML, Patterns, and Java

# Chapter 8, Object Design: Reusing Pattern Solutions



1

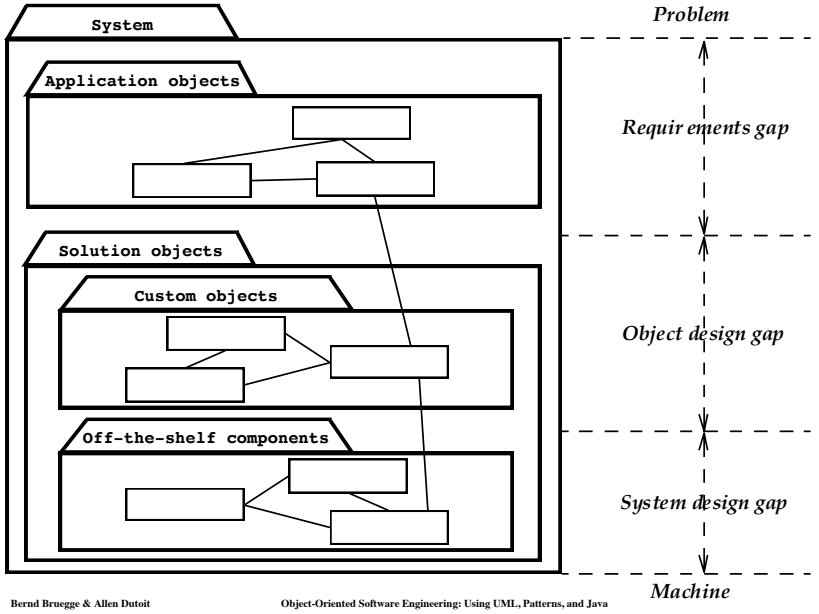
## *Object Design*

- Object design is the process of adding details to the requirements analysis and making implementation decisions
- The object designer must choose among different ways to implement the analysis model with the goal to minimize execution time, memory and other measures of cost.
- Requirements Analysis: Use cases, functional and dynamic model deliver operations for object model
- Object Design: We iterate on  where to put these operations in the object model
- Object Design serves as the basis of implementation



2

## *Object Design: Closing the Gap*






3

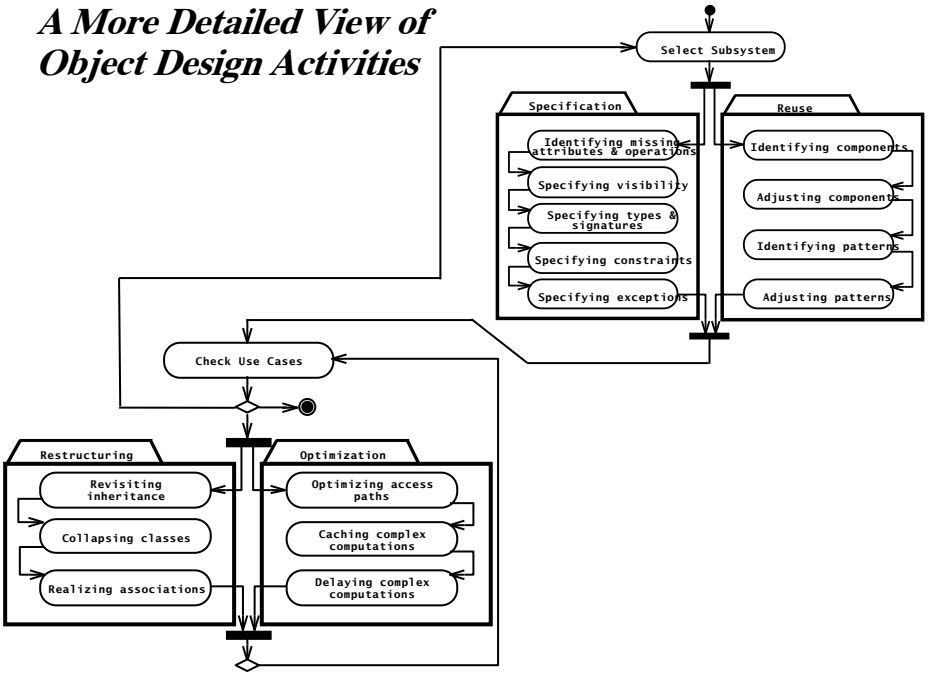
## *Examples of Object Design Activities*

- Identification of existing components
- Full definition of relations
- Full definition of classes (System Design => Service, Object Design => API)
- Specifying the contract for each component
- Choosing algorithms and data structures
- Identifying possibilities of reuse
- Detection of solution-domain classes
- Optimization
- Increase of inheritance
- Decision on control
- Packaging



4

## *A More Detailed View of Object Design Activities*






## *A Little Bit of Terminology: Activities*

- Object-Oriented Methodologies
  - ***System Design***
    - **Decomposition into subsystems**
  - ***Object Design***
    - **Implementation language chosen**
    - **Data structures and algorithms chosen**
- SA/SD uses different terminology:
  - ***Preliminary Design***
    - **Decomposition into subsystems**
    - **Data structures are chosen**
  - ***Detailed Design***
    - **Algorithms are chosen**
    - **Data structures are refined**
    - **Implementation language is chosen**
    - **Typically in parallel with preliminary design, not separate stage**

## *Finding Objects*

- The hardest parts in system development:
  - **Identifying objects**
  - **Decomposing a system into objects**
- Requirements Analysis focuses on application domain:
  - **Object identification**
- System Design addresses both, application and implementation domain:
  - **Subsystem Identification**
- Object Design focuses on implementation domain:
  - **More object identification**



## *Techniques for Finding Objects*

- Requirements Analysis
  - **Start with Use Cases. Identify participating objects**
  - **Textual analysis of flow of events (find nouns, verbs, ...)**
  - **Extract application domain objects by interviewing client (application domain knowledge)**
  - **Find objects by using general knowledge**
- System Design
  - **Subsystem decomposition**
  - **Try to identify layers and partitions**
- Object Design
  - **Find additional objects by applying implementation domain knowledge**

## *Application domain vs solution domain objects*

- Application objects, also called domain objects, represent concepts of the domain that are relevant to the system.
  - **They are identified by the application domain specialists and by the end users.**
- Solution objects represent concepts that do not have a counterpart in the application domain,
  - **They are identified by the developers**
  - **Examples: Persistent data stores, user interface objects, middleware.**

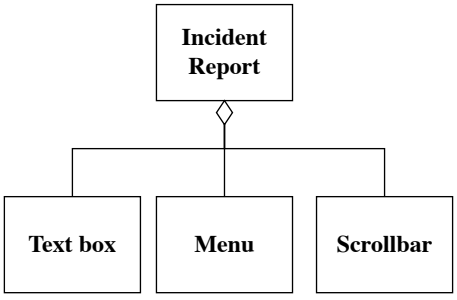


## *Application Domain vs Solution Domain Objects*

**Requirements Analysis (Language of Application Domain)**

**Incident Report**

**Object Design (Language of Solution Domain)**



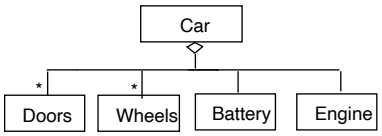
## *Another Source for Finding Objects : Design Patterns*

- Observation [Gamma et al 95]:
  - **Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.**
- There is a need for *reusable* and flexible designs
- Design knowledge complements application domain knowledge and implementation domain knowledge.
- 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 this solution a million times over, without ever doing it the same twice**

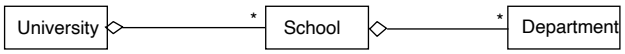


## *Review: Modeling Typical Aggregations*

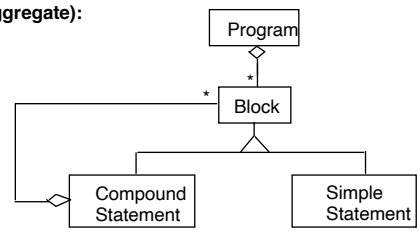
**Fixed Structure:**




**Organization Chart (variable aggregate):**

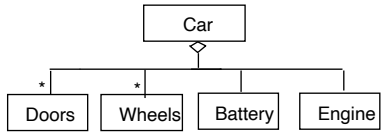



**Dynamic tree (recursive aggregate):**

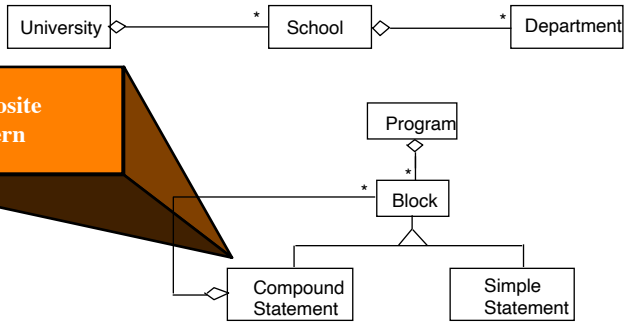


## *Review: Modeling Typical Aggregations*

**Fixed Structure:**



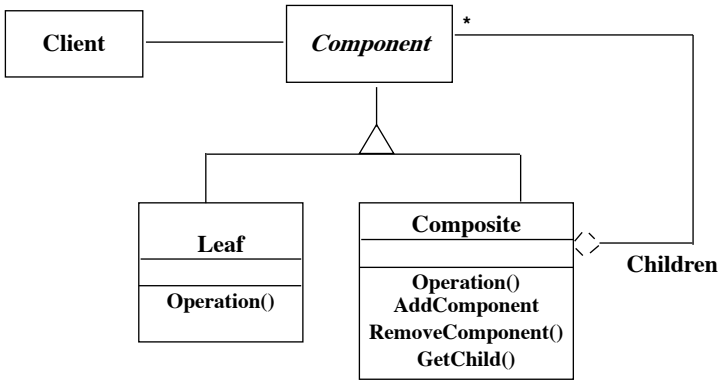

**Organization Chart (variable aggregate):**





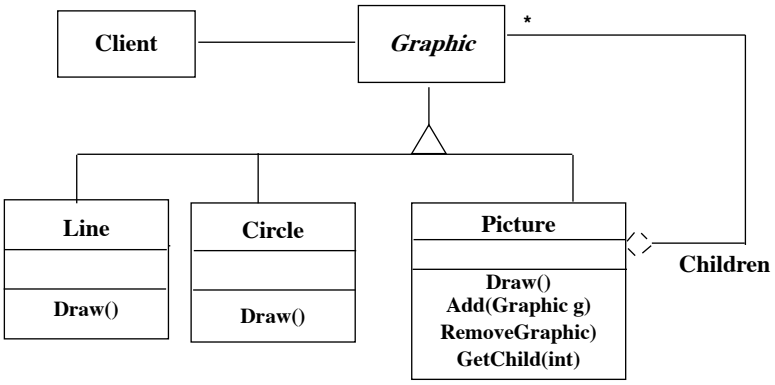

## *Composite Pattern*

- Composes objects into tree structures to represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly



## *Graphic Applications use Composite Patterns*

• The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)

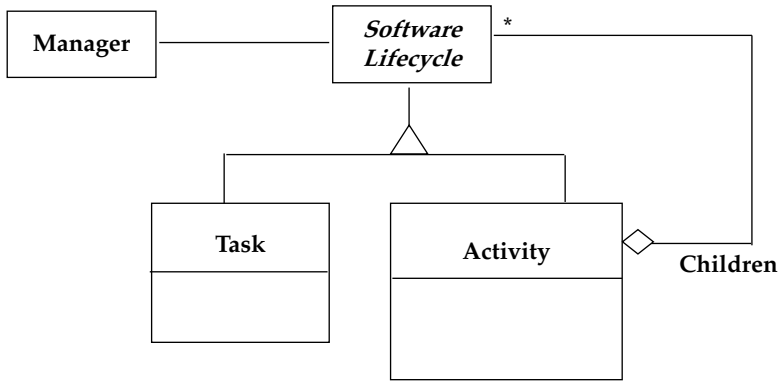





## *Modeling Software Development with Composite Patterns*

- Software Lifecycle:
  - **Definition: The software lifecycle consists of a set of development activities which are either other activities or collection of tasks**
  - **Composite: Activity (The software lifecycle consists of activities which consist of activities, which consist of activities, which....)**
  - **Leaf node: Task**
- Software System:
  - **Definition: A software system consists of subsystems which are either other subsystems or collection of classes**
  - **Composite: Subsystem (A software system consists of subsystems which consists of subsystems , which consists of subsystems, which...)**
  - **Leaf node: Class**

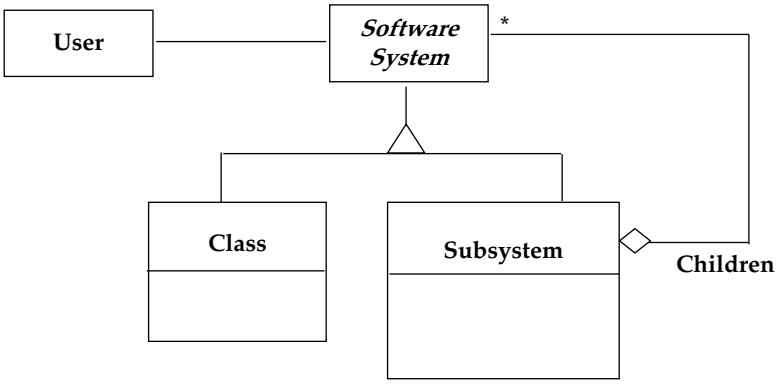
## *Modeling the Software Lifecycle with a Composite Pattern*






17

## *Modeling a Software System with a Composite Pattern*






18

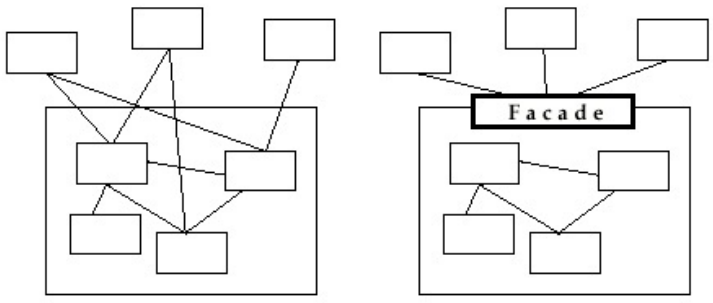
## *Ideal Structure of a Subsystem: Façade, Adapter, Bridge*

- 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!**



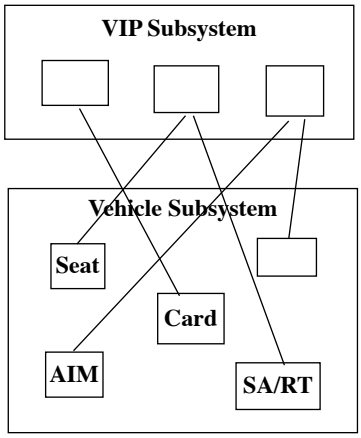
## *Facade Pattern*

- 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*

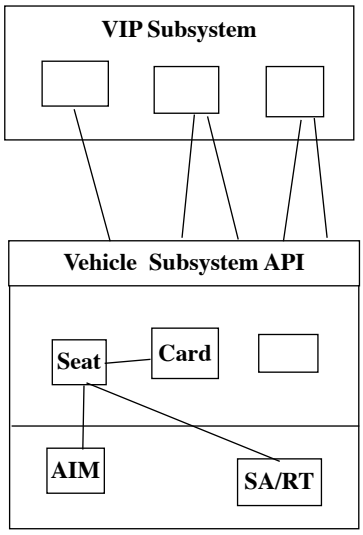
- Open architecture:
  - **Any client can see into the vehicle subsystem and call on any component or class operation at will.**
- Why is this good?
  - **Efficiency**
- Why is this bad?
  - **Can't expect the caller to understand how the subsystem works or the complex relationships within the subsystem.**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



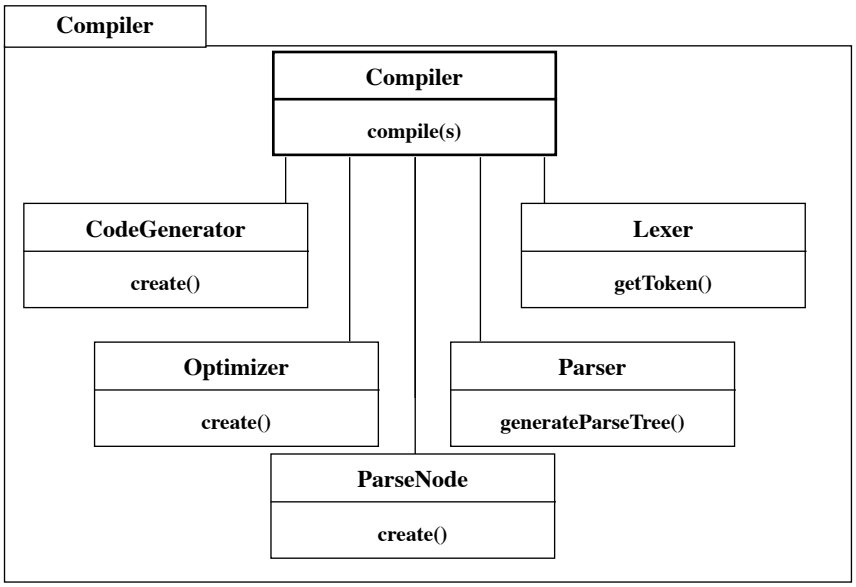



## *Realizing a Closed Architecture with a Facade*

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



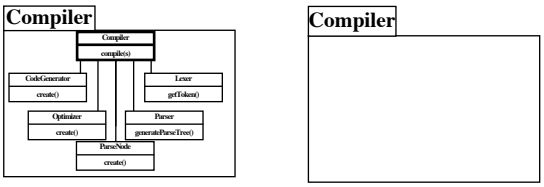
## *Realizing a Compiler with a Facade pattern*






## *UML Notation for subsystems: Package*

- Package = Collection of classes that are grouped together
- Packages are often used to model subsystems
- Notation:
  - **A box with a tab.**
  - **The tab contains the name of the package**




- In Together-J, every class is assigned to a default package
  - **When you create a class, the class is assigned to the default package directly containing the class diagram.**
  - **You can create other packages, but cannot delete the default package**

## *Some Additional Definitions*

- Before we go to the next pattern let's review the goal and some terms

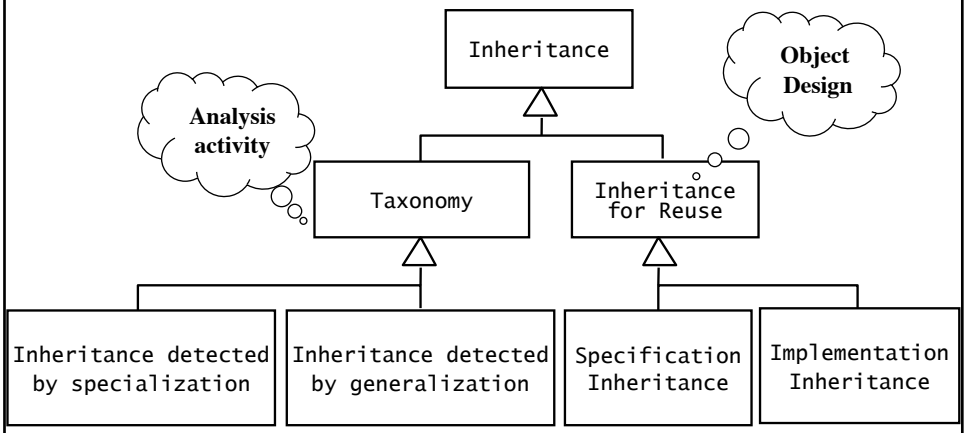


## *The use of inheritance*

- Inheritance is used to achieve two different goals
  - **Description of Taxonomies**
  - **Interface Specification**
- Identification of taxonomies
  - **Used during requirements analysis.**
  - **Activity: identify application domain objects that are hierarchically related**
  - **Goal: make the analysis model more understandable**
- Service specification
  - **Used during object design**
  - **Activity:**
  - **Goal: increase reusability, enhance modifiability and extensibility**
- Inheritance is found either by specialization or generalization

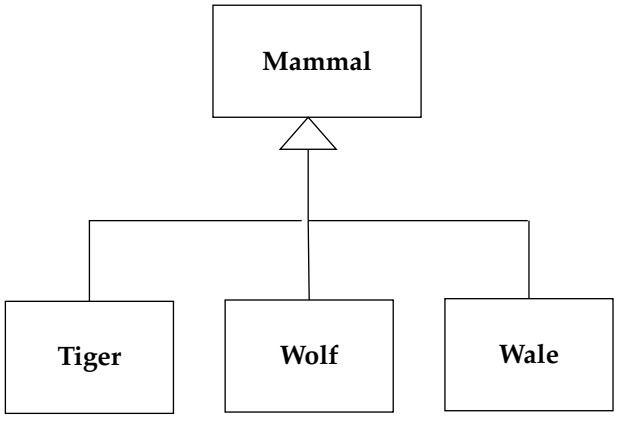
## *Metamodel for Inheritance*

- Inheritance is used during analysis and object design






## *Taxonomy Example*



## *Reuse*

- Main goal:
  - **Reuse knowledge from previous experience to current problem**
  - **Reuse functionality 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**

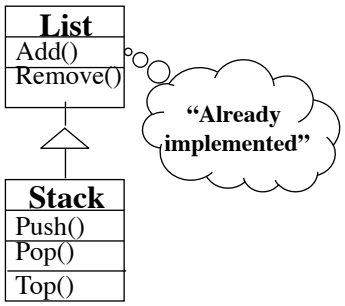


## *Implementation Inheritance vs Interface Inheritance*

- 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 subclassing the **Stack** class from the **List** class and providing three methods, **Push()** and **Pop(), Top()**?



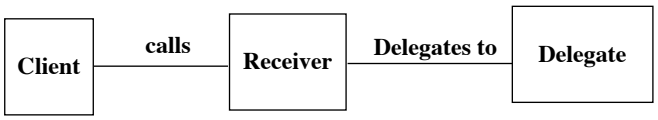

- Problem with implementation inheritance:

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



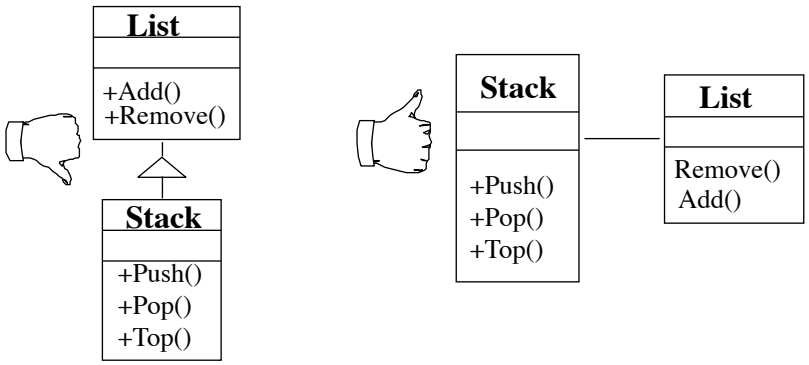
## *Delegation*

- 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 instead of Implementation Inheritance*

- **Inheritance**: Extending a Base class by a new operation or overwriting an operation.
- **Delegation**: Catching an operation and sending it to another object.
- Which of the following models is better for implementing a stack?






## *Comparison: Delegation vs Implementation 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 a subclass to the details of its parent class**
    - **Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)**

## *Design Heuristics*

- Never use implementation inheritance, always use interface inheritance
- A subclass should never hide operations implemented in a superclass
- If you are tempted to use implementation inheritance, use delegation instead



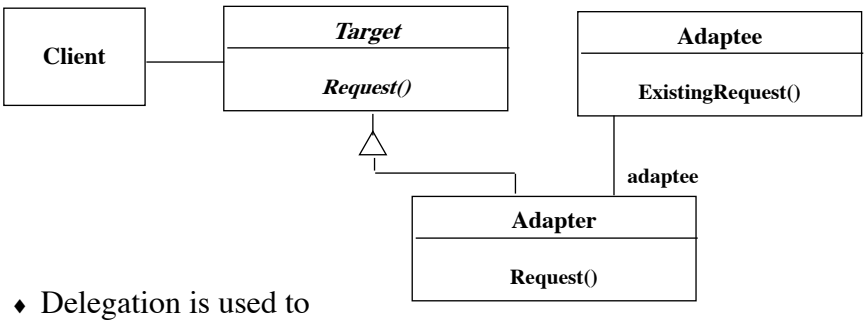
Many design patterns use a combination of inheritance and delegation

## *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**
- Object adapters are much more frequent. We will mostly use object adapters and call them simply adapters

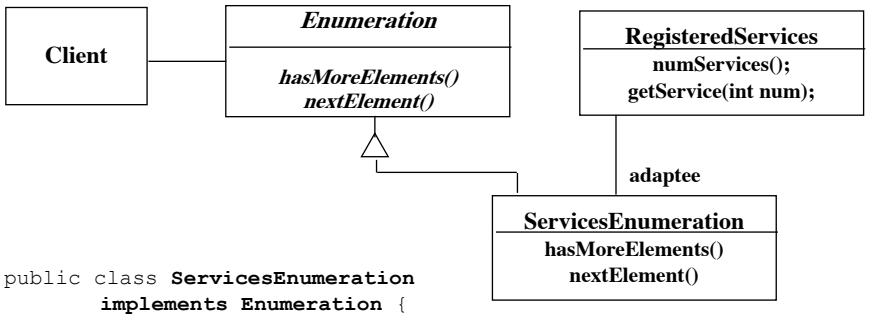


## *Adapter pattern*




- Delegation is used to bind an **Adapter** and an **Adaptee**
- Interface inheritance is used to specify the interface of the **Adapter** class.
- ***Target*** and **Adaptee** (usually called legacy system) pre-exist the **Adapter.**
- **Target** may be realized as an interface in Java.

## *Adapter pattern example*




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

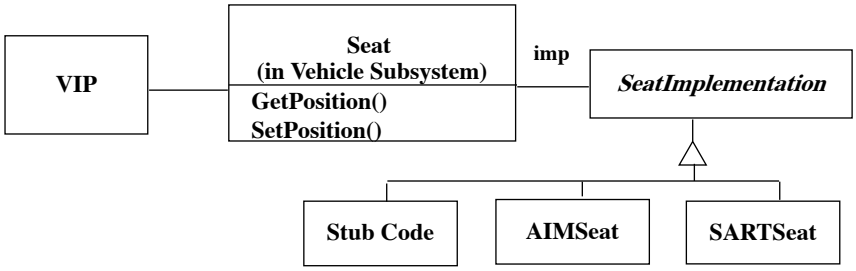


## *Bridge Pattern*

- Use a bridge to "decouple an abstraction from its implementation so that the two can vary independently". (From [Gamma et al 1995])
- Also know as a Handle/Body pattern.
- Allows different implementations of an interface to be decided upon dynamically.

## *Using a Bridge*

- The bridge pattern is used to provide multiple implementations under the same interface.
- Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- JAMES Project: if seat data is required to be read, but the seat is not yet implemented, not yet known or only available by a simulation, provide a bridge:






## *Seat Implementation*

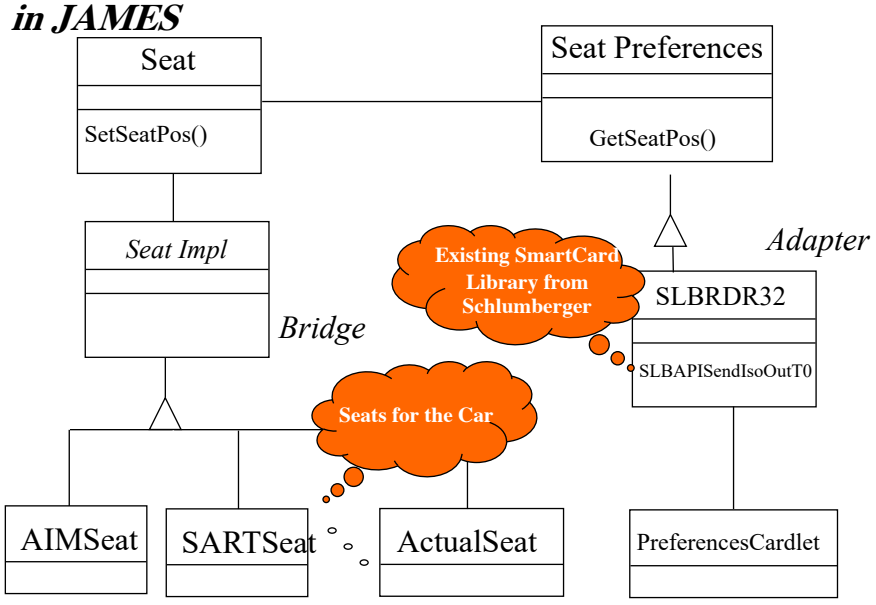
```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class Stubcode implements SeatImplementation
  {
  public int GetPosition() {
    // stub code for GetPosition
  }
  ...
}
public class AimSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the AIM simulation system
  }
  ….
}
public class SARTSeat implements SeatImplementation
  {
  public int GetPosition() {
    // actual call to the SART seat simulator
 }
  ...
}
```

## *Adapter vs Bridge*

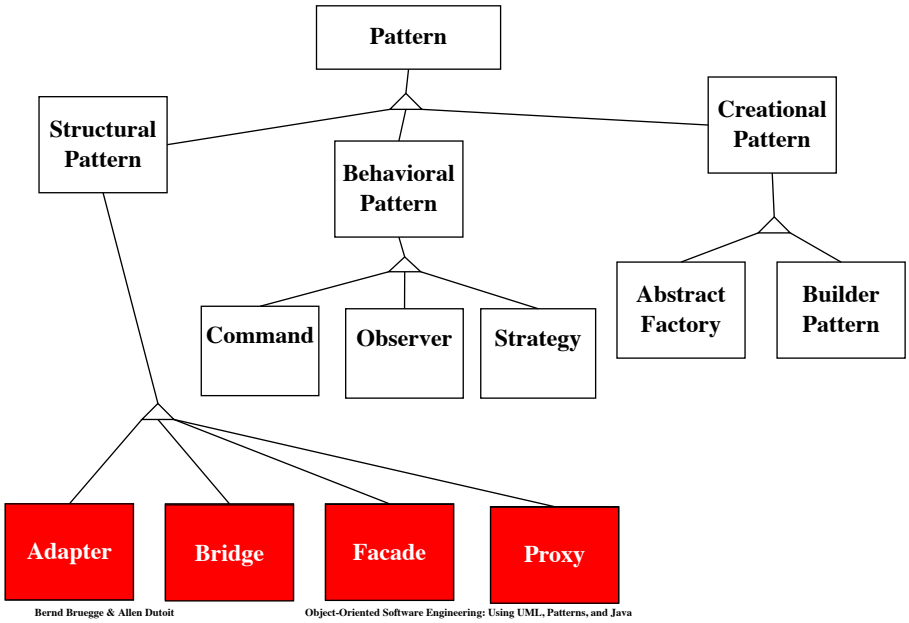
- Similarities:
  - **Both used to hide the details of the underlying implementation.**
- Difference:
  - **The adapter pattern is geared towards making unrelated components work together**
    - **Applied to systems after they're designed (reengineering, interface engineering).**
  - **A bridge, on 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.**



## *Example for Combination of Adapters and Bridges in JAMES*



## *More Patterns: A Pattern Taxonomy*






48

## *Proxy Pattern: Motivation*

- It is 15:00pm. I am sitting at my 14.4 baud modem connection and retrieve a fancy web site from the US, This is prime web time all over the US. So I am getting 10 bits/sec.
- What can I do?



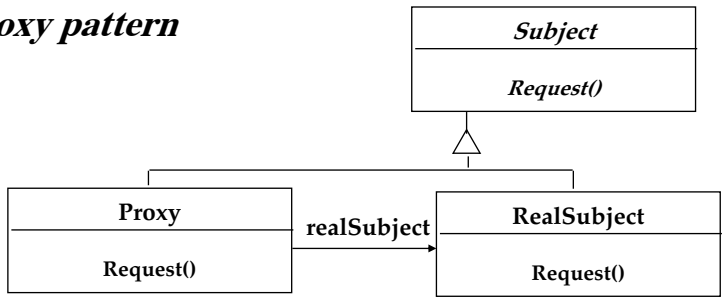
49

## *Proxy Pattern*

- What is expensive?
  - **Object Creation**
  - **Object Initialization**
- Defer object creation and object initialization to the time you need the object
- Proxy pattern:
  - **Reduces the cost of accessing objects**
  - **Uses another object ("the proxy") that acts as a stand-in for the real object**
  - **The proxy creates the real object only if the user asks for it**



## *Proxy pattern*



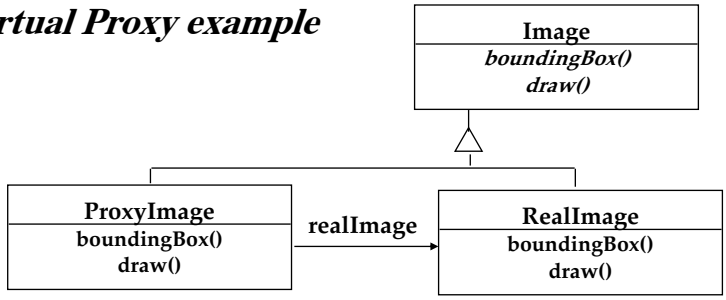

- Interface inheritance is used to specify the interface shared by **Proxy** and **RealSubject.**
- Delegation is used to catch and forward any accesses to the **RealSubject** (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.

## *Proxy Applicability*

- Remote Proxy
  - **Local representative for an object in a different address space**
  - **Caching of information: Good if information does not change too often**
- Virtual Proxy
  - **Object is too expensive to create or too expensive to download**
  - **Proxy is a standin**
- Protection Proxy
  - **Proxy provides access control to the real object**
  - **Useful when different objects should have different access and viewing rights for the same document.**
  - **Example: Grade information for a student shared by administrators, teachers and students.**



## *Virtual Proxy example*

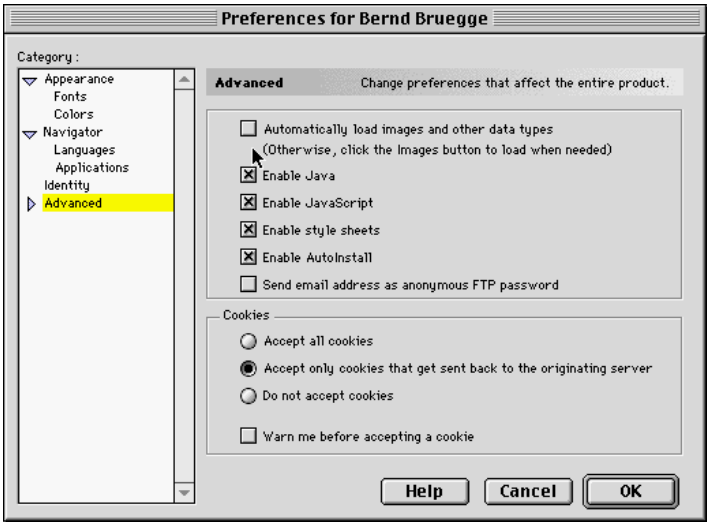



- **Images** are stored and loaded separately from text
- If a **RealImage** is not loaded a **ProxyImage** displays a grey rectangle in place of the image
- The client cannot tell that it is dealing with a **ProxyImage** instead of a **RealImage**
- A proxy pattern can be easily combined with a **Bridge**

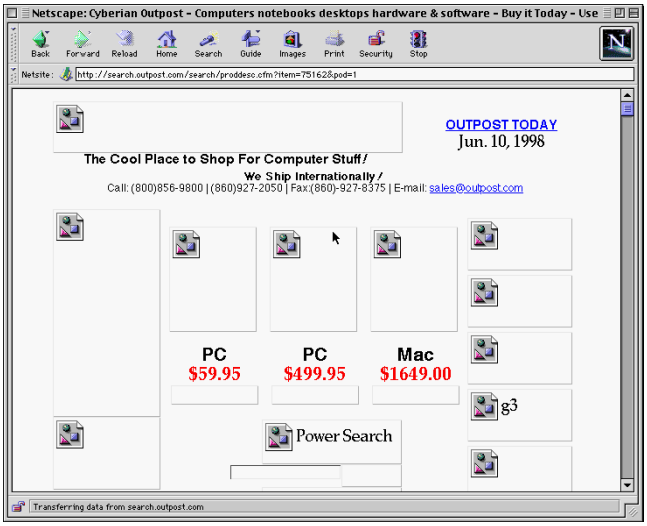
## *Before*




## *Controlling Access*



## After

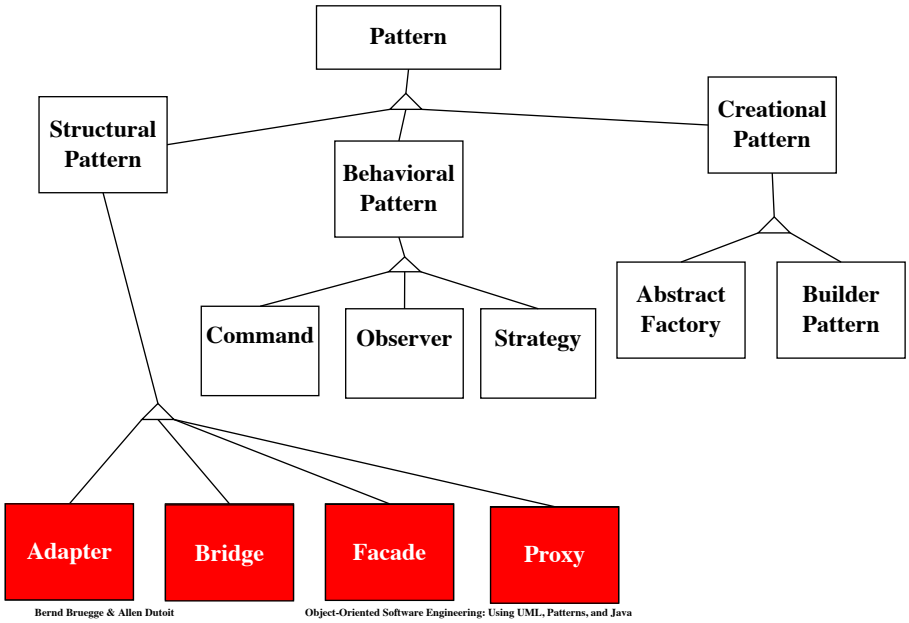





## Summary

- Composite Pattern:
  - **Models trees with dynamic width and dynamic depth**
- Adapters, Bridges, Facades, and Proxies (structural Patterns) are variations on a single theme:
  - **They reduce the coupling between two or more classes**
  - **They introduce an abstract class to enable future extensions**
  - **They encapsulate complex structures**
- Facade Pattern:
  - **Interface to a Subsystem, Closed vs Open Architecture**
- Adapter Pattern:
  - **Interface to Reality**
- Bridge Pattern:
  - **Interface Reality and Future**
- Proxy Patterns
  - **Defer object creation and initialization to the time you need the object**

## *A Pattern Taxonomy*

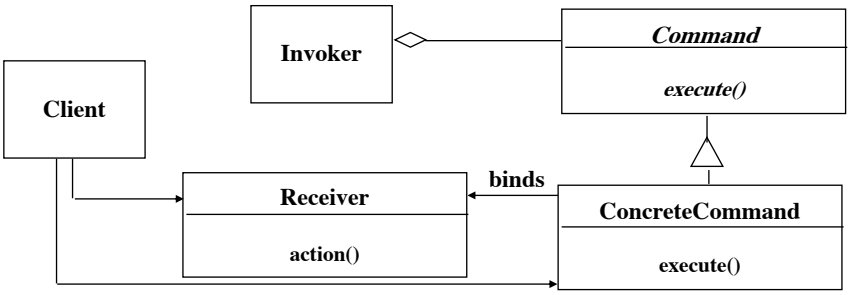





## *Command Pattern: Motivation*

- You want to build a user interface
- You want to provide menus
- You want to make the user interface reusable across many applications
  - **You cannot hardcode the meanings of the menus for the various applications**
  - **The applications only know what has to be done when a menu is selected.**
- Such a menu can easily be implemented with the Command Pattern

## *Command pattern*

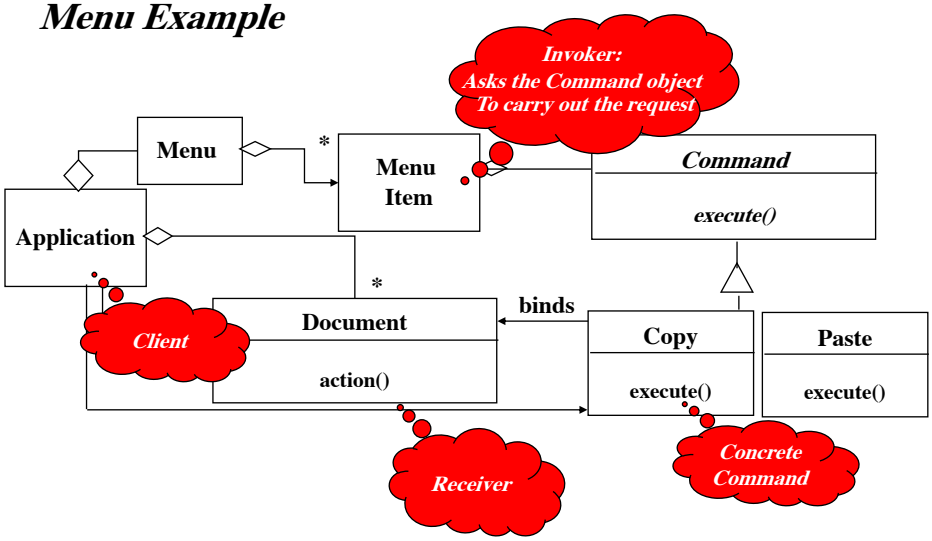



- **Client** creates a **ConcreteCommand** and binds it with a **Receiver.**
- **Client** hands the **ConcreteCommand** over to the **Invoker** which stores it.
- The **Invoker** has the responsibility to do the command ("execute" or "undo")**.**



60

## *Menu Example*





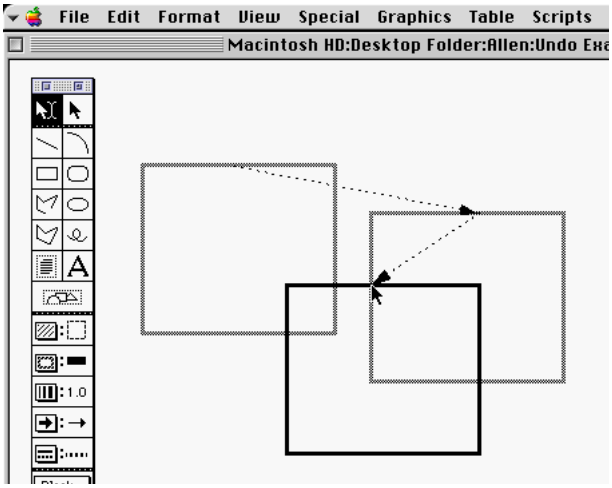

61

## *Command pattern Applicability*

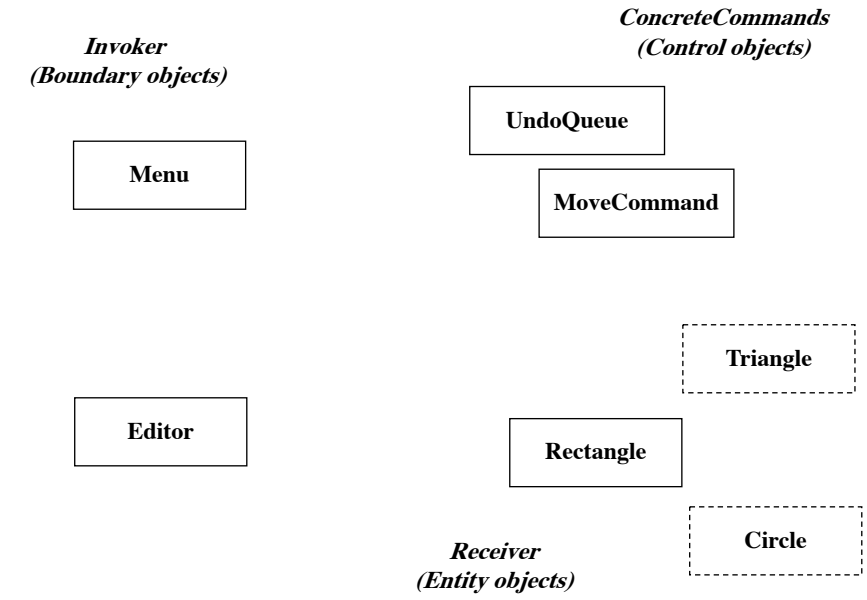
- "Encapsulate a request as an object, thereby letting you
  - **parameterize clients with different requests,**
  - **queue or log requests, and**
  - **support undoable operations."**
- Uses:
  - **Undo queues**
  - **Database transaction buffering**



## *Command pattern: Editor with unlimited undos*



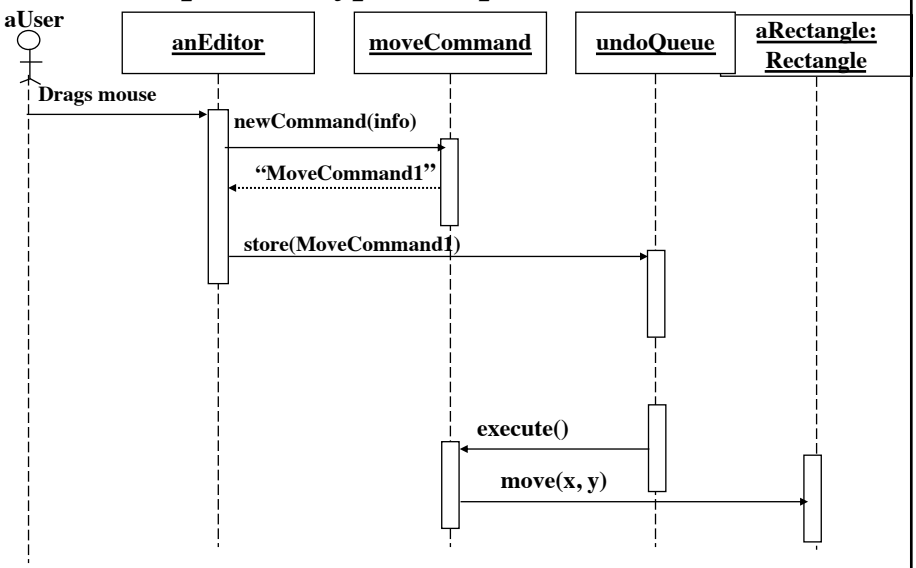
## *Structuring the objects*






64

## *Command pattern: typical sequence*






65

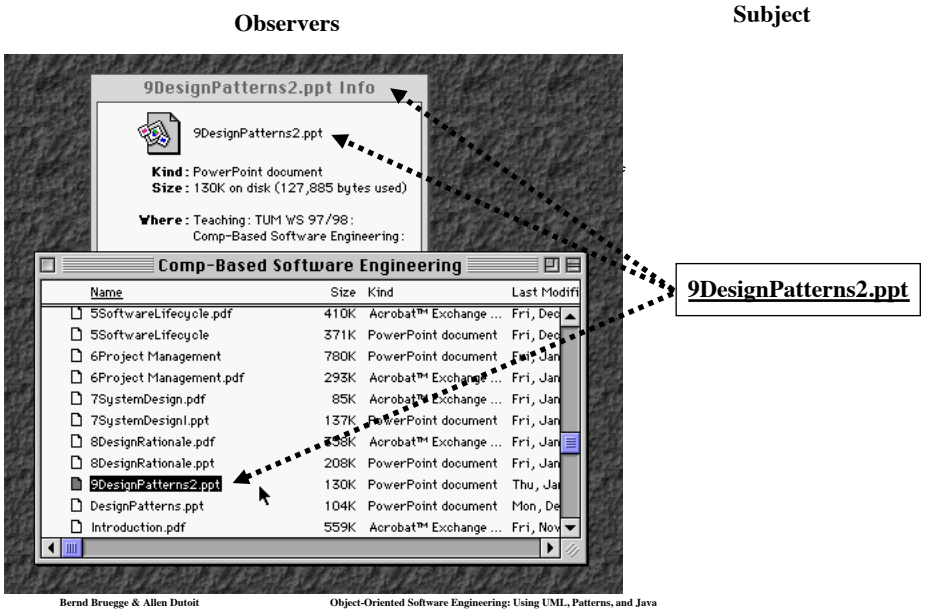
## *Observer pattern (293)*

- "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." (p. 293)
- Also called "Publish and Subscribe"

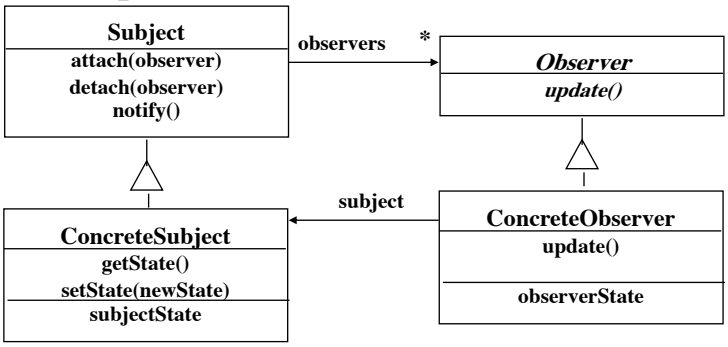
- Uses:
  - **Maintaining consistency across redundant state**
  - **Optimizing batch changes to maintain consistency**



## *Observer pattern (continued)*



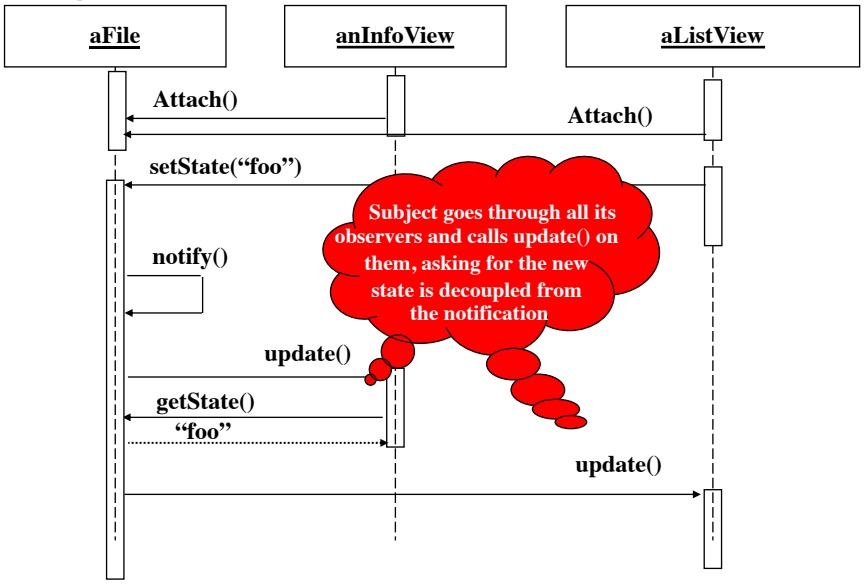
## *Observer pattern (continued)*



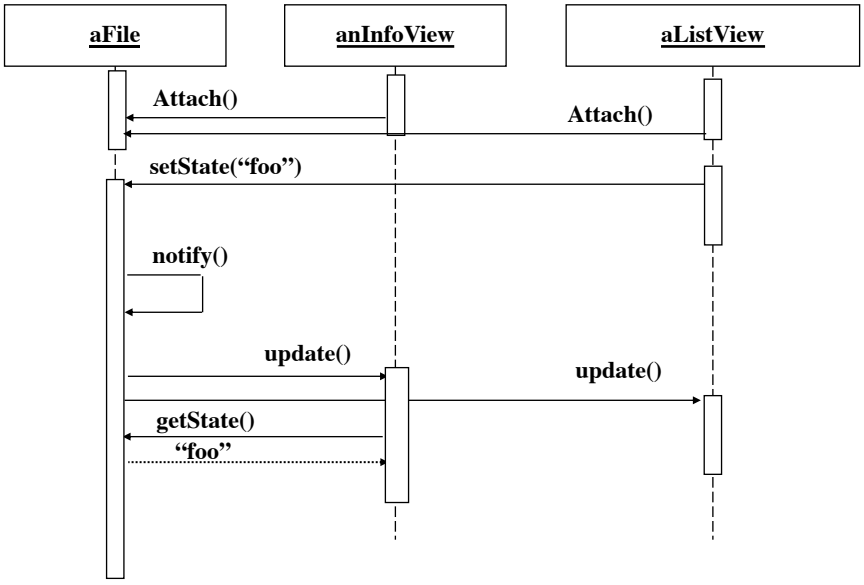

- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) ***not*** an interface.



## *Sequence diagram for scenario: Change filename to "foo"*



## *Animated Sequence diagram*






70

## *Observer pattern implementation in Java*

```
// import java.util;

public class Observable extends Object {
   public void addObserver(Observer o);
   public void deleteObserver(Observer o);
   public boolean hasChanged();
   public void notifyObservers();
   public void notifyObservers(Object arg);
}

public interface Observer {
   public void update(Observable o, Object arg);
}
public class Subject extends Observable{
   public void setState(String filename);
   public string getState();
}
```



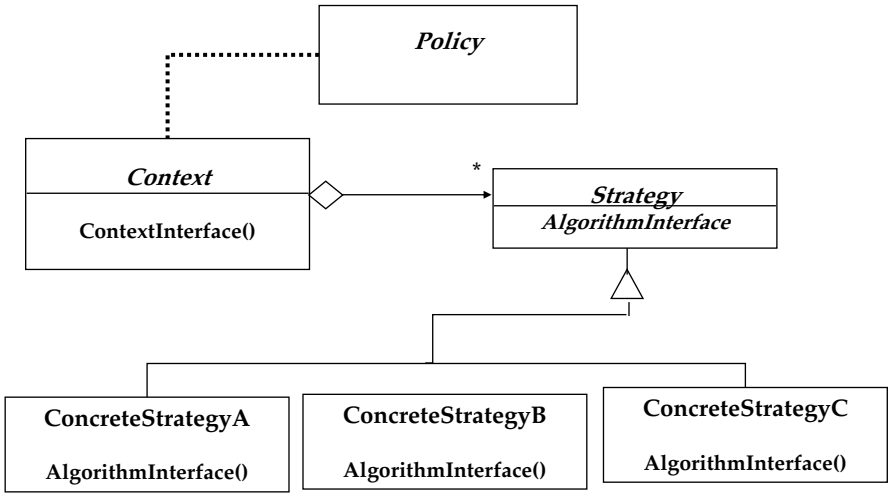
71

## *Strategy Pattern*

- Many different algorithms exists for the same task
- Examples:
  - **Breaking a stream of text into lines**
  - **Parsing a set of tokens into an abstract syntax tree**
  - **Sorting a list of customers**
- The different algorithms will be appropriate at different times
  - **Rapid prototyping vs delivery of final product**
- We don't want to support all the algorithms if we don't need them
- If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm

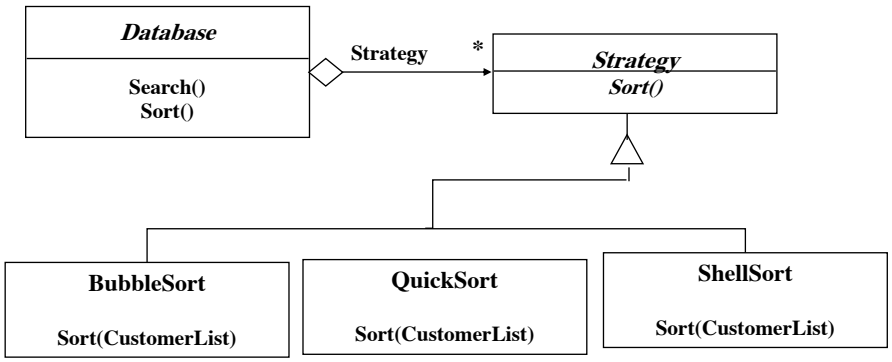


## *Strategy Pattern*




**`Policy` decides which `Strategy` is best given the current `Context`**

## *Applying a Strategy Pattern in a Database Application*

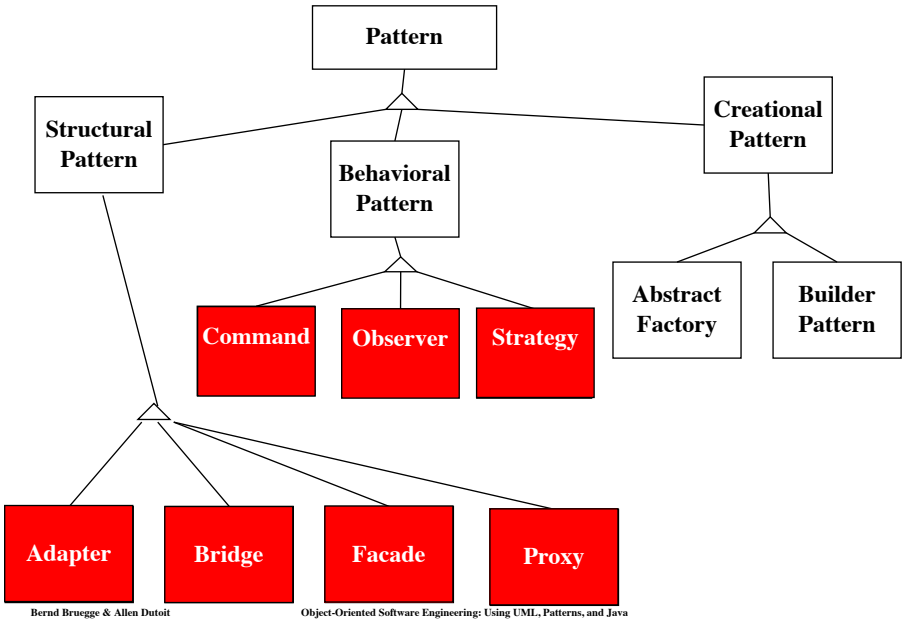





## *Applicability of Strategy Pattern*

- Many related classes differ only in their behavior. Strategy allows to configure a single class with one of many behaviors
- Different variants of an algorithm are needed that trade-off space against time. All these variants can be implemented as a class hierarchy of algorithms

## A Pattern Taxonomy

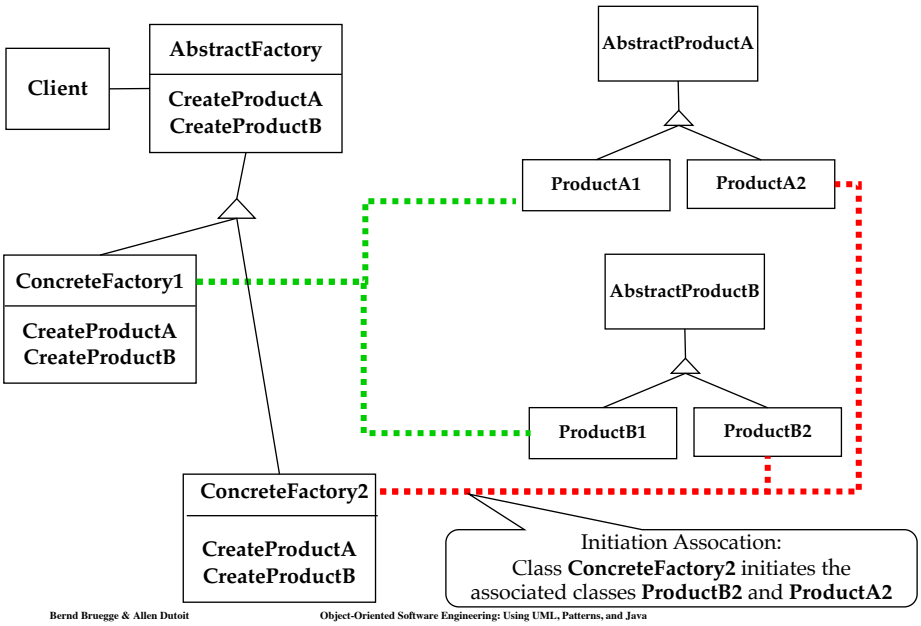





## Abstract Factory Motivation

- Consider a user interface toolkit that supports multiple looks and feel standards such as Motif, Windows 95 or the finder in MacOS.
  - **How can you write a single user interface and make it portable across the different look and feel standards for these window managers?**
- Consider a facility management system for an intelligent house that supports different control systems such as Siemens' Instabus, Johnson & Control Metasys or Zumtobe's proprietary standard.
  - **How can you write a single control system that is independent from the manufacturer?**

## *Abstract Factory*

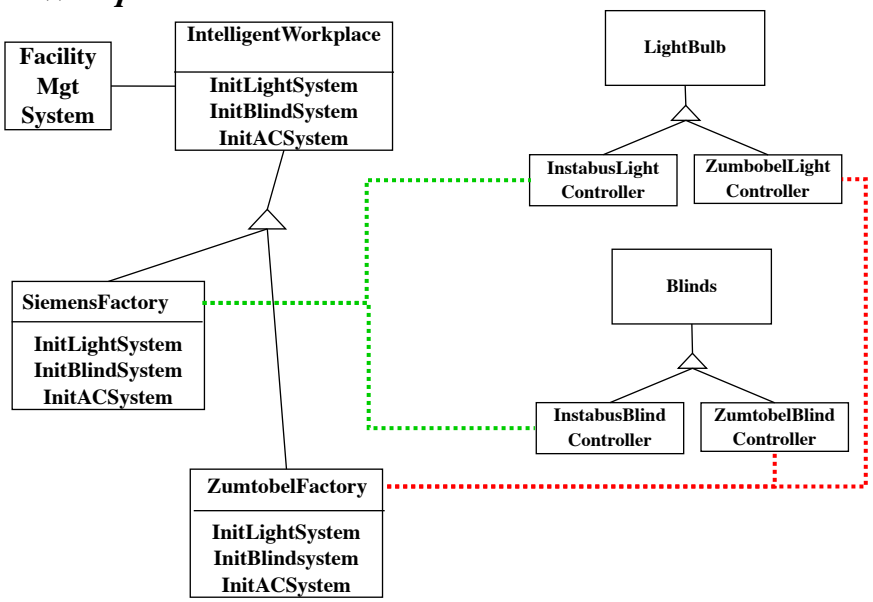





## *Applicability for Abstract Factory Pattern*

- Independence from Initialization or Representation:
  - **The system should be independent of how its products are created, composed or represented**
- Manufacturer Independence:
  - **A system should be configured with one of multiple family of products**
  - **You want to provide a class library for a customer ("facility management library"), but you don't want to reveal what particular product you are using.**
- Constraints on related products
  - **A family of related products is designed to be used together and you need to enforce this constraint**
- Cope with upcoming change:
  - **You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.**

## *Example: A Facility Management System for the Intelligent Workplace*

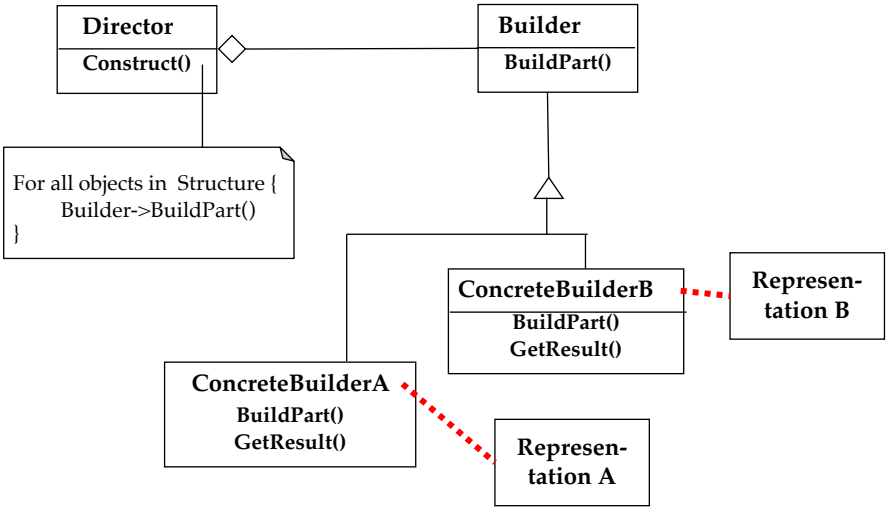





## *Builder Pattern Motivation*

- Conversion of documents
- Software companies make their money by introducing new formats, forcing users to upgrades
  - **But you don't want to upgrade your software every time there is an update of the format for Word documents**
- Idea: A reader for RTF format
  - **Convert RTF to many text formats (EMACS, Framemaker 4.0, Framemaker 5.0, Framemaker 5.5, HTML, SGML, WordPerfect 3.5, WordPerfect 7.0, ….)**
    - ***Problem: The number of conversions is open-ended.***
- Solution
  - **Configure the RTF Reader with a "builder" object that specializes in conversions to any known format and can easily be extended to deal with any new format appearing on the market**

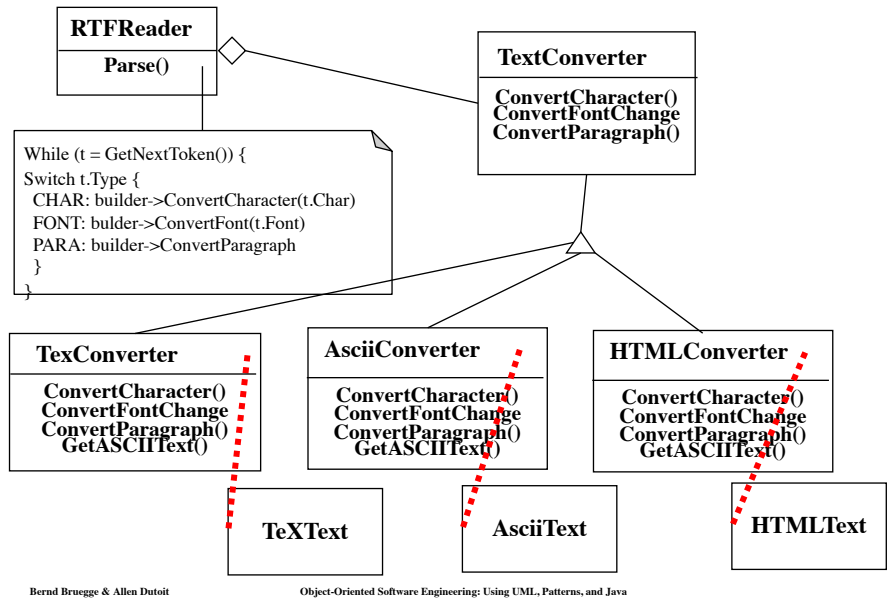
## *Builder Pattern*






82

## *Example*






83

## *When do you use the Builder Pattern?*

- The creation of a complex product must be independent of the particular parts that make up the product
  - **In particular, the creation process should not know about the assembly process (how the parts are put together to make up the product)**
- The creation process must allow different representations for the object that is constructed. Examples:
  - **A house with one floor, 3 rooms, 2 hallways, 1 garage and three doors.**
  - **A skyscraper with 50 floors, 15 offices and 5 hallways on each floor. The office layout varies for each floor.**



## *Abstract Factory vs Builder*

- Abstract Factory
  - **Focuses on product family**
    - **The products can be simple ("light bulb") or complex**
  - **The abstract factory does not hide the creation process**
    - **The product is immediately returned**
- Builder
  - **The underlying product needs to be constructed as part of the system but is very complex**
  - **The construction of the complex product changes from time to time**
  - **The builder patterns hides the complex creation process from the user:**
    - **The product is returned after creation as a final step**
- Abstract Factory and Builder work well together for a family of multiple complex products

## *Summary*

- Structural Patterns
  - **Focus: How objects are composed to form larger structures**
  - **Problems solved:**
    - **Realize new functionality from old functionality,**
    - **Provide flexibility and extensibility**
- Behavioral Patterns
  - **Focus: Algorithms and the assignment of responsibilities to objects**
  - **Problem solved:**
    - **Too tight coupling to a particular algorithm**
- Creational Patterns
  - **Focus: Creation of complex objects**
  - **Problems solved:**
    - **Hide how complex objects are created and put together**



## *Conclusion*

- Design patterns
  - **Provide solutions to common problems.**
  - **Lead to extensible models and code.**
  - **Can be used as is or as examples of interface inheritance and delegation.**
  - **Apply the same principles to structure and to behavior.**
- Design patterns solve all your software engineering problems