15-413

Design Patterns

Bernd Bruegge

Carnegie Mellon University

Department of Computer Science

15 October 1998

Outline of the Lecture

- Design Patterns
  - **Usefulness of design patterns**
  - **Design Pattern Categories**
- Patterns covered in this Lecture
  - **Composite: Model dynamic aggregates**
  - **Facade: Interfacing to subsystems**
  - **Adapter: Interfacing to existing systems (legacy systems)**
  - **Bridge: Interfacing to existing and future systems**
  - **Proxy: Controlling access**
  - **Observer: Publish and subscribe**
  - **Abstract Factory: Creation of product family hiding the manufacturer**
  - **Builder: Creation of complex object hiding its representation**

What is a design pattern?

A design pattern is…

…a template solution to a recurring design problem

- **Look before re-inventing the wheel just one more time**

…reusable design knowledge

- **Higher level than link lists or binary trees**
- **Lower level than application frameworks**

…an example of modifiable and reusable design

- **Learning to design starts by studying other designs**

Why are modifiable designs important?

A modifiable design enables…

…an iterative and incremental development cycle

- **concurrent development**
- **risk management**
- **flexibility to change**

…to minimize the introduction of new problems when fixing old ones

…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?

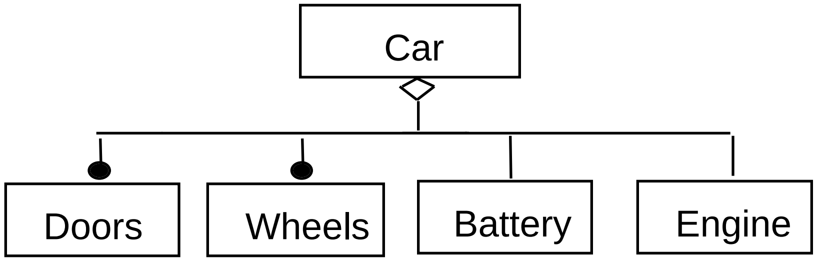
- They are generalized from existing systems
- They provide a shared vocabulary to designers
- They provide examples of modifiable designs
    - **Abstract classes**
    - **Delegation**

Design Patterns Notation

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995:
- Notation is based on OMT, Modification to UML Notation:
  - **Attributes come after the Operations**
  - **Associations are called acquaintance**
  - **Multiciplicity "many" is shown as solid circle at end of association**
- Important notations used in patterns:
  - **Dashed line: Instantiation Assocation (Class can instantiate objects of associated class)**
  - **Class Names and Operations in Italics denote Abstract Classes and Abstract Operations**
  - **Dogear box (connected by dashed line to class operation): Pseudo-code implementation of operation**

Review: Modeling Typical Aggregations

Fixed Structure:



Car

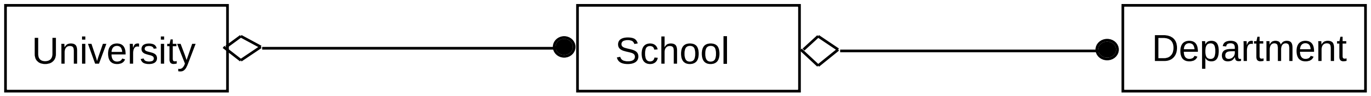
Doors

Wheels

Battery

Engine

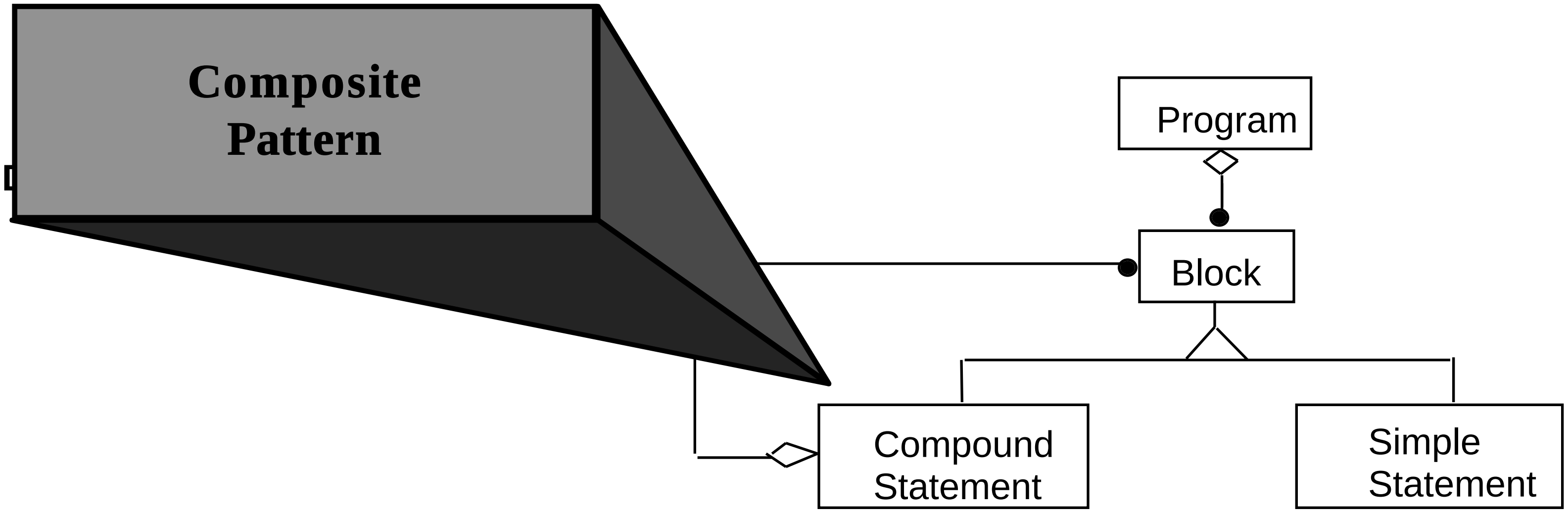
Organization Chart (variable aggregate):



University

School

Department



Composite Pattern

Program

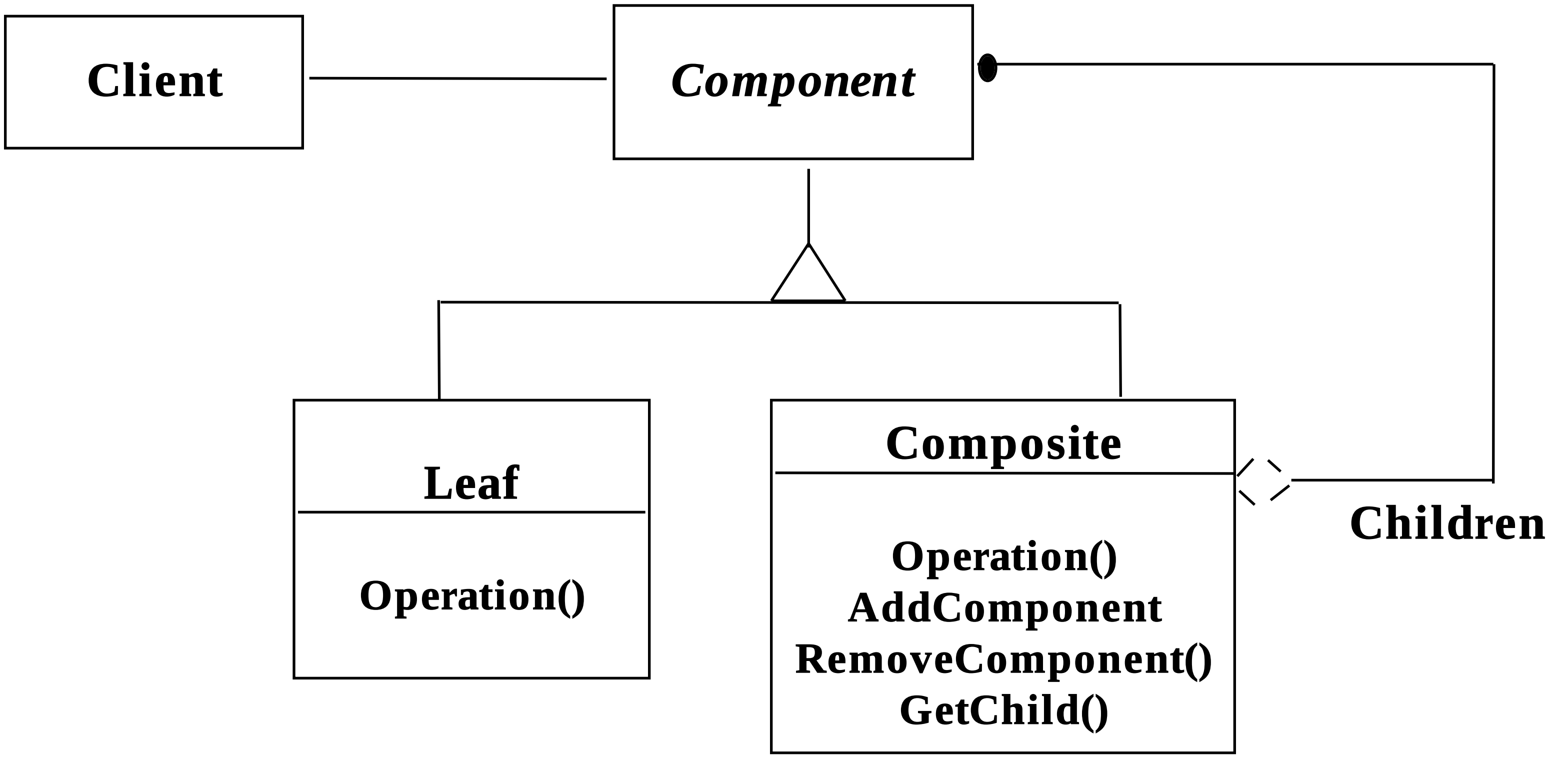
Block

Compound Statement

Simple Statement

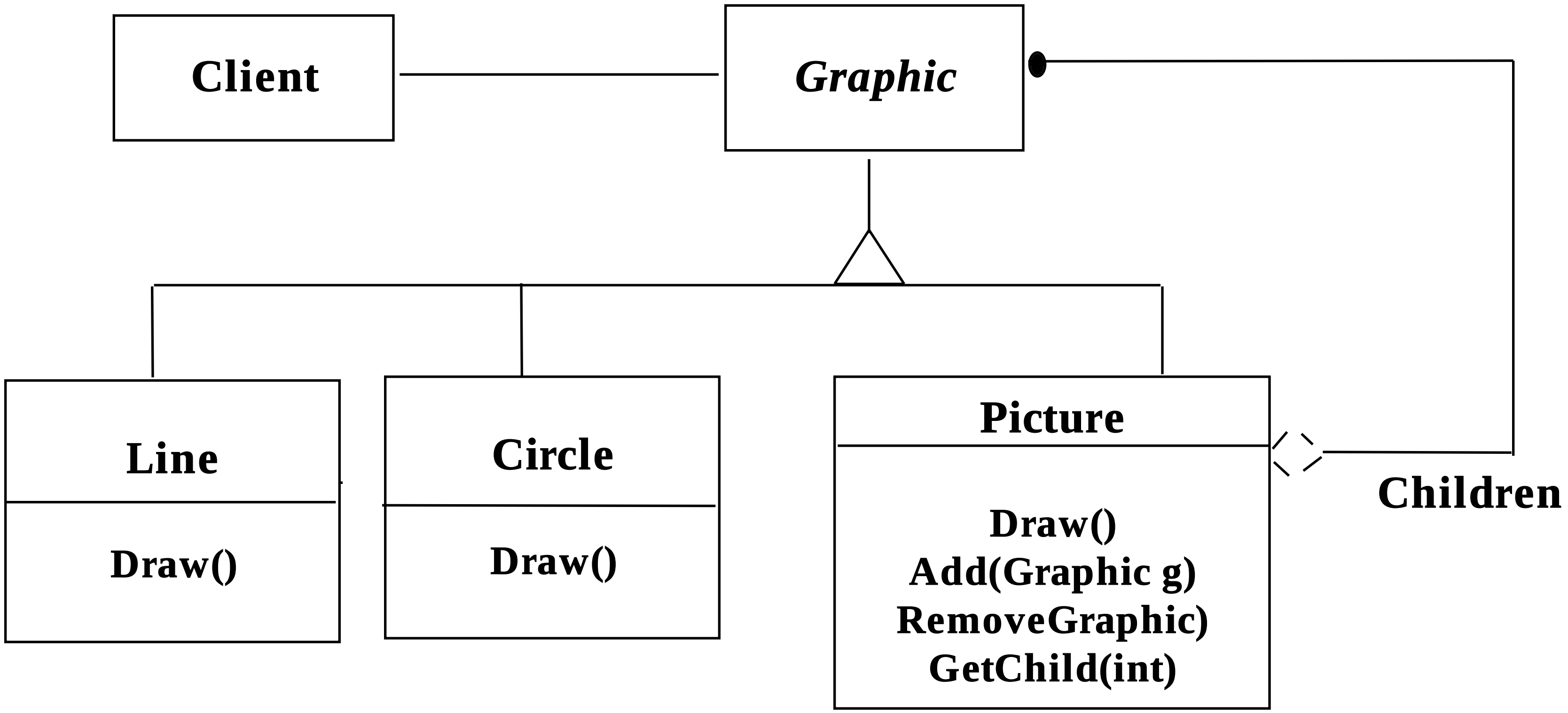
Composite Pattern

- 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



Example: Graphic Applications use Composite Patterns

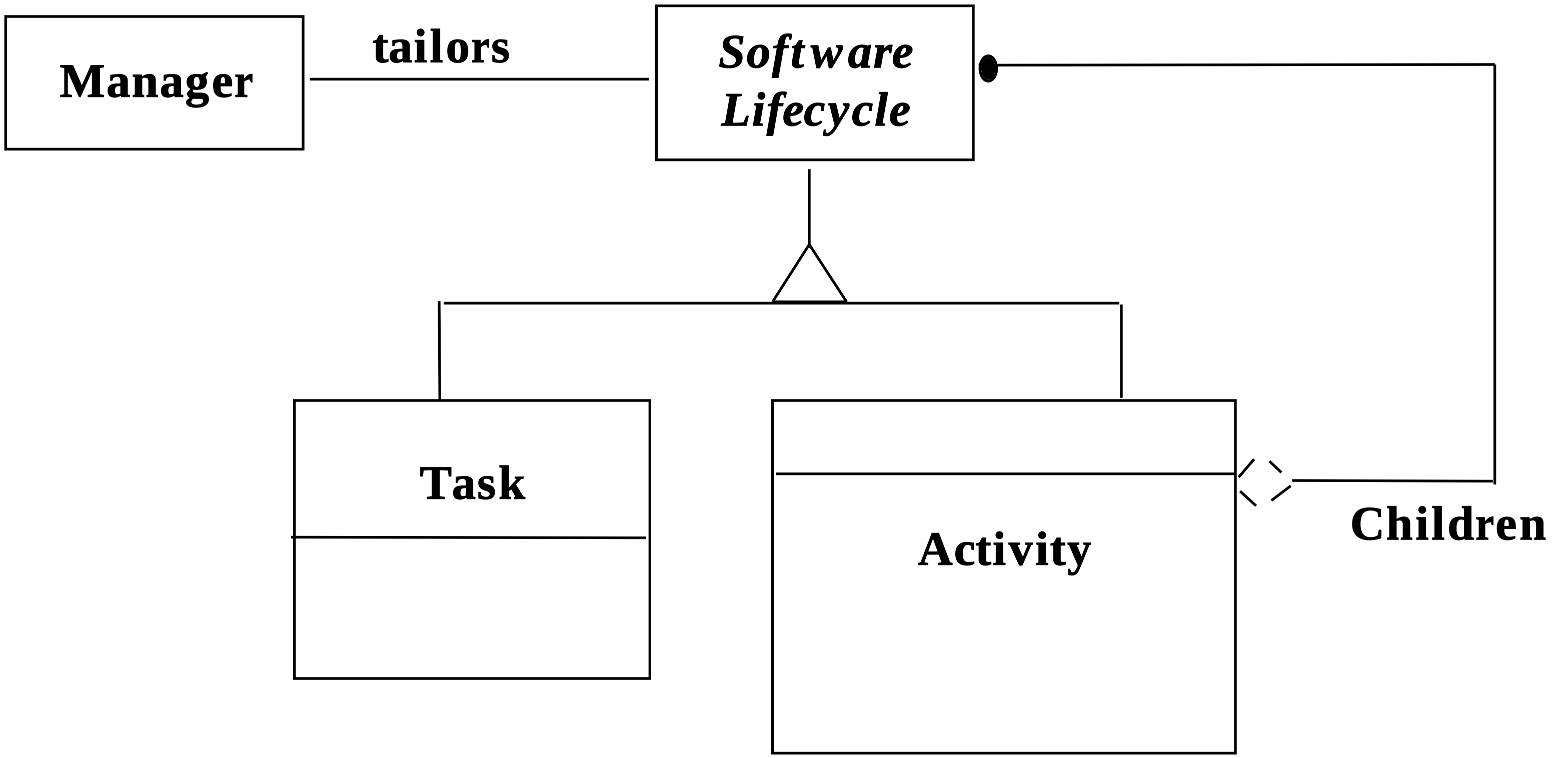
- The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)



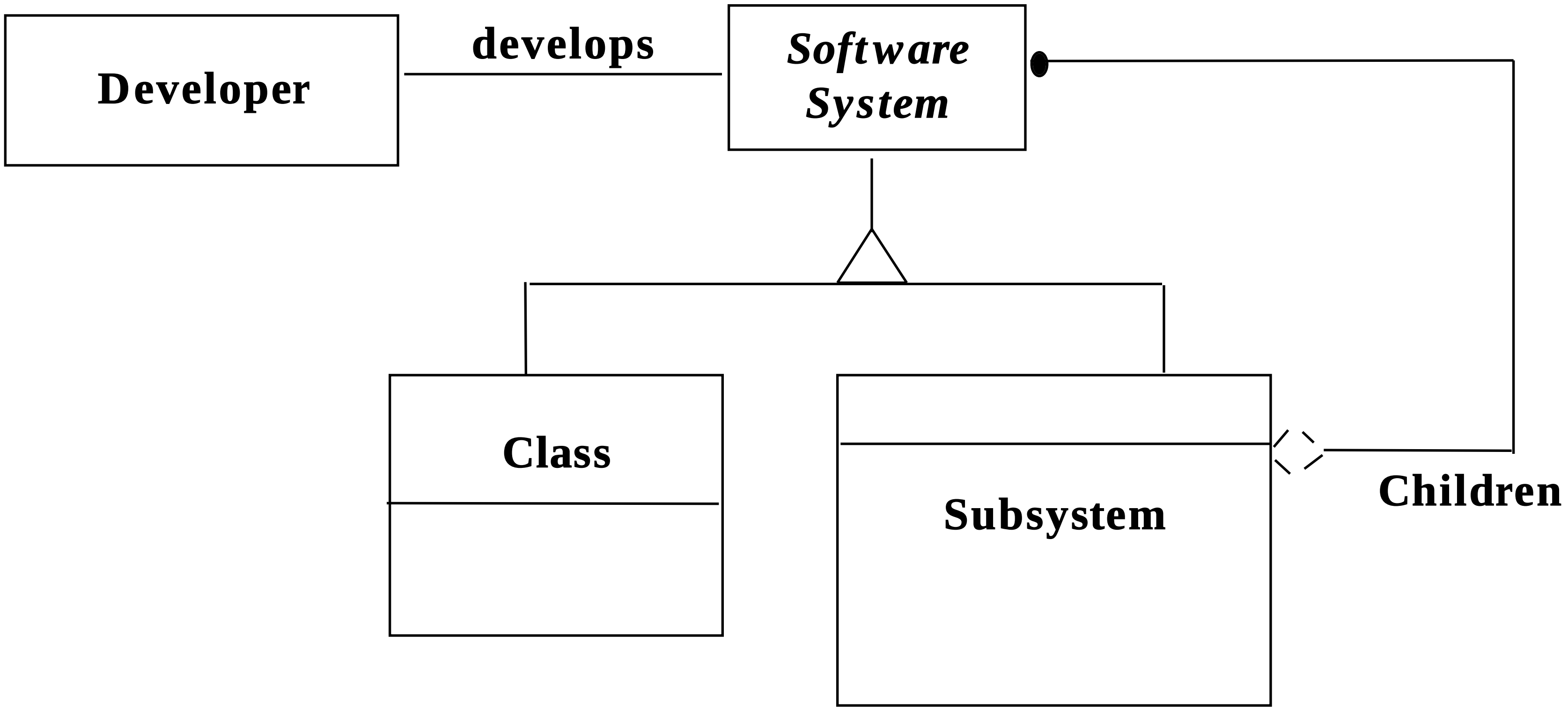
Modeling Software Development with Composite Patterns

- Software Lifecycle:
  - **Definition: The software lifecycle consists of a set of development activities which are either other actitivies 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



Modeling a Software System with a Composite Pattern



Odds and Ends

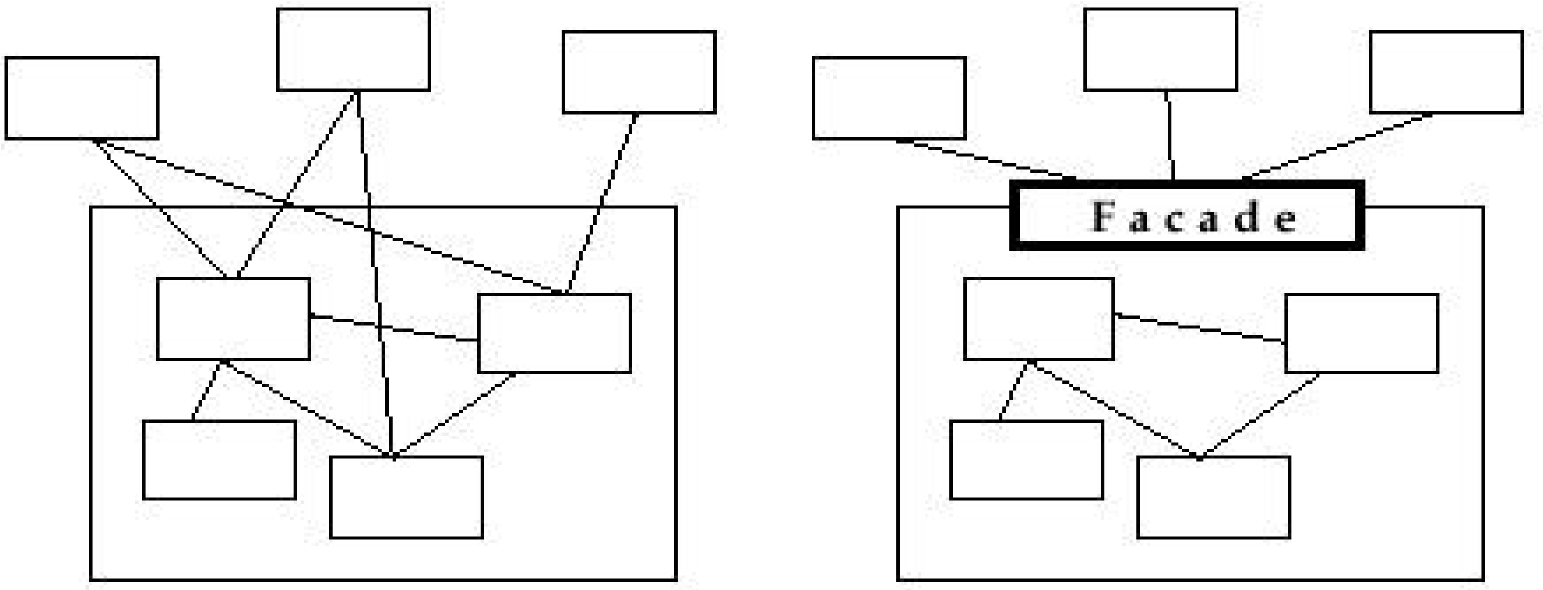
- Our communication structure is not working very well
- Speakers needed for Requirements Analysis presentation:
  - **Oct 22:**
    - **User Interface: Speaker?**
    - **Authentication: Speaker?**
    - **Learning: Speaker?**
  - **Oct 27:**
    - **Network: Speaker?**
    - **Database: Speaker**
    - **Project Management: Speaker?**

Applying Design Patterns to Subsystems

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

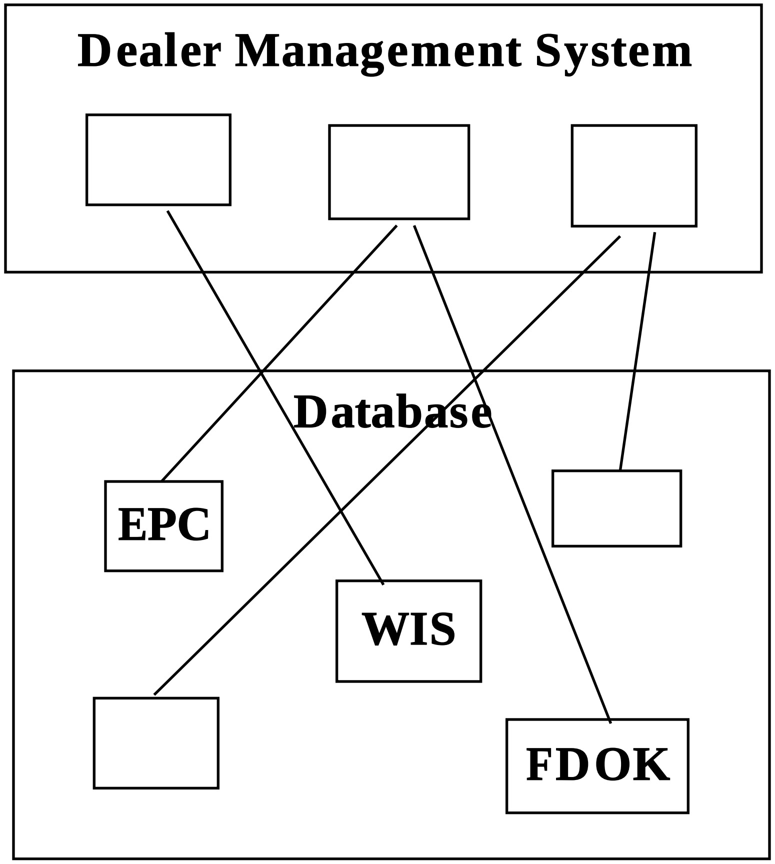
Facade Pattern

- 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



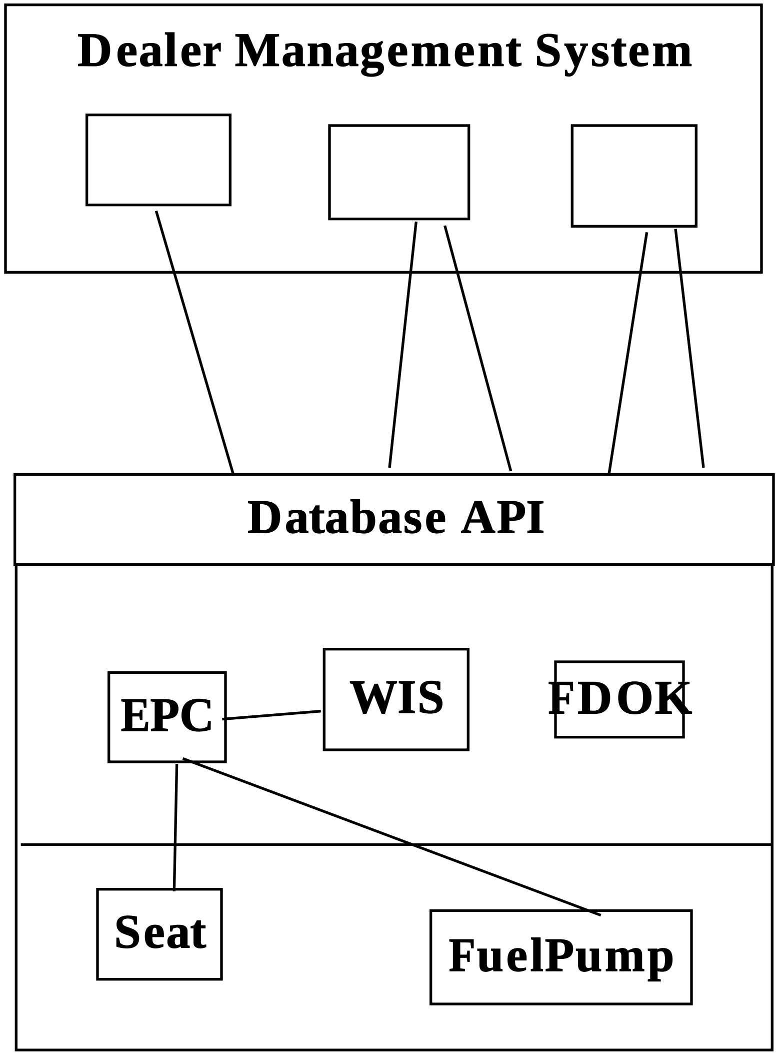
Open vs Closed Architecture

- Open architecture:
  - **Any dealer management system can call any component or class operation of the PAID databases.**
- Why is this good?
  - **Efficiency**
- Why is this bad?
  - **Can’t expect the client to understand how the subsystem works or any of the complex relationships that may exist within the subsystem.**
  - **We can (pretty much) be assured that the subsystem will be misused, leading to non-portable code**

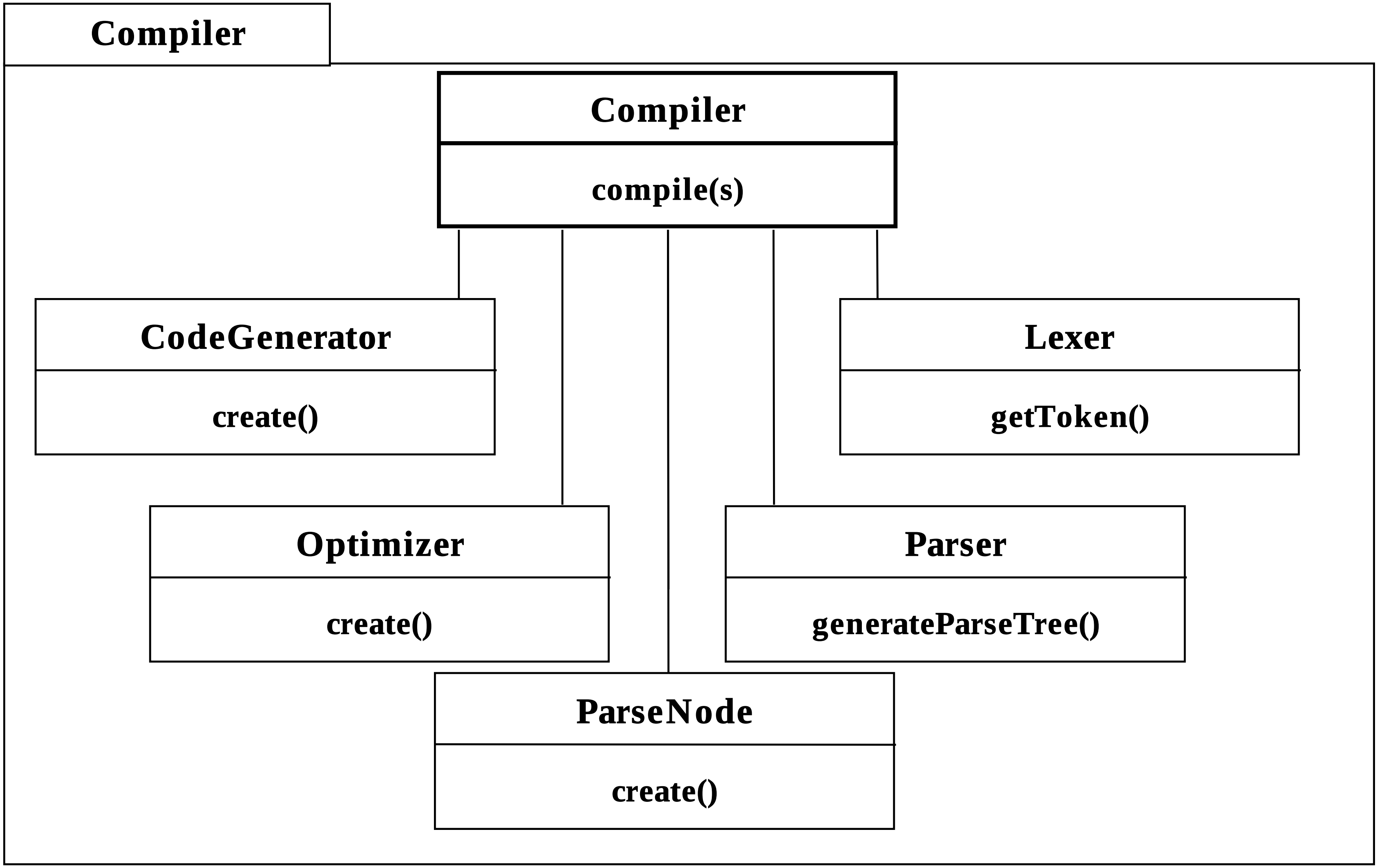


Realizing a Closed Architecture with a Facade

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

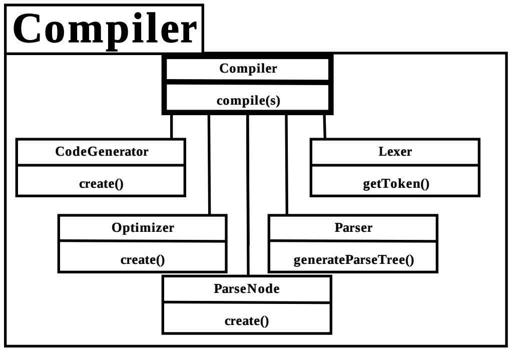


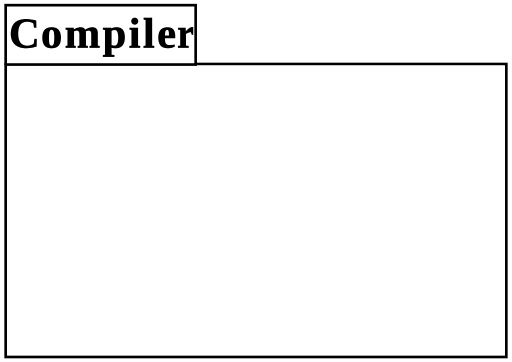
Realizing a Compiler with a Facade pattern



UML Notation: 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**

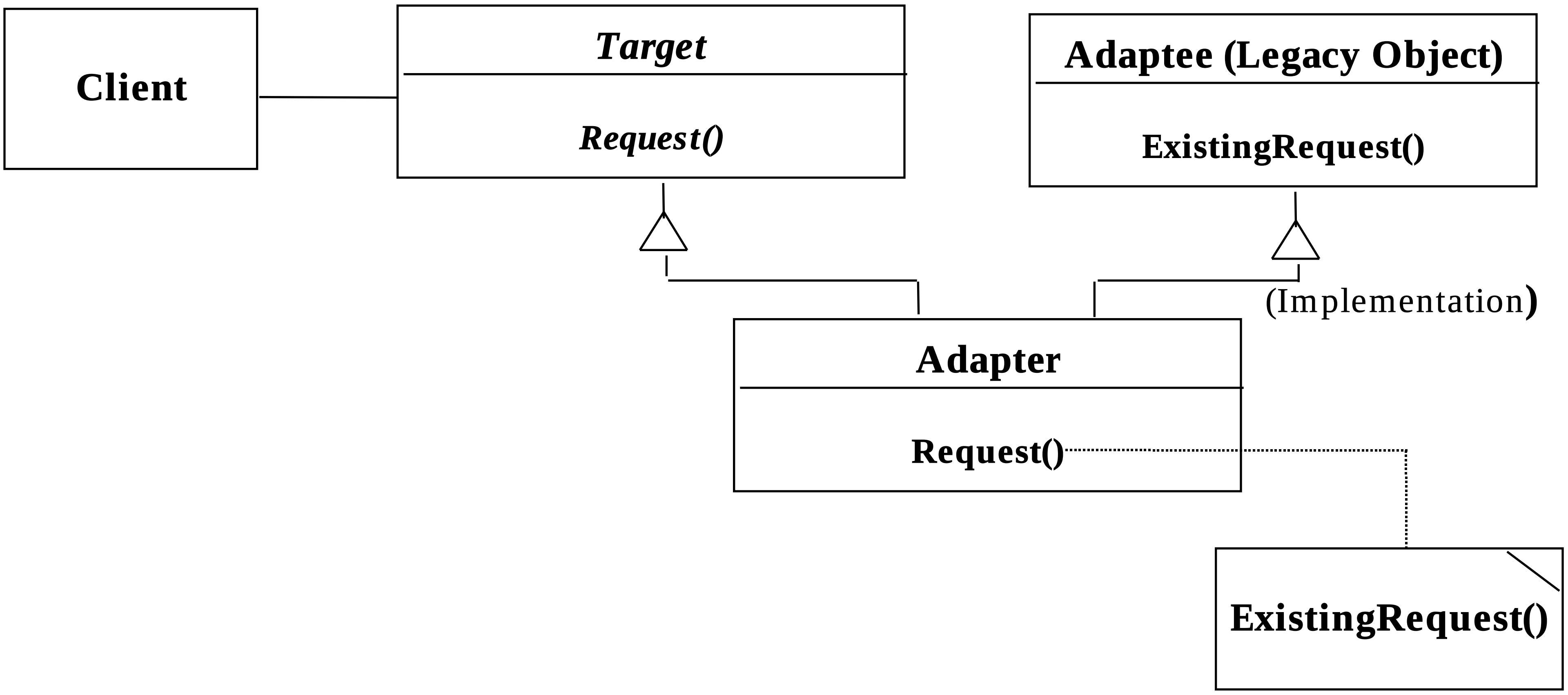




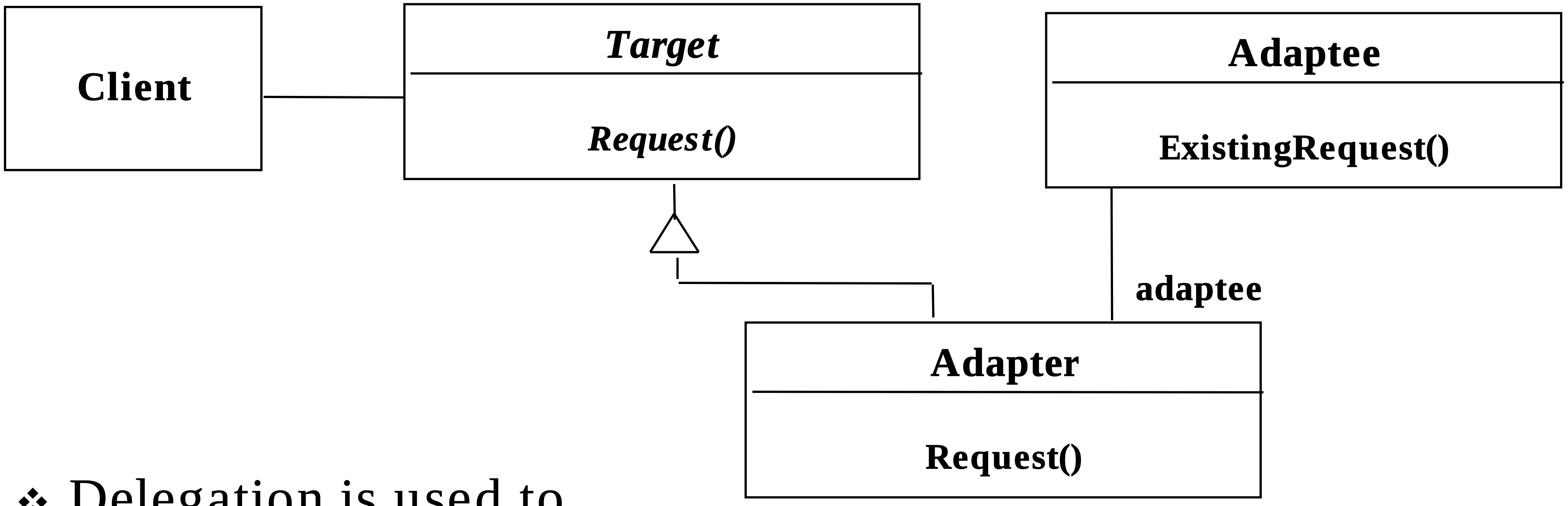
Adapter Pattern

- “Convert the interface of a class into another interface expected by the client. 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

Class Adapter Pattern (based on Multiple Inheritance)

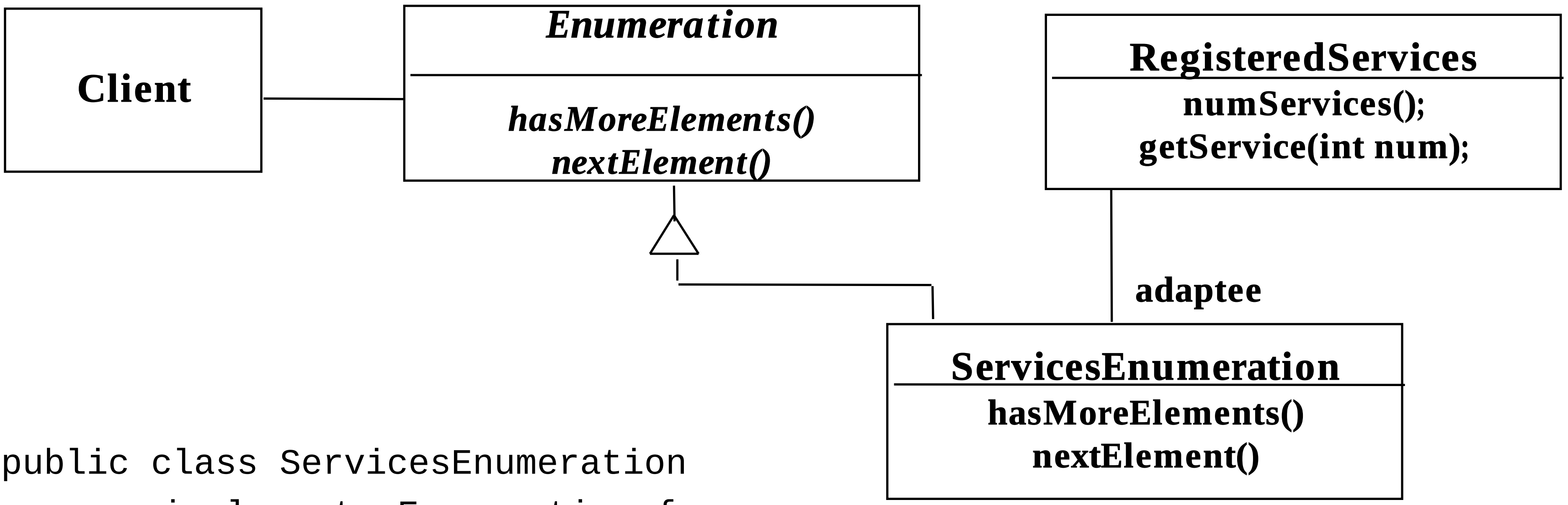


Adapter pattern uses delegation and inheritance



- Delegation is used to bind an **Adapter** and an **Adaptee**
- Interface inheritance is use to specify the interface of the **Adapter** class.
- **Adaptee**, usually called legacy system, pre-exists 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++);
  }
```

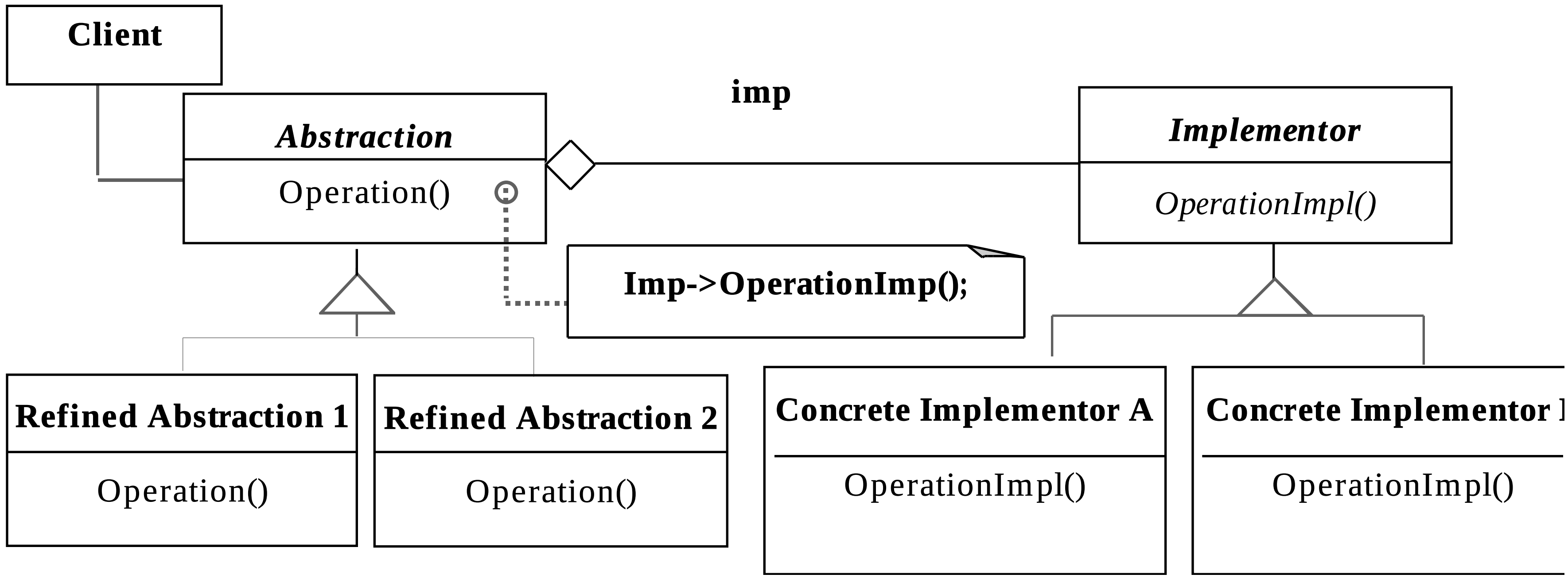
Bridge Pattern

- 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.

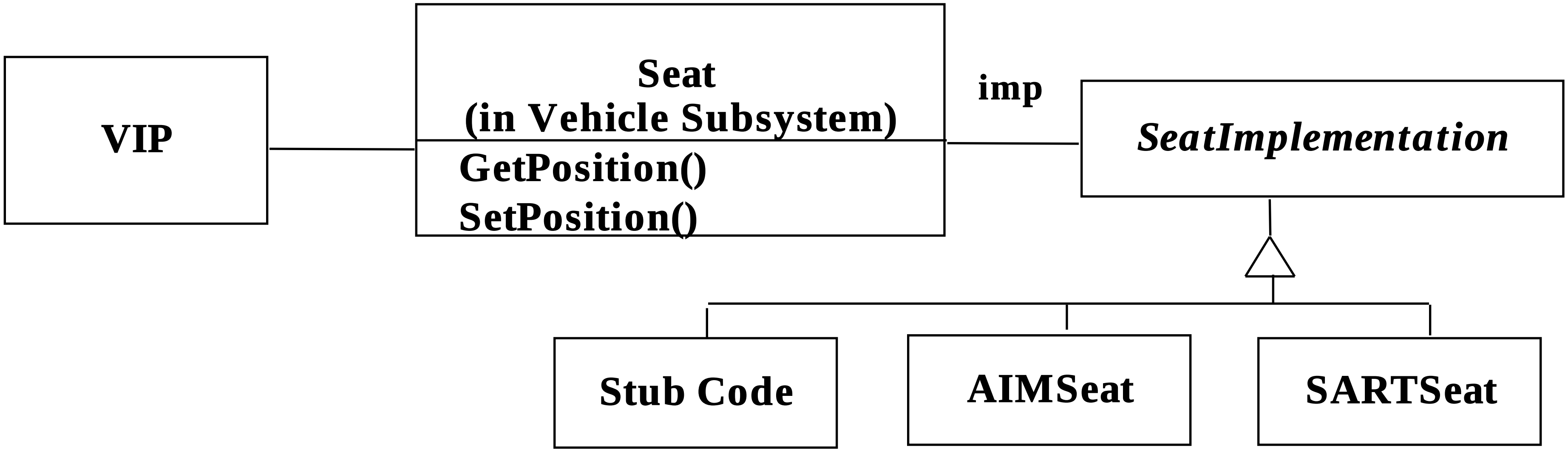
- Allows different implementations of an interface to be decided upon dynamically.

Bridge Pattern(151)



Using a Bridge

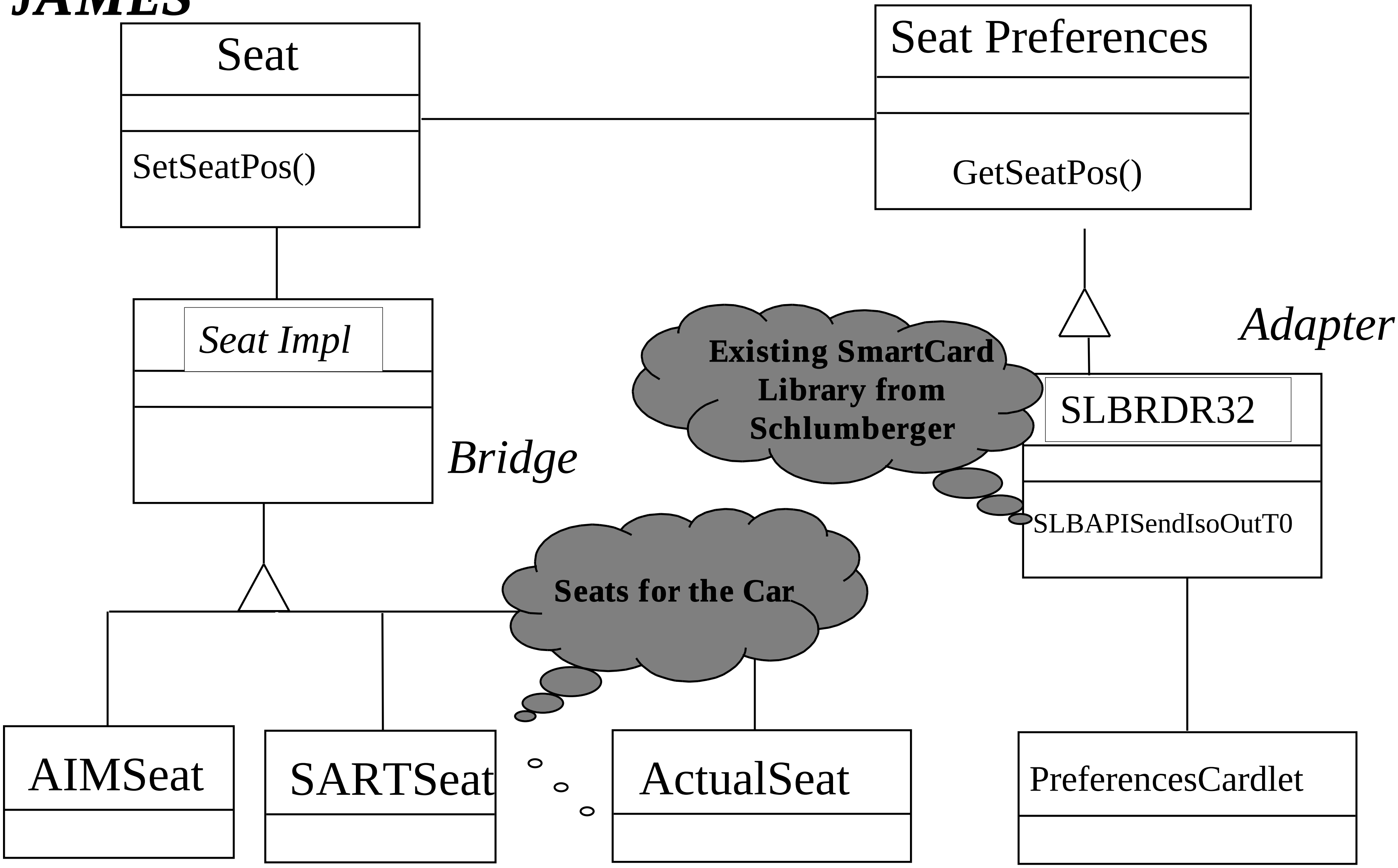
- ❖ Use the bridge pattern to provide multiple implementations under the same interface.
- ❖ Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- ❖ JAMES Project (WS 97-98): if seat data is required to be read, but the seat is not yet implemented, not yet known or only available by a simulation, provide a bridge:



Adapter vs Bridge

- Both used to hide the details of the underlying
- 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



When do I use the Façade?
When do I use an adapter or a bridge?

- A facade pattern are used by all subsystems in the 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).**

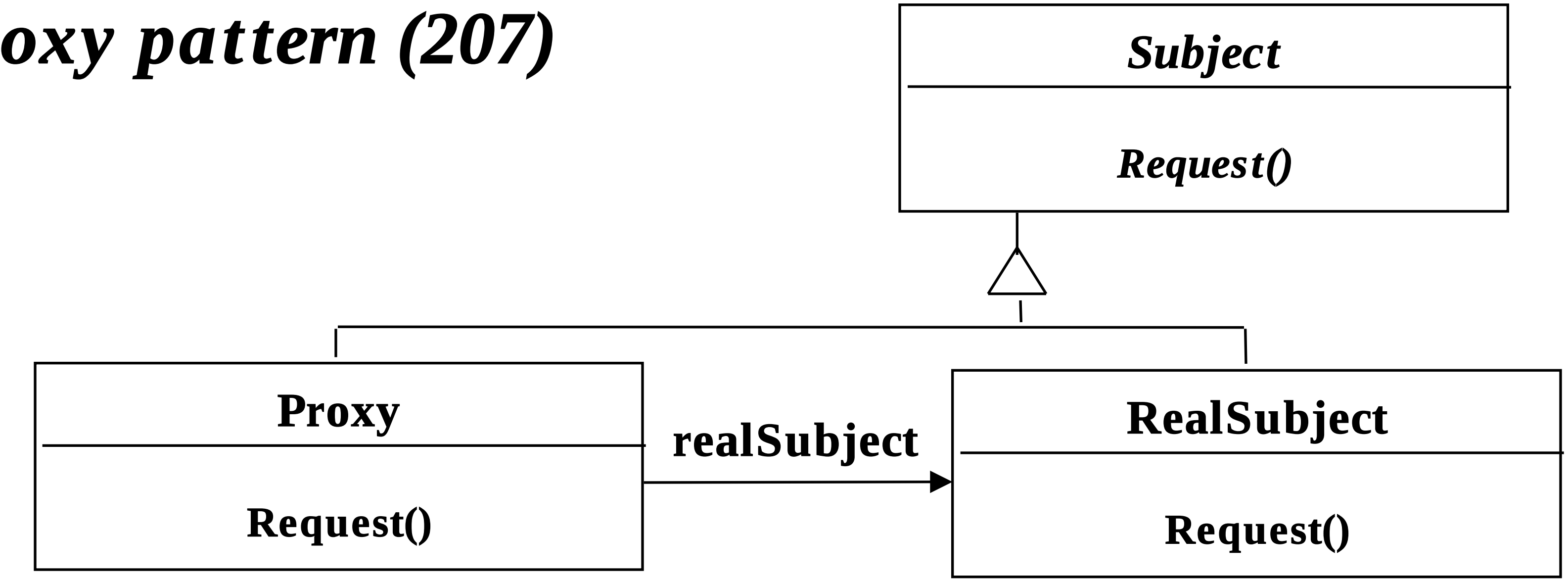
Proxy Pattern: Motivation

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

Proxy Pattern

- What is expensive?
  - **Object Creation**
  - **Object Initialization**
- Defer creation and initialization to the time you need the object
- Reduce the cost of access to objects
  - **Use 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 (207)

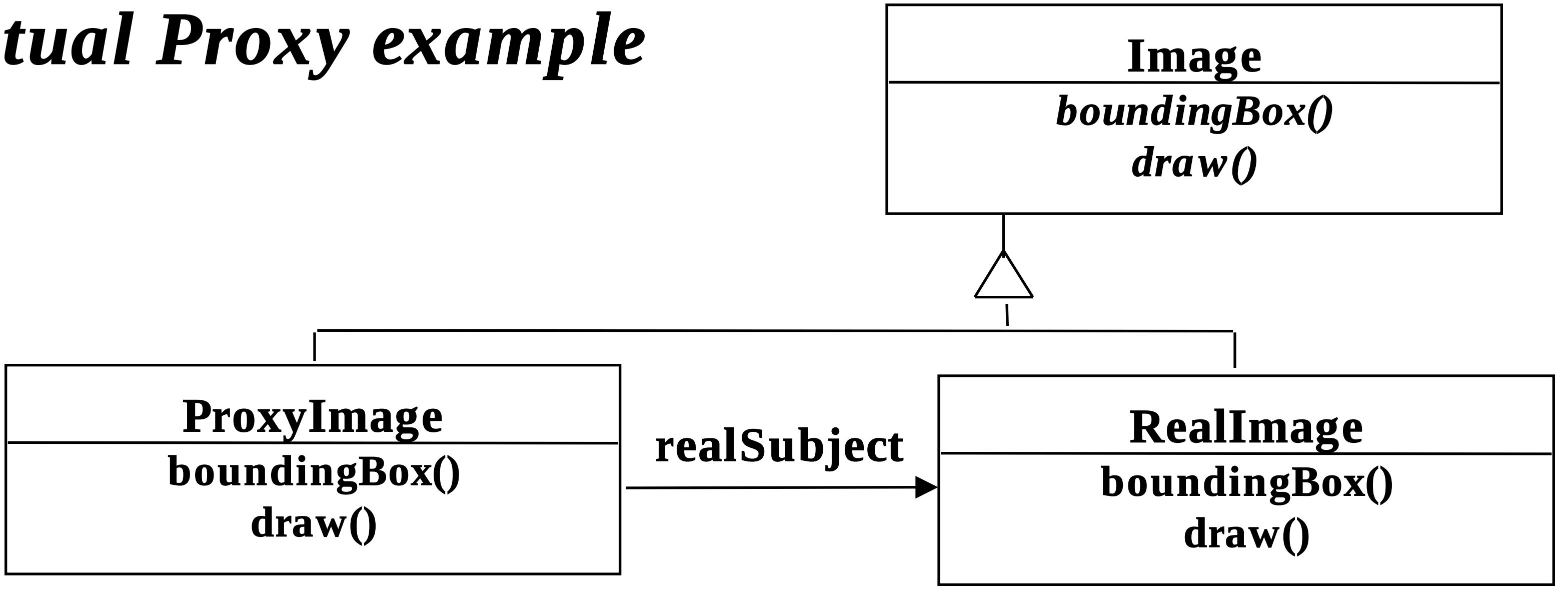


- Interface inheritance is used to specify the interface shared by **Proxy** and **RealSubject.**
- Delegation is used to catch and forward accesses to the **RealSubject**.
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.

Proxy Applicability in PAID

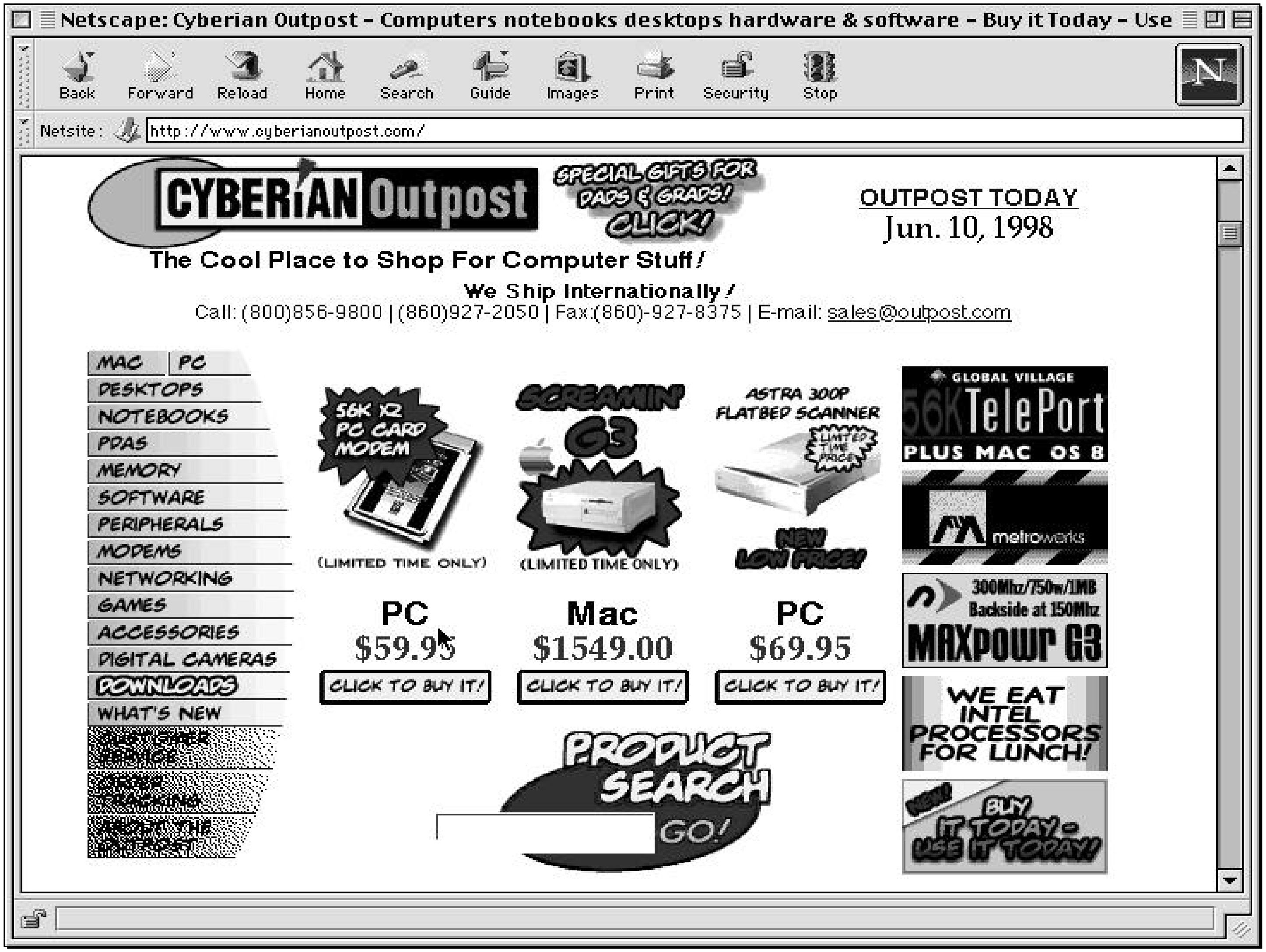
- Remote Proxy
  - **Local representative for an object in a different address space**
  - **Caching of information: Good if information does not change too often**
- Virtual Proxy
  - **Object is too expensive to create or too expensive to download**
  - **Proxy is standin**
- Protection Proxy
  - **Proxy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

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



Netscape: Cyberian Outpost - Computers notebooks desktops hardware & software - Buy it Today - Use

http://www.cyberianoutpost.com/

CYBERiAN Outpost

SPECIAL GIFTS FOR DADS & GRADS! CLICK!

OUTPOST TODAY
Jun. 10, 1998

The Cool Place to Shop For Computer Stuff!

We Ship Internationally!

Call: (800)856-9800 | (860)927-2050 | Fax:(860)-927-8375 | E-mail: sales@outpost.com

56K X2 PC CARD MODEM
(LIMITED TIME ONLY)
PC
$59.95

SCREAMIN' G3
(LIMITED TIME ONLY)
Mac
$1549.00

ASTRA 300P FLATBED SCANNER
LIMITED TIME PRICE!
NEW LOW PRICE!
PC
$69.95

GLOBAL VILLAGE 56K TelePort PLUS MAC OS 8

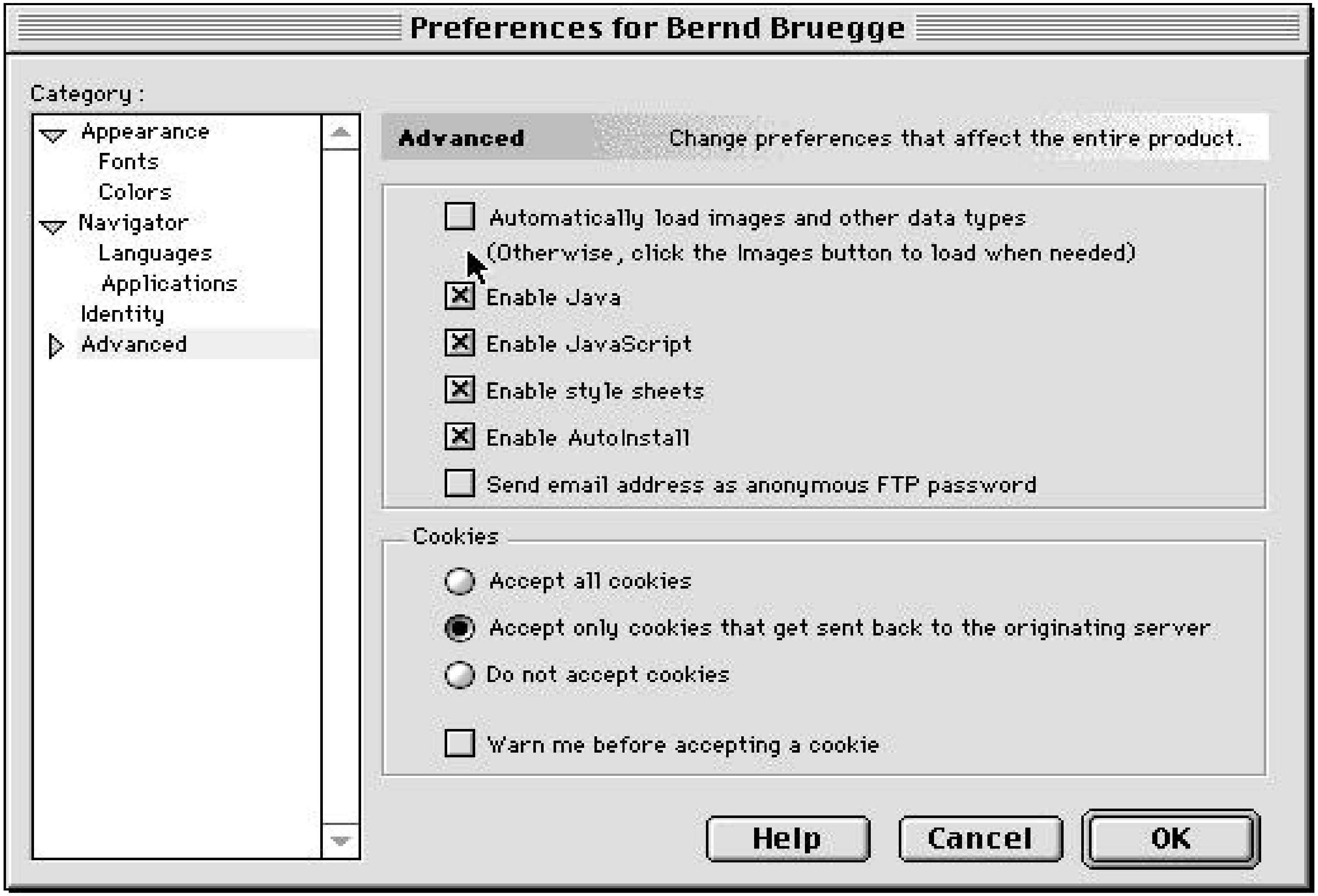
metrowerks

300Mhz/750w/1MB Backside at 150Mhz MAXpowr G3

WE EAT INTEL PROCESSORS FOR LUNCH!

NEW! BUY IT TODAY - USE IT TODAY!

Controlling Access



Preferences for Bernd Bruegge

Category :

Advanced Change preferences that affect the entire product.

☐ Automatically load images and other data types (Otherwise, click the Images button to load when needed)

☒ Enable Java

☒ Enable JavaScript

☒ Enable style sheets

☒ Enable AutoInstall

☐ Send email address as anonymous FTP password

Cookies

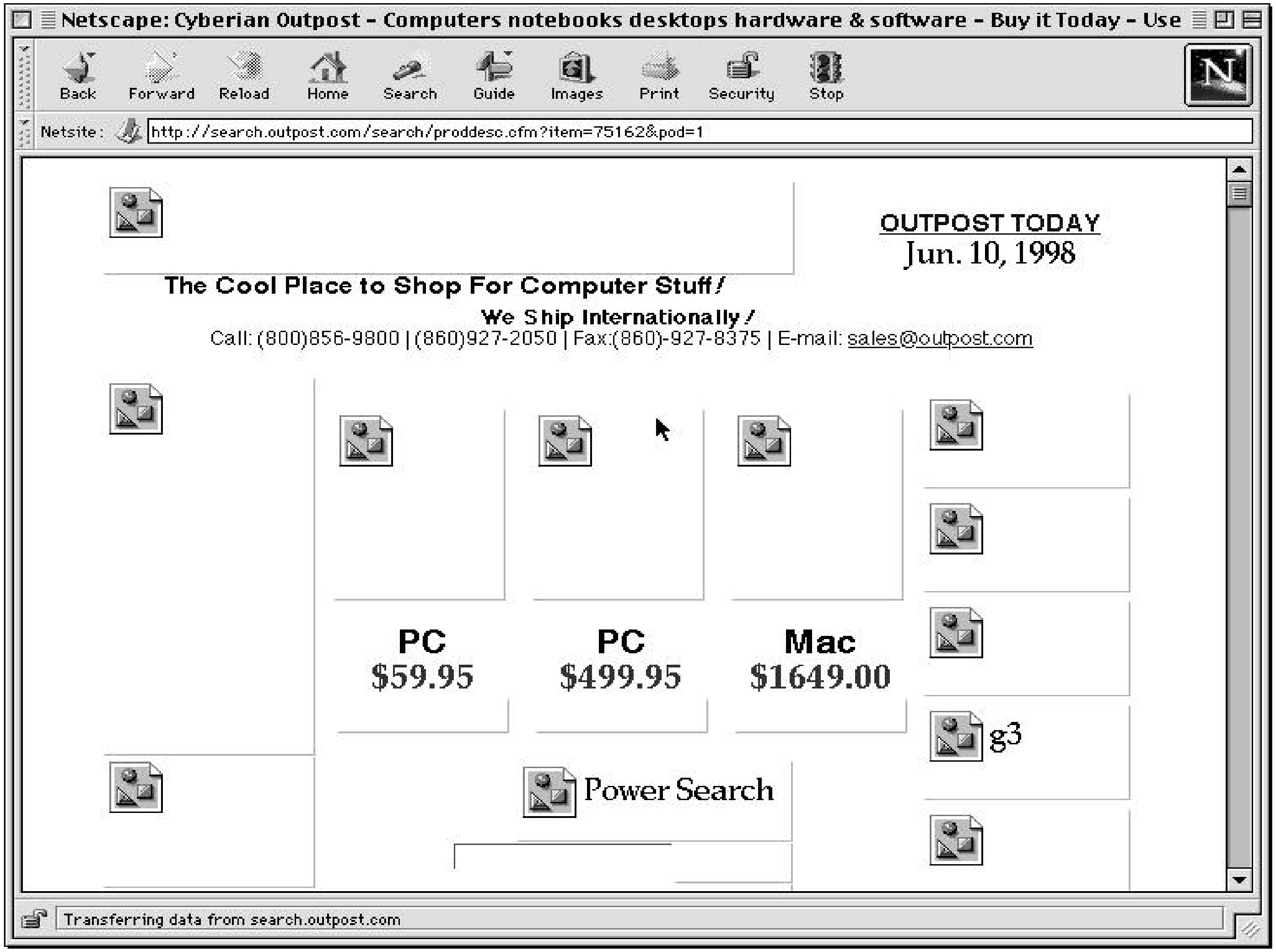
○ Accept all cookies

● Accept only cookies that get sent back to the originating server

○ Do not accept cookies

☐ Warn me before accepting a cookie

After



Netscape: Cyberian Outpost - Computers notebooks desktops hardware & software - Buy it Today - Use

http://search.outpost.com/search/proddesc.cfm?item=75162&pod=1

OUTPOST TODAY
Jun. 10, 1998

The Cool Place to Shop For Computer Stuff!

We Ship Internationally!

Call: (800)856-9800 | (860)927-2050 | Fax:(860)-927-8375 | E-mail: sales@outpost.com

PC
$59.95

PC
$499.95

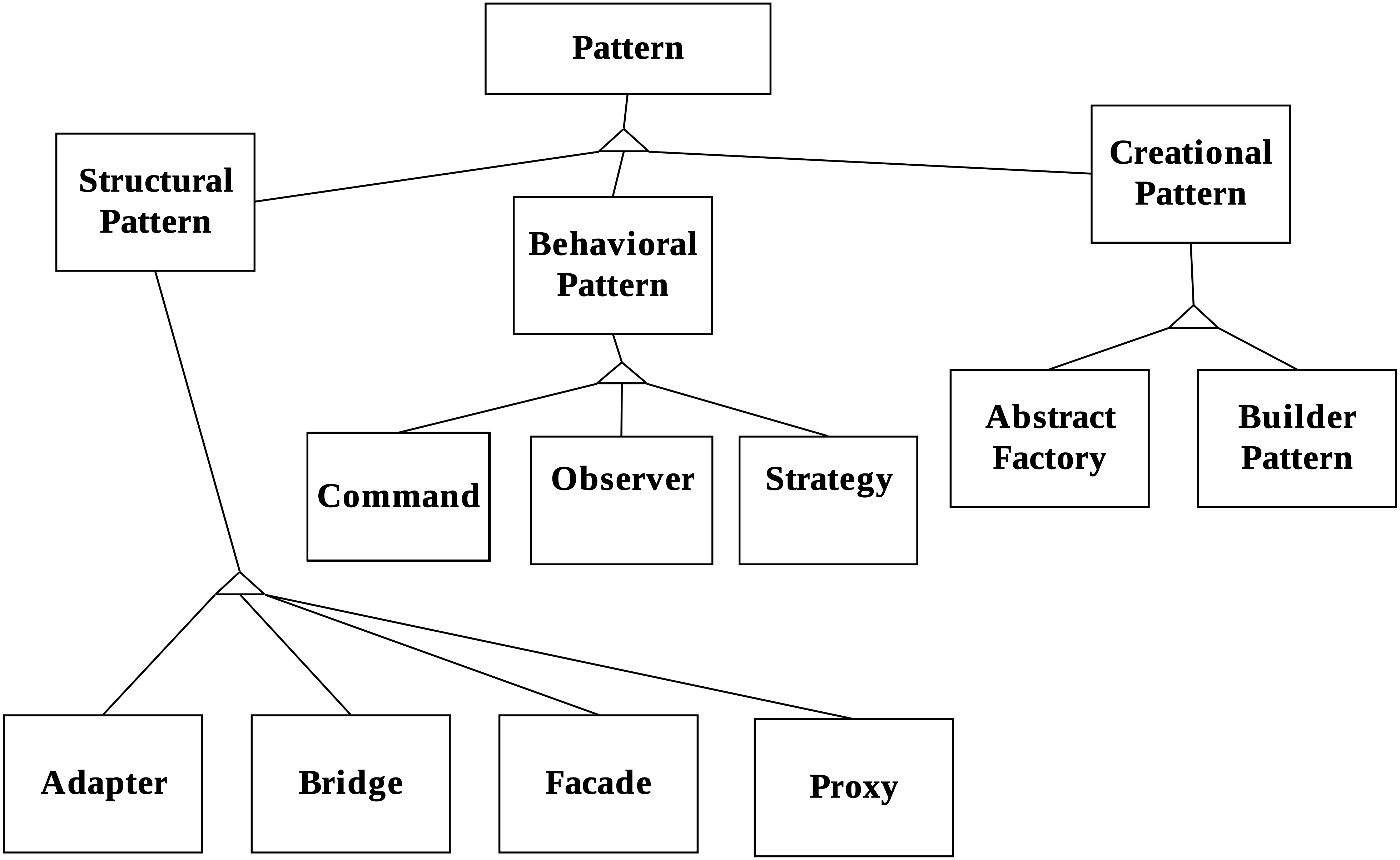
Mac
$1649.00

Transferring data from search.outpost.com

Towards a Pattern Taxonomy

- Structural Patterns
  - **Adapters, Bridges, Facades, 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**
- Behavioral Patterns
  - **Concerned with algorithms and the assignment of responsibilies 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.**

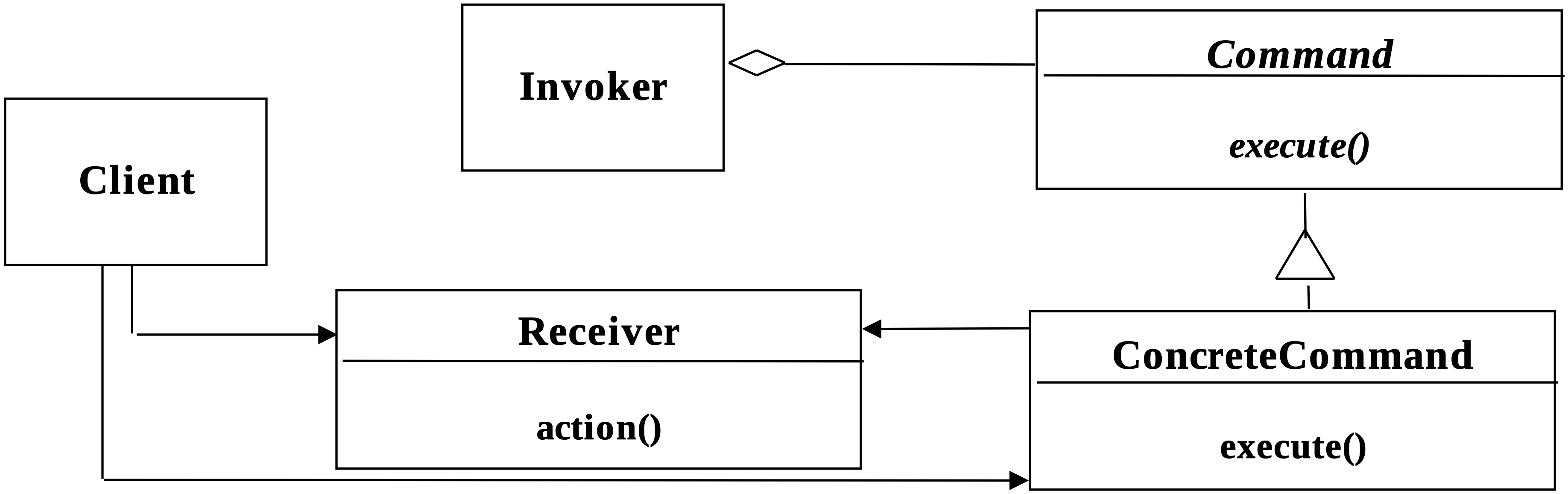
A Pattern Taxonomy



Command Pattern: Motivation

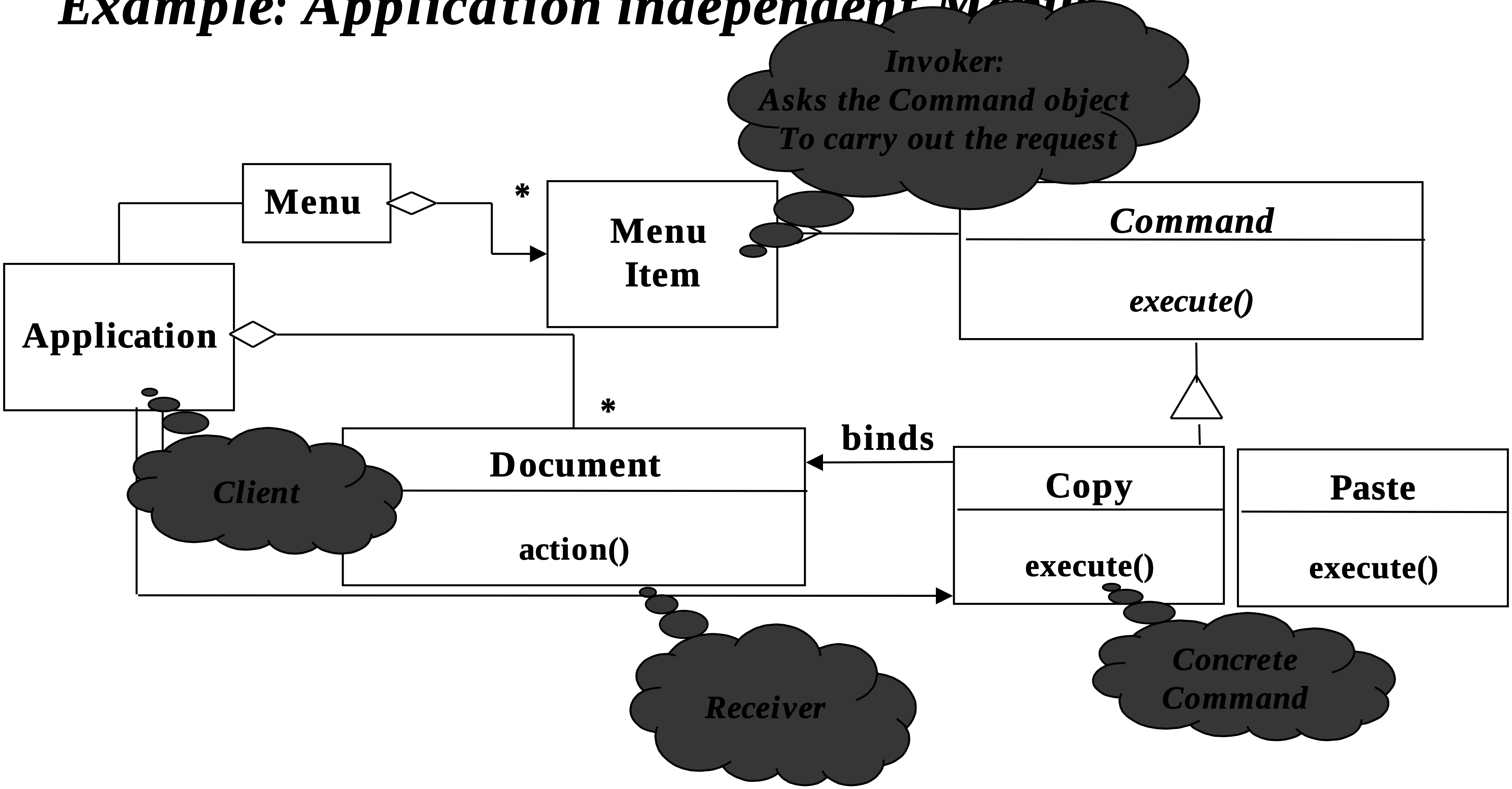
- You want to build a user interface
- You want to provide menus
- You want to make the user interface reusable across many applications
  - **You cannot hardcode the meanings of the menus for the various applications**
  - **The applications only know what has to be done when a menu is selected.**
-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

Command pattern (238)



- The **Invoker** offers a variety of commands (“execute”, “copy”, “paste, “undo”)**.**
- **ConcreteCommand** implements execute() by calling corresponding operation(s) in **Receiver.**
- **Receiver** knows how to perform the operation.
- **Client** instantiates the **ConcreteCommand**s and sets its **Receiver.** (**Client** is a bad name, should be **Application**….)

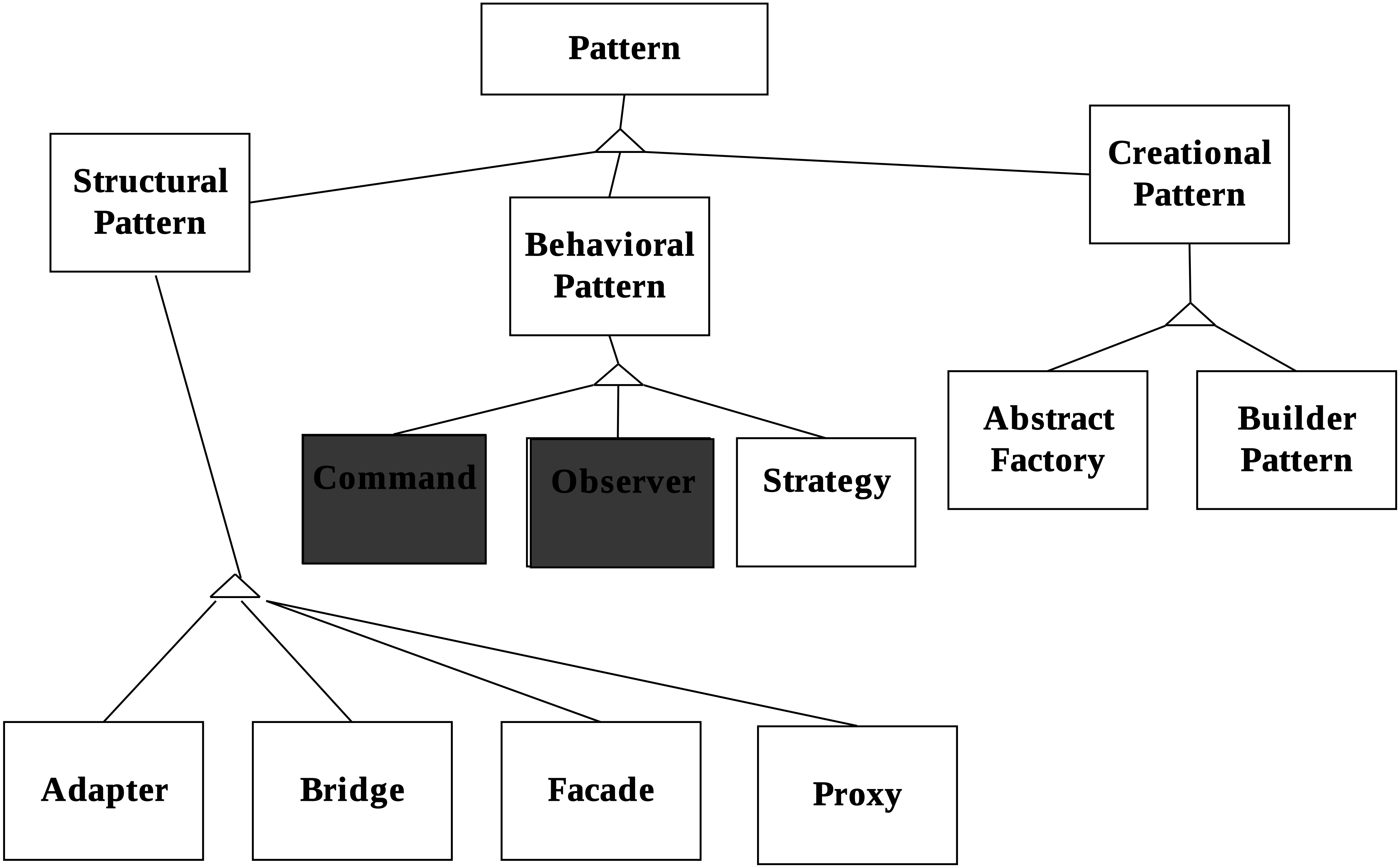
Example: Application independent Menus



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.” (p. 233)**
- Uses:
  - **Undo queues**
  - **Database transaction buffering**

A Pattern Taxonomy



Pattern

Structural Pattern

Behavioral Pattern

Creational Pattern

Command

Observer

Strategy

Abstract Factory

Builder Pattern

Adapter

Bridge

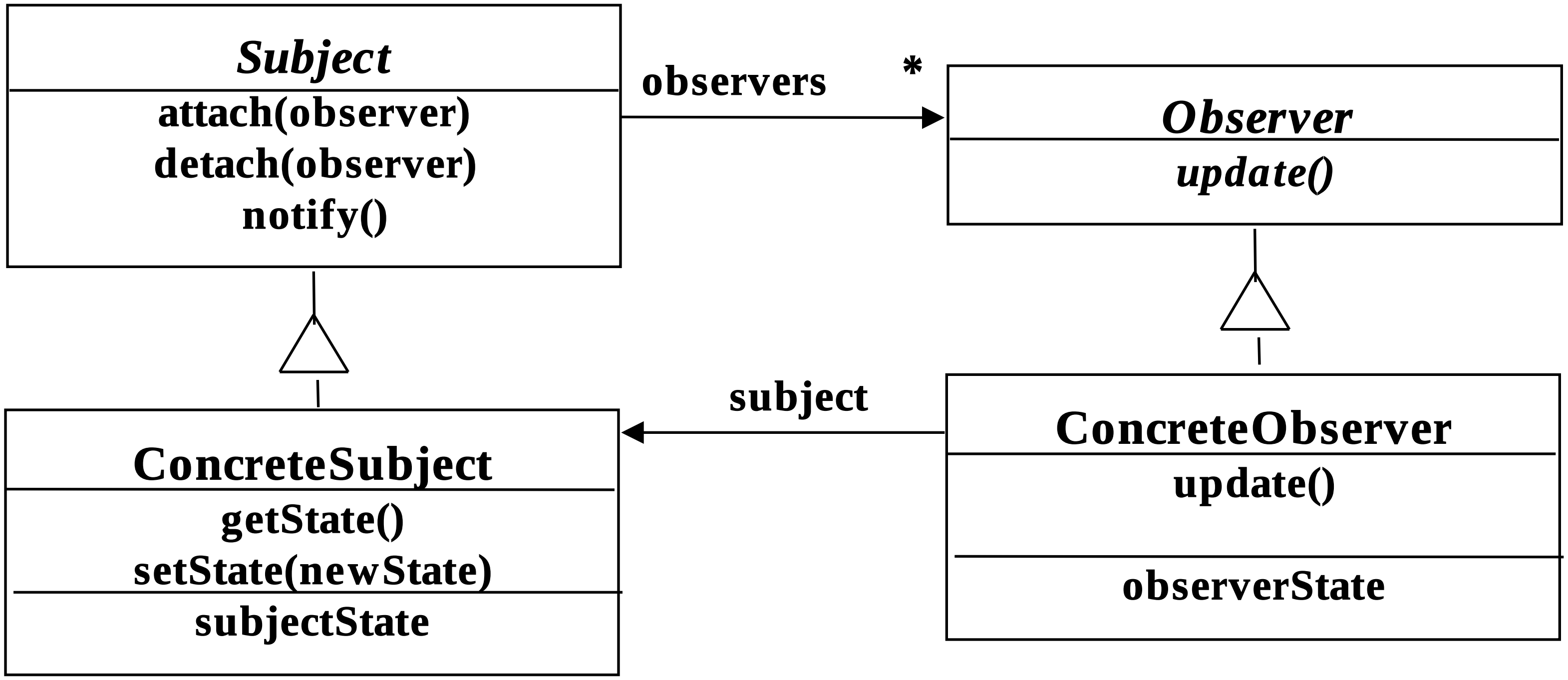
Facade

Proxy

Observer pattern

- “Defines 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 (293)

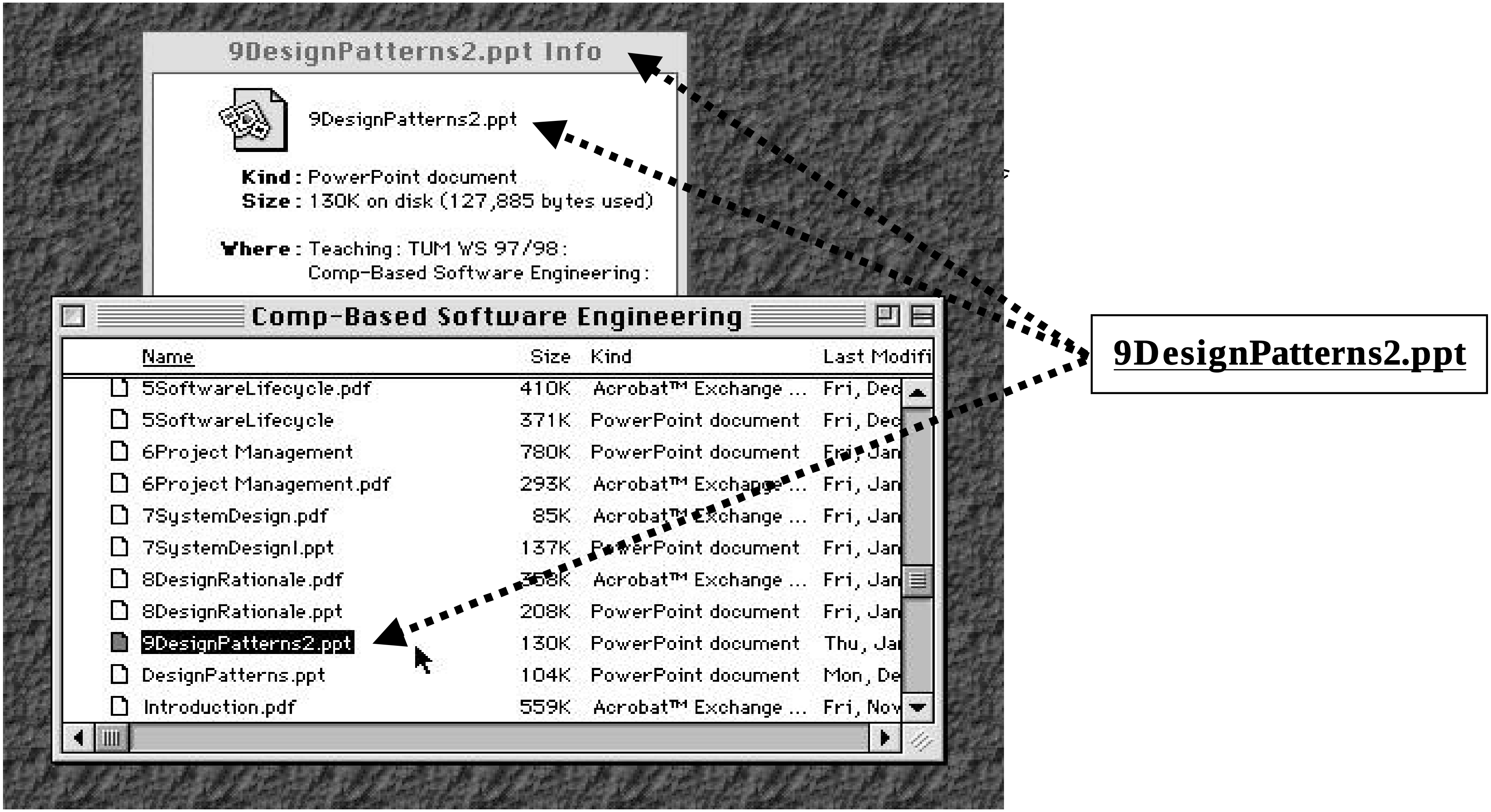


- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) ***not*** an interface.

Observer pattern Example

Observers

Subject



9DesignPatterns2.ppt Info

9DesignPatterns2.ppt

Kind : PowerPoint document
Size : 130K on disk (127,885 bytes used)

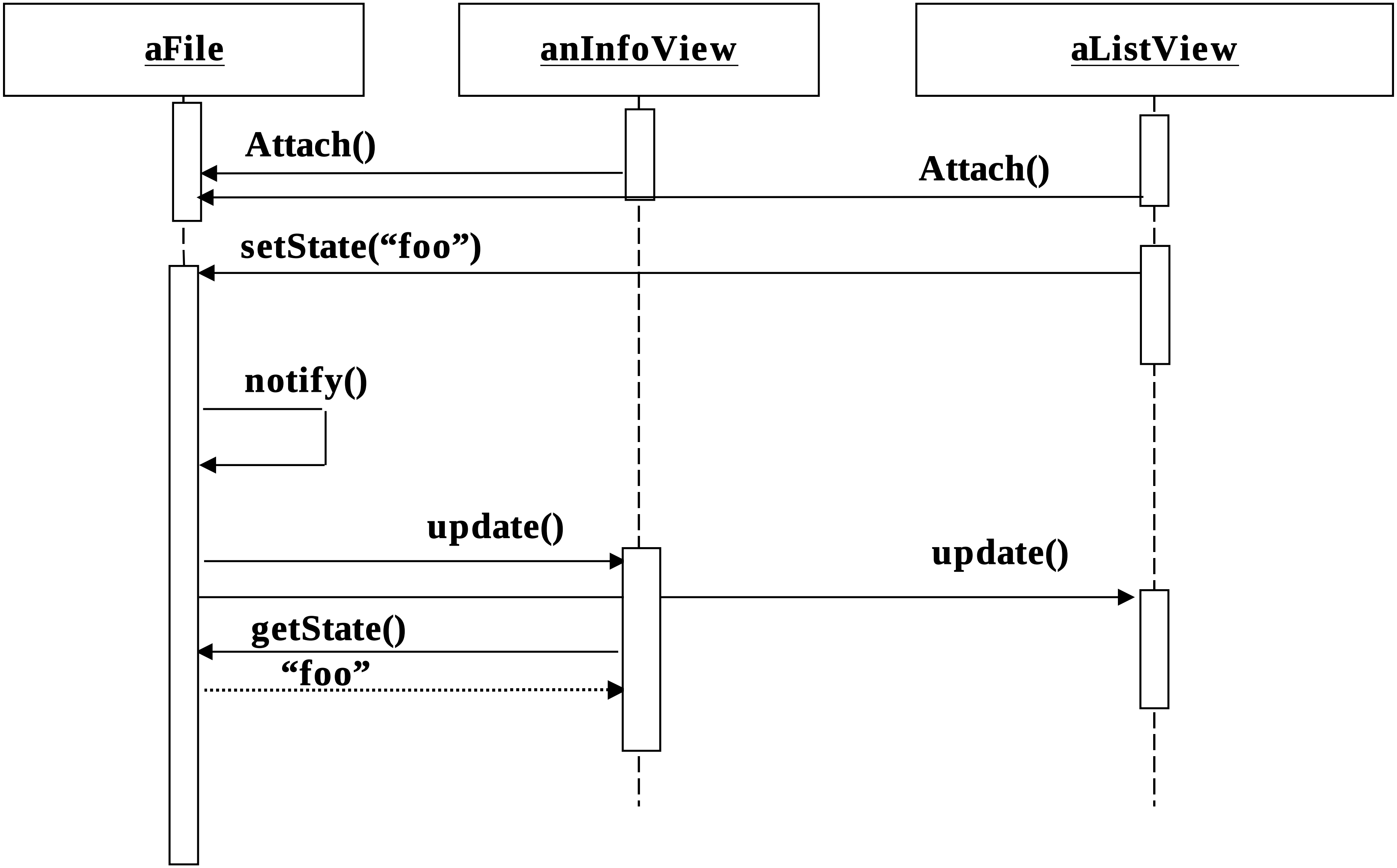
Where : Teaching : TUM WS 97/98 :
Comp-Based Software Engineering :

Comp-Based Software Engineering

Name	Size	Kind	Last Modifi
5SoftwareLifecycle.pdf	410K	Acrobat™ Exchange ...	Fri, Dec
5SoftwareLifecycle	371K	PowerPoint document	Fri, Dec
6Project Management	780K	PowerPoint document	Fri, Jan
6Project Management.pdf	293K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesign.pdf	85K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesignI.ppt	137K	PowerPoint document	Fri, Jan
8DesignRationale.pdf	358K	Acrobat™ Exchange ...	Fri, Jan
8DesignRationale.ppt	208K	PowerPoint document	Fri, Jan
9DesignPatterns2.ppt	130K	PowerPoint document	Thu, Ja
DesignPatterns.ppt	104K	PowerPoint document	Mon, De
Introduction.pdf	559K	Acrobat™ Exchange ...	Fri, Nov

9DesignPatterns2.ppt

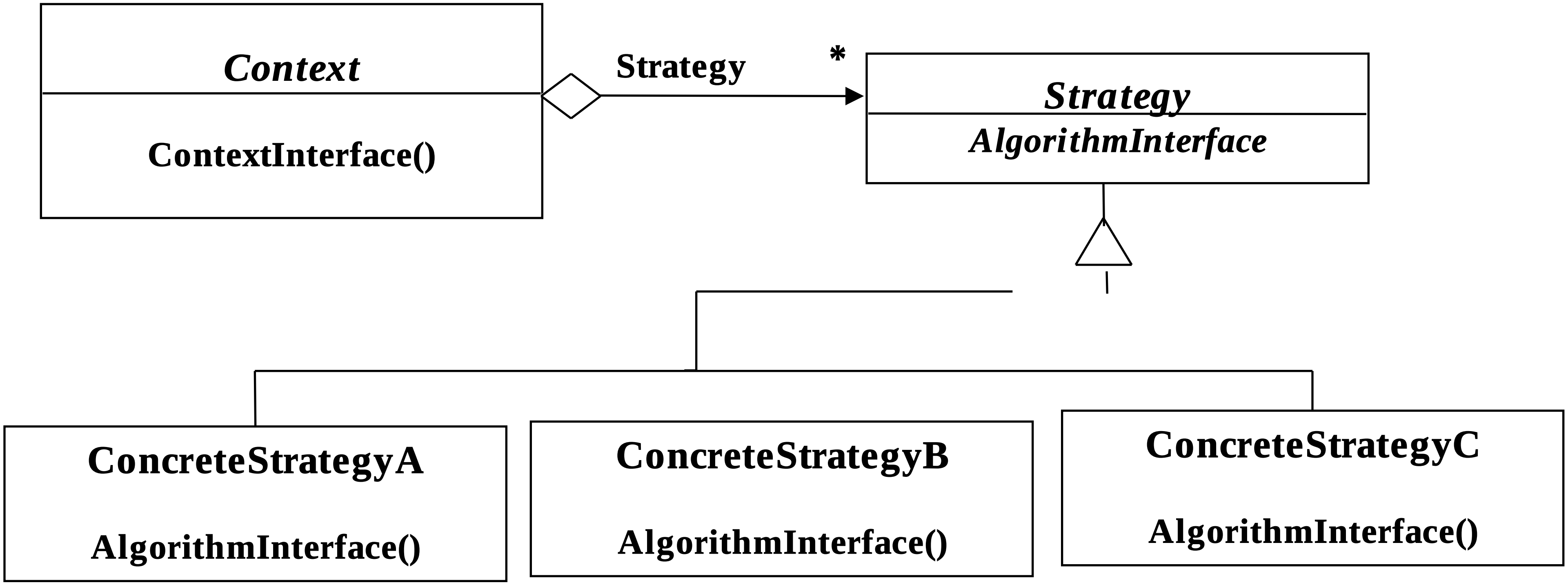
Sequence diagram for scenario: Change filename to "foo"



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 (315)



Context

ContextInterface()

Strategy

*

Strategy

AlgorithmInterface

ConcreteStrategyA

AlgorithmInterface()

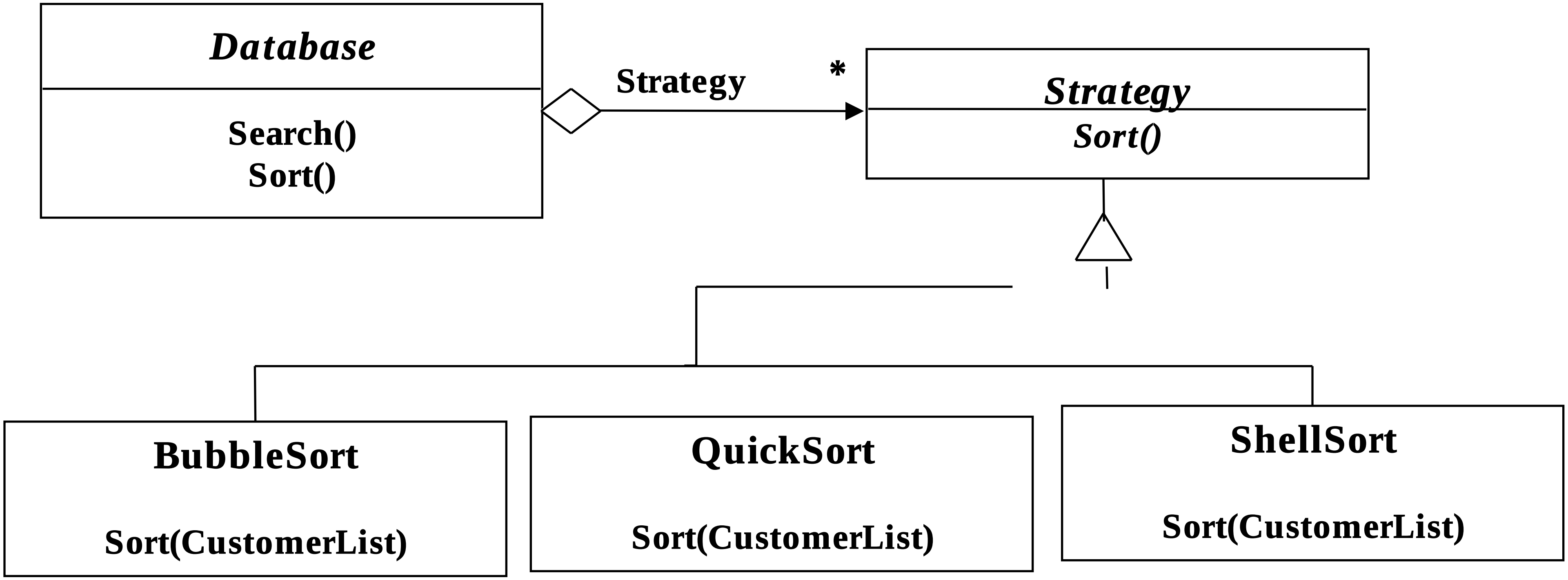
ConcreteStrategyB

AlgorithmInterface()

ConcreteStrategyC

AlgorithmInterface()

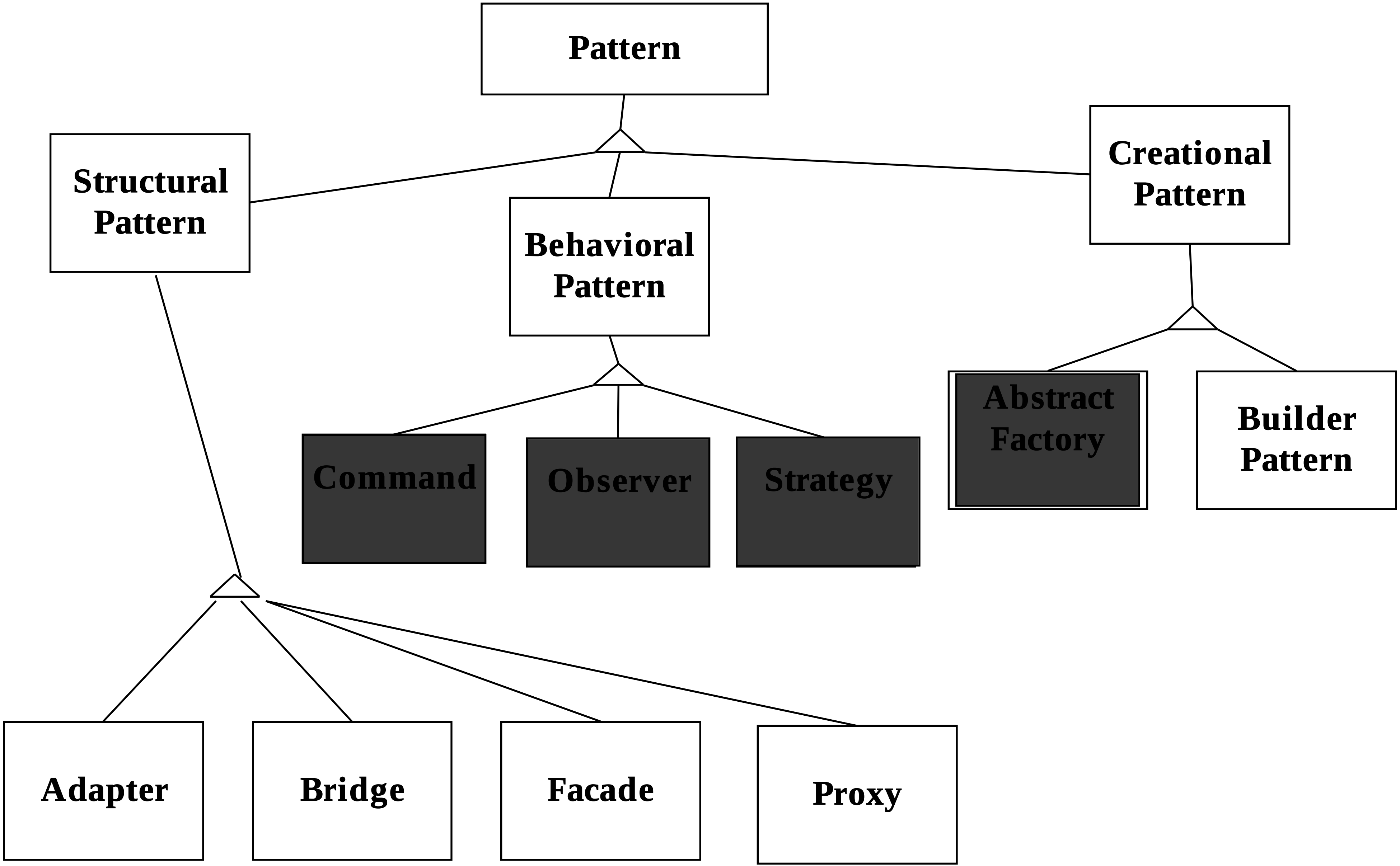
Applying a Strategy Pattern in a Database Application



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

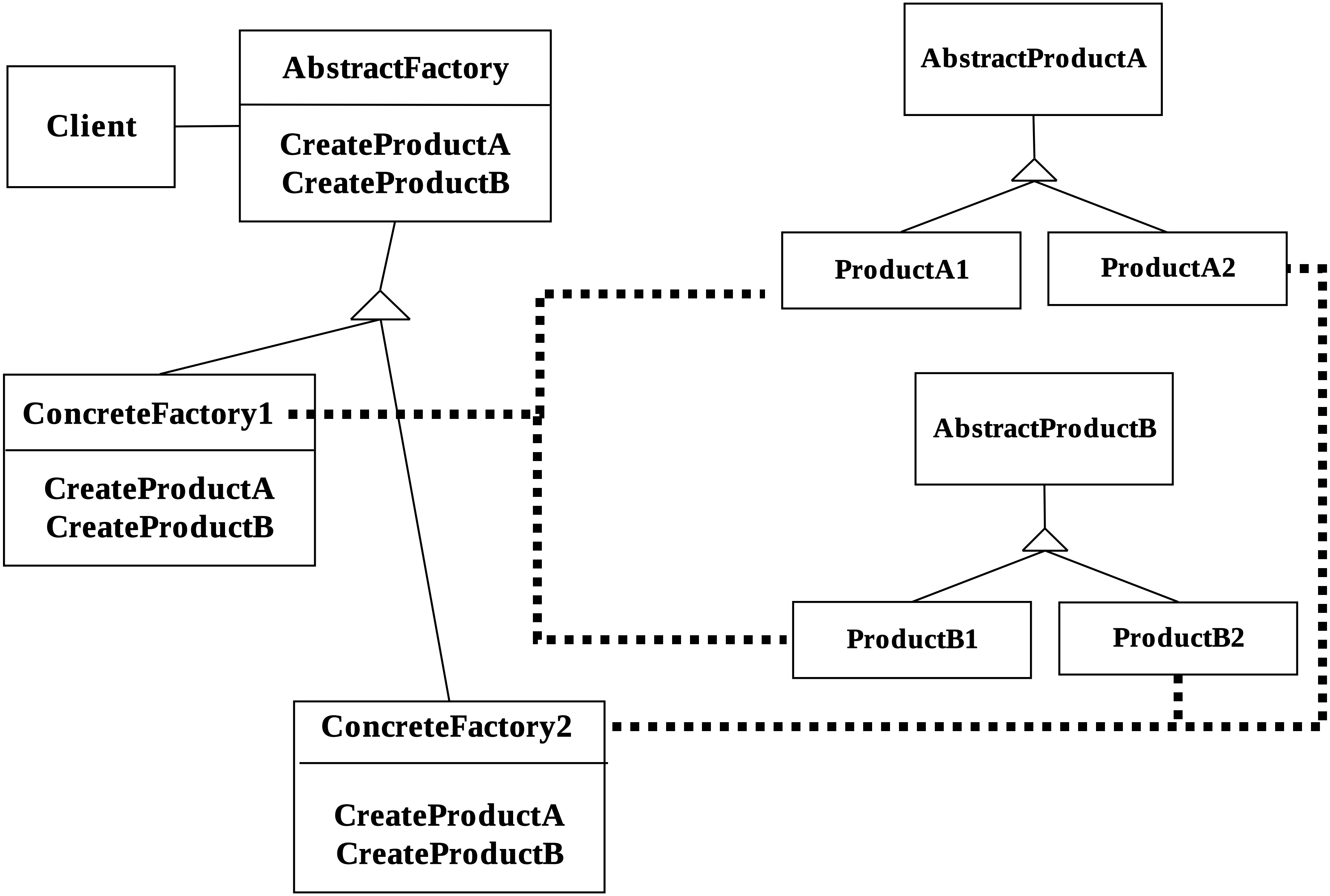
A Pattern Taxonomy



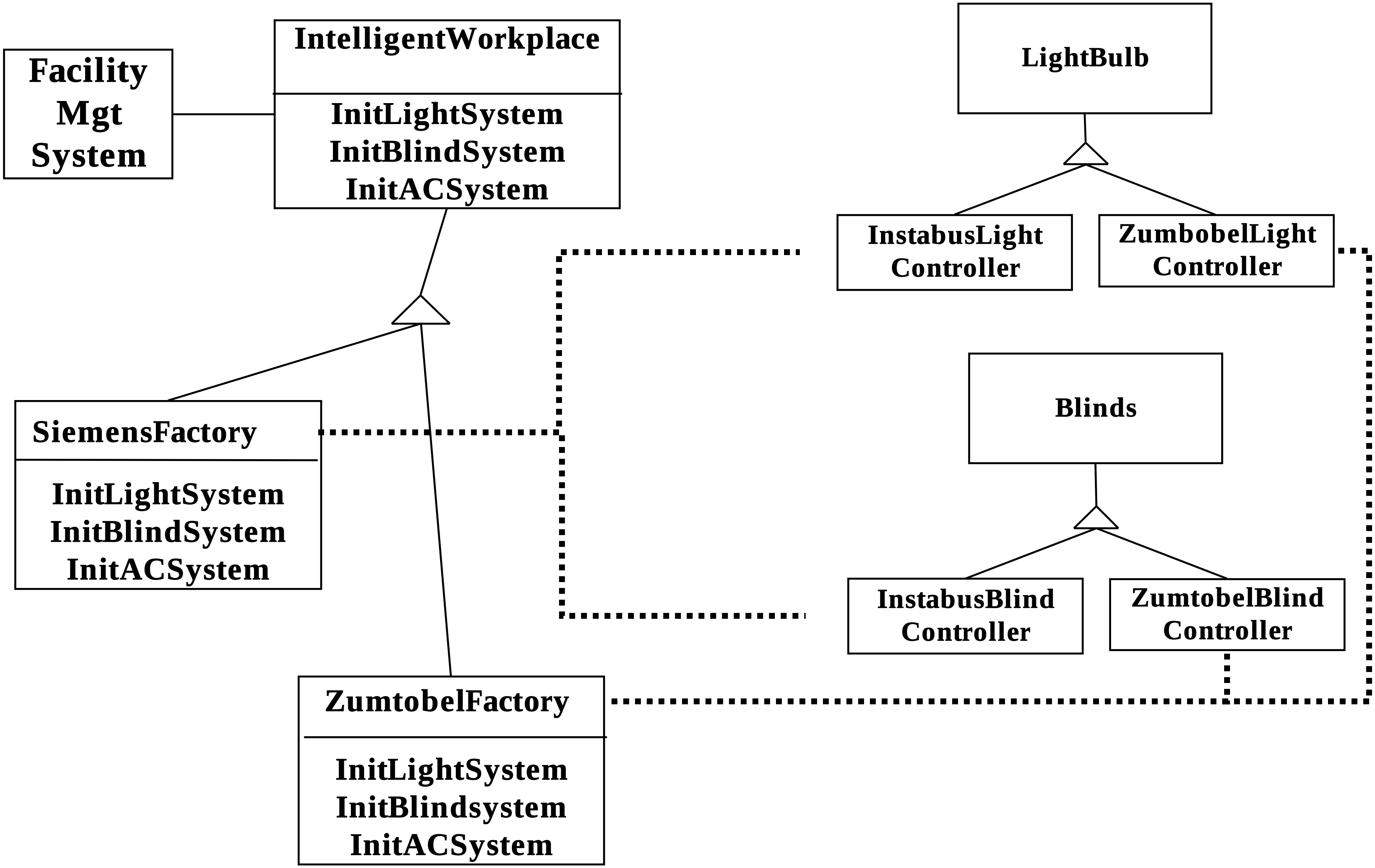
Abstract Factory Motivation

- Implement 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?**
- Implement 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 (87)



Example: OWL System for the The Intelligent Workplace at Carnegie Mellon University (15-413 Fall 96)



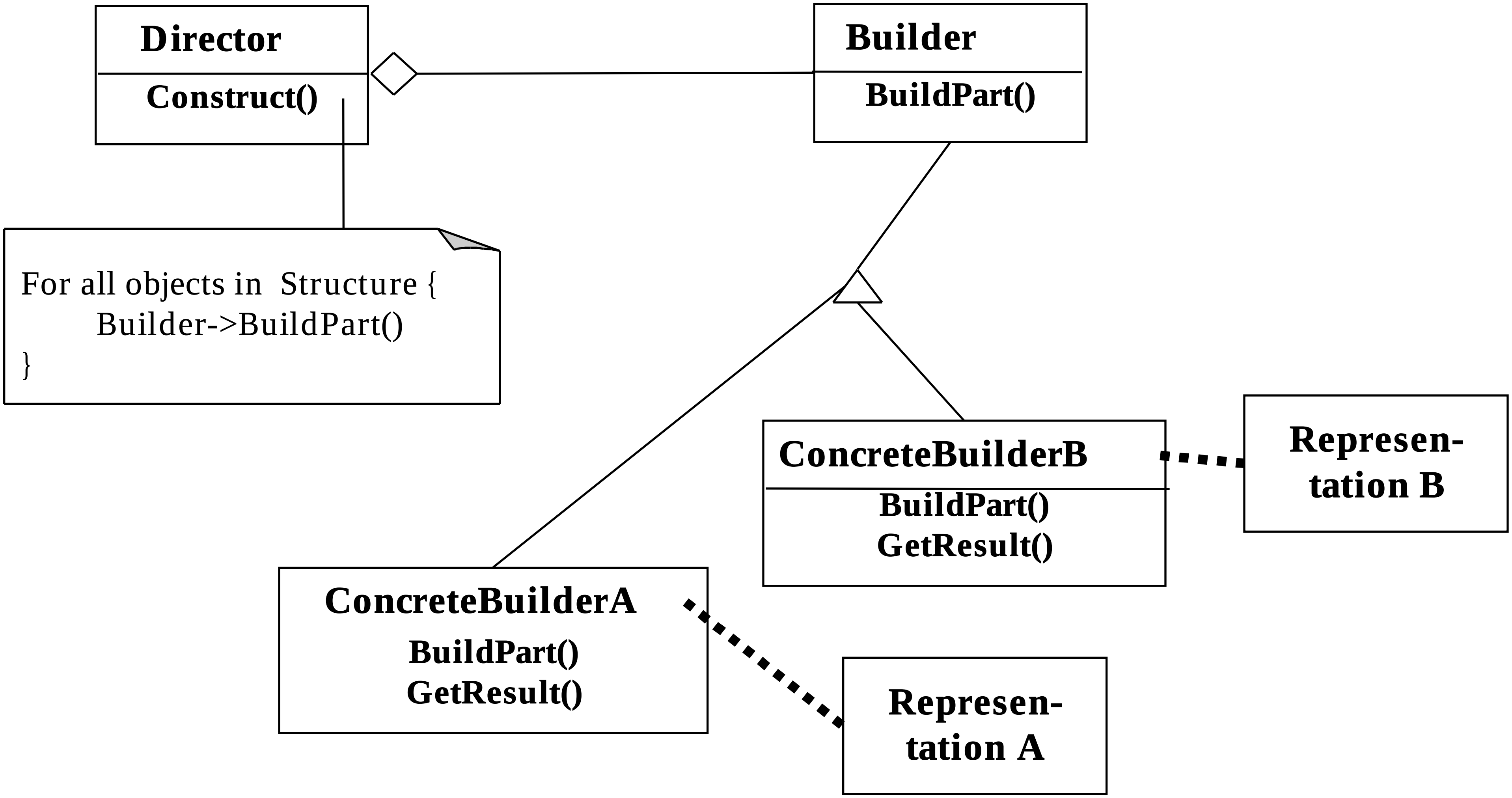
Applicability for Abstract Factory Pattern

- Independence from Initialization or Represenation:
  - **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.**

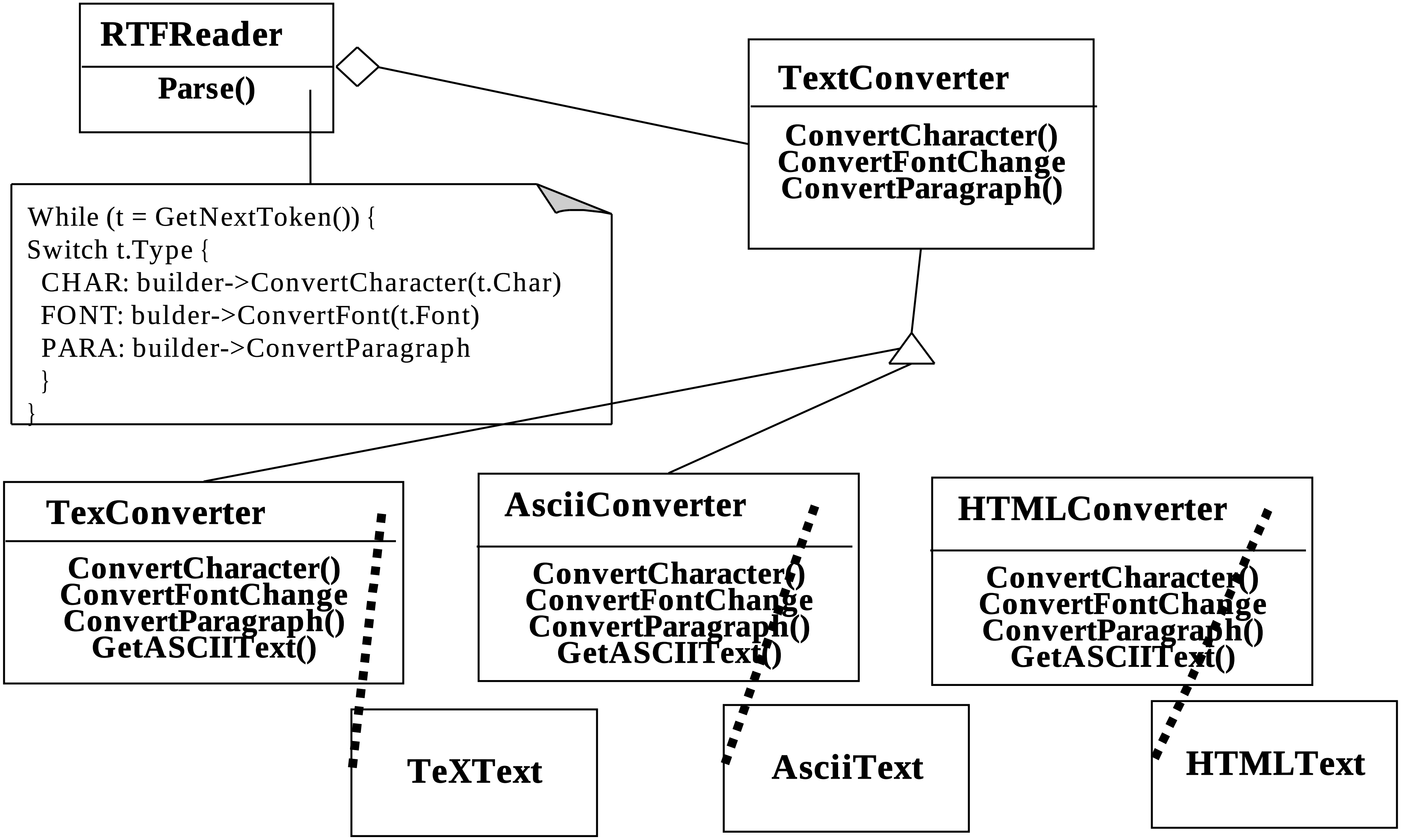
Builder Pattern Motivation

- 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 (97)



Example



RTFReader

Parse()

```
While (t = GetNextToken()) {
Switch t.Type {
  CHAR: builder->ConvertCharacter(t.Char)
  FONT: bulder->ConvertFont(t.Font)
  PARA: builder->ConvertParagraph
  }
}
```

TextConverter

ConvertCharacter()
ConvertFontChange
ConvertParagraph()

TexConverter

ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()

AsciiConverter

ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()

HTMLConverter

ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()

TeXText

AsciiText

HTMLText

Applicability for 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

- Composite Pattern:
  - **Models trees with dynamic width and dynamic depth**
- Facade Pattern:
  - **Interface to a Subsystem**
  - **Closed vs Open Architecture**
- Adapter Pattern:
  - **Interface to Reality (“Wrapper”)**
- Bridge Pattern:
  - **Interface Reality and Future**
- Proxy Pattern
  - **Control access to a remote object**
- Command Pattern
  - **Interface for executing operations**
- Observer Pattern
  - **Scalability**
  - **“Publish/Subscribe”**
- Abstract Factory Pattern
  - **Manufacturer independent Interface to a product family**
- Builder Pattern
  - **Create complex object without knowing its (changing) representation**

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**