TUM

Lecture Notes on Design Patterns II

Bernd Brügge

Technische Universität München

Lehrstuhl für Angewandte Softwaretechnik

24 January 2002

Outline

- Review of design pattern concepts
  - **What is a design pattern?**
  - **Modifiable designs**

More patterns:

  - **Abstract Factory**
  - **Proxy**
  - **Command**
  - **Observer**
  - **Strategy**

Review: 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 classes or datastructures (link lists,binary trees...)**
- **Lower level than application frameworks**

…an example of *modifiable* 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**

On to More Patterns!

- Structural pattern
  - **Proxy (207)**

- Creational Patterns
  - **Abstract Factory (87)**
  - **Builder (97)**

- Behavioral pattern
  - **Command (233)**
  - **Observer (293)**
  - **Strategy (315)**

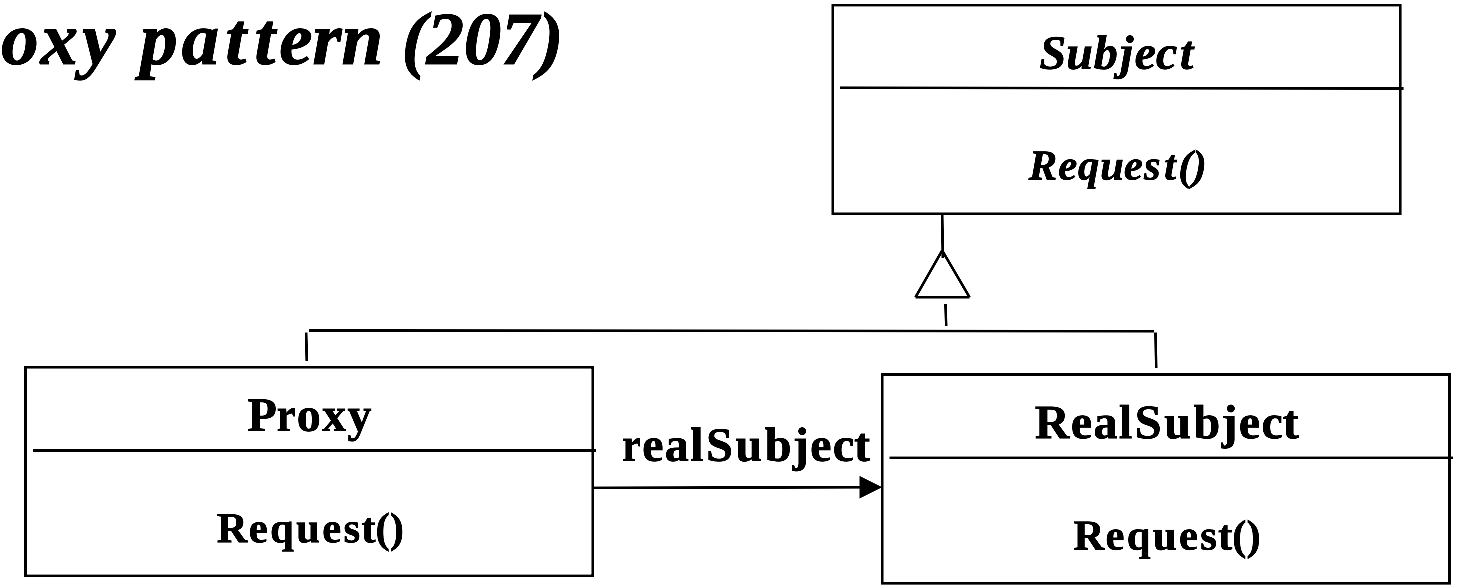
Proxy Pattern: Motivation

- 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 you do?

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 (207)

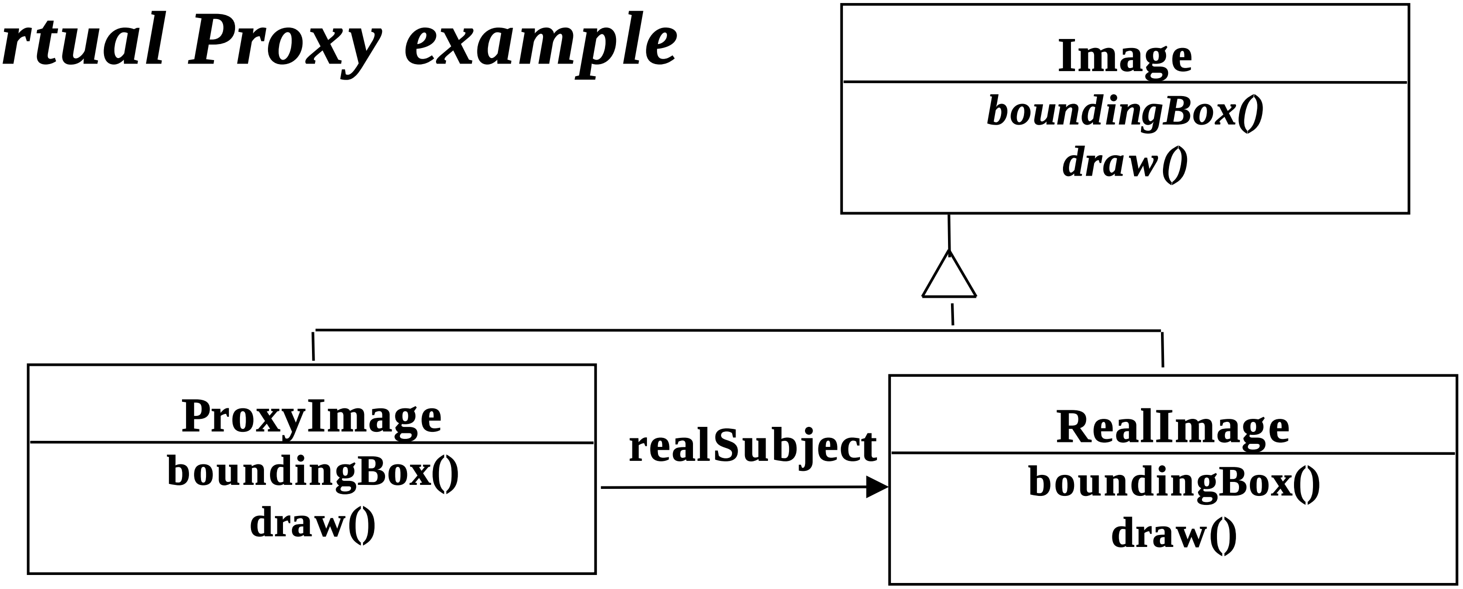


- Interface inheritance is used to specify the interface shared by **Proxy** and **RealSubject.**
- Delegation is used to catch and forward any accesses to the **RealSubject** (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented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

Proxy Applicability

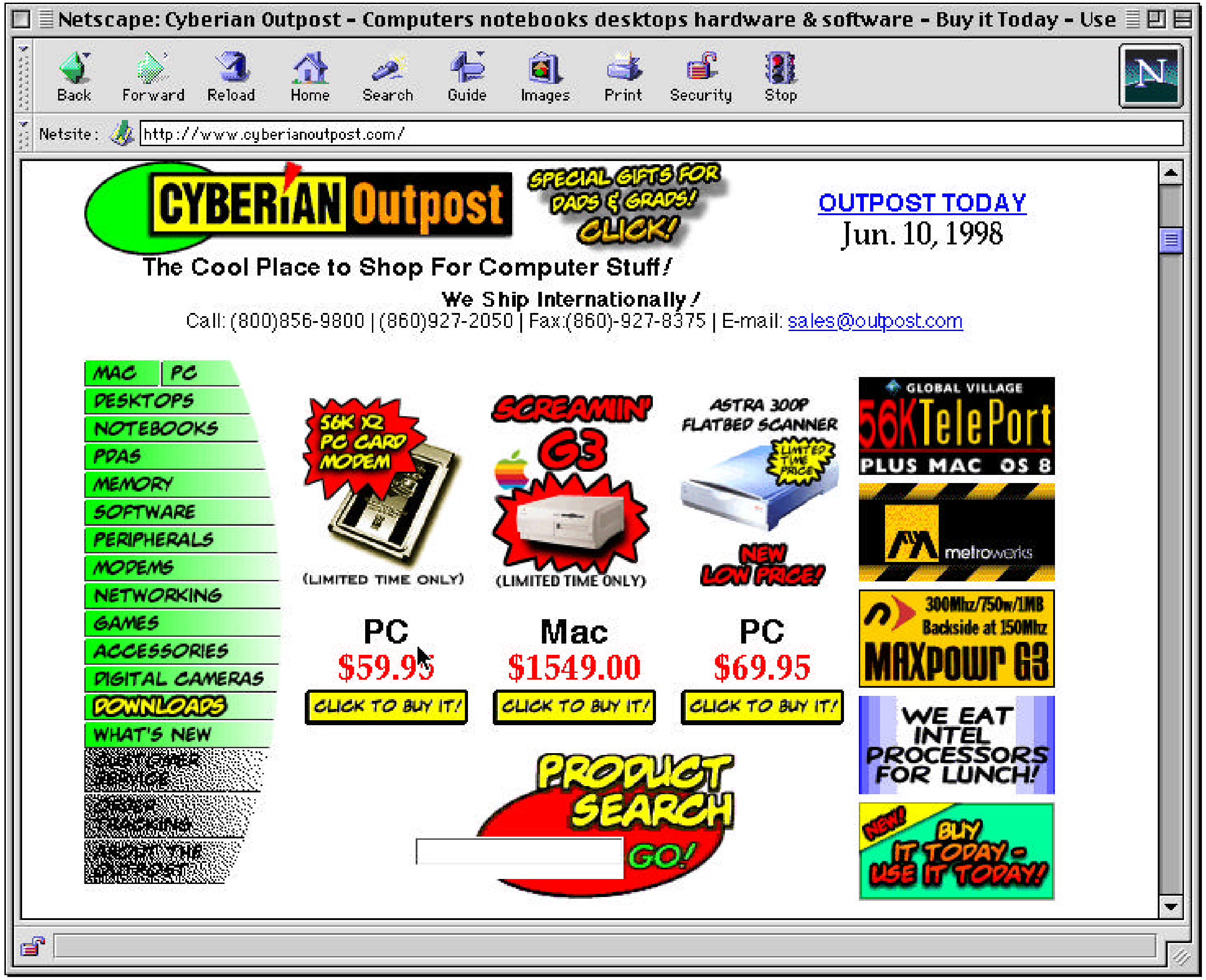
- Remote Proxy
  - **Local representative for an object in a different address space**
  - **Caching of information: Good if information does not change too often**
- Virtual Proxy
  - **Object is too expensive to create or too expensive to download**
  - **Proxy is a standin**
- Protection Proxy
  - **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



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

Netsite: http://www.cyberianoutpost.com/

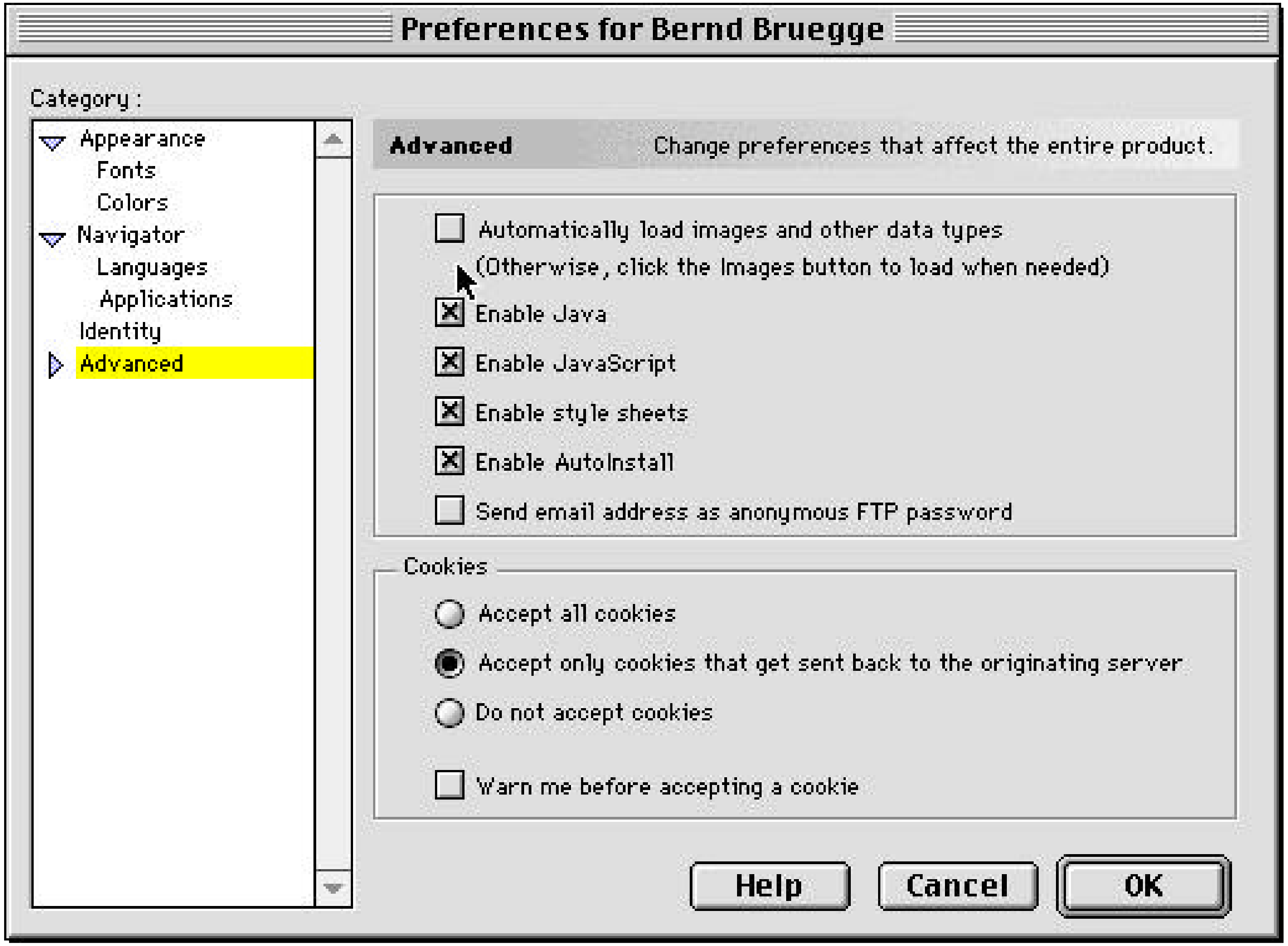
The Cool Place to Shop For Computer Stuff!

We Ship Internationally!

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

OUTPOST TODAY
Jun. 10, 1998

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

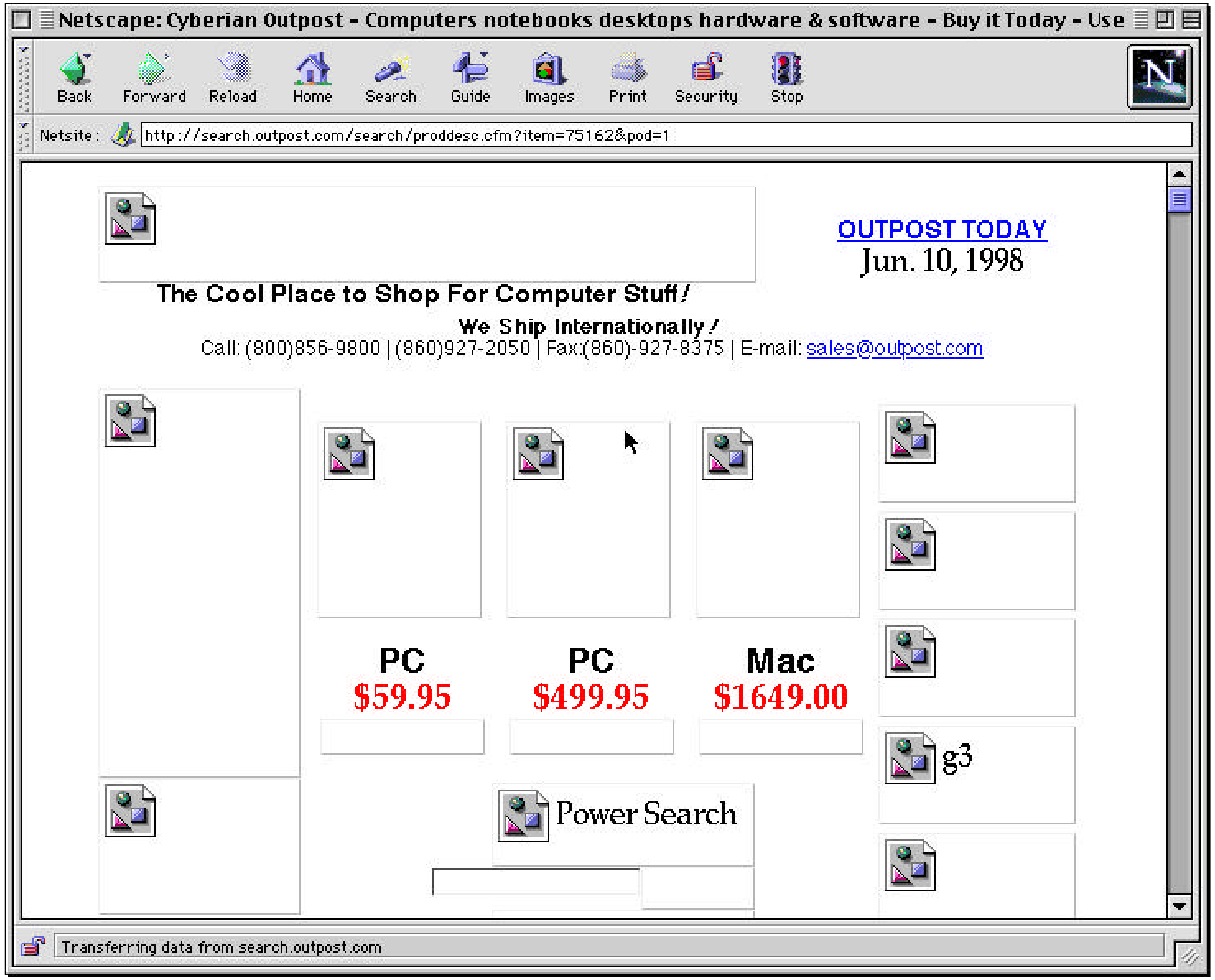
○ Accept all cookies

● 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=7516&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

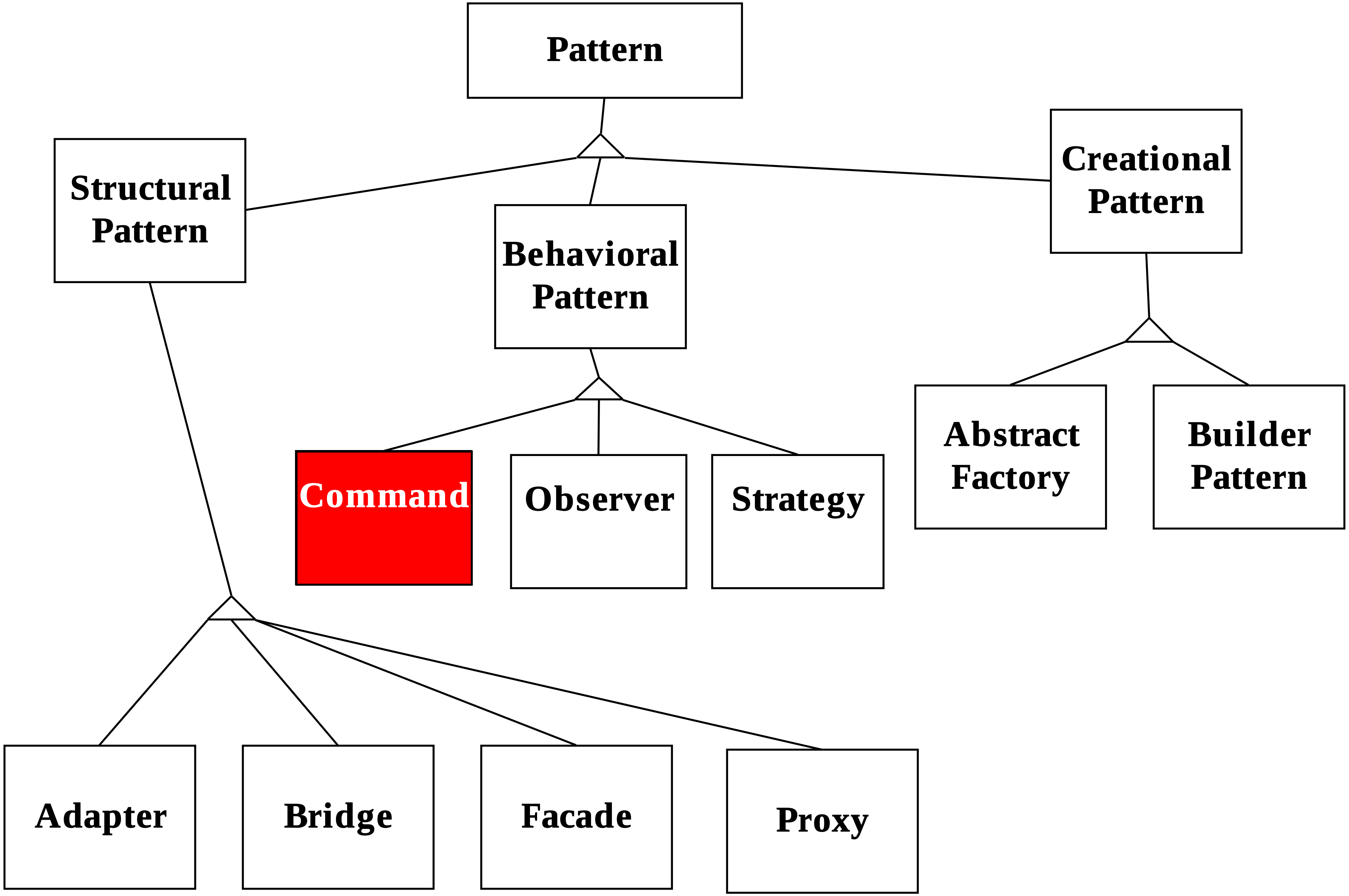
Mac $1649.00

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.**

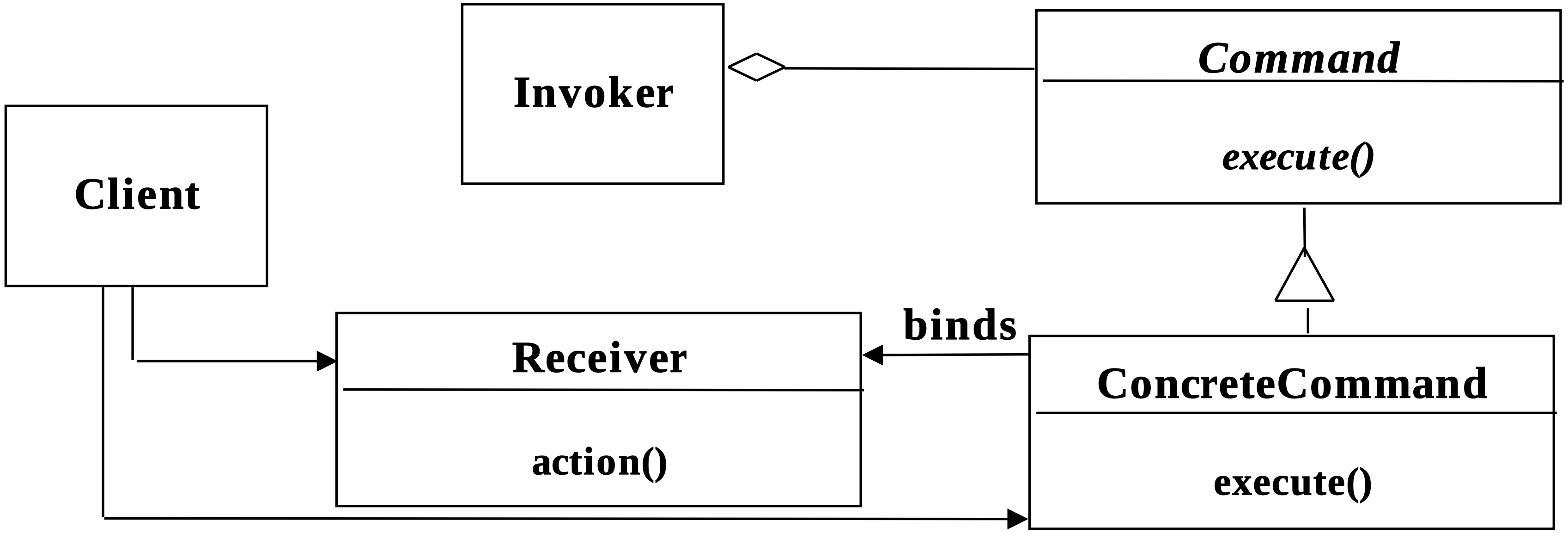
A Pattern Taxonomy



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)



- **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”)**.**

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**

Observer pattern (293)

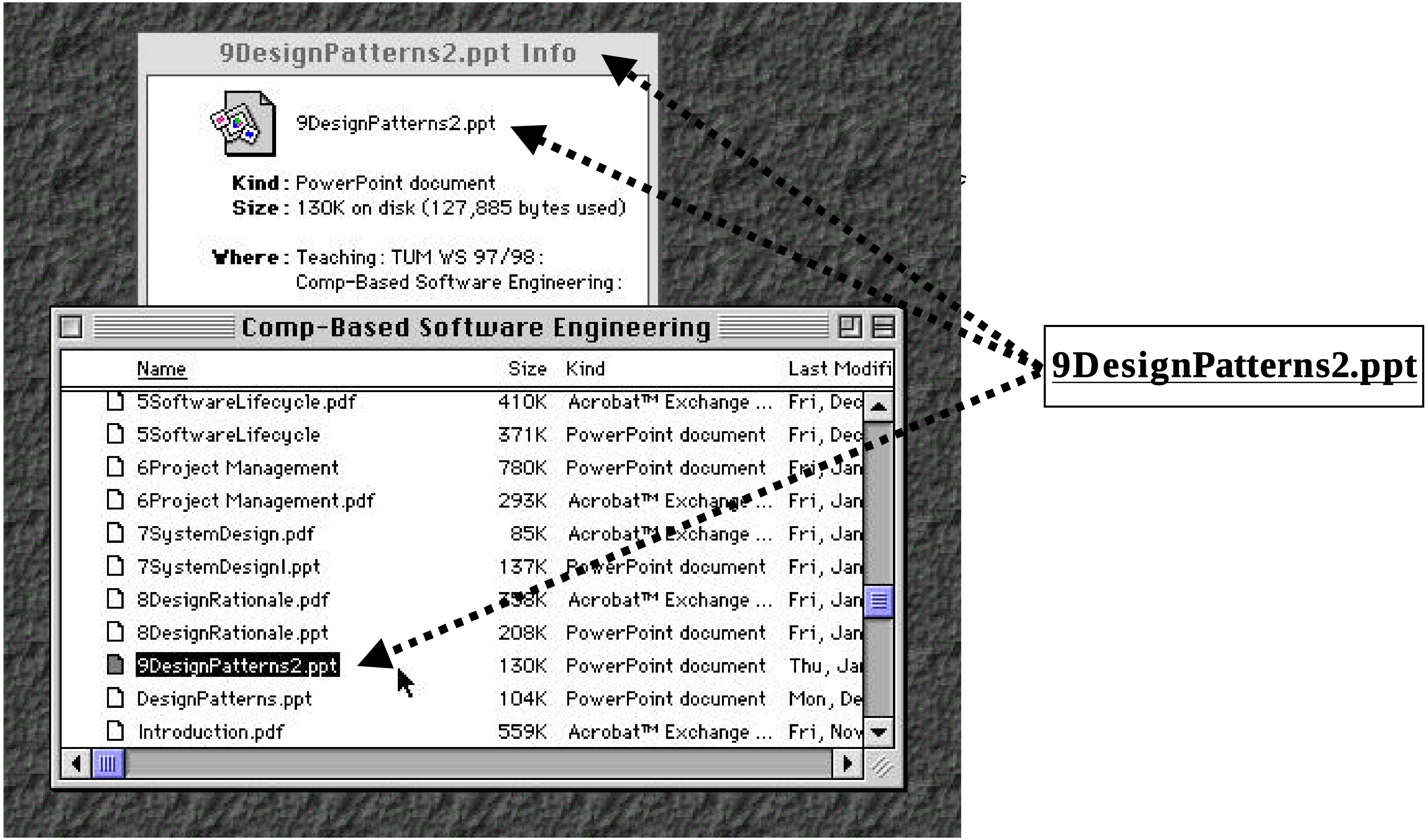
- “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” (p. 293)
- Also called “Publish and Subscribe”

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

Observer pattern (continued)

Observers

Subject



9DesignPatterns2.ppt Info

9DesignPatterns2.ppt

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

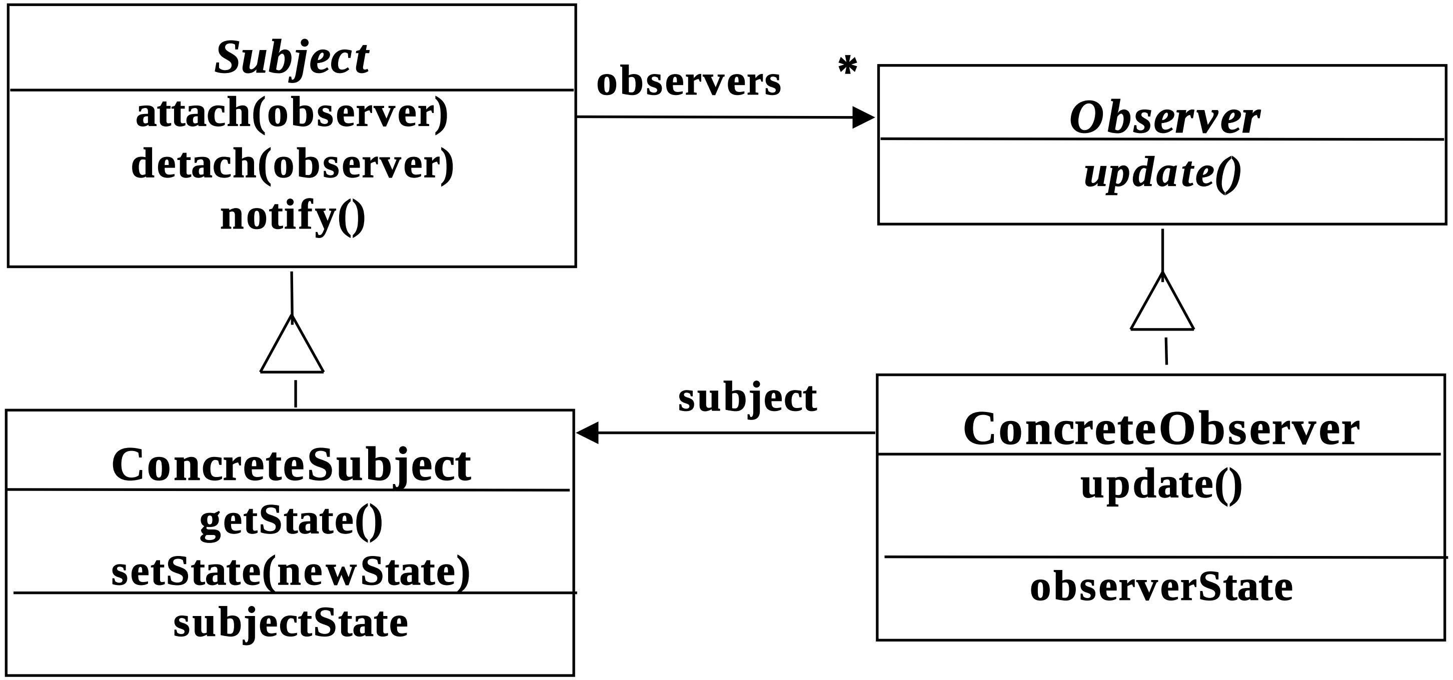
Where : Teaching : TUM WS 97/98 :
Comp-Based Software Engineering :

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	758K	Acrobat™ Exchange ...	Fri, Jan
8DesignRationale.ppt	208K	PowerPoint document	Fri, Jan
9DesignPatterns2.ppt	130K	PowerPoint document	Thu, Jan
DesignPatterns.ppt	104K	PowerPoint document	Mon, De
Introduction.pdf	559K	Acrobat™ Exchange ...	Fri, Nov

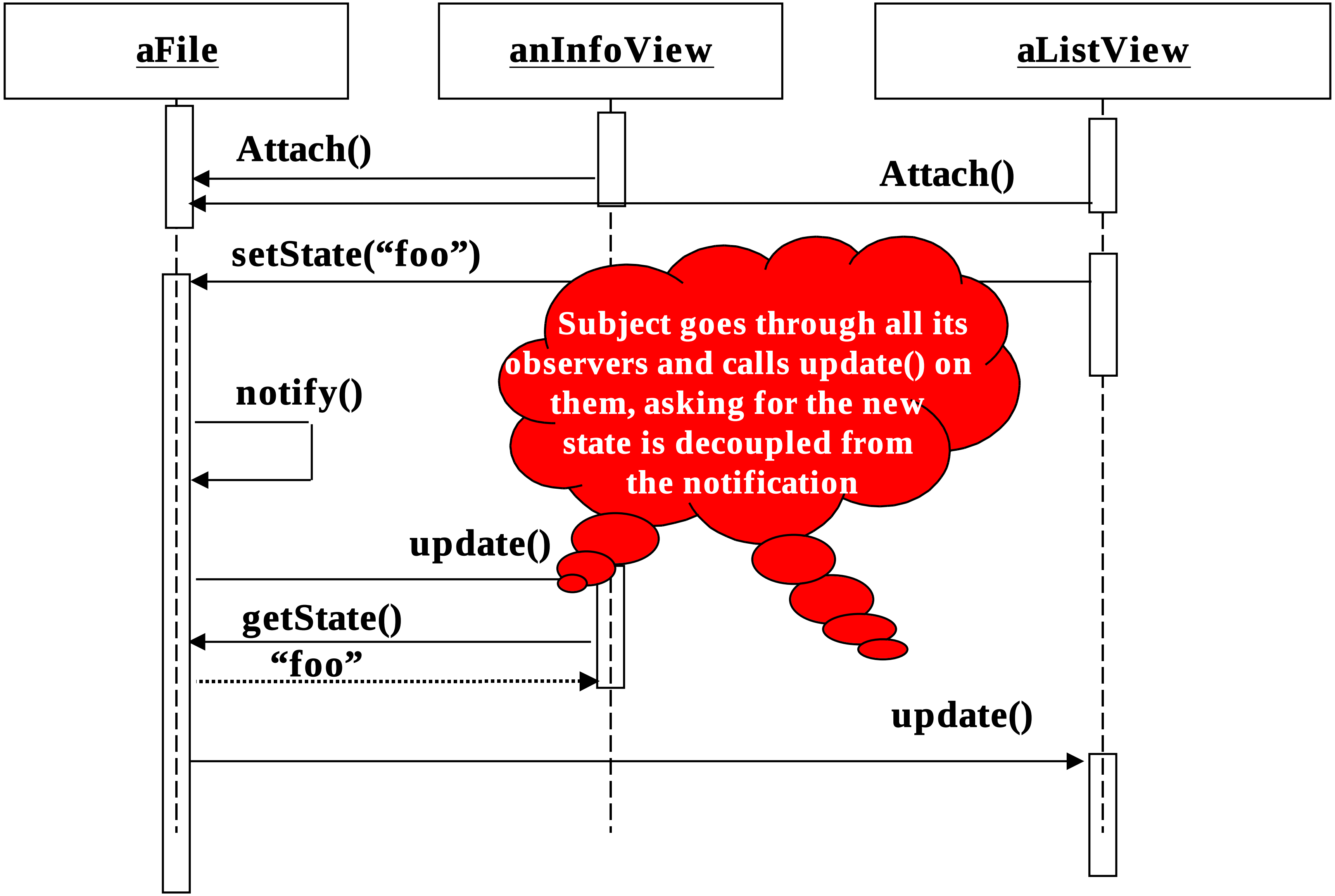
9DesignPatterns2.ppt

Observer pattern (cont'd)



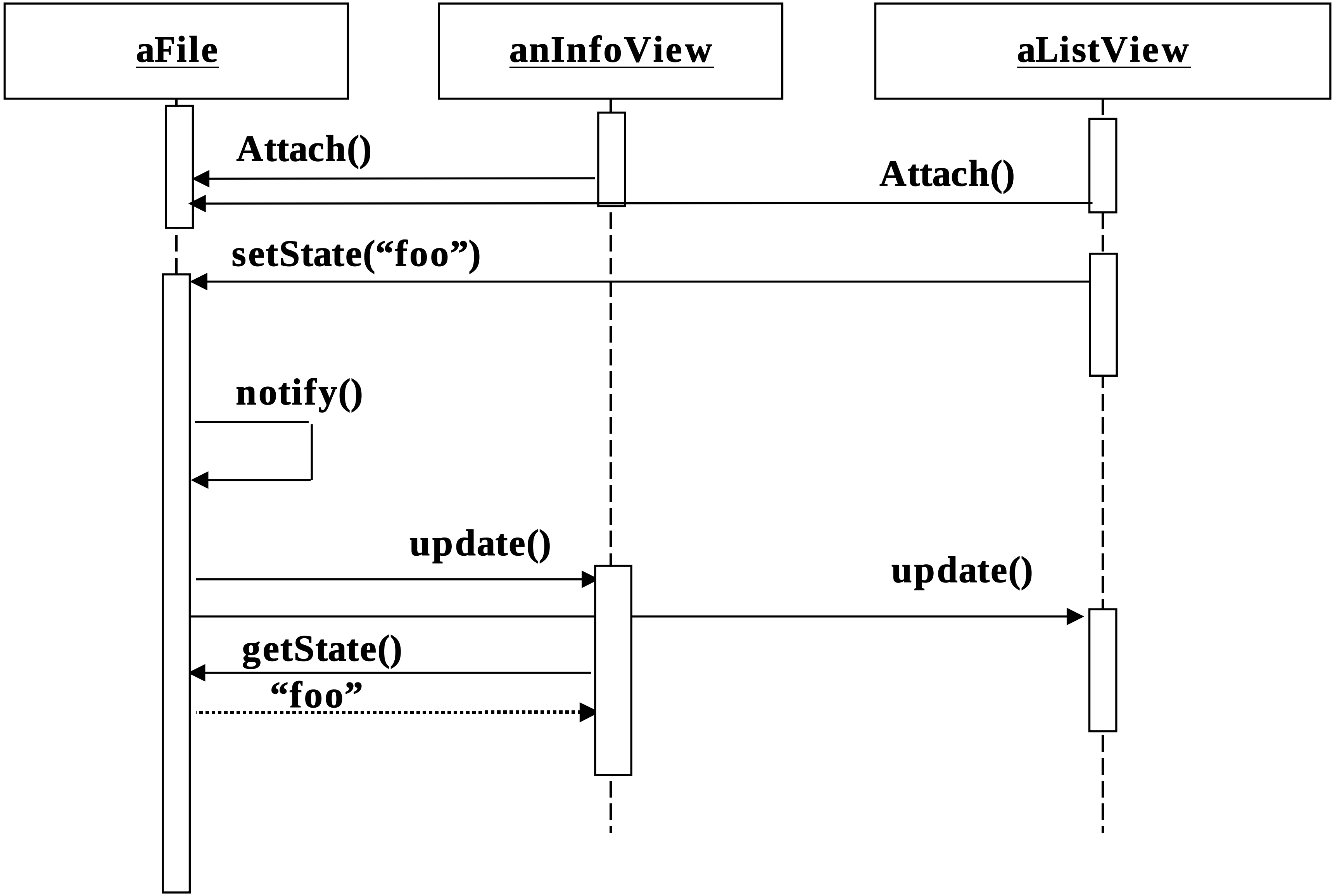
- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) ***not*** an interface.

Sequence diagram for scenario: Change filename to “foo”



Subject goes through all its observers and calls update() on them, asking for the new state is decoupled from the notification

Animated Sequence diagram



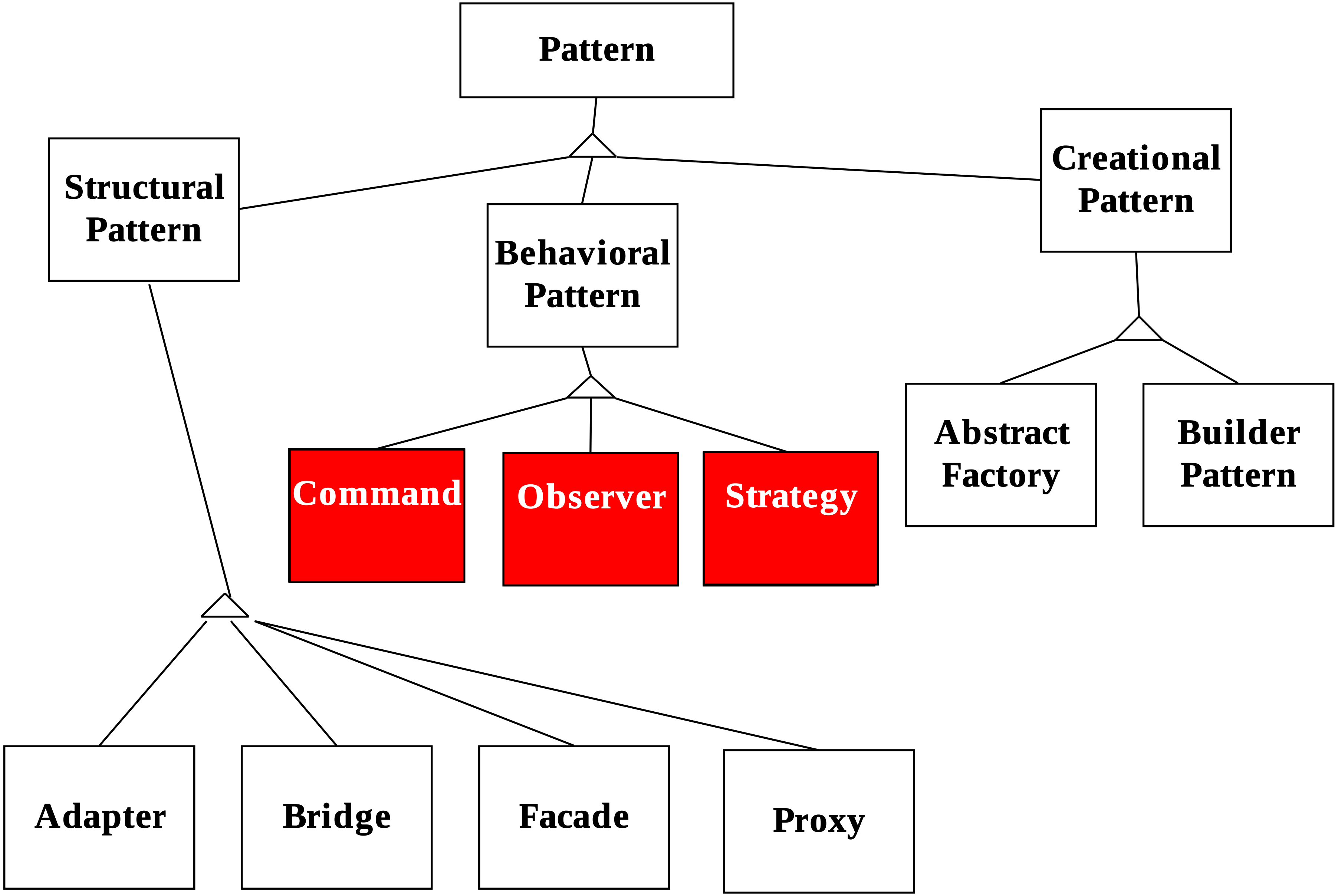
Observer pattern implementation in Java

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

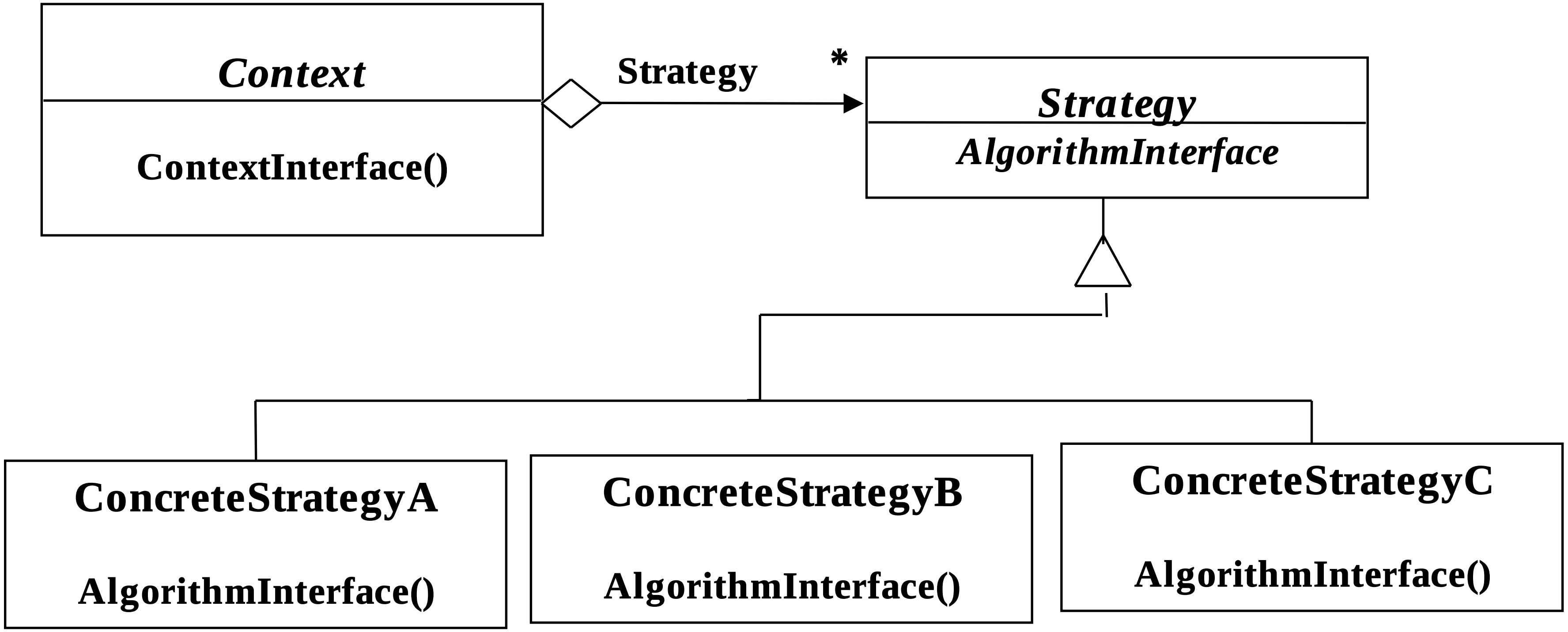
A Pattern Taxonomy



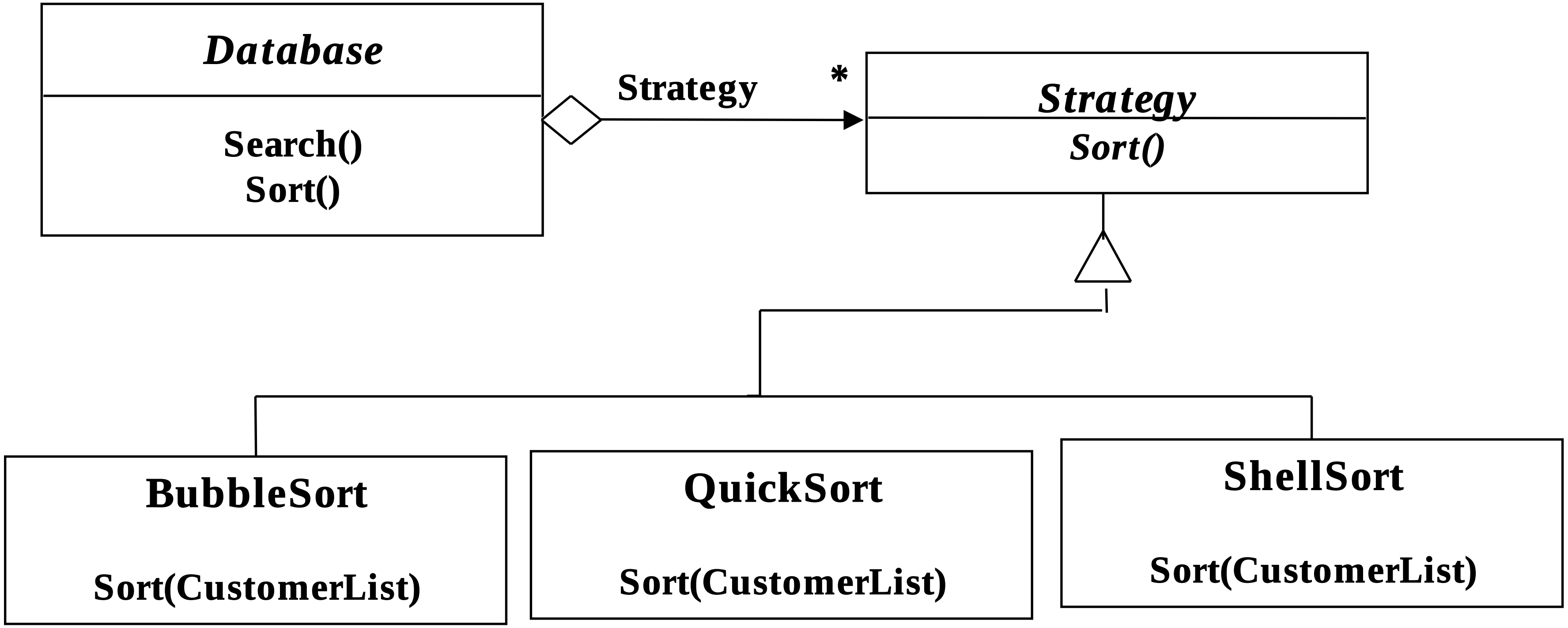
Strategy Pattern

- 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)



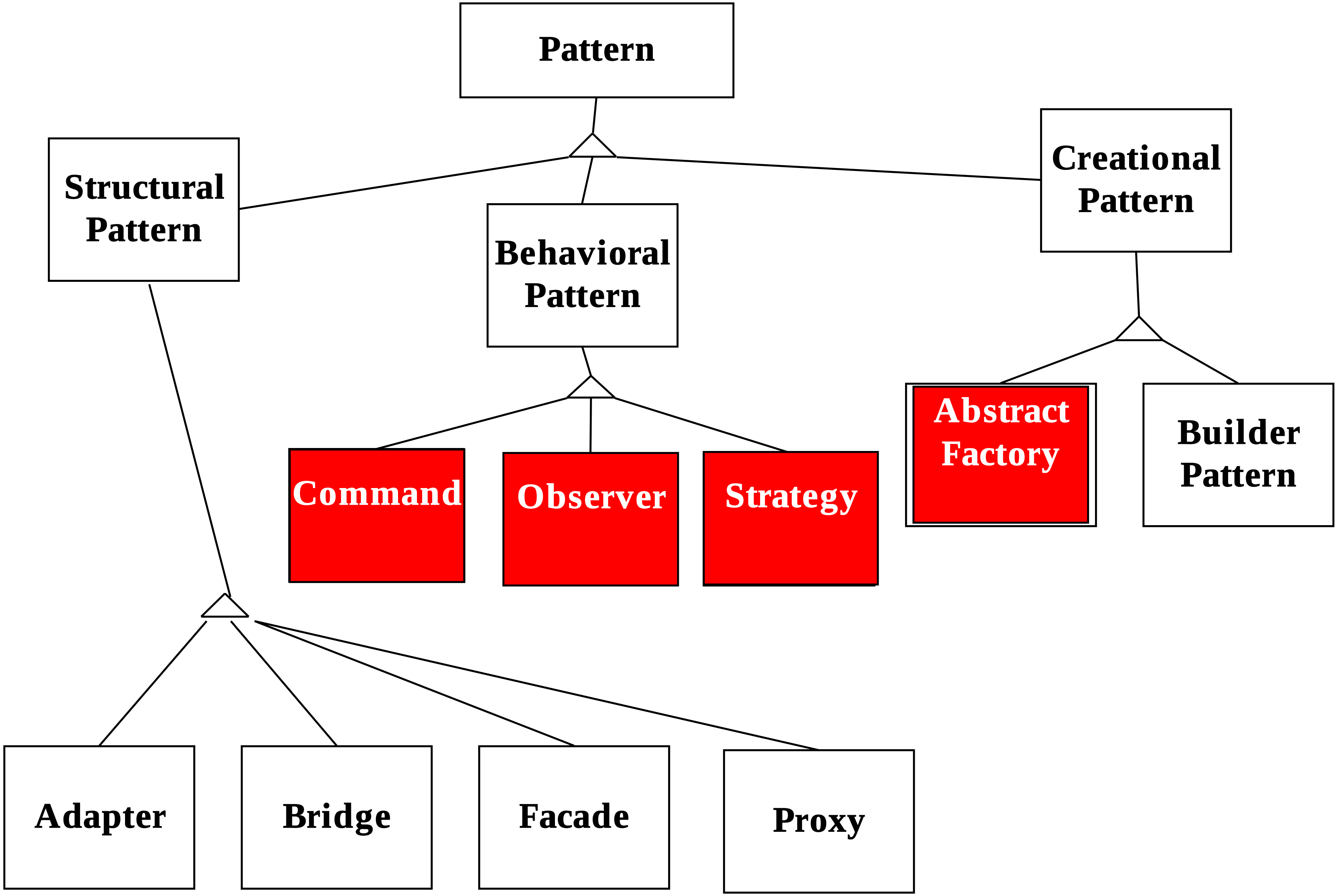
Applying a Strategy Pattern in a Database Application



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

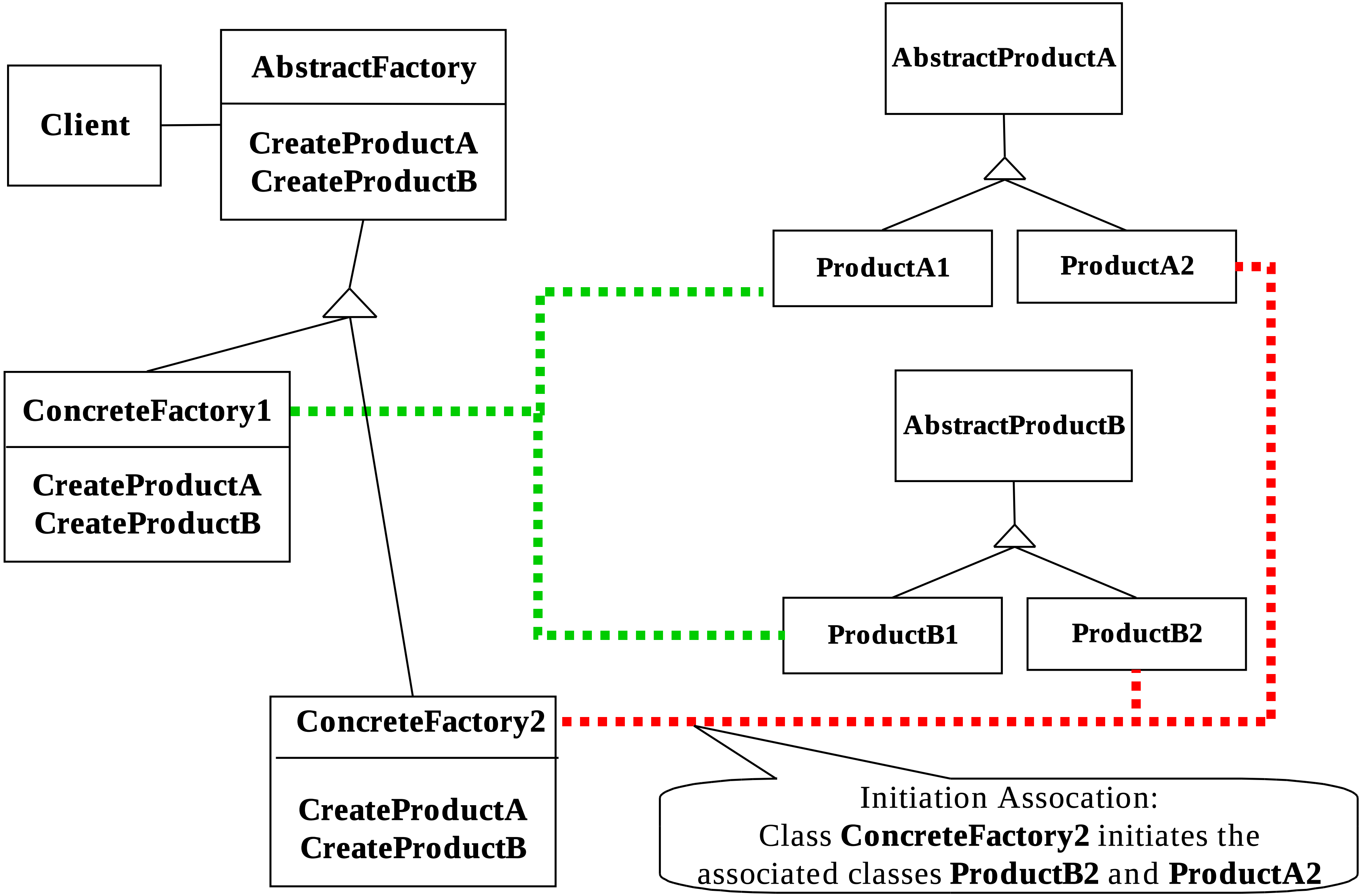
A Pattern Taxonomy



Abstract Factory Motivation

- 2 Examples
- 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 (87)

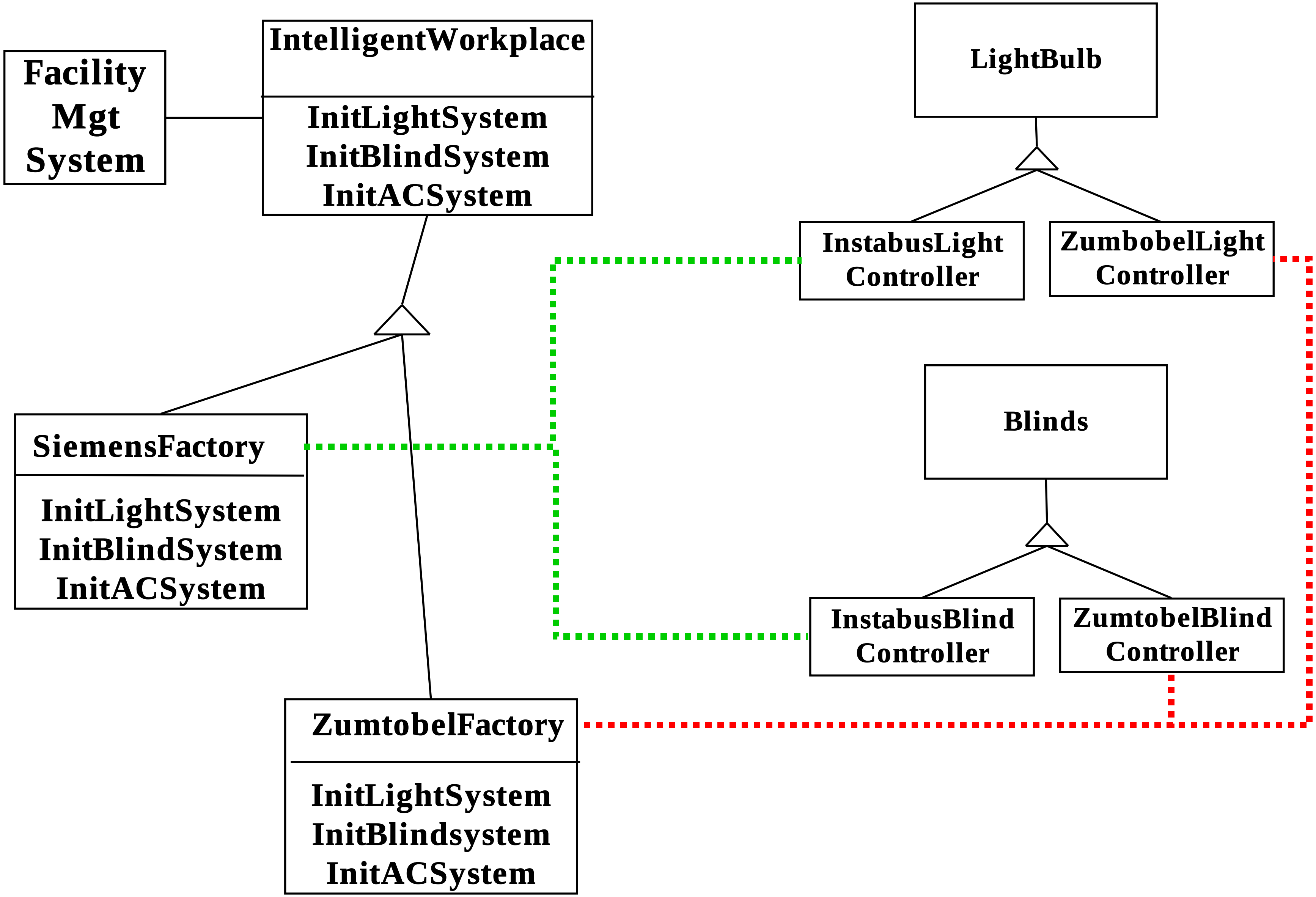


Initiation Assocation:
Class **ConcreteFactory2** initiates the associated classes **ProductB2** and **ProductA2**

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 family of products, where one has a choice from many different families.**
  - **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.**

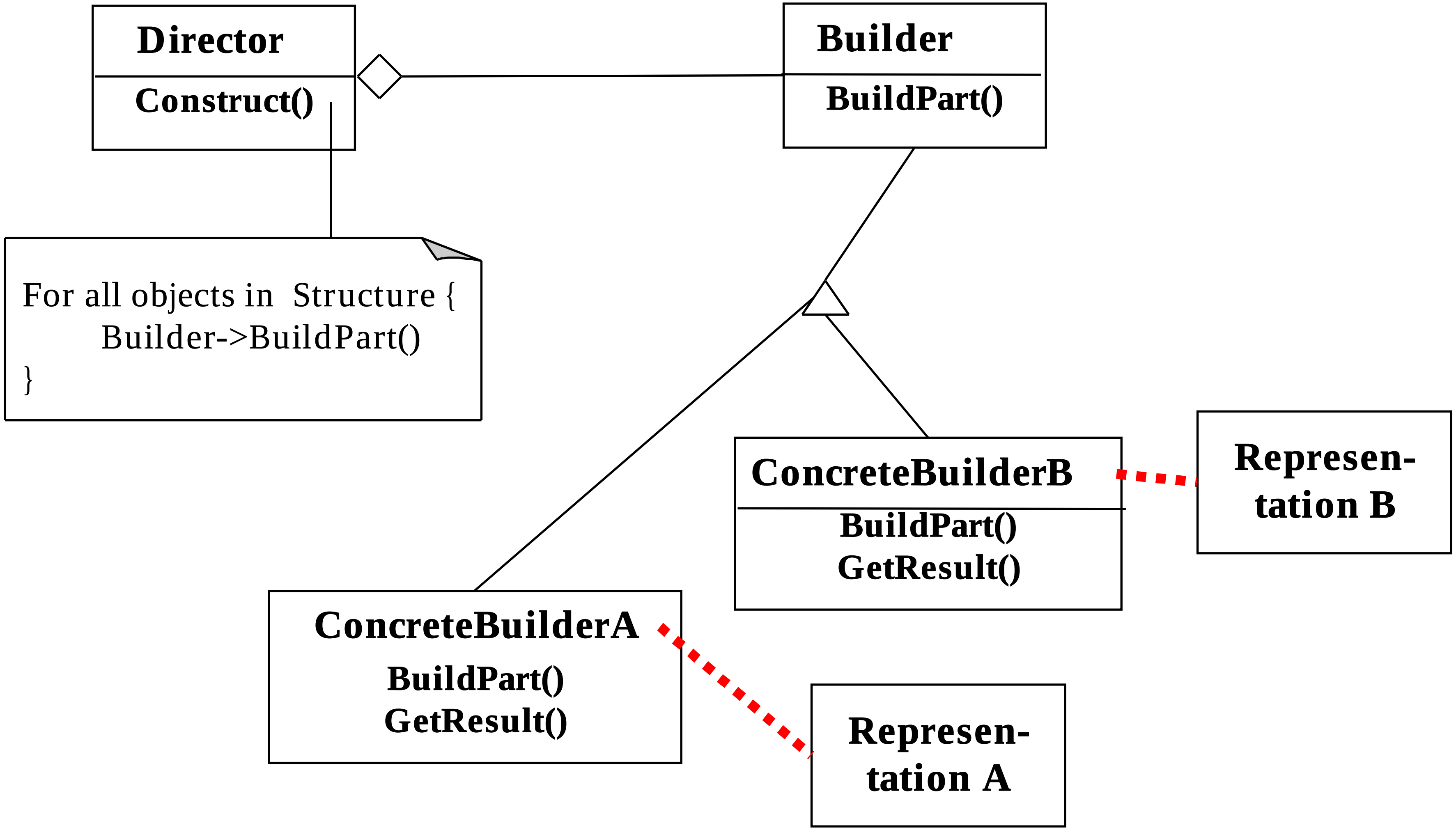
Example: A Facility Management System



Builder Pattern Motivation

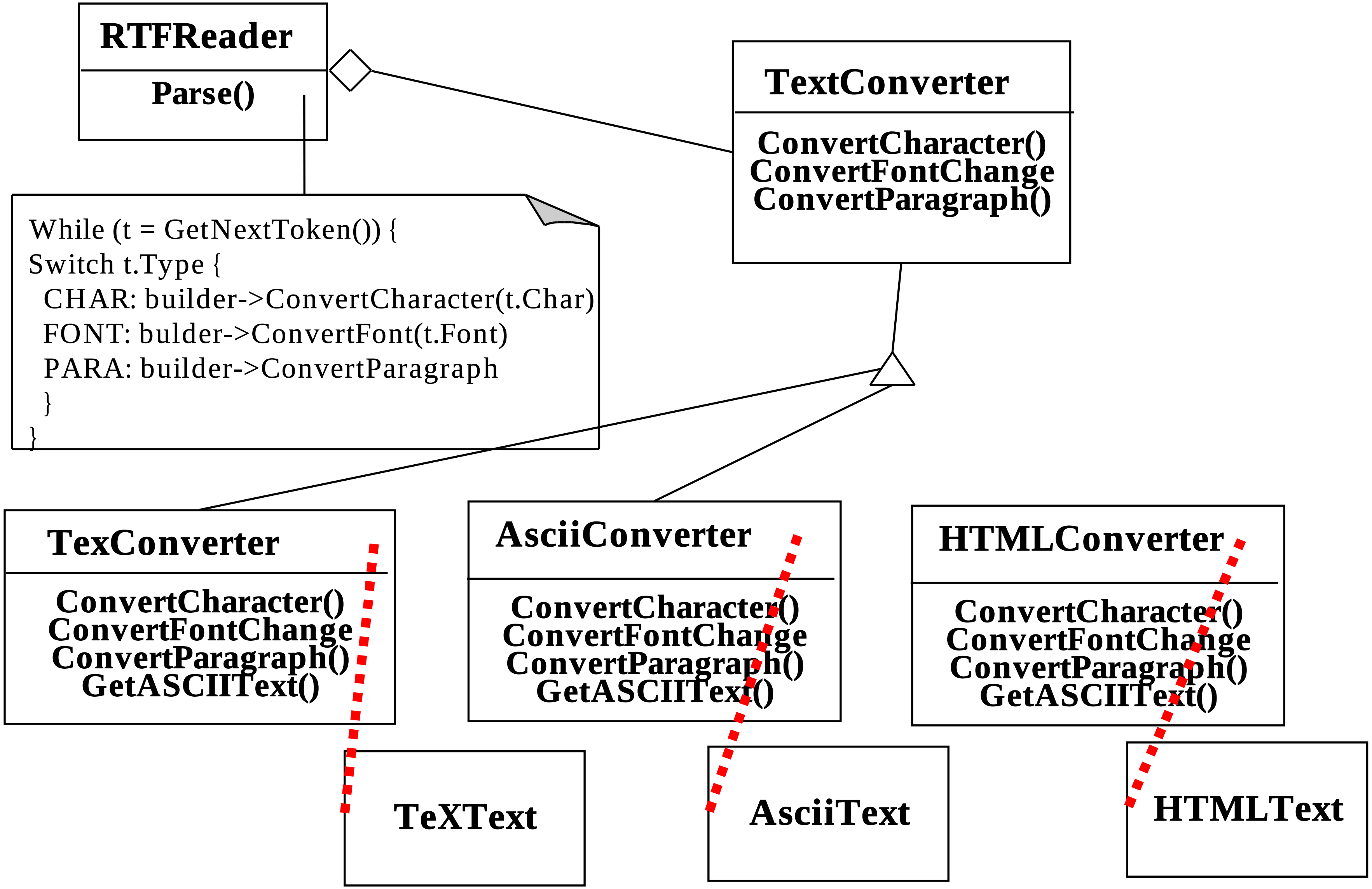
- Conversion of documents
- Software companies make their money by introducing new formats, forcing users to upgrades
  - **But you don’t want to upgrade your software every time there is an update of the format for Word documents**
- Idea: A reader for RTF format
  - **Convert RTF to many text formats (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)



```
For all objects in  Structure {
        Builder->BuildPart()
}
```

Example



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

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.**

Comparison: Abstract Factory vs Builder

- Abstract Factory
  - **Focuses on product family**
    - **The products can be simple (“light bulb”) or complex (“engine”)**
  - **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 the creation is very complex**
  - **The construction of the complex product changes from time to time**
  - **The builder patterns hides the 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 ☺
- My favorites: Composite, Bridge, Builder and Observer