Course "Softwaretechnik"
Book Chapter 8
Object Design: Reuse and Patterns II

Lutz Prechelt, Bernd Bruegge, Allen H. Dutoit

Freie Universität Berlin, Institut für Informatik

- Proxy
- Command
- Observer
- Strategy
- Abstract Factory
- Builder

Wo sind wir?: Taxonomie "Die Welt der Softwaretechnik"

Freie Universität Berlin

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
  - Anforderungen (Problemraum)
  - **Entwurf (Lösungsraum)**
- Prozess (psycho-soziale P.)
  - Kognitive Beschränkungen
  - Mängel der Urteilskraft
  - Kommunikation, Koordination
  - Gruppendynamik
  - Verborgene Ziele
  - Fehler

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
  - **Abstraktion**
  - **Wiederverwendung**
  - Automatisierung
- Methodische Ansätze ("weich")
  - Anforderungsermittlung
  - **Entwurf**
  - Qualitätssicherung
  - Projektmanagement

Wo sind wir?: Entwurf

- Einsicht: Man sollte *vor* dem Kodieren über eine günstige Struktur der Software nachdenken
  - und diese als Koordinationsgrundlage schriftlich festhalten

- Prinzipien:
  - **Trennung von Belangen**
  - **Architektur**: Globale Struktur festlegen (Grobentwurf), insbes. für das Erreichen der nichtfunktionalen Anforderungen
  - **Modularisierung**: Trennung von Belangen durch Modularisierung, Kombination der Teile durch Schnittstellen (information hiding, Lokalität)
  - ***Wiederverwendung***: Erfinde Architekturen und Entwurfsmuster nicht immer wieder neu
  - **Dokumentation**: Halte sowohl Schnittstellen als auch zu Grunde liegende Entwurfsentscheidungen und deren Begründungen fest

Outline of the lecture

Freie Universität Berlin

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

- More patterns:
  - Abstract Factory: Provide manufacturer independence
  - Builder: Hide a complex creation process
  - Proxy: Provide transparency
  - Command: Encapsulate control flow
  - Observer: Provide publisher/subscribe mechanism
  - Strategy: Support family of algorithms, separate policy from mechanism

Review: design pattern

A design pattern is...

- ...a template solution to a recurring design problem
  - Consider them before re-inventing the wheel
- ...reusable design knowledge
  - Higher level than classes or common data structures
  - Lower level than application frameworks
- ...an example of good design
  - Learning to design starts by studying other designs
- ...generalized from existing systems
  - i.e., realistic (rather than armchair philosophy)
- ...powerful shared vocabulary for designers

The patterns we consider here focus on modifiable designs

Modifiable designs

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?

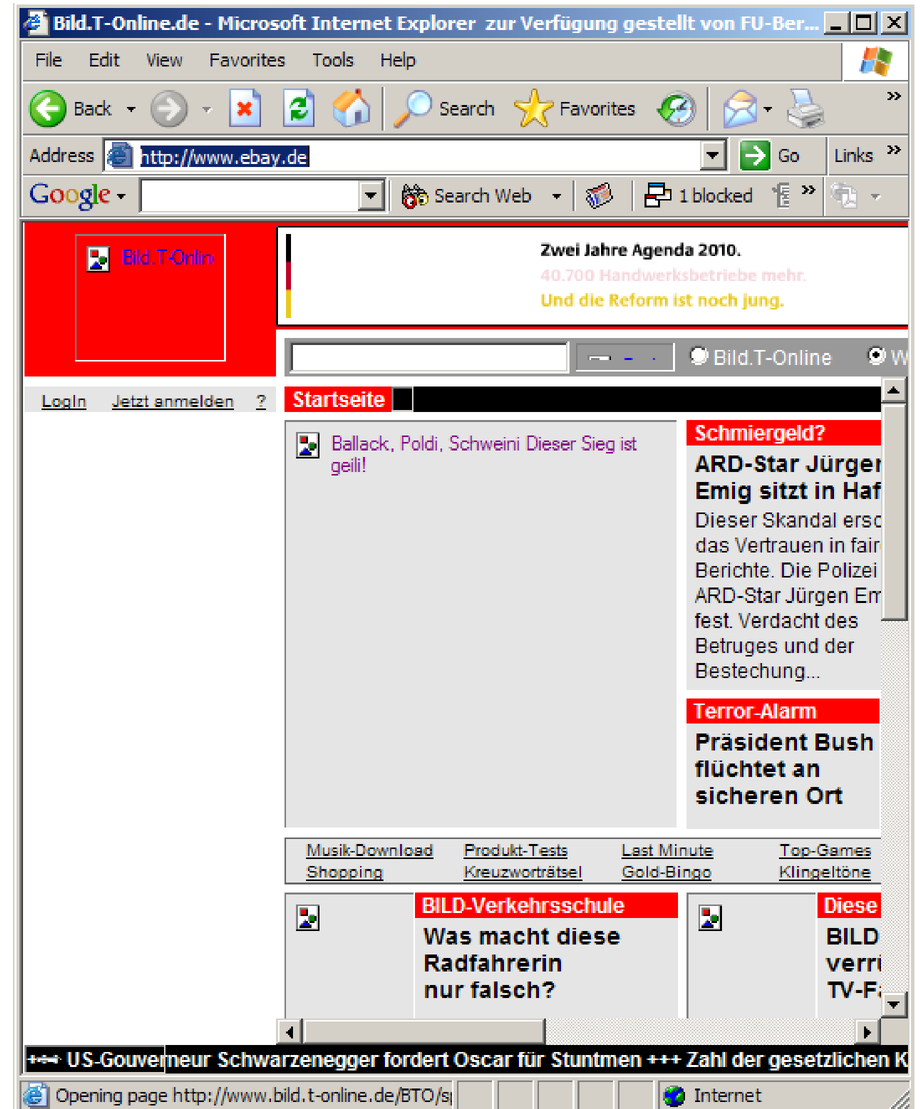
- Encapsulated design decisions
  - Clear dependencies
  - Explicit assumptions
- Reduced coupling

On to more patterns!

- Structural Pattern (Strukturmuster)
  - Proxy — Stellvertreter
- Creational Patterns (Erzeugungsmuster)
  - Abstract Factory — Abstrakte Fabrik
  - Builder — Erbauer
- Behavioral Patterns (Verhaltensmuster)
  - Command — Kommando
  - Observer — Beobachter
  - Strategy — Strategie

Proxy Pattern: motivation

- I want to access an image-intensive webpage
  - but am connected only via a slow mobile phone connection: 56000 bit/s
- Can my browser help?
- Solution idea:
  - In place of each image, display only a placeholder at first
  - Only after clicking on an image, this image will be downloaded



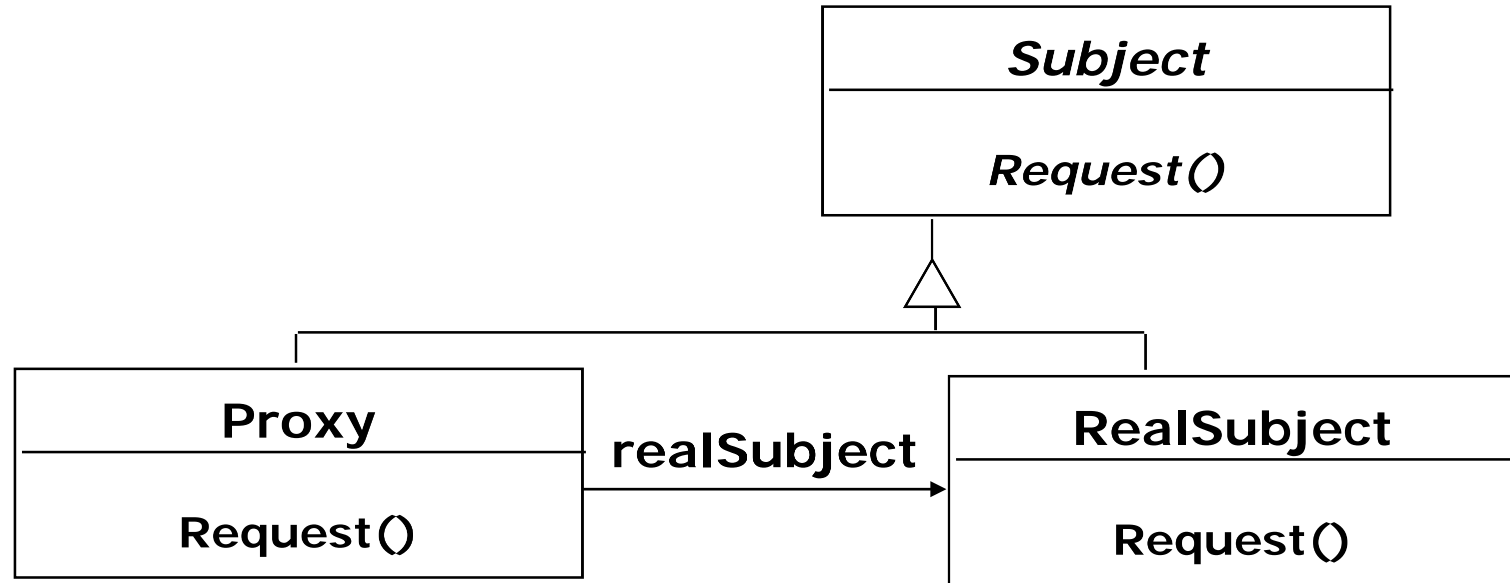
Proxy Pattern (Stellvertreter)

Also known as *Surrogate*

- Problem:
  I need access to a certain object but I cannot (or do not want to) access it directly
- Solution idea:
  Provide a replacement object (with the same interface as the original object) that performs the access for me
- http://c2.com/cgi/wiki?ProxyPattern

Proxy Pattern

Freie Universität Berlin



- 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 and when desired)
  - Client accesses Proxy only
- Proxy patterns can be implemented with a Java interface

Proxy applications

Freie Universität Berlin

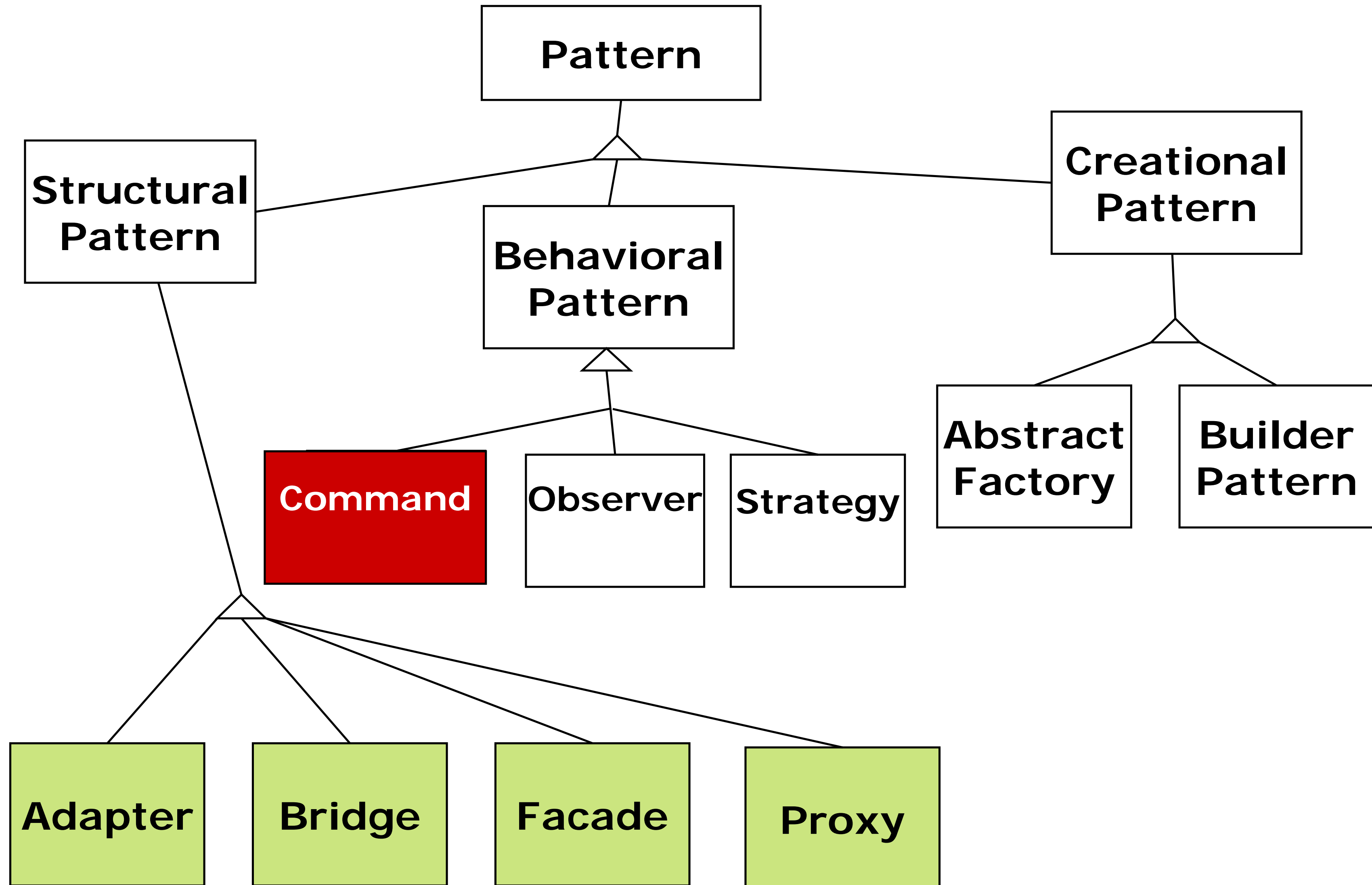
- Remote Proxy: Local representative for a remote object
  - Provides location transparency
    - Client need not know where the object lives
  - Provides access transparency
    - Client need not know the access mechanisms
  - May provide caching of information
- Decorator: Invisibly add some functionality
  - e.g. add a scrollbar to a text pane so it can fit in less space
- Virtual Proxy: Stand-in object
  - When creating the object is expensive (→ browser images above)
- Protection Proxy: Access control object
  - Proxy object encapsulates the access rules
  - Different proxies can be used for different clients

Towards a pattern taxonomy

Freie Universität Berlin

- Structural Patterns ("Strukturmuster")
  - Composite, Adapter, Bridge, Facade, and Proxy are variations on a single theme:
    - They reduce the coupling between two or more classes
    - They introduce abstract classes to enable future extensions
    - They encapsulate complex structures
- Behavioral Patterns ("Verhaltensmuster")
  - Here we are concerned with algorithms and the assignment of responsibilies between objects: Who does what?
  - Behavioral patterns allow us to characterize complex control flows that are difficult to follow at runtime
- Creational Patterns ("Erzeugungsmuster")
  - Here we our goal is to provide an abstraction for a (possibly complex) instantiation process
  - We want to make the system independent from the way its objects are created, composed, and represented

A pattern taxonomy



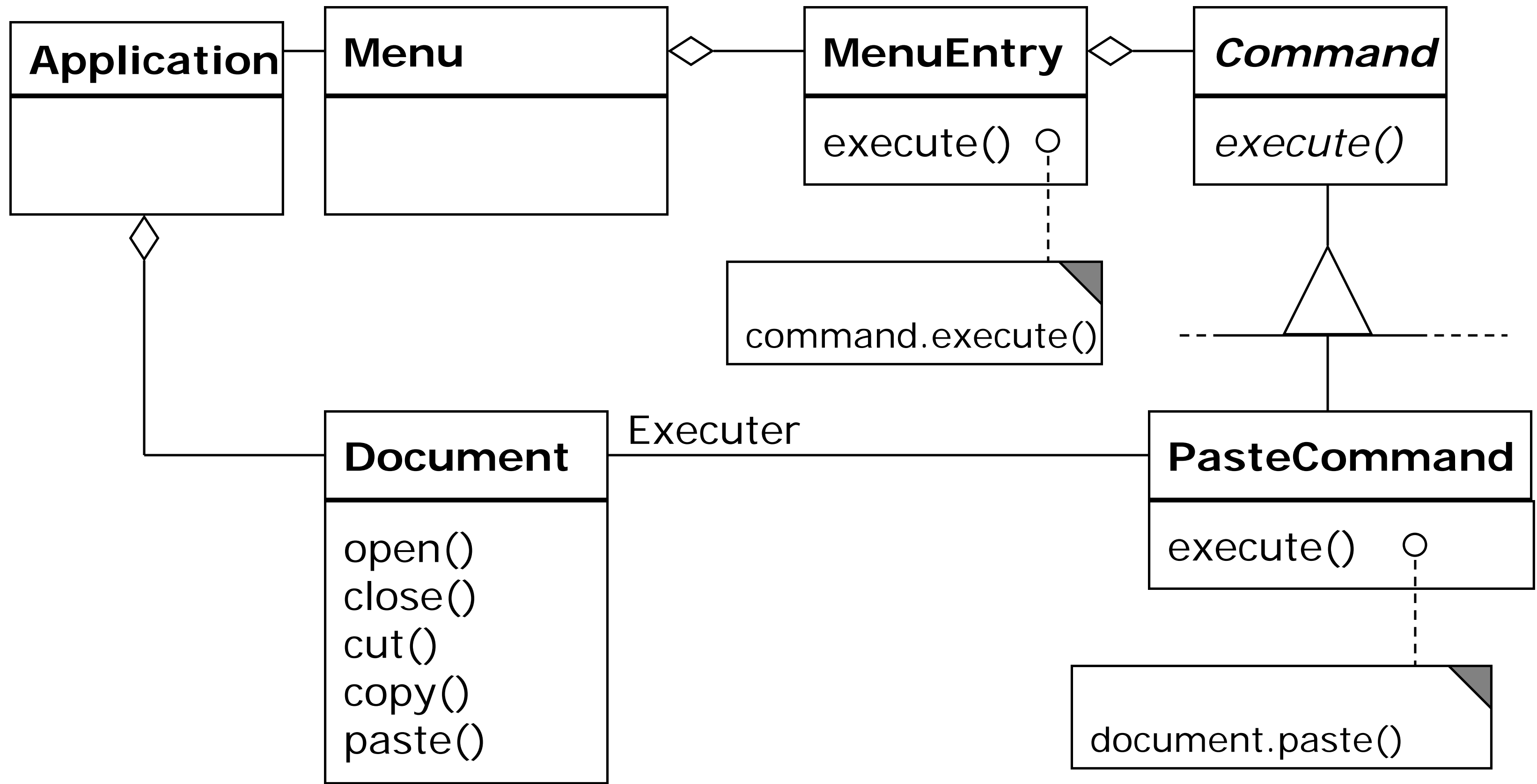
Command Pattern: motivation

- You want to build a user interface
  - including menus

- You want to make the user interface reusable across many applications and reconfigurable within each
  - You cannot hard-code the meanings of the menus for the various applications
    - A Menu object should be just a container for MenuEntry objects
  - So the operation called by the application when a menu entry is selected must be the same for any MenuEntry

- Such a menu can be implemented with the Command Pattern
  - MenuEntries are (or contain) Command objects

Command example: menu entries



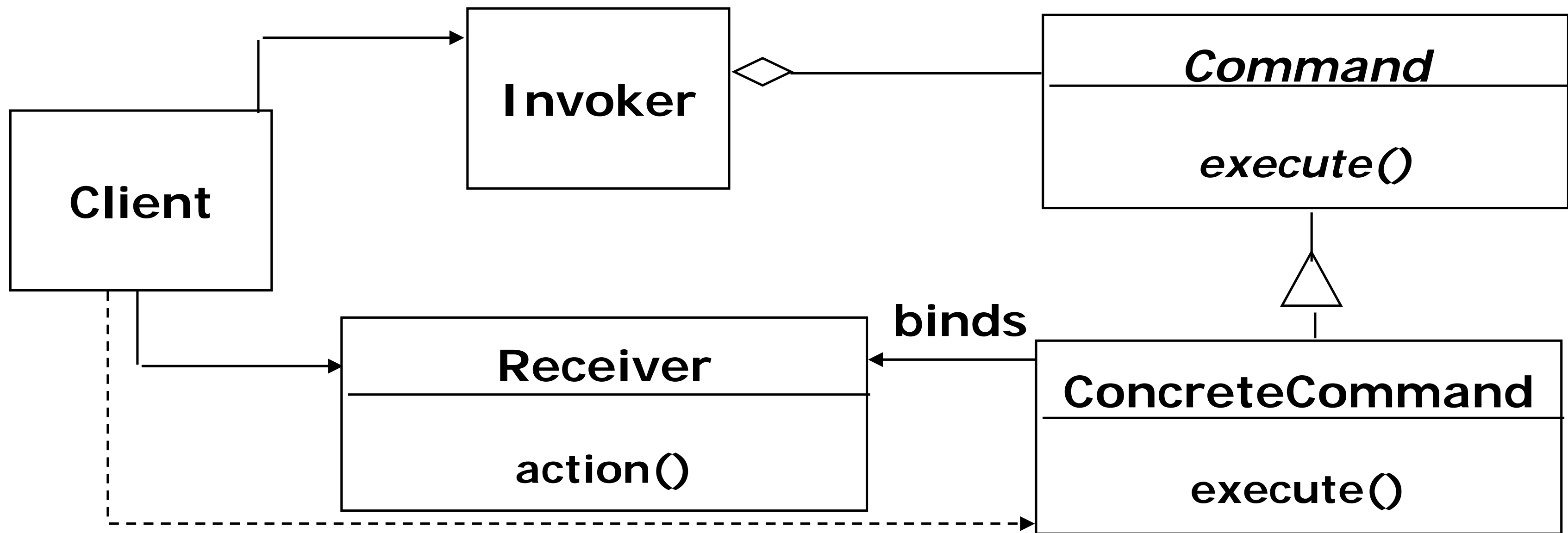
Command Pattern (Kommando)

Also known as *Action* or *Transaction*

- Problem: We need to handle actions just like data
  - move them around, store them, copy them, pass them as parameters, etc.

- Solution idea: Package different actions into objects with a fixed interface

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 perform the command ("execute" or "undo")

Command Pattern applicability

Freie Universität Berlin

- Encapsulate a request as an object, thereby letting you...
  - parameterize clients with different requests,
  - queue requests,
  - log requests,
  - support undoable operations.

- Can be used for:
  - Generic Undo mechanisms
    - Upon 'execute', the command object stores all information required for the 'undo' operation, e.g. the data deleted by the 'execute'
  - Database transaction buffering

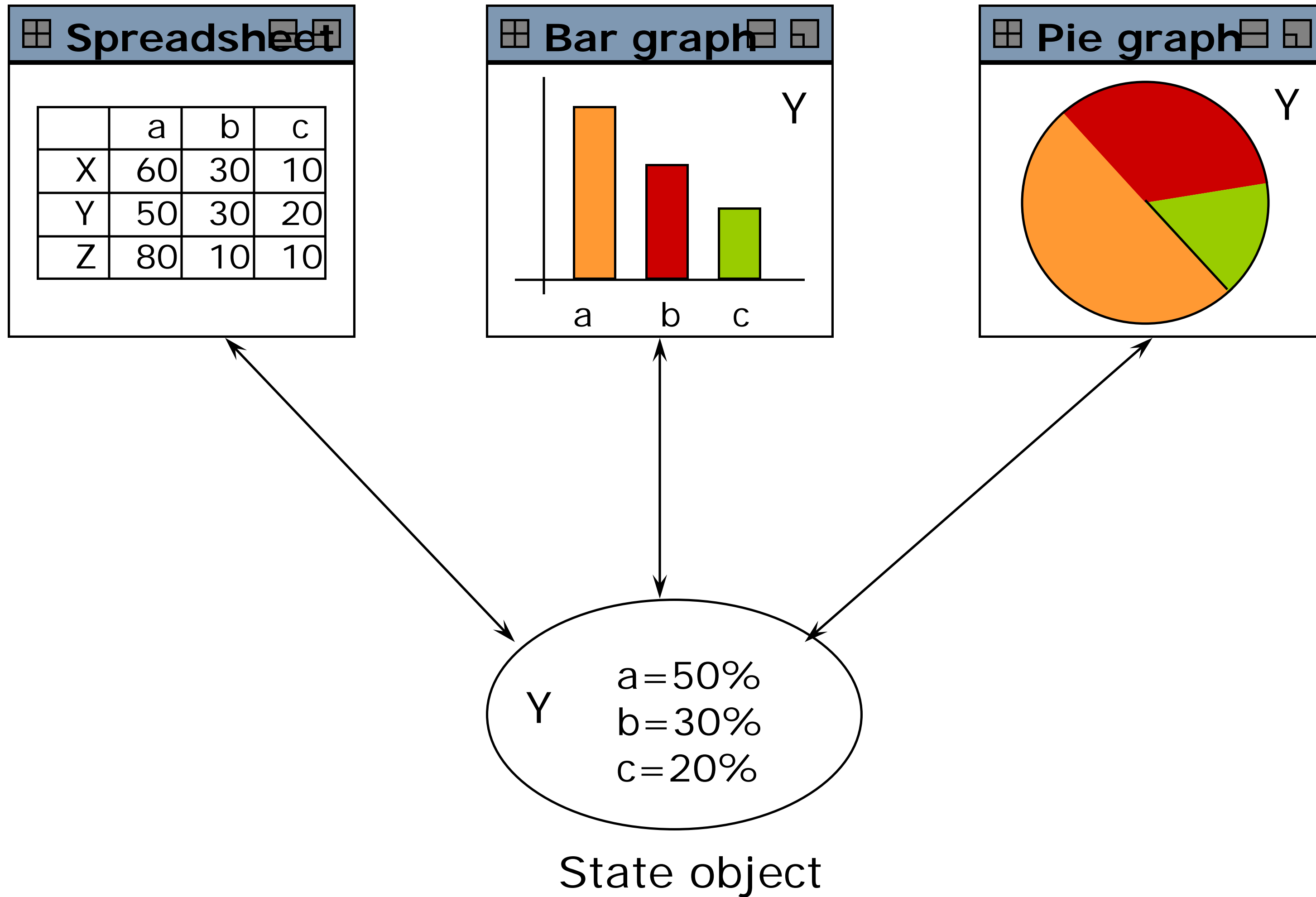
Observer Pattern (Beobachter)

Also known as *Publish/Subscribe* or *Listener* pattern

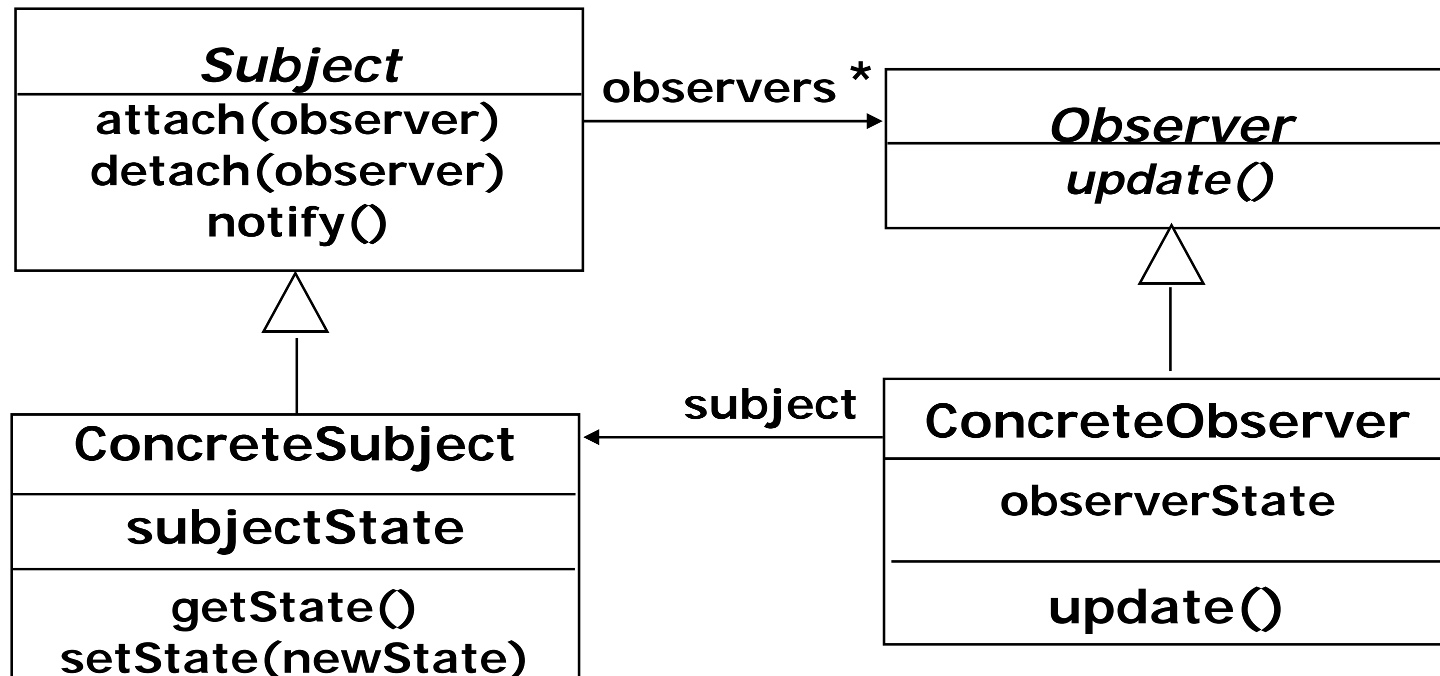
- Problem:
  Whenever one particular object changes state,
  several dependent objects must be modified
  - The number and identity of the dependent objects is not known statically

- Solution idea:
  All dependents provide the same notification interface and register with the state object
  - All state objects (called *subjects*) provide the same registration interface

Observer Pattern example

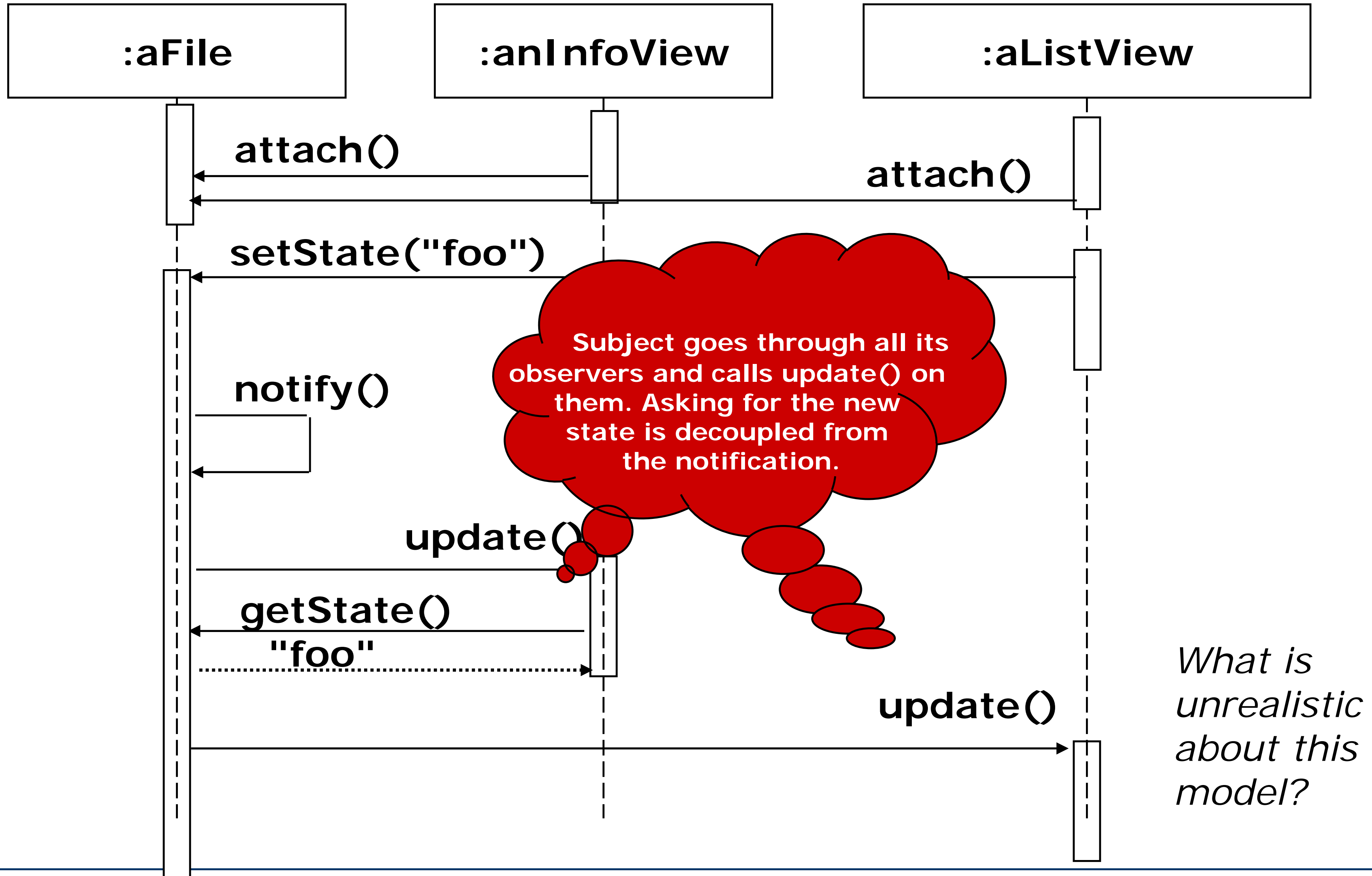


Observer Pattern (continued)



- 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 superclass, not usually an interface
  - needs to manage the list of Observers and perform notification

Sequence diagram for scenario: change filename to "foo"



What is unrealistic about this model?

Observer Pattern implementation in Java

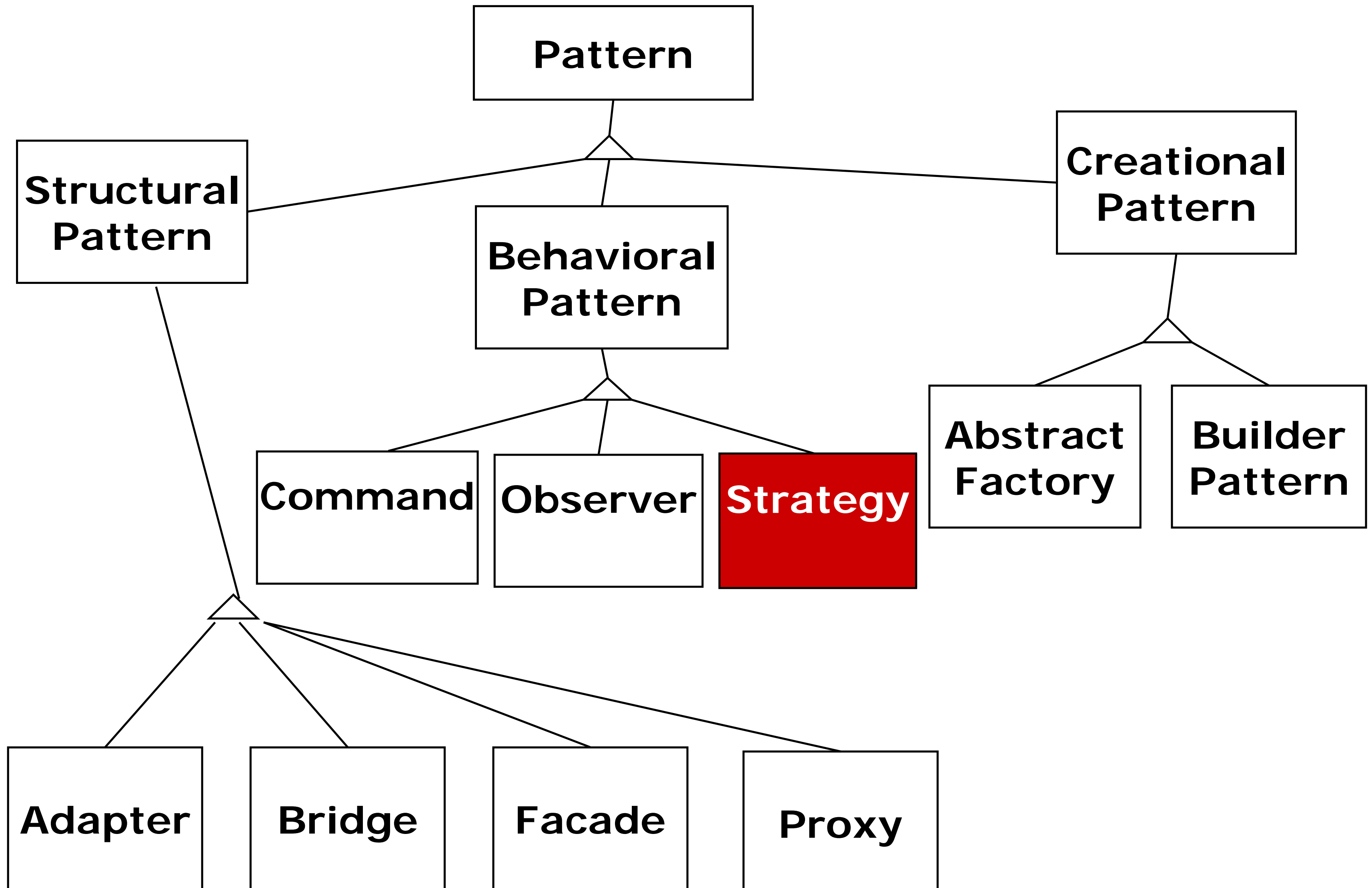
Freie Universität Berlin

```
package java.util;
public class Observable<T> {  // subject superclass
   public void     addObserver(Observer<T> o) {…}
   public void     deleteObserver(Observer<T> o) {…}
   public boolean  hasChanged() {…}
   public void     setChanged() {…}
   public void     notifyObservers() {…}
   public void     notifyObservers(T arg) {…}
}
public interface Observer<T> { // observer interface
   public void     update(Observable<T> o, T arg);
}
```

Observer Pattern example


```
package mypackage; // a subject
public class File extends Observable<String> {
  …
  public void setFilename(String filename) {
    …
    if (!this.filename.equals(filename)){
      …
      setChanged();
    }
    …
    notifyObservers(filename);
  }
  public String getFilename() {…}
}
```

A pattern taxonomy



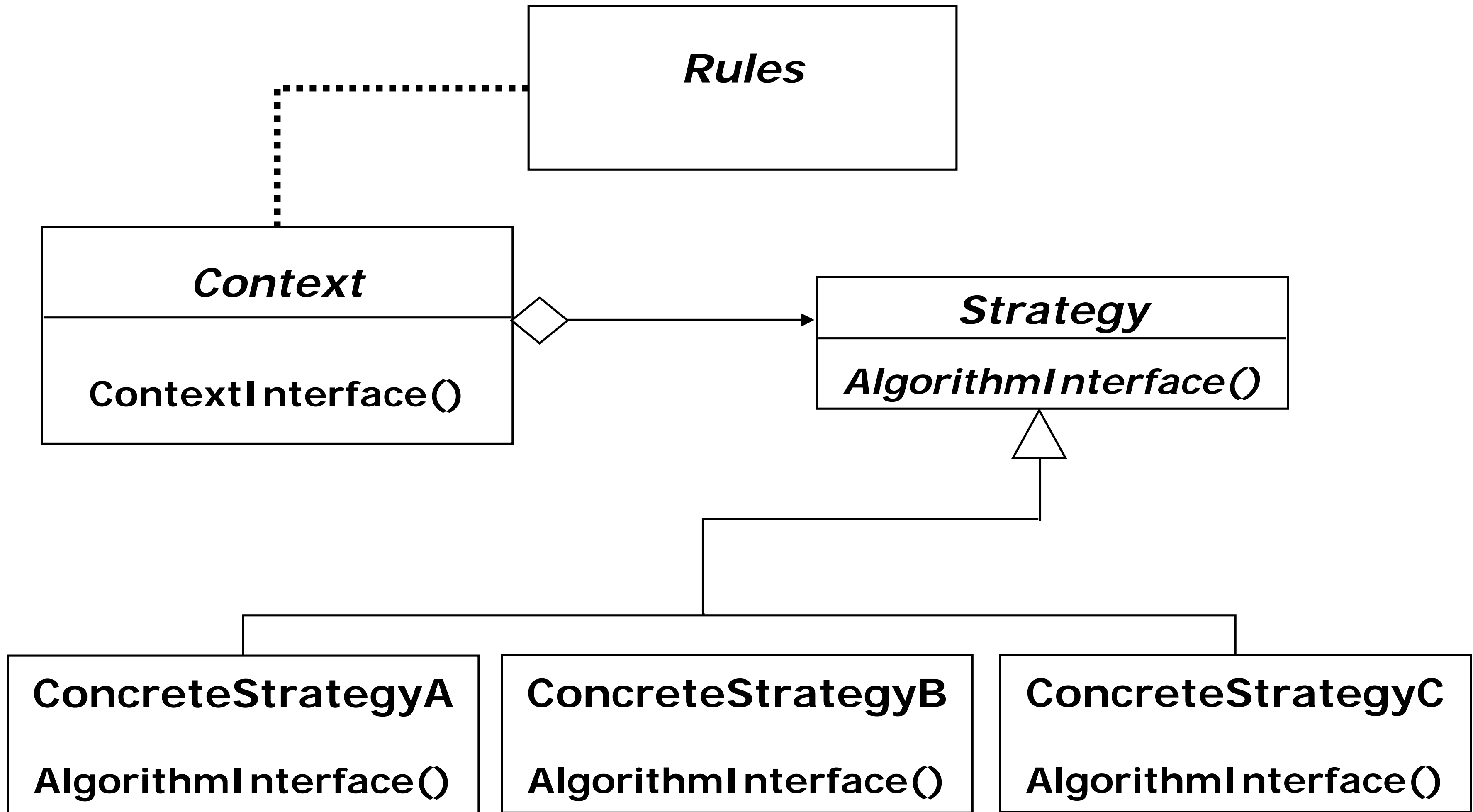
Strategy Pattern (Strategie)

Also known as *Policy* pattern

- Problem:
  There are multiple ways of doing something.
  We want to add, use, and exchange them freely (perhaps even dynamically), depending on context:
  - Different algorithms for identical results (e.g. sorting)
  - Different variants of equivalent results (e.g. codecs, file formats)
  - Different purposes (e.g. access control policies)

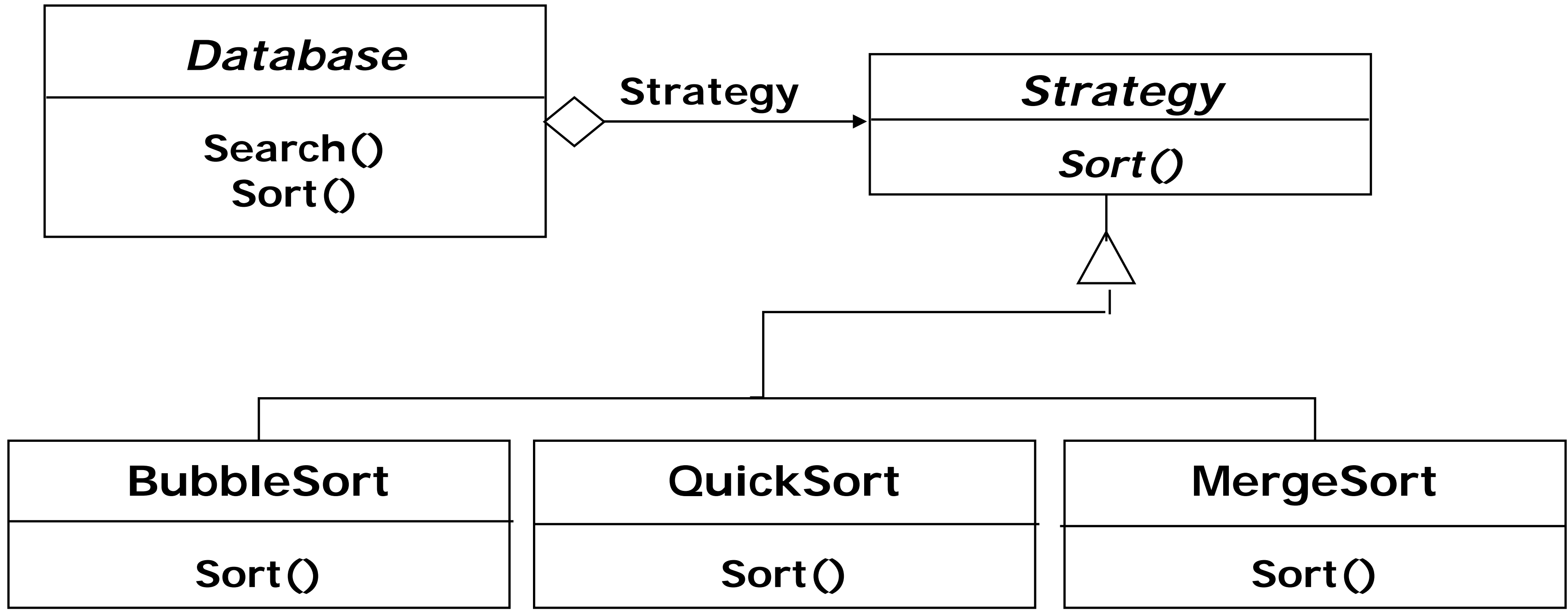
- Solution idea:
  Dynamically associate an object whose interface is the same for all variants.

Strategy Pattern

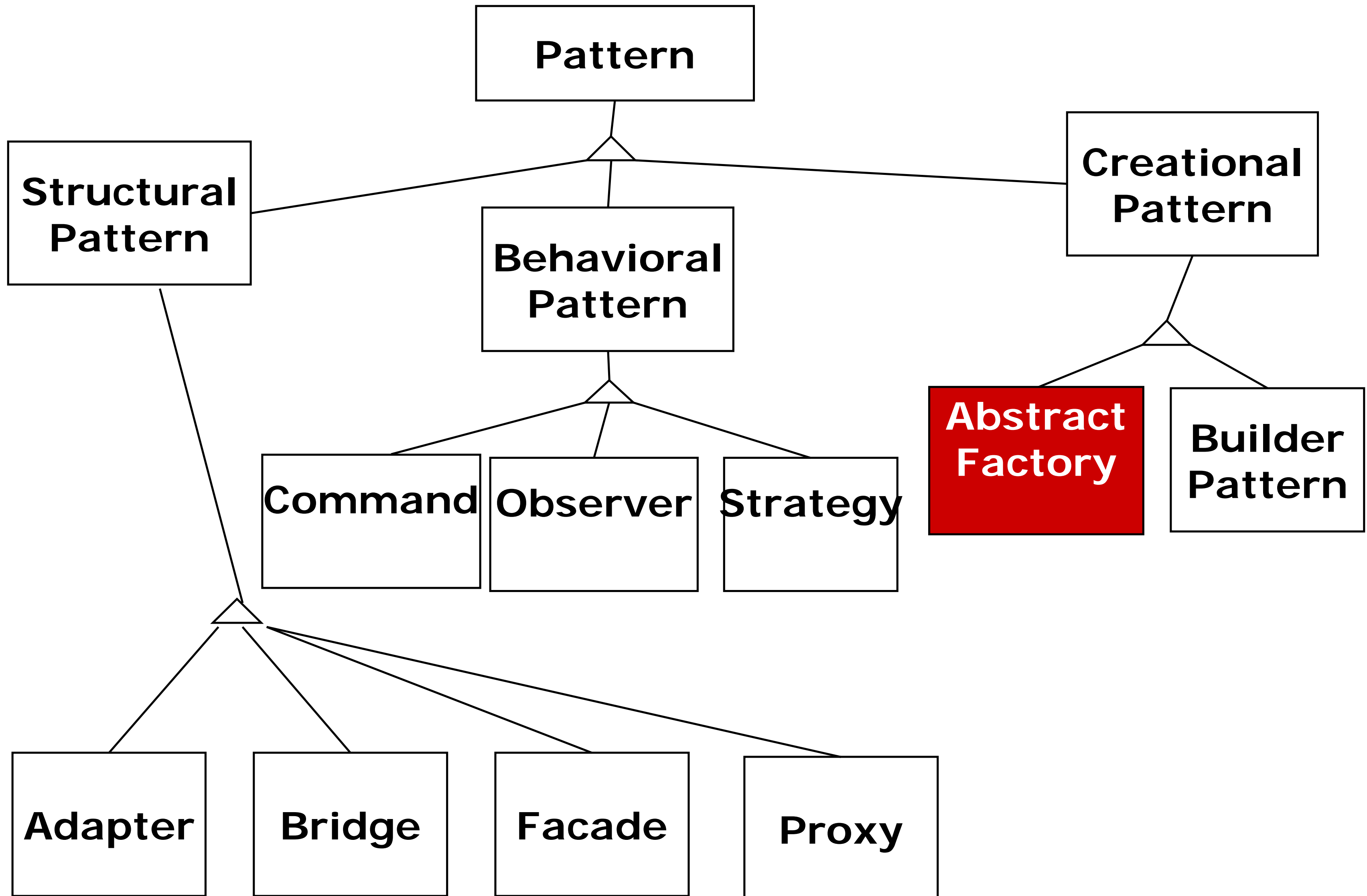


Rules may decide which Strategy is best in the current Context

Applying a Strategy Pattern in a database application



A pattern taxonomy



Abstract Factory: motivation

Examples:

- Consider a user interface toolkit that supports multiple Look-and-Feel standards such as KDE, Windows or Mac OS X
  - How can you write a single user interface and make it portable across the different window managers? ➔ Bridge Pattern
  - When using the Bridge pattern, how do we switch between implementations?

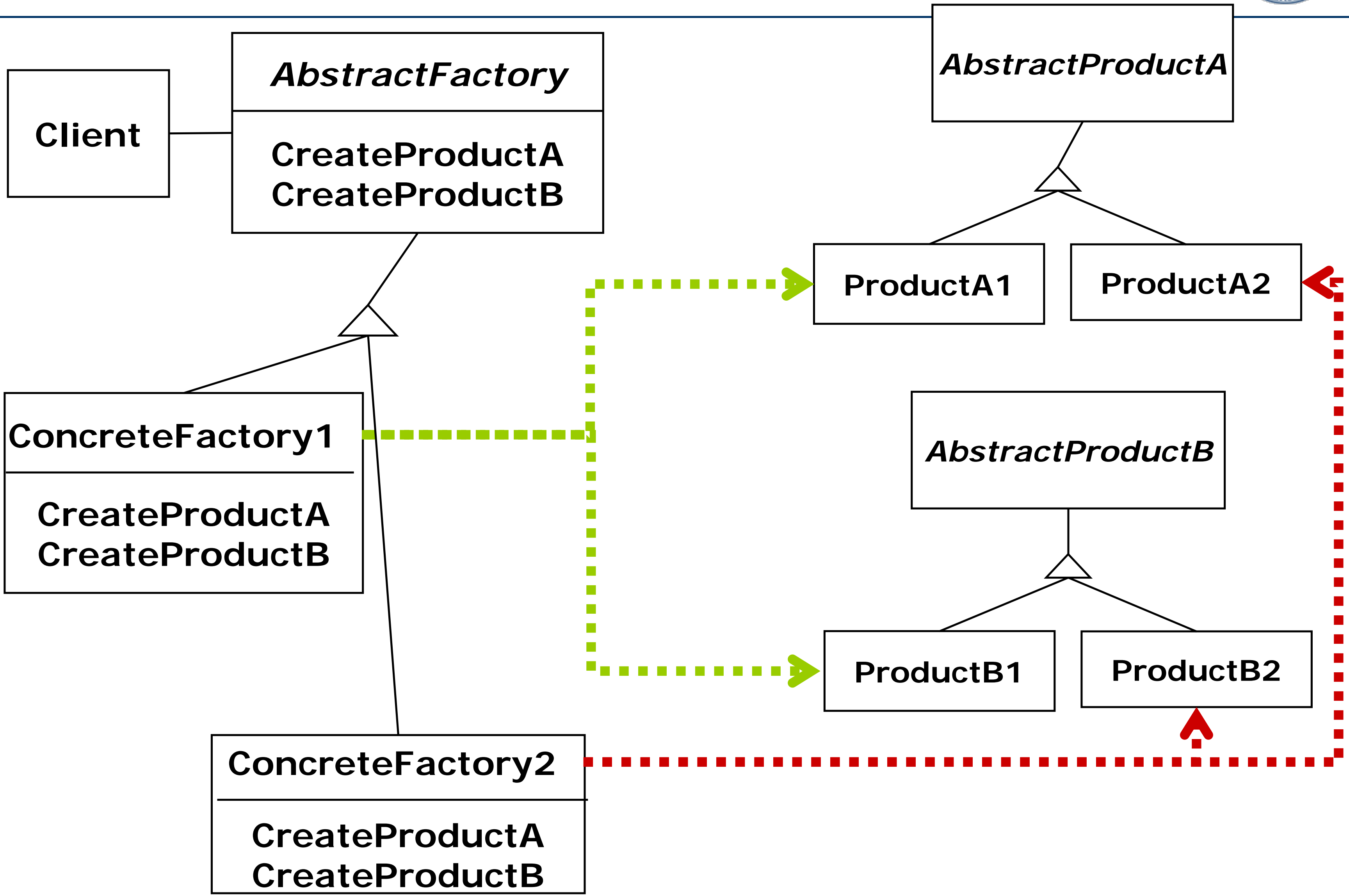
Abstract Factory Pattern

Also known as *Kit* pattern

- Problem:
  - We create many objects from a family of related classes
  - There are several implementation variants of that family
  - But for any one program run, all objects created must be from the same family variant
  - We want to create objects without caring which family variant is currently active
- Solution idea:
  - For each family *member*, all variants share the same interface
  - We create objects by calling a factory method, not a constructor
  - The factory methods are grouped in a factory class
  - There is an abstract interface for the factory class
  - and one implementation of the factory class per family variant
- Note how complex this description is!

Abstract Factory

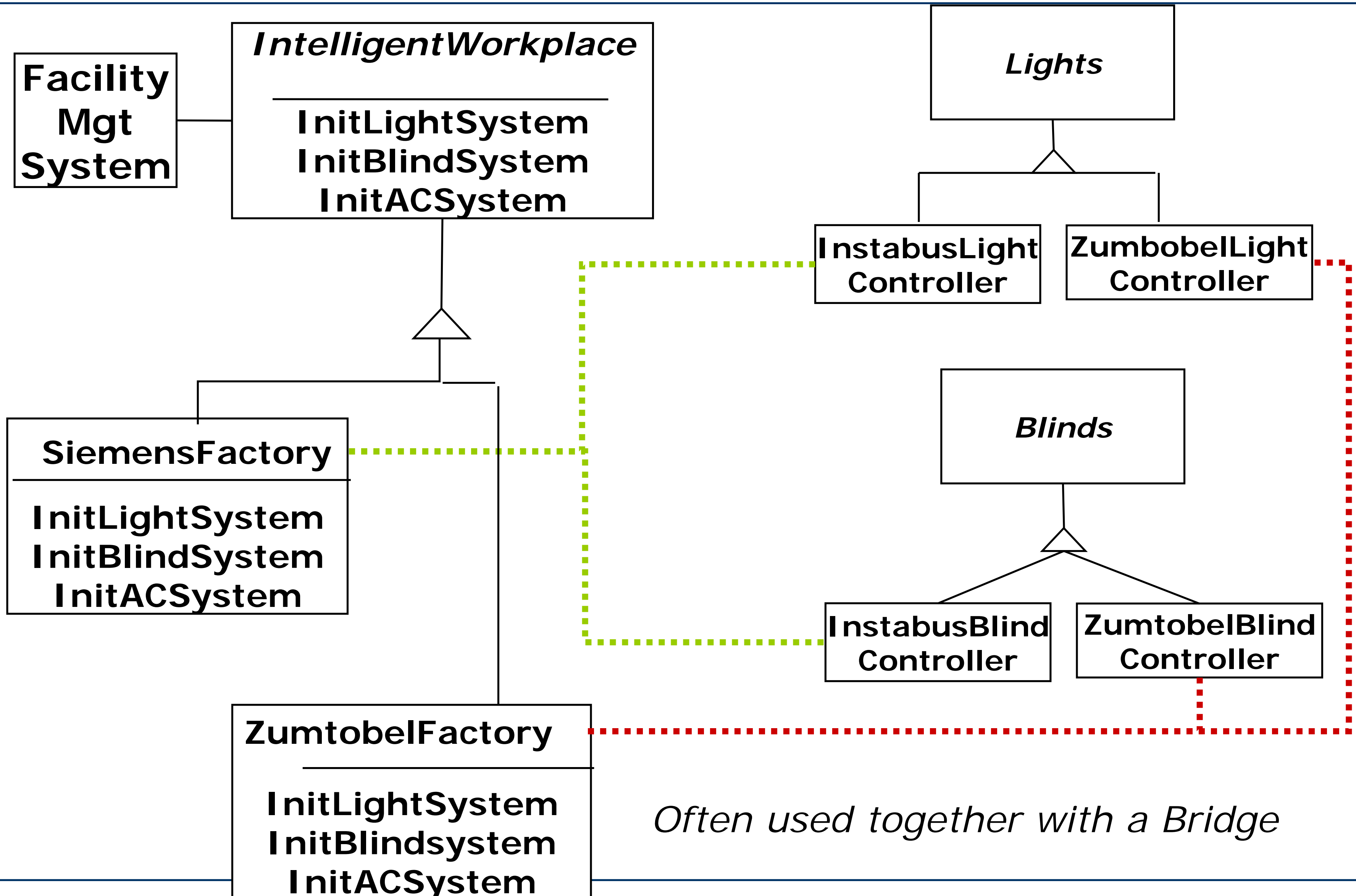
Freie Universität Berlin



Applicability for Abstract Factory Pattern

- 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
    - Used in many places in the Java API for instance with XML libs
- Constraints on related products
  - A family of related products is designed to be used together and you need to enforce this constraint
- Independence from Initialization or Representation:
  - The system should be independent of how its products are created, composed or represented
- Cope with upcoming change:
  - You use one particular product family, but you expect that the underlying technology will change soon

Example: A Facility Management System for the Intelligent Workplace

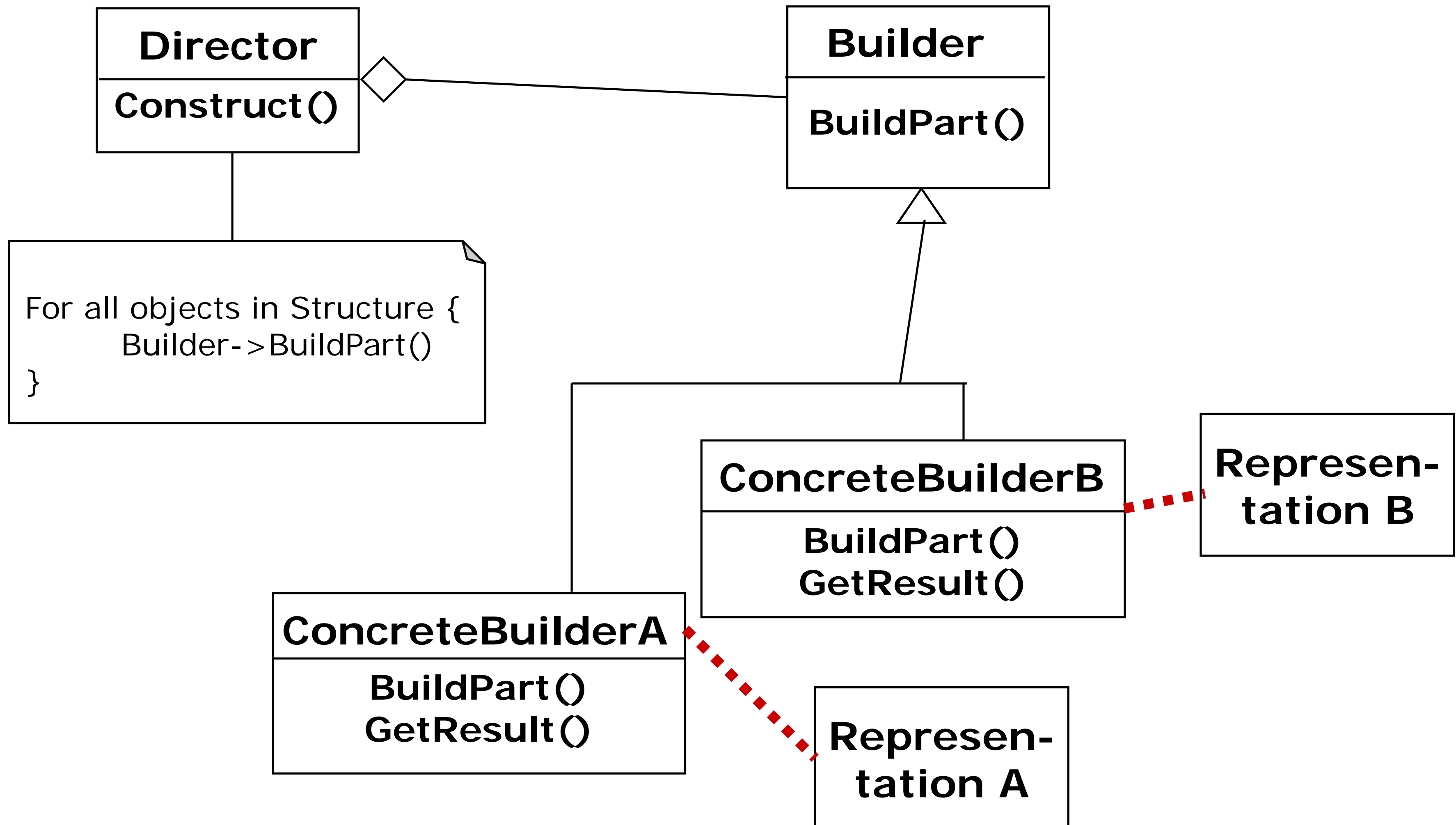


Often used together with a Bridge

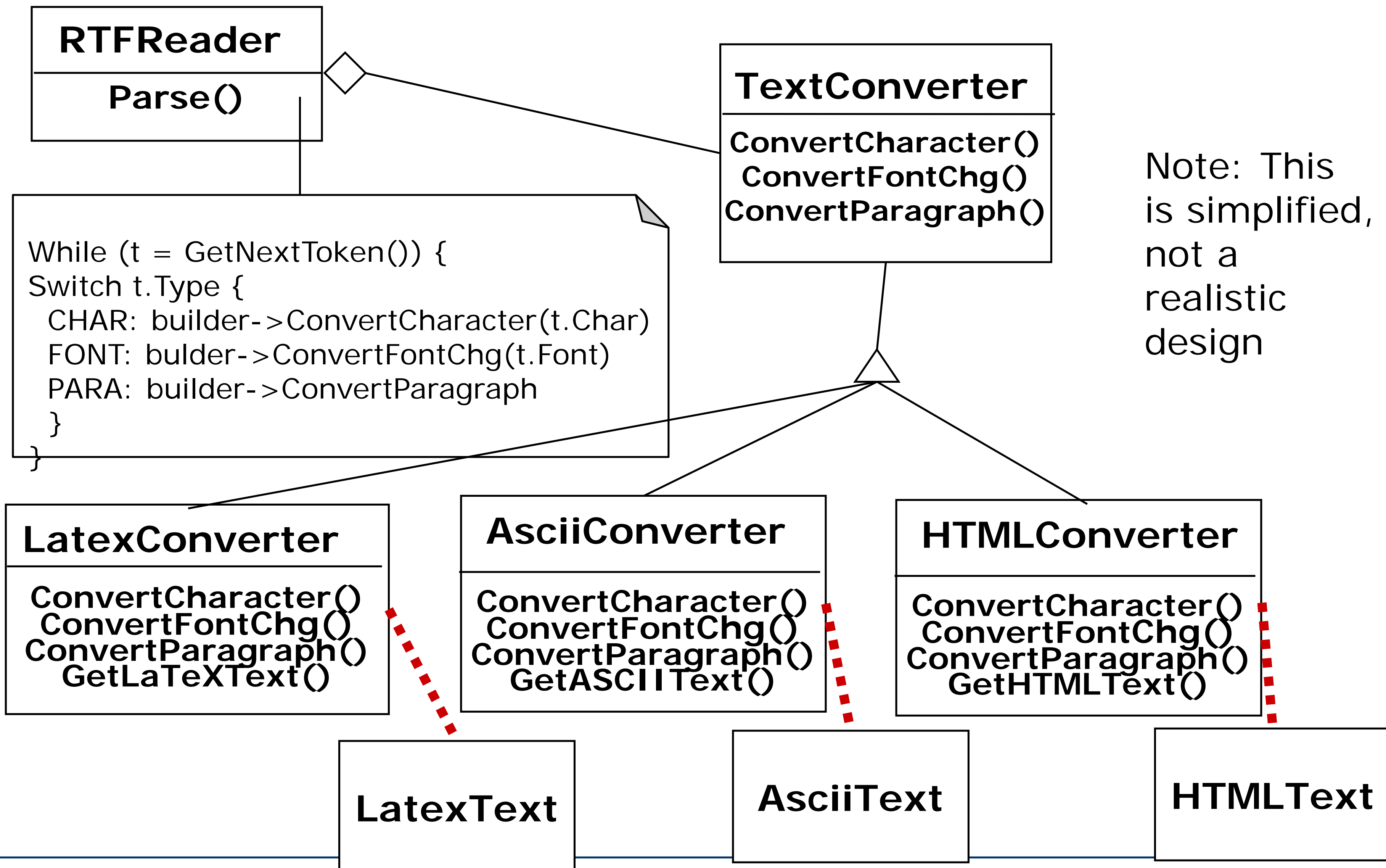
Builder Pattern: Motivation

- Conversion of documents
- Some 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 (Framemaker 4.0, Framemaker 5.0, Framemaker 5.5, LaTeX, 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.
  - A Builder subclass specializes in converting to one known format.
    - It has one method for each "build event" to be handled.
  - New Builder classes can easily be added to deal with any new format appearing on the market

Builder Pattern (Erbauer)



Builder Pattern Example



When do you use the Builder Pattern?

- The creator of a complex product must not know which of several different variant forms of the product is being built
  - The production process must look exactly the same although the products do not
- We need a simplified view of the creation process for a complex product
  - Creator should not need to know how the parts are put together to make up the product
- The creation process must allow different descriptions for the object that is constructed
  - Different build processes can lead to the same product
  - The Builder class API provides alternative abstractions

Comparison: Abstract Factory vs. Builder

- Abstract Factory
  - Focuses on product family
    - The products can be simple ("light bulb") or complex ("engine")
  - The creation process takes only <u>one step</u>
    - The product is immediately returned
- Builder
  - Creates only one type of product
  - The creation process is complex: many separate steps
    - and those steps can vary a lot
- Abstract Factory and Builder work well together for a family of multiple complex products
  - One or more of the factory methods may yield a Builder rather than directly a product object

Summary

- Structural Patterns — Adapter, Bridge, Façade, Proxy
  - Focus: How objects are composed to form larger structures
  - Problems solved:
    - Provide flexibility and extensibility
- Behavioral Patterns — Command, Observer, Strategy
  - Focus: Algorithms and the assignment of responsibilities to objects
  - Problem solved:
    - Reduce coupling to a particular algorithm
- Creational Patterns — Abstract Factory, Builder
  - Focus: Creation of complex objects
  - Problems solved:
    - Hide how complex objects are created and put together

Conclusion

Design patterns...

- provide reusable solution ideas for recurring problems
- lead to extensible models and code
- simplify talking about a design
  - Because they provide powerful abstractions
- are examples of change-resistant design
  - using interface inheritance and delegation
- apply the same principles to structure and to behavior

Thank you!