Design Patterns
Part II

Software Engineering
Lecture 14

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Outline of the Lecture

- Short review of the concepts from the previous lecture
  - What is a design pattern?
  - Modifiable designs

More patterns:

- **Proxy:** Provide Location transparency
- **Command:** Encapsulate control flow
- **Observer:** Provide publisher/subscribe mechanism
- **Strategy:** Support family of algorithms, separate of policy and mechanism
- **Template Pattern:** Support the structure of an algorithm, with steps to be filled in
- **Abstract Factory:** Provide manufacturer independence
- **Builder:** Hide a complex creation process.

Schedule for Final Exam

- Saturday 17 February 2007
- Time: 10-12:30

Winner of Yesterday's Competition

- 3. Prize:
  - Atanas Gregov
- 2. Prize:
  - Vladislav Lazarov
- 1. Prize:
  - Lejing Wang, Eduardo Aguilar, Irena Kostadinovic, Carla Guilen

Review: Design pattern

A design pattern is…

…a template for a solution to a recurring design problem

- You can search a library of existing design knowledge before re-inventing the wheel

…reusable design knowledge

- You can learn design by studying existing designs

…an example of *modifiable* design

- You can extend and customize an existing design.

Definitions

- Extensibility (Expandibility)
  - A system is extensible, if new functional requirements can easily be added to the existing system
- Customizability
  - A system is customizable, if new nonfunctional requirements can be addressed in the existing system
- Scalability
  - A system is scalable, if existing components can easily be multiplied in the system
- Reusability
  - A system is reusable, if it can be used by another system without requiring major changes in the existing system model (design reuse) or code base (code reuse).

What makes a Design reusable?

- Low coupling between subsystems and high cohesion within subsystems
- Clear dependencies
- Explicit assumptions

How do design patterns help?

- They are generalizations from existing designs
- They provide a shared vocabulary to designers
- They provide examples of reusable designs
  - Inheritance (abstract classes)
  - Delegation (or aggregation)

Why are reusable Designs important?

A reusable design…

…enables an iterative and incremental development cycle

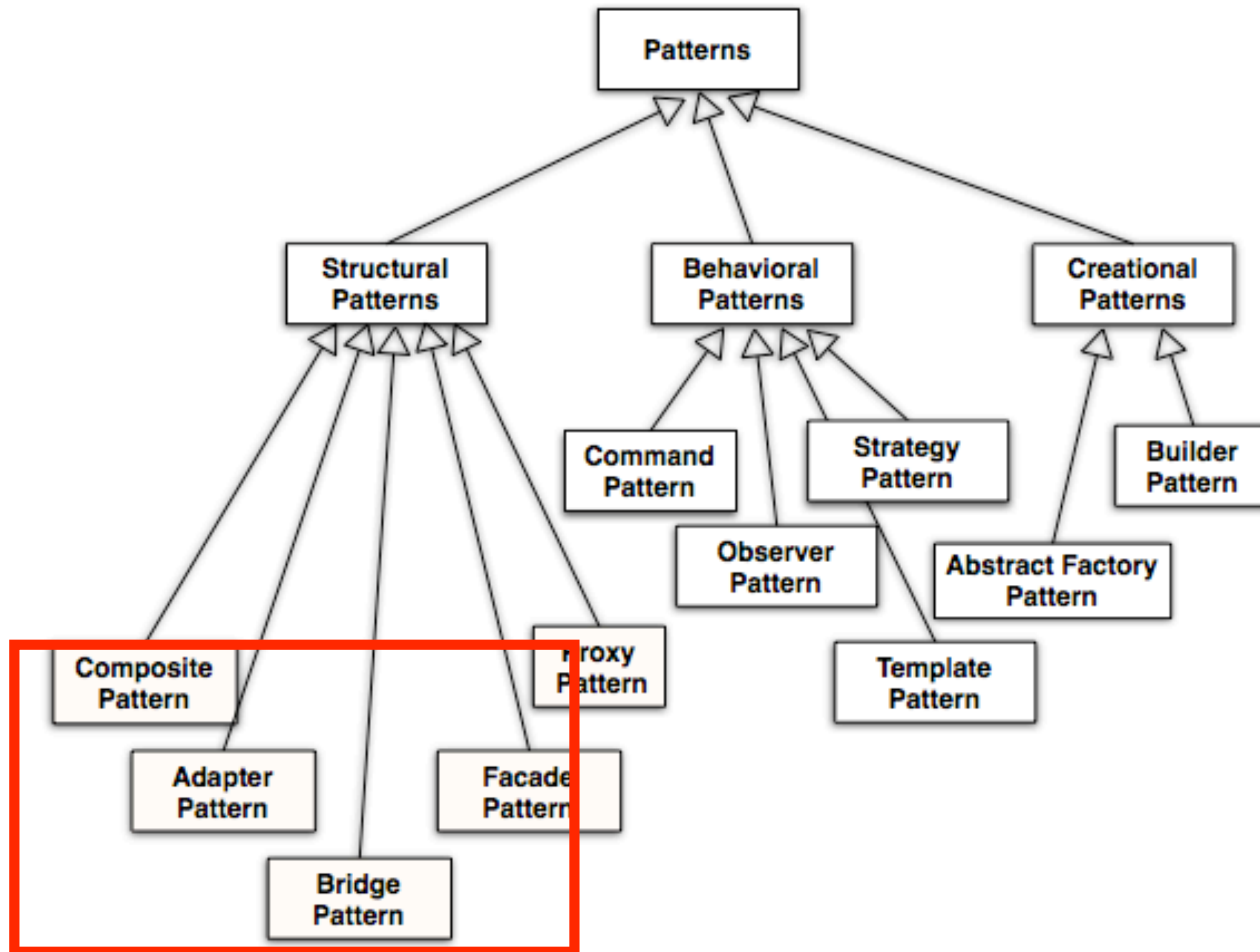
- concurrent development
- risk management
- flexibility to change

…minimizes the introduction of new problems when fixing old ones

…allows the delivery of more functionality after an initial delivery (Extensibility).

Categorization of Patterns

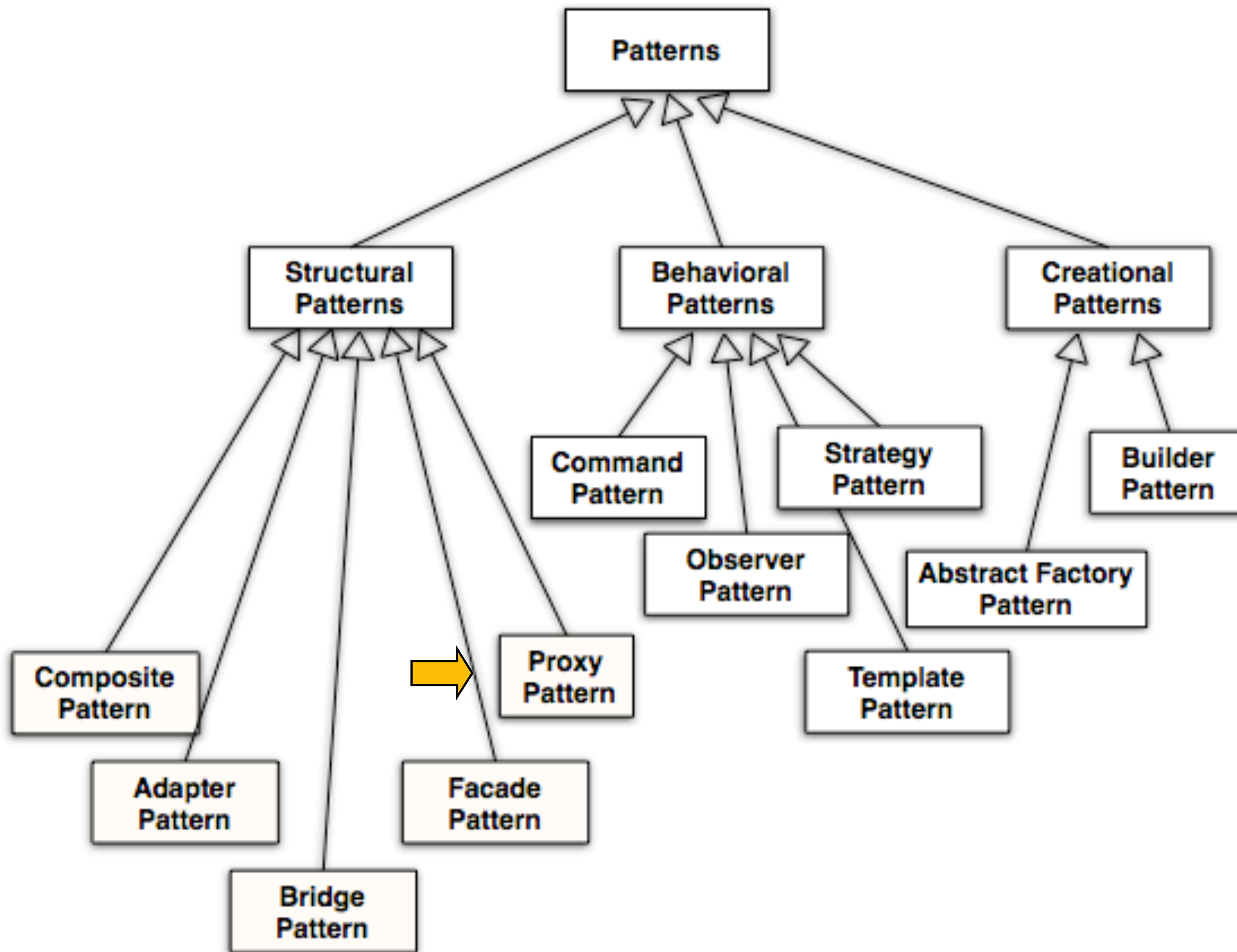
- Structural Patterns
  - reduce coupling between two or more classes
  - introduce an abstract class to enable future extensions
  - encapsulate complex structures
- Behavioral Patterns
  - allow a choice between algorithms and the assignment of responsibilies to objects (“Who does what?”)
  - characterize complex control flows that are difficult to follow at runtime
- Creational Patterns
  - allow to abstract from complex instantiation processes
  - make the system independent from the way its objects are created, composed and represented.



© 2007 Bernd Bruegge

Lecture Plan for Today

- Structural patterns
  - Proxy
- Behavioral patterns
  - Command
  - Observer
  - Strategy
  - Template
- Creational patterns
  - Abstract Factory
  - Builder



Patterns
Structural Patterns
Behavioral Patterns
Creational Patterns
Command Pattern
Strategy Pattern
Builder Pattern
Observer Pattern
Abstract Factory Pattern
Proxy Pattern
Template Pattern
Composite Pattern
Adapter Pattern
Facade Pattern
Bridge Pattern

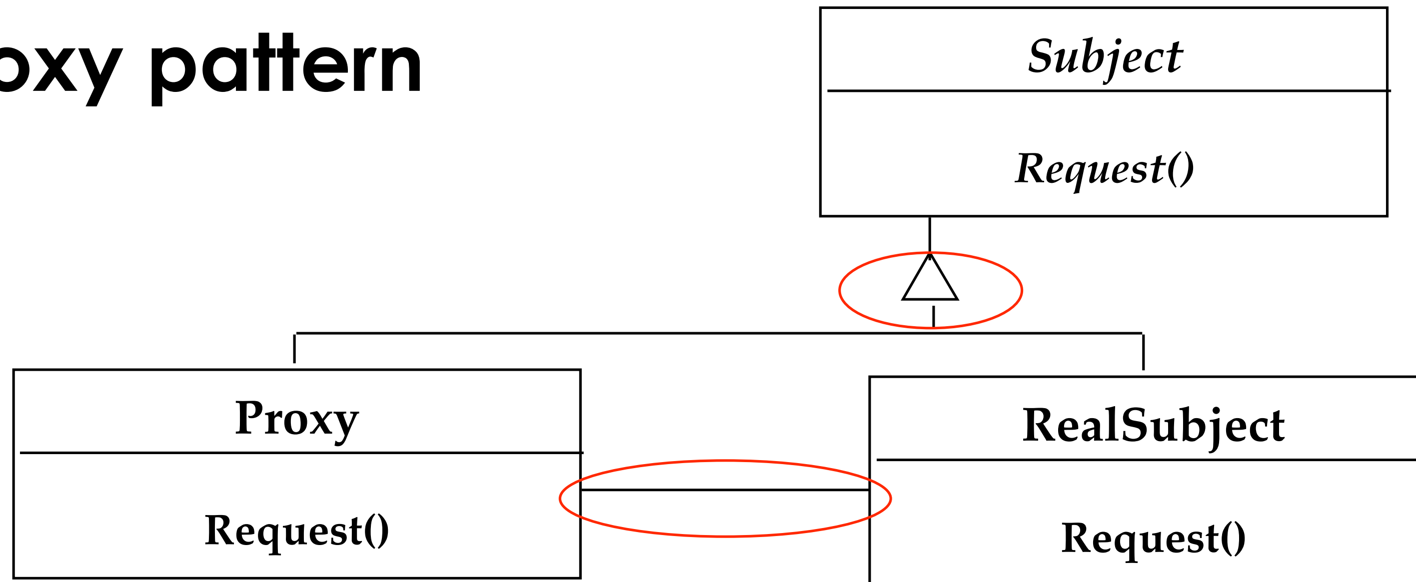
Proxy Pattern: Motivation

- I am sitting at my 768Kb DSL modem connection and try to retrieve a page during a busy time.
- I am getting 10 bits/sec.
- What can I do?

Proxy Pattern

- Design Problem: What is particularly expensive in object-oriented systems?
  - Object creation
  - Object initialization
- Solution:
  - 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

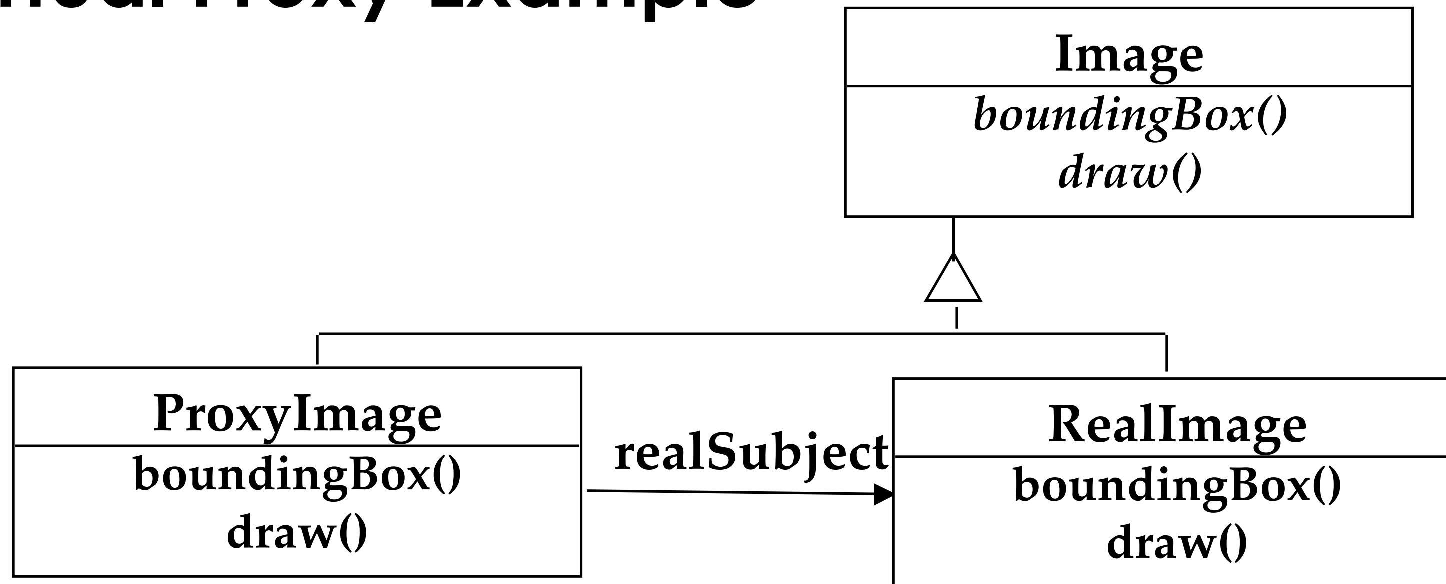


- Interface inheritance is used to specify the interface shared by Proxy and RealSubject
- Delegation is used by Proxy to forward any accesses to the RealSubject (if desired).

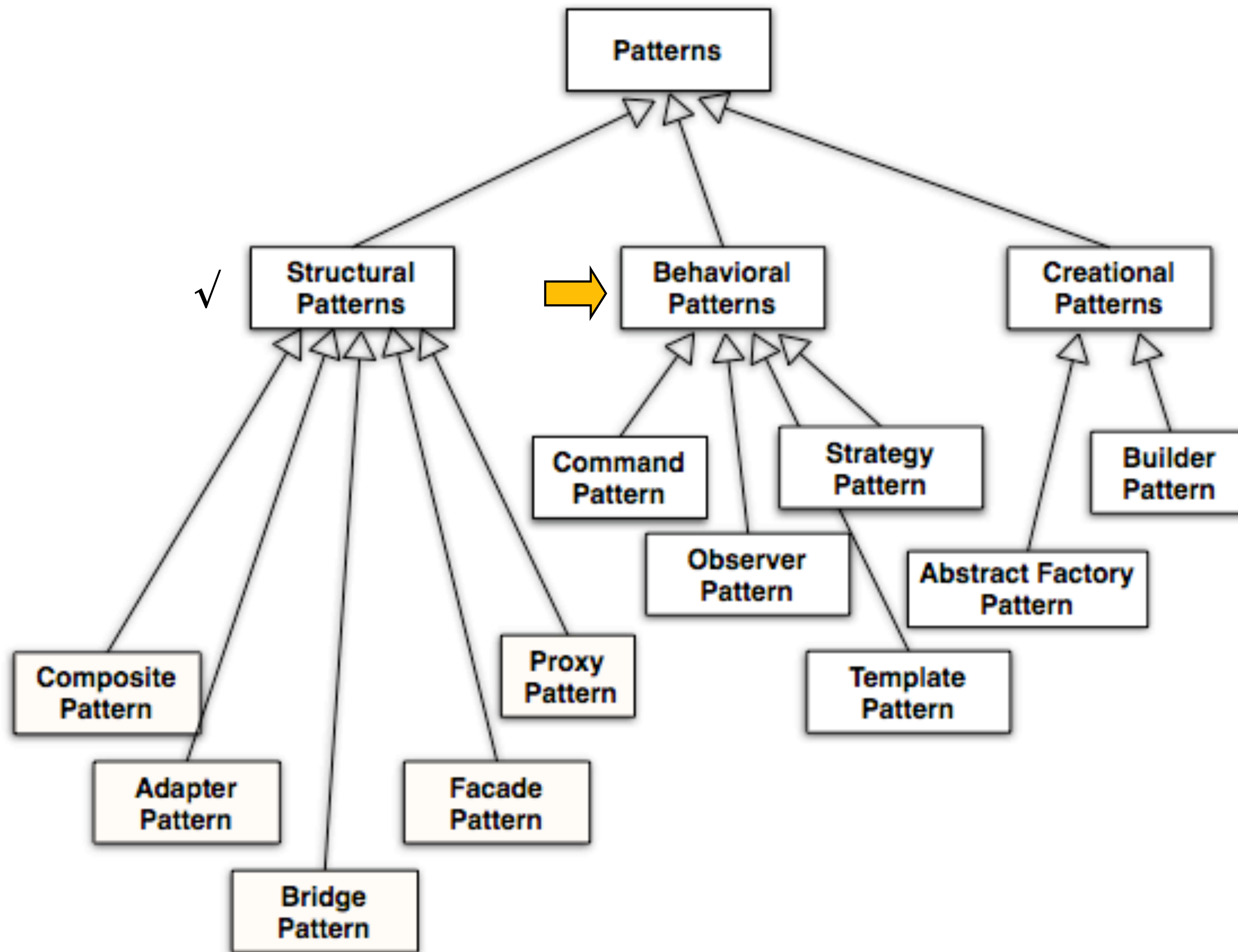
Proxy Applicability

- Remote Proxy:
  - The Proxy object is a 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. The Proxy object is a standin
  - Good if the real object is rarely accessed
- Protection Proxy:
  - The Proxy object provides access control to the real object
  - Good when different objects should have different access and viewing rights for the same document
    - Example: Grade information accessed by administrators, teachers and students.

Virtual Proxy Example



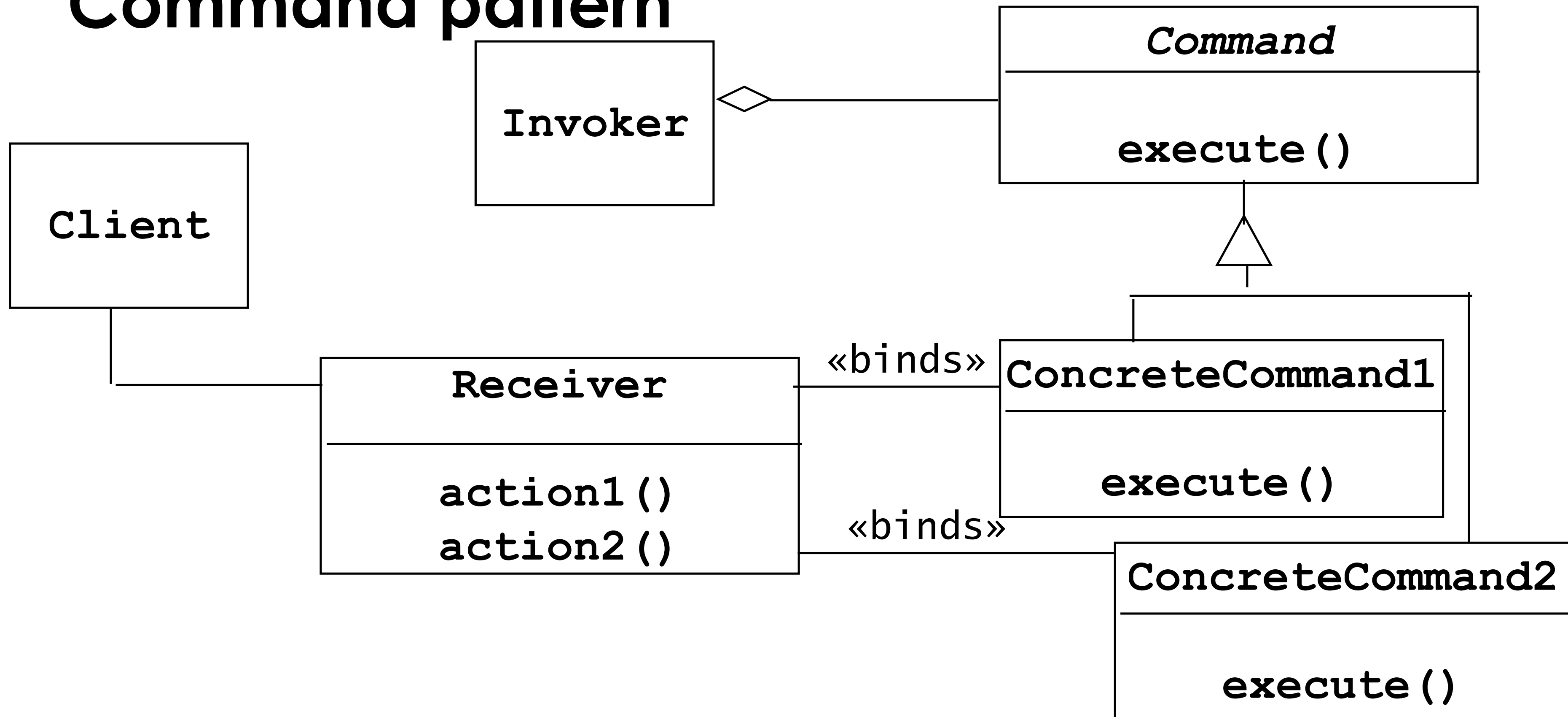
- The RealImage is stored and loaded separately
- If the RealImage is not loaded, a ProxyImage draws a grey rectangle in place of the image
- The class user of Image cannot tell, if it is dealing with ProxyImage instead of RealImage
- A proxy pattern can be easily combined with a Bridge.


Patterns
Structural Patterns
Behavioral Patterns
Creational Patterns
Command Pattern
Strategy Pattern
Builder Pattern
Observer Pattern
Abstract Factory Pattern
Composite Pattern
Proxy Pattern
Template Pattern
Adapter Pattern
Facade Pattern
Bridge Pattern

Command Pattern: Motivation

- You want to build a user interface
- You want to provide menus
- You want to make the menus reusable across many applications
  - The applications only know what has to be done when a command from the menu is selected
  - You don't want to hardcode the menu commands for the various applications
- Such a user interface can easily be implemented with the Command Pattern.

Command pattern



- **Client** (usually a user interface builder) creates a **Concrete-Command** and binds it to an action operation in **Receiver**
- Client hands the ConcreteCommand over to the **Invoker** which stores it (for example in a menu)
- The Invoker has the responsibility to execute() the command (based on a string entered by the user).

Comments to the Command Pattern

- The abstract class **Command** declares the interface supported by all ConcreteCommands
- The **Client** is a class in a user interface builder or in a class executed during startup of the application to build the user interface
- The client creates subclasses of Command, **ConcreteCommands,** and binds them to specific **Receivers** of type string. These strings are entered by the user (Examples: “commit”, “execute”, “undo”)
  - All user-visible commands are subclasses of Command
- The **Invoker** class - in the application program offering a menu of commands - selects the ConcreteCommand based on the string and the binding between **action()** and ConcreteCommand.

Advantages of the Command Pattern

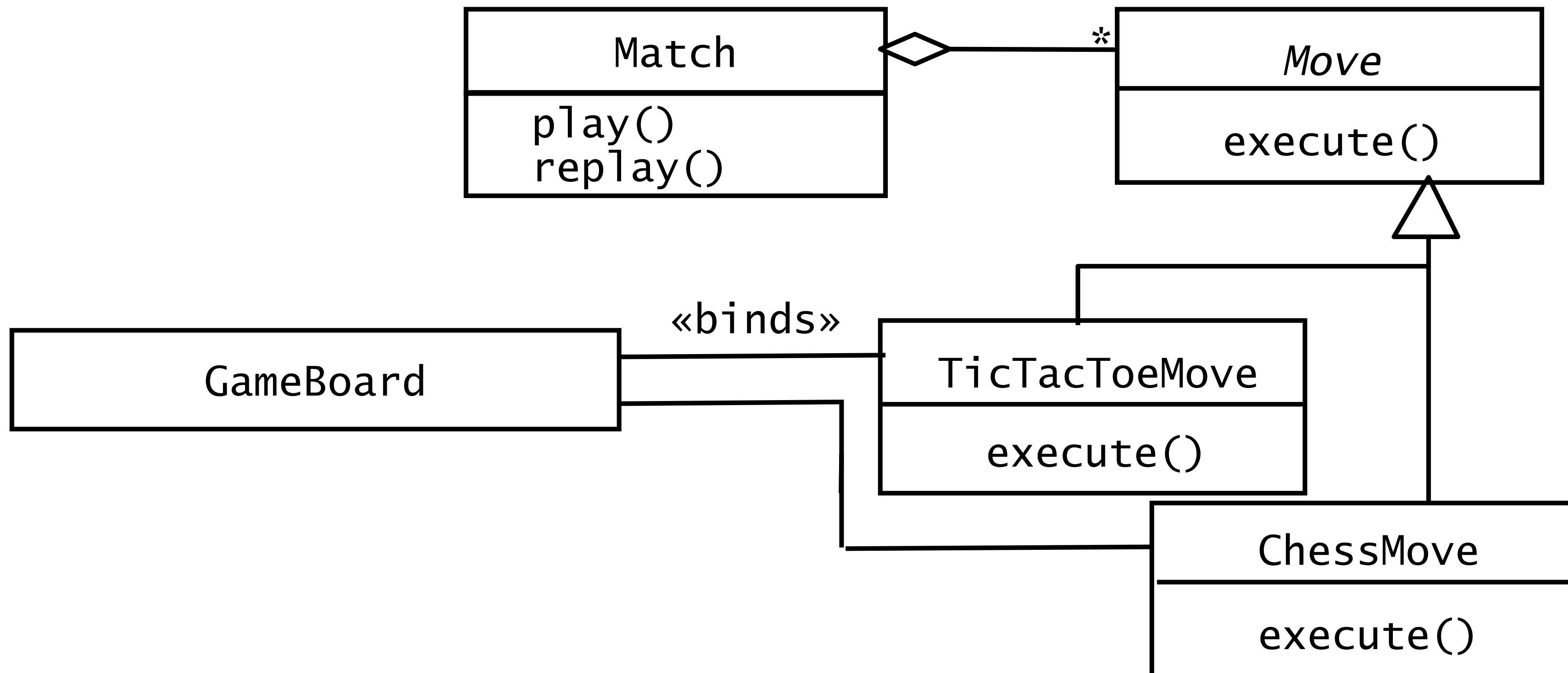
- The command pattern can be nicely used to decouple boundary objects from control objects:
  - Examples of boundary objects:
    - menu items, buttons,
- Only the boundary objects can create and send messages to objects of type **Command**
- Only objects of type **Command** can modify entity objects
- When the user interface is changed (for example, a menu bar is replaced by a tool bar), only the boundary objects have to be modified.

Command Pattern Applicability

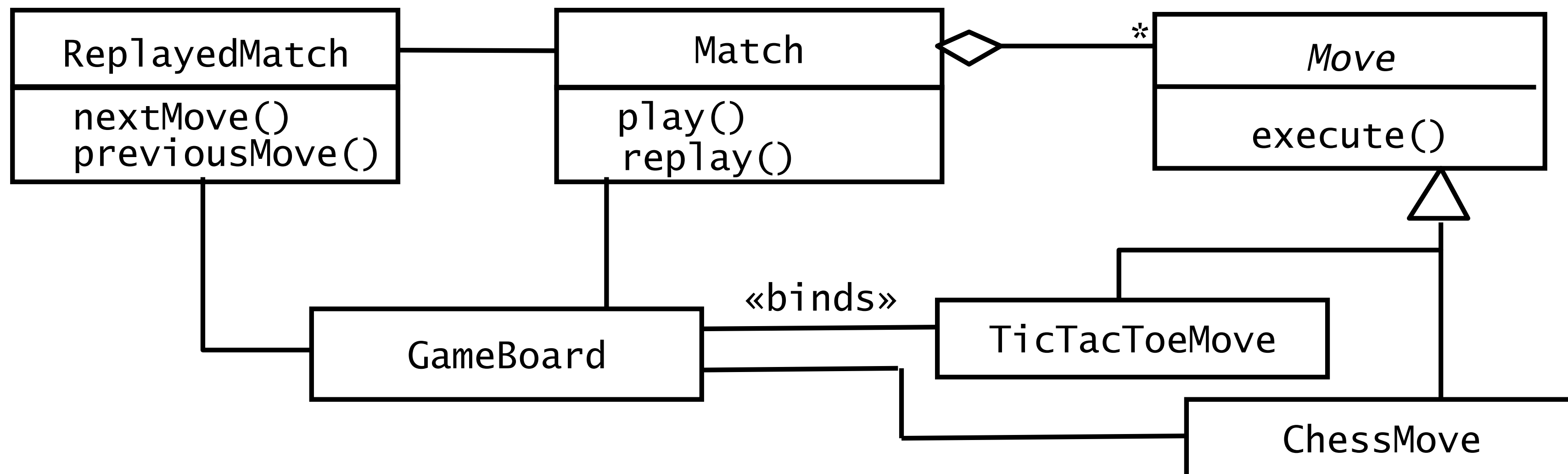
- Parameterize clients with different requests
- Queue or log requests
- Support undoable operations

- Uses:
  - Undo queues
  - Database transaction buffering

Applying the Command Pattern to Command Sets



Applying the Command design pattern to Replay Matches in ARENA



Observer Pattern Motivation

- Models a 1-to-many dependency between objects
  - When one object changes state, all its dependents are notified and updated automatically.
- Also called Publish and Subscribe

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

Miscellaneous

- Mid-Term
  - Quiz-based exams out today
    - Published on lecture portal if you agreed to the internet-based publication of your grade
    - Otherwise posted in the glass box in front of Max Koegel's office
  - Project-based exams out on Thursday
- Next exercise session still in multimedia room 2
  - Space still a constraint, but get to know each other:-)
- Interesting events
  - CDTM: Thursday 19:00, room 2502 in TUM, Arcisstrasse.
    - Look at www.cdtm.de
    - A few more details in tomorrow's lecture

3 Views of the File Name for a Presentation

3 Possibilities to really change the File name

List View

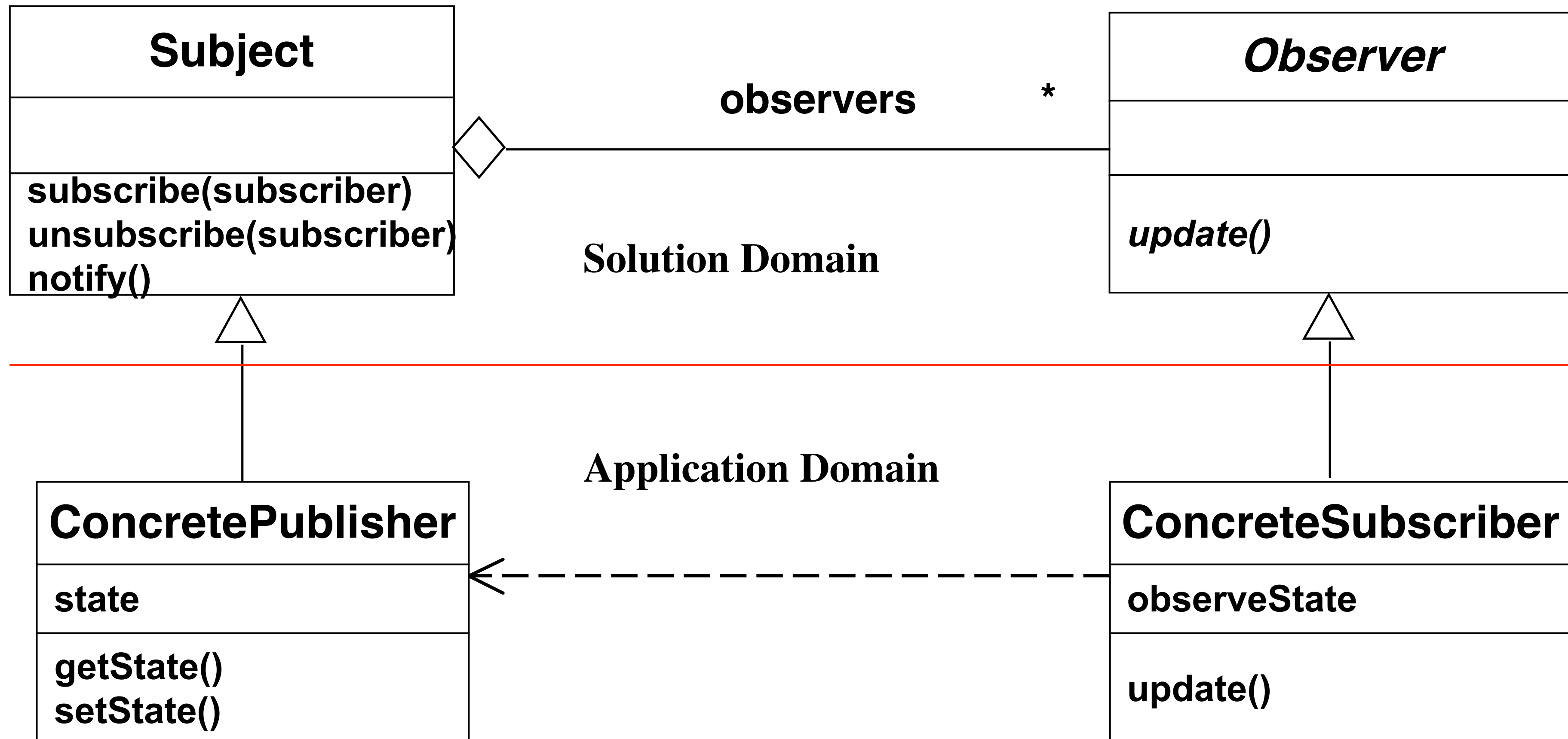
Powerpoint View

InfoView

What happens
if I change
the file name of this presentation
to foo?

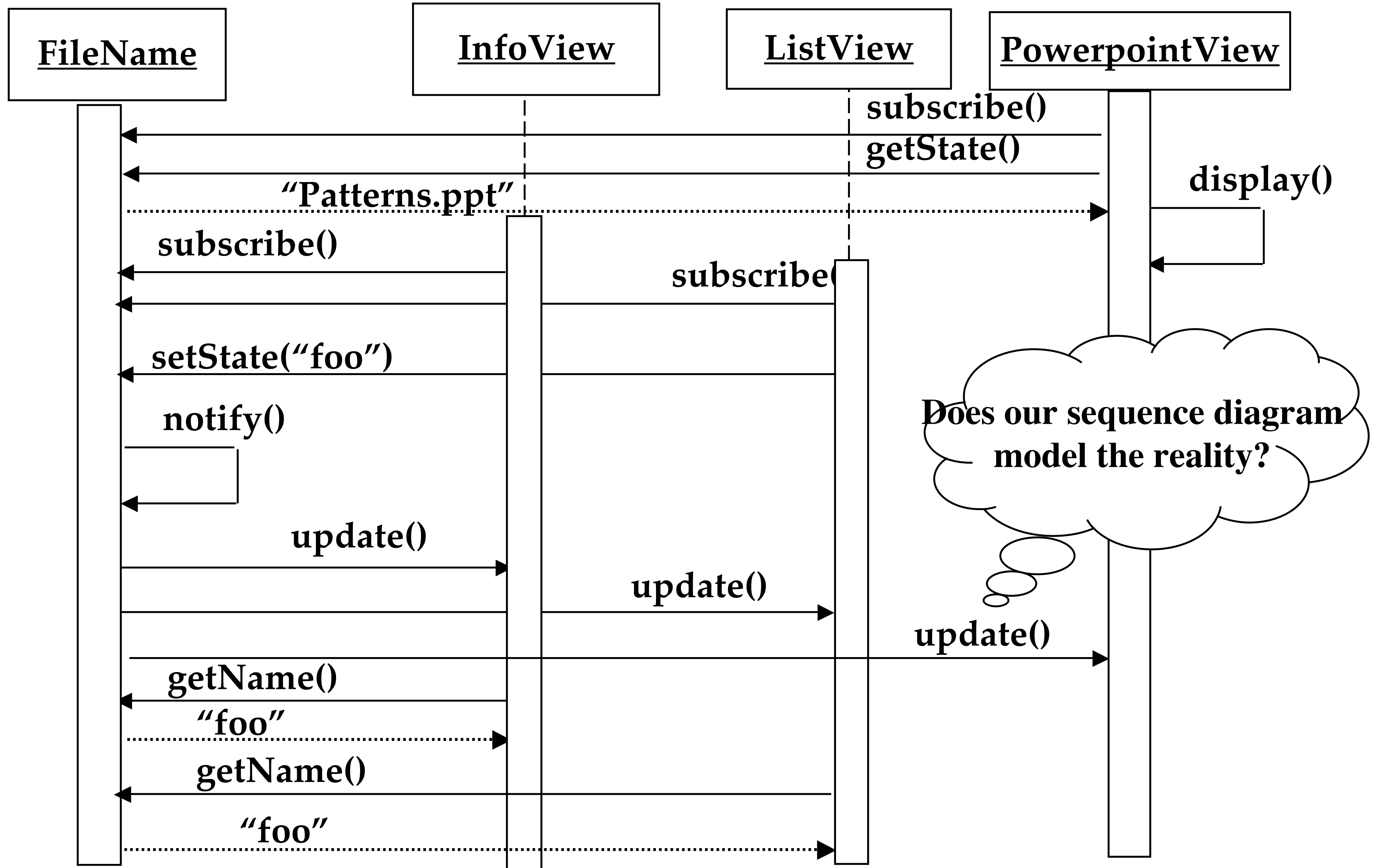
© 2007 Bernd

Observer Pattern: Decoupling Entities from Views

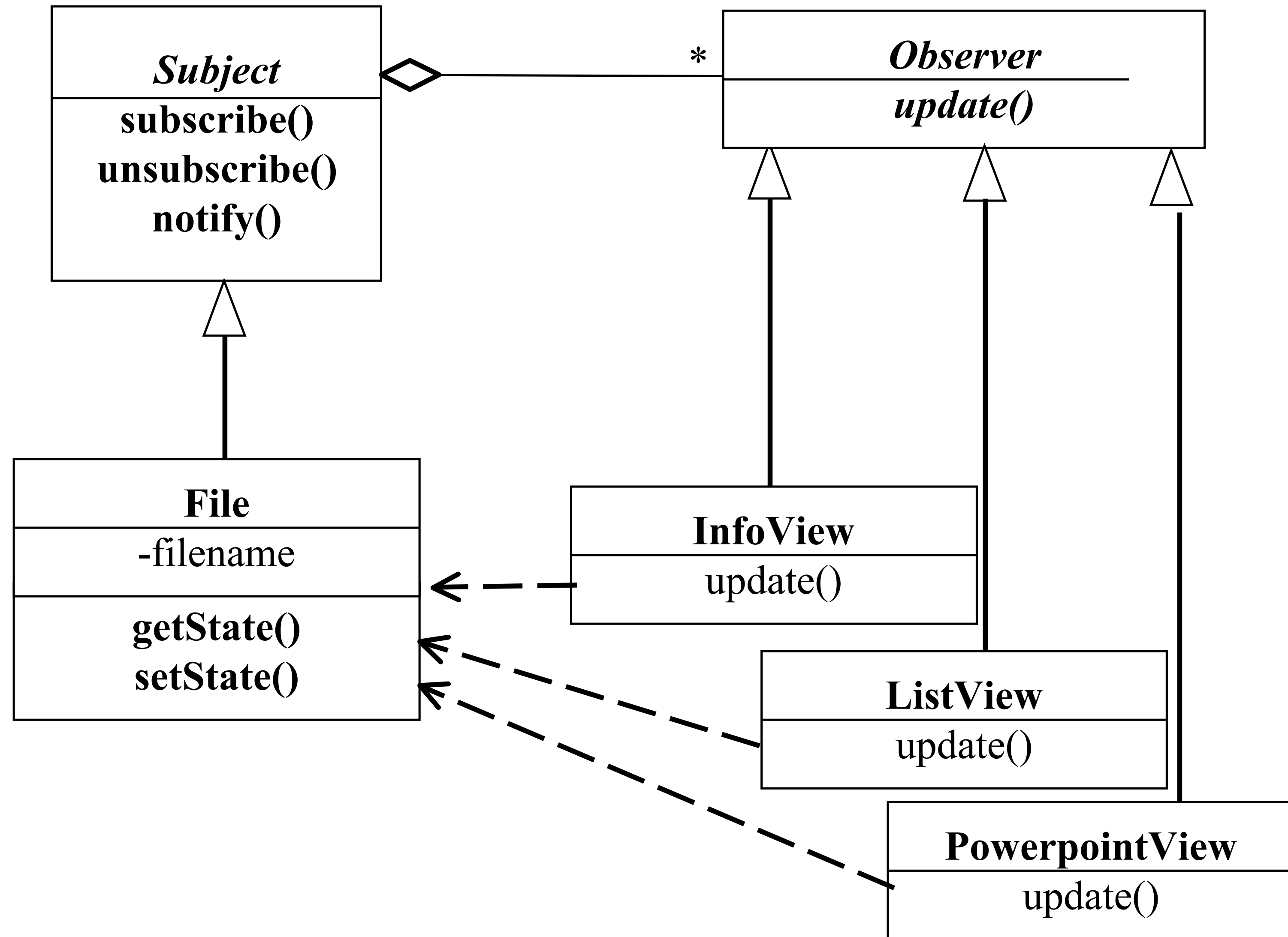


The **Subject** (“Publisher”) represents the actual state, each **Observer** (“Subscriber”) represents a different view of the state.

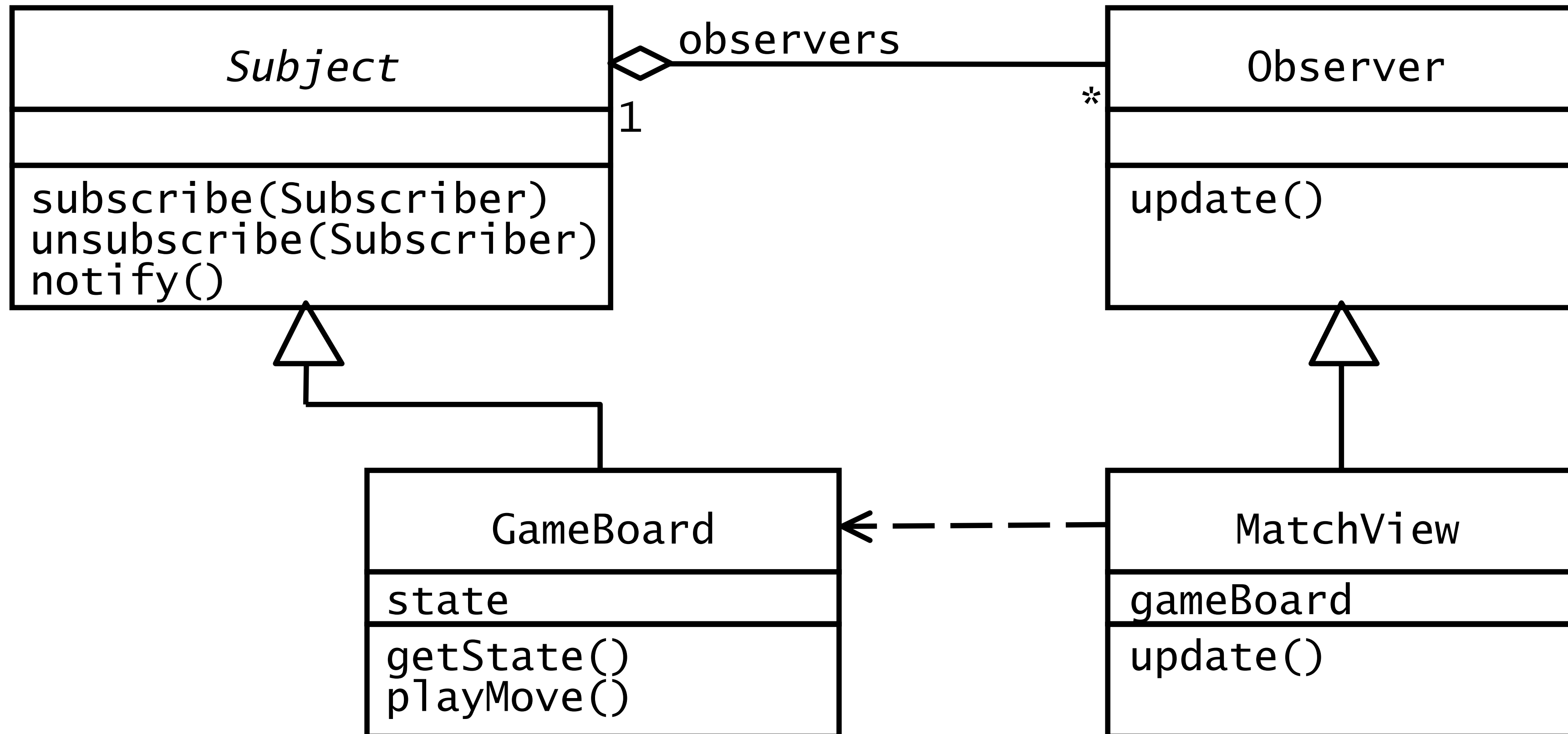
Modeling the scenario: Change FileName to "foo"



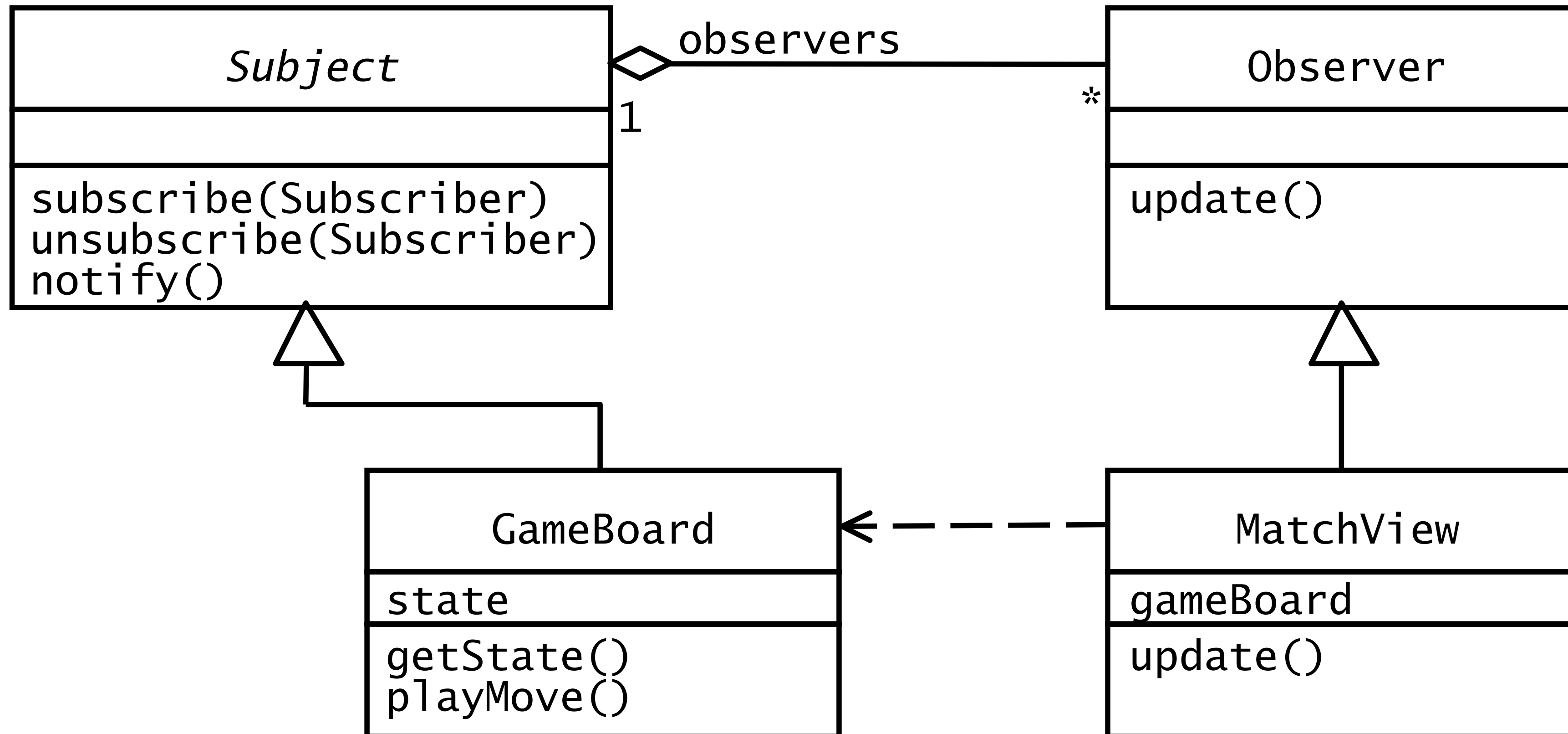
Applying the Observer Pattern to maintain Consistency across Views



Applying the Observer Design Pattern to maintain Consistency across MatchViews



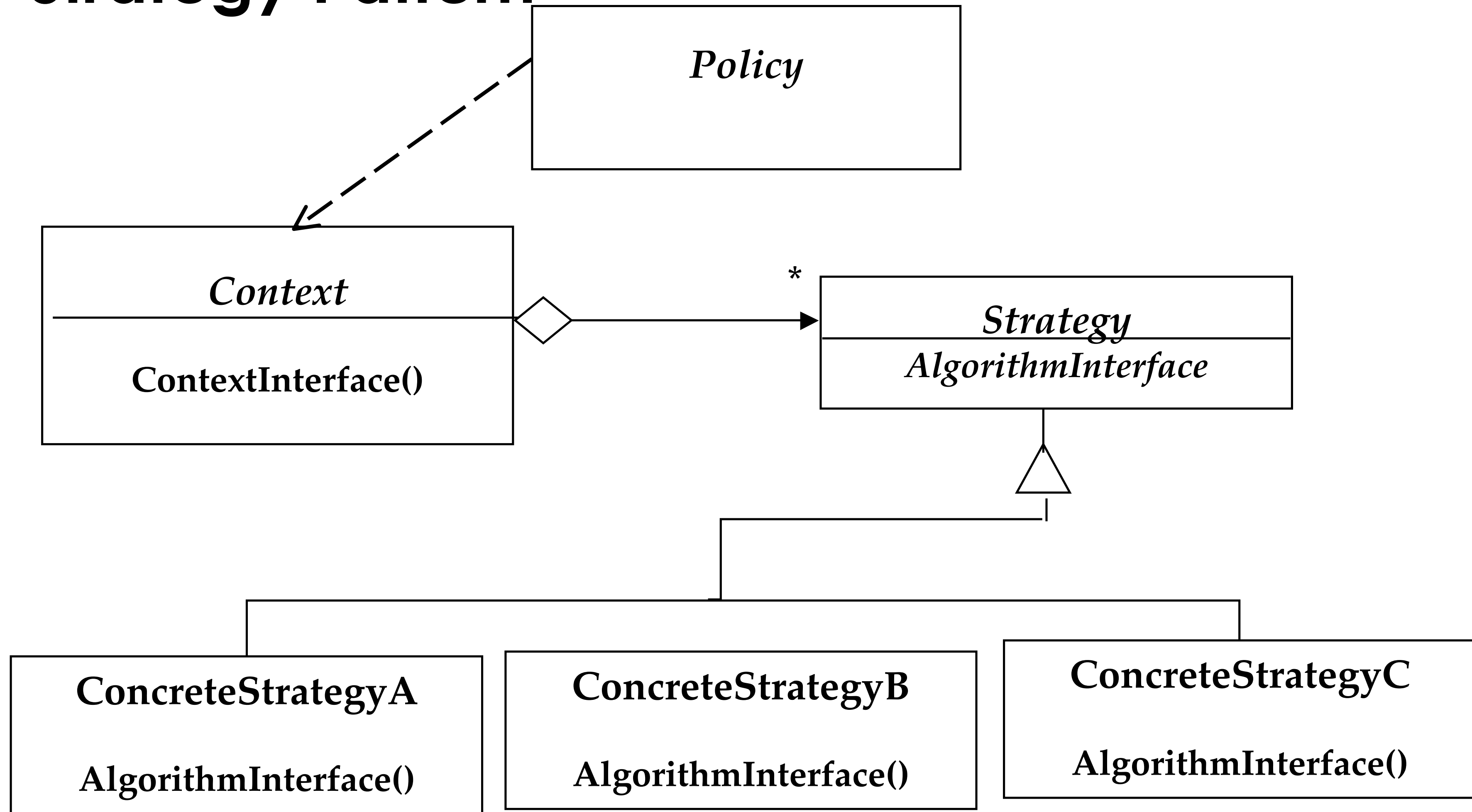
Applying the Observer Design Pattern to maintain Consistency across MatchViews



Motivation for the Strategy Pattern

- Different algorithms exists for a specific task
- Examples of tasks:
  - Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
  - Sorting a list of customers (Bubble sort, mergesort, quicksort)
- 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 other algorithms.

Strategy Pattern

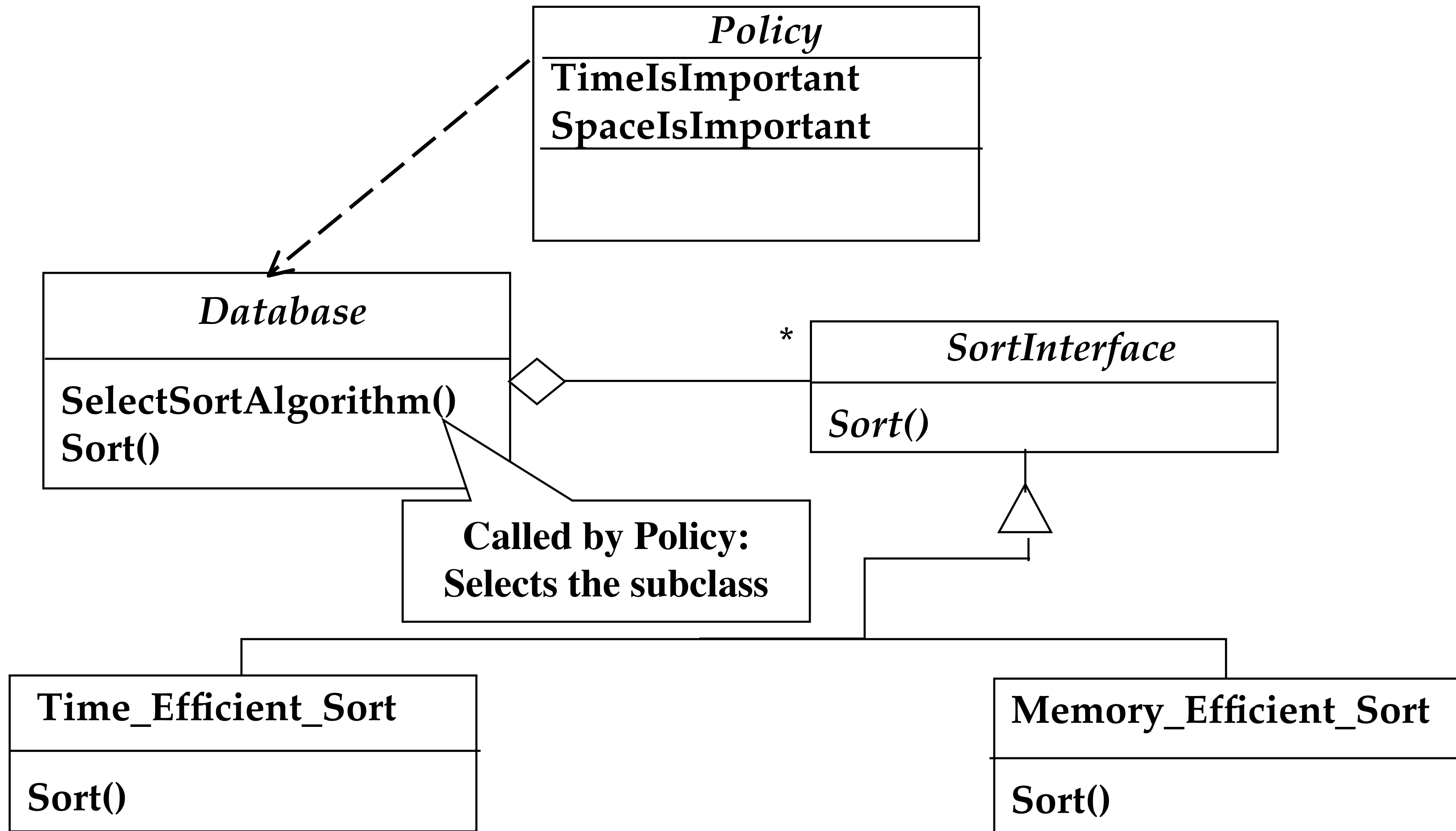


Policy decides which ConcreteStrategy is best in the current Context.

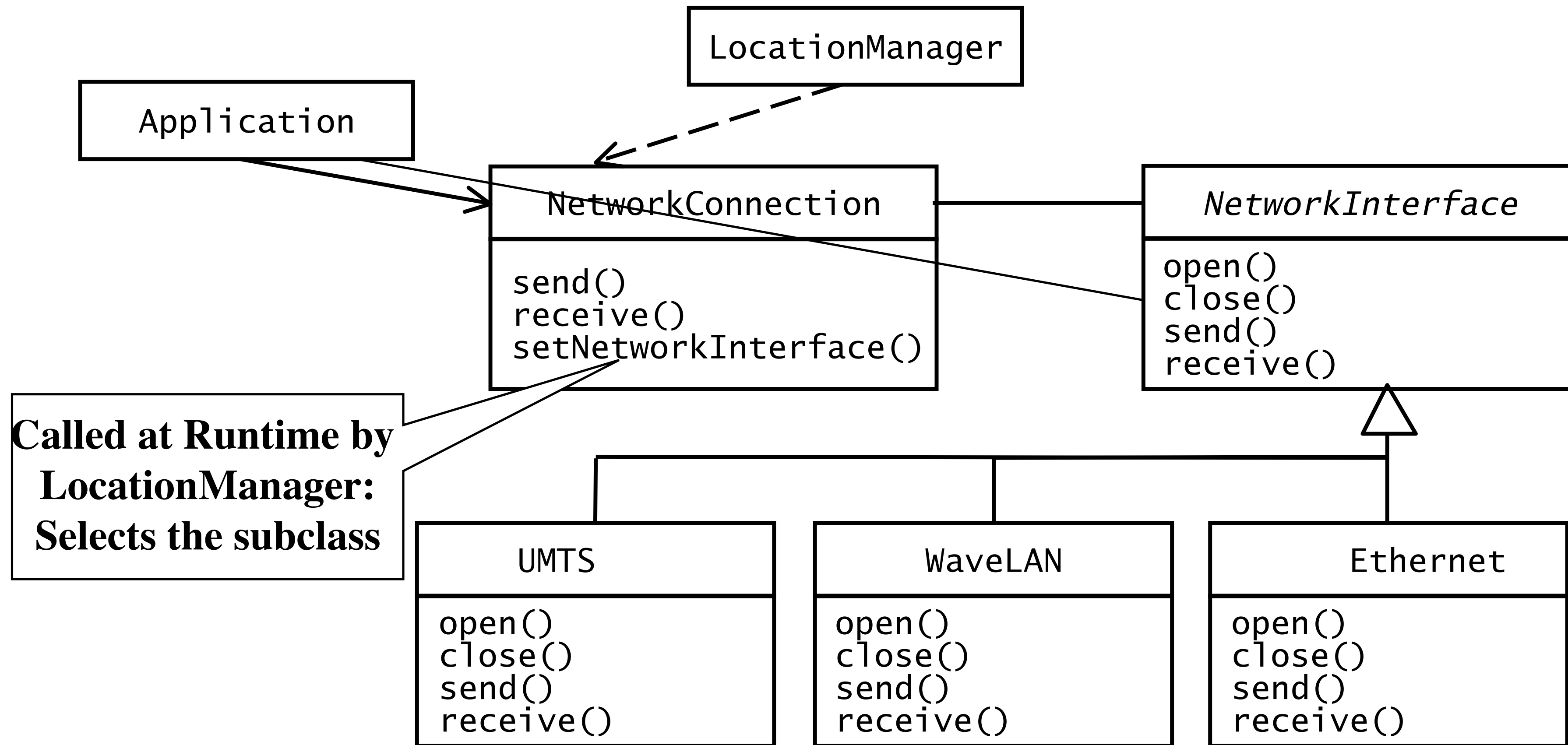
Applicability of Strategy Pattern

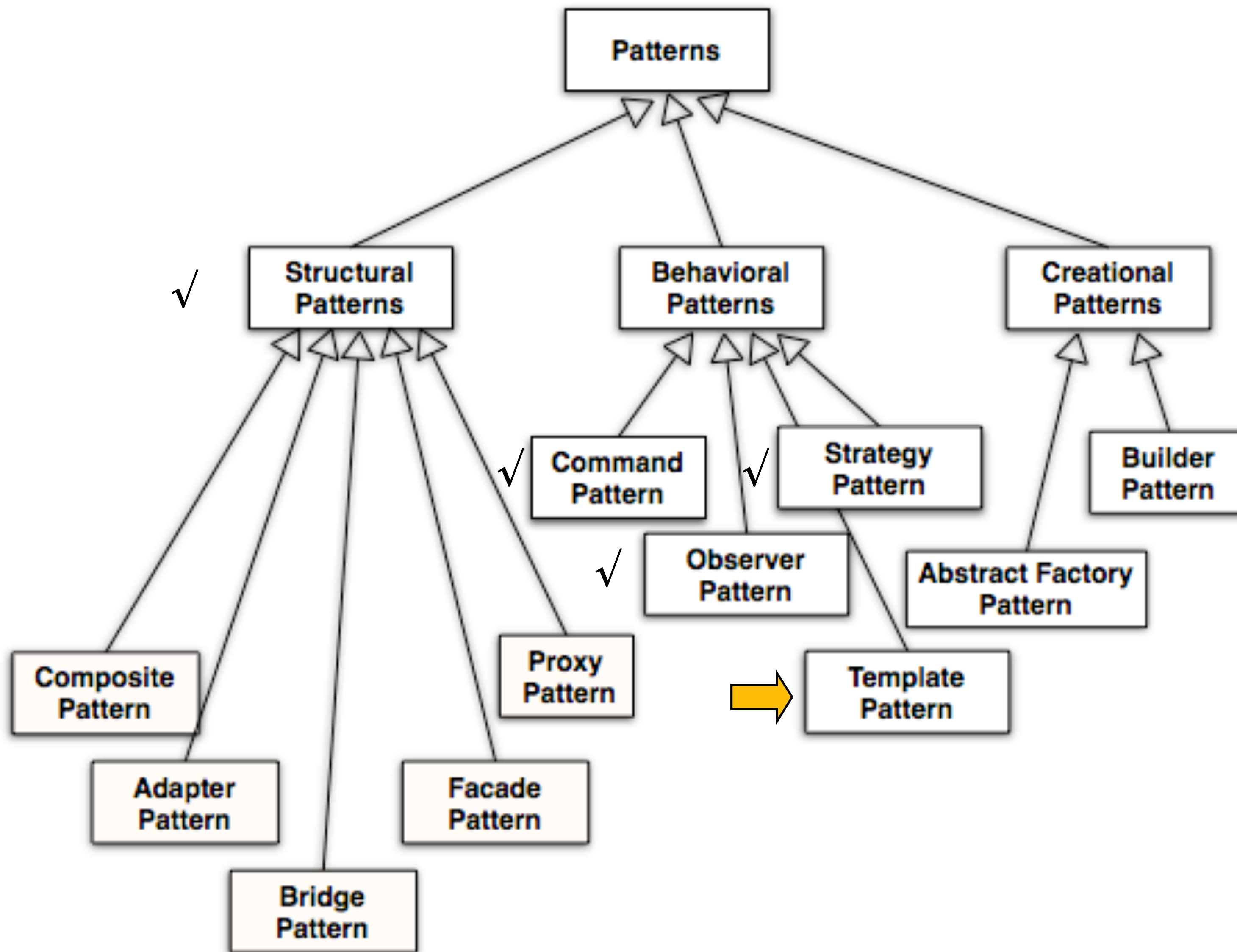
- Many related classes differ only in their behavior
- Different variants of an algorithm are needed that trade-off space against time
- A specific implementation needs to be selected based on the current context.

Using a Strategy Pattern to Decide between Algorithms at Runtime



Supporting Multiple implementations of a Network Interface



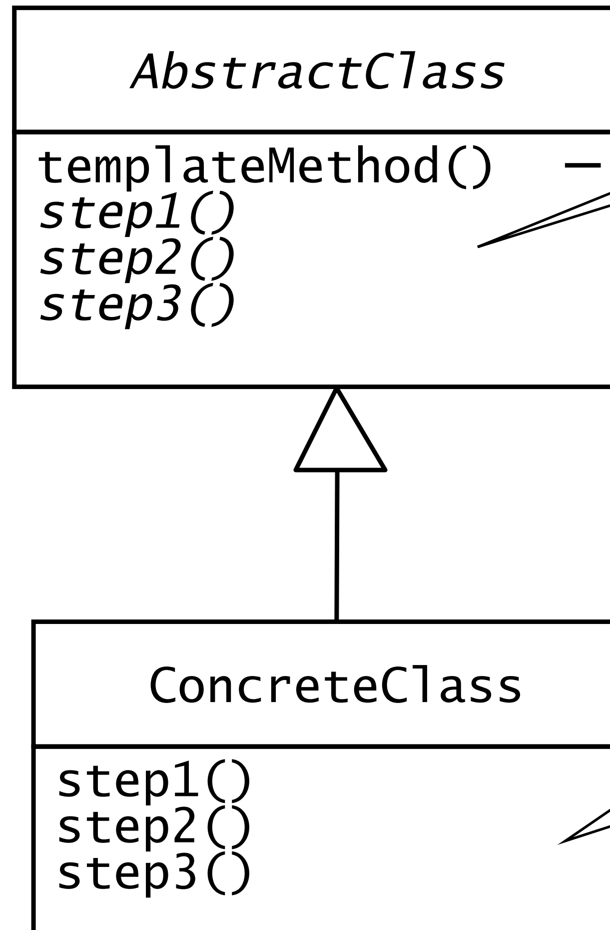

Patterns
Structural Patterns
Behavioral Patterns
Creational Patterns
Command Pattern
Strategy Pattern
Builder Pattern
Observer Pattern
Abstract Factory Pattern
Composite Pattern
Proxy Pattern
Template Pattern
Adapter Pattern
Facade Pattern
Bridge Pattern

Template Method Motivation

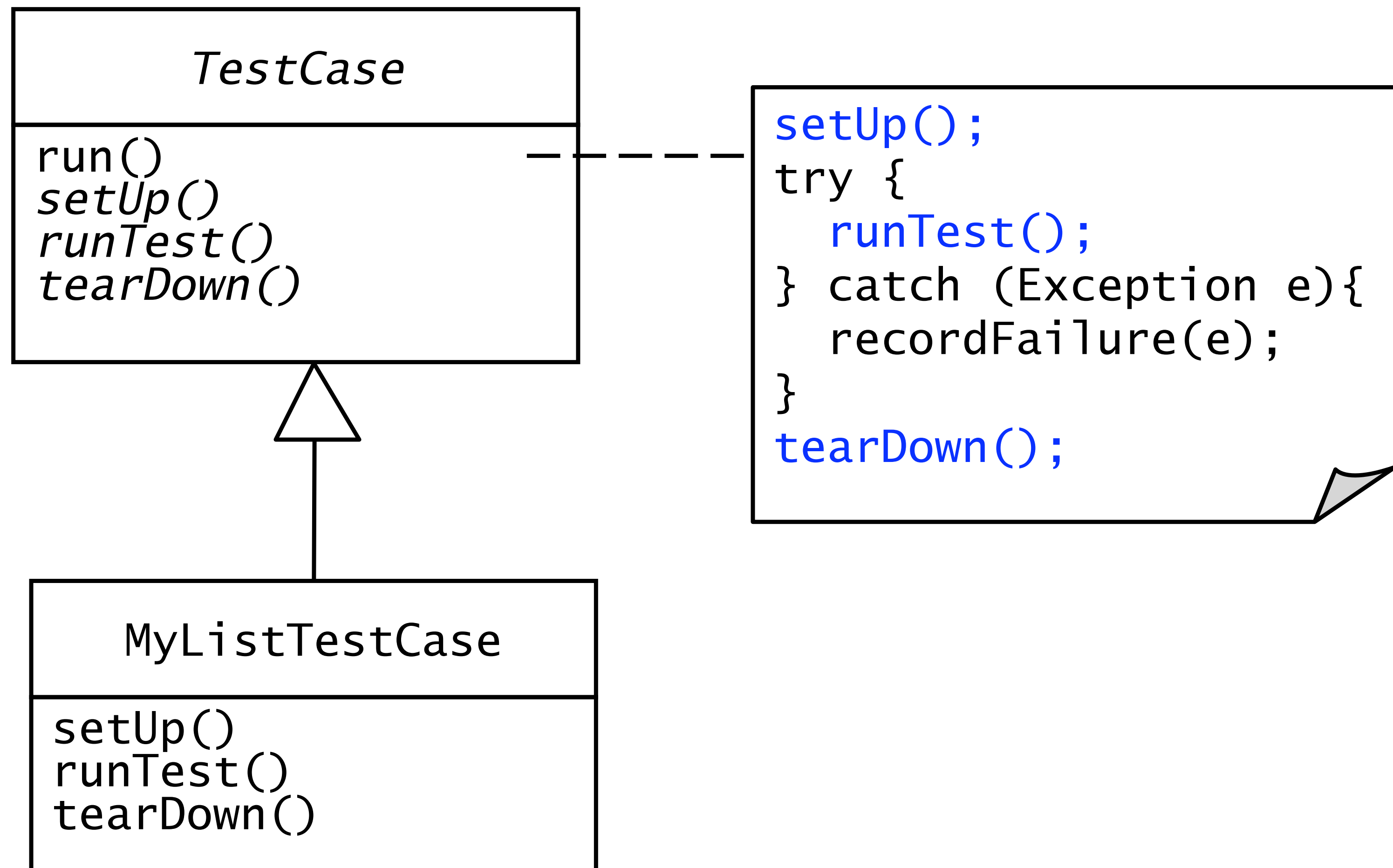
- Several subclasses share the same algorithm but differ in some aspects
- Examples:
  - Opening documents of different types consists of the same sequence of operations with different realizations
    - open; {read|write}*; close;
  - Executing a set of different test cases
    - startup; run_test; finish_test
- Approach
  - The common steps of the algorithm are factored out into an abstract class
    - Abstract methods are defined for each step
    - Subclasses provide the different realizations for each of the steps.

```
step1();
   …
step2();
   …
step3();
```

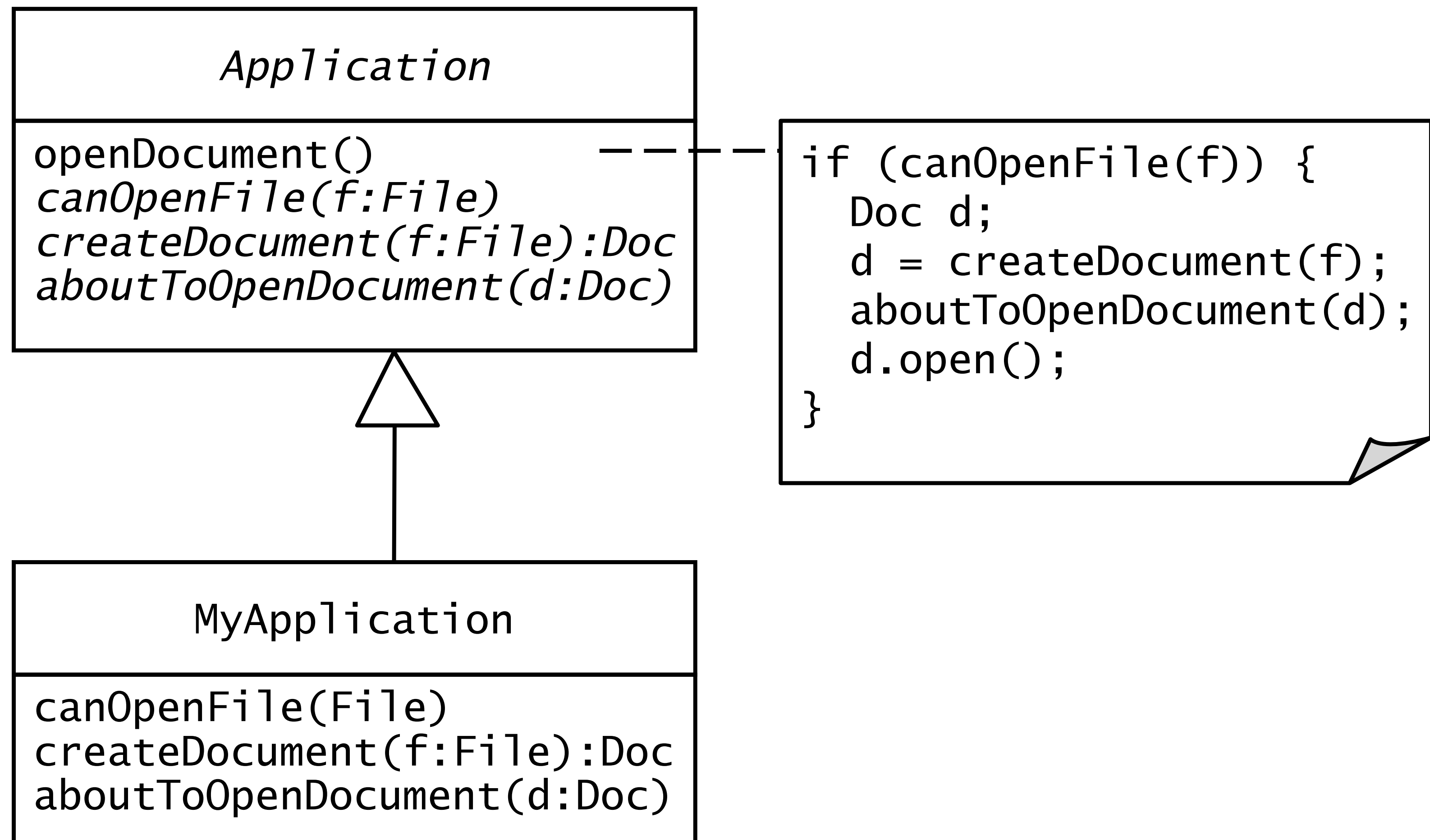
Template Method



Template Method Example: Test Cases

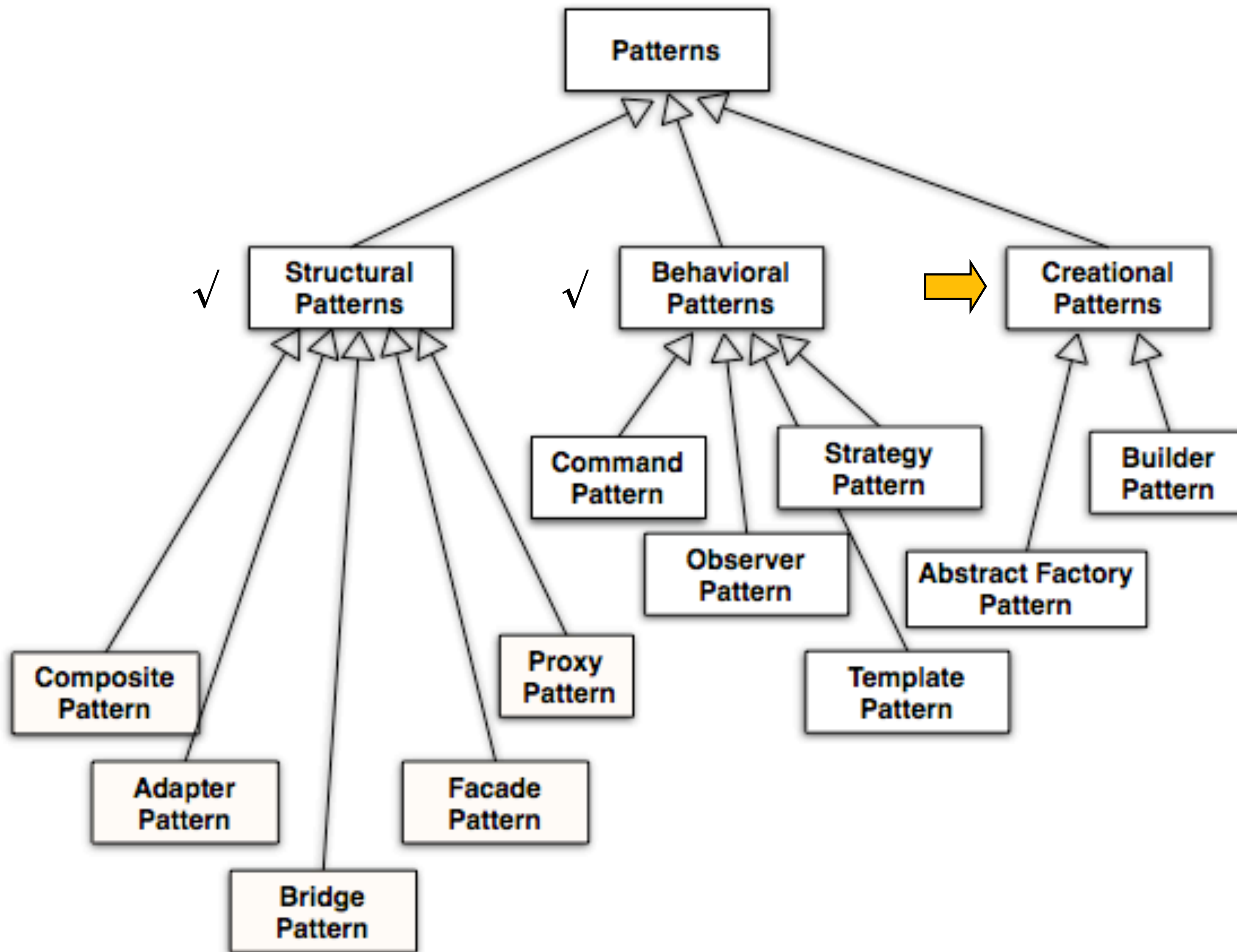


Template Method Example: Opening Documents



Template Method Pattern Applicability

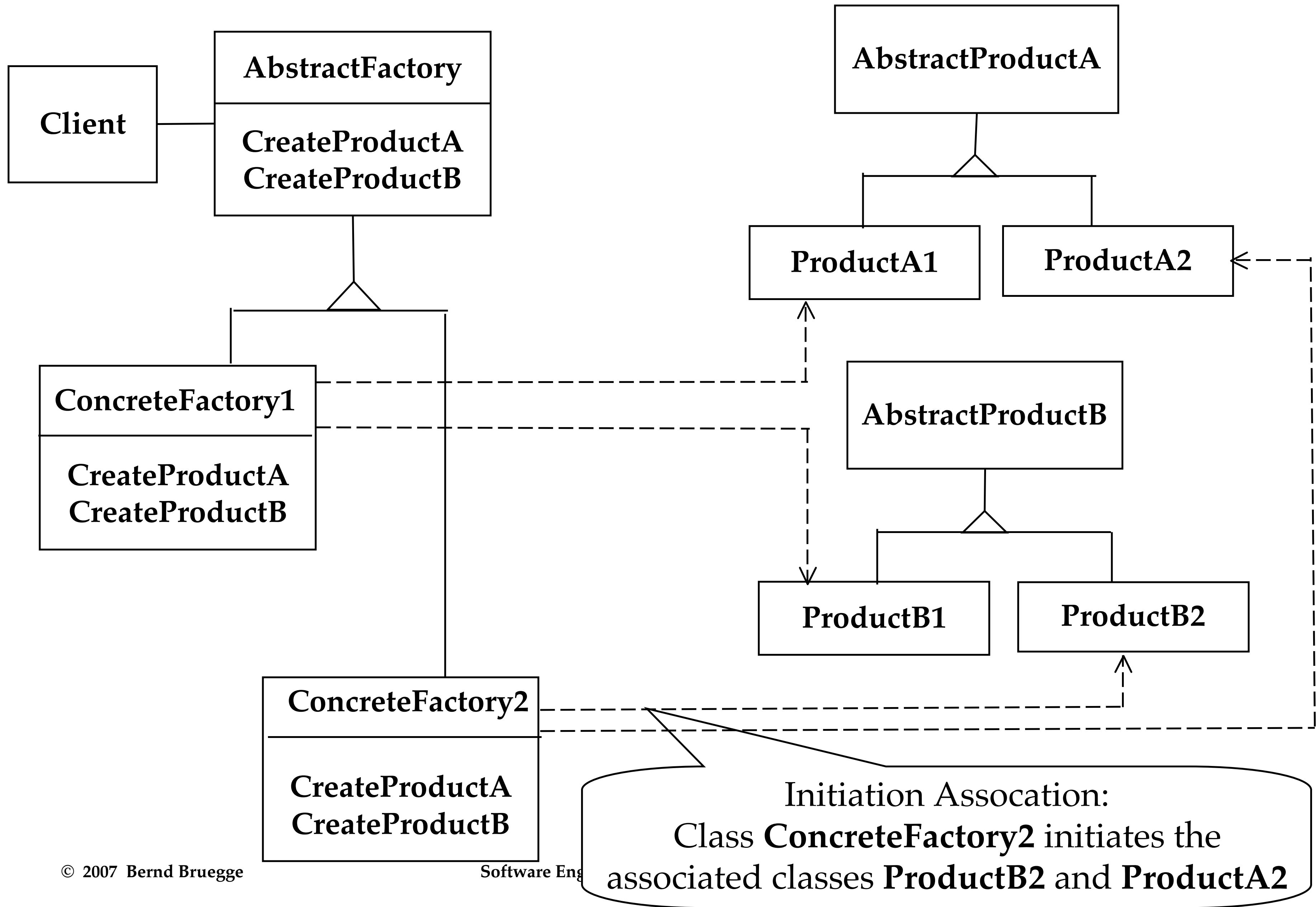
- Template method pattern uses inheritance to vary part of an algorithm
- Strategy pattern uses delegation to vary the entire algorithm
- Template Method is used in frameworks
  - The framework implements the invariants of the algorithm
  - The client customizations provide specialized steps for the algorithm
- Principle: "Don't call us, we'll call you"


Patterns
Structural Patterns
Behavioral Patterns
Creational Patterns
Command Pattern
Strategy Pattern
Builder Pattern
Observer Pattern
Abstract Factory Pattern
Composite Pattern
Proxy Pattern
Template Pattern
Adapter Pattern
Facade Pattern
Bridge Pattern

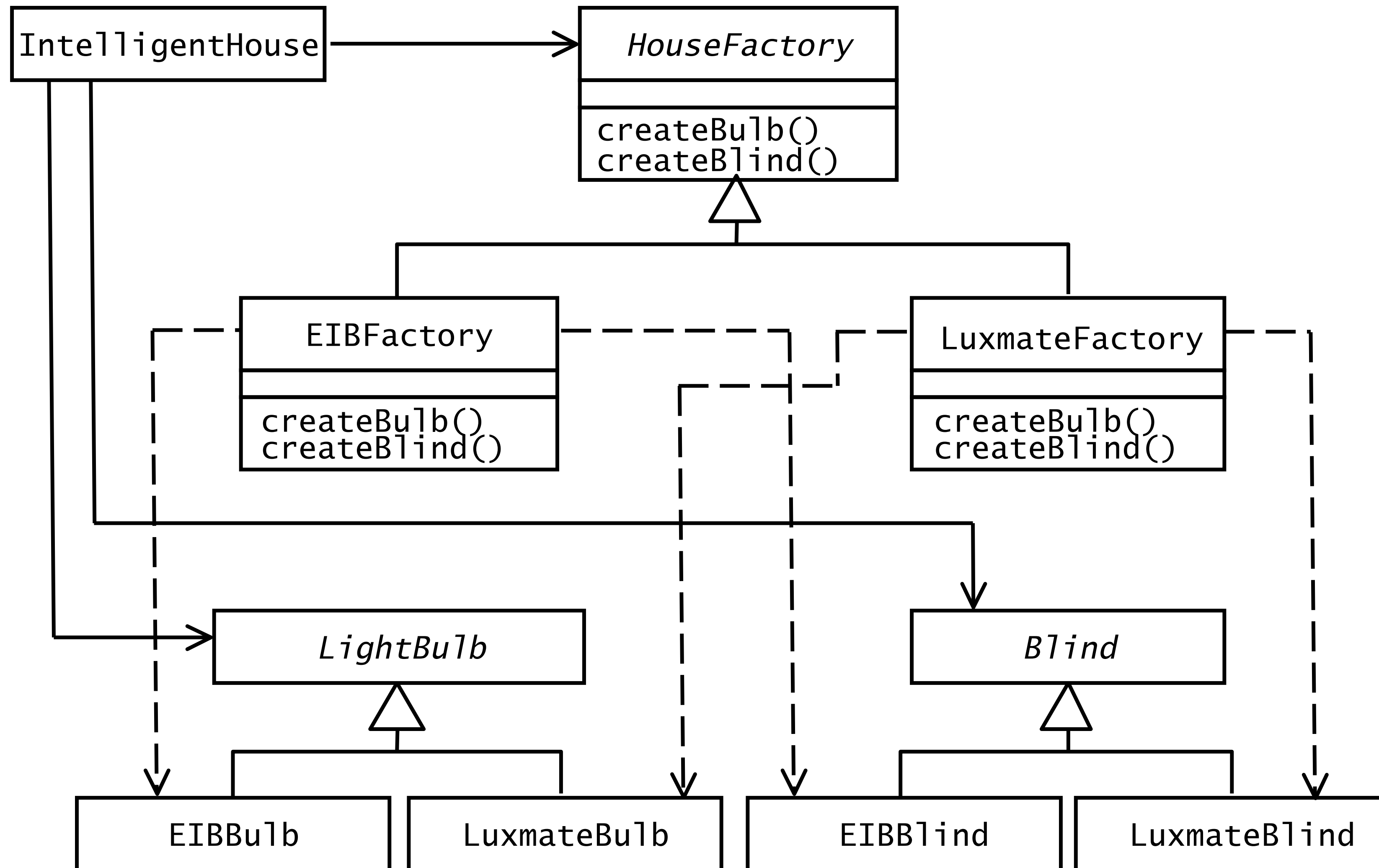
Abstract Factory Pattern Motivation

- Consider a user interface toolkit that supports multiple looks and feel standards for different operating systems:
  - 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:
  - How can you write a single control system that is independent from the manufacturer?

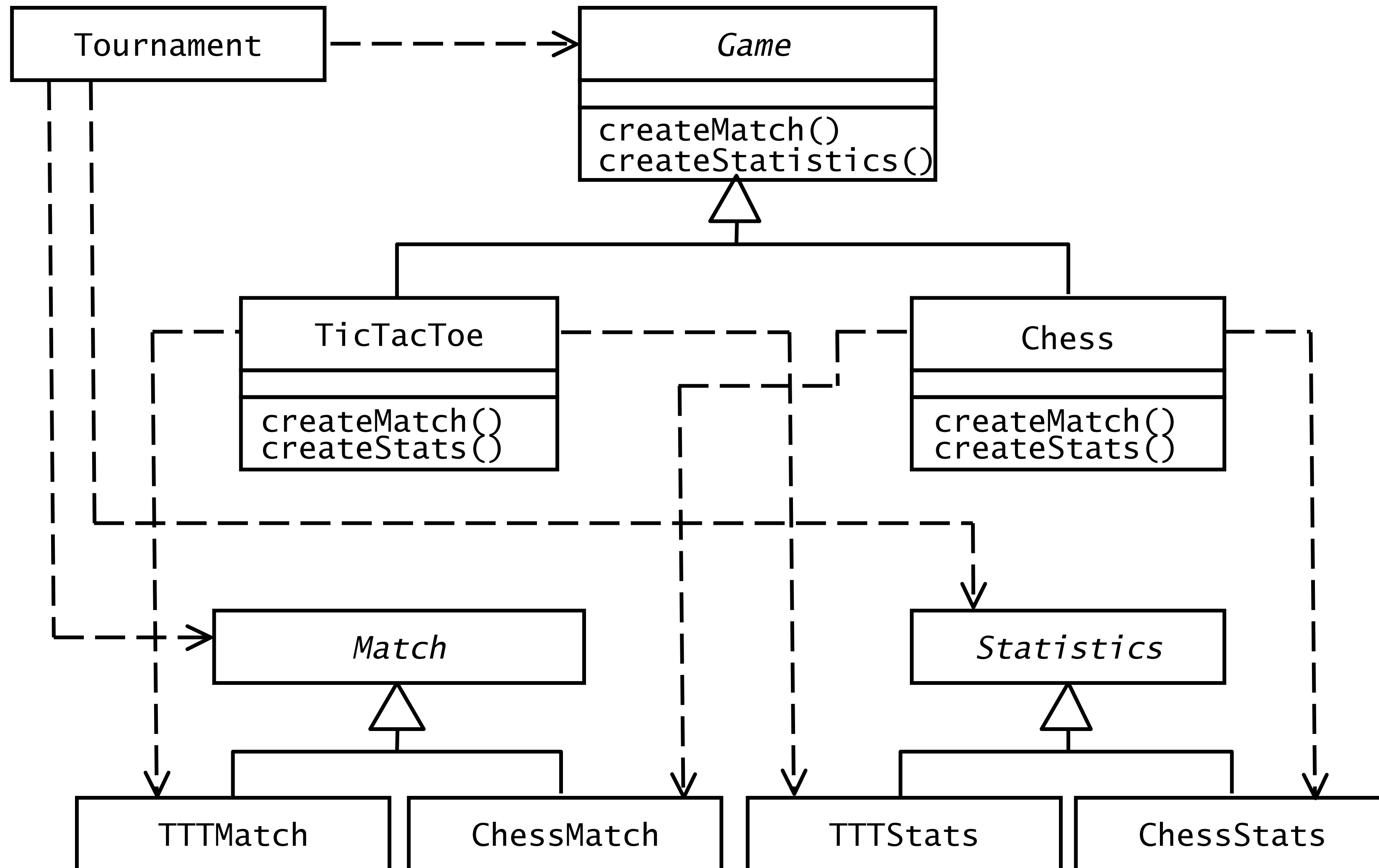
Abstract Factory



Example: A Facility Management System for a House



Applying the Abstract Factory Pattern to Games



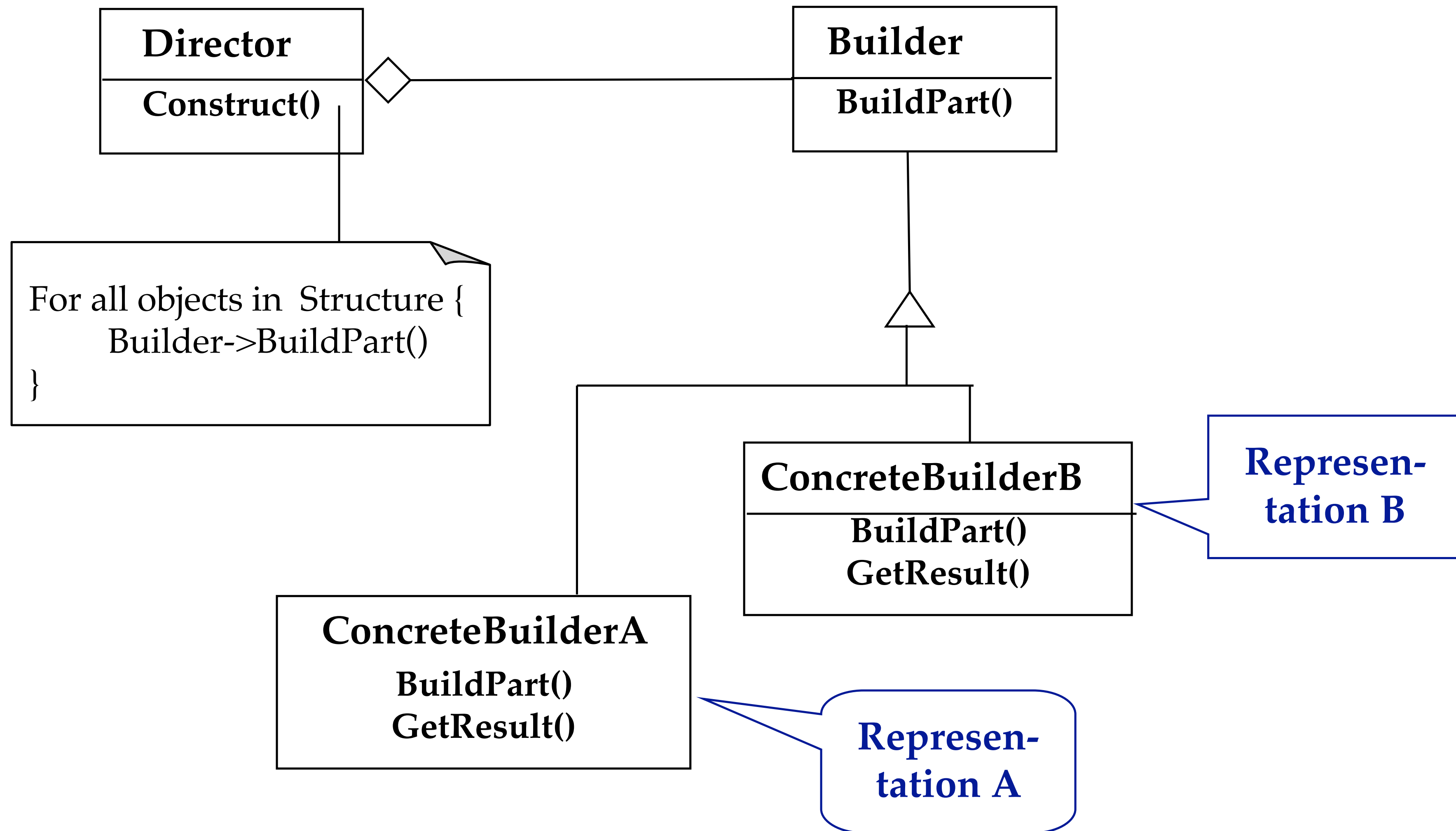
Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation
- Manufacturer Independence
- Constraints on related products
- Cope with upcoming change

Builder Pattern Motivation

- The construction of a complex object is common across several representations
- Example
  - Converting a document to a number of different formats
    - the steps for writing out a document are the same
    - the specifics of each step depend on the format
- Approach
  - The construction algorithm is specified by a single class (the "director")
  - The abstract steps of the algorithm (one for each part) are specified by an interface (the "builder")
  - Each representation provides a concrete implementation of the interface (the "concrete builders")

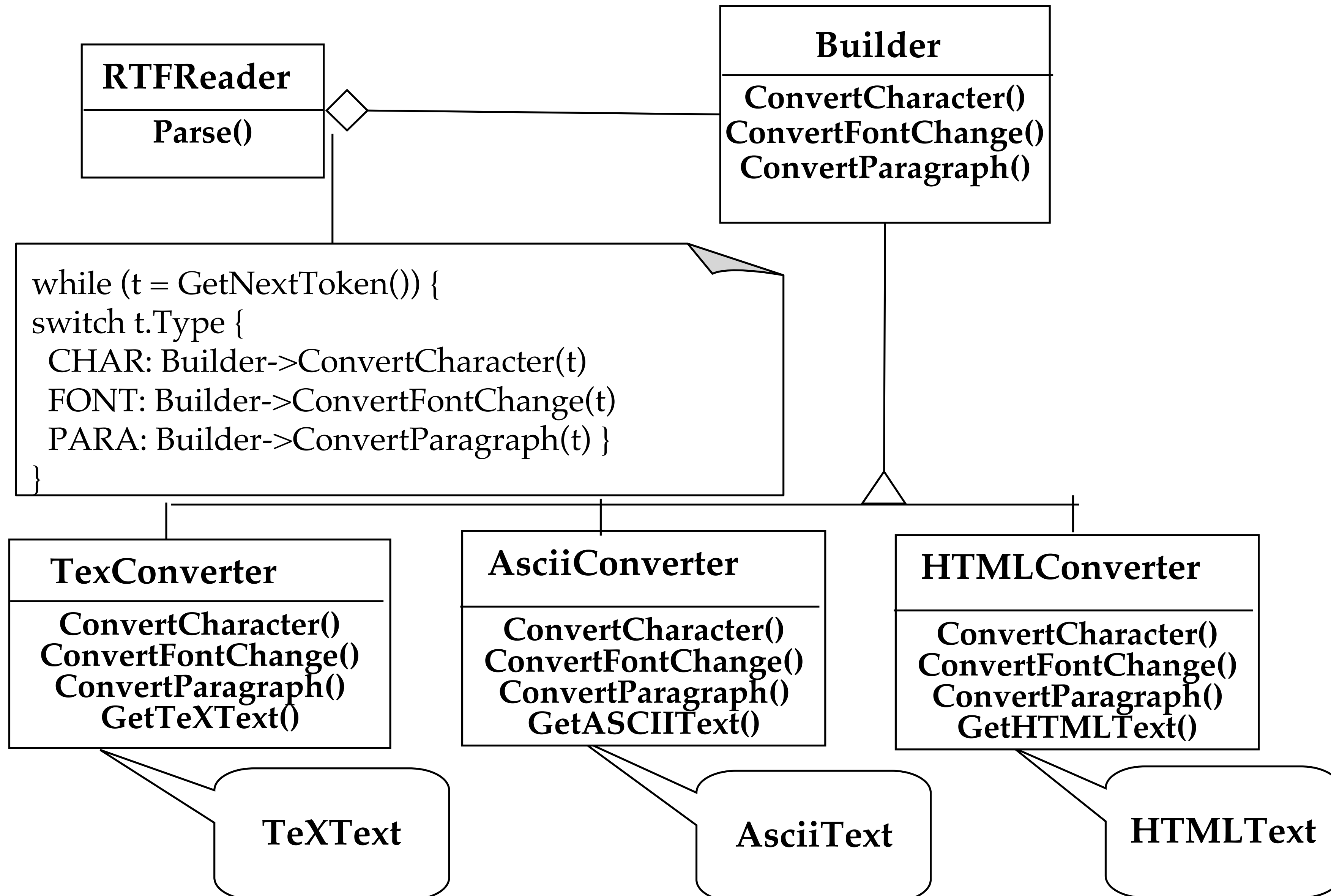
Builder Pattern



Applicability of Builder Pattern

- The creation of a complex product must be independent of the particular parts that make up the product
- The creation process must allow different representations for the object that is constructed.

Example: Converting an RTF Document into different representations



Comparison: Abstract Factory vs Builder

- Abstract Factory
  - Focuses on product family
  - Does not hide the creation process
- 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
  - Hides the creation process from the user
- Abstract Factory and Builder work well together for a family of multiple complex products

Clues in Nonfunctional Requirements for the Use of Design Patterns

- *Text:* “manufacturer independent”,
  “device independent”,
  “must support a family of products”

  => Abstract Factory Pattern
- *Text:* “must interface with an existing object”

  => Adapter Pattern
- *Text:* “must interface to several systems, some of them to be developed in the future”,
  “ an early prototype must be demonstrated”

  =>Bridge Pattern
- *Text:* “must interface to existing set of objects”

  => Façade Pattern

Clues in Nonfunctional Requirements for use of Design Patterns (2)

- *Text:* “complex structure”,
  “must have variable depth and width”

  => Composite Pattern
- *Text:* “must be location transparent”

  => Proxy Pattern
- *Text:* “must be extensible”,
  “must be scalable”

  => Observer Pattern
- *Text:* “must provide a policy independent from the mechanism”

  => Strategy Pattern

Summary

- Composite, Adapter, Bridge, Façade, Proxy (Structural Patterns)
  - Focus: Composing objects to form larger structures
    - Realize new functionality from old functionality,
    - Provide flexibility and extensibility
- Command, Observer, Strategy, Template (Behavioral Patterns)
  - Focus: Algorithms and assignment of responsibilities to objects
    - Avoid tight coupling to a particular solution
- Abstract Factory, Builder (Creational Patterns)
  - Focus: Creation of complex objects
    - Hide how complex objects are created and put together

Conclusion