

# Chapter 8, Object Design: Design Patterns



# Design Pattern

## **A design pattern is:**

- a template solution to a recurring design problem

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

- an example of *modifiable* design

Learning to design starts by studying other designs

- reusable design knowledge
- 7+-2 classes and their associations (5+-2 classes)

# Design Patterns

**Design patterns** are template solutions that developers have refined over time to solve a range of recurring problems [Gamma et al., 1994]. A design pattern has four elements:

1. A ***name*** that uniquely identifies the pattern from other patterns.
2. A ***problem description*** that describes the situations in which the pattern can be used. Problems addressed by design patterns are usually the realization of modifiability and extensibility design goals and nonfunctional requirements.

# Design Patterns\_cont

**Design patterns** are template solutions that developers have refined over time to solve a range of recurring problems [Gamma et al., 1994]. A design pattern has four elements:

3. A ***solution*** stated as a set of collaborating classes and interfaces.
4. A set of ***consequences*** that describes the trade-offs and alternatives to be considered with respect to the design goals being addressed.

# Why reusable Designs?

## A design:

- enables flexibility to change (reusability)
- minimizes the introduction of new problems when fixing old ones (maintainability)
- allows the delivery of more functionality after an initial delivery (extensibility).

# Definitions:

## **Extensibility (Expandability)**

- 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

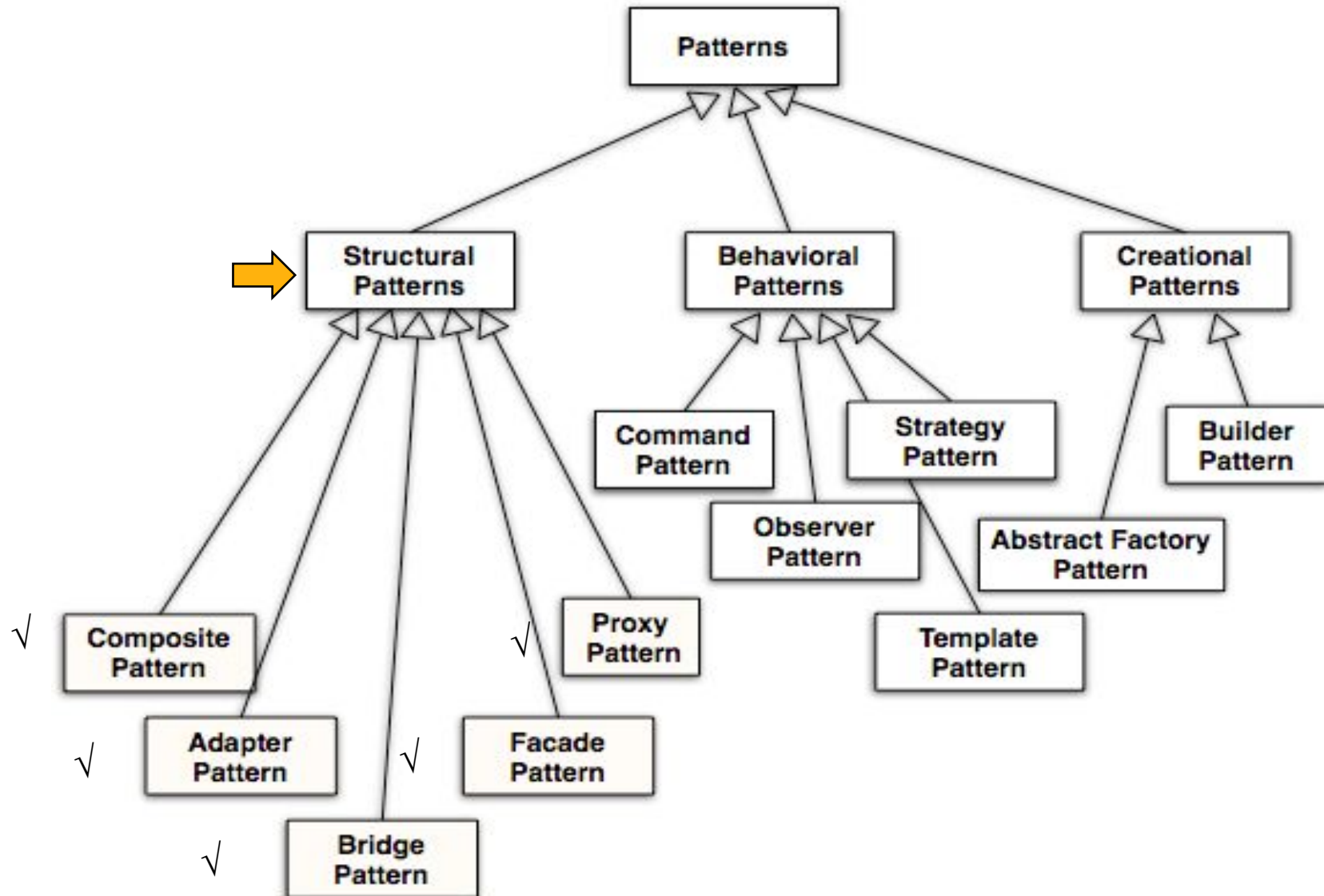
## **Scalability**

- A system is scalable, if existing components can easily be multiplied in the system

## **Reusability**

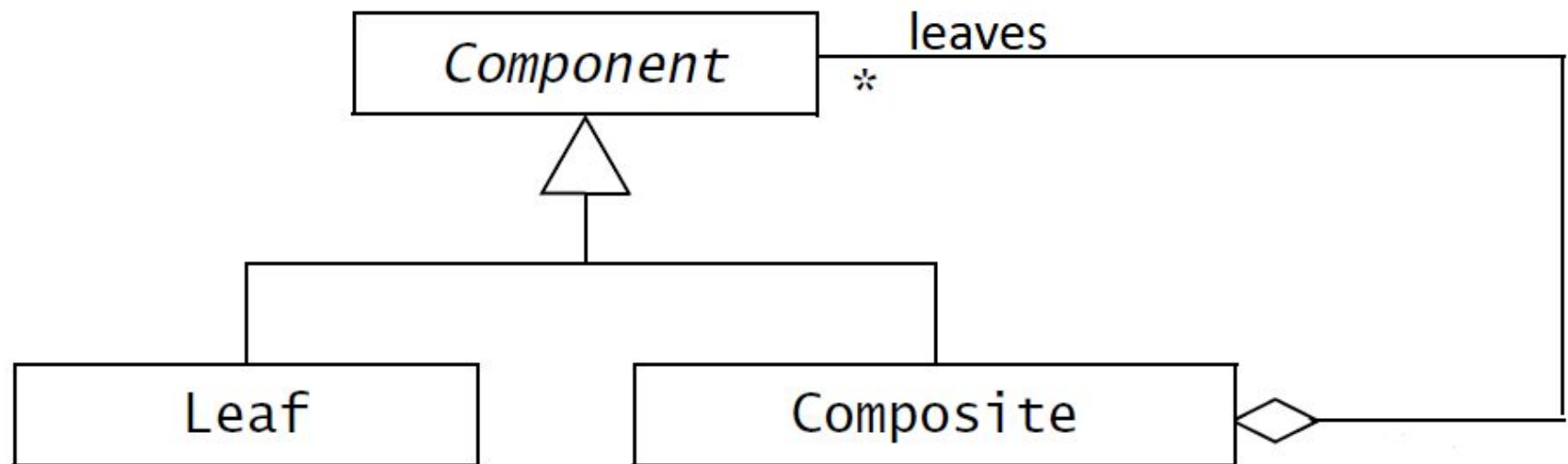
- A system is reusable, if it can be used by another system without requiring major changes in the existing system model (design reuse) or code base (code reuse).

# A Taxonomy of Design Patterns



# The composition pattern

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





# What is common between these definitions?

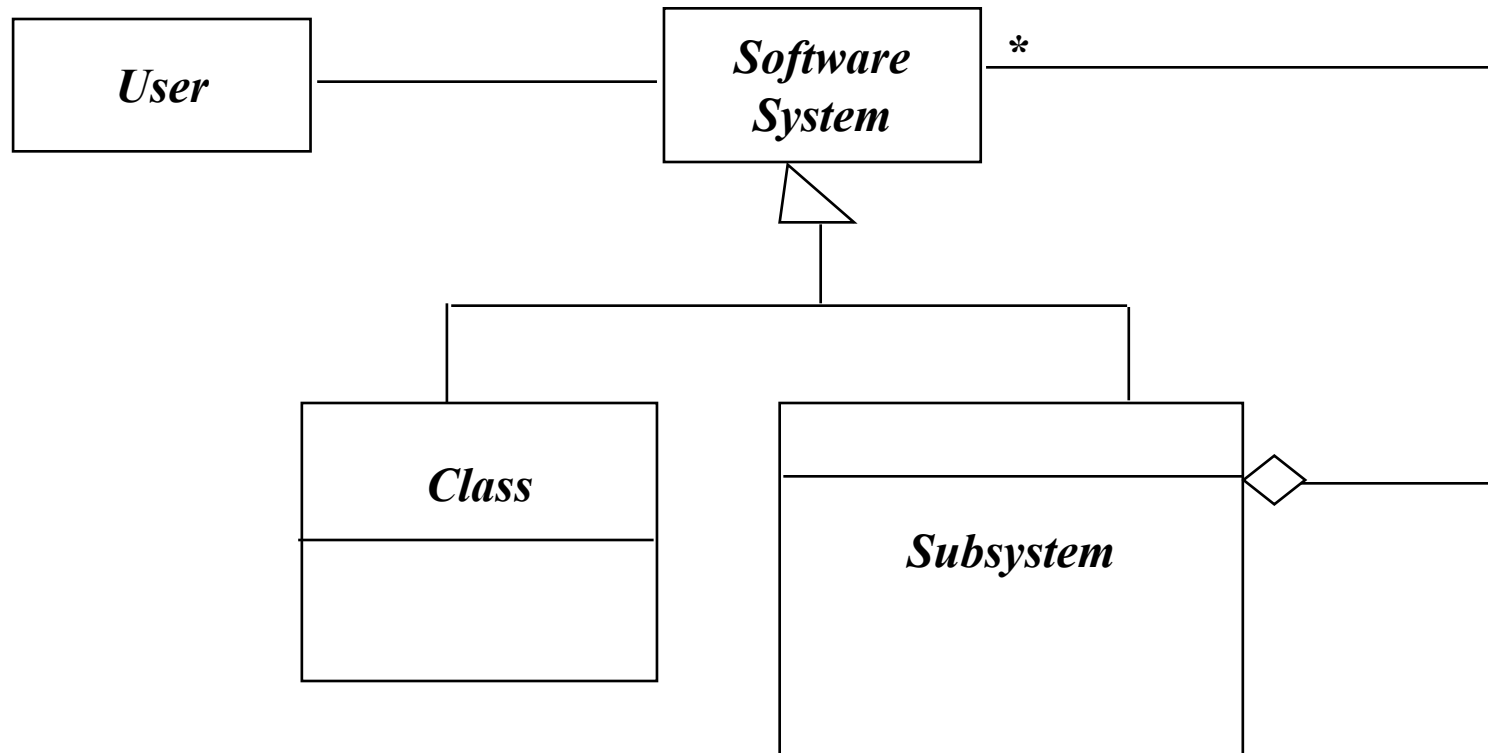
## Software System

- **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

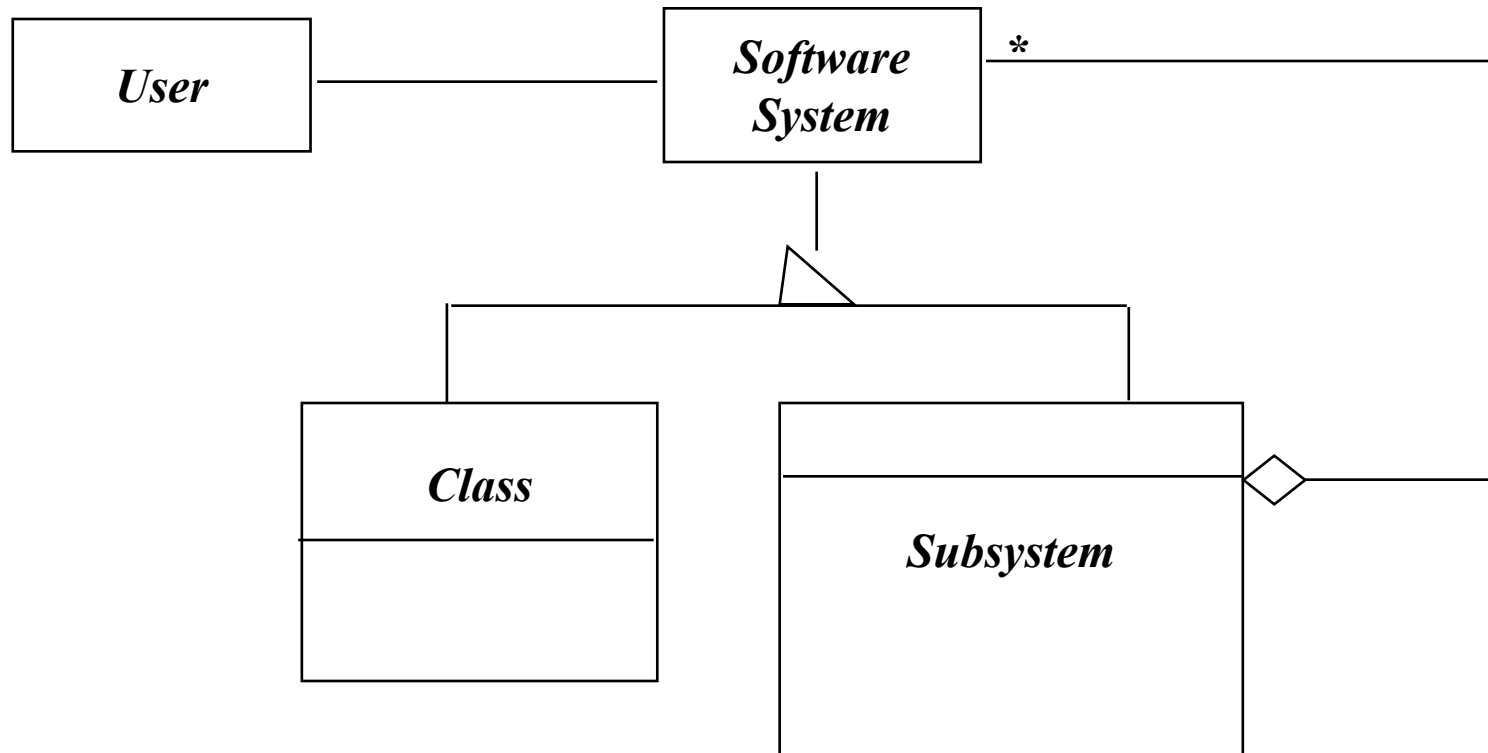
## Software Lifecycle

- **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.

# Modeling a Software System with a Composite Pattern

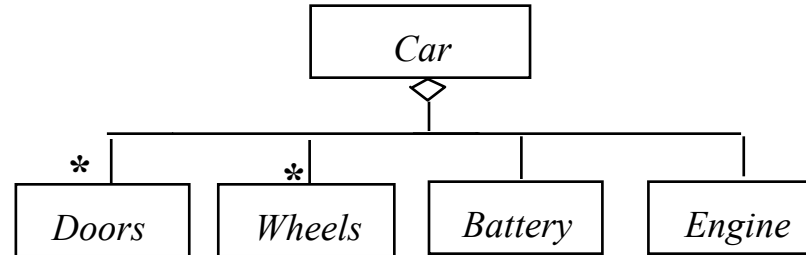


# Modeling a Software Lifecycle with a Composite Pattern

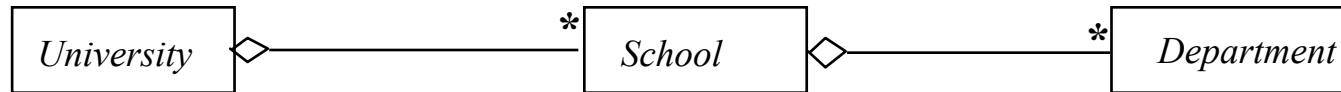


# The Composite Patterns models dynamic aggregates

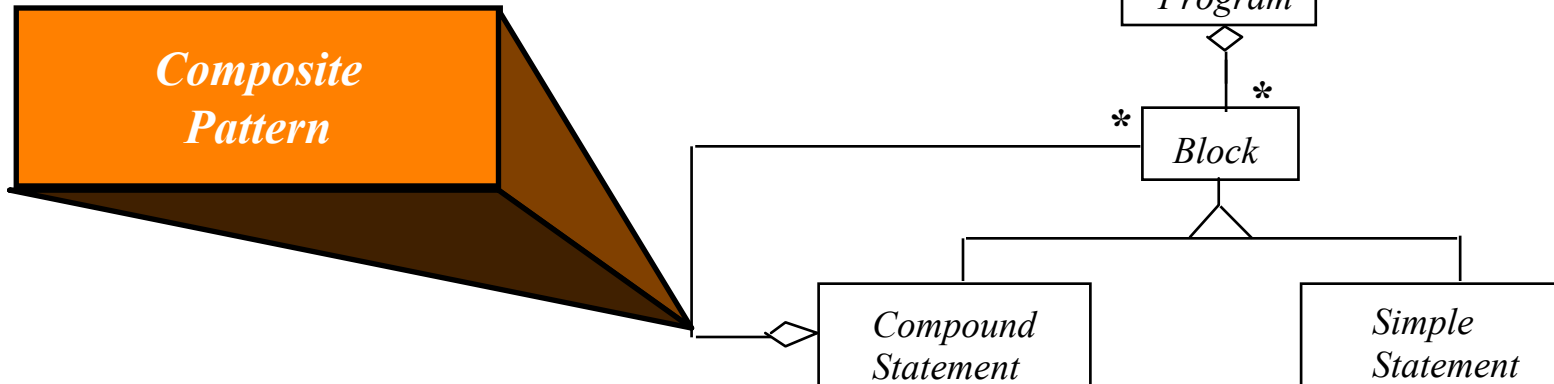
*Fixed Structure:*



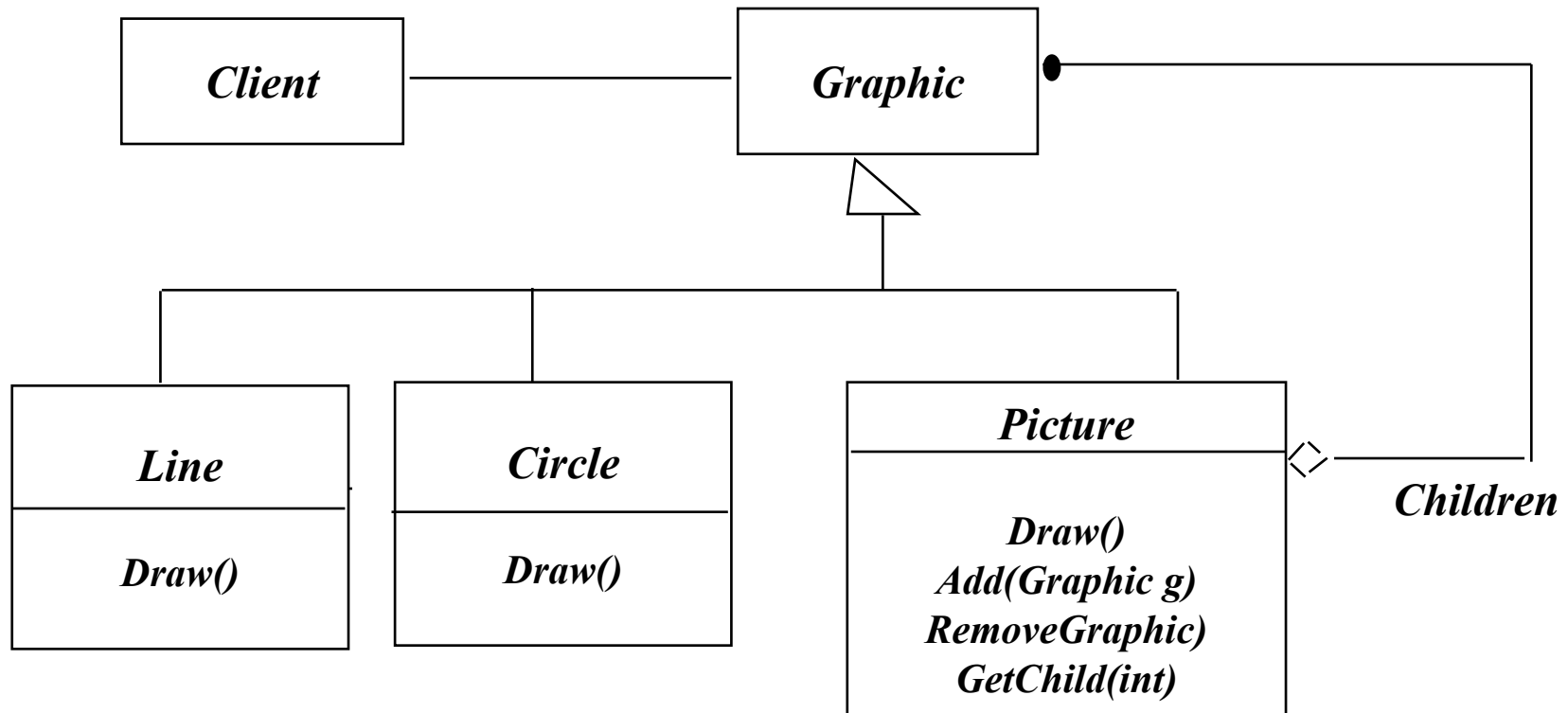
*Organization Chart (variable aggregate):*



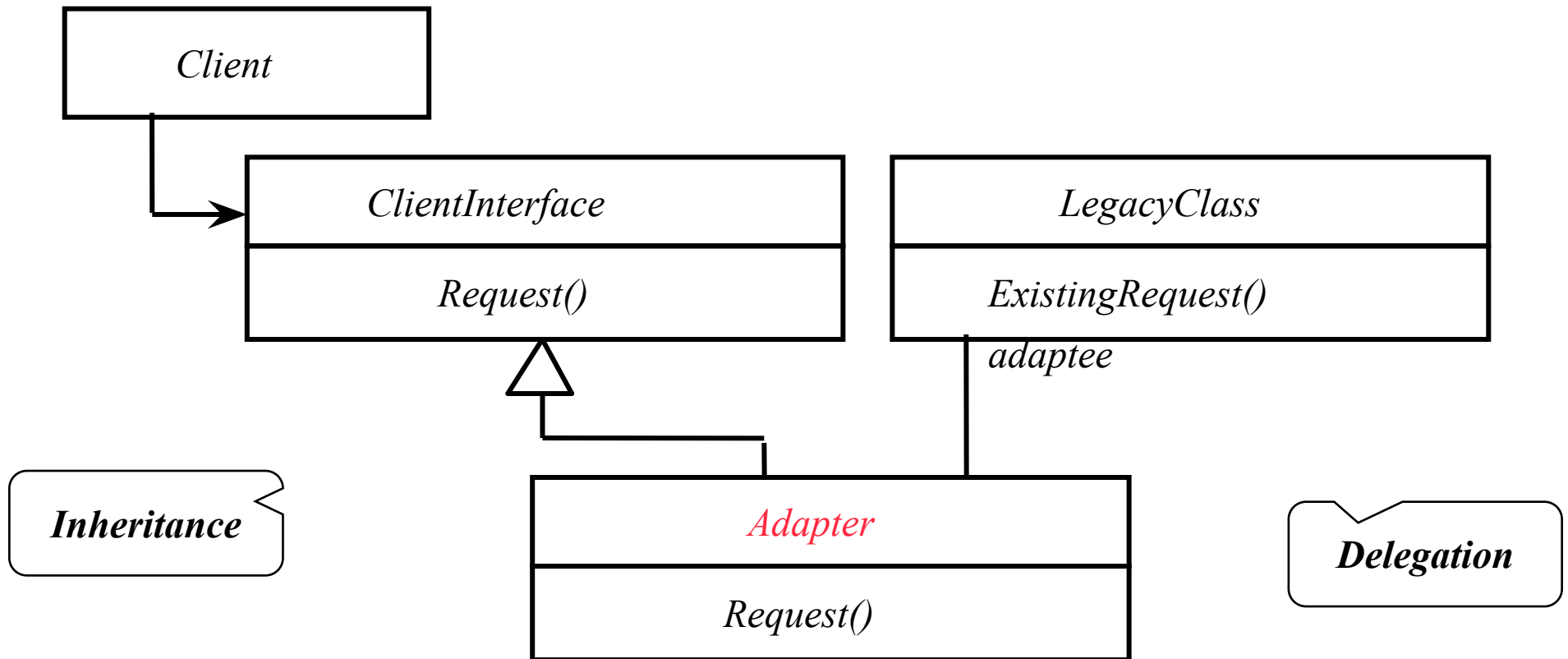
*Dynamic tree (recursive aggregate):*



# Graphic Applications also use Composite Patterns



# The Adapter Pattern



The adapter pattern uses inheritance as well as delegation:

- Interface inheritance is used to specify the interface of the Adapter class.
- Delegation is used to bind the Adapter and the LegacyClass(adaptee)

# The Adapter Pattern\_cont

The Adapter Pattern lets classes work together that couldn't otherwise because of incompatible interfaces

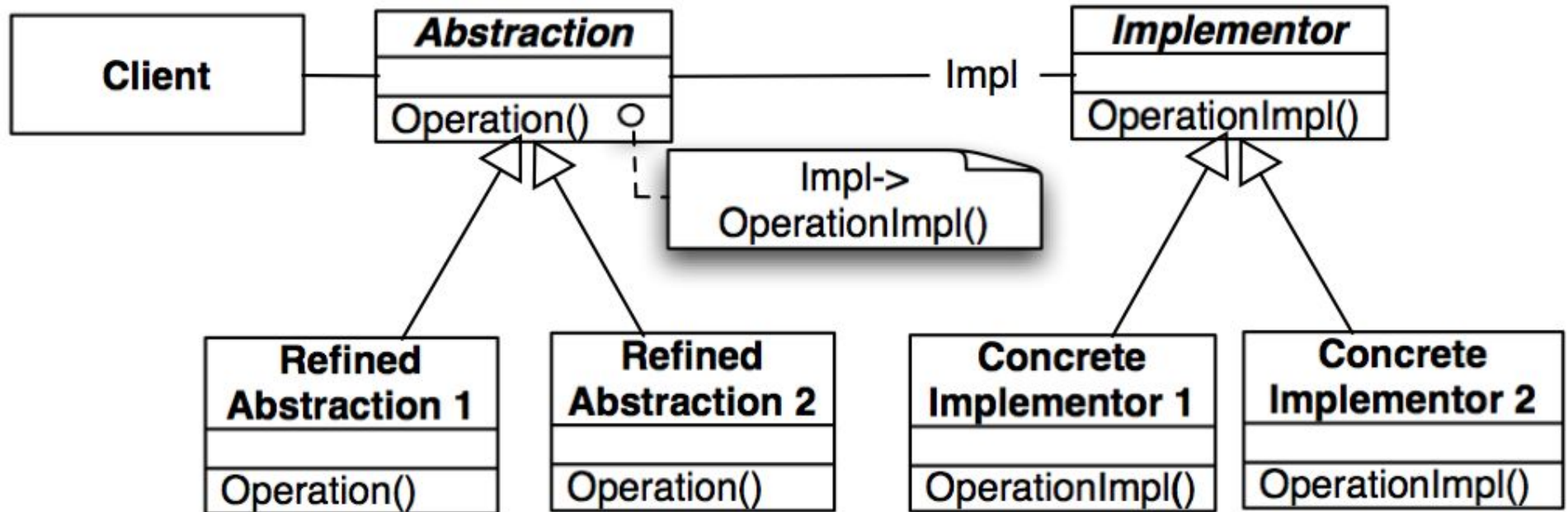
- “Convert the interface of a class into another interface expected by a client class.”
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Two Adapter Patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another. Between the Adapter and Adaptee classes there is an implementation inheritance relationship.
  - Object adapter:
    - Uses single inheritance and delegation
- Object adapters are much more frequent.
  - We cover only object adapters (and call them adapters).

# The 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.
- It provides a bridge between the Abstraction (in the application domain) and the Implementor (in the solution domain)



# The Bridge Pattern\_2



*Taxonomy in  
Application  
Domain*

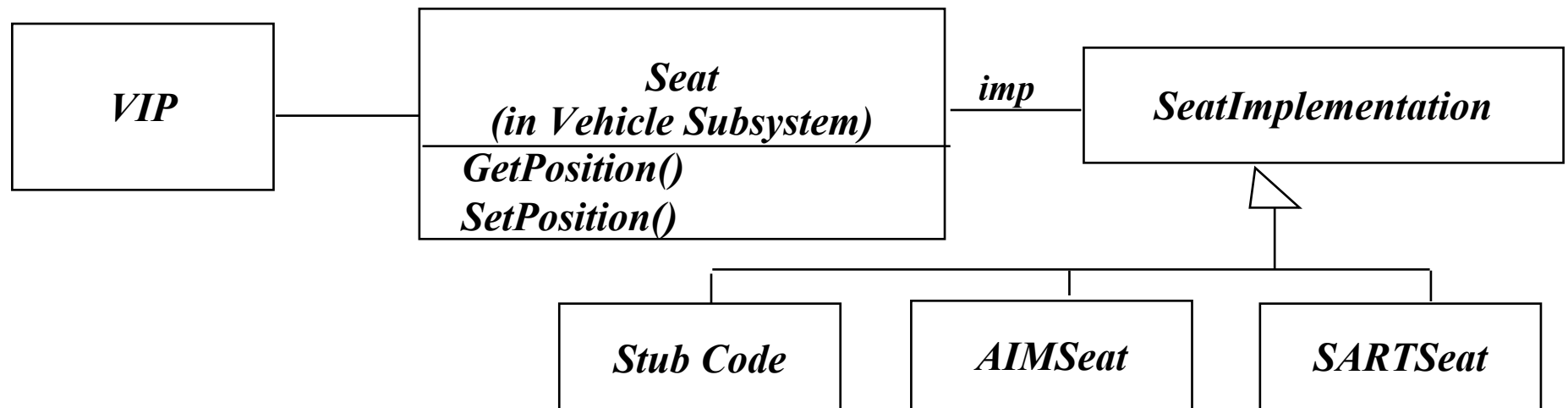
*Taxonomy in  
Solution Domain*

# Motivation for the Bridge Pattern

- Decouples an abstraction from its implementation so that the two can vary independently
- This allows to bind one from many different implementations of an interface to a client dynamically
- Design decision that can be realized any time during the runtime of the system
  - However, usually the binding occurs at start up time of the system (e.g. in the constructor of the interface class)

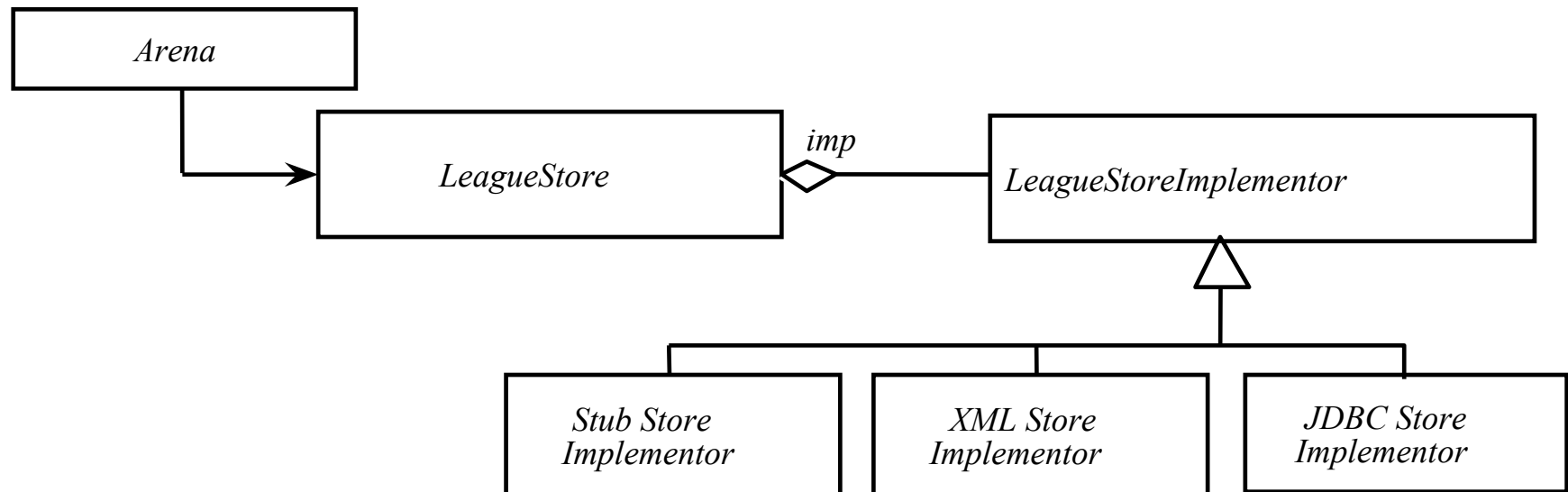
# Using a Bridge

- The bridge pattern can be used to provide multiple implementations under the same interface
- Interface to a component that is incomplete (only Stub Code is available), not yet known or unavailable during testing
- If seat data are required to be read, but the seat is not yet implemented (only stub code available), or only available by a simulation (AIM or SART), the bridge pattern can be used:



# Another use of the Bridge Pattern:

Support multiple Database Vendors



# Adapter vs Bridge

## Similarities:

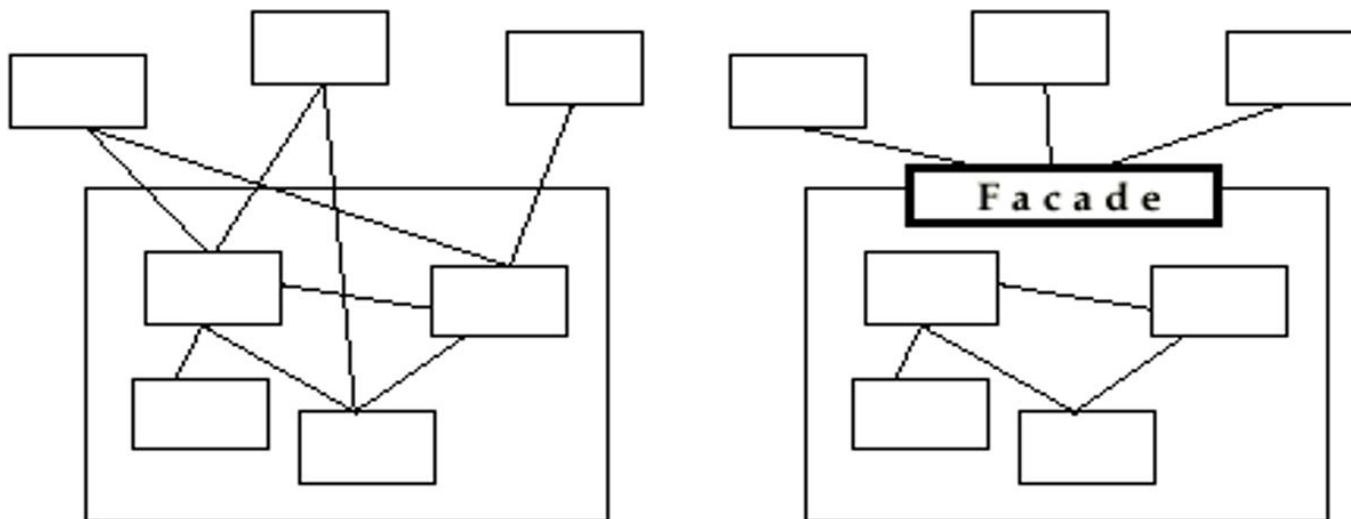
- Both are used to hide the details of the underlying implementation.

## Difference:

- The adapter pattern is geared towards making unrelated components work together:
  - Applied to systems after they're designed (reengineering, interface engineering).
  - "Inheritance followed by delegation"
  - 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.
    - "Delegation followed by inheritance"

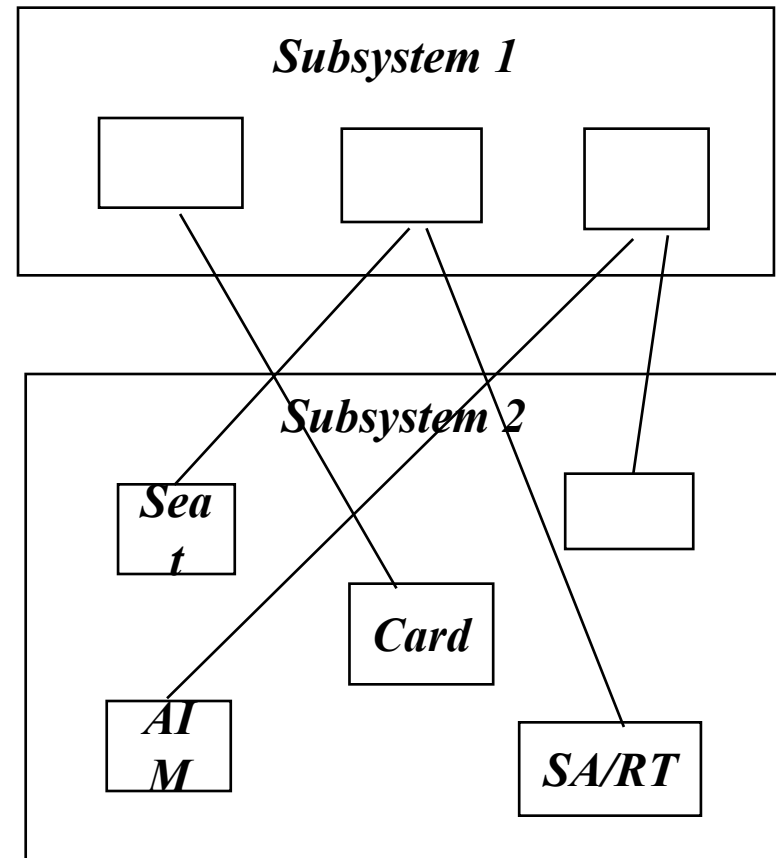
# The Façade Pattern

- 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



# Design Example

- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
  - This is “Ravioli Design”
  - **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



# Subsystem Design with Façade, Adapter, Bridge

- **The ideal structure of 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
- **We can use design patterns to realize this subsystem structure**
  - Realization of the Interface Object: **Façade**
    - 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!

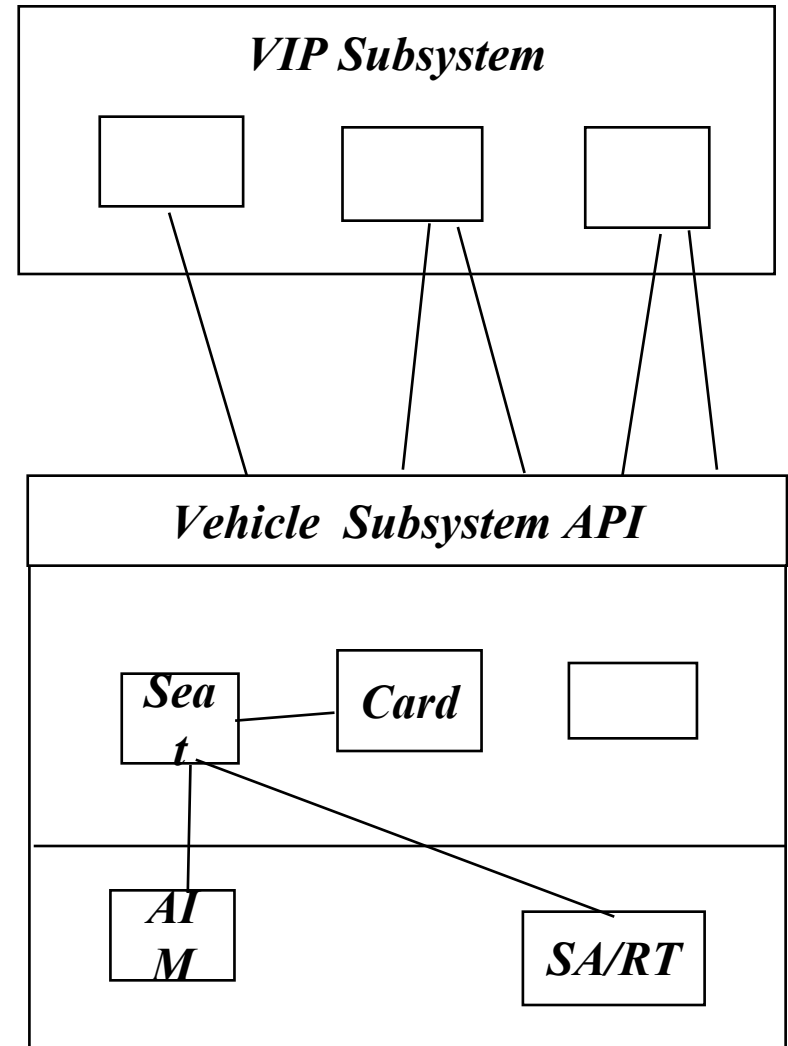


# When should you use these Design Patterns?

- A façade should be offered by all subsystems in a software system who a services
  - The façade delegates requests to the appropriate components within the subsystem. The façade usually does not have to be changed, when the components are changed
- The adapter design pattern should be used to interface to existing components
  - Example: A smart card software system should use an adapter for a smart card reader from a specific manufacturer
- The bridge design pattern should be used to interface to a set of objects
  - where the full set of objects is not completely known at analysis or design time.
  - when a subsystem or component must be replaced later after the system has been deployed and client programs use it in the field.

# Realizing an Opaque 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

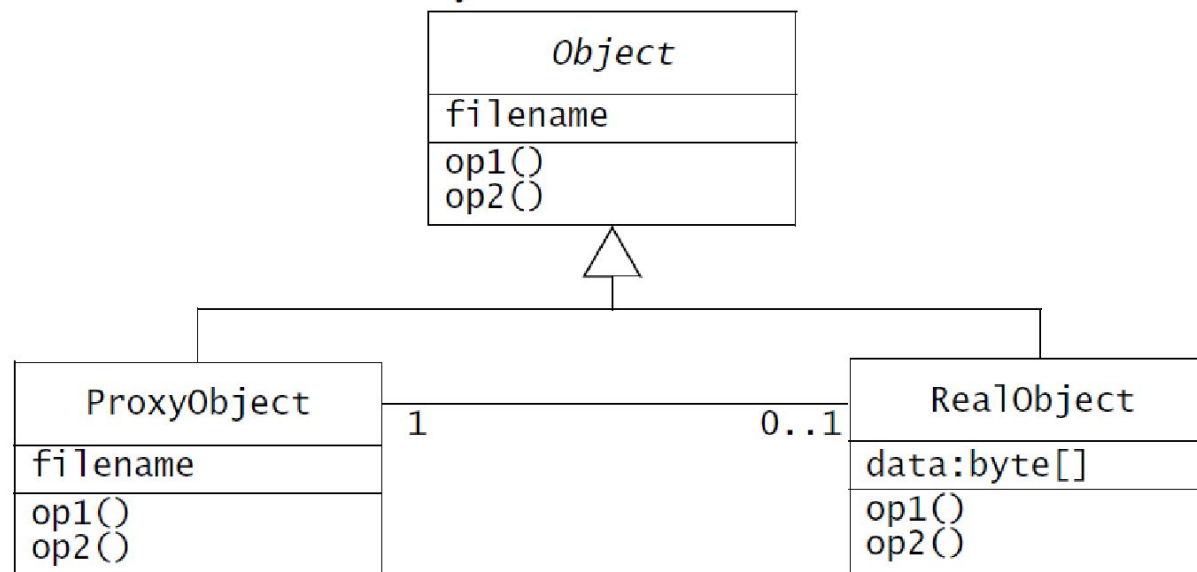


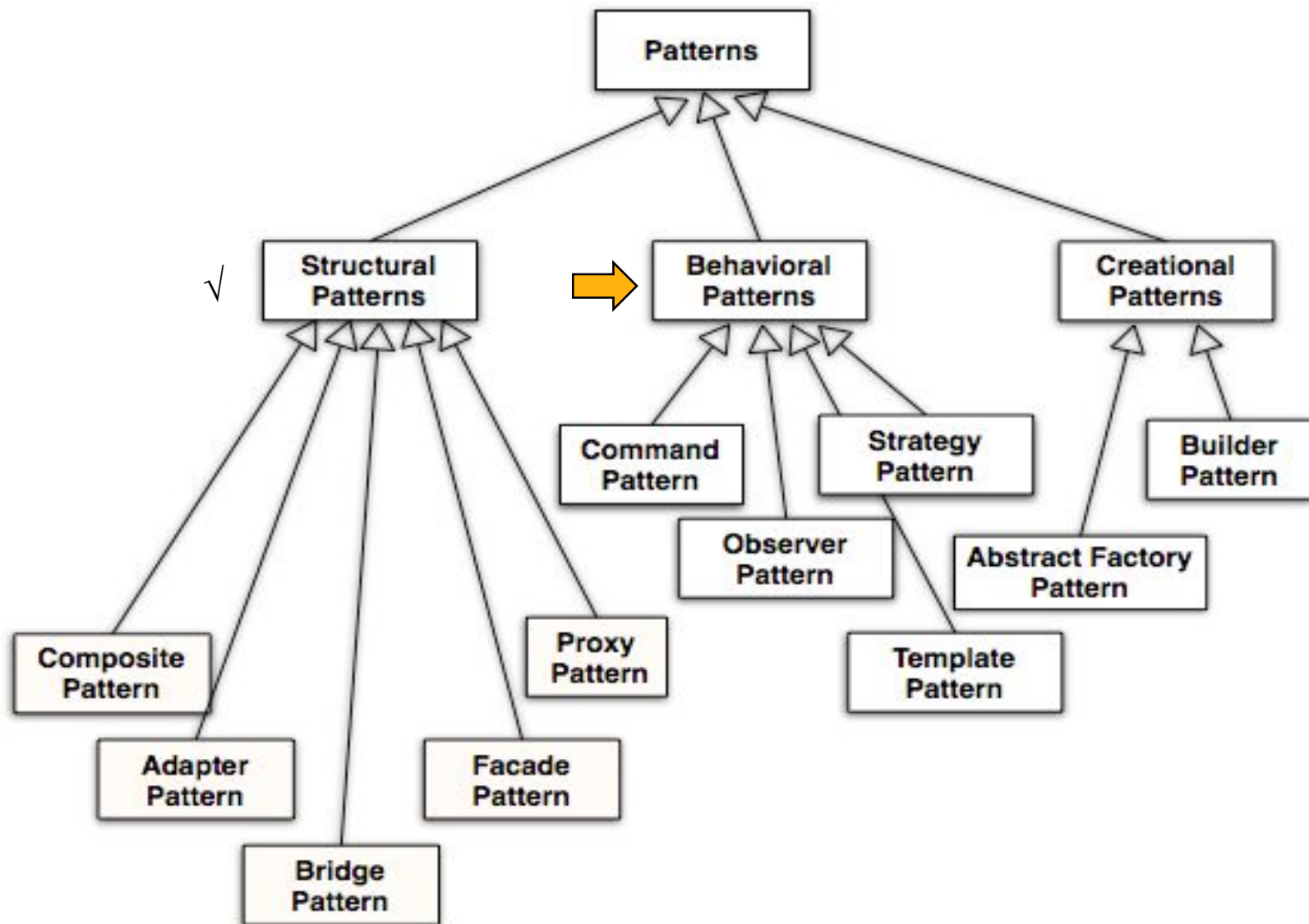
# The Proxy Pattern: 3 Types

- **Caching of information** ("Remote Proxy")
  - The Proxy object is a local representative for an object in a different address space
  - Good if information does not change too often
- **Standin** ("Virtual Proxy")
  - Object is too expensive to create or too expensive to download.
  - Good if the real object is not accessed too often
- **Access control** ("Protection Proxy")
  - The proxy object provides protection for the real object
  - Good when different actors should have different access and viewing rights for the same object
    - Example: Grade information accessed by administrators, teachers and students.

# The Proxy Pattern (Virtual Proxy)

- Improve the performance or the security of a system by delaying expensive computations, using memory only when needed, or checking access before loading an object into memory.
- The ProxyObject class acts on behalf of a RealObject class. Both classes implement the same interface. The ProxyObject stores a subset of the attributes of the RealObject. The ProxyObject handles certain requests completely (e.g., determining the size of an image), whereas others are delegated to the RealObject. After delegation, the RealObject is created and loaded in memory.

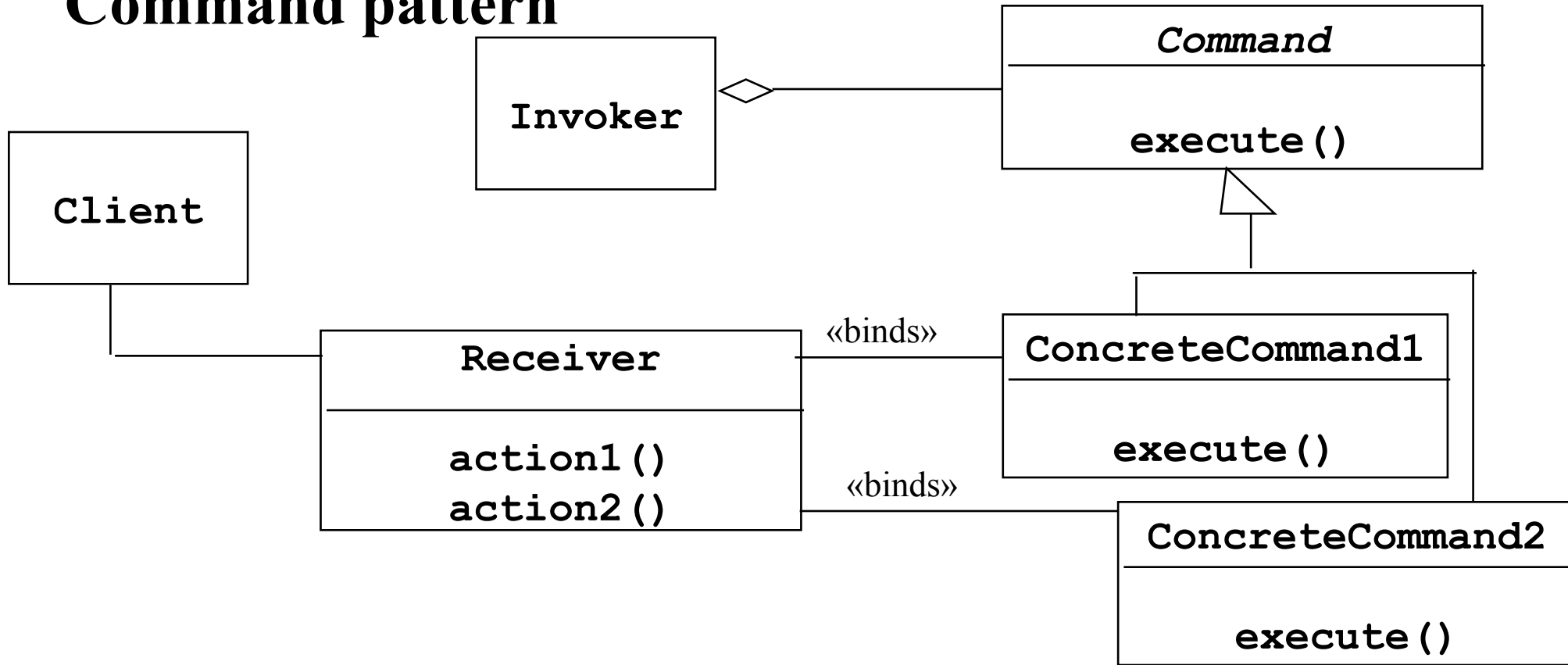




# Command Pattern: Motivation

- 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 (in this case a user interface builder) creates a ConcreteCommand 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 or undo a command (based on a string entered by the user)

# Comments to the Command Pattern

- The Command abstract class declares the interface supported by all ConcreteCommands.
- The client is a class in a user interface builder or in a class executing during startup of the application to build the user interface.
- The client creates concreteCommands and binds them to specific Receivers, this can be strings like “commit”, “execute”, “undo”.
  - So all user-visible commands are sub classes of the Command abstract class.
- The invoker - the class in the application program offering the menu of commands or buttons - invokes the concreteCommand based on the string entered and the binding between action and ConcreteCommand.



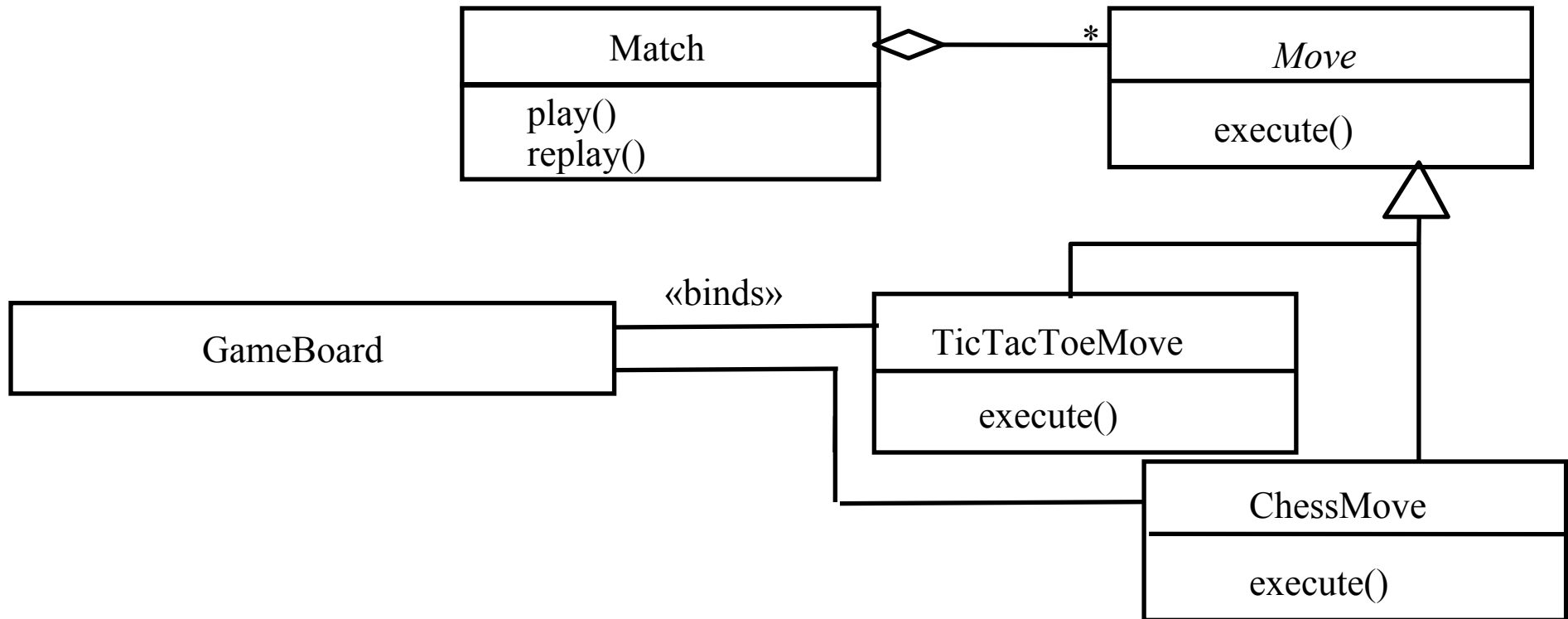
# Decouples boundary objects from control objects

- The command pattern can be nicely used to decouple boundary objects from control objects:
  - Boundary objects such as menu items and buttons, send messages to the command objects (I.e. the control objects)
  - Only the command objects modify entity objects
- When the user interface is changed (for example, a menu bar is replaced by a tool bar), only the boundary objects are modified.

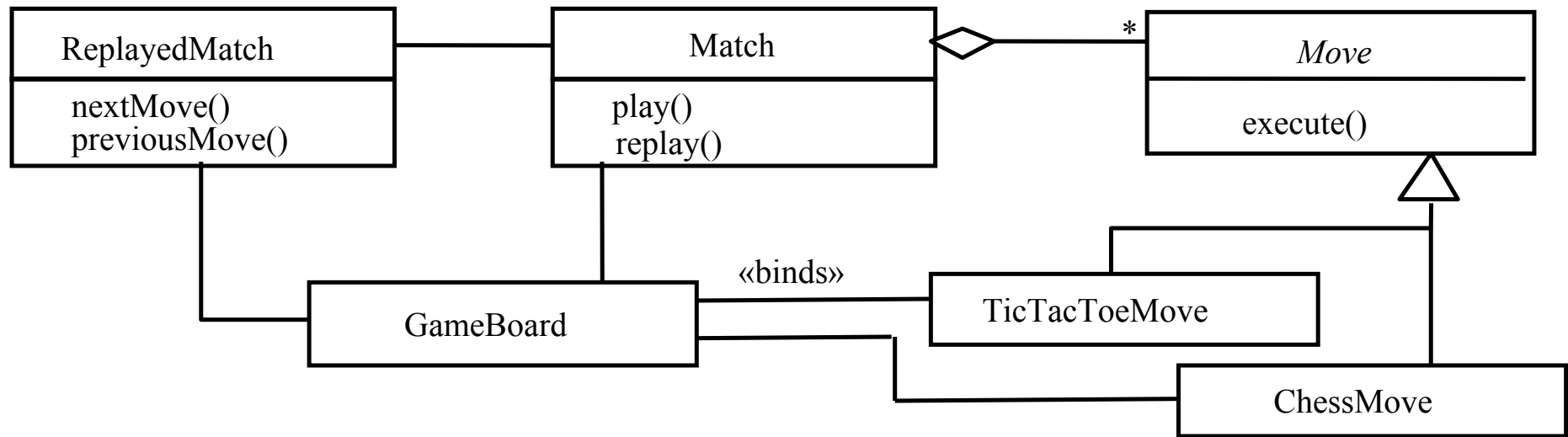
# Command Pattern Applicability

- 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

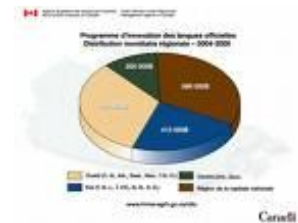
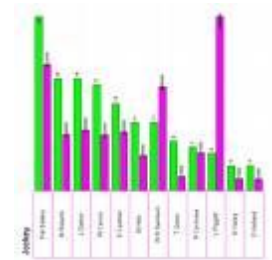
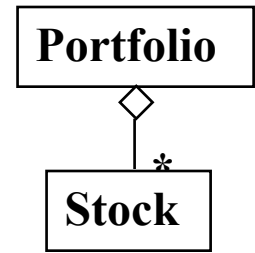


# Applying the Command design pattern to Replay Matches in ARENA



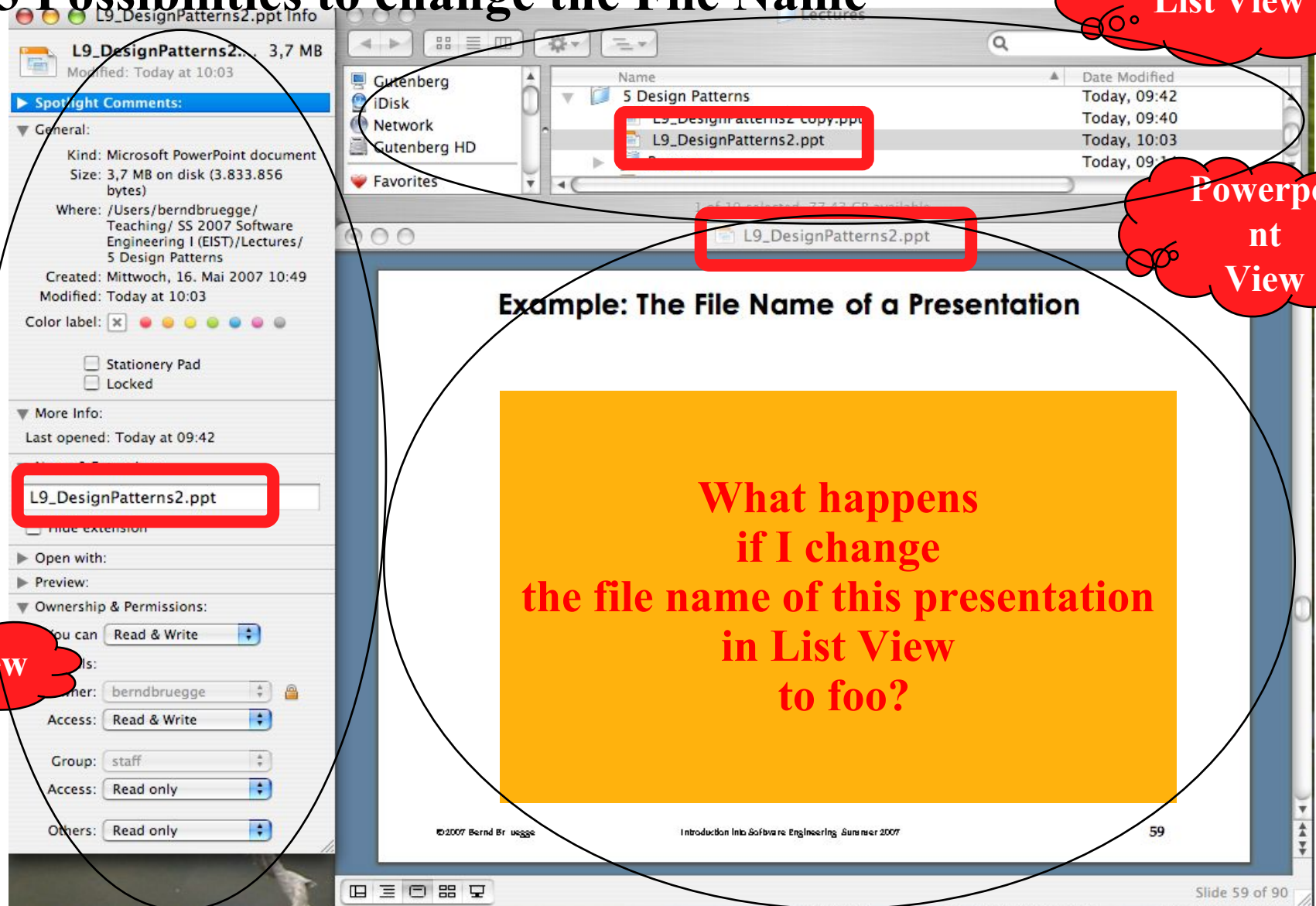
# Observer Pattern Motivation

- Problem:
  - We have an object that changes its state quite often
    - Example: A Portfolio of stocks
  - We want to provide multiple views of the current state of the portfolio
    - Example: Histogram view, pie chart view, time line view, alarm
- Requirements:
  - The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes
  - The system design should be highly extensible
    - It should be possible to add new views without having to recompile the observed object or the existing views.

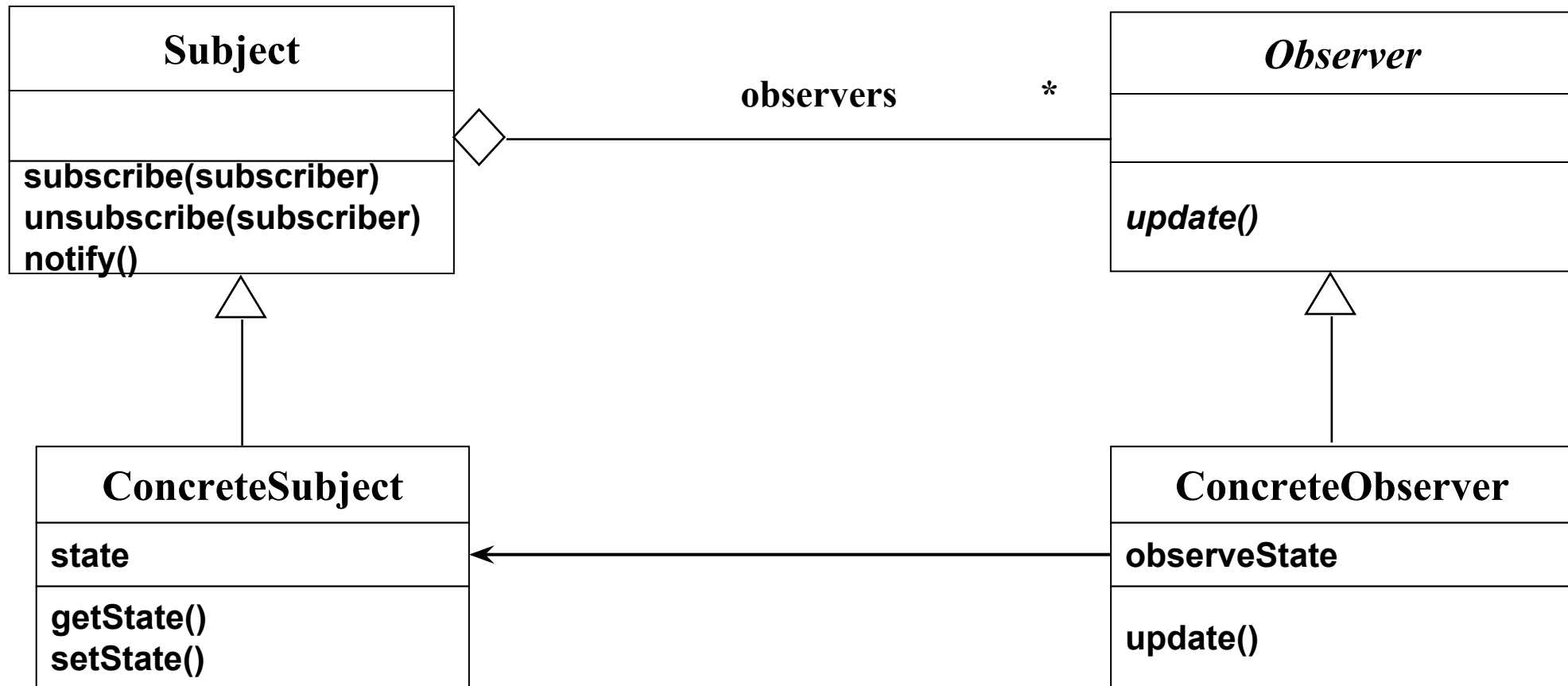


# Example: The File Name of a Presentation

## 3 Possibilities to change the File Name



# Observer Pattern: Decouples an Abstraction from its Views



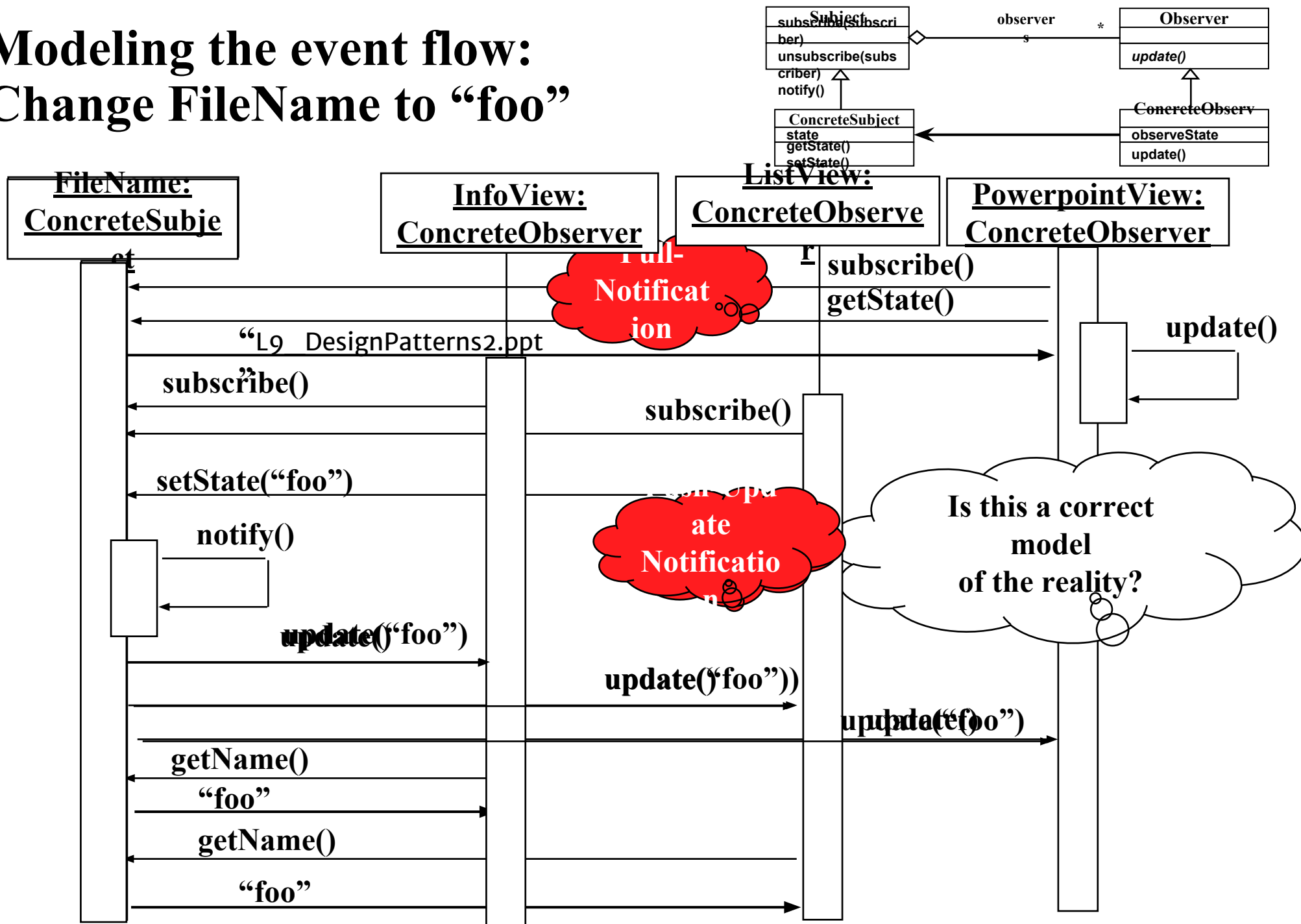
- The **Subject** ("Publisher") represents the entity object
- **Observers** ("Subscribers") attach to the Subject by calling **subscribe()**
- Each Observer has a different view of the state of the entity object
  - The **state** is contained in the subclass **ConcreteSubject**
  - The state can be **obtained and set** by subclasses of type **ConcreteObserver**.

# Observer Pattern

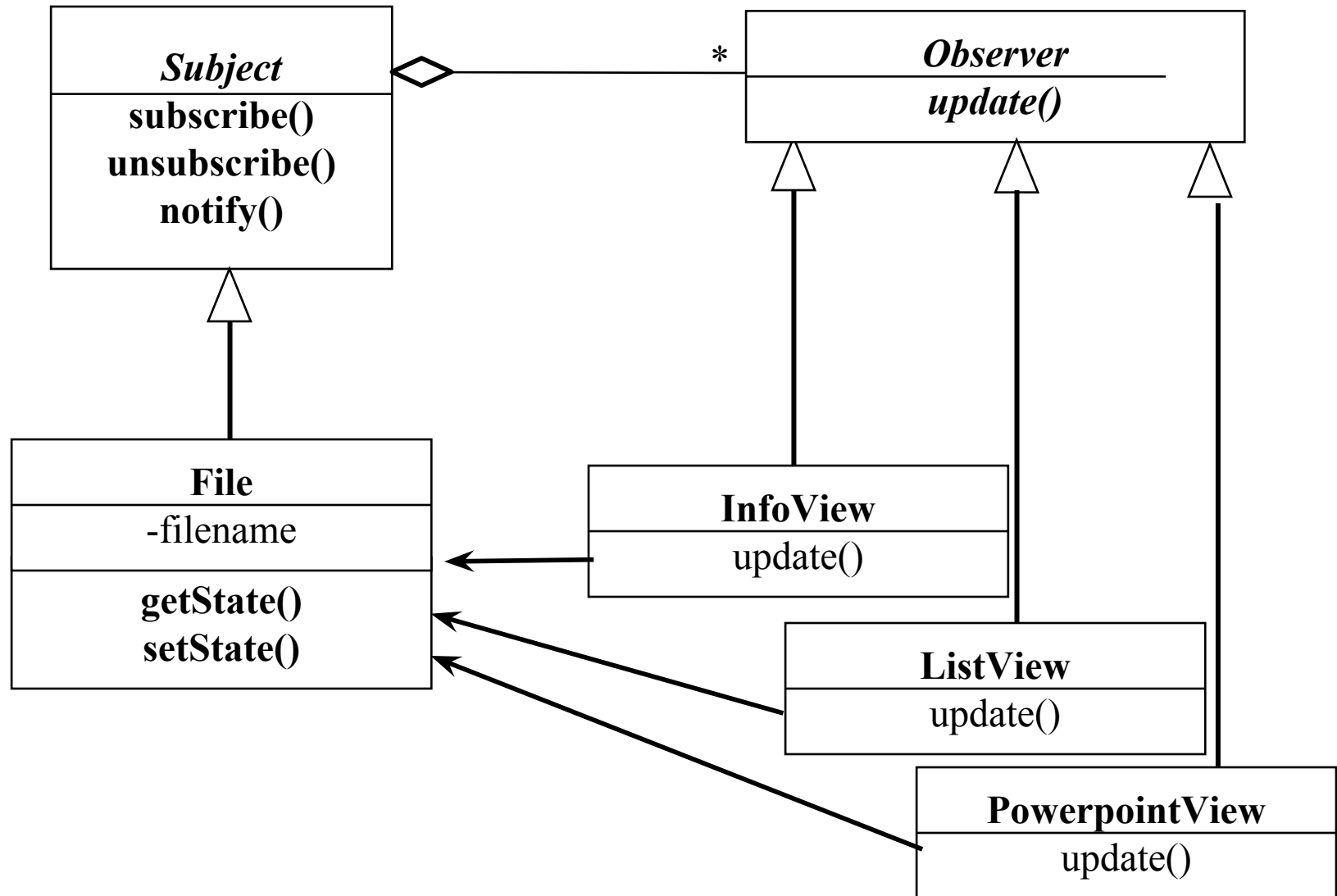
- Models a 1-to-many dependency between objects
  - Connects the state of an observed object, the **subject** with many observing objects, the **observers**
- Usage:
  - Maintaining consistency across redundant states
  - Optimizing a batch of changes to maintain consistency
- Three variants for maintaining the consistency:
  - **Push Notification**: Every time the state of the subject changes, *all* the observers are notified of the change
    - **Push-Update Notification**: The subject also sends the state that has been changed to the observers
  - **Pull Notification**: An observer inquires about the state the of the subject
- Also called **Publish and Subscribe**.



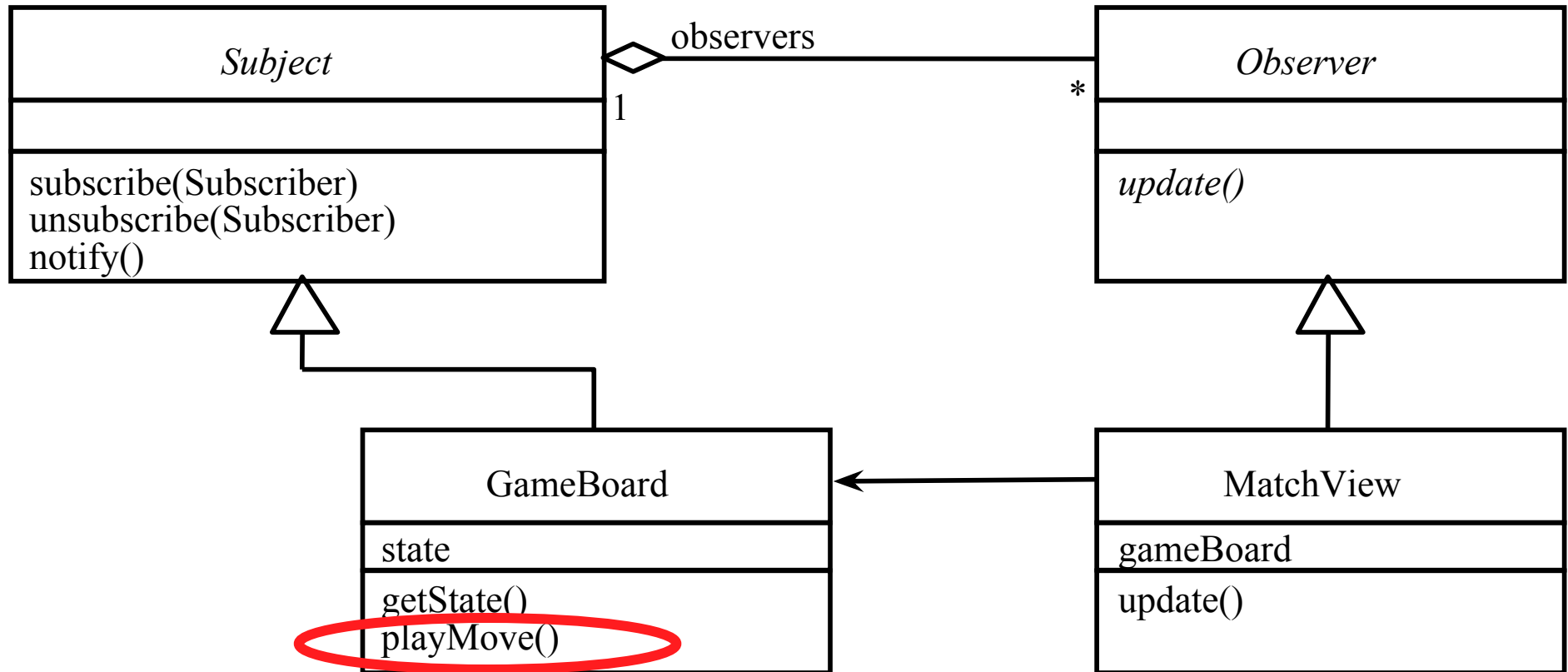
# Modeling the event flow: Change FileName to “foo”



# Applying the Observer Pattern to maintain Consistency across Views



# Applying the Observer Design Pattern to maintain Consistency across MatchViews



**Push, Pull or Push-Update Notification?**

# Strategy Pattern

- Different algorithms exists for a specific task
  - We can switch between the algorithms at run time
- Examples of tasks:
  - Different collision strategies for objects in video games
  - Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
  - Sorting a list of customers (Bubble sort, mergesort, quicksort)
- Different algorithms will be appropriate at different times
  - First build, testing the system, delivering the final product
- If we need a new algorithm, we can add it without disturbing the application or the other algorithms.

# Strategy Pattern

*Policy*

*Context*

ContextInterface()

\*

*Strategy*

*AlgorithmInterface*

**ConcreteStrategyA**

AlgorithmInterface()

**ConcreteStrategyB**

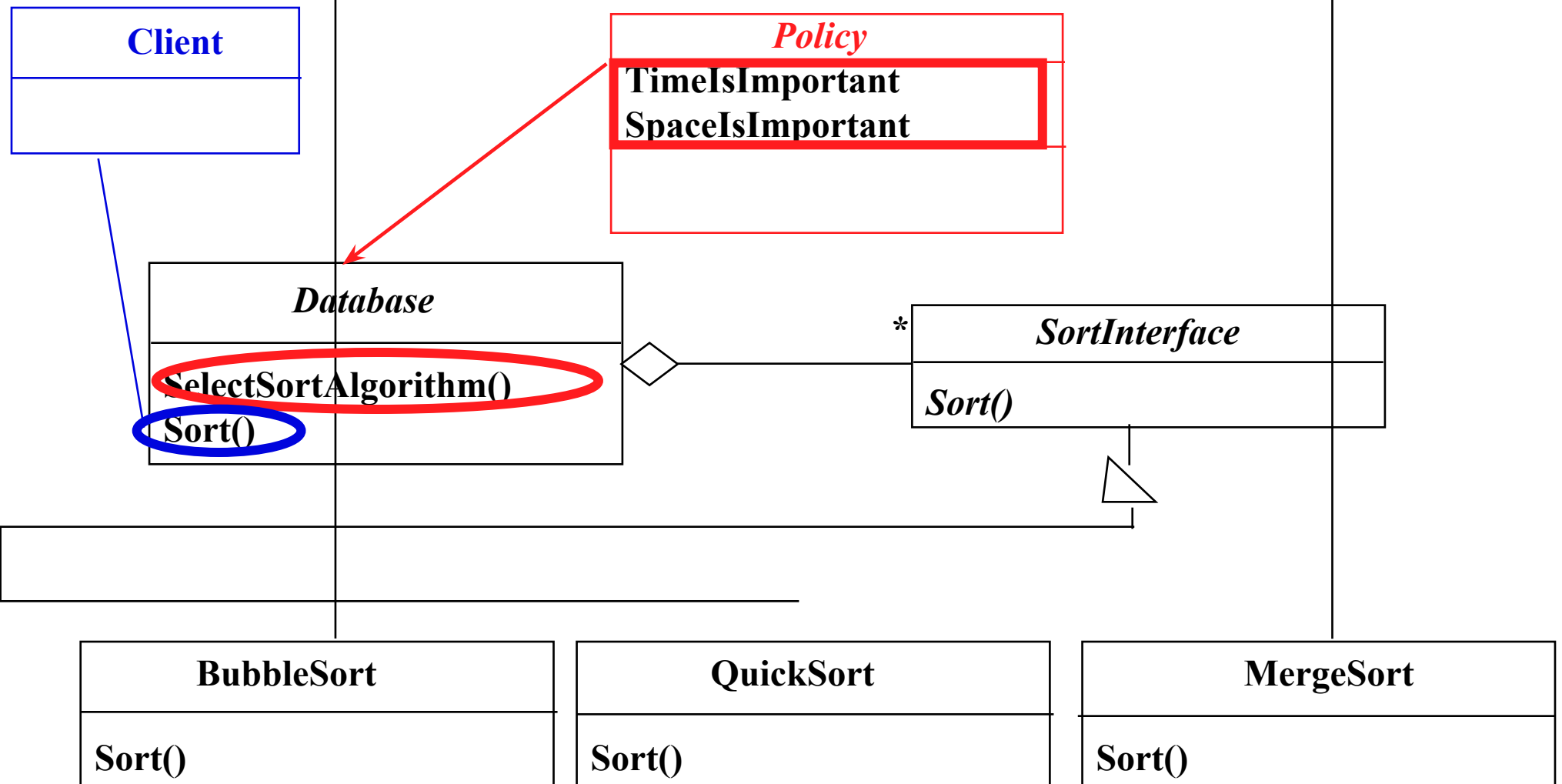
AlgorithmInterface()

**ConcreteStrategyC**

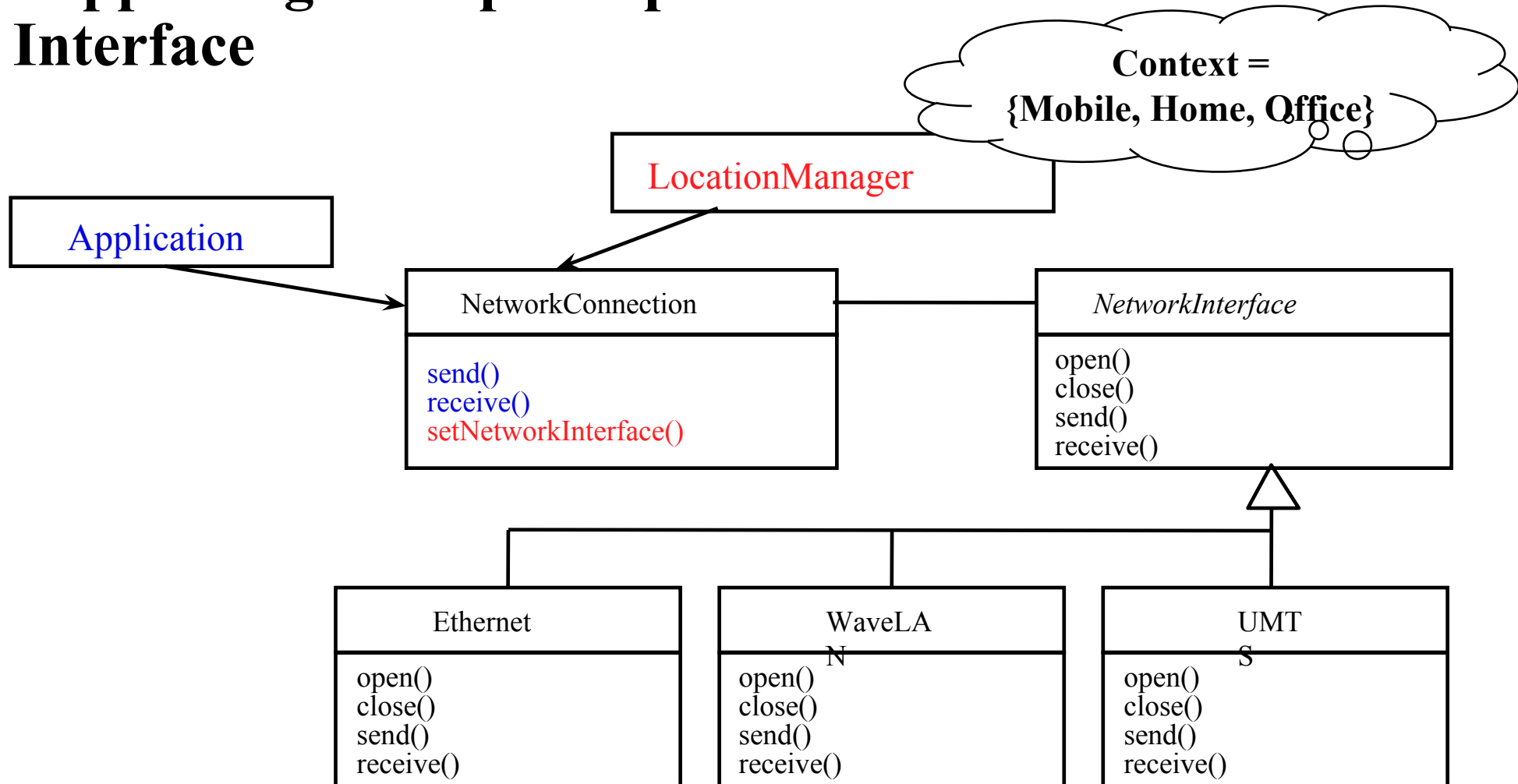
AlgorithmInterface()

Policy decides which ConcreteStrategy is best in the current Context.

# Using a Strategy Pattern to Decide between Algorithms at Runtime

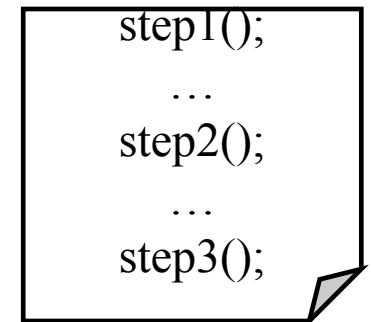


# Supporting Multiple implementations of a Network Interface



# Template Method Motivation

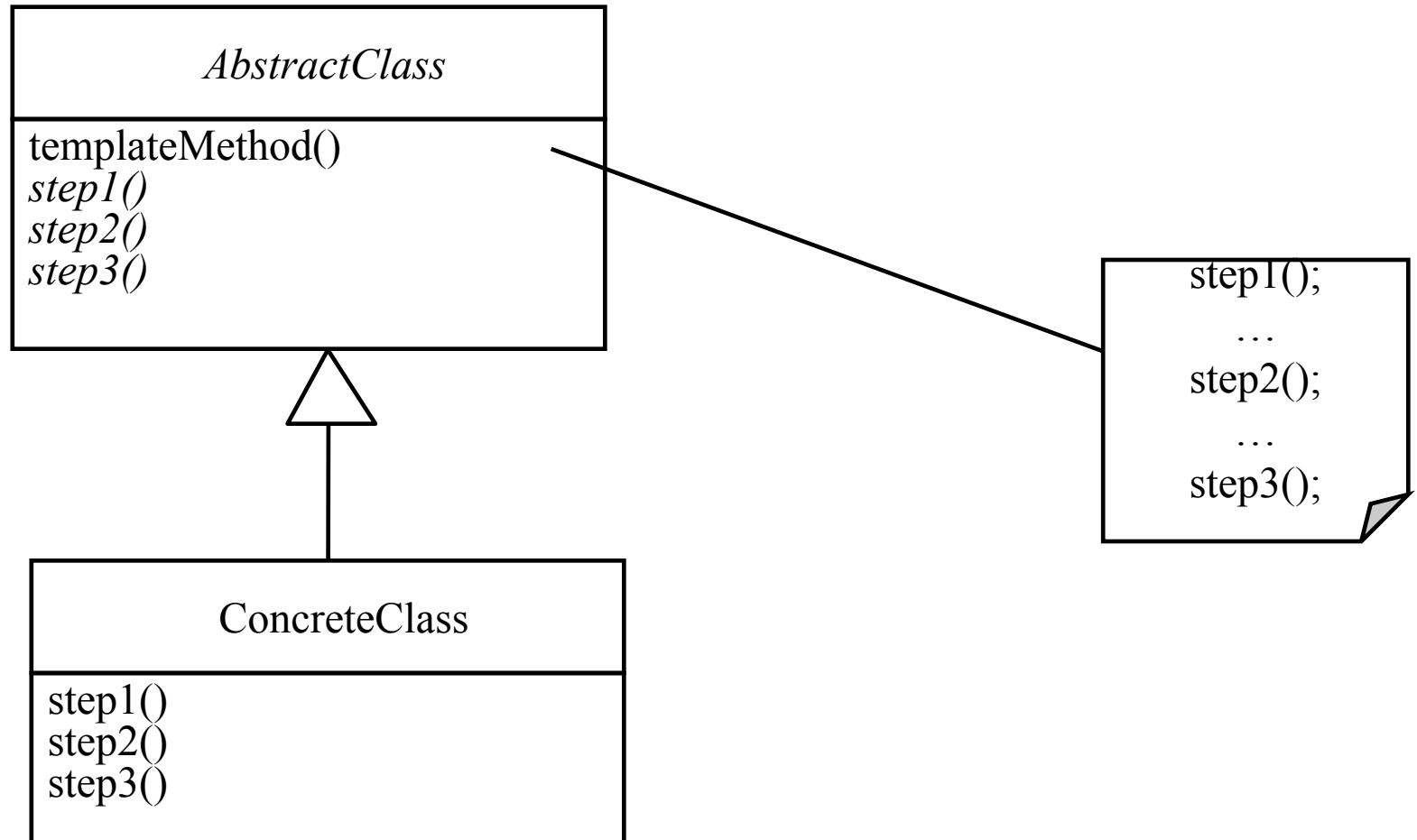
- Several subclasses share the same algorithm but differ on the specifics
- Common steps should not be duplicated in the subclasses
- Examples:
  - Executing a test suite of test cases
  - Opening, reading, writing documents of different types
- Approach
  - The common steps of the algorithm are factored out into an abstract class
    - Abstract methods are specified for each of these steps
  - Subclasses provide different realizations for each of these steps.



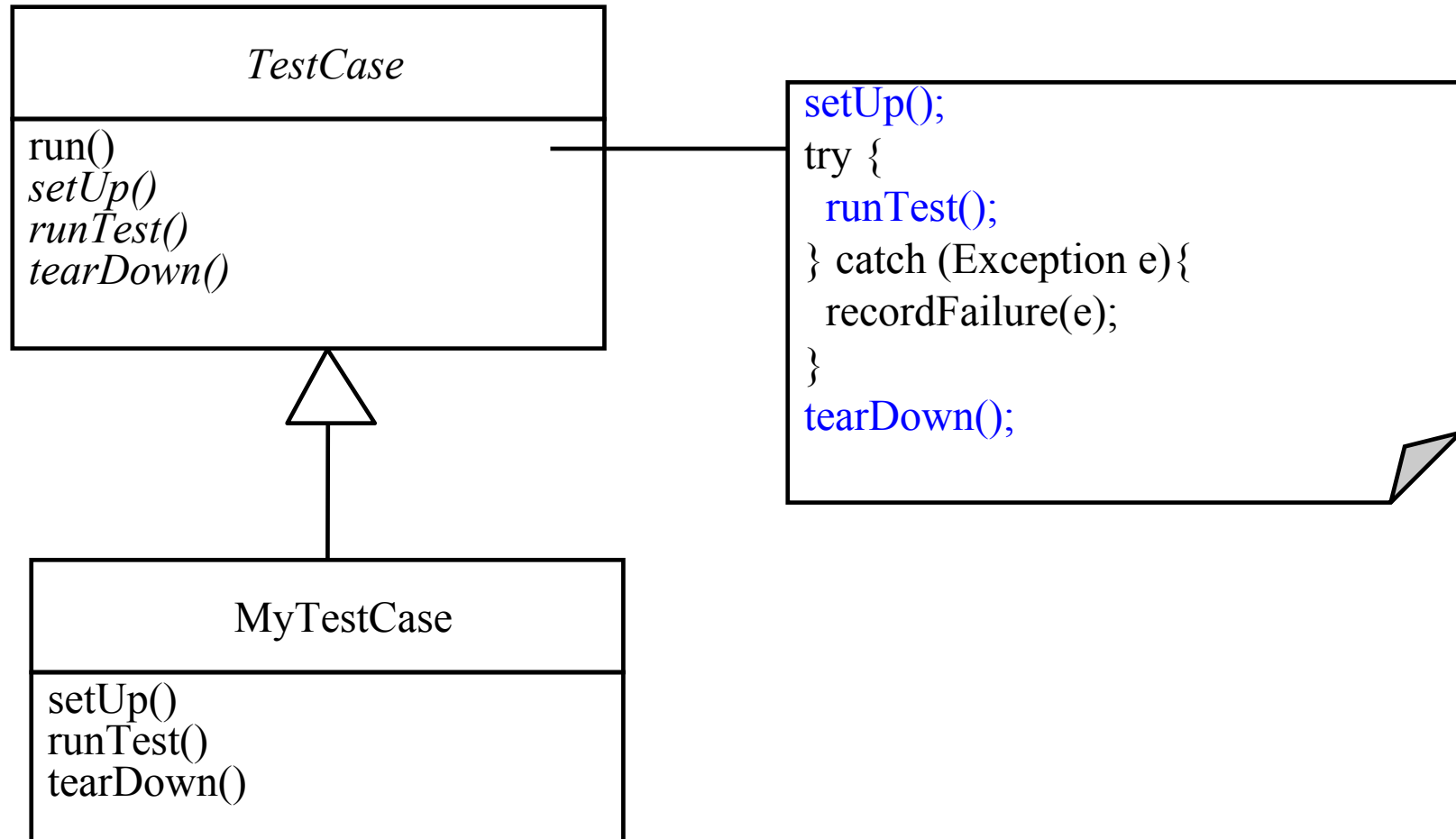
```
step1();  
...  
step2();  
...  
step3();
```



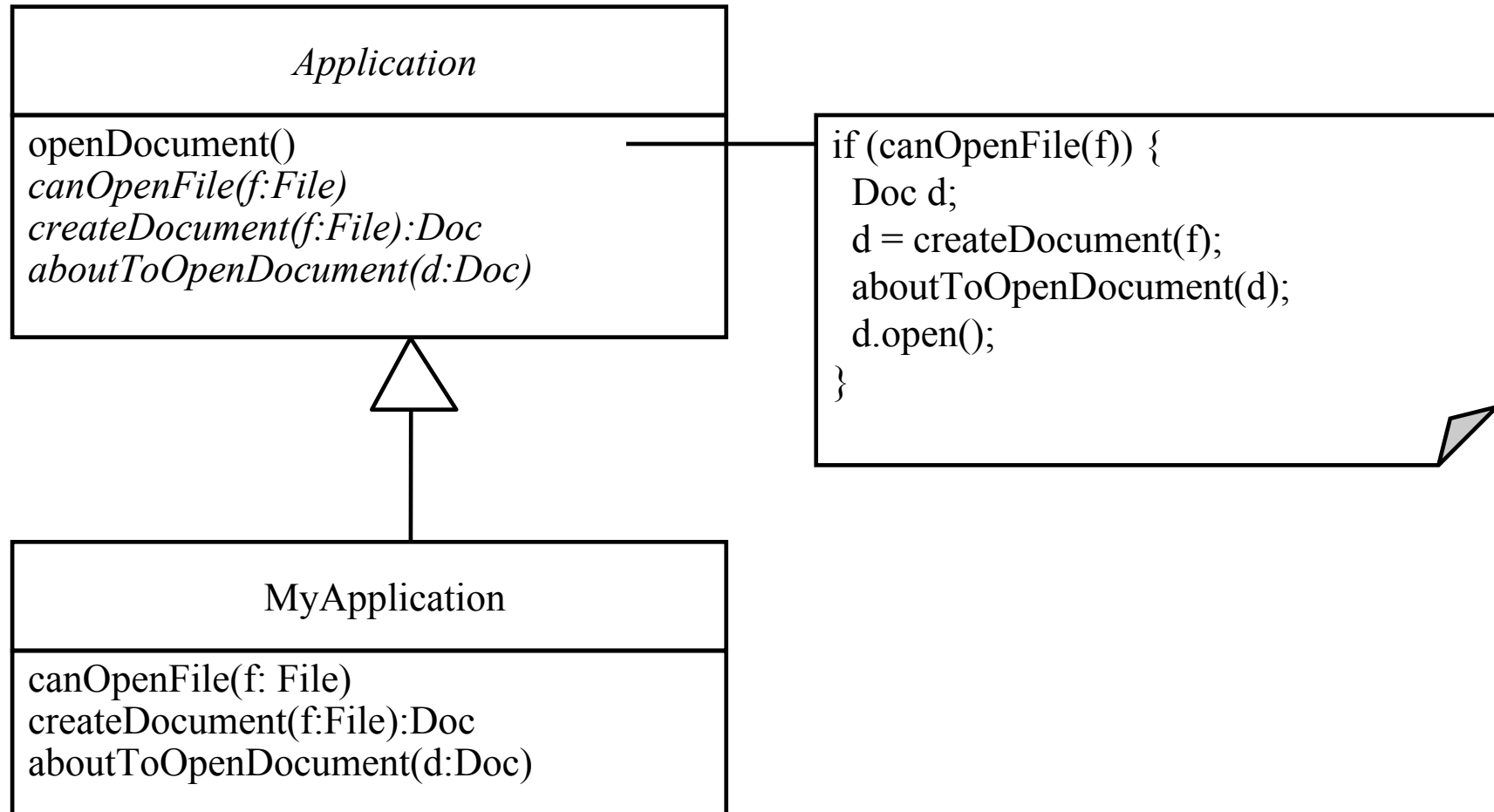
# Template Method



# Template Method Example: Test Cases

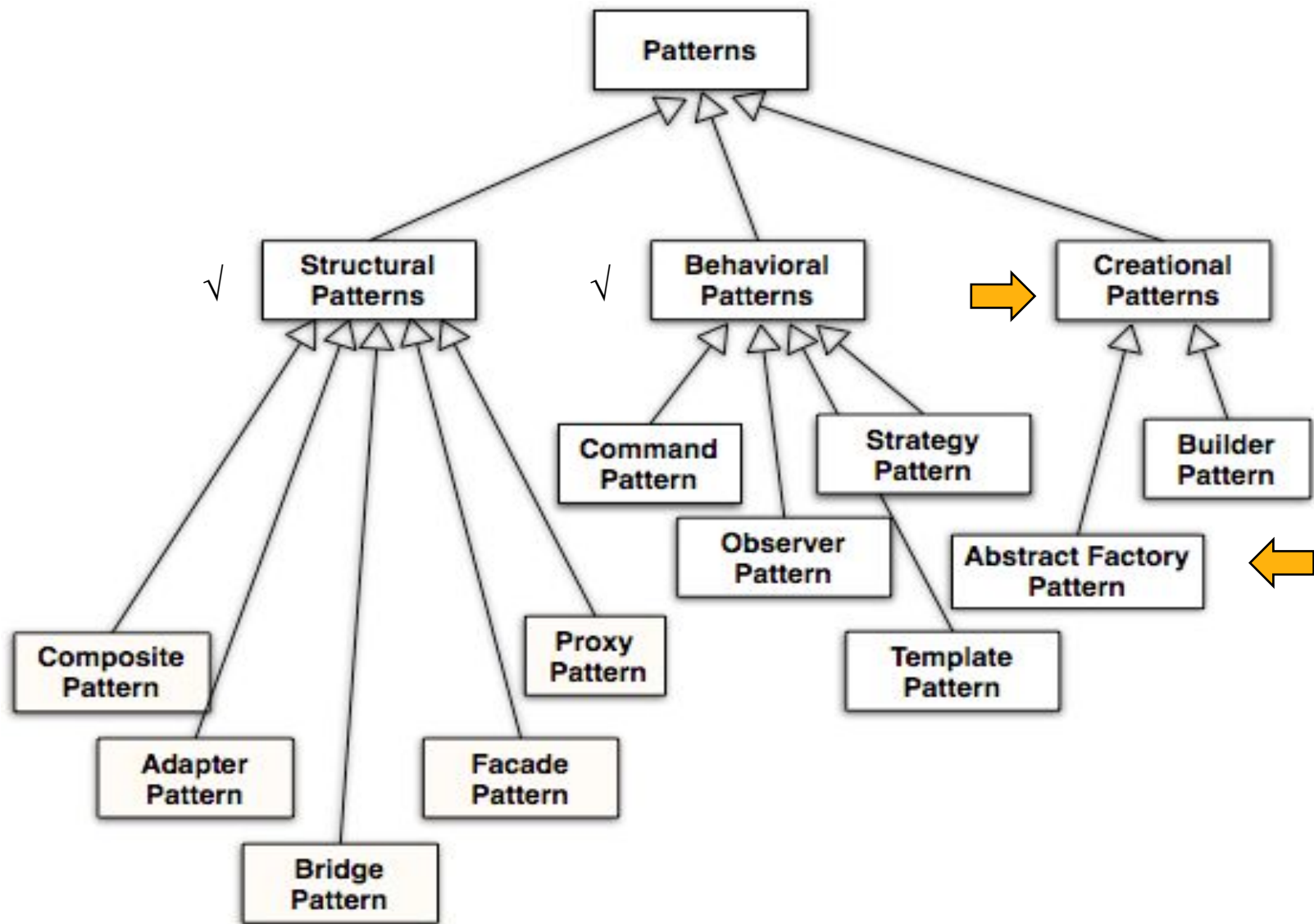


# Template Method Example: Opening Documents



# Template Method Pattern Applicability

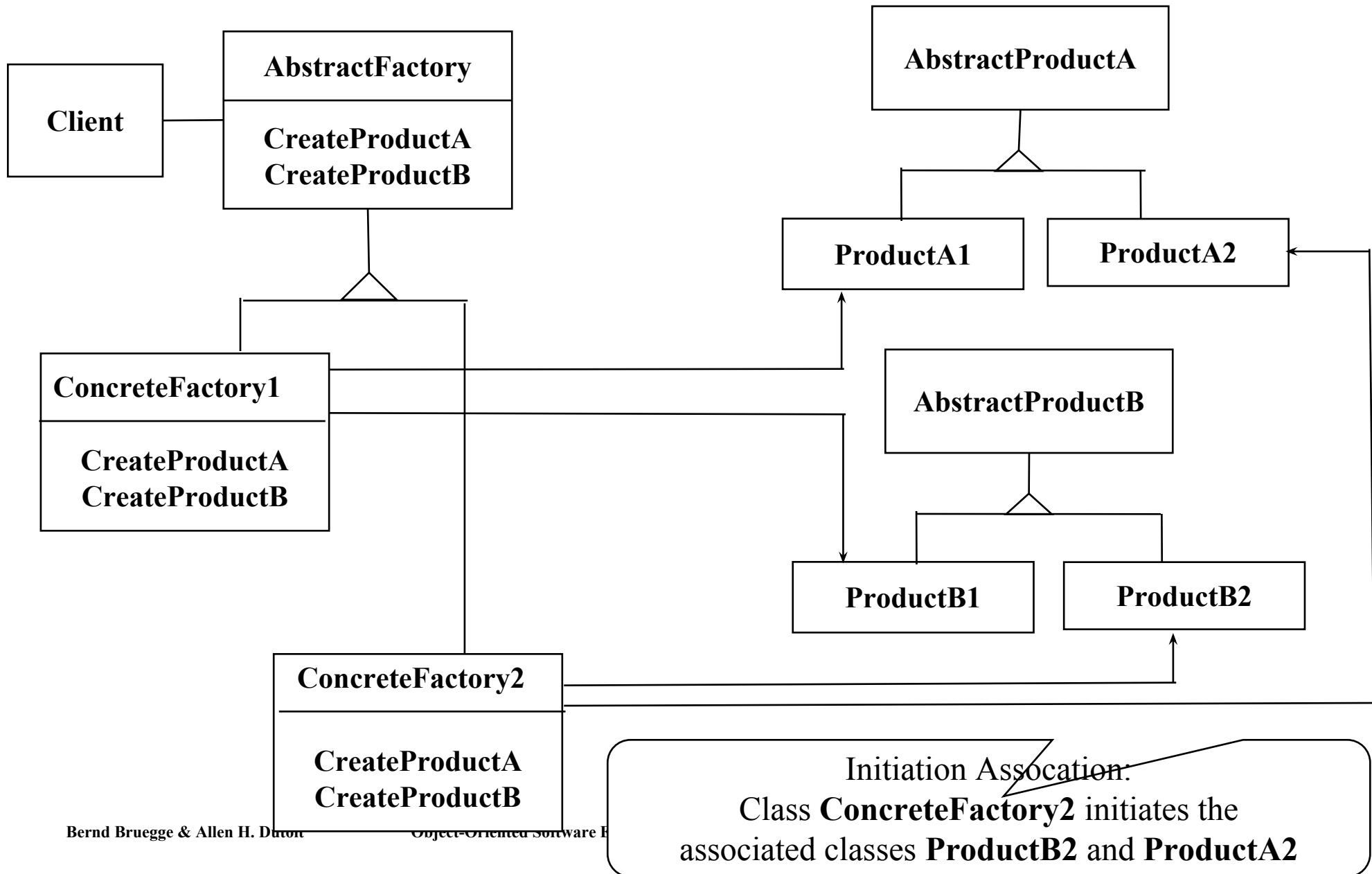
- Template method pattern **uses inheritance to vary part of an algorithm**
- Strategy pattern **uses delegation to vary the entire algorithm**
- Template Method is used in frameworks
  - The framework implements the invariants of the algorithm
  - The client customizations provide specialized steps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



# 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

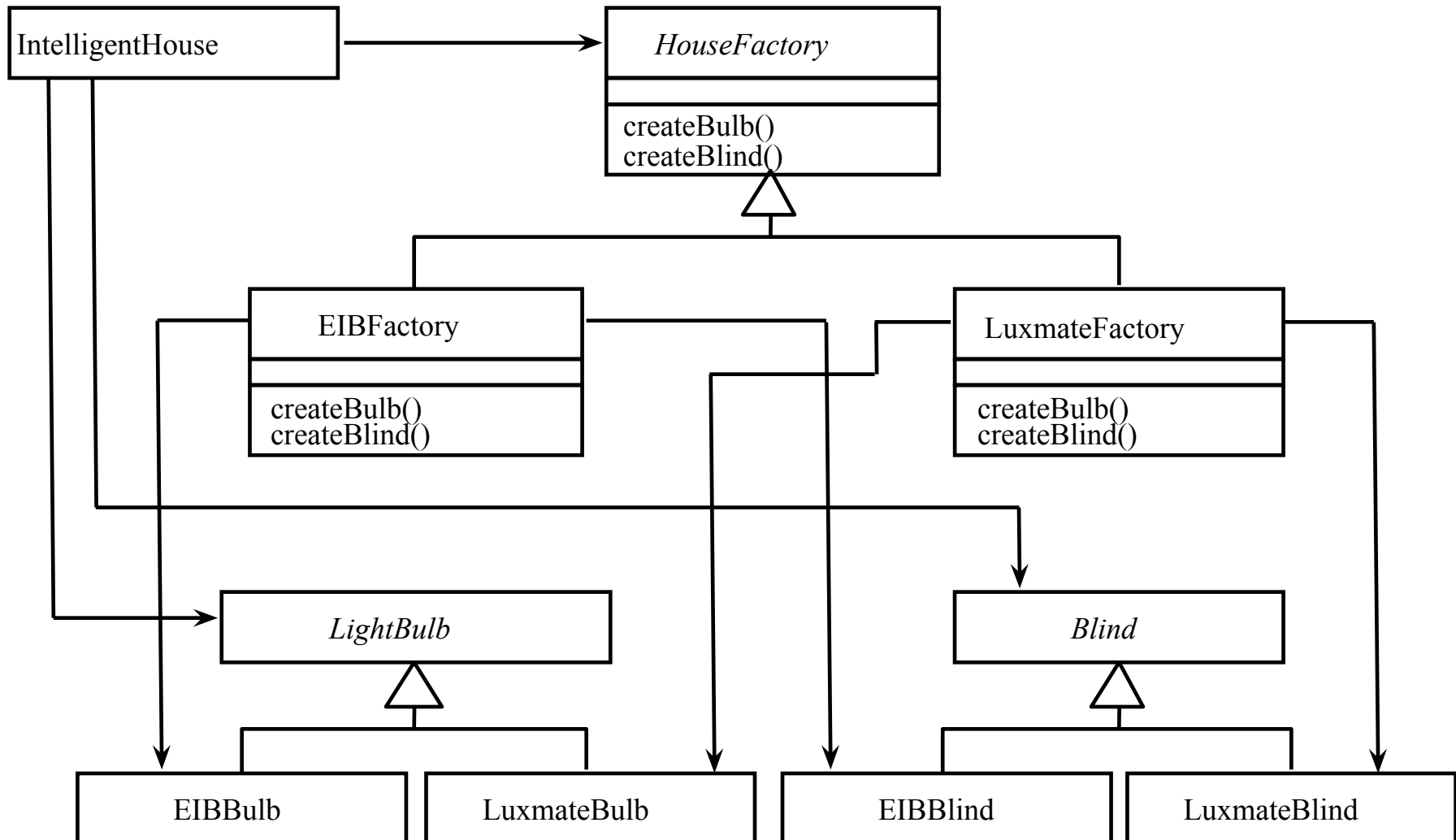


# Applicability for Abstract Factory Pattern

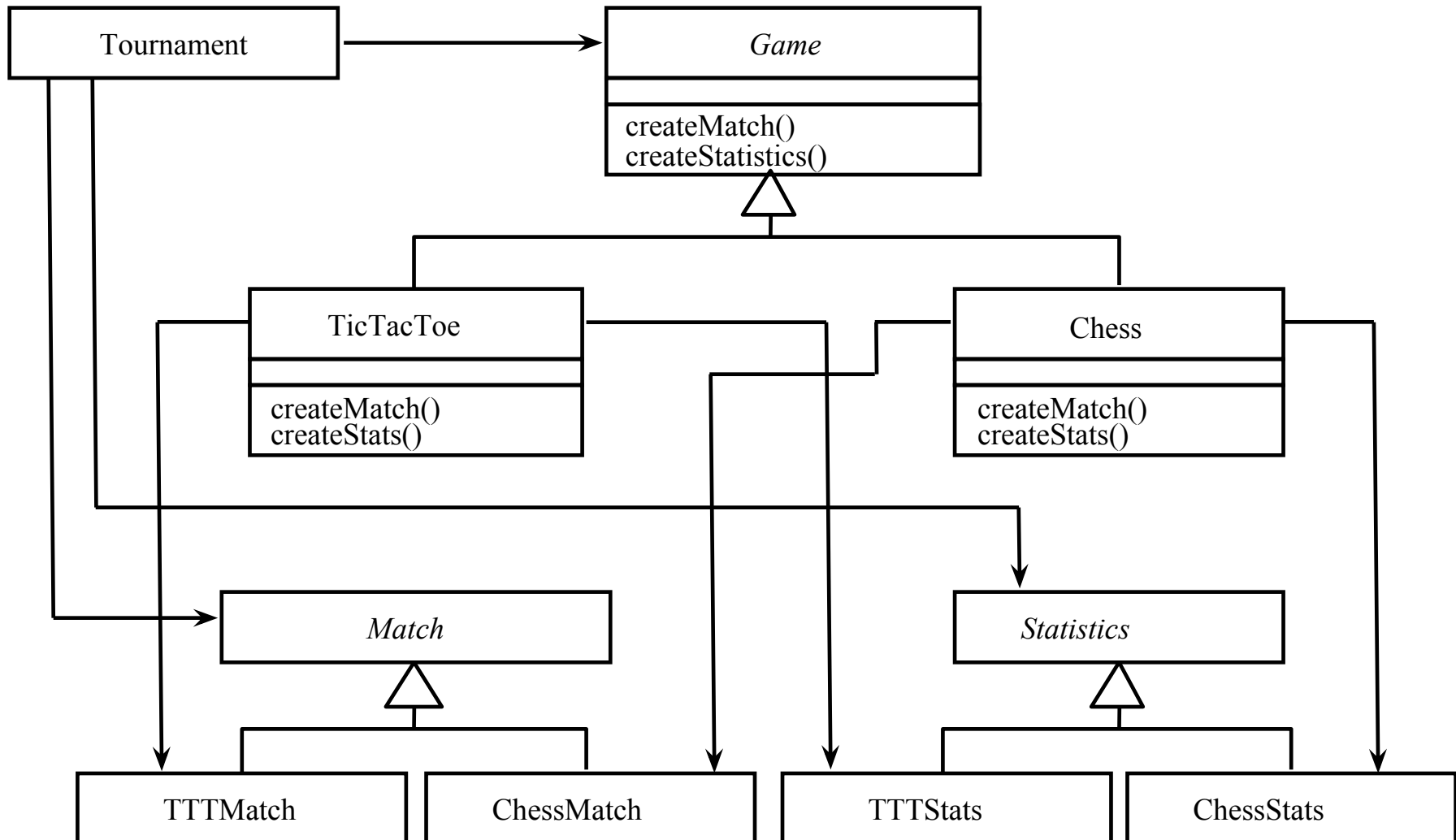
- Independence from Initialization or Representation
- Manufacturer Independence
- Constraints on related products
- Cope with upcoming change



# Example: A Facility Management System for a House



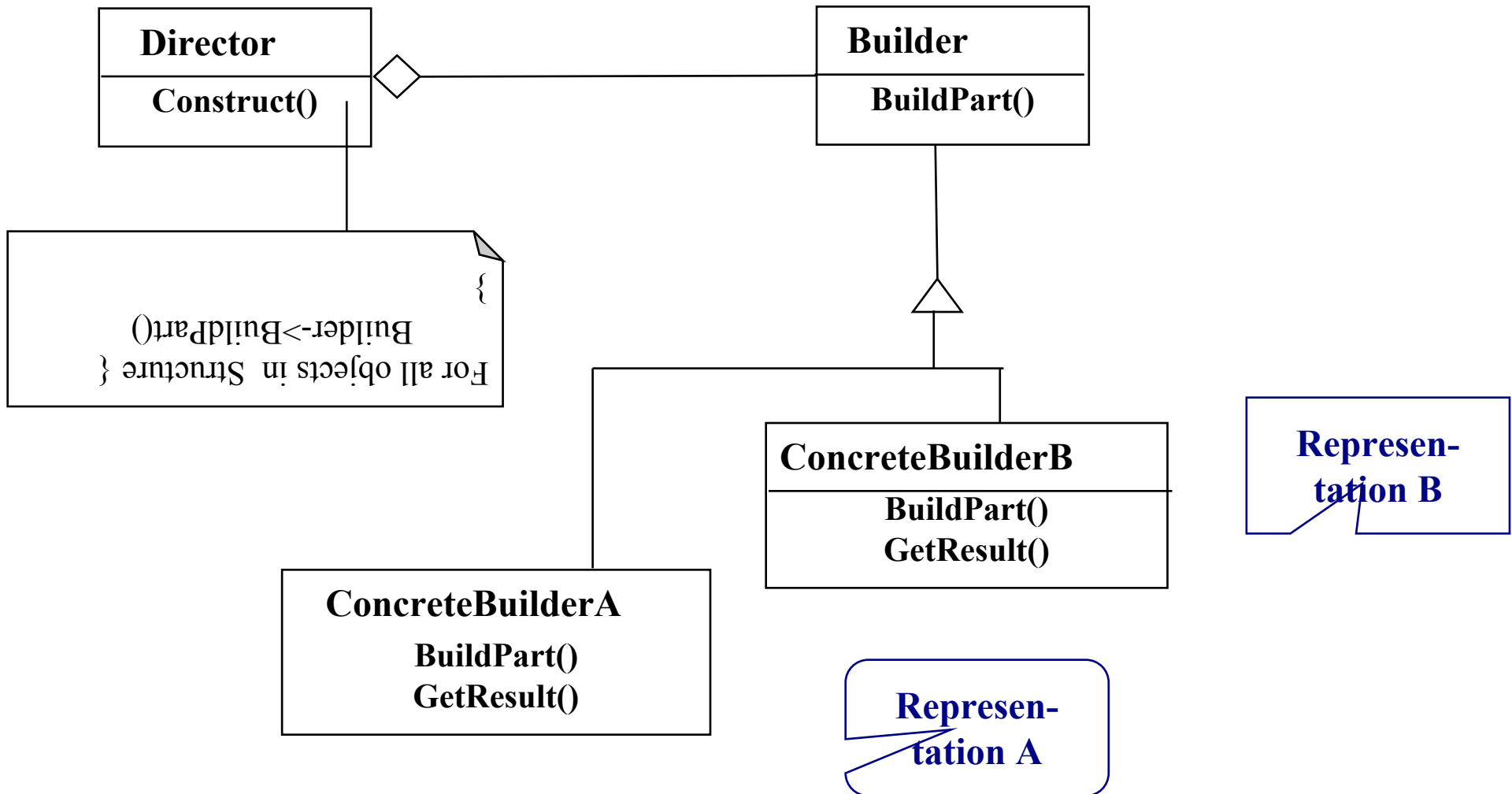
# Applying the Abstract Factory Pattern to Games



# 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")

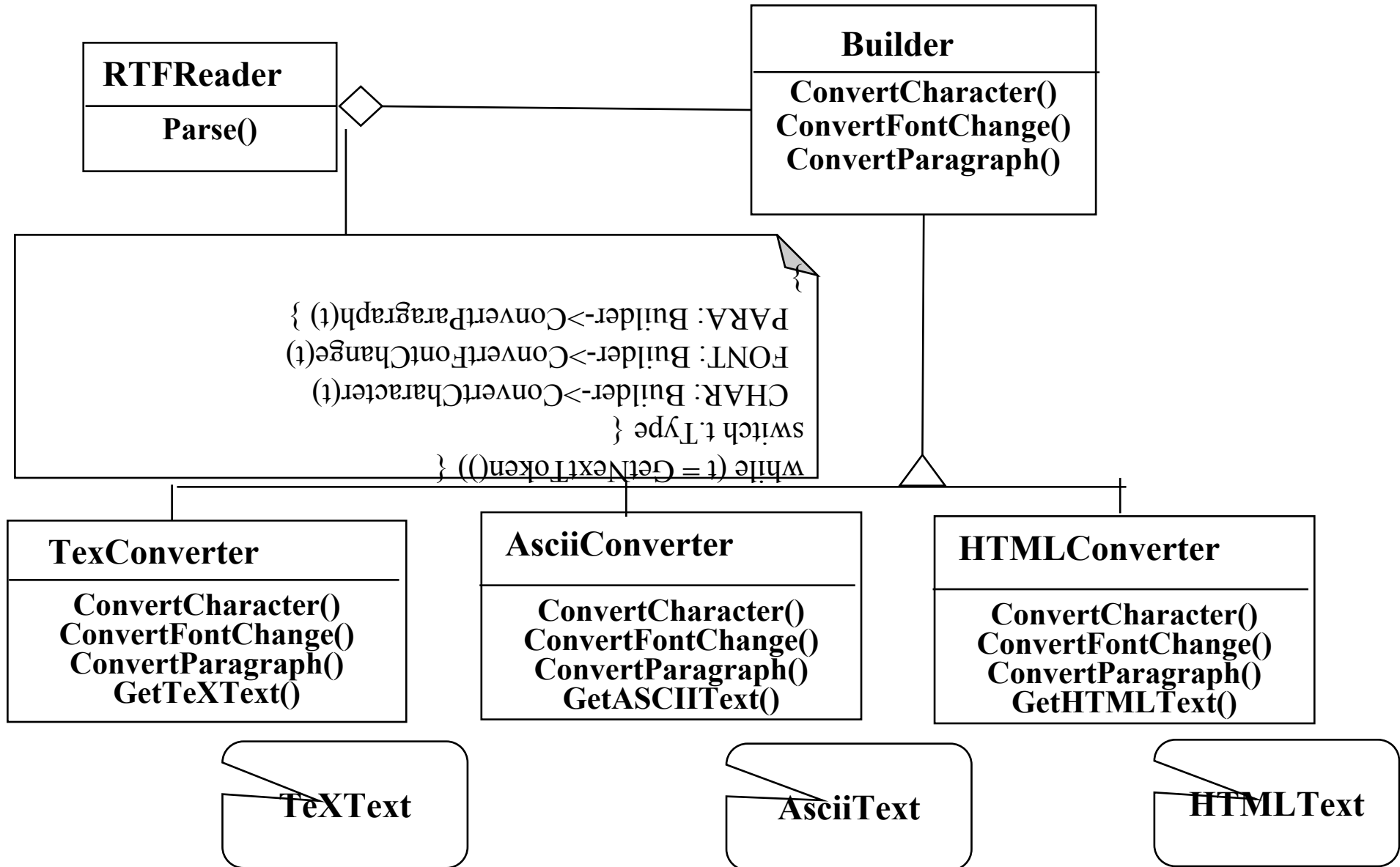
# Builder Pattern



# 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

## 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 a lot of your software development problems
  - Pattern-oriented development