Design Patterns II

Introduction into Software Engineering
Lecture 9

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Reverse Engineering Challenge: Post Mortem Thoughts

- 5 teams had a solution when the project started!
  - Lesson learned 1 (For developers): When you reuse a design or source code, make sure the requirements have not changed:-)
- First handed-in solution
  - Seemed to have passed the client acceptance test
  - But it was not correct:
    - It did not reduce the speed by 50%
  - Lesson learned 2 (for Management): Make sure the client acceptance test covers all the requirements.
  - Consolation prize: Jakob Mund
- We have a winner: Team „Philip Lorenz"
- Lottery for second prize (>40 submissions!)

Miscellaneous

- The "Prüfungsausschuß" requires most students to register in HISQIS for their exams until May 25

  => Please see our website for more details

Is this a good Model?

```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class Stubcode implements SeatImplem   ation {
  public int GetPo
    // stub code f
  }
  ...
}
public class AimSe                                tion {
  public int GetPo
    // actual cal                                m
  }
  ….
}
public class SARTSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the SART seat simulator
 }
  ...
}
```

It depends!

Reverse Engineering Challenge: Post Mortem Thoughts

- 5 teams had a solution when the project started!
  - Lesson learned 1 (For developers): When you reuse a design or source code, make sure the requirements have not changed:-)
- First handed-in solution
  - Seemed to have passed the client acceptance test
  - But it was not correct:
    - It did not reduce the speed by 50%
  - Lesson learned 2 (for Management): Make sure the client acceptance test covers all the requirements.
  - Consolation prize: Jakob Mund
- We have a winner: Team „Philip Lorenz“
- Lottery for second prize (>40 submissions!)

Miscellaneous

- The "Prüfungsauschuß" requires most students to register in HISQIS for their exams until May 25

  => Please see our website for more details

A Game: Get-15

- Start with the nine numbers 1,2,3,4, 5, 6, 7, 8 and 9.
- You and your opponent take alternate turns, each taking a number
- Each number can be taken only once: If you opponent has selected a number, you cannot also take it.
- The first person to have any three numbers that total 15 wins the game.
- Example:

You:	**1**	**5**	**3**	**8**	
Opponent:	**6**	**9**	**7**	**2**	**Opponent Wins!**

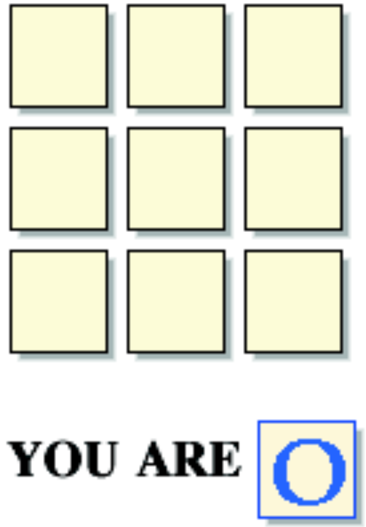
Characteristics of Get-15

- Hard to play,
- The game is especially hard, if you are not allowed to write anything done.

- Why?
  - All the numbers need to be scanned to see if you have won/lost
  - It is hard to see what the opponent will take if you take a certain number
  - The choice of the number depends on all the previous numbers

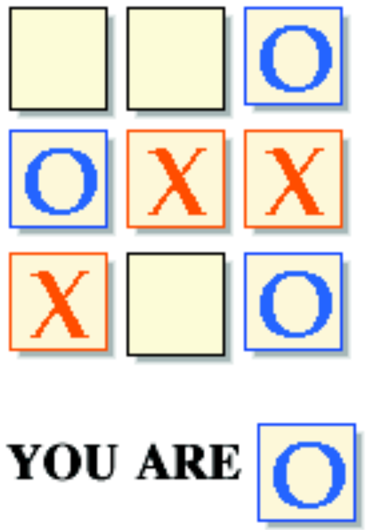
  - Not easy to devise an simple strategy

Another Game: Tic-Tac-Toe

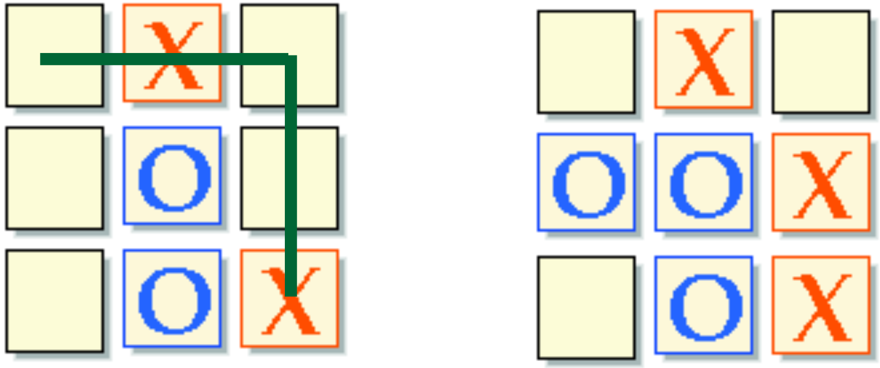


Source: http://boulter.com/ttt/index.cgi

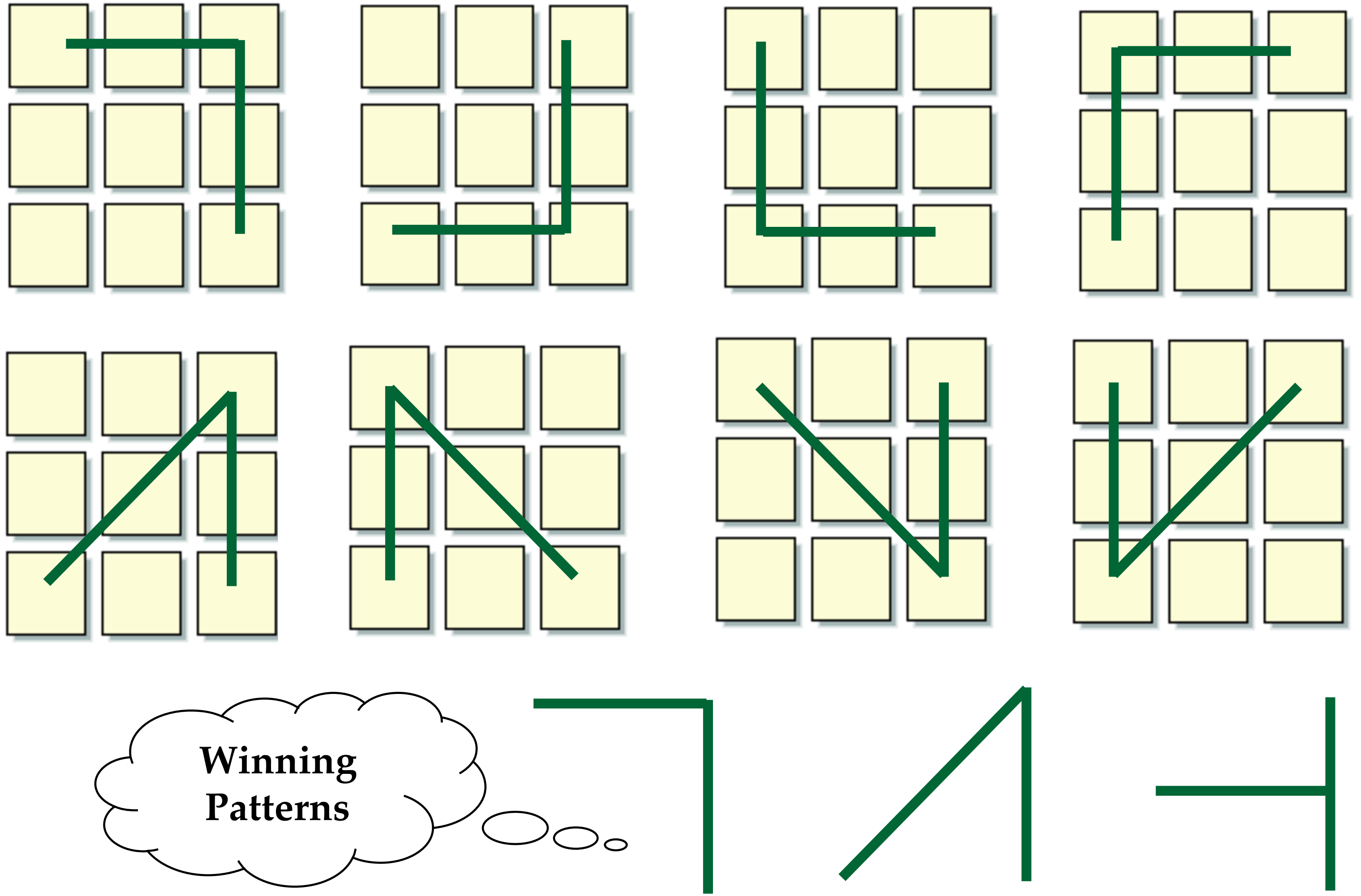
A Draw Sitation



Strategy for determining a winning move

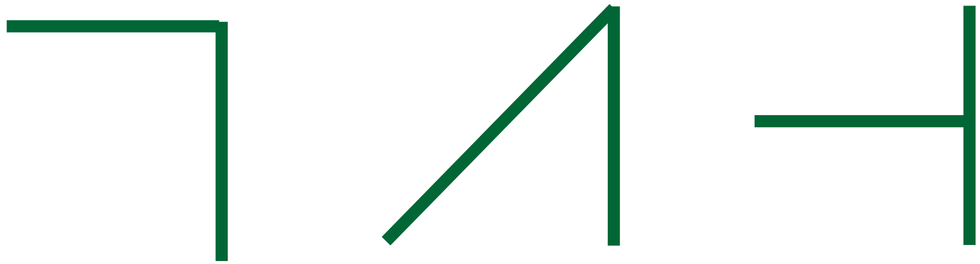


Winning Situations for Tic-Tac-Toe



Tic-Tac-Toe is "Easy"

- Why?   Reduction of complexity through patterns and symmetry
- **Patterns**: Knowing the following two patterns, the player can anticipate the opponents move



- **Symmetry**:
  - The player needs to remember only these three patterns to deal with 8 different game situations
  - The player needs to memorize only 3 opening moves and their responses

Get-15 and Tic-Tac-Toe are identical problems

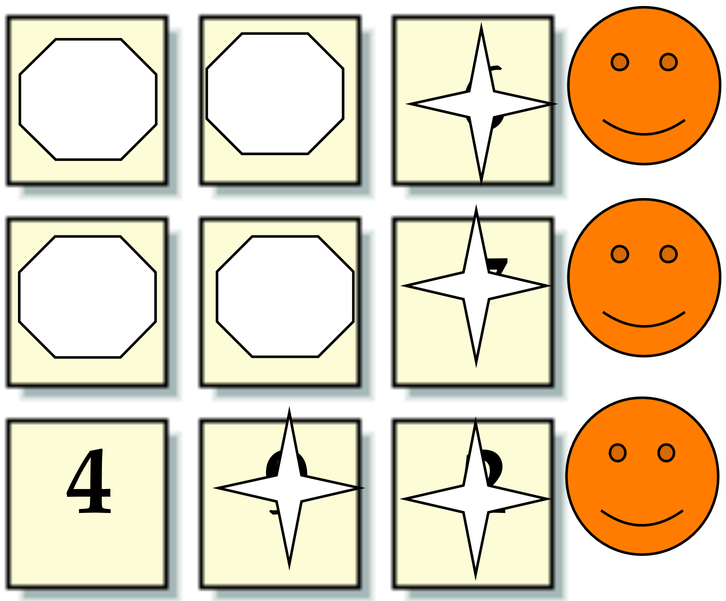
- Any Get-15 solution is a solution to a tic-tac-toe problem
- Any tic-tac-toe solution is a solution to a Get-15 problem
- To see the relationship between the two games, we simply arrange the 9 digits into the following pattern

8	1	6
3	5	7
4	9	2

You: 1 5 3 8

Opponent: 6 9 7 2

8	1	6
3	5	7
4	9	2



- During object modeling we do many transformations and changes to the object model
- It is important to make sure the object model stays simple!

- Design patterns are used to keep system models simple (and reusable).

Modeling Heuristics

- Modeling must address our mental limitations:
  - Our short-term memory has only limited capacity (7+-2)
- Good Models deal with this limitation, because they
  - Do not tax the mind
    - A good model requires a small mental effort
  - Reduce complexity
    - Turn complex tasks into easy ones (choice of representation)
    - Use of symmetries
  - Use abstractions
    - Taxonomies
  - Have organizational structure:
    - Memory limitations are overcome with an appropriate representation ("natural model").

Outline

- Design Patterns
  - Usefulness of design patterns
  - Design Pattern Categories
- Patterns already covered: Proxy, Strategy
- Patterns covered in this lecture
  - Composite: Modeling of dynamic aggregates
  - Facade: Interfacing to subsystems
  - Adapter: Interfacing to existing systems (legacy systems)
  - Bridge: Interfacing to existing and future systems
- Patterns covered next week and in the exercises
  - Command, Observer, Template Method, Abstract Factory, Builder.

What is common between these definitions?

Recursion

- Definition Software System
  - A software system consists of subsystems which are either other subsystems or collection of classes

- Definition Software Lifecycle
  - A software lifecycle consists of a set of development activities which are either other activities or collection of tasks.

Recursion

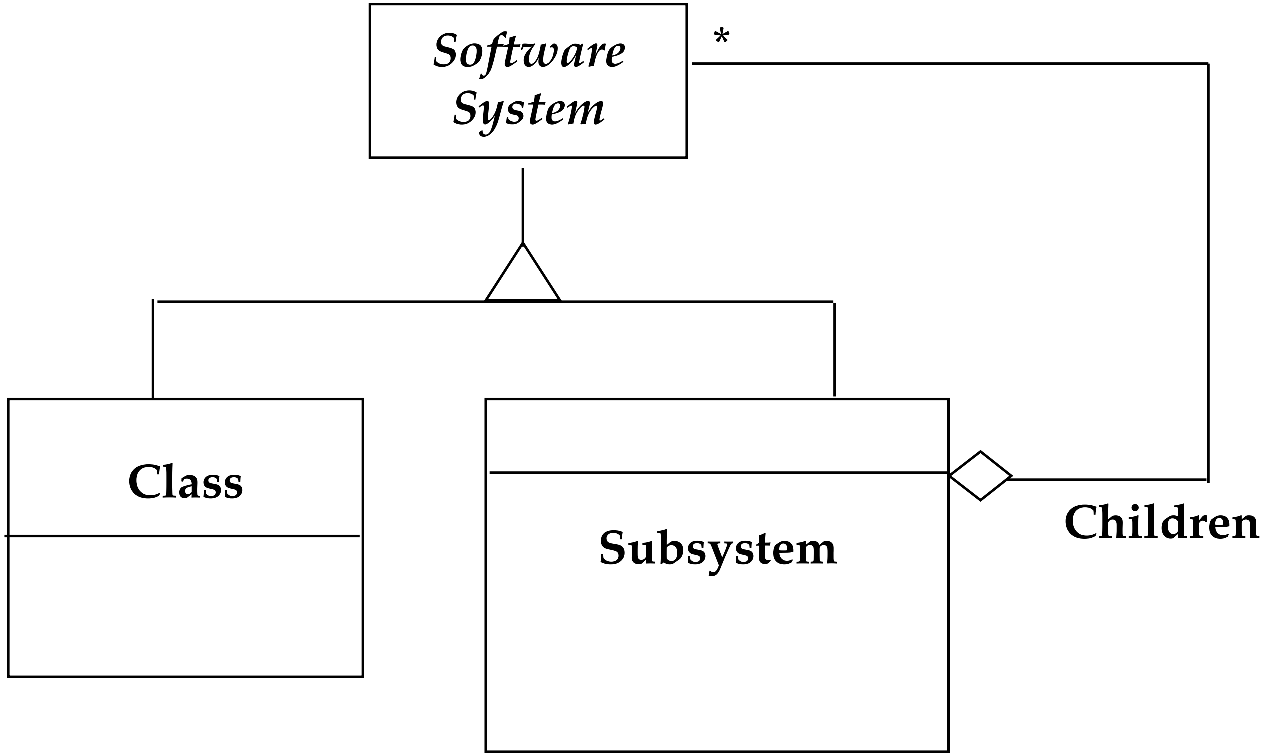
- Recursion
  - An abstraction being defined is used within its own definition
  - More general: Description of an abstraction based on self-similarity.

What is common between these definitions?

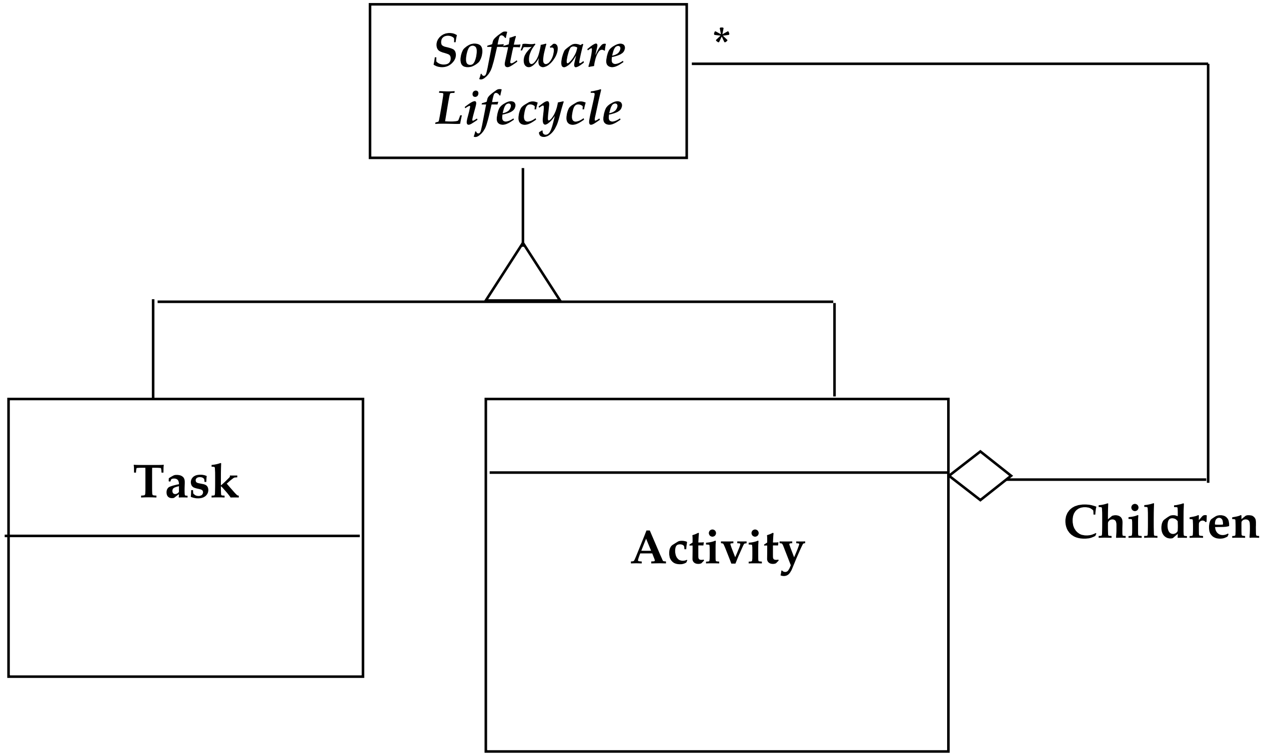
- Definition Software System
  - A software system consists of subsystems which are either other subsystems or collection of classes
  - Composite: Subsystem
    - A software system consists of subsystems which consists of subsystems, which consists of subsystems, which...
  - Base case: Class

- Definition Software Lifecycle
  - 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....
  - Base case: Task.

Modeling a Software System

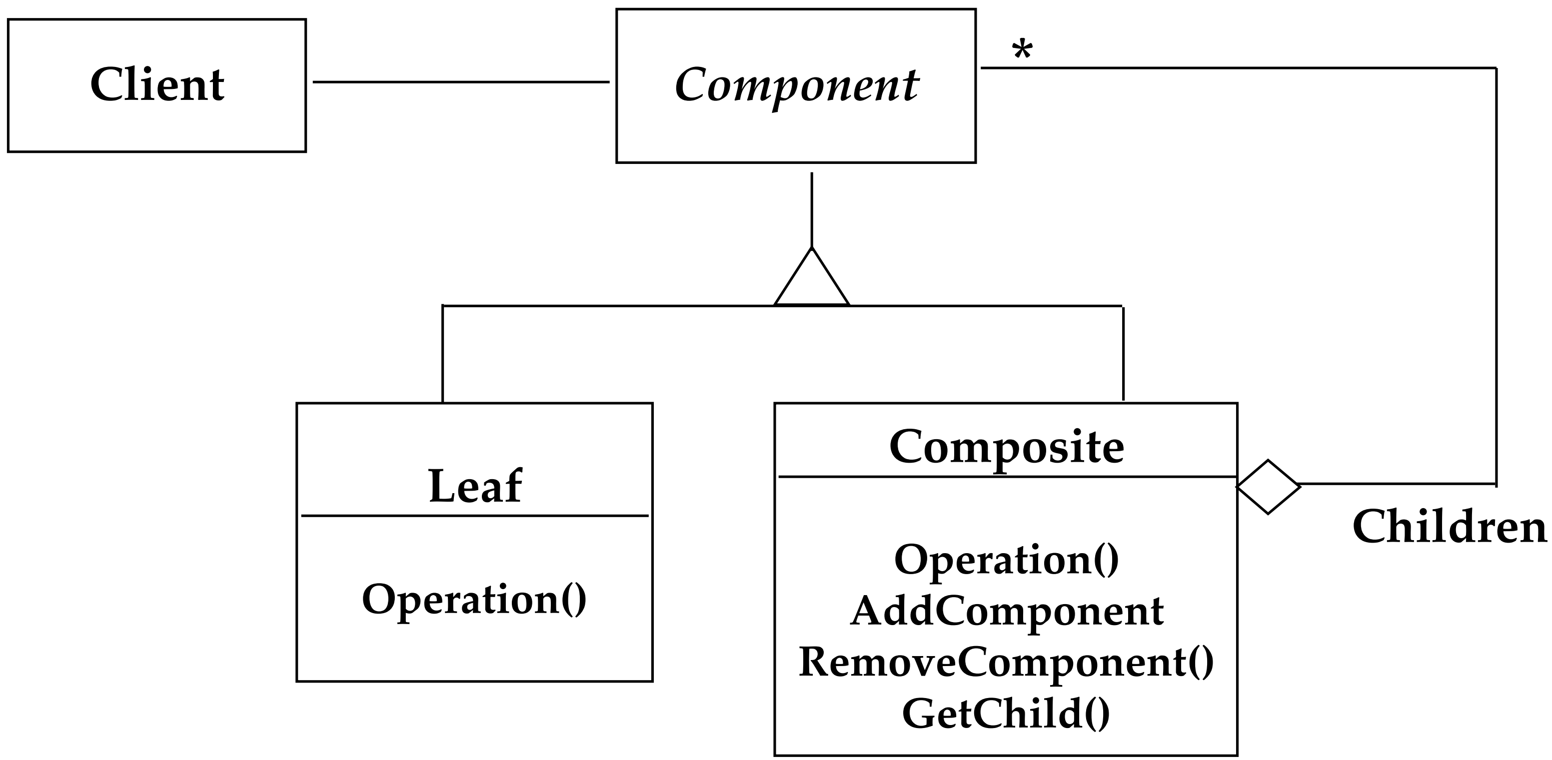


Modeling the Software Lifecycle



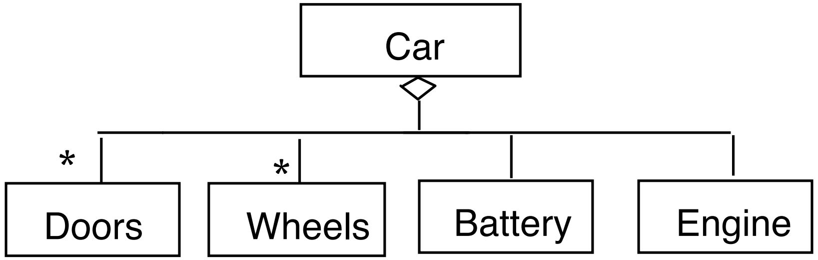
Introducing the Composite Pattern

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

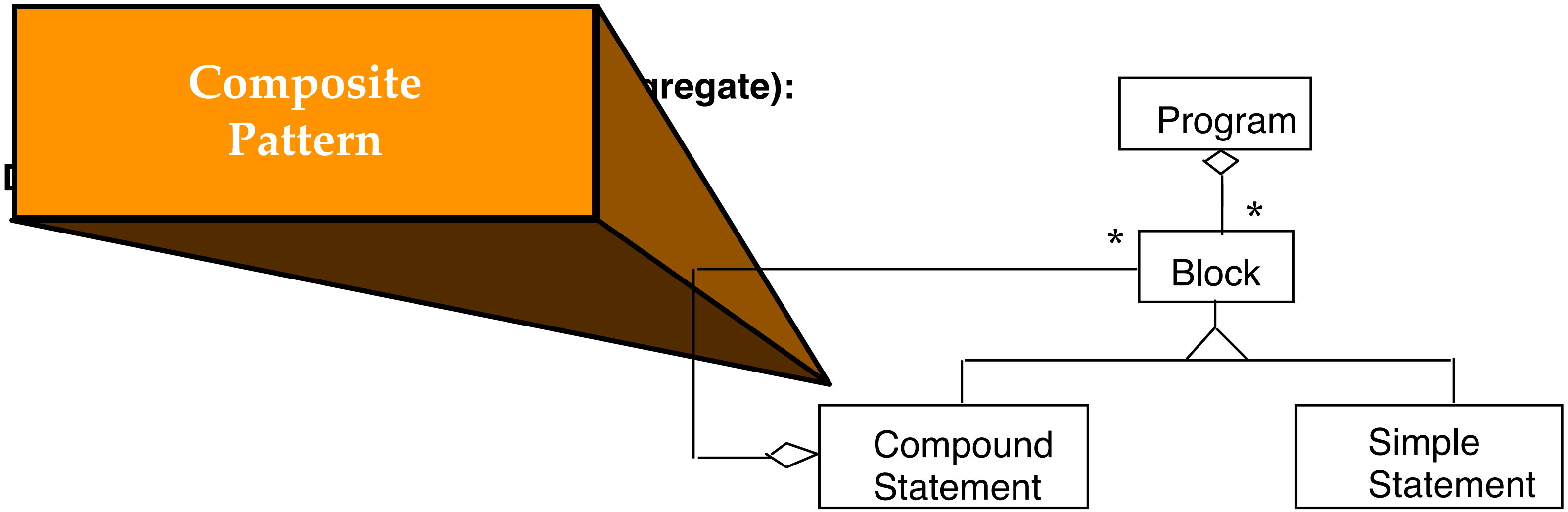


The Composite Patterns models dynamic aggregates

Fixed Structure:

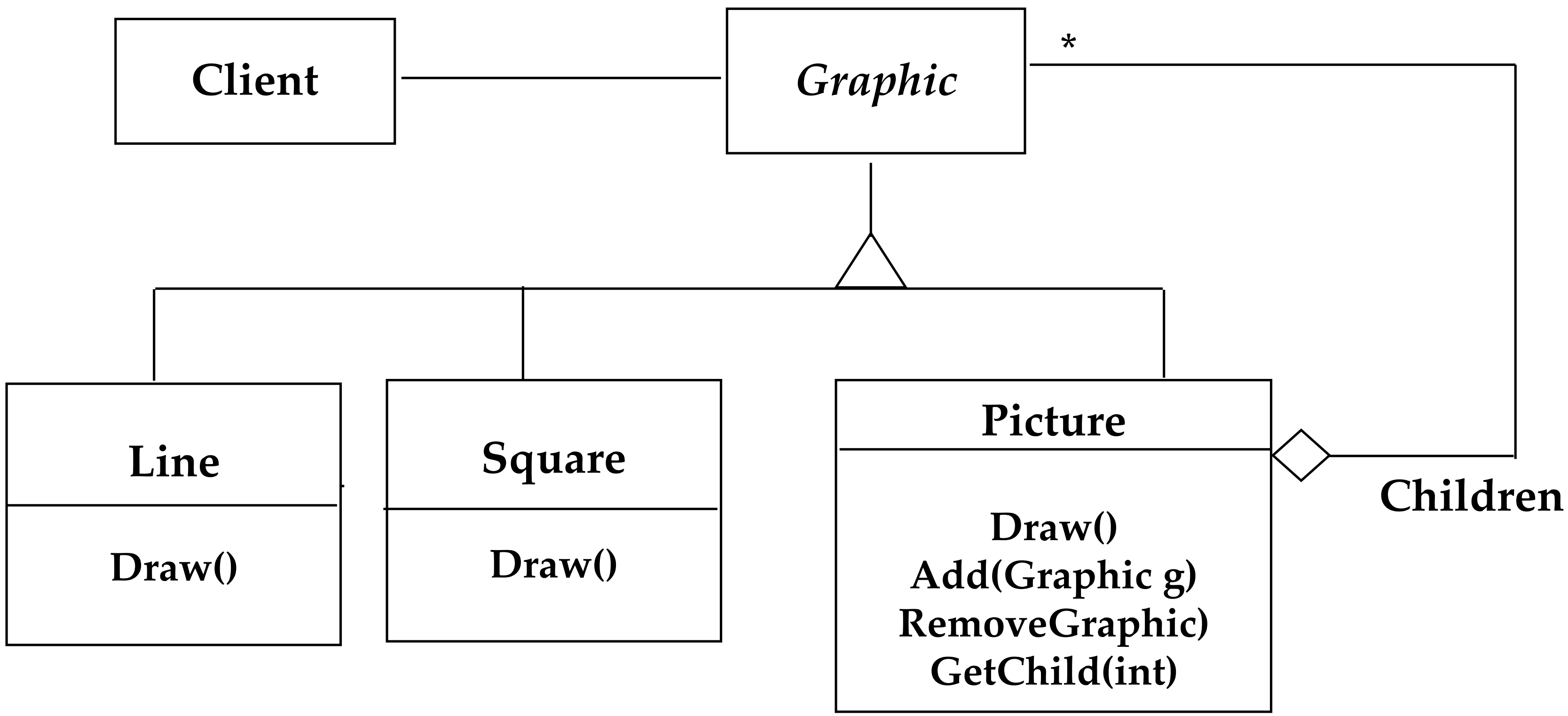


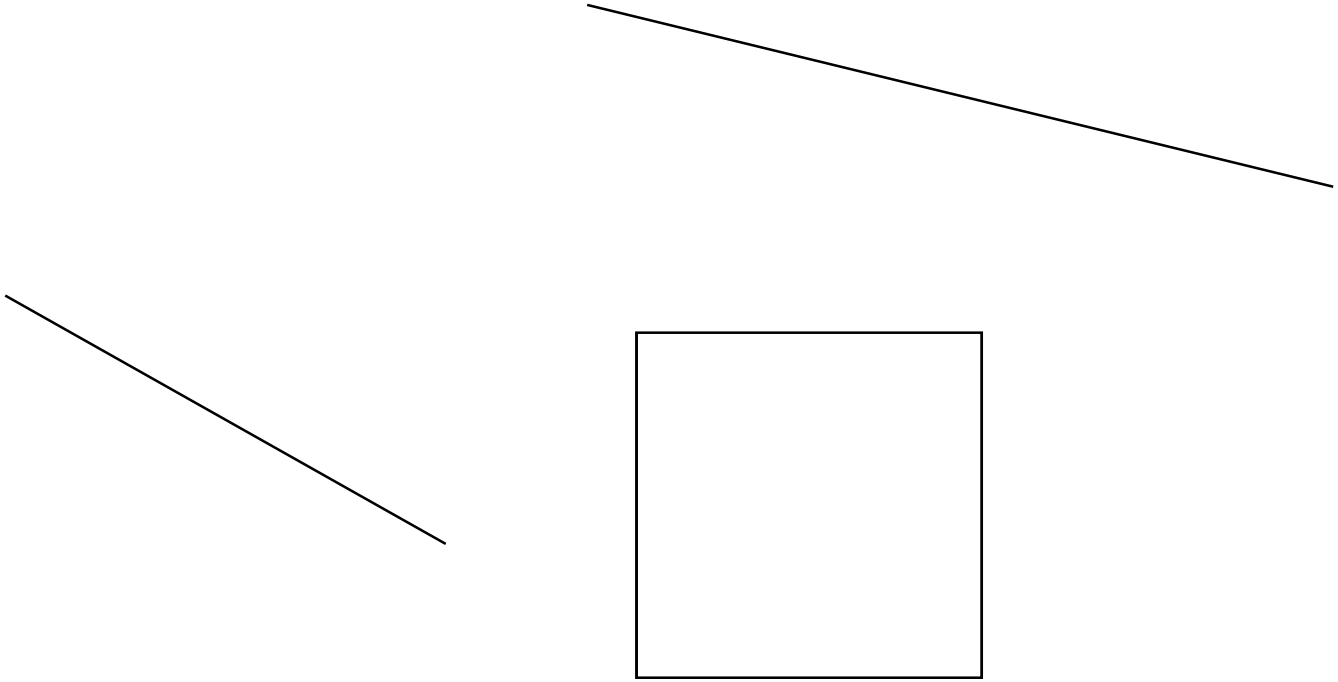
Organization Chart (variable aggregate):



Graphic Applications also Composite Patterns

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

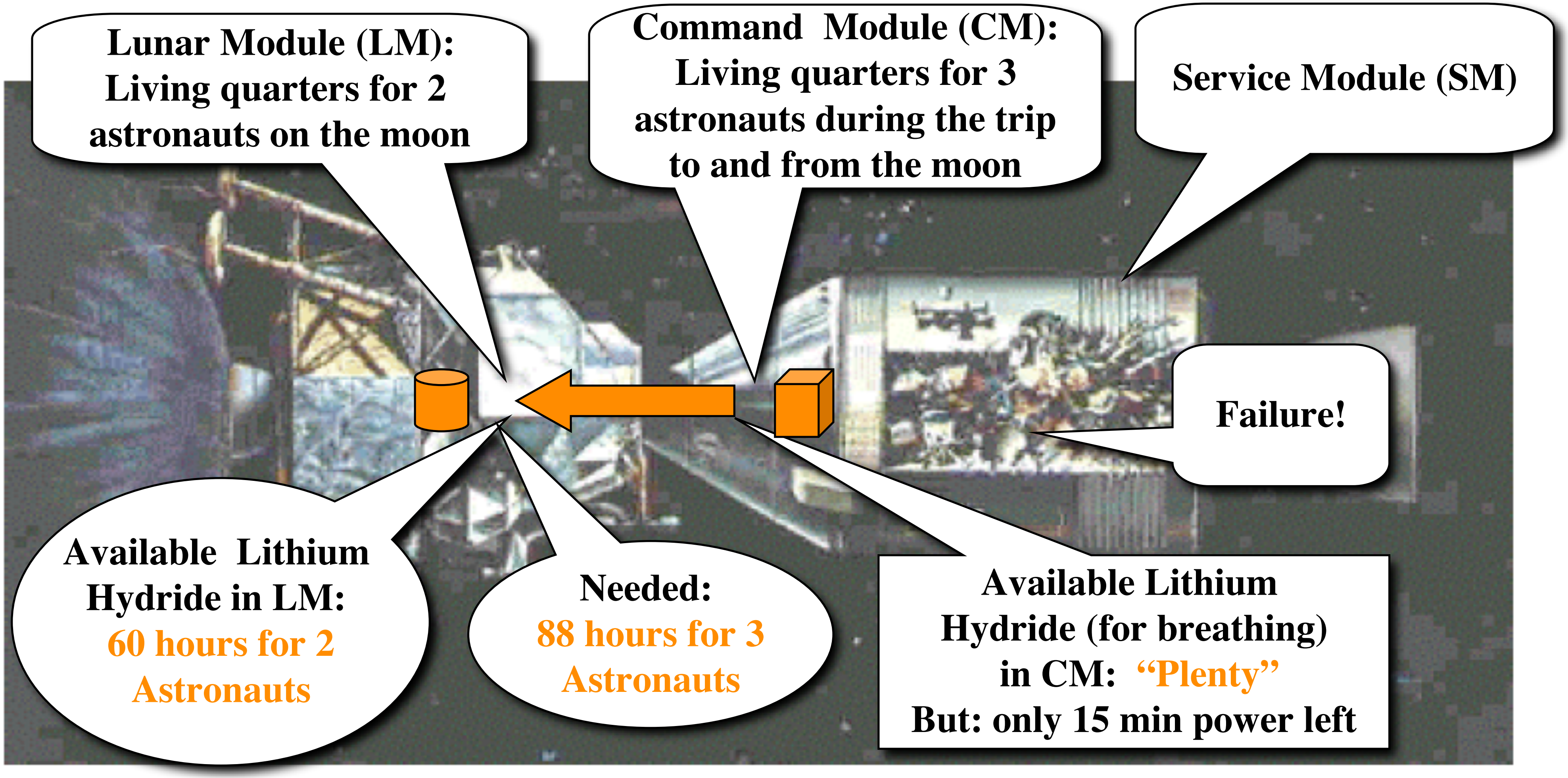




Adapter Pattern

- Adapter Pattern: Converts the interface of a component into another interface expected by the calling component
- 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.

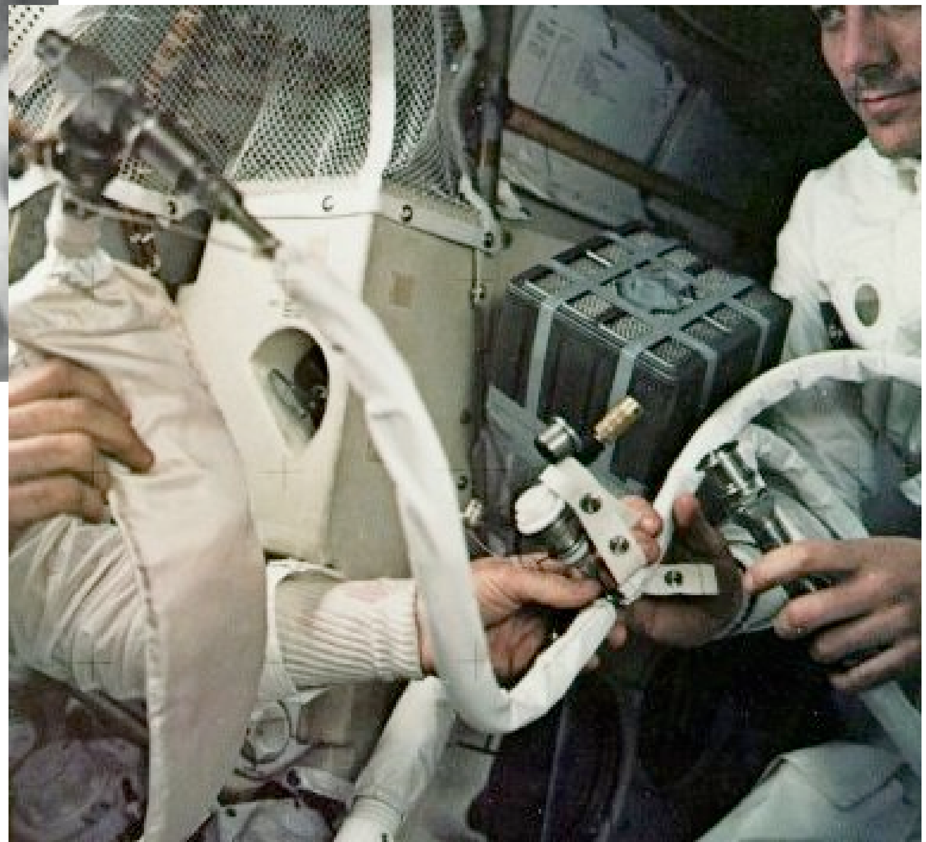
Apollo 13: “Houston, we’ve had a Problem!”



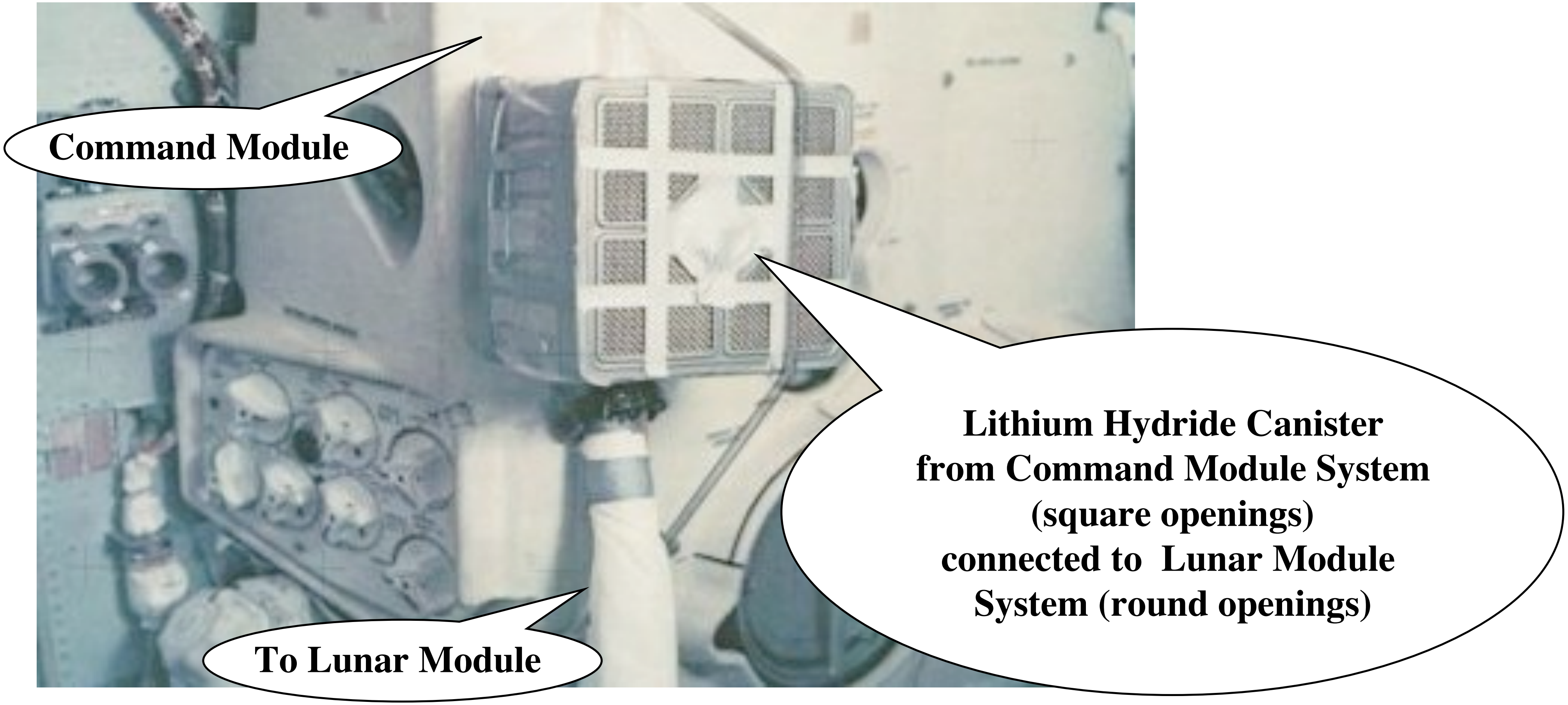
The LM was designed for 60 hours for 2 astronauts (2 days on the moon)
Could its resources be used for 12 man-days (2 1/2 days until reentry)?

Source: http://www1.jsc.nasa.gov/er/seh/apollo13.pdf

Apollo 13: “Fitting a square peg in a round hole”

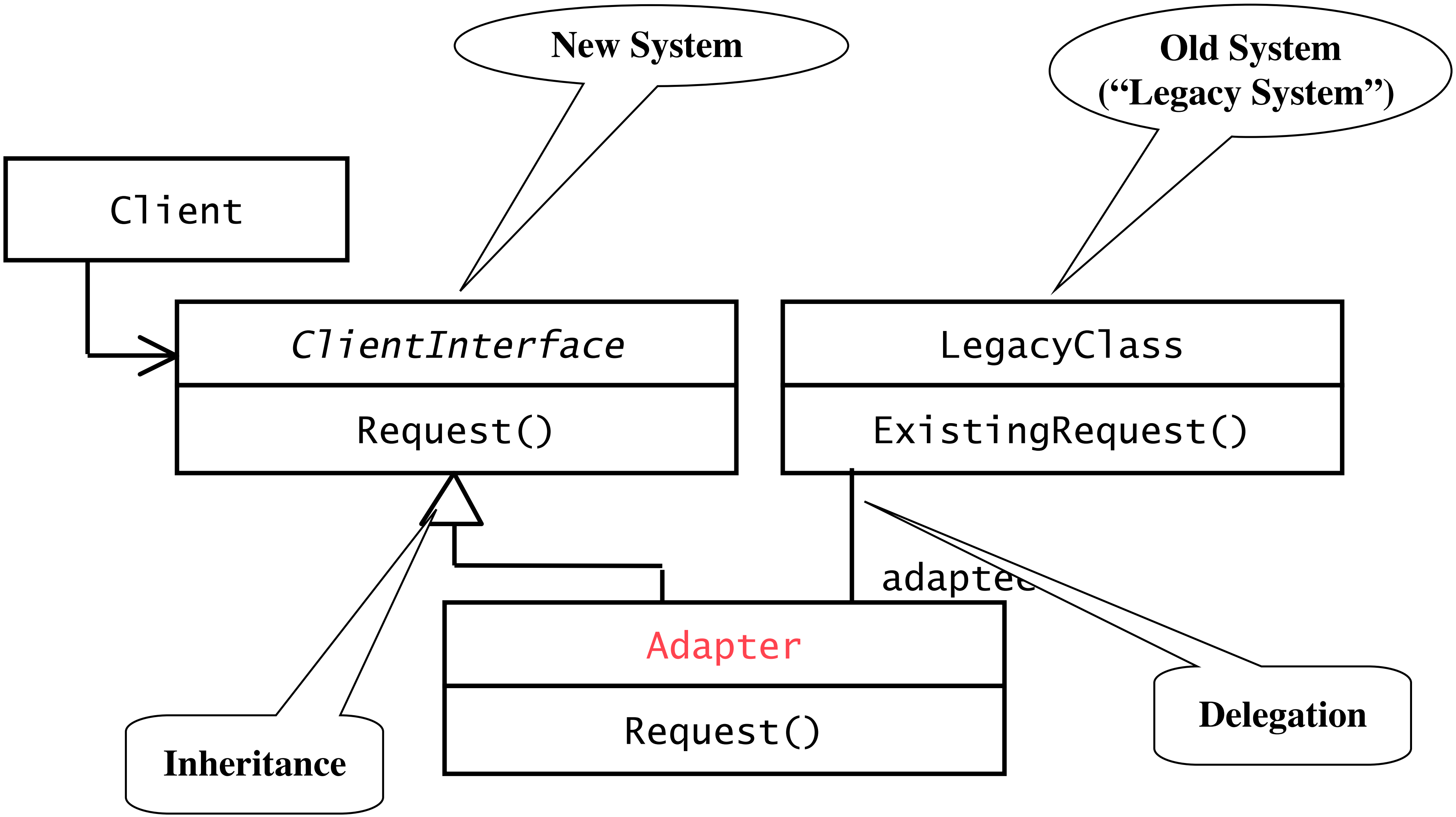


A Typical Object Design Challenge: Connecting Incompatible Components

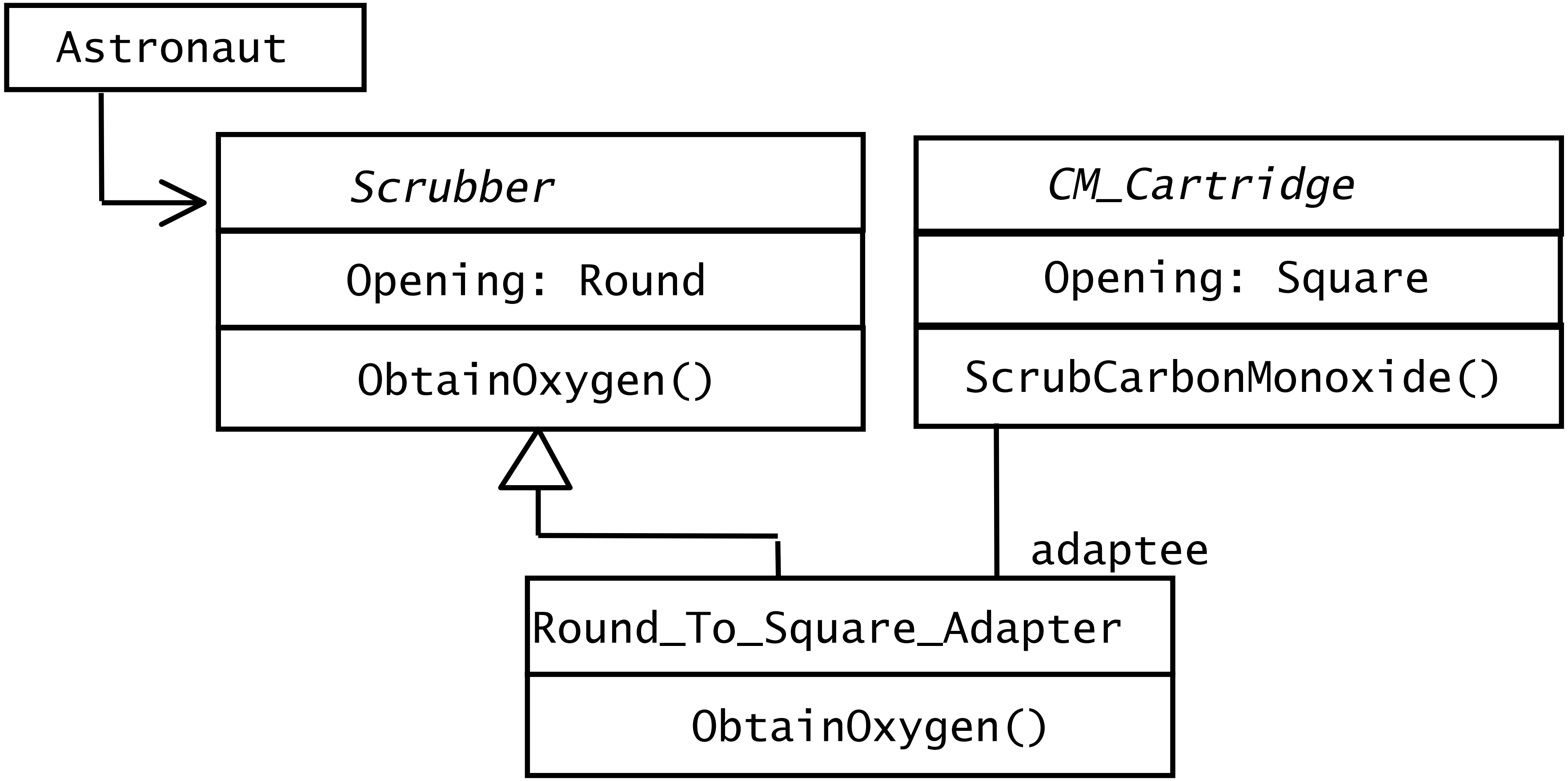


Source: http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html

Adapter Pattern



Adapter for Scrubber in Lunar Module



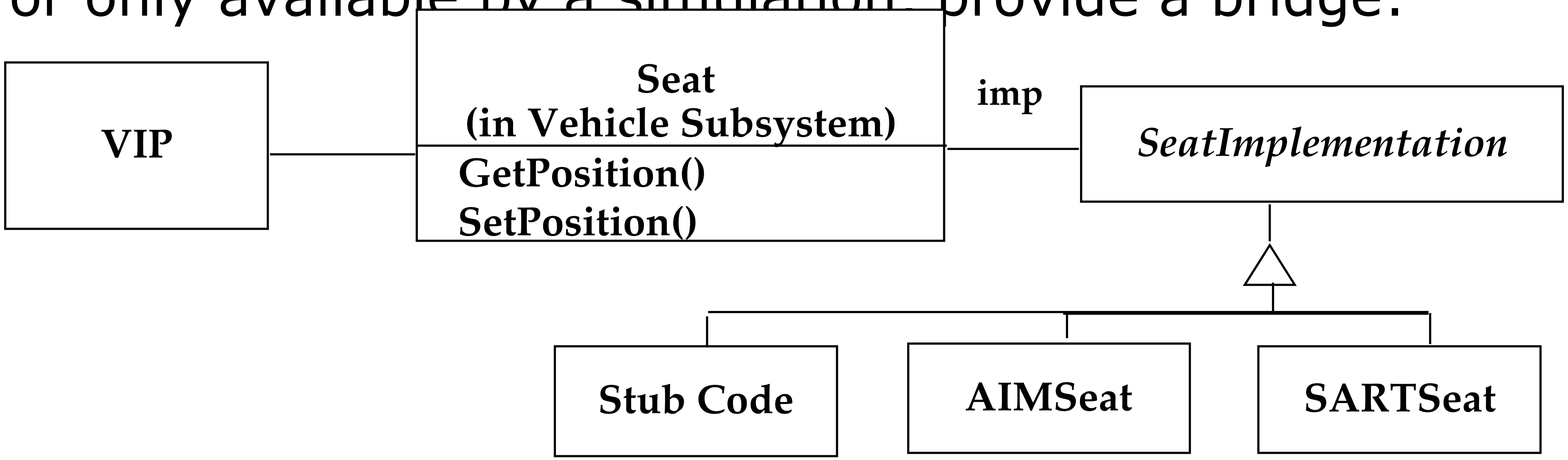
- Using a carbon monoxide scrubber (round opening) in the lunar module with square cartridges from the command module (square opening)

Motivation for the Bridge Pattern

- Decouple 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

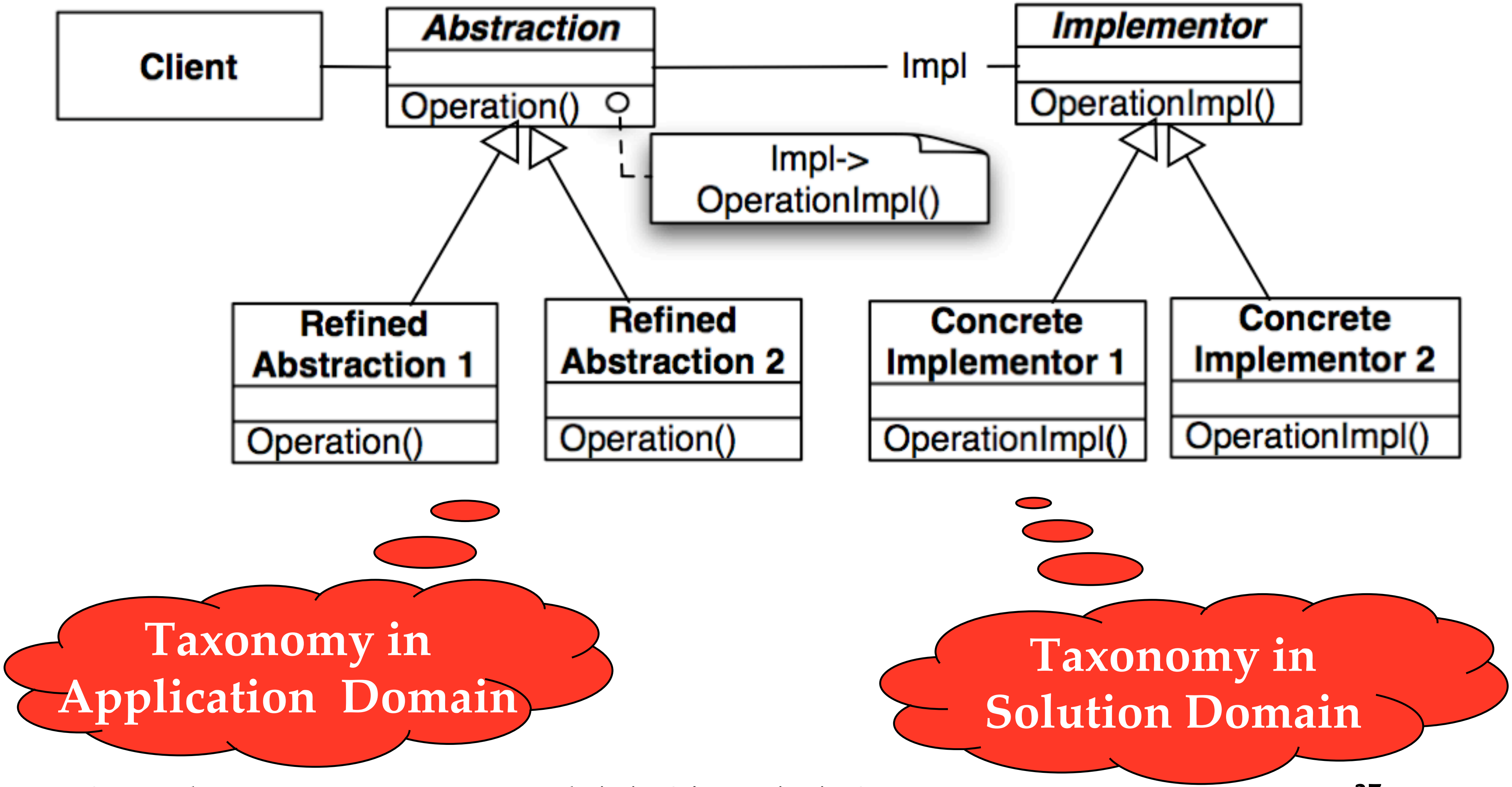
- The bridge pattern is used to provide multiple implementations under the same interface.
- Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- Example Smardcard Project: if seat data is required to be read, but the seat is not yet implemented, known, or only available by a simulation, provide a bridge:



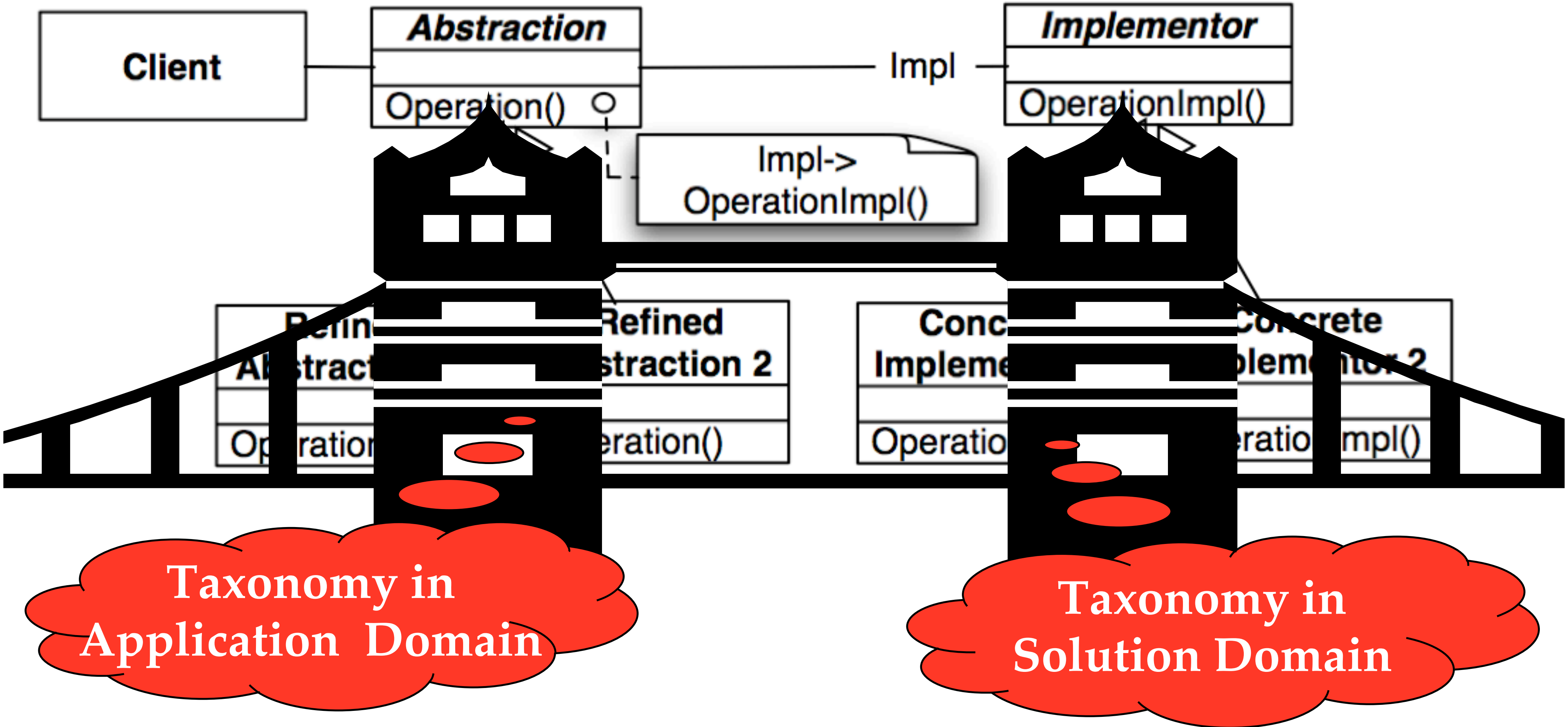
Seat Implementation

```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class Stubcode implements SeatImplementation {
  public int GetPosition() {
    // stub code for GetPosition
  }
  ...
}
public class AimSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the AIM simulation system
  }
  ….
}
public class SARTSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the SART seat simulator
 }
  ...
}
```

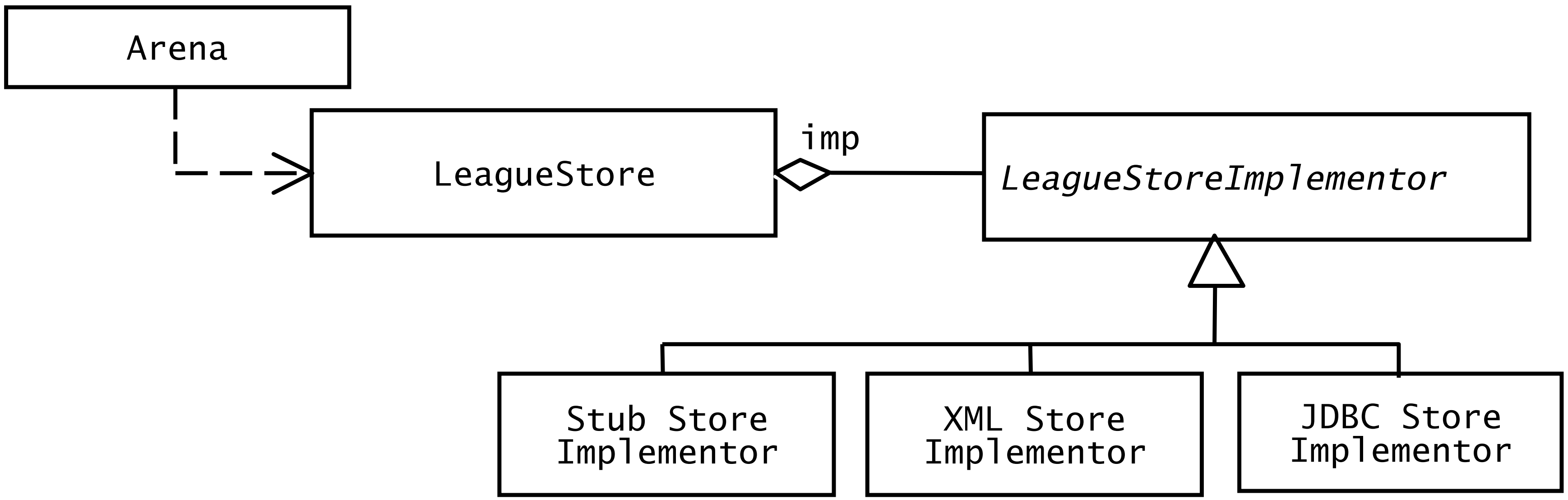
Bridge Pattern



Why the Name Bridge Pattern?



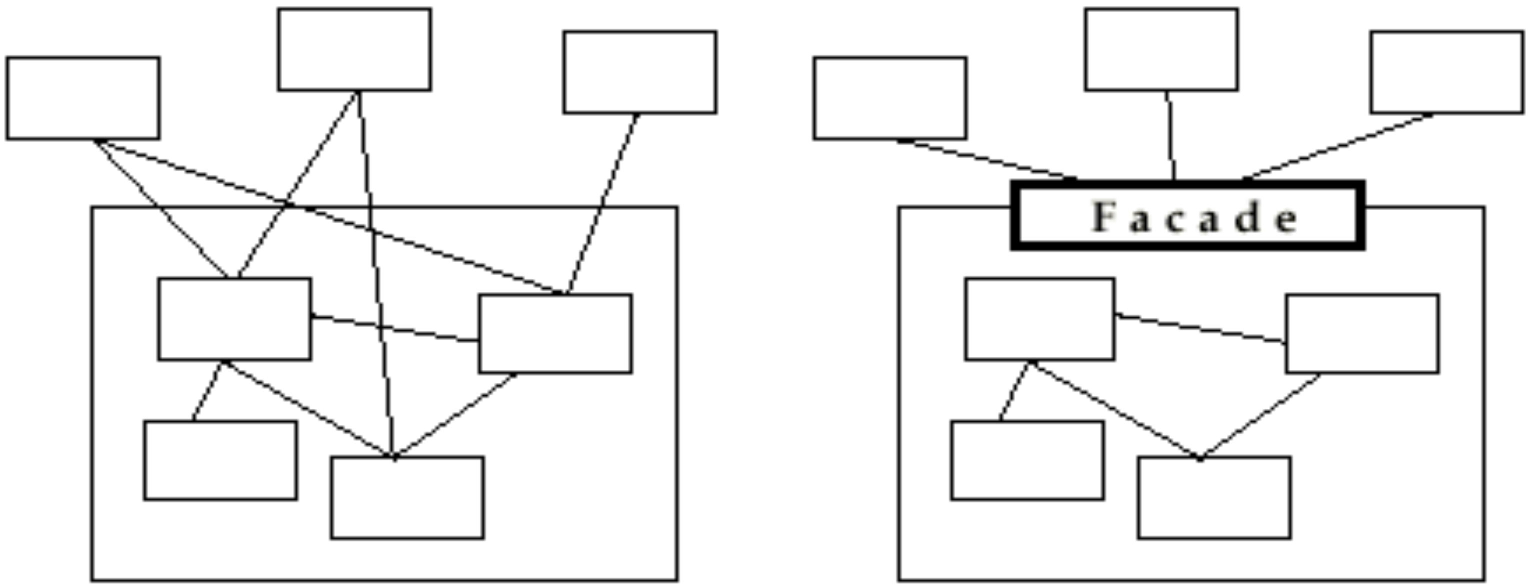
Using the Bridge Pattern to support multiple Database Vendors



Adapter vs Bridge

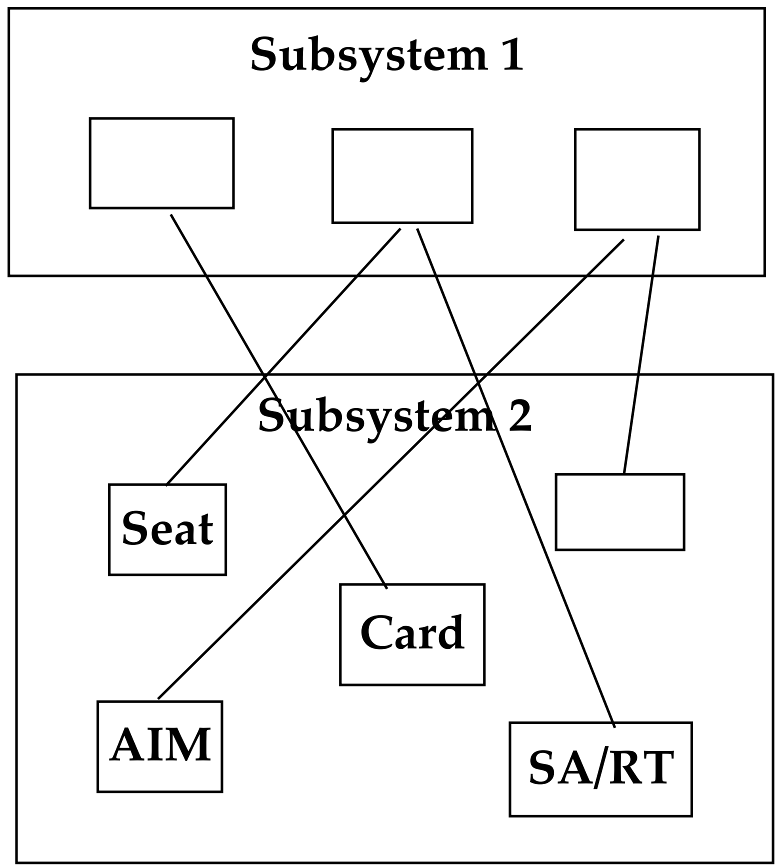
- Similarities:
  - Both are used to hide the details of the underlying implementation.
- Difference:
  - The adapter pattern is geared towards making unrelated components work together
    - Applied to systems after they're designed (reengineering, interface engineering).
    - "Inheritance followed 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"

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)
- 

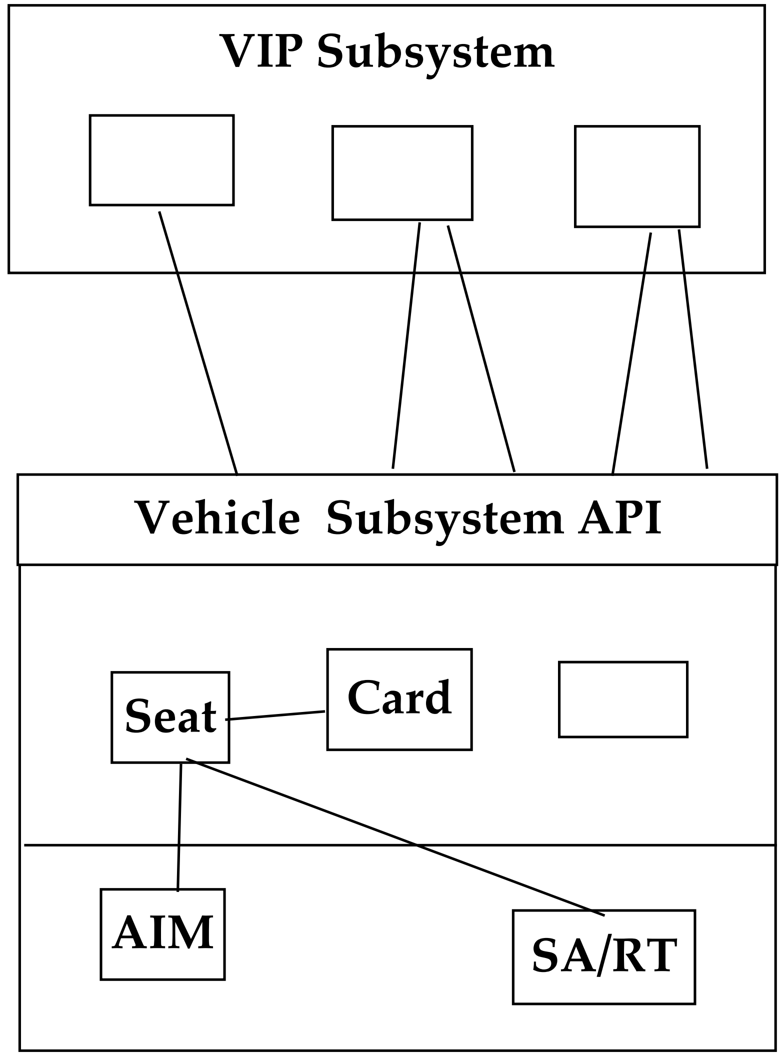
Design Example

- Subsystem 1 can look into the Subsystem 2 and call any class operation at will
- This is “Ravioli Design”
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can’t expect the calling subsystem to understand how the called subsystem works or the complex relationships within the subsystem.
  - We can be assured that the access to subsystem 2 will be misused, leading to non-portable code.



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



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: Facade
  - 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?

- The façade design pattern should be used by all subsystems in a software system. The façade defines the services of a subsystem
  - The facade will delegate requests to the appropriate components within the subsystem. Most of the time the façade does not need to be changed, when the component is changed
- The adapter design pattern should be used to interface to existing components
  - For example, a smart card software system should provide an adapter for a smart card reader from a particular manufacturer
- The bridge design pattern should be used to interface to a set of objects
  - where the full set is not completely known at analysis or design time.
  - when the subsystem must be extended later after the system has been deployed and client programs are in the field.

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).

Recall: Why are reusable Designs important?

A design…

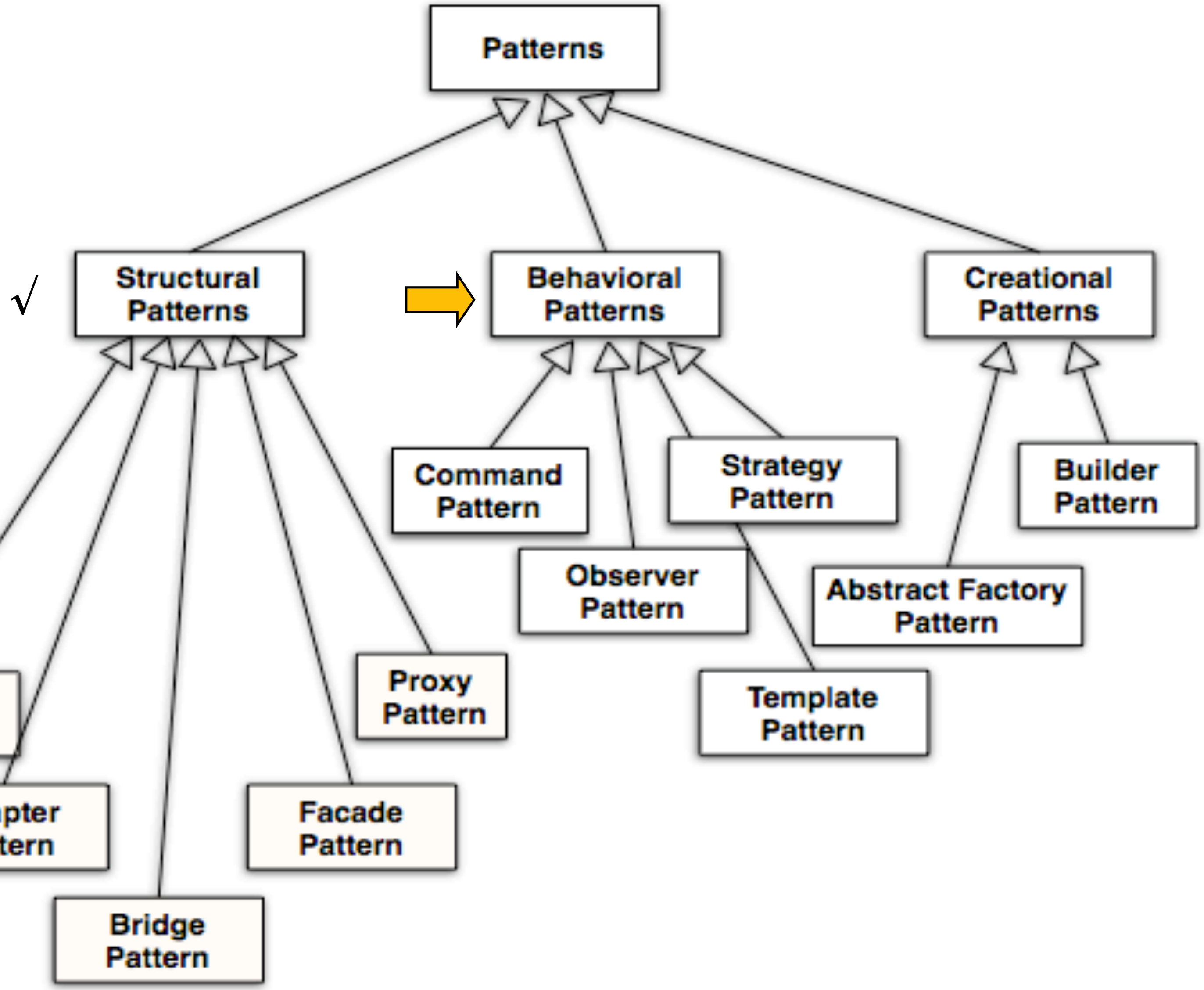
…enables flexibility to change (Reusability)

…minimizes the introduction of new problems when fixing old ones

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

The Proxy Pattern is a reusable design

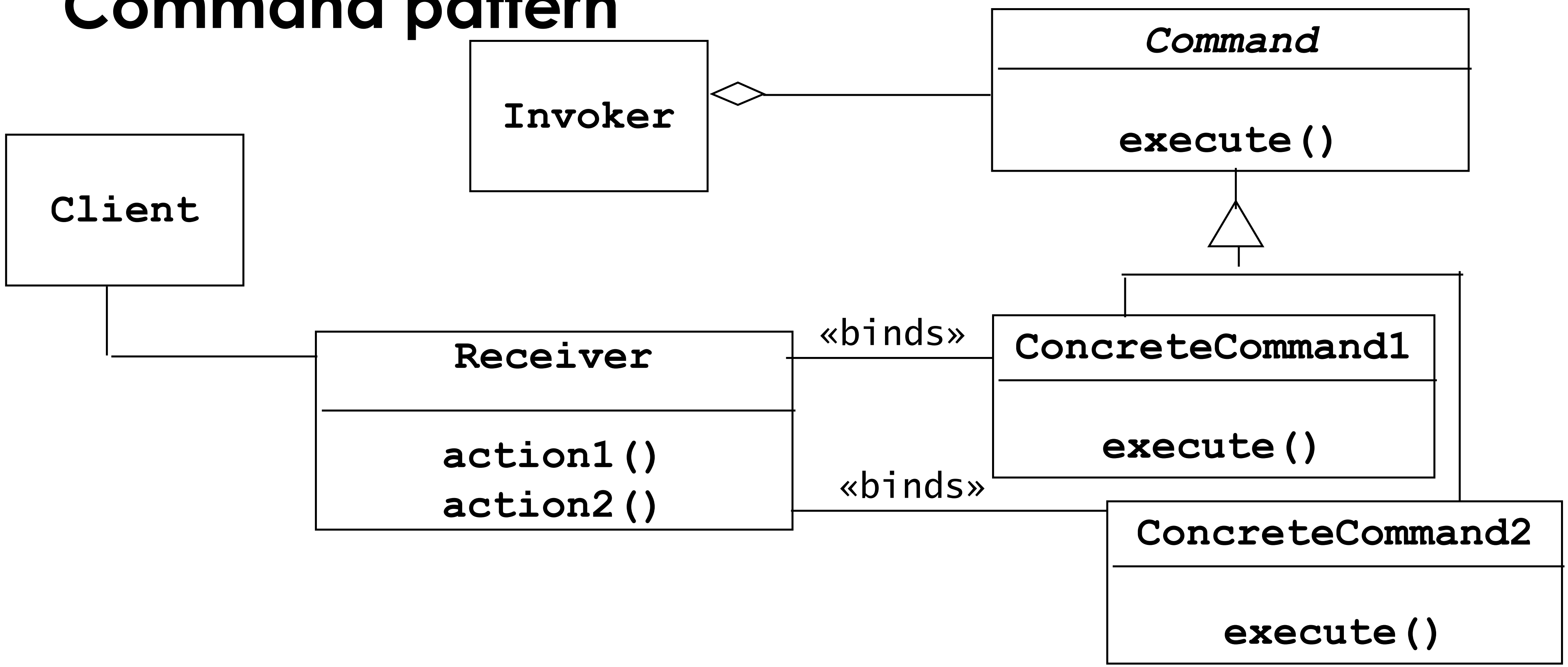
- Caching of information ("Remote Proxy")
  - The Proxy object is a local representative for an object in a different address space
  - Good if information does not change too often
- Standin ("Virtual Proxy")
  - Object is too expensive to create or too expensive to download.
  - Good if the real object 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.


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

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 theconcreteCommand based on the string entered and the binding between action and ConcreteCommand.

Decouples boundary objects from control objects

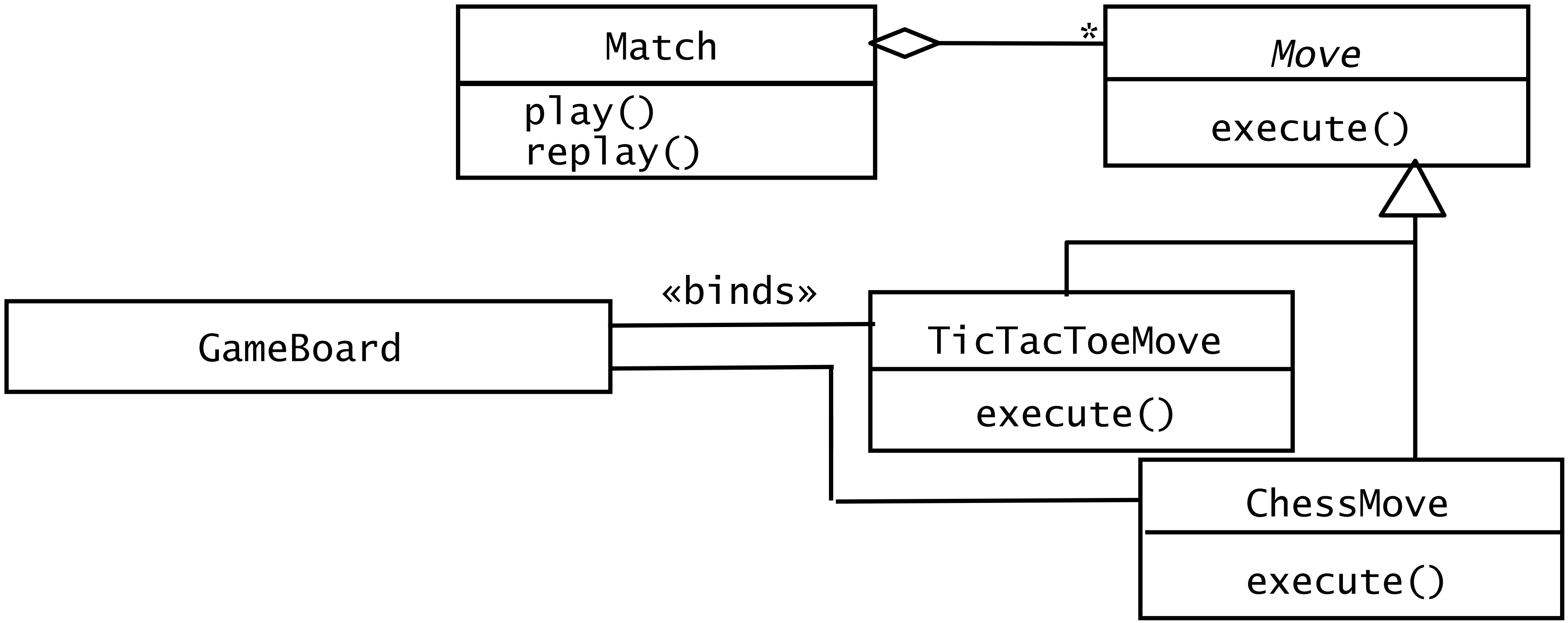
- The command pattern can be nicely used to decouple boundary objects from control objects:
  - Boundary objects such as menu items and buttons, send messages to the command objects (I.e. the control objects)
  - Only the command objects modify entity objects
- When the user interface is changed (for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

Command Pattern Applicability

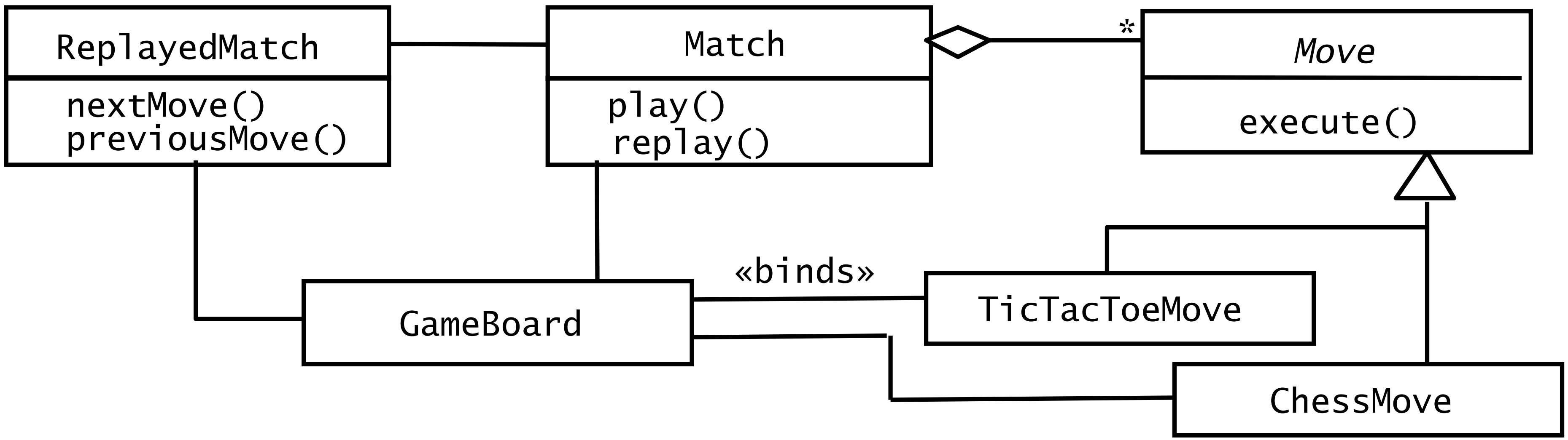
- Parameterize clients with different requests
- Queue or log requests
- Support undoable operations

- Uses:
  - Undo queues
  - Database transaction buffering

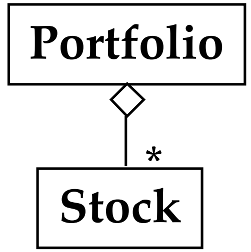
Applying the Command Pattern to Command Sets



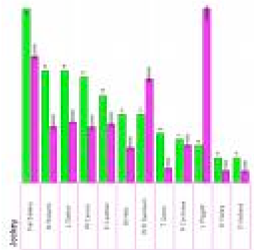
Applying the Command design pattern to Replay Matches in ARENA

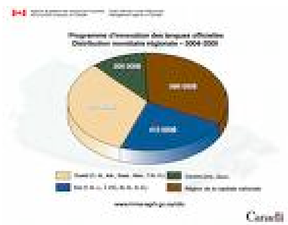


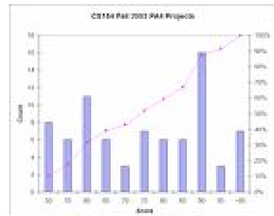
Observer Pattern Motivation 5 16 2007



- 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.



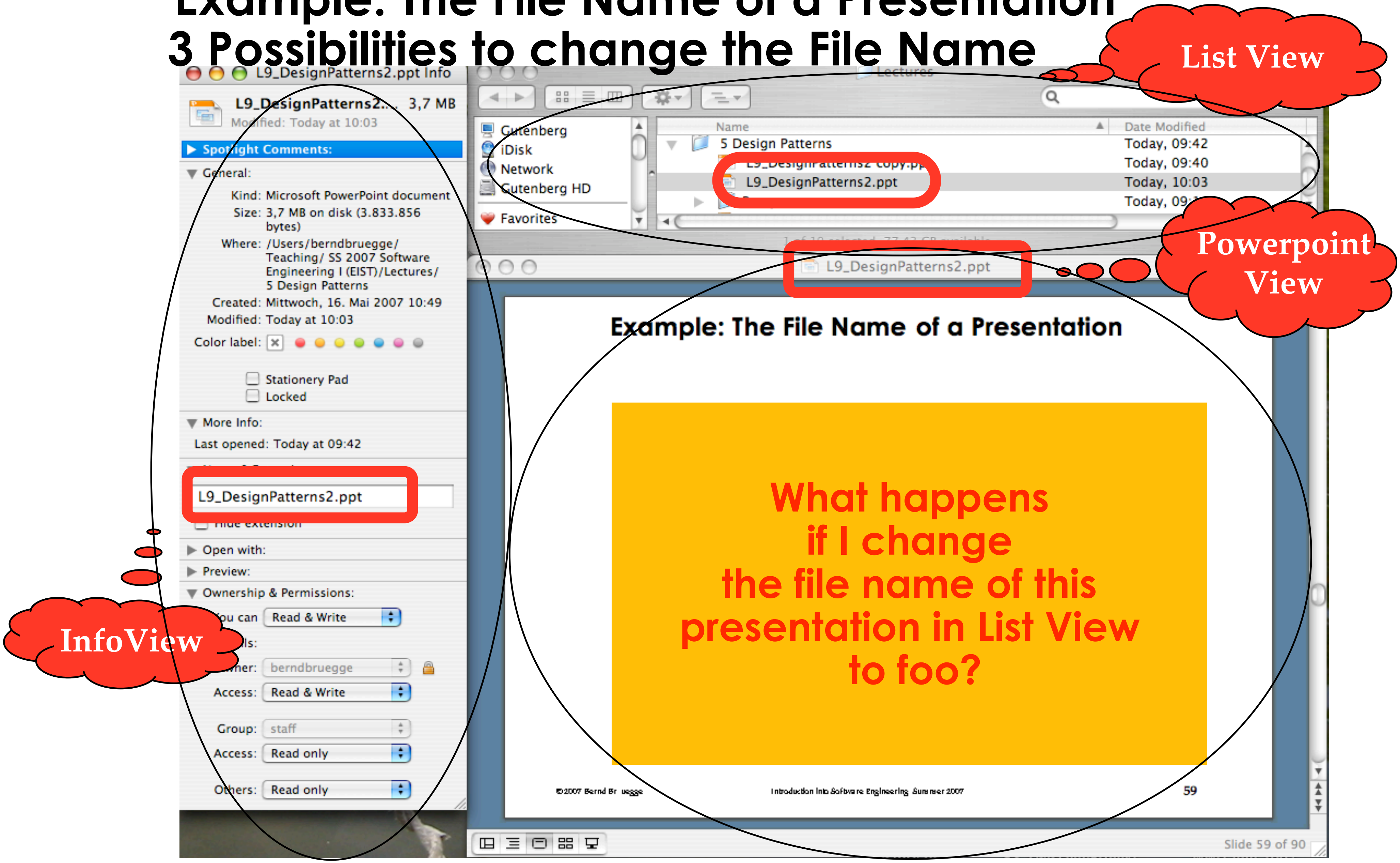




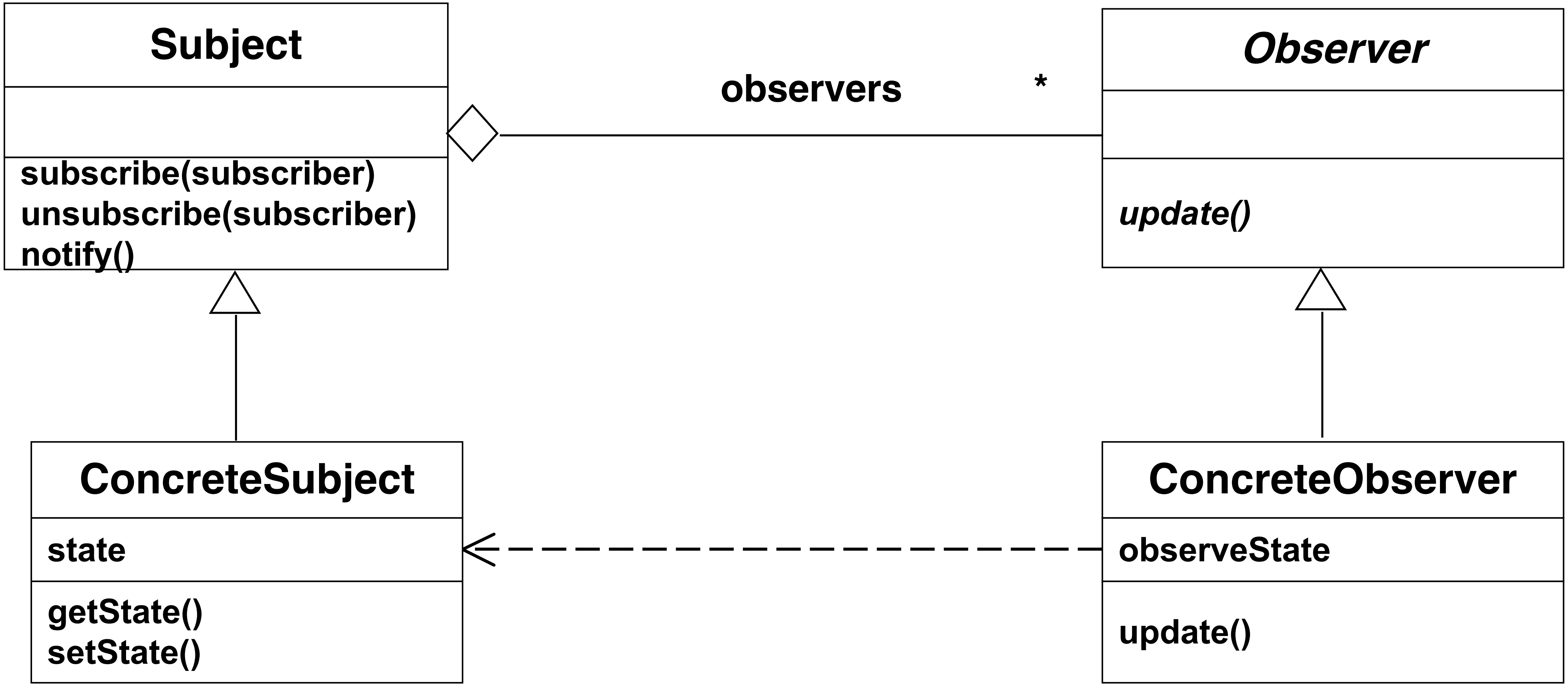
Miscellaneous Announcements

1. Next week
   - Monday is a holiday
   - No lecture on Tuesday
   - No exercises next week
2. Lecture on Wednesday as planned!
3. Mid-term
   - Time: 2 June 2007
   - Optional
   - If you want to participate in the midterm, you have to register with the „Grundstudiumstool".

Example: The File Name of a Presentation
3 Possibilities to change the File Name



Observer Pattern: Decouples an Abstraction from its Views

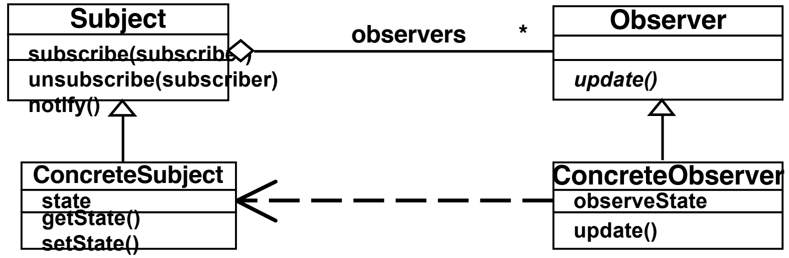


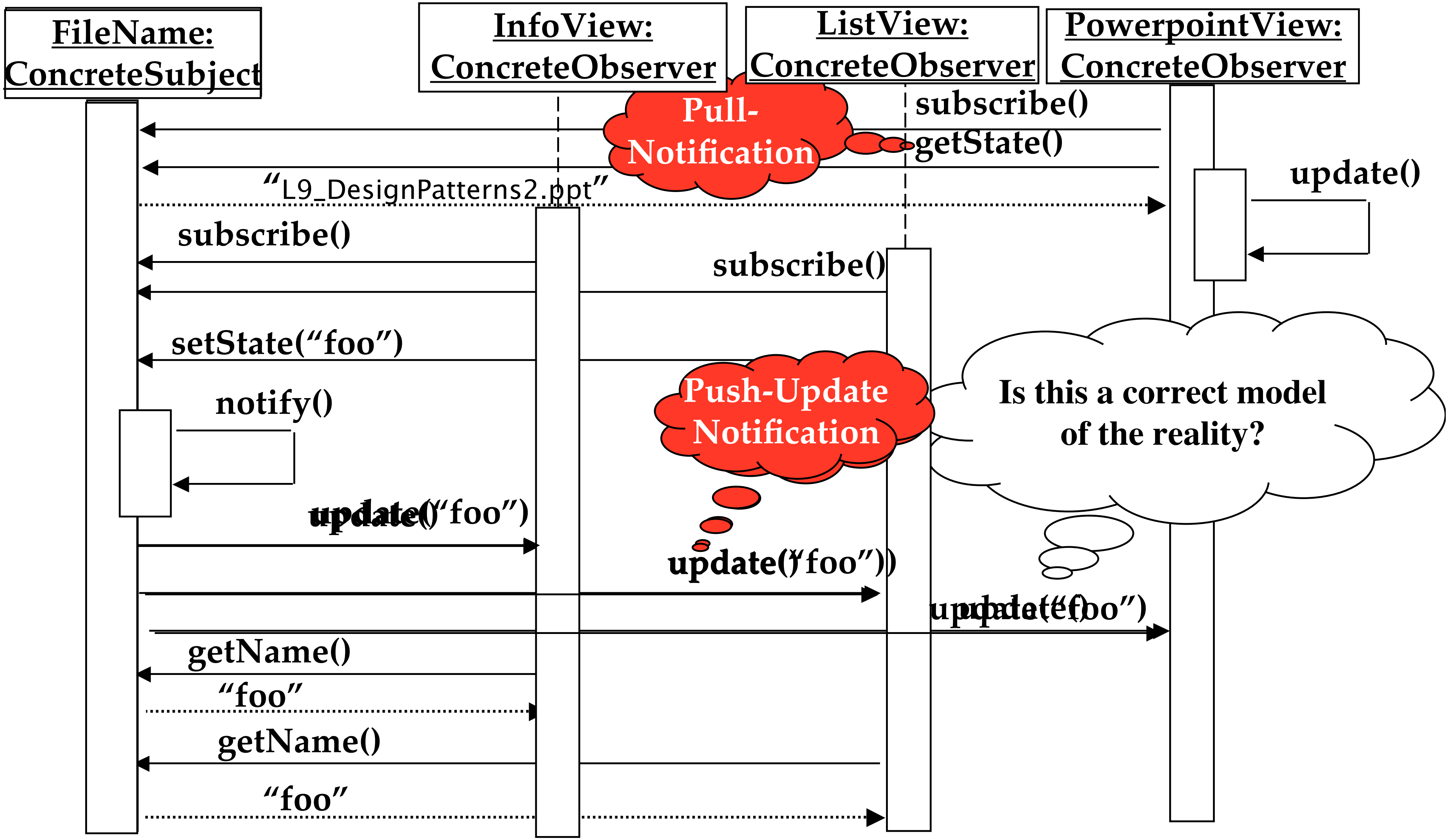
- The **Subject** ("Publisher") represents the entity object
- **Observers** ("Subscribers") attach to the Subject by calling **subscribe()**
- Each Observer has a different view of the state of the entity object
  - The **state** is contained in the subclass **ConcreteSubject**
  - The state can be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

Observer Pattern

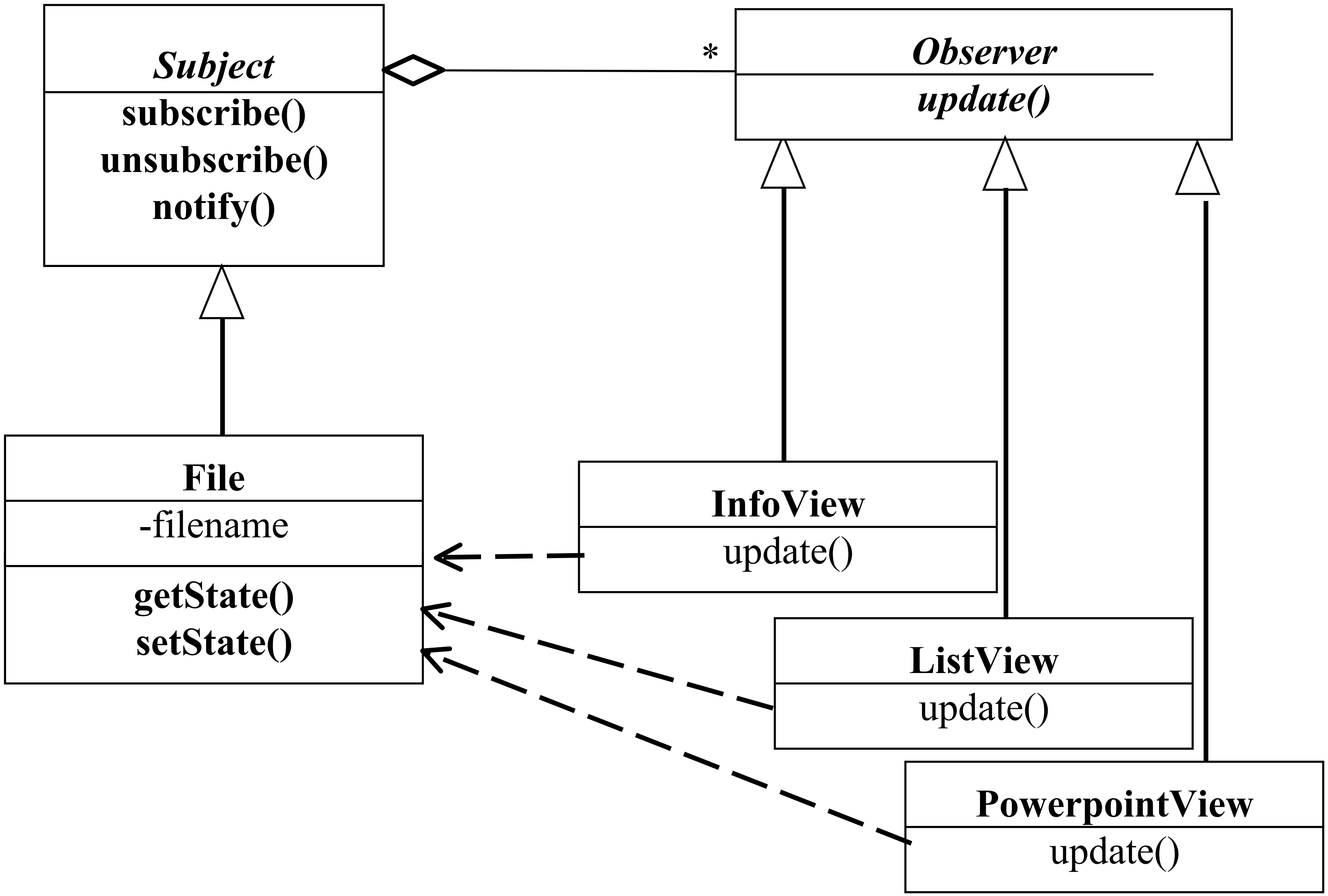
- Models a 1-to-many dependency between objects
  - Connects the state of an observed object, the subject with many observing objects, the observers
- Usage:
  - Maintaining consistency across redundant states
  - Optimizing a batch of changes to maintain consistency
- Three variants for maintaining 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”

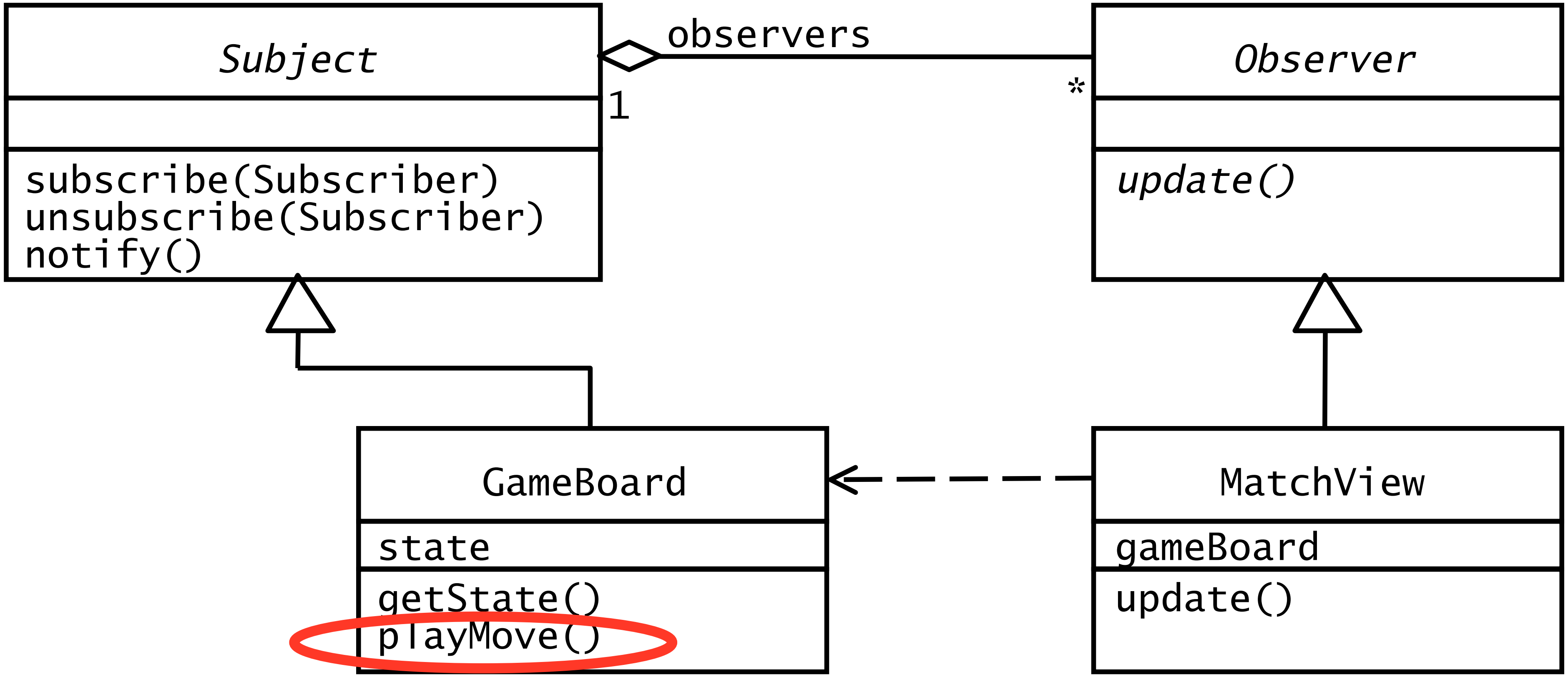




Applying the Observer Pattern to maintain Consistency across Views



Applying the Observer Design Pattern to maintain Consistency across MatchViews

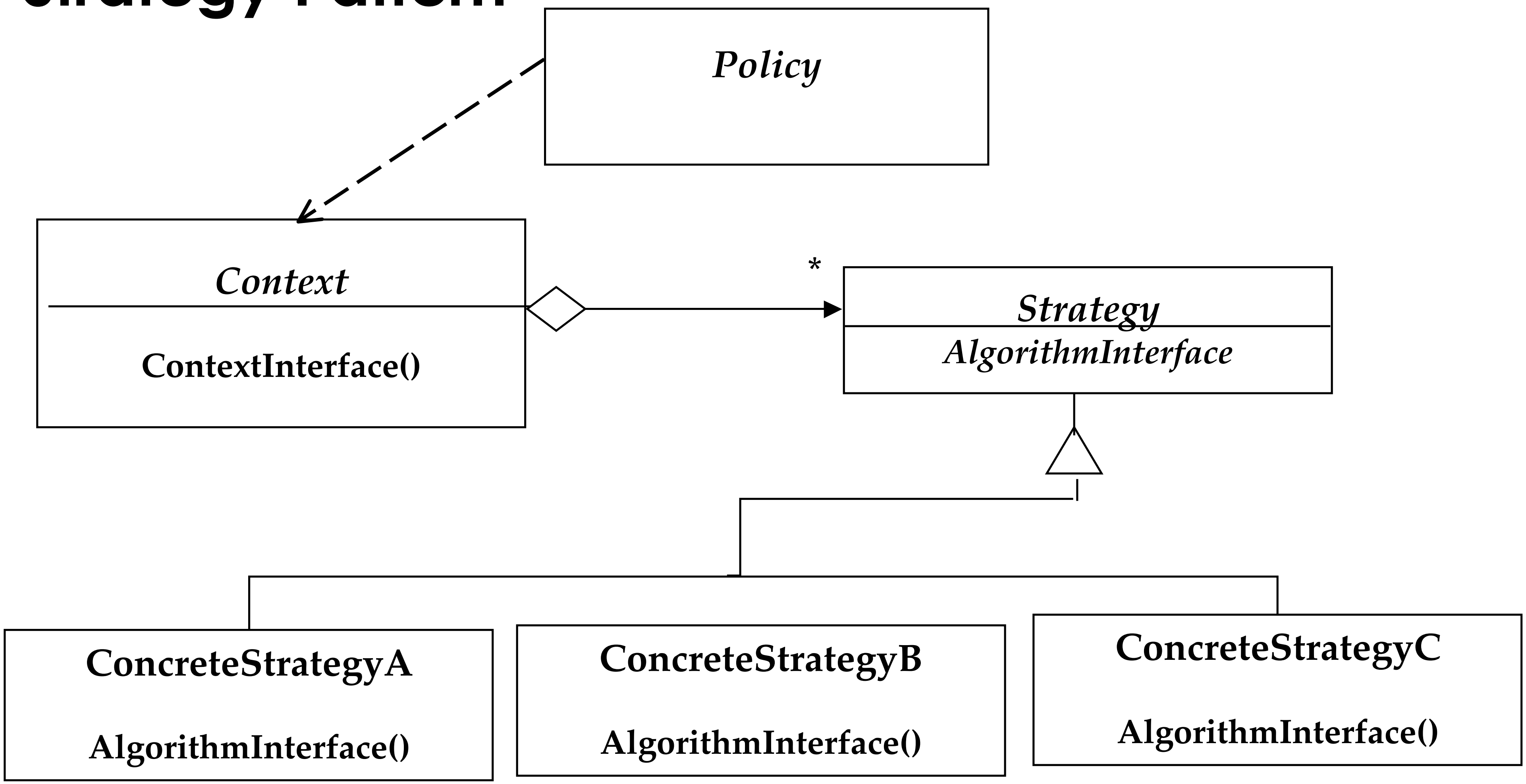


Push, Pull or Push-Update Notification?

Strategy Pattern

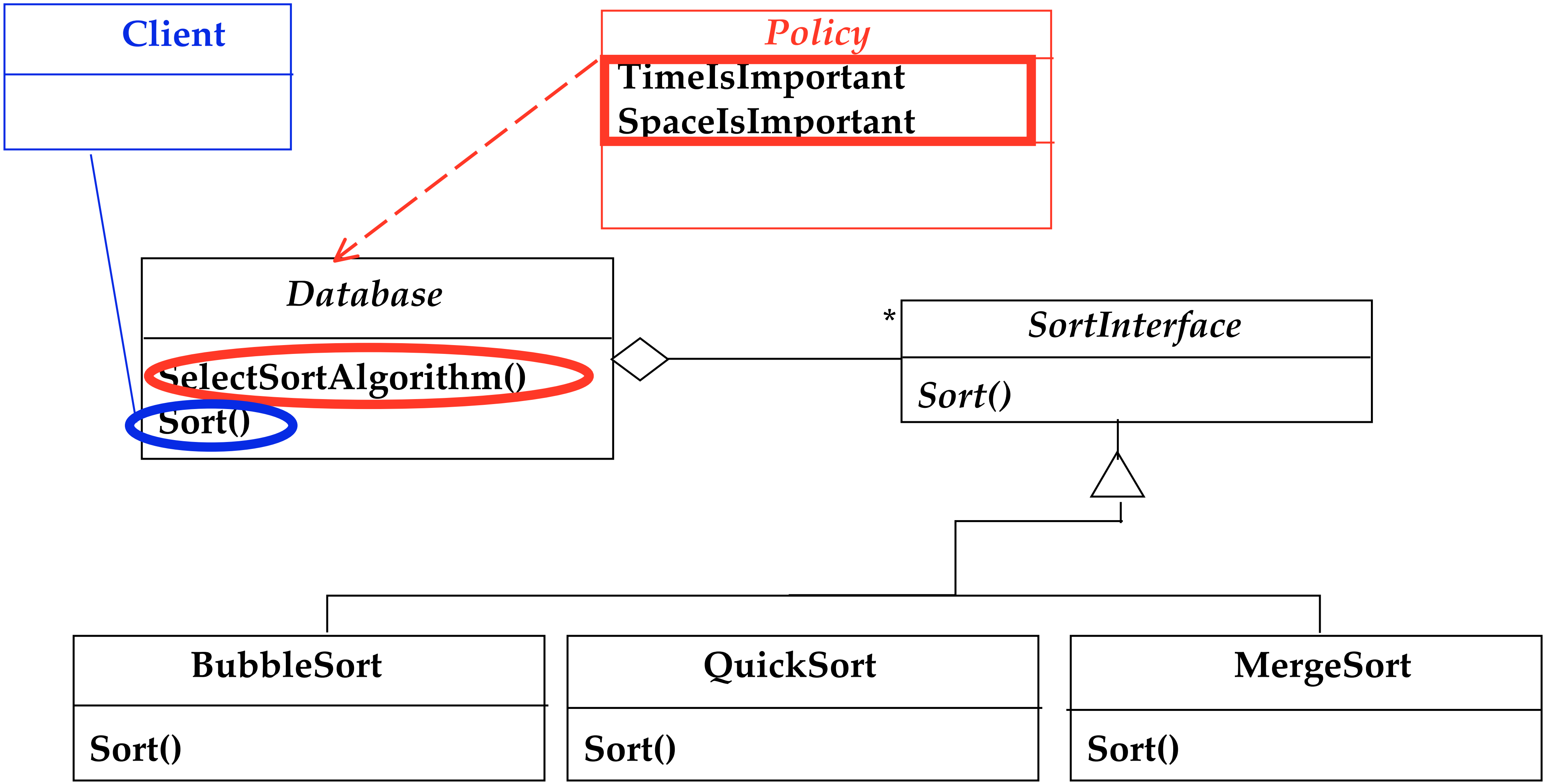
- Different algorithms exists for a specific task
  - We can switch between the algorithms at run time
- Examples of tasks:
  - Different collision strategies for objects in video games
  - Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
  - Sorting a list of customers (Bubble 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

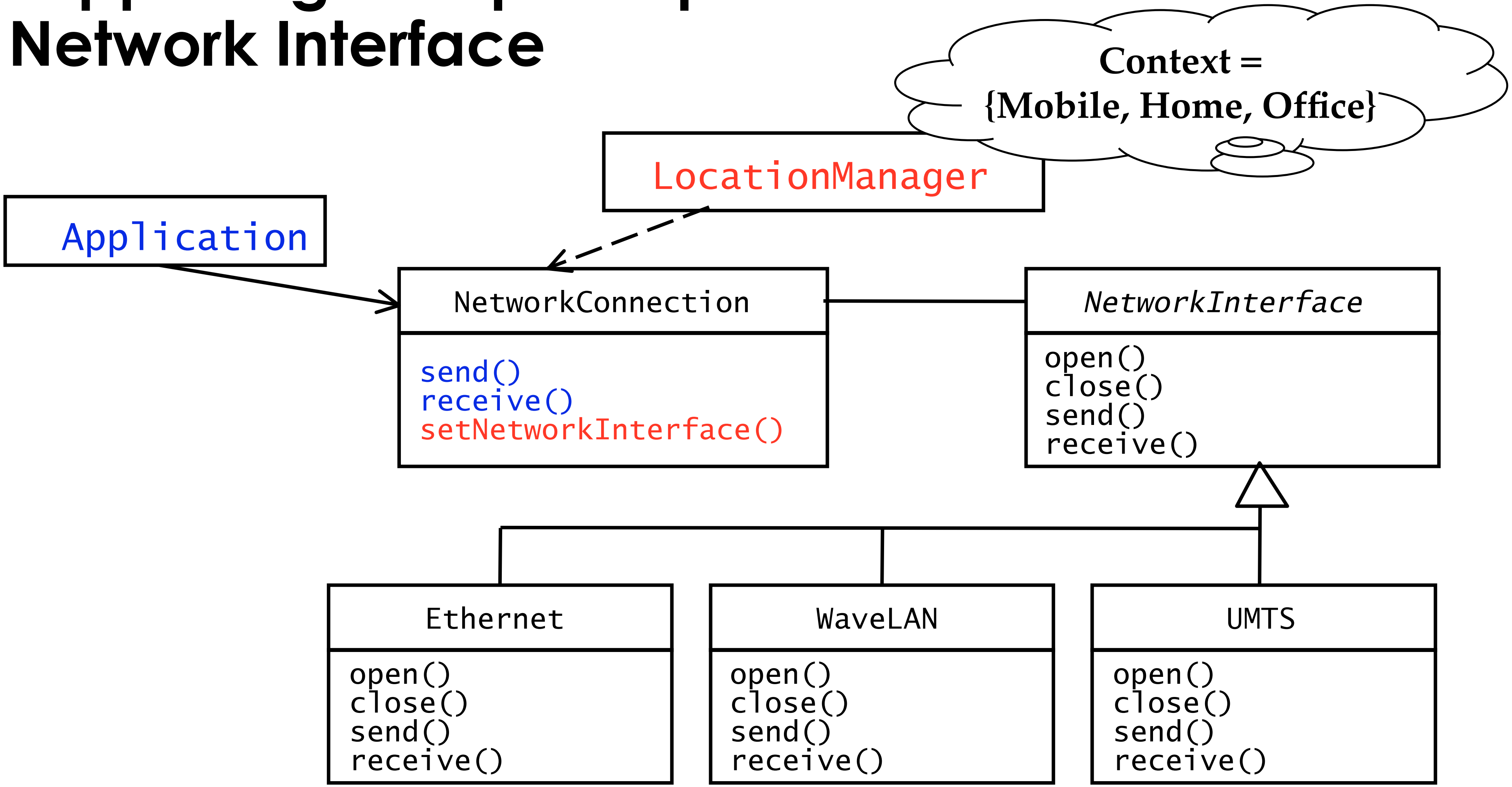


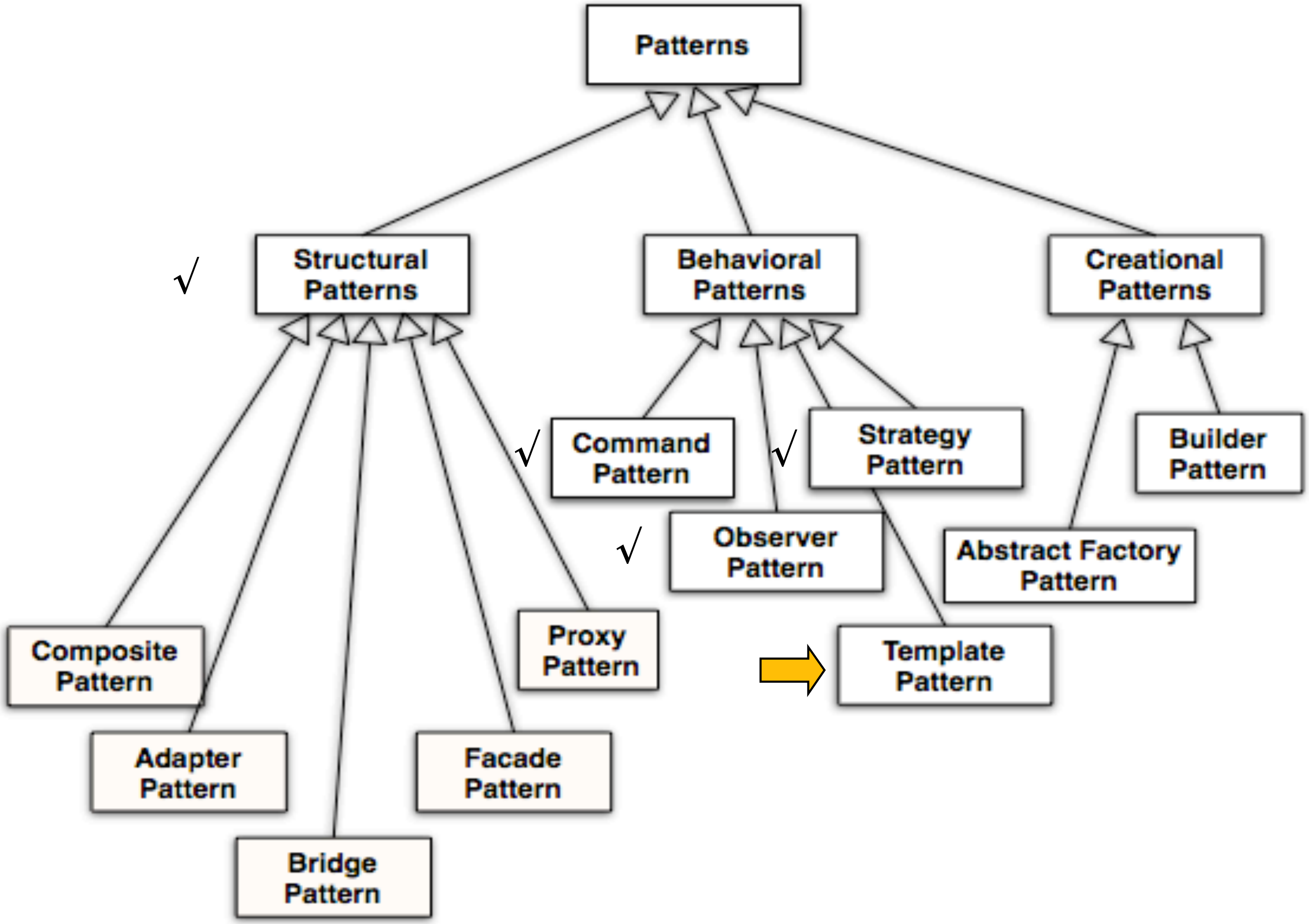
Policy decides which ConcreteStrategy is best in the current Context.

Using a Strategy Pattern to Decide between Algorithms at Runtime



Supporting Multiple implementations of a Network Interface



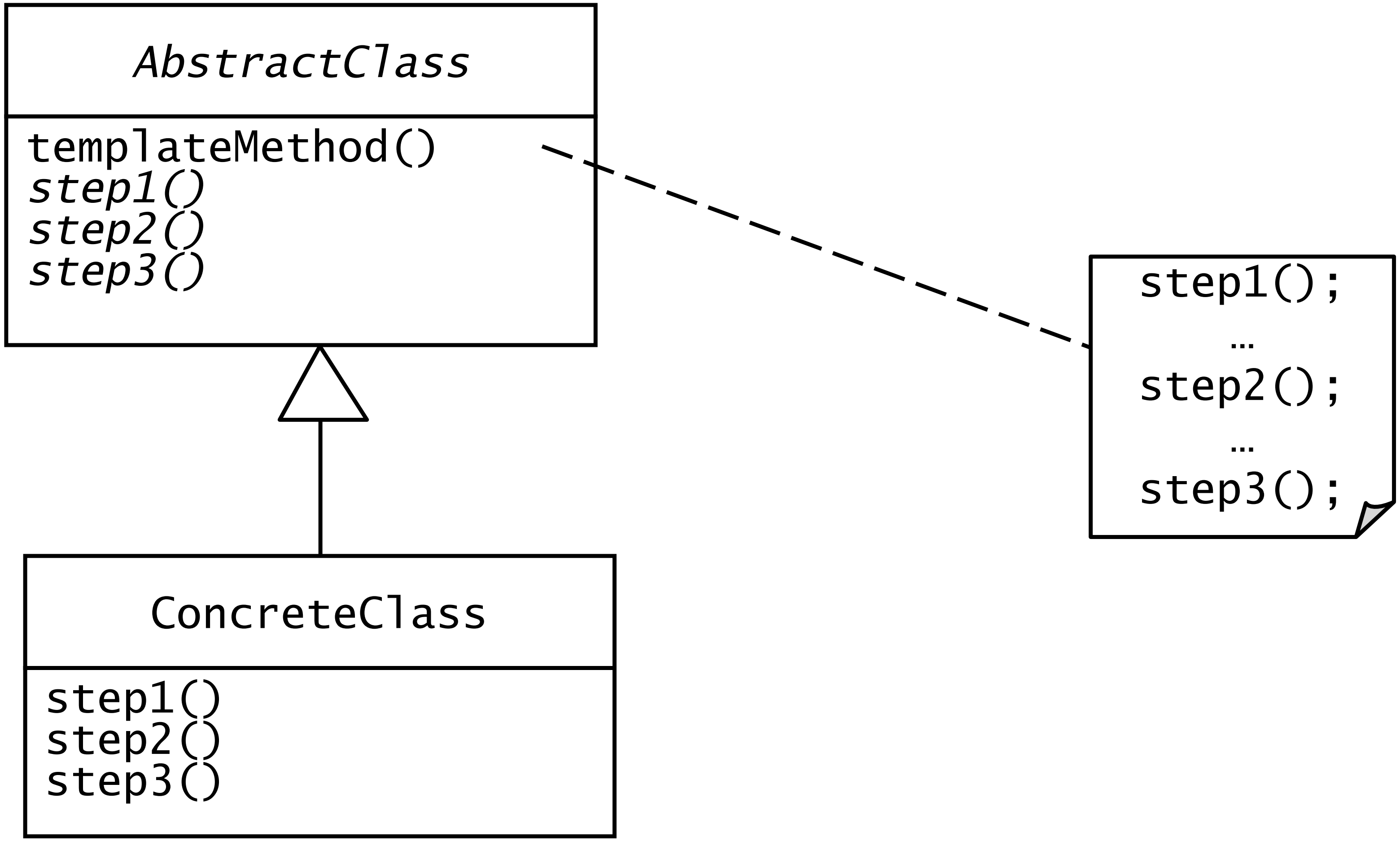

Patterns
Structural Patterns
Behavioral Patterns
Creational Patterns
√
√
Command Pattern
√
Strategy Pattern
Builder Pattern
√
Observer Pattern
Abstract Factory Pattern
Composite Pattern
Proxy Pattern
Template Pattern
Adapter Pattern
Facade Pattern
Bridge Pattern

Template Method Motivation

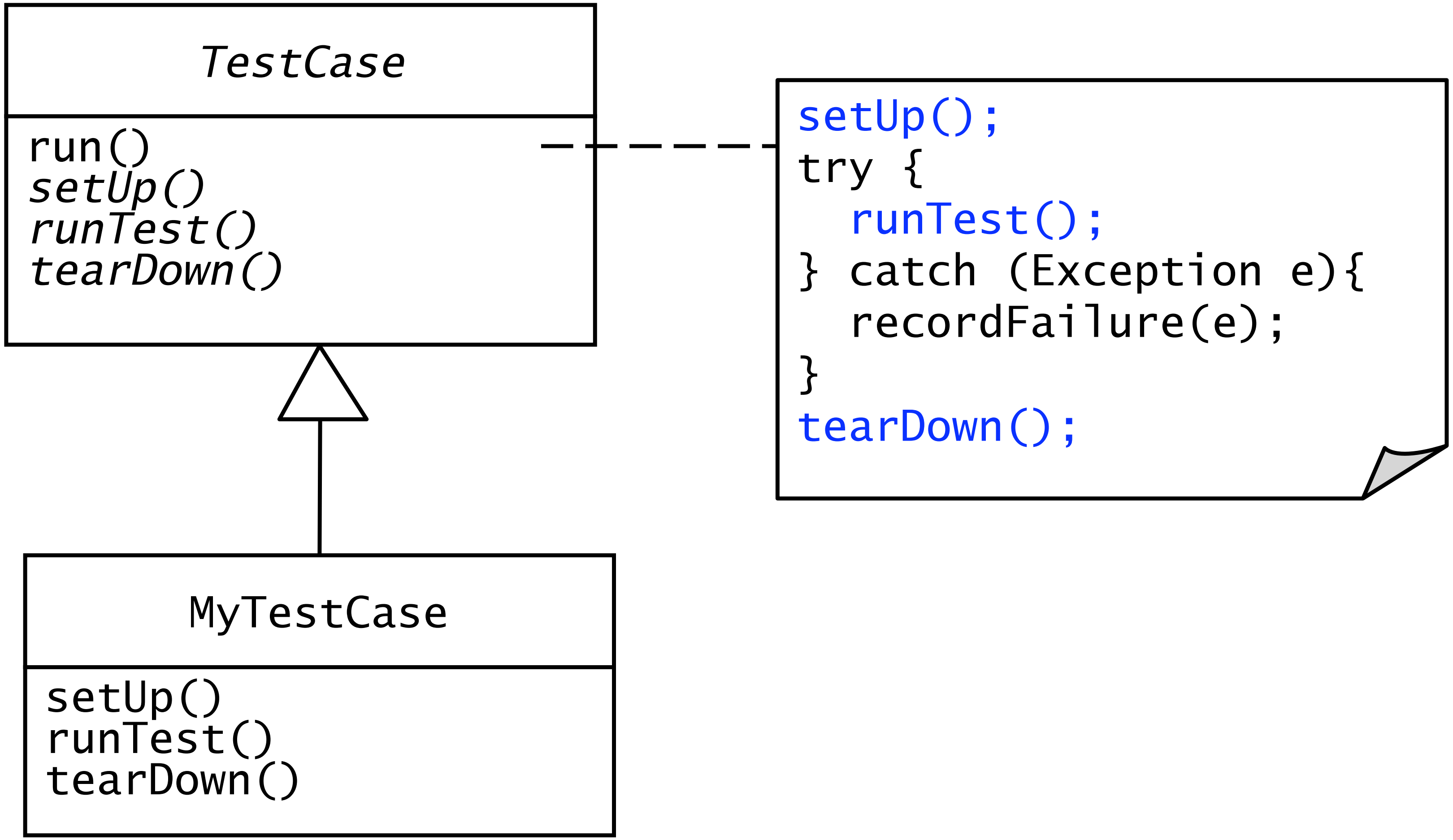
- Several subclasses share the same algorithm but differ on the specifics
- Common steps should not be duplicated in the subclasses
- Examples:
  - Executing a test suite of test cases
  - Opening, reading, writing documents of different types
- Approach
  - The common steps of the algorithm are factored 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();
```

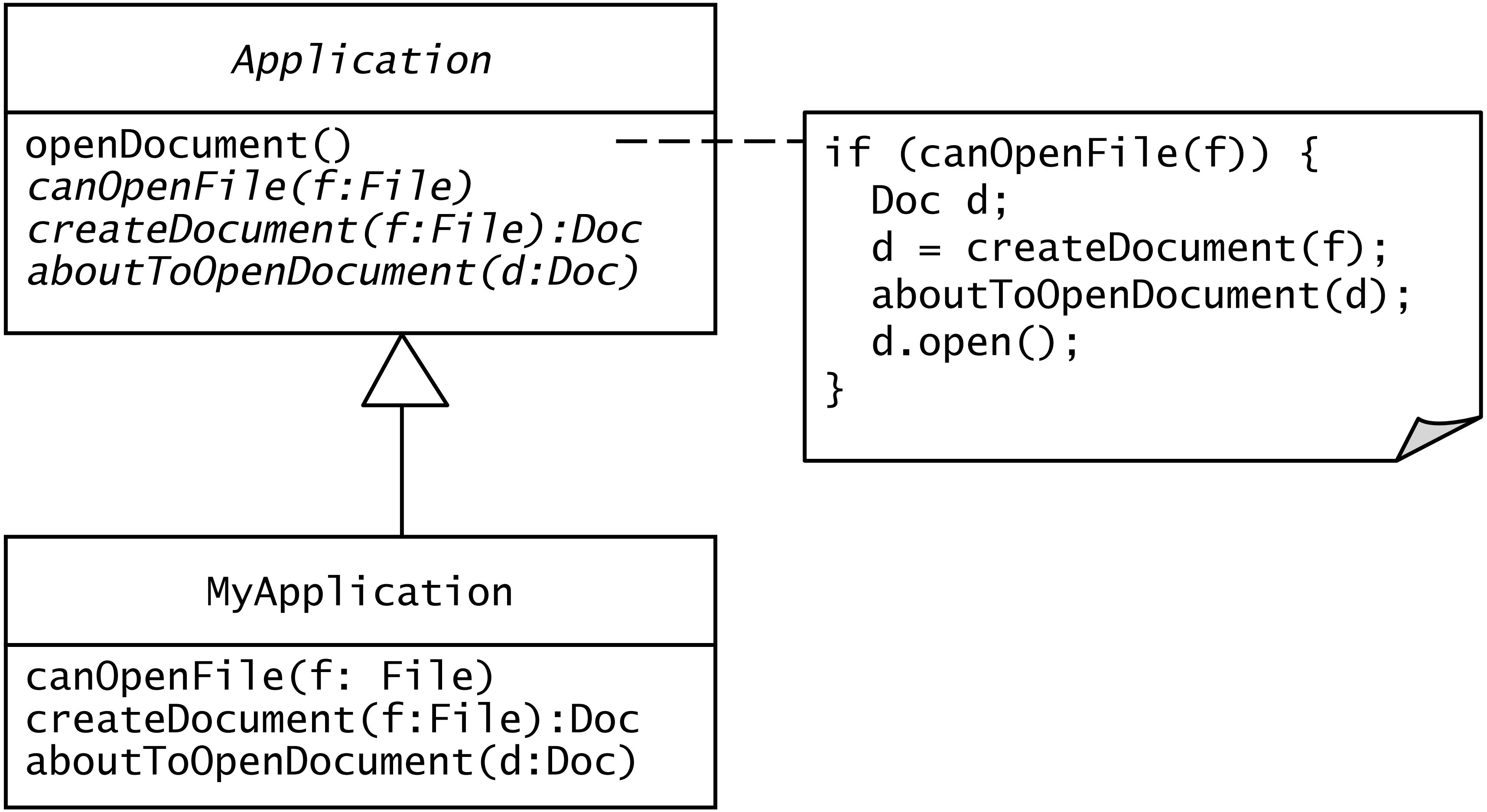
Template Method



Template Method Example: Test Cases



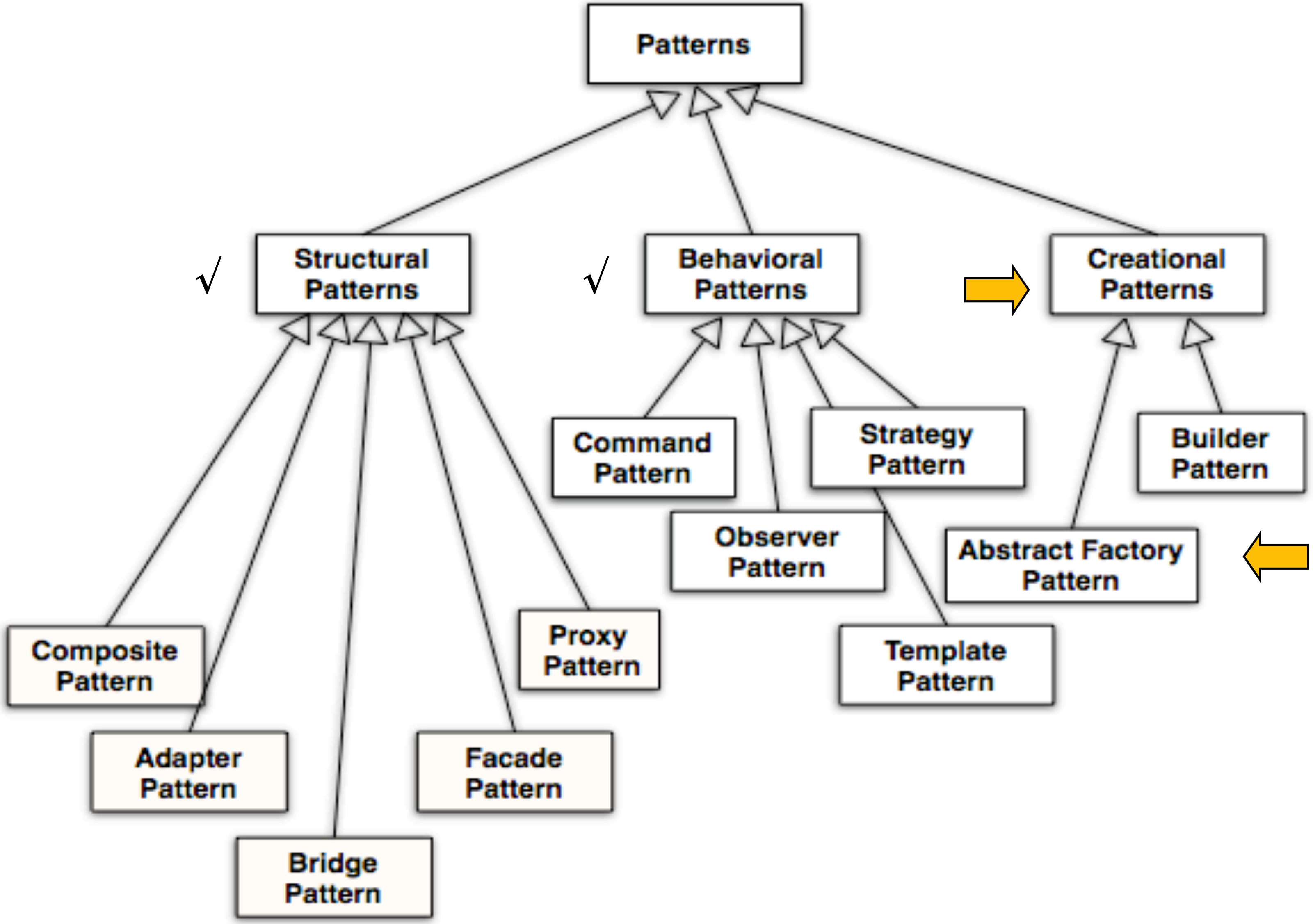
Template Method Example: Opening Documents



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

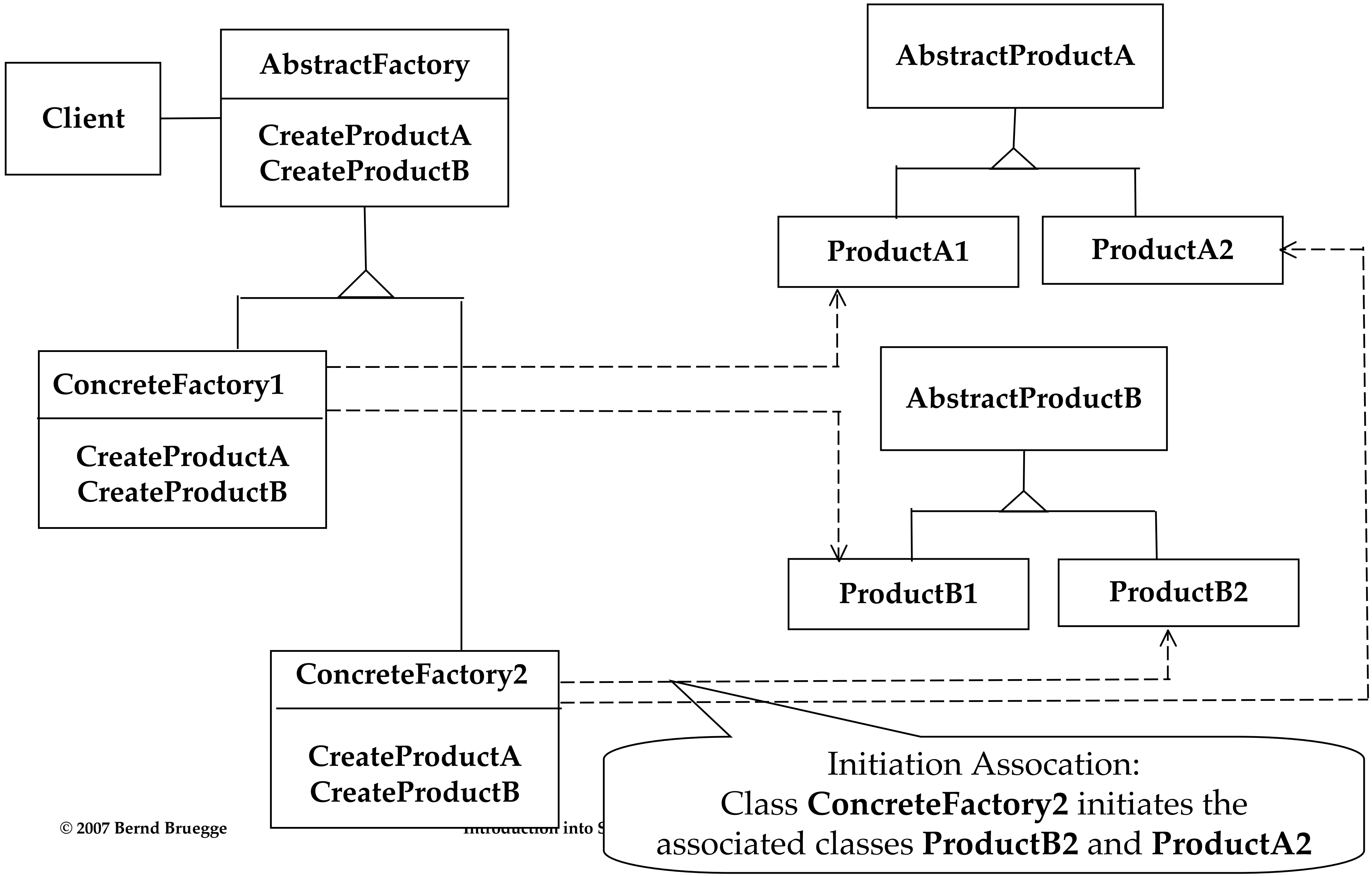
- Principle: “Don’t call us, we’ll call you”.


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?

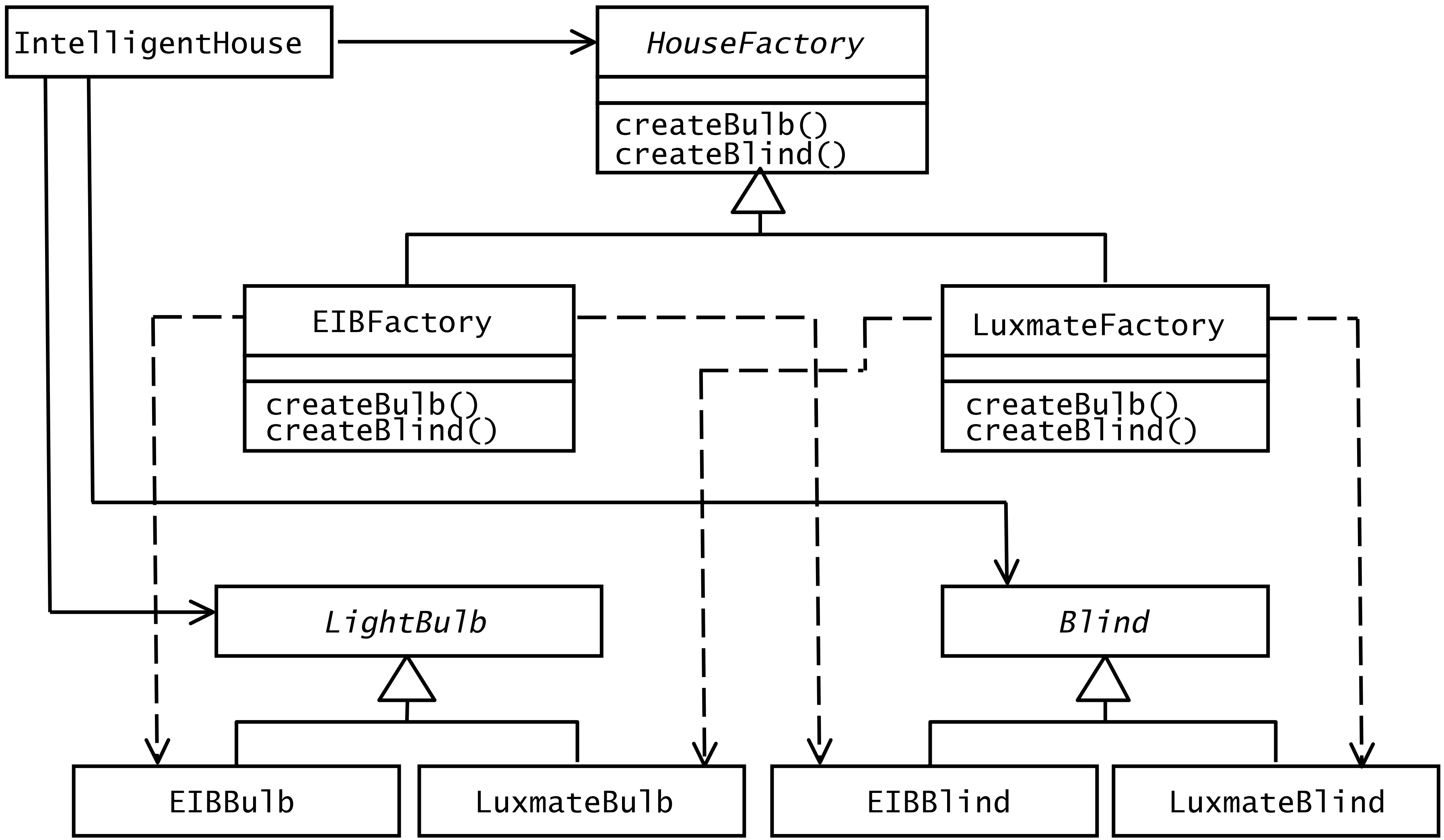
Abstract Factory



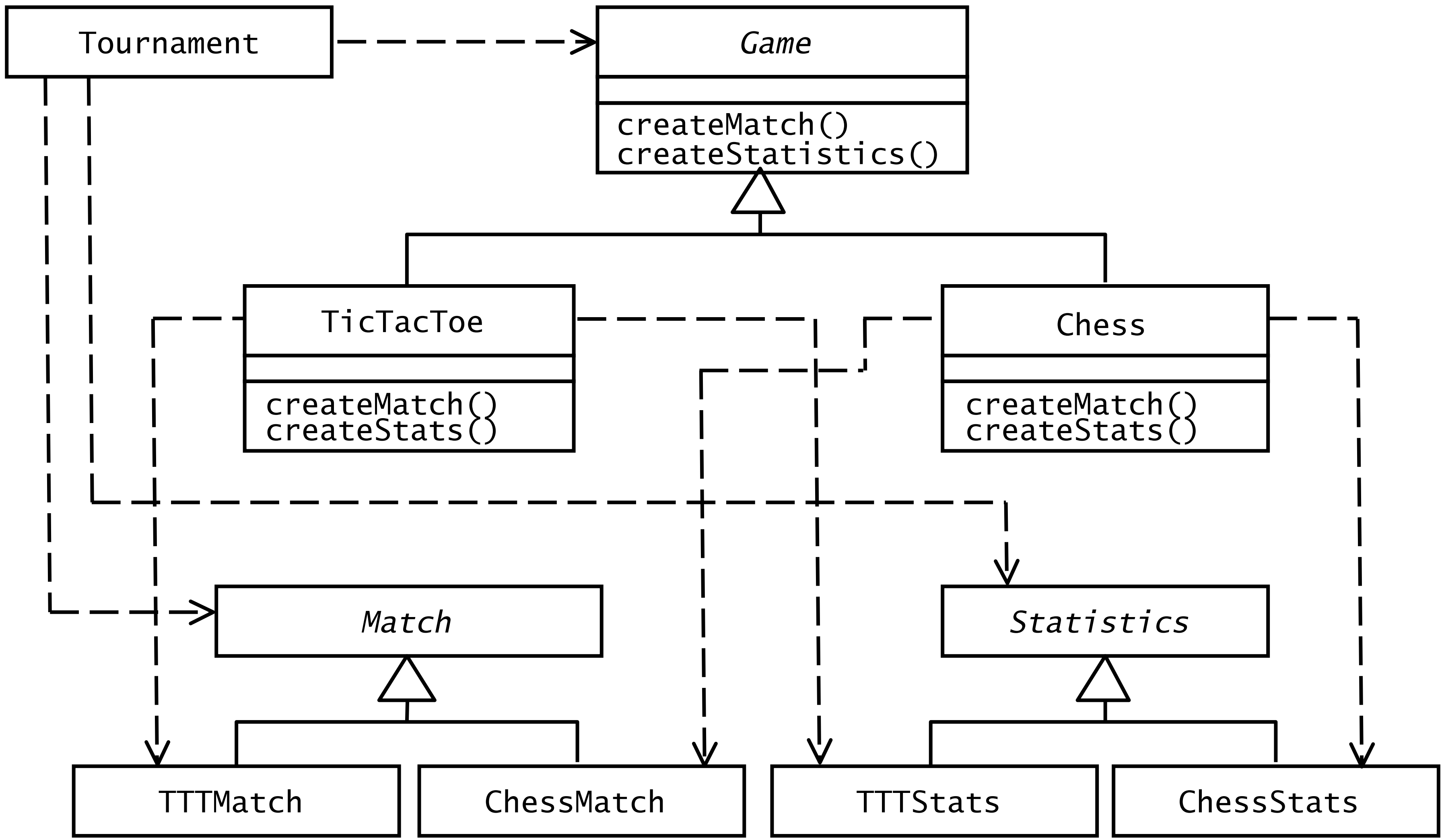
Applicability for Abstract Factory Pattern

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

Example: A Facility Management System for a House



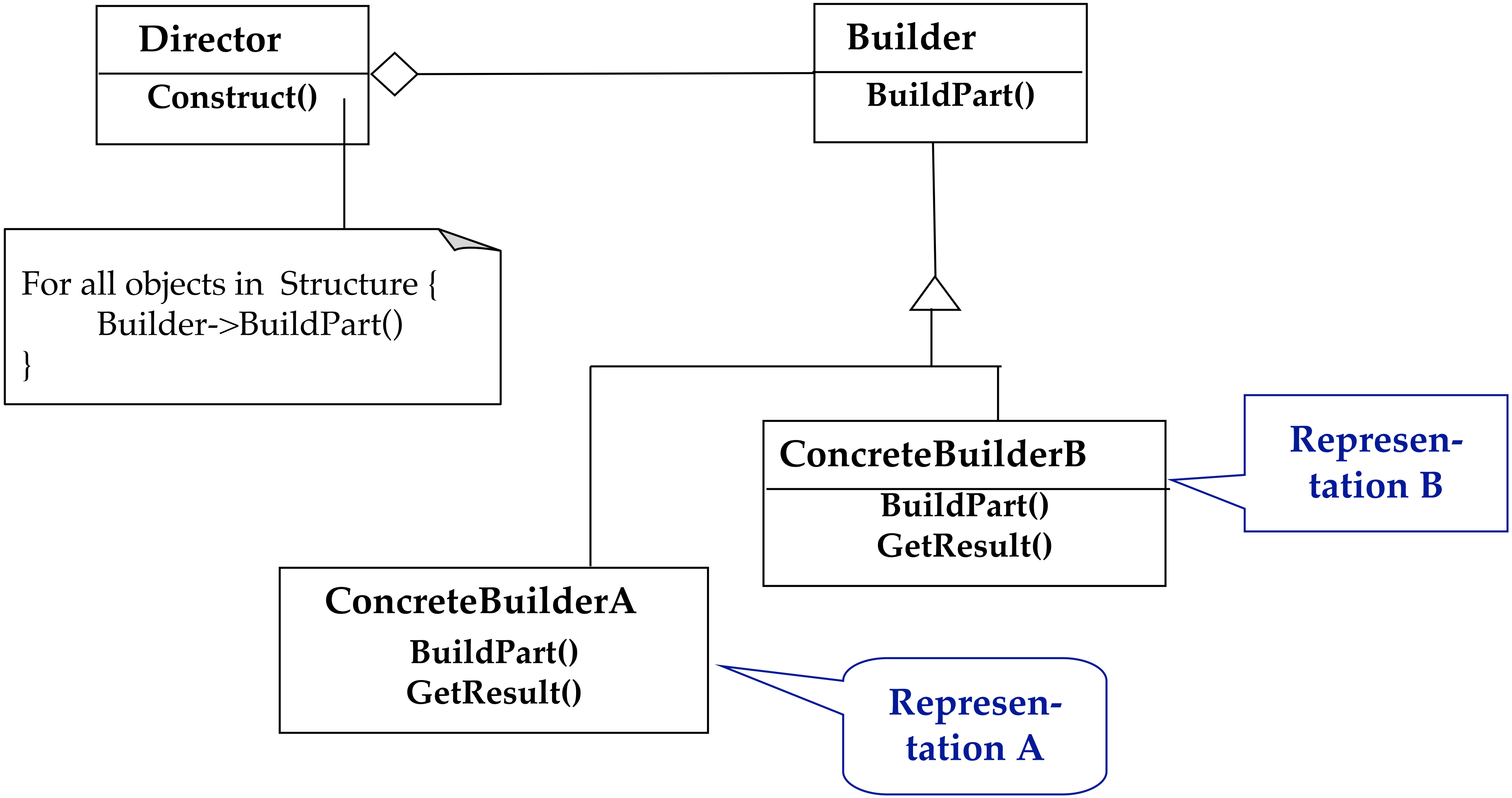
Applying the Abstract Factory Pattern to Games



Builder Pattern Motivation 5 22 2007

- 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")

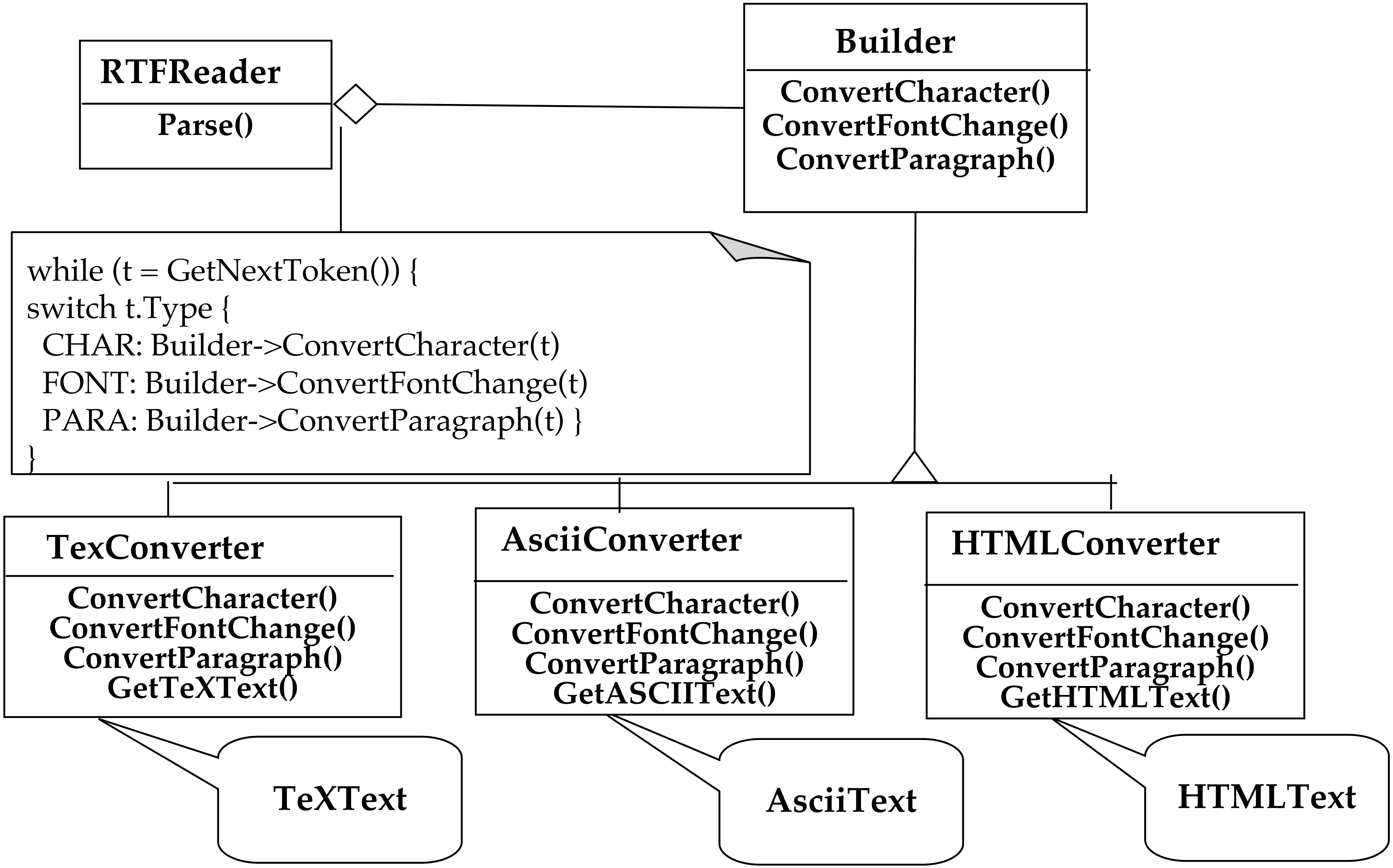
Builder Pattern



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 all your software engineering problems
  - Pattern-oriented development
- My favorites: Composite, Strategy, Builder and Observer.