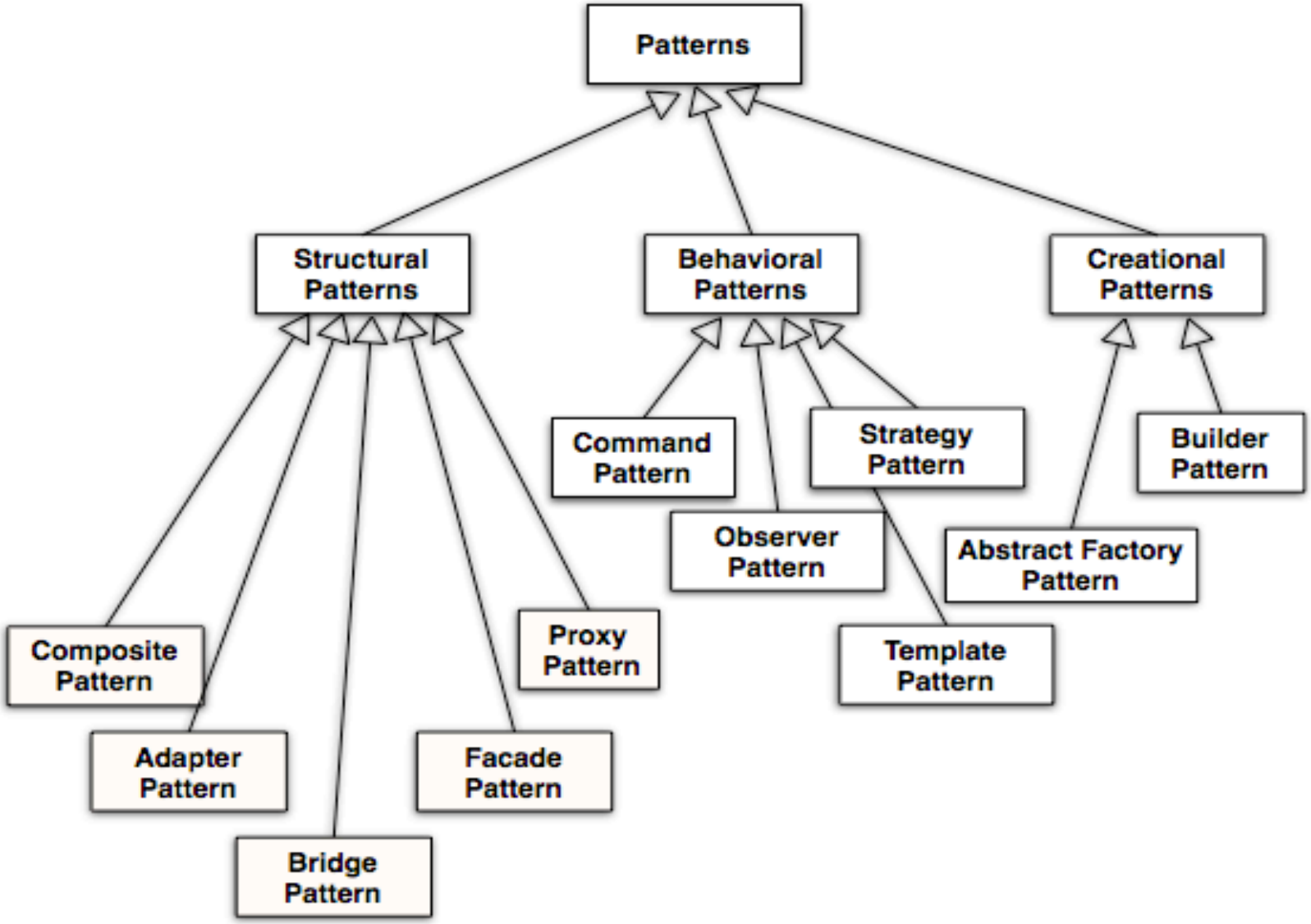
Chapter 8, Object Design: Design Patterns II

A Taxonomy of Design Patterns



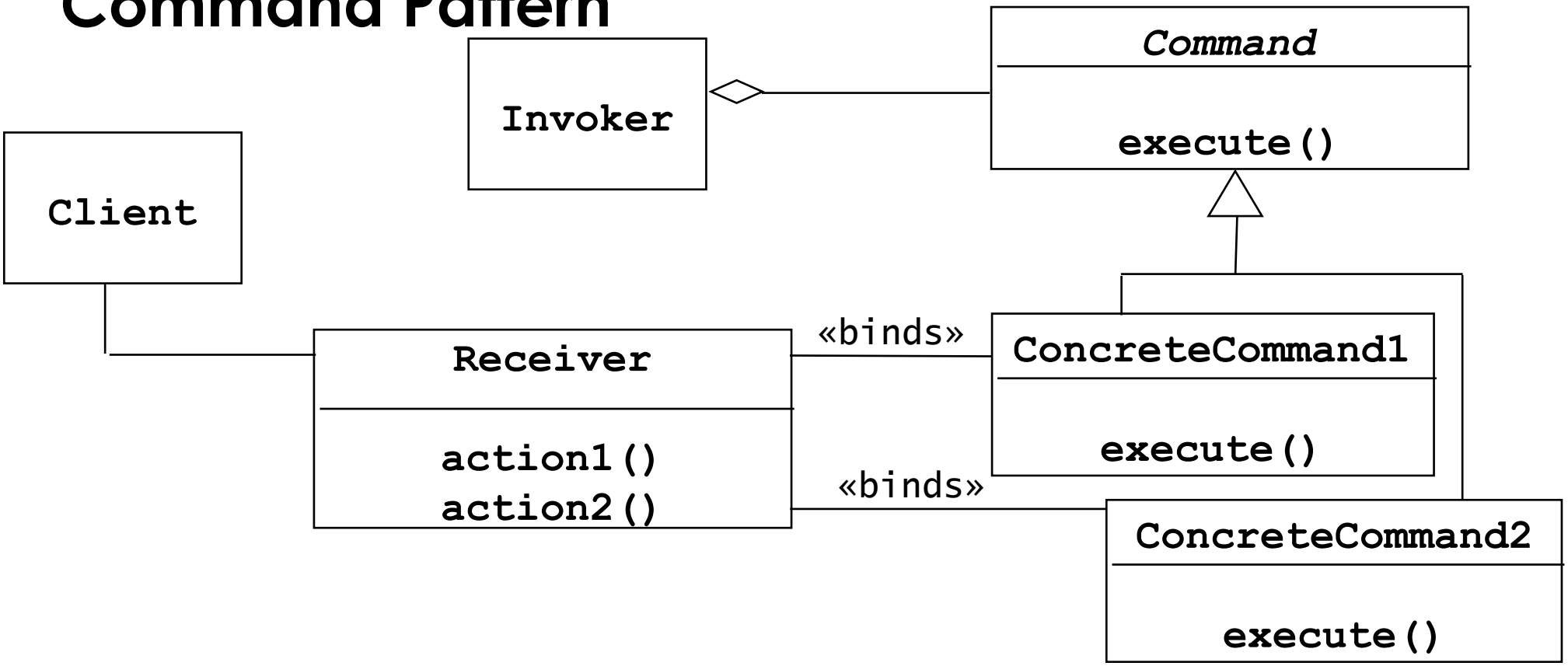
The Proxy Pattern: 3 Types

- 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
    - Example: RealImage and ImageProxy
- 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.

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

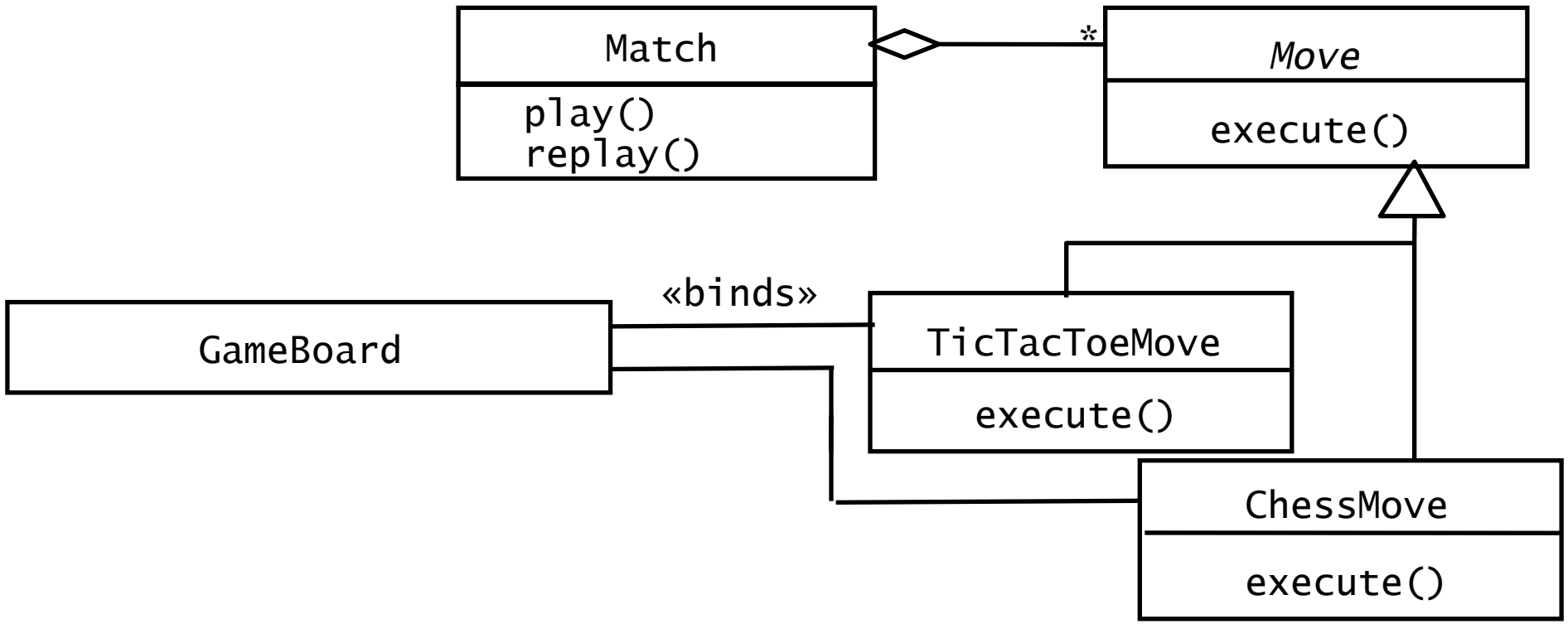
- The command pattern can be nicely used to decouple boundary objects from control objects:
  - Boundary objects such as menu items and buttons, send messages to the command objects (I.e. the control objects)
  - Only the command objects modify entity objects
- When the user interface is changed (for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

Command Pattern Applicability

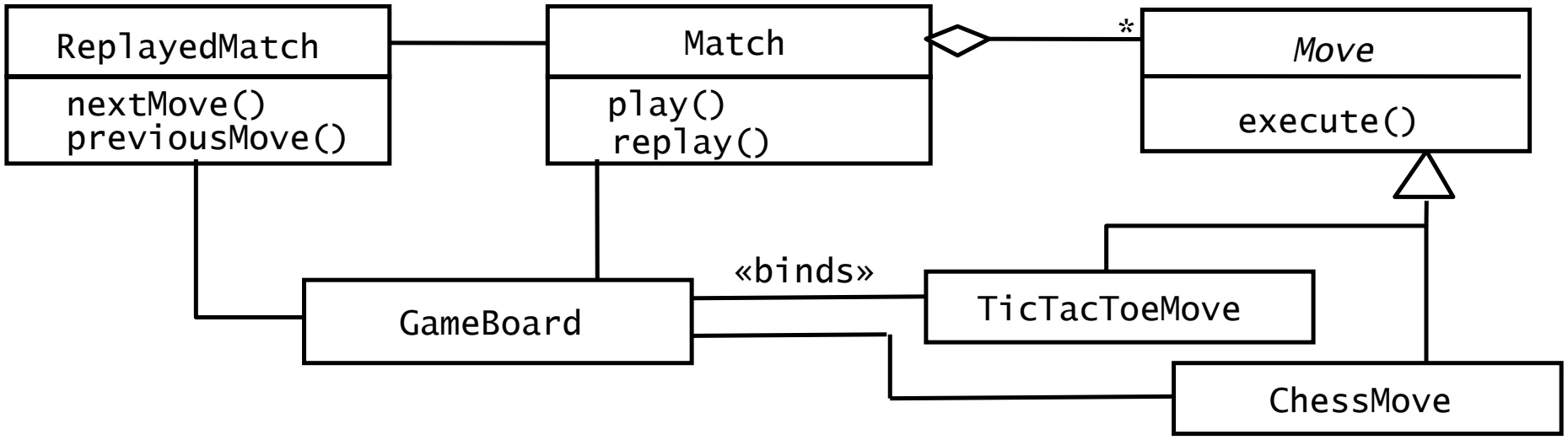
- Parameterize clients with different requests
- Queue or log requests
- Support undoable operations

- Uses:
  - Undo queues
  - Database transaction buffering

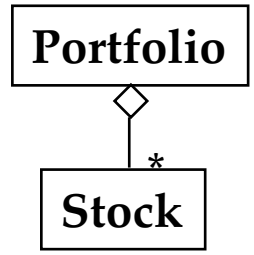
Applying the Command Pattern to Command Sets



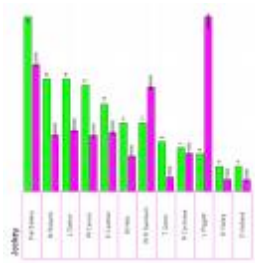
Applying the Command design pattern to Replay Matches in ARENA

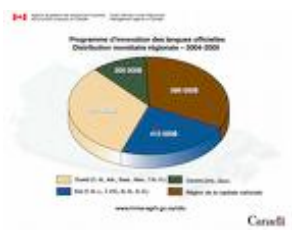


Observer Pattern Motivation

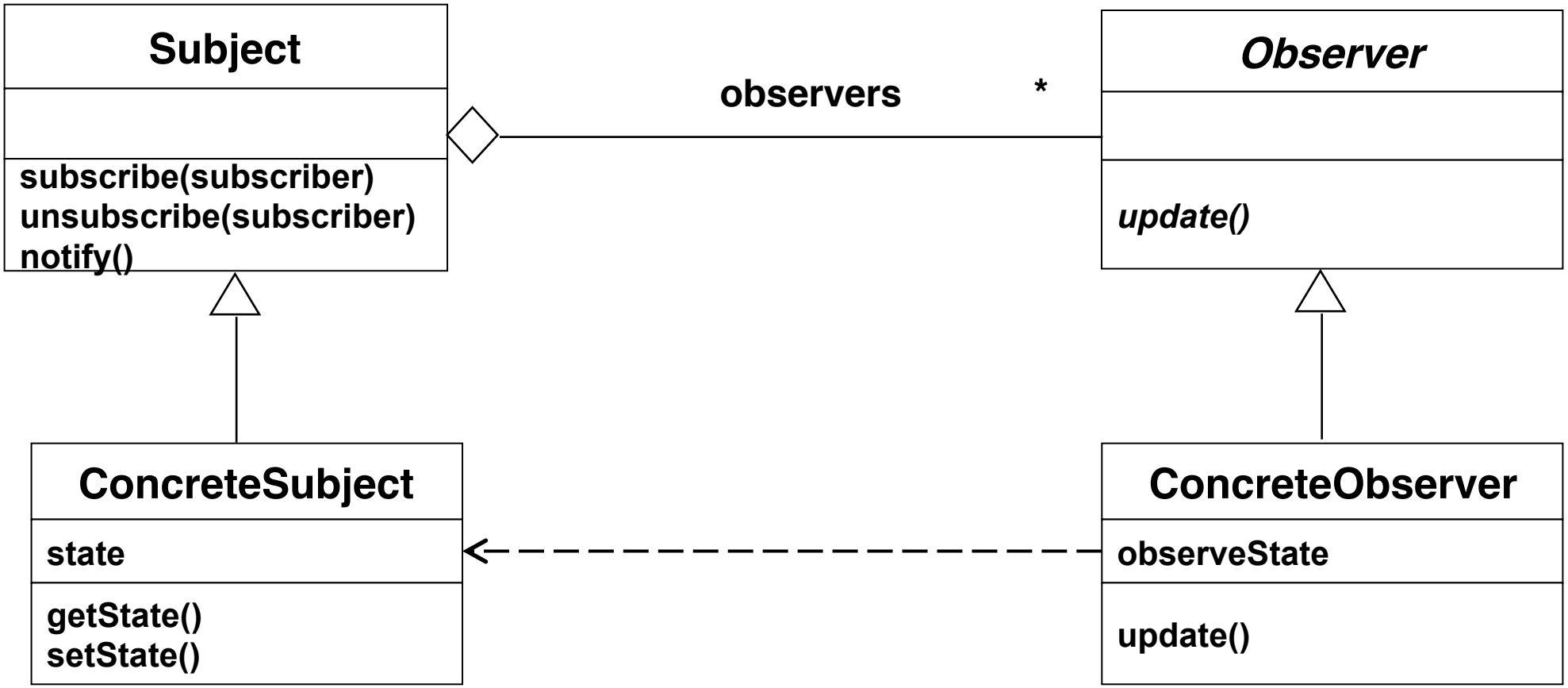


- Problem:
  - We have an object that changes its state quite often
    - Example: A Portfolio of stocks
  - We want to provide multiple views of the current state of the portfolio
    - Example:Histogram view, pie chart view, time line view, alarm
- Requirements:
  - The system should maintain consistency 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.





Observer Pattern: Decouples an Abstraction from its Views

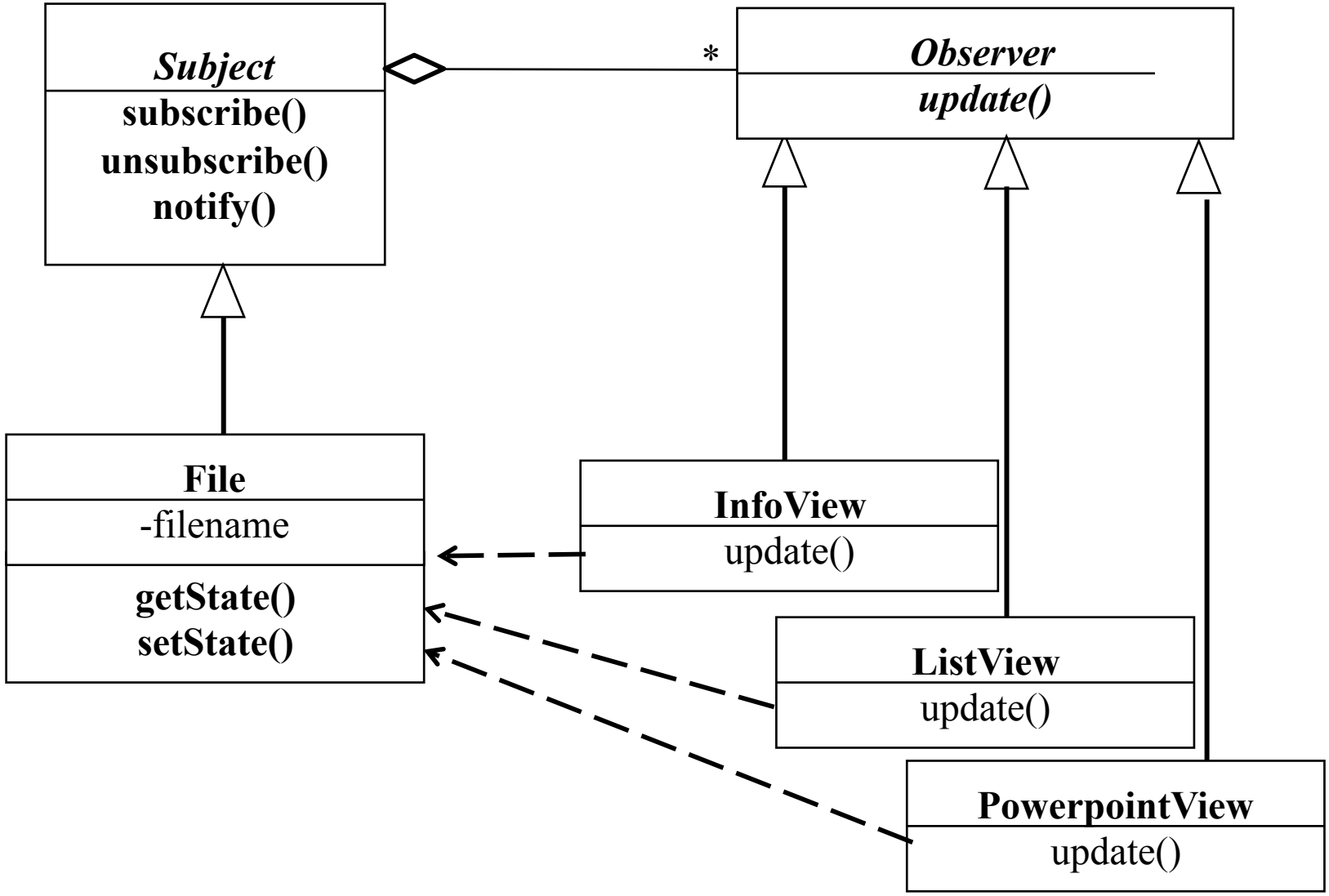


- The **Subject** ("Publisher") represents the entity object
- **Observers** ("Subscribers") attach to the Subject by calling **subscribe()**
- Each Observer has a different view of the state of the entity object
  - The **state** is contained in the subclass **ConcreteSubject**
  - The state can be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

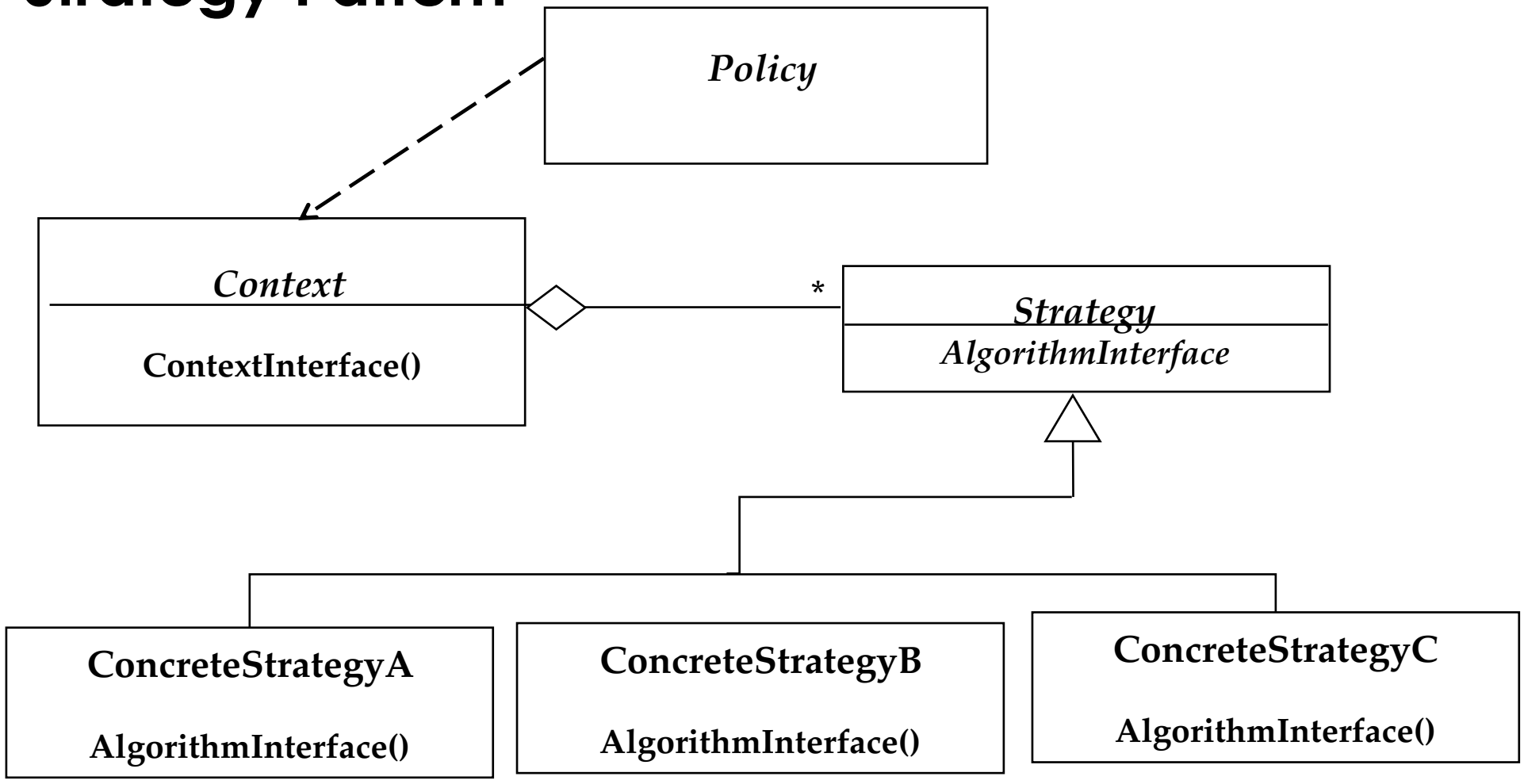
Applying the Observer Pattern to maintain Consistency across Views



Strategy Pattern

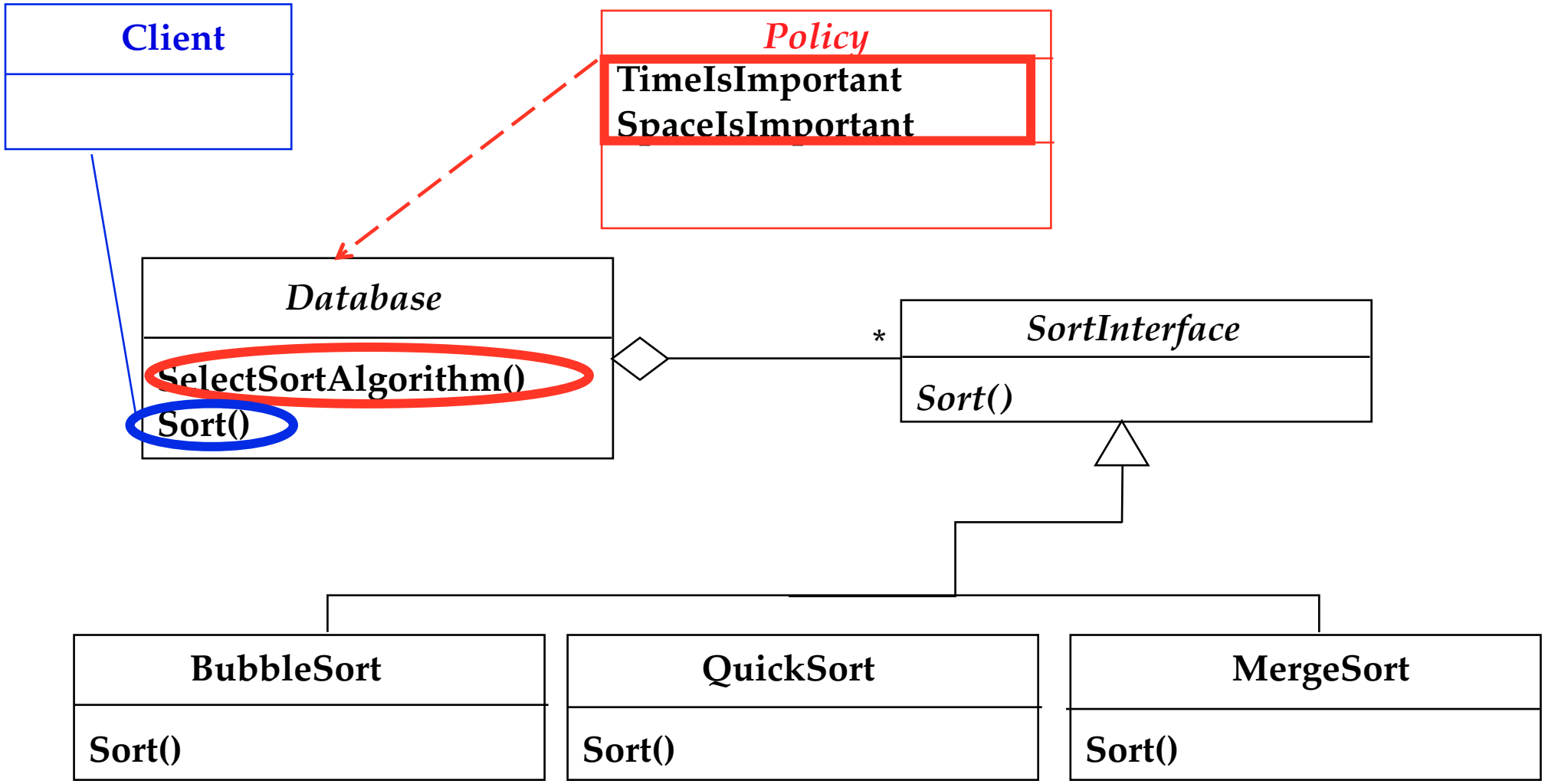
- Different algorithms exists for a specific task
  - We can switch between the algorithms at run time
- Examples of tasks:
  - Different collision strategies for objects in video games
  - Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
  - Sorting a list of customers (Bubble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

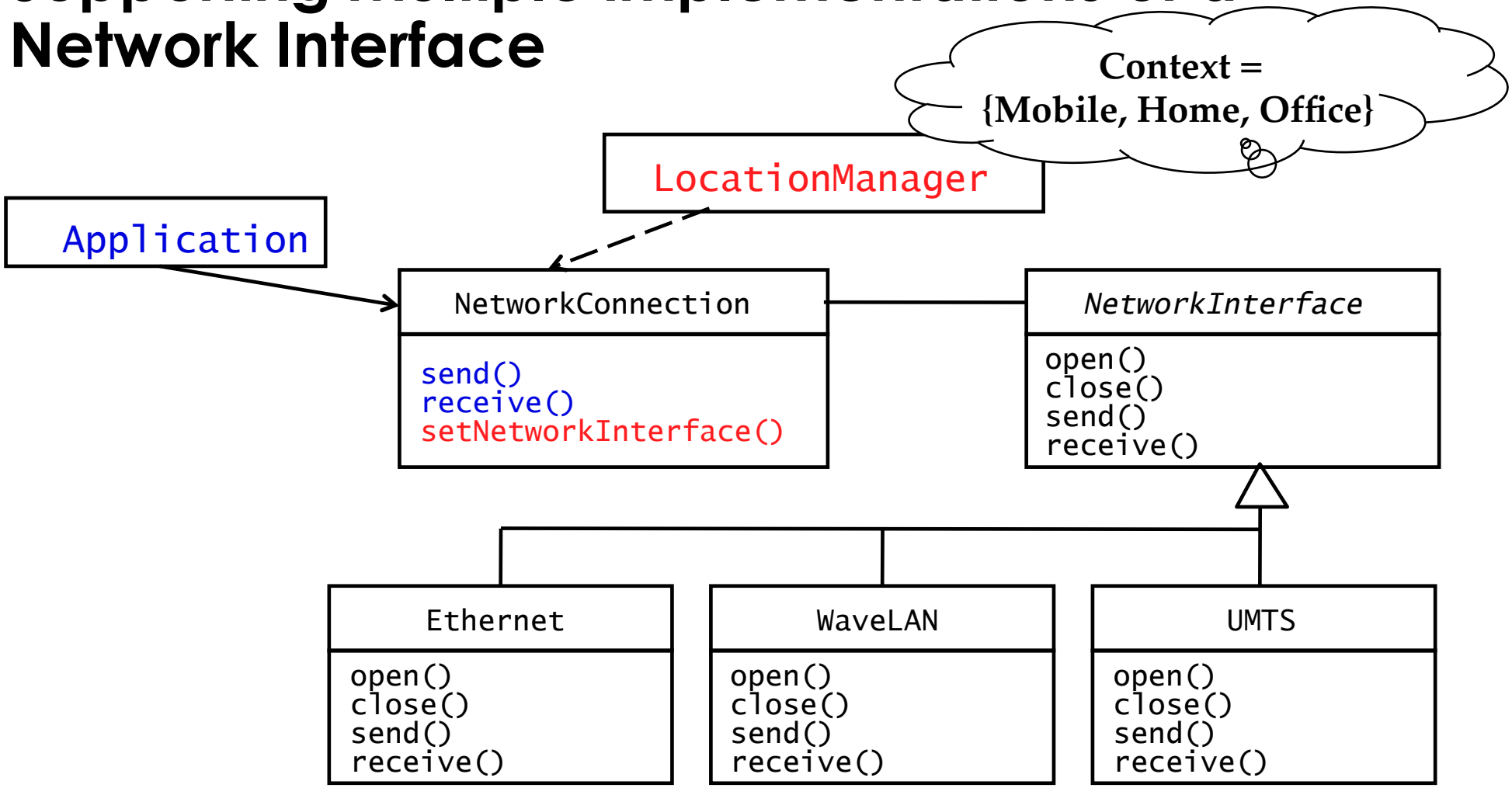


Policy decides which ConcreteStrategy is best in the current Context.

Using a Strategy Pattern to Decide between Algorithms at Runtime



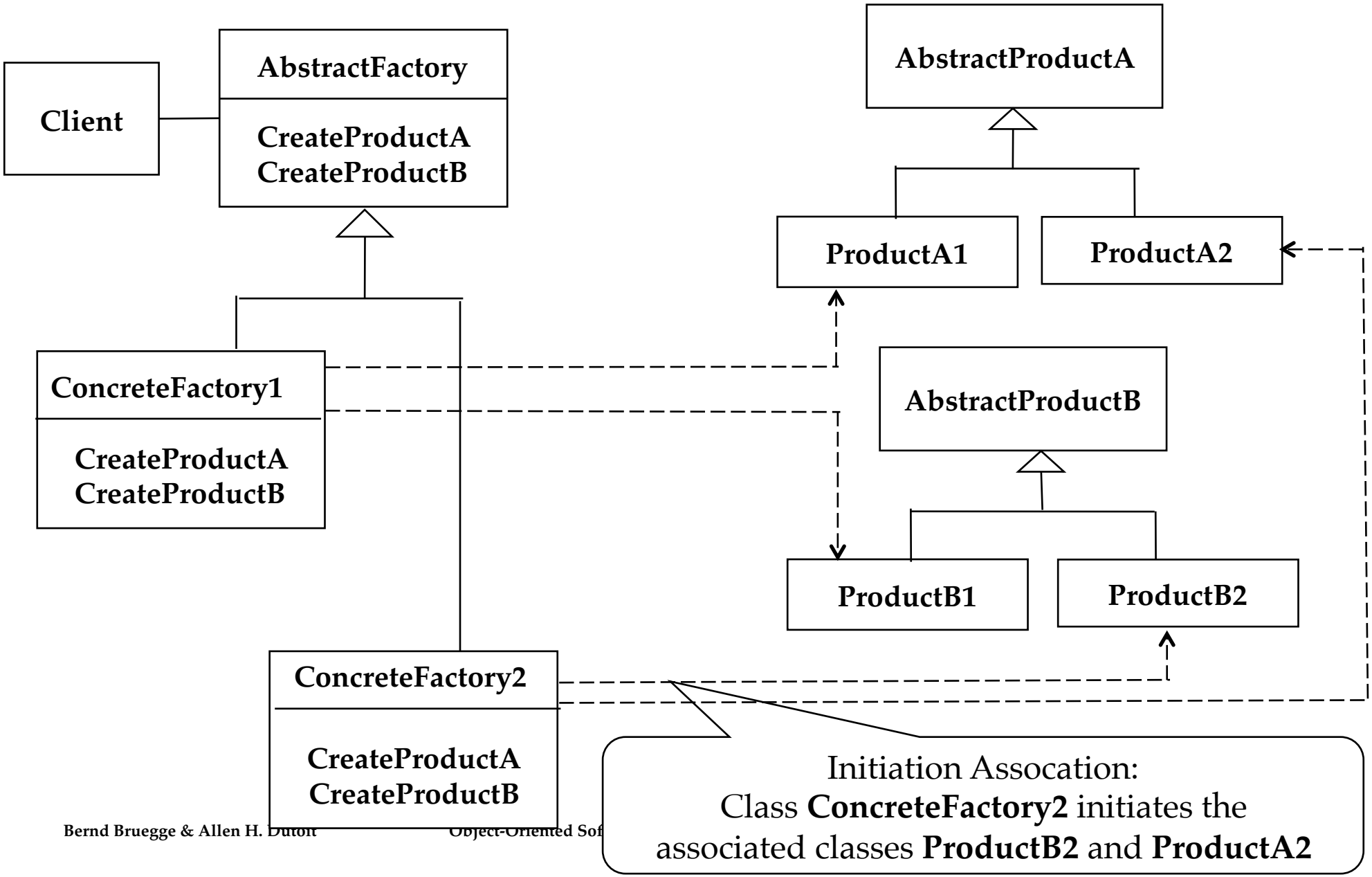
Supporting Multiple implementations of a Network Interface



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?

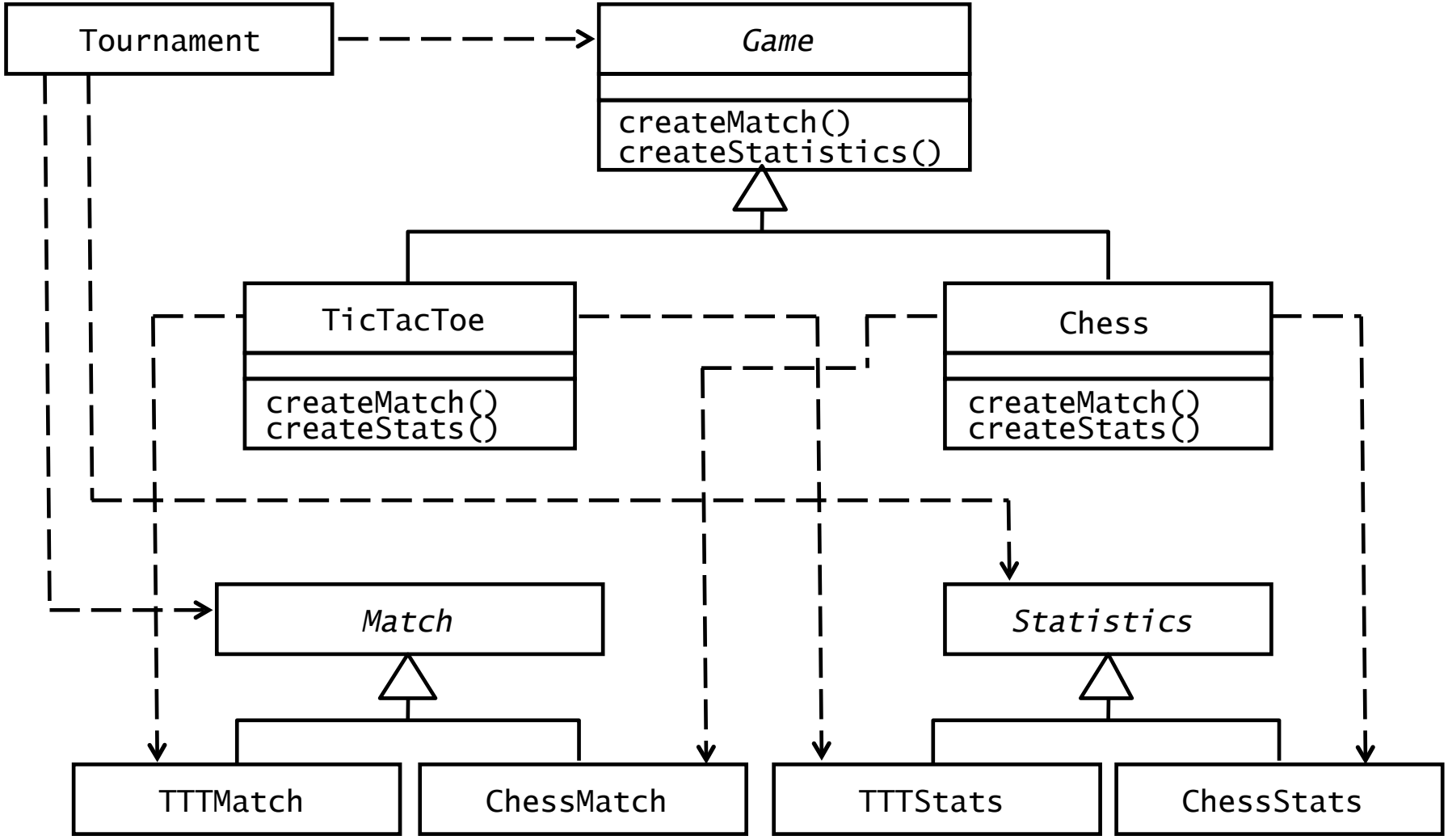
Abstract Factory



Applicability for Abstract Factory Pattern

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

Applying the Abstract Factory Pattern to Games



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