Section 3.3
Design Patterns

1. Overview
2. Types of design patterns
3. Selected design patterns

3.3.1 Overview

- What are design patterns?
  - they are a set of classes and the associations between them
  - they provide a partial solution to common design problems
  - each pattern addresses a **specific** design problem

- Characteristics
  - robust, modifiable, adaptable to different applications

- Original reference

  E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

3.3.2 Types of Design Patterns

- Selection of original design patterns (Gamma *et al.*)
  - *creational*
    - they deal with object creation mechanisms
  - *structural*
    - they simplify the implementation of relationships between objects
  - *behavioural*
    - they realize common communication patterns between objects

3.3.3 Selected Design Patterns

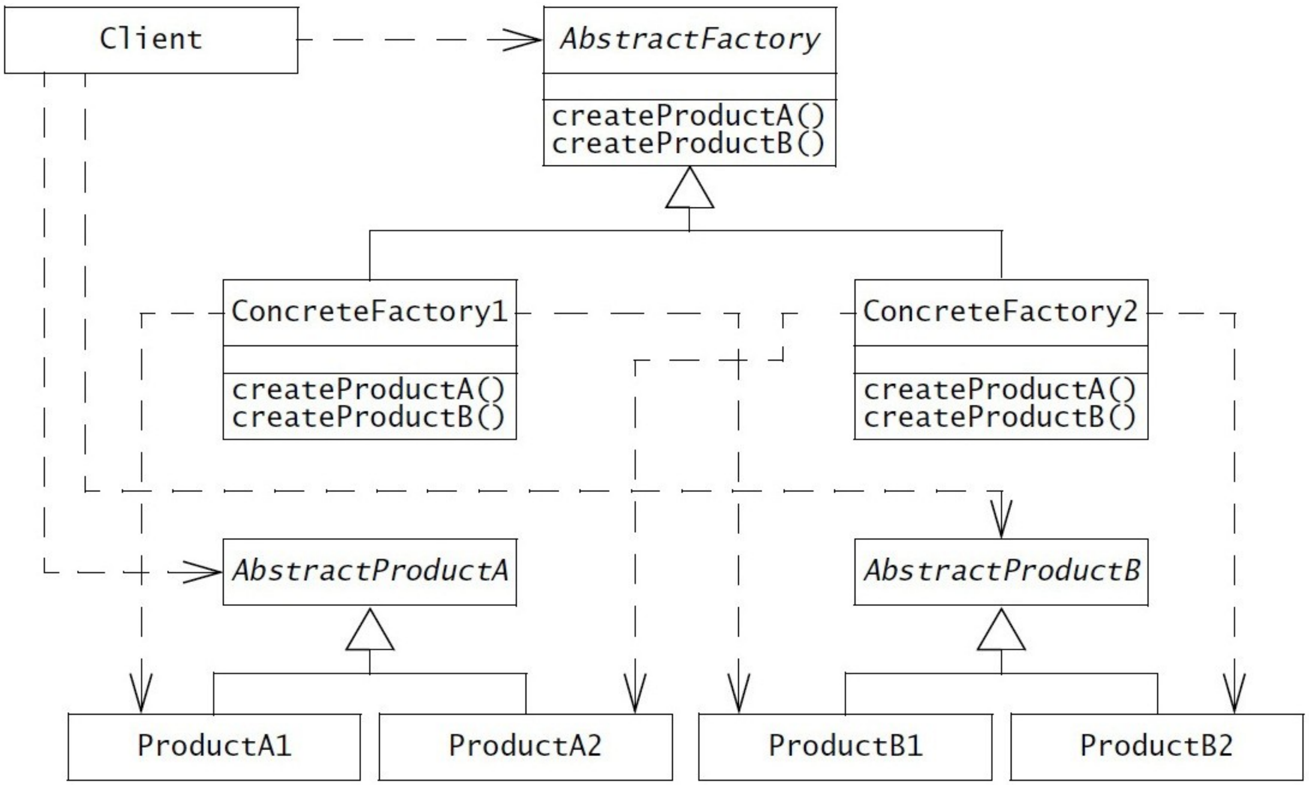
- Creational
  - Abstract Factory

- Structural
  - Adapter
  - Bridge
  - Composite
  - Facade
  - Proxy

- Behavioural
  - Command
  - Observer
  - Strategy

Abstract Factory

- Characteristics:
  - enables client-independent creation of objects
  - provides client with interface to classes with different implementations



Abstract Factory (cont.)

- Solution for encapsulating platforms:
  - used for substituting family of concrete products transparently from the client
  - example: application with products from different manufacturers

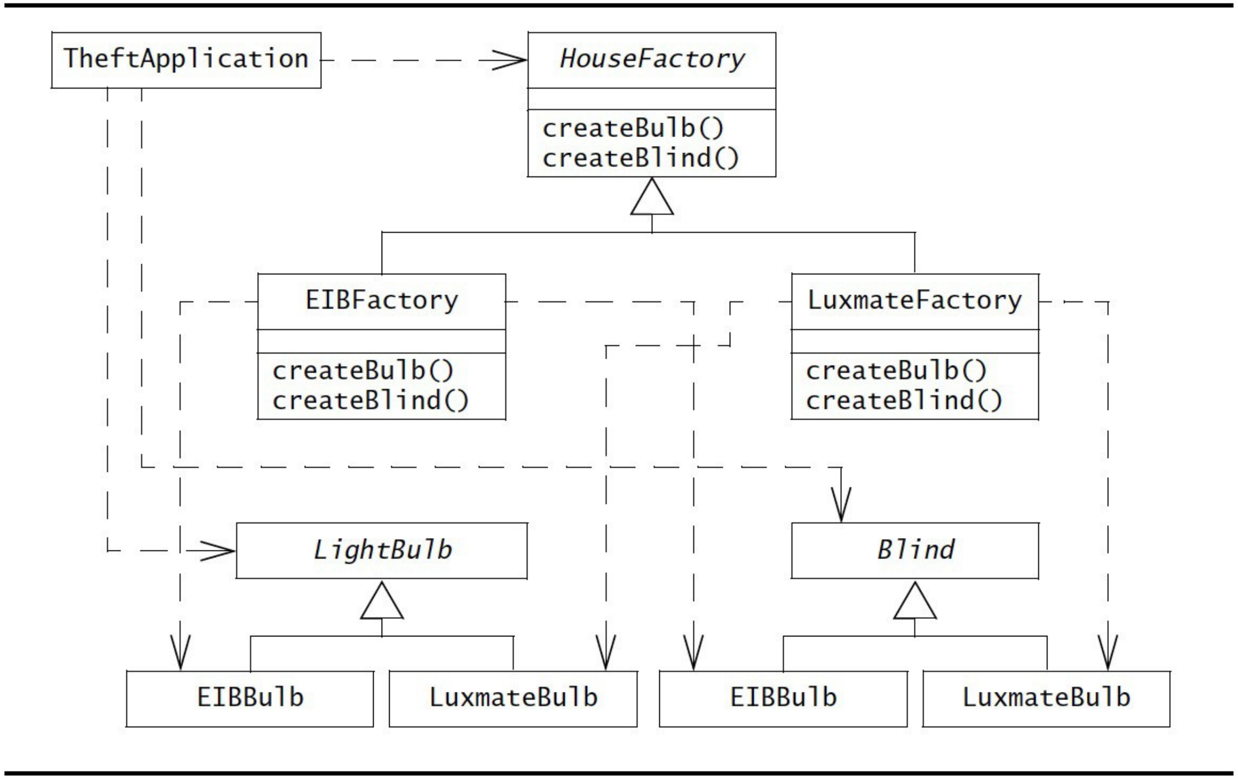


Figure 8-12 Applying the Abstract Factory design pattern to different intelligent house platforms (UML class diagram, dependencies represent «call» relationships).

Abstract Factory (cont.)

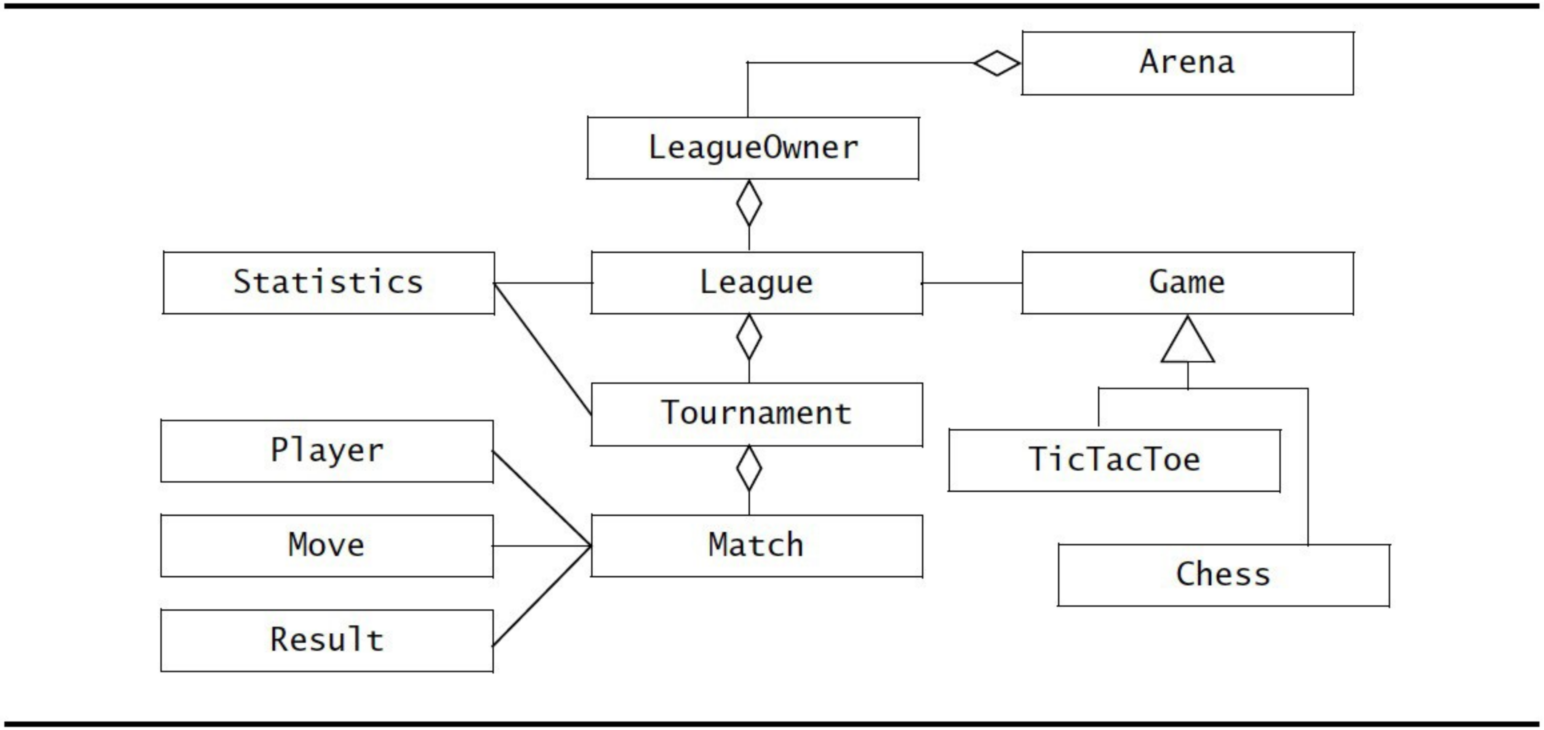


Figure 8-19 ARENA analysis objects related to Game independence (UML class diagram).

Abstract Factory (cont.)

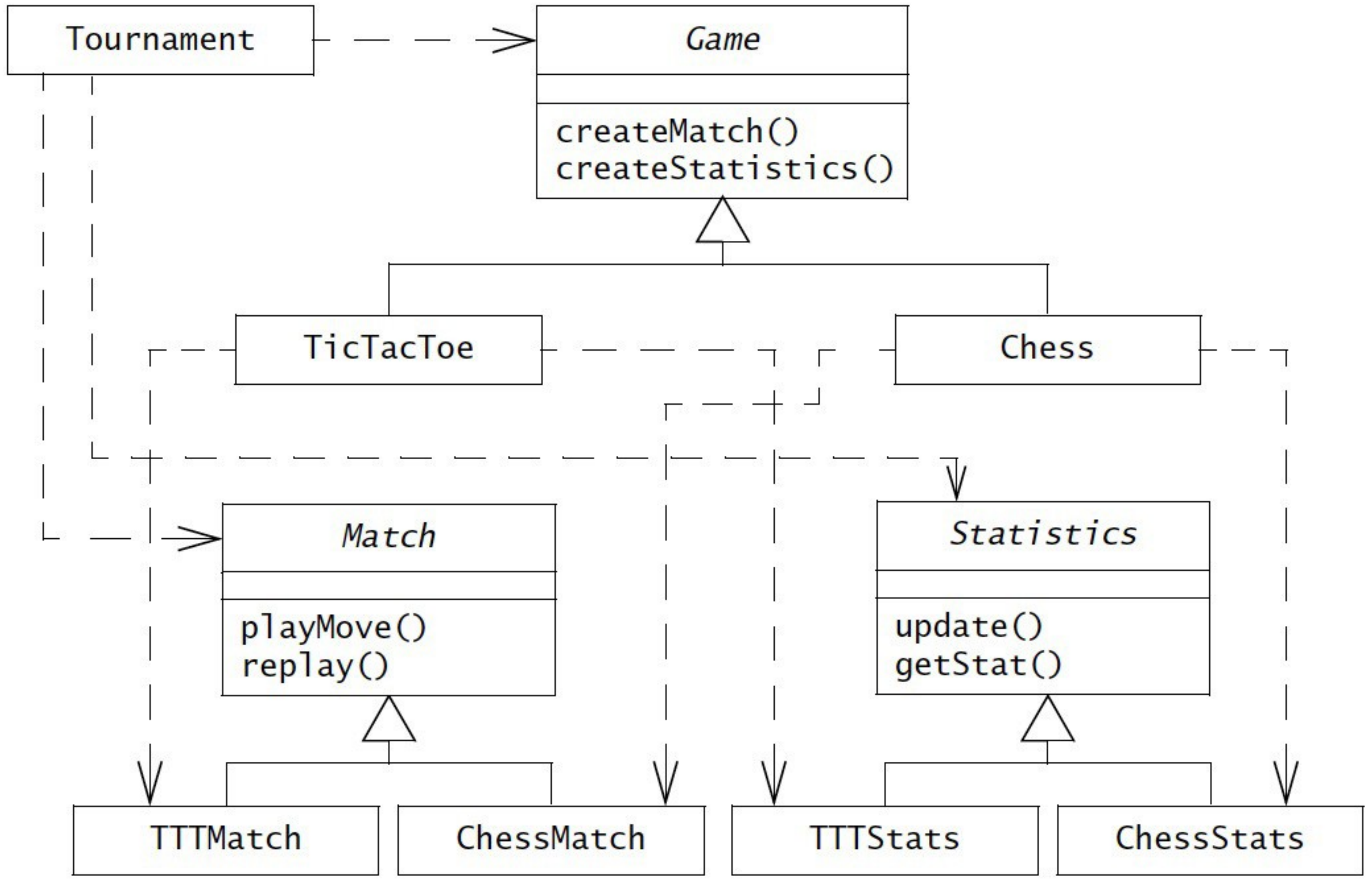
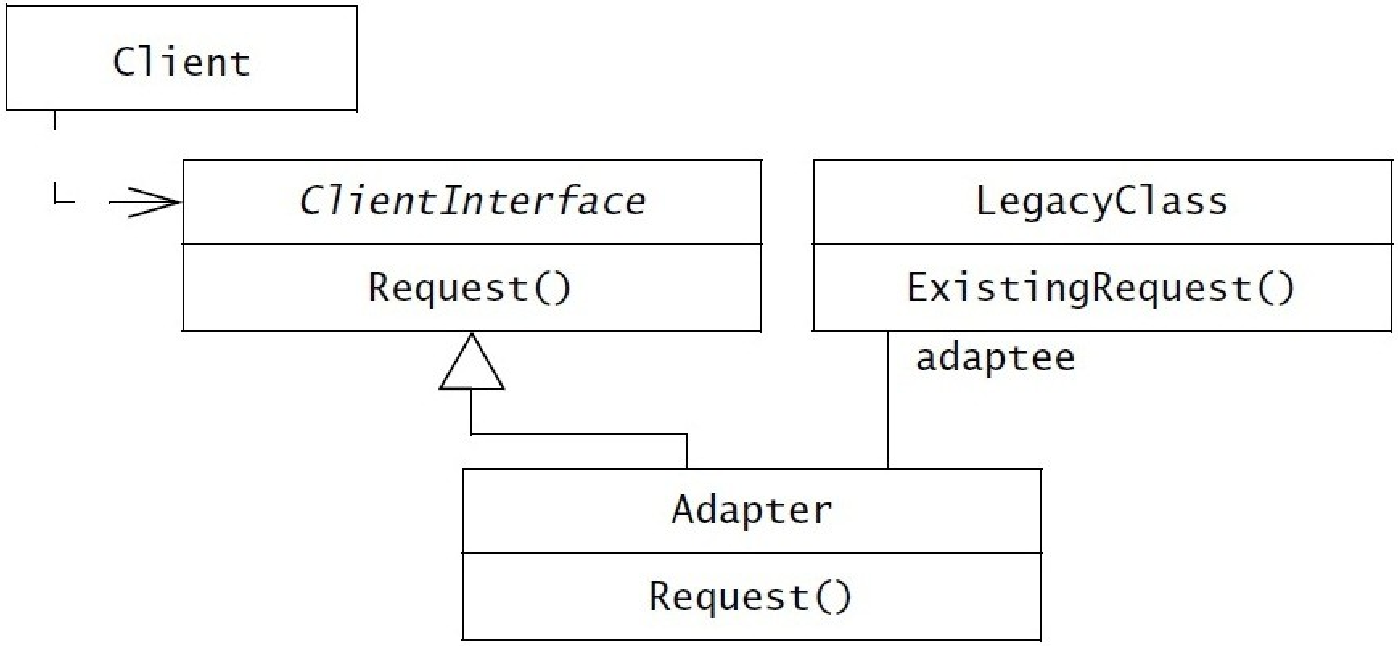


Figure 8-20 Applying the Abstract Factory design pattern to Games (UML class diagram).

Adapter

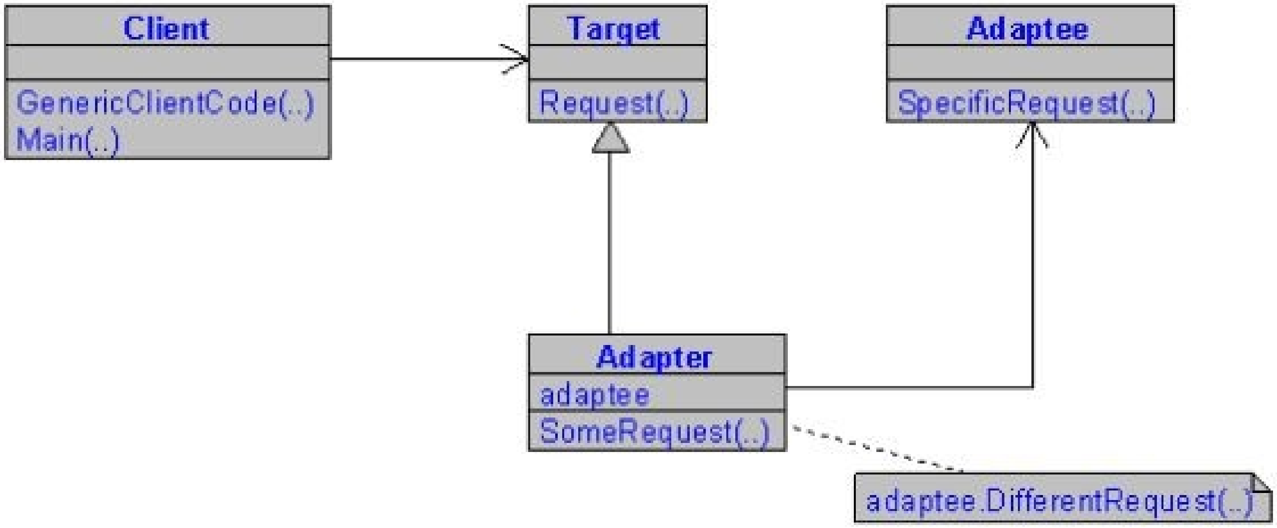
- Characteristics
  - wraps around existing code
  - sits between client and legacy code, providing legacy services with a new interface



©2018 Christine Laurendeau, based on Bruegge and Dutoit, Object-Oriented Software Engineering, 3rd ed., Pearson, 2010

Adapter (cont.)

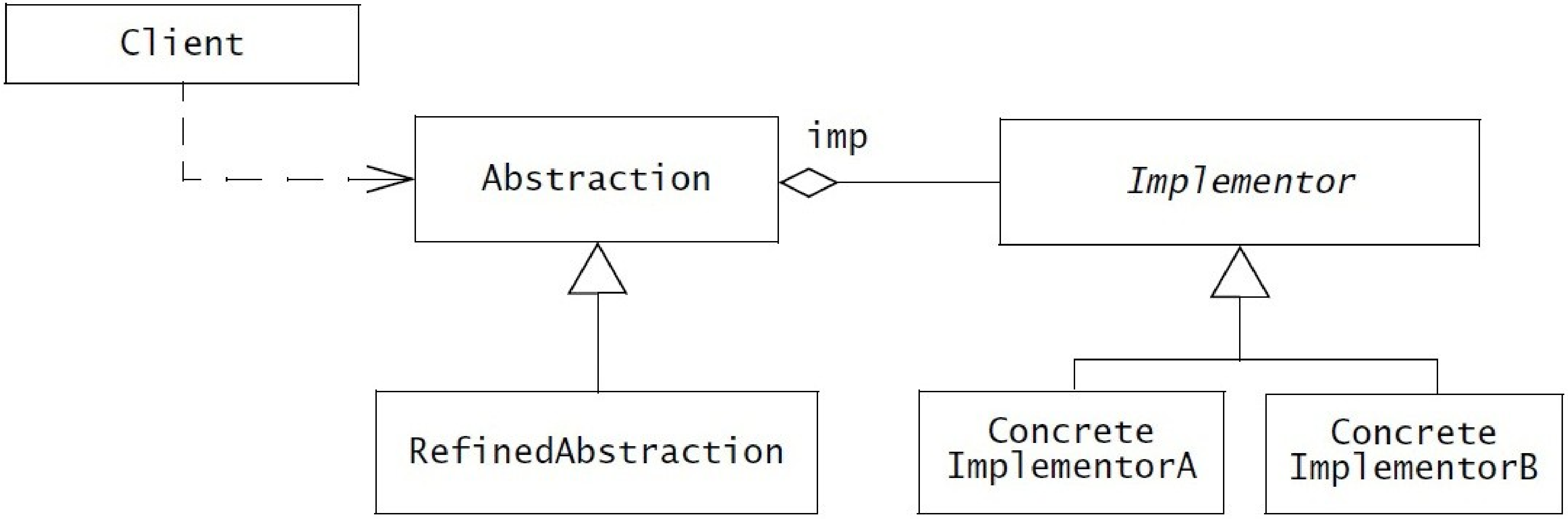
- Solution for encapsulating legacy components:
  - used for converting existing (legacy) component interface into one that the client expects
  - similar to Bridge, but for dealing with existing components
  - example: new UI on an existing back end



©2018 Christine Laurendeau, based on Bruegge and Dutoit, Object-Oriented Software Engineering, 3rd ed., Pearson, 2010

Bridge

- Characteristics
  - allows for alternate implementation, with a single interface



©2018 Christine Laurendeau, based on Bruegge and Dutoit, Object-Oriented Software Engineering, 3rd ed., Pearson, 2010

Bridge (cont.)

- Solution for encapsulating data stores:
  - used for substituting multiple realizations of the same interface for different uses
  - example: multiple implementations of a data store

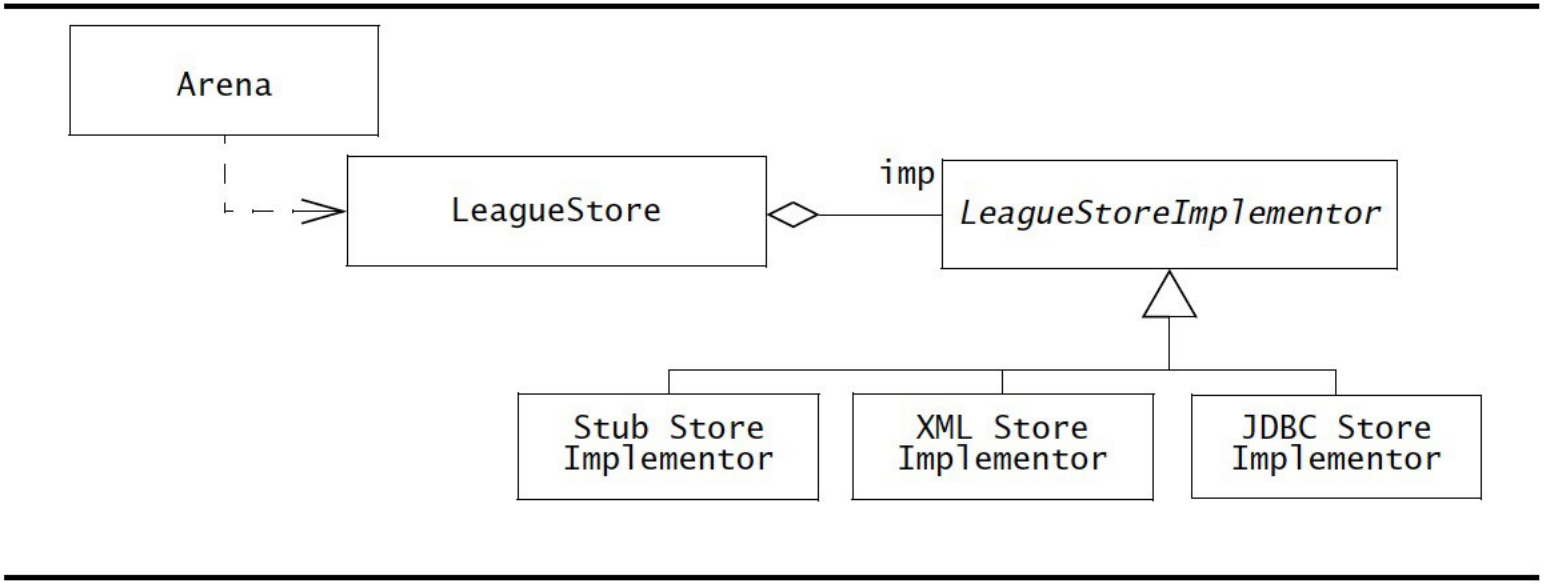
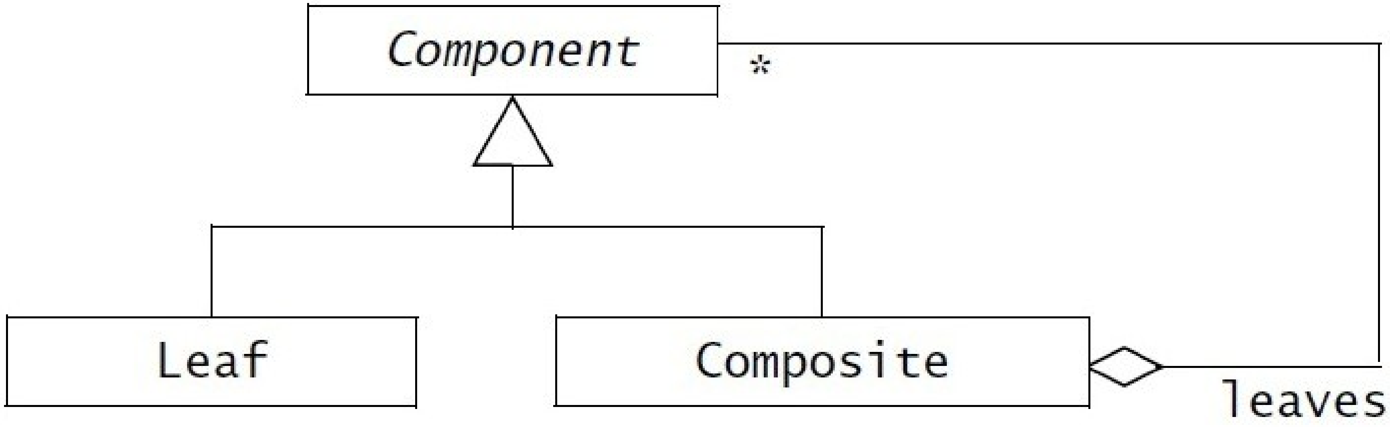


Figure 8-7 Applying the Bridge design pattern for abstracting database vendors (UML class diagram).

Composite

- Characteristics
  - represents a recursive hierarchy
  - leaves and composites provide a common interface
  - commands on composites propagated recursively over all its components



Composite (cont.)

- Solution for encapsulating hierarchies:
  - used for representing recursive hierarchy, such as components and composites
  - example: UI toolkits, such as Java Swing

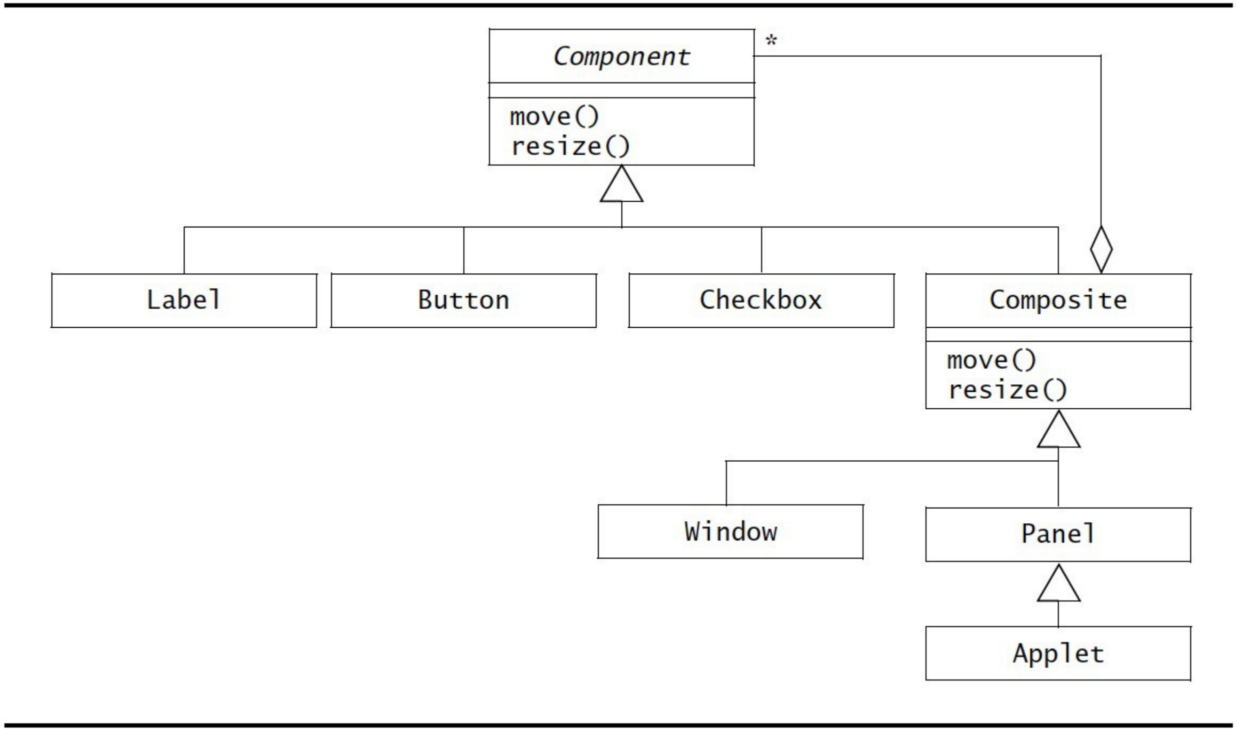
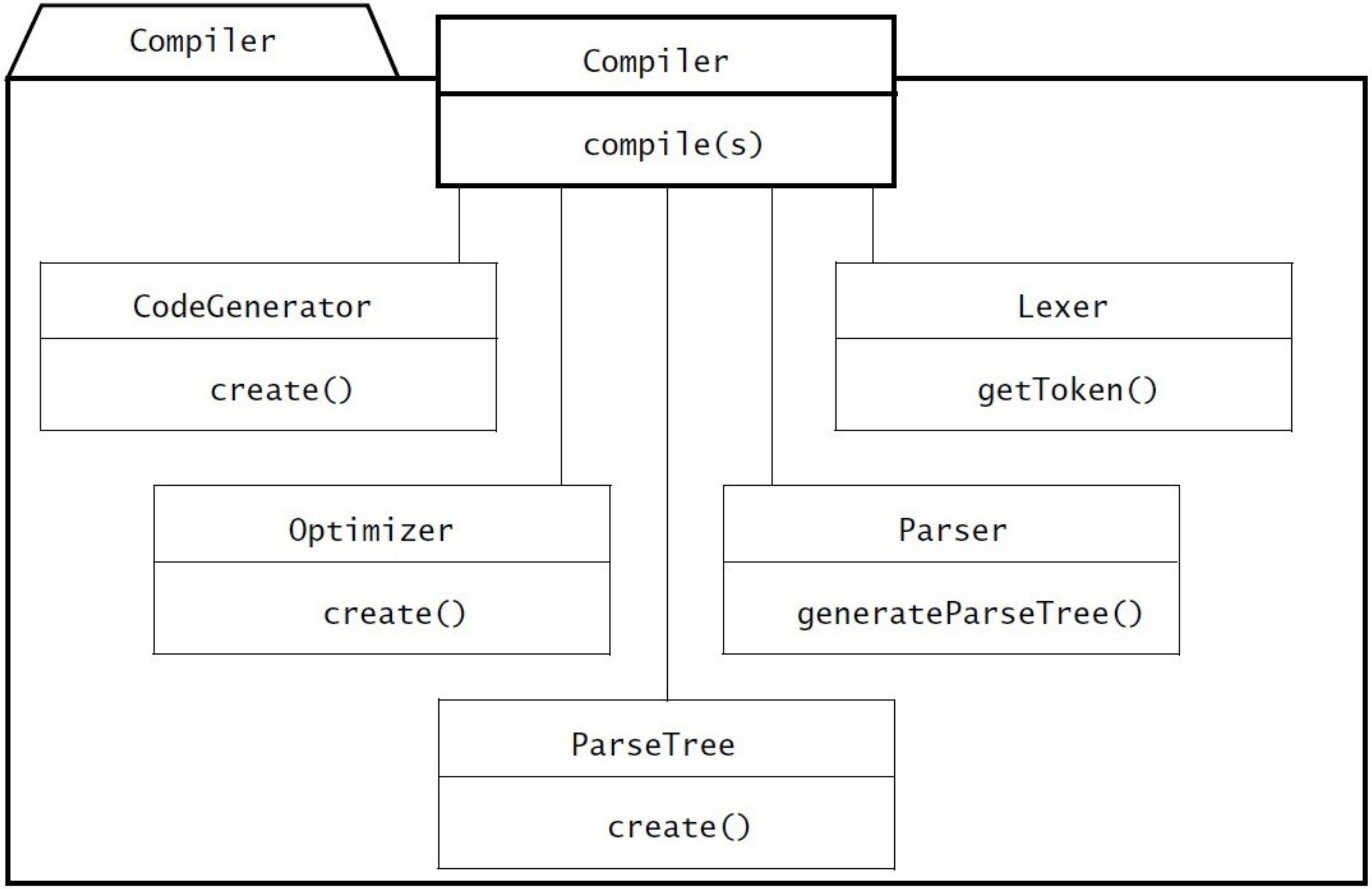


Figure 8-16 Applying the Composite design pattern to user interface widgets (UML class diagram). The Swing Component hierarchy is a Composite in which leaf widgets (e.g., Checkbox, Button, Label) specialize the Component interface, and aggregates (e.g., Panel, Window) specialize the Composite abstract class. Moving or resizing a Composite impacts all of its children.

Façade

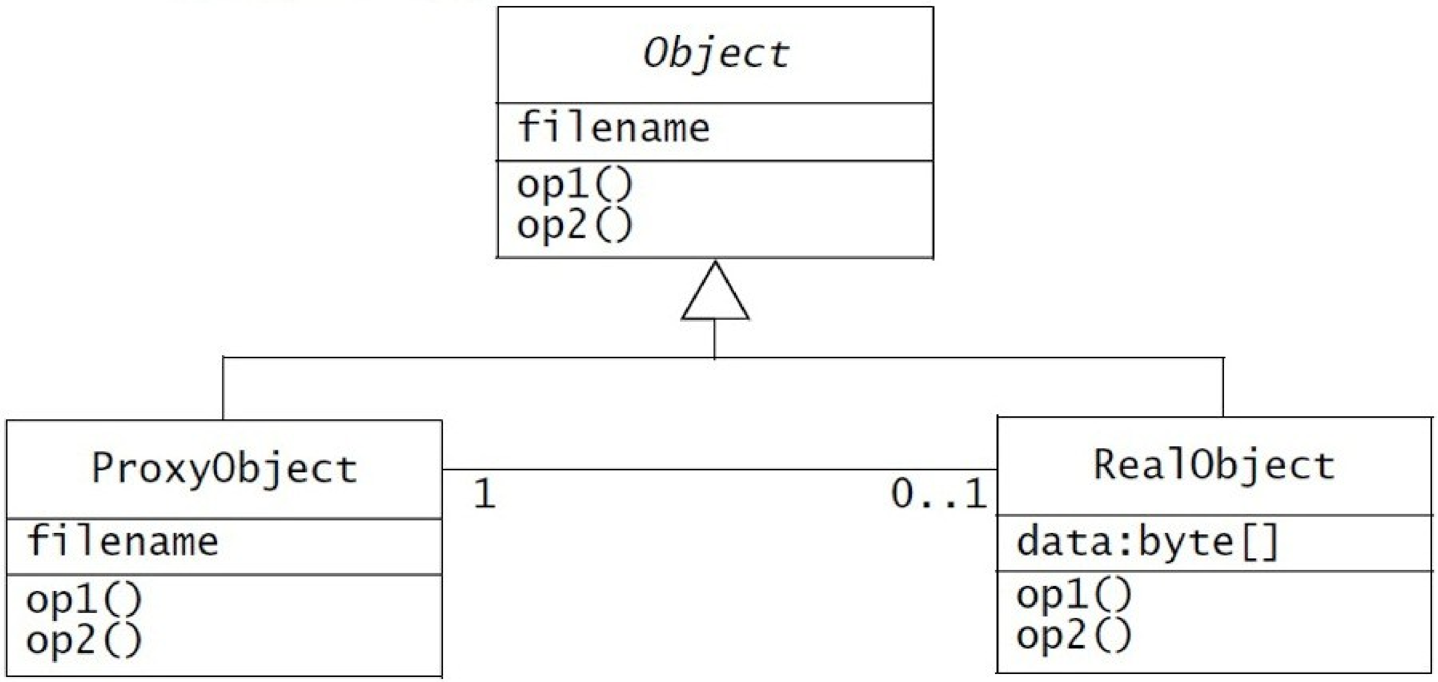
- Characteristics
  - used to encapsulate subsystems
  - provides high-level interface that uses lower-level class operations



©2018 Christine Laurendeau, based on Bruegge and Dutoit, Object-Oriented Software Engineering, 3rd ed., Pearson, 2010

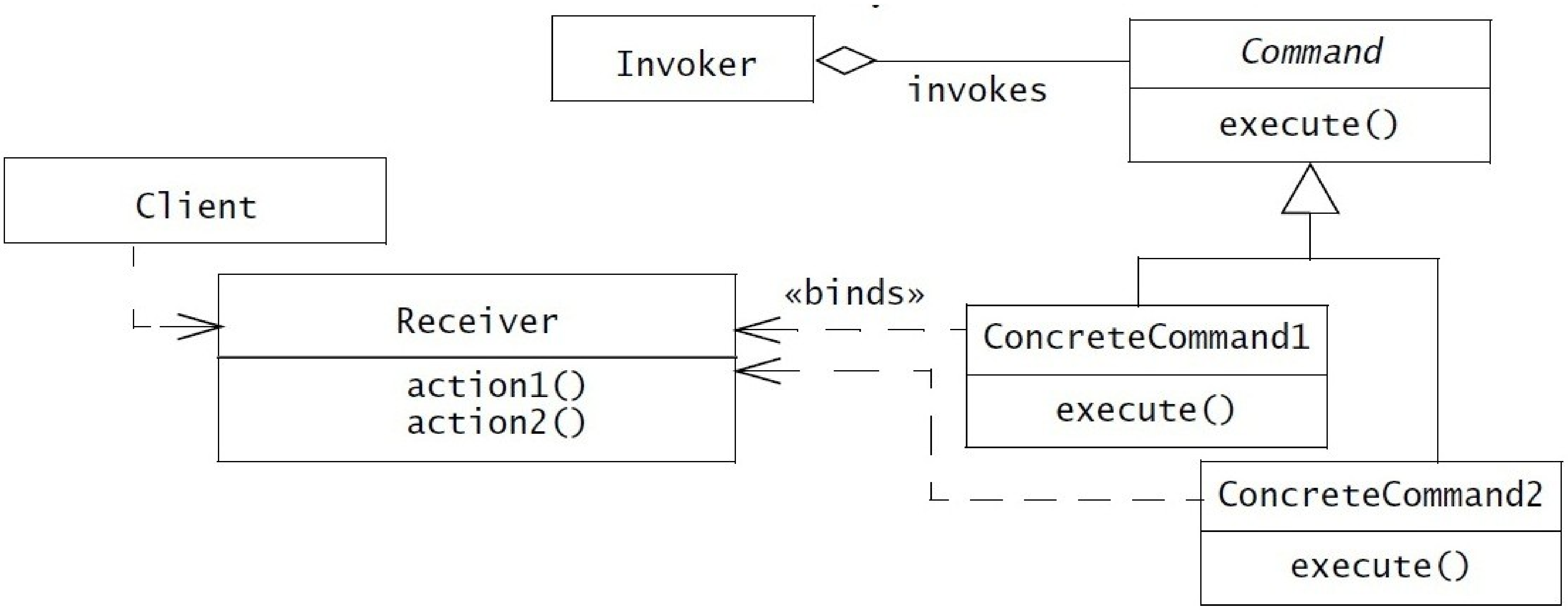
Proxy

- Characteristics
  - encapsulates expensive (performance-wise, security-wise) objects
  - proxy objects provide a gateway to their corresponding real objects



Command

- Characteristics
  - used to encapsulate control flow
  - provides interface that groups operations on all requests



©2018 Christine Laurendeau, based on Bruegge and Dutoit, Object-Oriented Software Engineering, 3rd ed., Pearson, 2010

Command (cont.)

- Solution for encapsulating control flow:
  - used for providing generic user requests, without knowing content of request
  - example: execute, undo, store

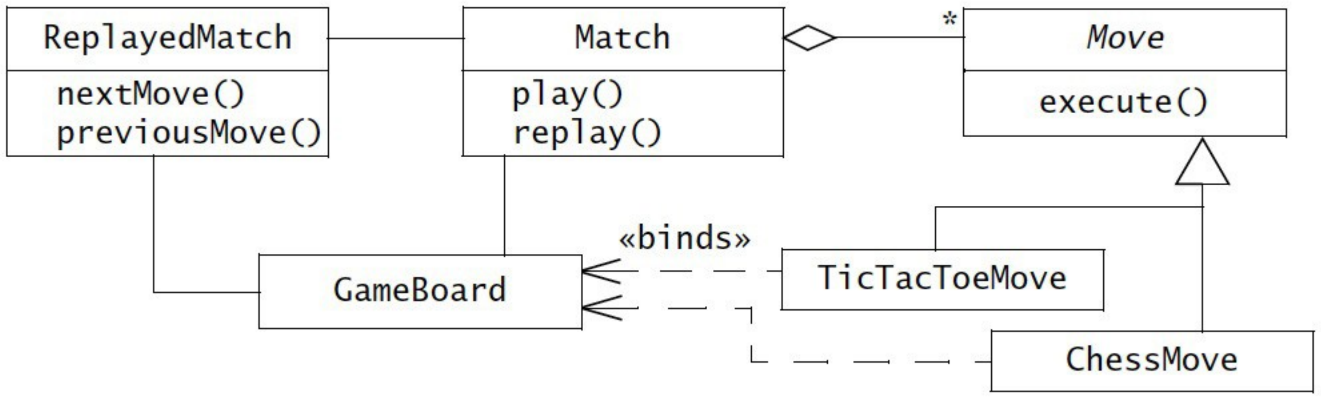
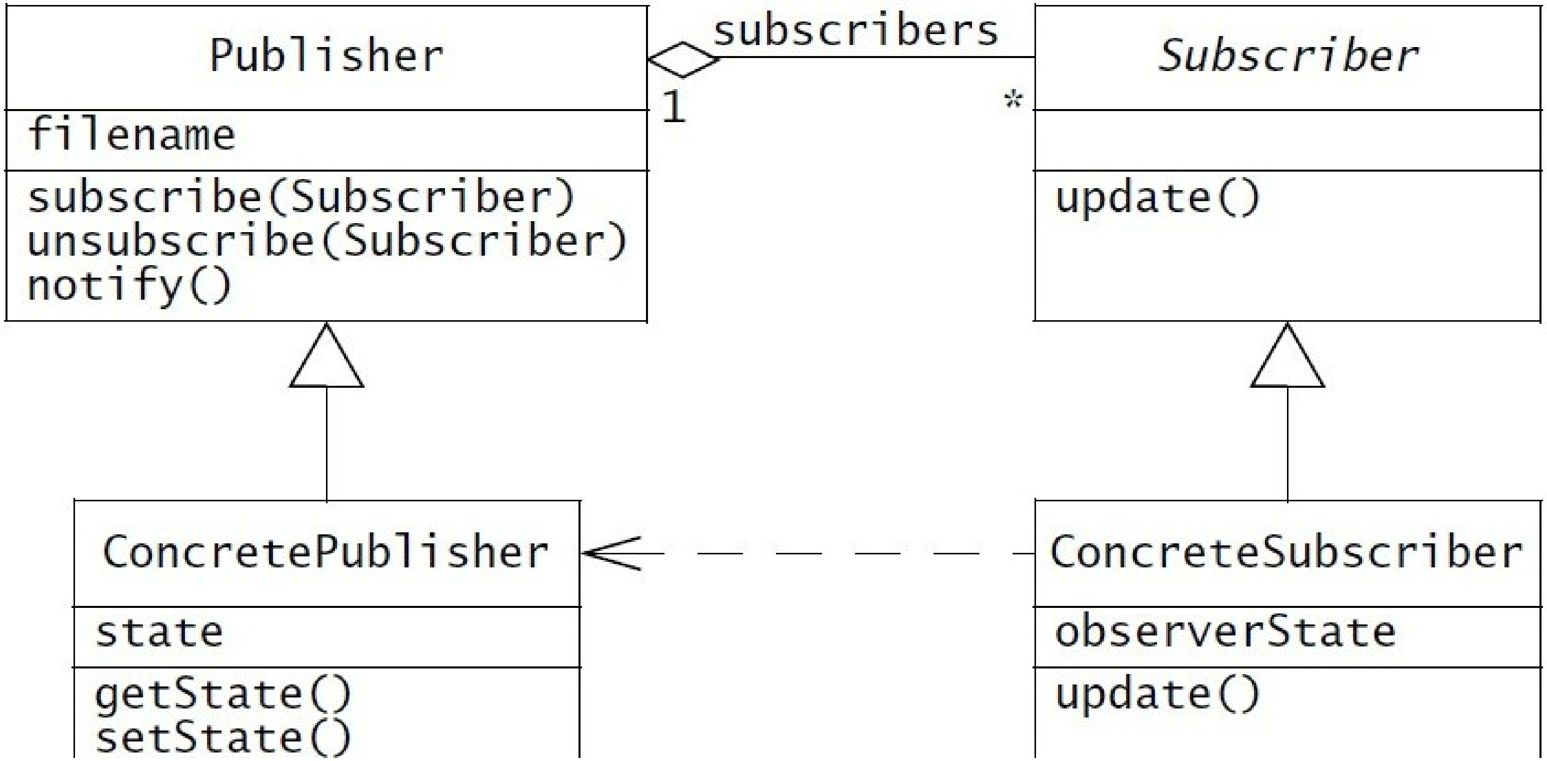


Figure 8-21 Applying the Command design pattern to Matches and ReplayedMatches in ARENA (UML class diagram).

Observer

- Characteristics
  - used to separate entity objects from view
  - changes to one object (publisher/subject) are communicated to interested parties (subscriber/observer)



Observer (cont.)

- Solution for maintaining consistency:
  - used for propagating model changes across views
  - example: MVC architecture

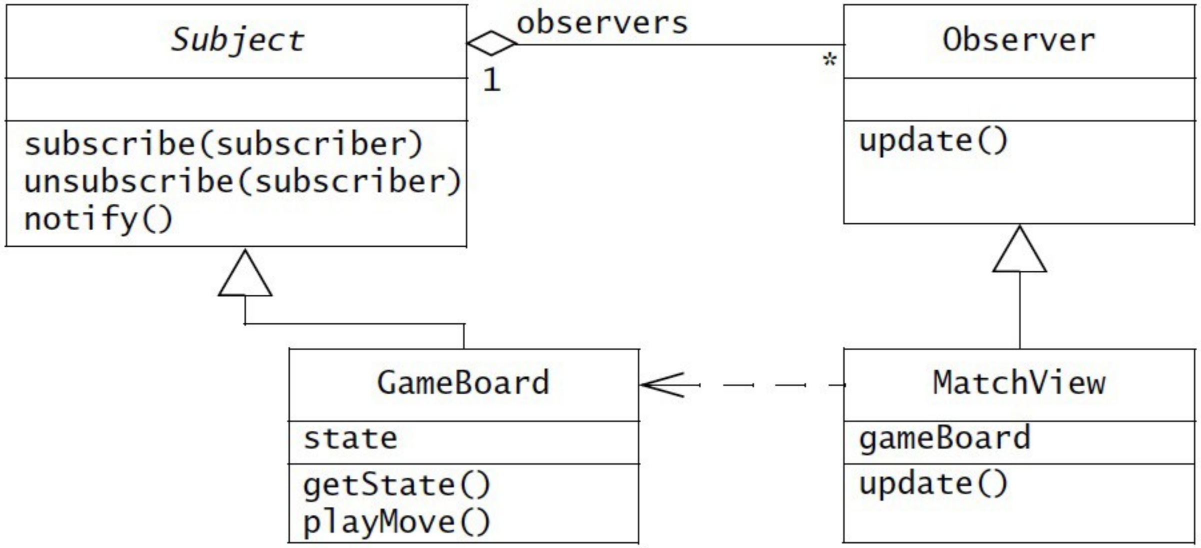
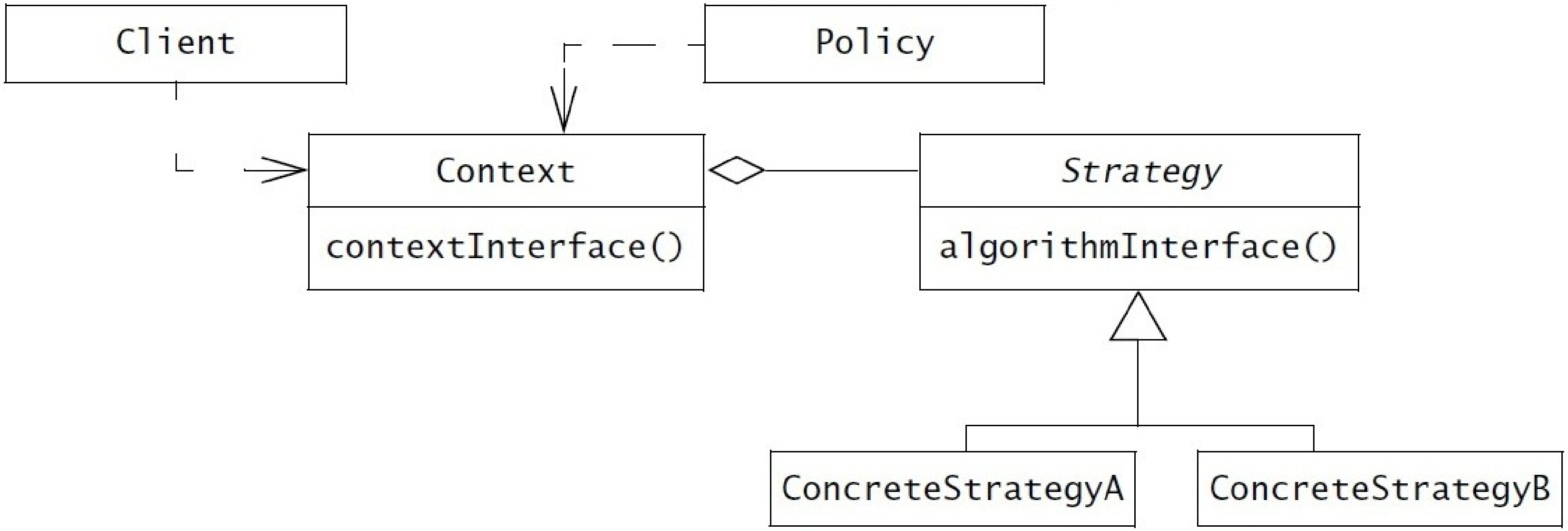


Figure 8-22 Applying the Observer design pattern to maintain consistency across MatchViews (UML class diagram).

Strategy

- Characteristics
  - used to encapsulate algorithms
  - separate policy decides which algorithm performs a task



Strategy (cont.)

- Solution for encapsulating context:
  - used for dynamically substituting multiple realizations of the same interface for different contexts
  - similar to Bridge, but client decides which implementation to use
  - example: substituting different network connections dynamically

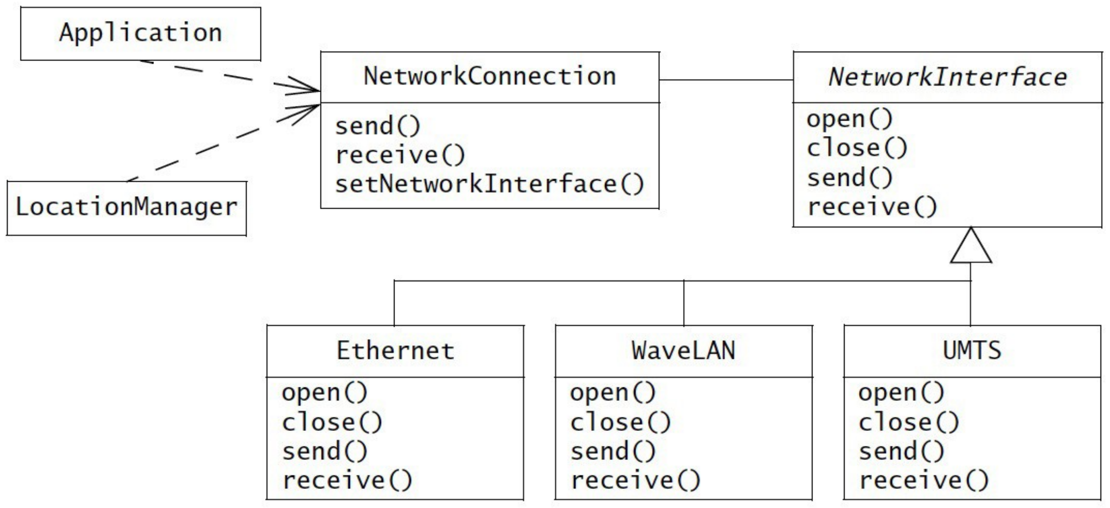


Figure 8-10 Applying the Strategy pattern for encapsulating multiple implementations of a NetworkInterface (UML class diagram). The LocationManager implementing a specific policy configures NetworkConnection with a concrete NetworkInterface (i.e., the mechanism) based on the current location. The Application uses the NetworkConnection independently of concrete NetworkInterfaces. See corresponding Java code in Figure 8-11.