Section 4.2
Reusing Pattern Solutions

1. Overview
2. Concepts
3. Activities

4.2.1 Overview

- Learning outcomes
  - understand the need for code reuse
  - review the major design patterns

Overview (cont.)

- Why reuse existing code or design strategies?
  - never reinvent the wheel
    - applies to design as well as code
  - reuse is faster (and cheaper) than designing from scratch
  - allows for:
    - modifiability
    - extensibility

Overview (cont.)

- How can we incorporate reuse in our system?
  - we have to know what's available out there
    - class libraries
    - security tools
    - other useful existing components
  - we have to know where and how to look
  - we have to understand, use, adapt existing design patterns
    - these are tried and true

4.2.2 Reuse Concepts

- Application objects and solution objects
- Specification inheritance and implementation inheritance
- Delegation
- Liskov substitution principle
- Delegation and inheritance in design patterns

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

Application and Solution Objects

- What are *application objects*?
  - also known as domain objects, or application domain objects
  - they represent concepts from the application domain
  - they must be relevant to the system

- What are *solution objects*?
  - objects specific to the solution domain
  - they do not have an application domain counterpart
  - for example: persistent data stores, UI objects

Application and Solution Objects (cont.)

- Identifying objects throughout development life-cycle
  - analysis
    - identify application objects
      - these are usually entity objects
    - identify solution objects visible to the user
      - boundary objects
      - control objects
  - system design
    - identify solution objects in terms of hardware/software platforms
  - object design
    - refine application and solution objects
    - identify new solution objects

Specification and Implementation Inheritance

- Focus of inheritance in analysis phase
  - set up generalization/specialization taxonomy
- Focus of inheritance in object design phase
  - reuse
    - reduce redundancy
    - enhance extensibility
- Impact of inheritance on coupling
  - decouples client classes using superclass from subclasses
  - introduces strong coupling between superclass and subclasses

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

Specification and Implementation Inheritance (cont.)

- What is *specification inheritance*?
  - use of inheritance to classify concepts into type hierarchies
  - “is-a” relationship between generalized and specialized classes
  - this is the “usual” kind of inheritance

Specification and Implementation Inheritance (cont.)

- What is *implementation inheritance*?
  - it's **not** an "is-a" relationship
  - it uses inheritance purely for the purpose of code reuse
  - the superclass functionality is reused by:
    - subclassing from the superclass, and
    - refining the superclass behaviour in the subclasses
  - it's a quick and dirty way to reuse operations
    - it usually results in unintended consequences
    - you often get more than you bargained for
  - it's not an intuitive use of inheritance

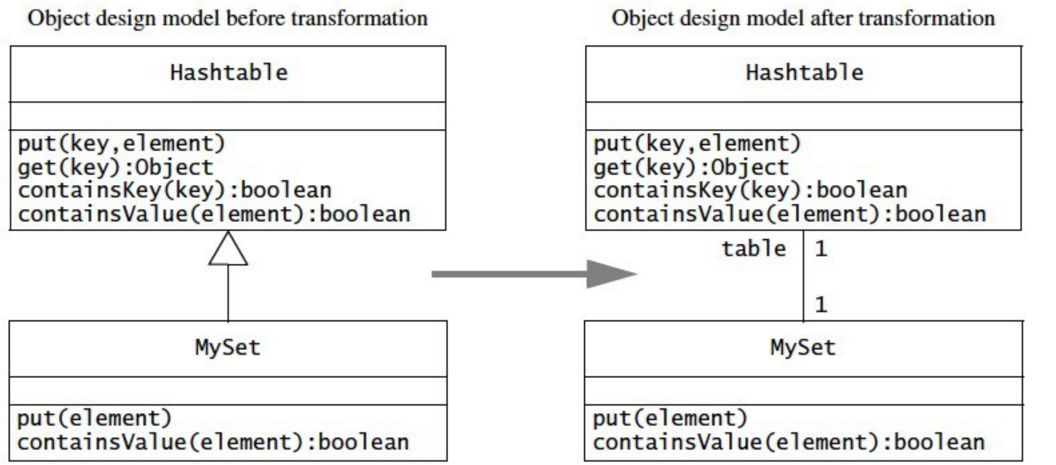
- Example: container class

Delegation

- What is delegation?
  - it’s an alternative to implementation inheritance
  - an operation re-sends a message to another class
    - also called “pass-the-buck”, “double dispatching”

- Characteristics
  - it makes explicit the dependencies between:
    - a reused class, and
    - a new class
  - preferable to implementation inheritance
    - special case: private inheritance in C++

Delegation (cont.)




```java
/* Implementation of MySet using
inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void put(Object element) {
        if (!containsKey(element)){
            put(element, this);
        }
    }
    boolean containsValue(Object
            element){
        return containsKey(element);
    }
    /* Other methods omitted */
}
```

```java
/* Implementation of MySet using
delegation */
class MySet {
    private Hashtable table;
    MySet() {
        table = Hashtable();
    }
    void put(Object element) {
        if (!containsValue(element)){
            table.put(element,this);
        }
    }
    boolean containsValue(Object
            element) {
        return
        (table.containsKey(element));
    }
    /* Other methods omitted */
}
```

Figure 8-3 An example of implementation inheritance. The left column depicts a questionable implementation of `MySet` using implementation inheritance. The right column depicts an improved implementation using delegation (UML class diagram and Java).

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Liskov Substitution Principle

- What is this principle?
  - assume T is a superclass and S is a subclass of T
  - “if an object of type S can be substituted in all places where an object of type T is expected, then S is a sub-type of T”
  - consequences:
    - an operation on T can be called on instances of S, without knowing that it is called on a subclass instance
    - client classes using operations on T don’t have to change when new subclasses of T are added

- *Strict inheritance*
  - it’s when all inheritance associations are specification inheritance

Delegation and Inheritance in Design Patterns

- Design patterns
  - they are template solutions to recurring design problems
  - a set of classes that provide partial solution to common problems
- Characteristics
  - they are 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.

- Most design patterns use both inheritance and delegation

4.2.3 Reuse Activities

- Goals in design activities:
  - modifiability
  - extensibility
- Challenge:
  - unanticipated changes during development
  - correct use of design patterns can minimize the effects of change

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

Sources of Change During Development

- New vendor or new technology
  - vendors go out of business, components become unavailable
  - design patterns can isolate the code that uses these components
  - possible patterns: Bridge, Adapter, Strategy, Abstract Factory

- New implementation
  - system performance is difficult to predict at the design stage
  - possible patterns: Bridge, Adapter, Strategy

Sources of Change During Development (cont.)

- New views
  - usability problems translate into additional views on the same data
  - design patterns can manage updates of multiple views
  - possible pattern: Observer

- New complexity of application domain
  - generalizations may be realized late in development
  - possible pattern: Composite, to encapsulate class hierarchies

- Errors
  - requirement errors
  - missing requirements

Design Pattern Example Uses

Table 8-1 Selected design patterns and the changes they anticipate.

Design Pattern	Anticipated Change	References
Bridge	*New vendor, new technology, new implementation.* This pattern decouples the interface of a class from its implementation. It serves the same purpose as the Adapter pattern except that the developer is not constrained by an existing component.	Section 8.4.1 Appendix A.3
Adapter	*New vendor, new technology, new implementation.* This pattern encapsulates a piece of legacy code that was not designed to work with the system. It also limits the impact of substituting the piece of legacy code for a different component.	Section 8.4.2 Appendix A.2
Strategy	*New vendor, new technology, new implementation.* This pattern decouples an algorithm from its implementation(s). It serves the same purpose as the Adapter and Bridge patterns, except that the encapsulated unit is a behavior.	Section 8.4.3 Appendix A.9
Abstract Factory	*New vendor, new technology.* Encapsulates the creation of families of related objects. This shields the client from the creation process and prevents the use of objects from different (incompatible) families.	Section 8.4.4 Appendix A.1
Command	*New functionality.* This patterns decouples the objects responsible for command processing from the commands themselves. This pattern protects these objects from changes due to new functionality.	Section 8.4.5 Appendix A.4
Composite	*New complexity of application domain.* This pattern encapsulates hierarchies by providing a common superclass for aggregate and leaf nodes. New types of leaves can be added without modifying existing code.	Section 8.4.6 Appendix A.5

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Selecting Design Patterns

- Encapsulating data stores
- Encapsulating legacy components
- Encapsulating context
- Encapsulating platforms
- Encapsulating control flow
- Encapsulating hierarchies
- Maintaining consistency
- Heuristics for selecting design patterns

Encapsulating Data Stores

- Bridge design pattern:
  - it’s a solution for substituting multiple realizations of the same interface for different uses
  - example: multiple implementations of a data store

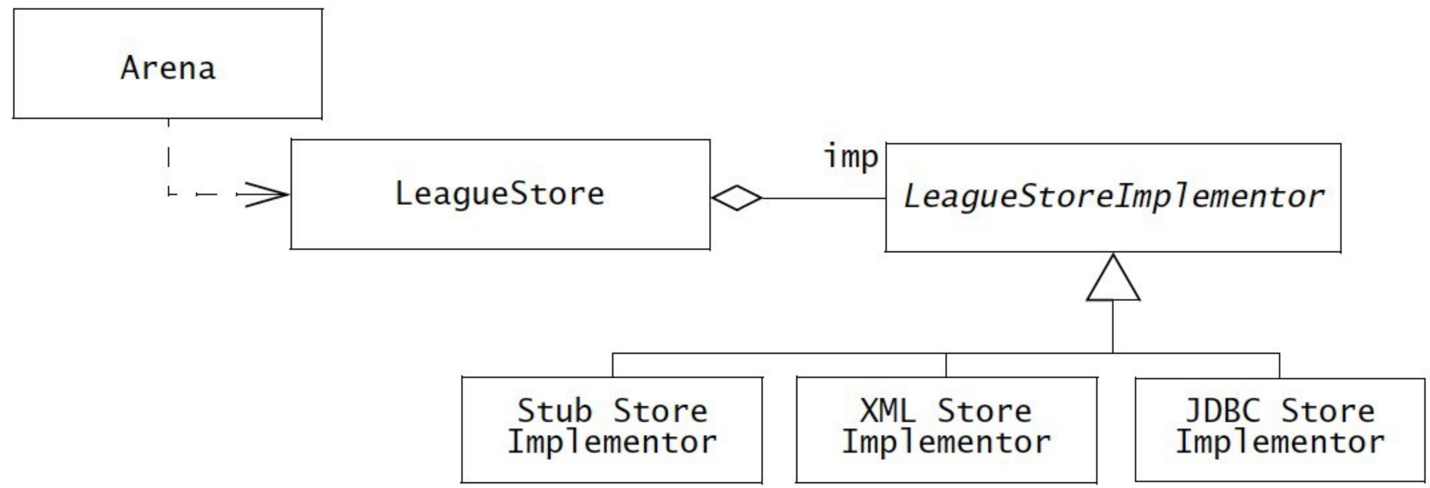
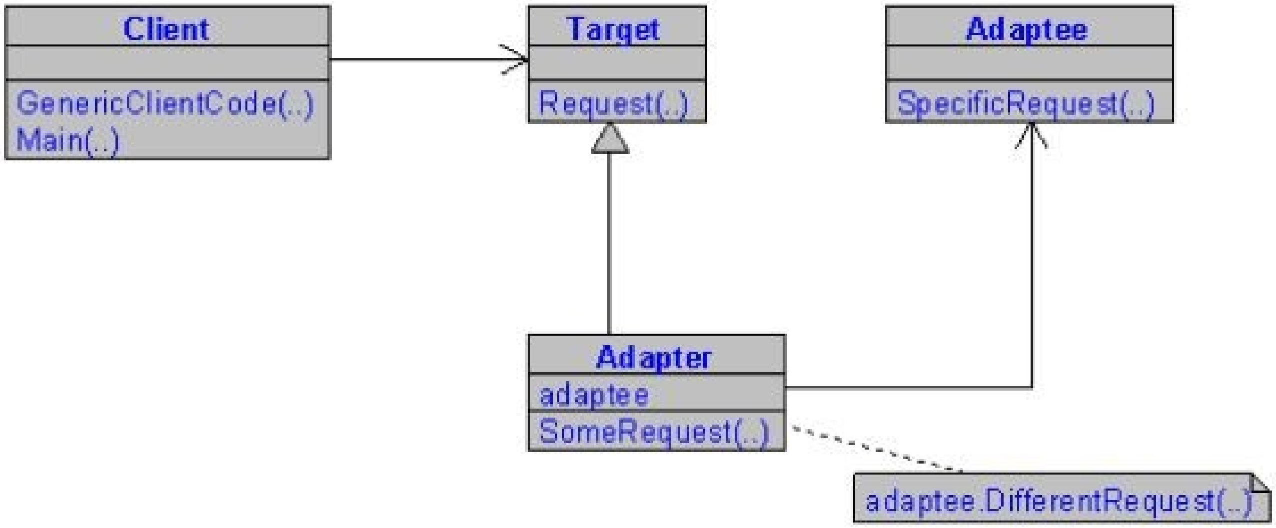


Figure 8-7 Applying the Bridge design pattern for abstracting database vendors (UML class diagram).

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Encapsulating Legacy Components

- Adapter design pattern:
  - it’s a solution 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



Encapsulating Context

- Strategy design pattern:
  - it’s a solution 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

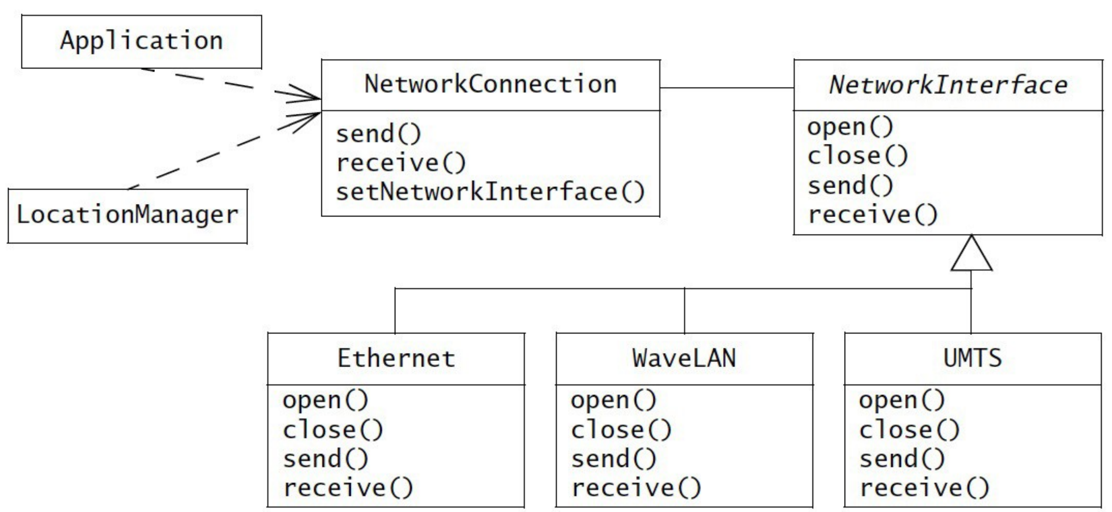


Figure 8-10 Applying the Strategy pattern for encapsulating multiple implementations of a NetworkInterface (UML class diagram). The LocationManager implementing a specific policy configures NetworkConnection with a concrete NetworkInterface (i.e., the mechanism) based on the current location. The Application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

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Encapsulating Platforms

- Abstract factory design pattern:
  - it's a solution for substituting family of concrete products transparently from the client
  - example: application with products from different manufacturers

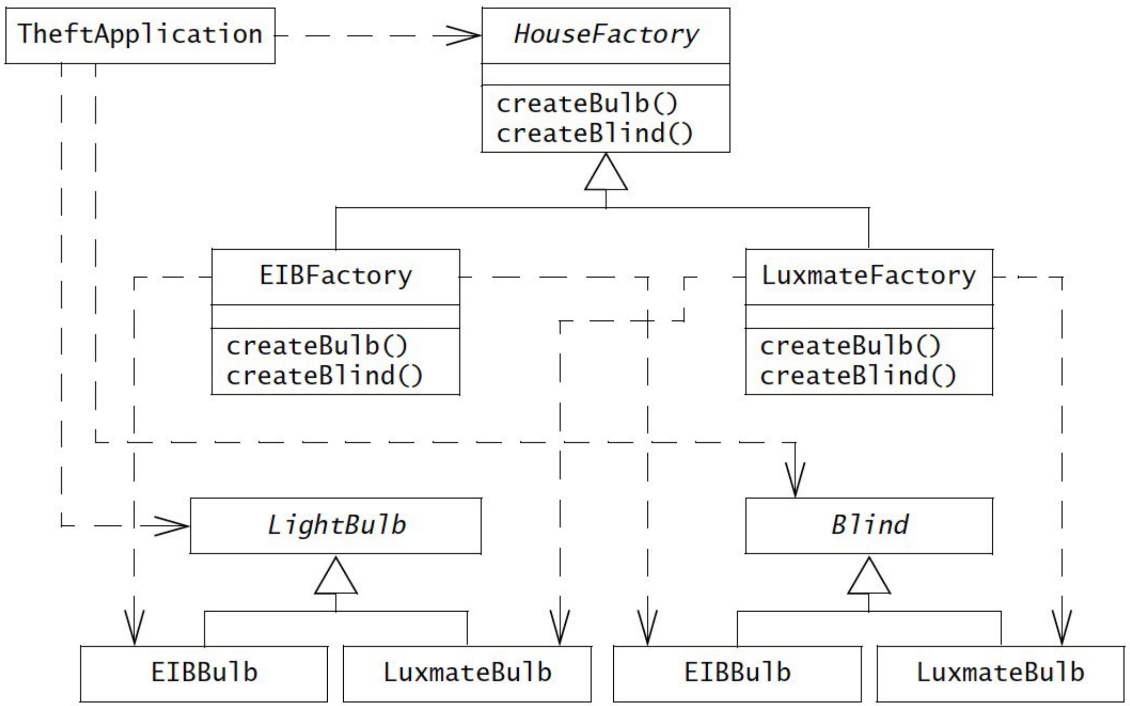


Figure 8-12 Applying the Abstract Factory design pattern to different intelligent house platforms (UML class diagram, dependencies represent «call» relationships).

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Encapsulating Platforms (cont.)

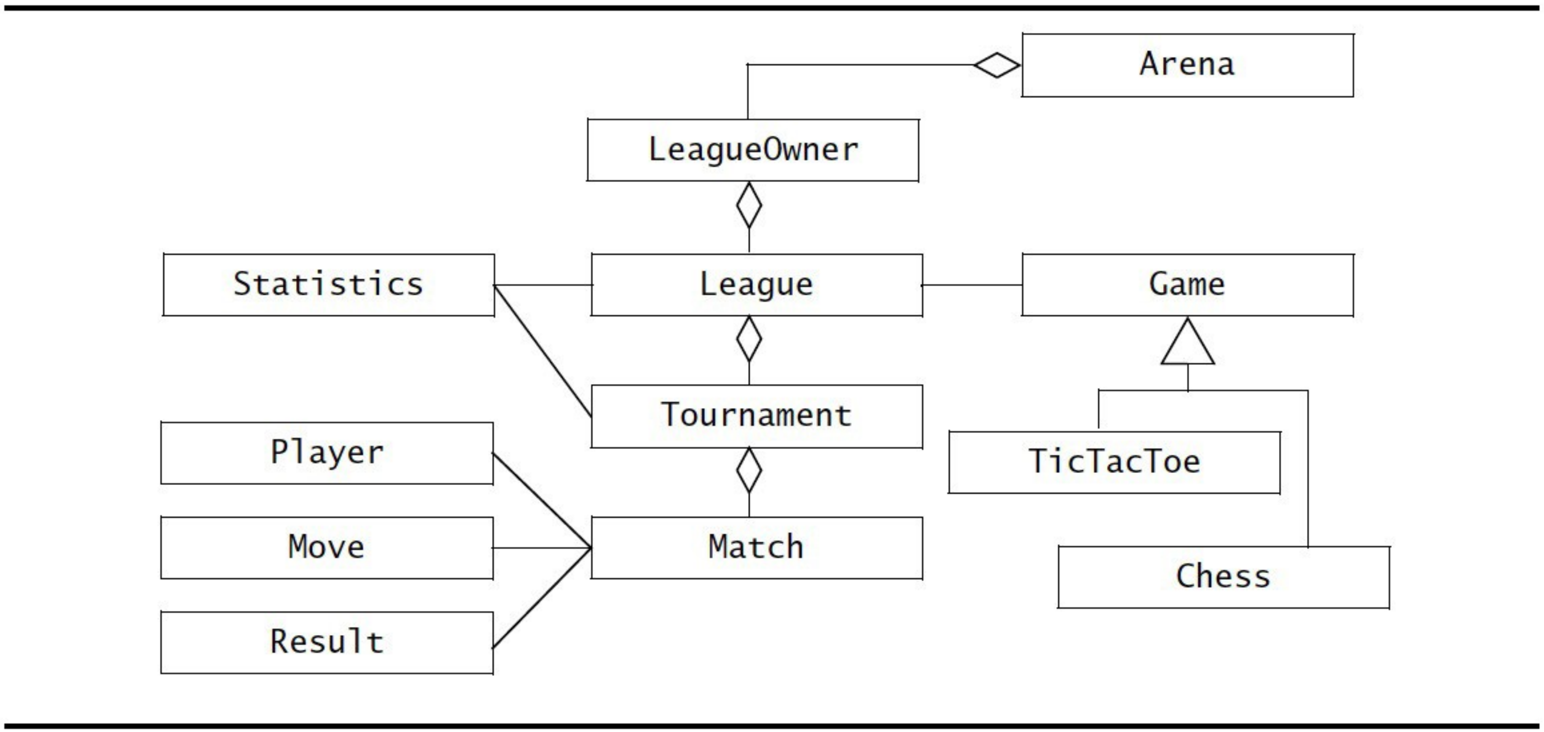


Figure 8-19 ARENA analysis objects related to Game independence (UML class diagram).

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Encapsulating Platforms (cont.)

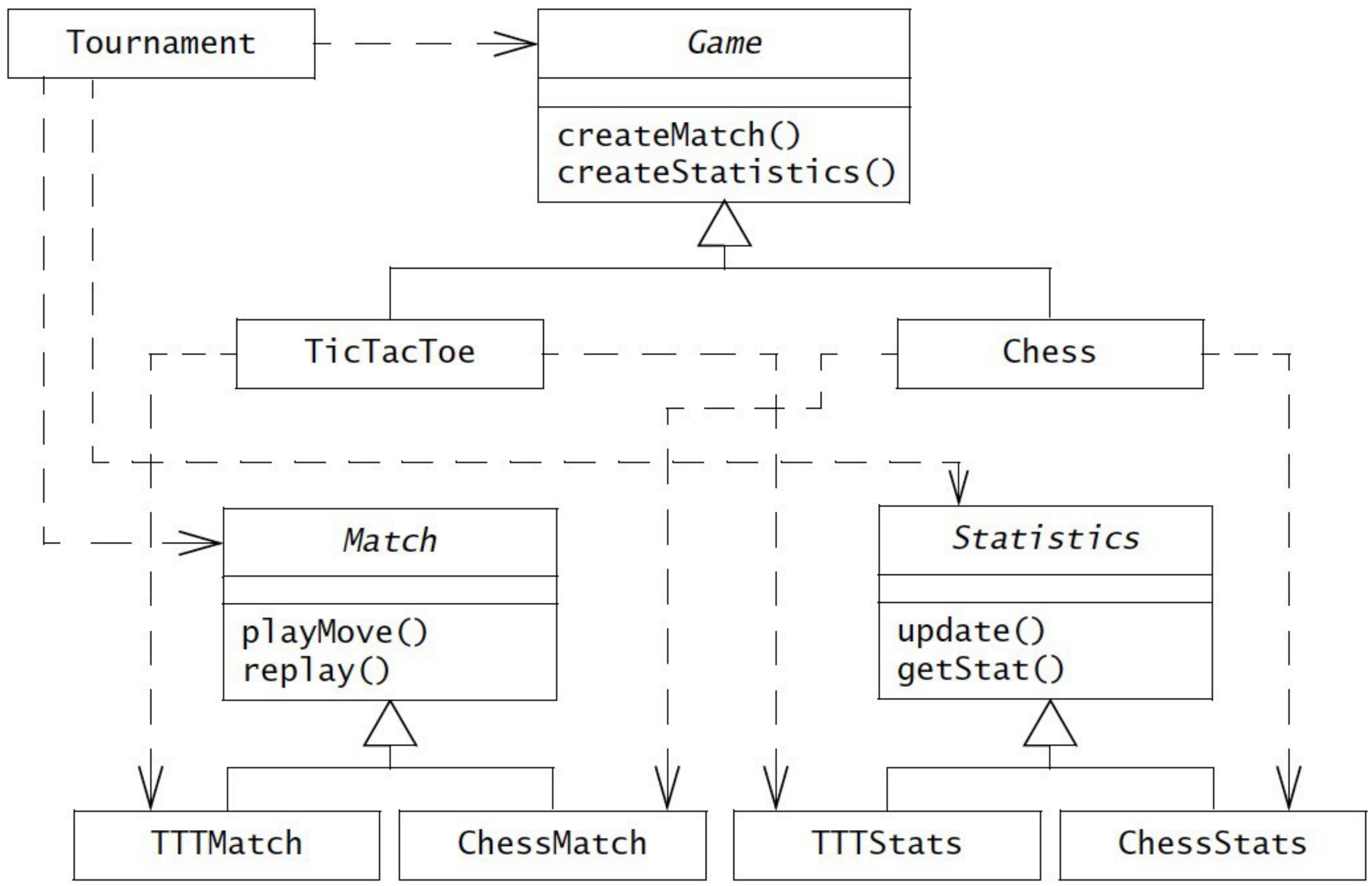


Figure 8-20 Applying the Abstract Factory design pattern to Games (UML class diagram).

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Encapsulating Control Flow

- Command design pattern:
  - it's a solution for providing generic user requests, without knowing content of request
  - example: execute, undo, store

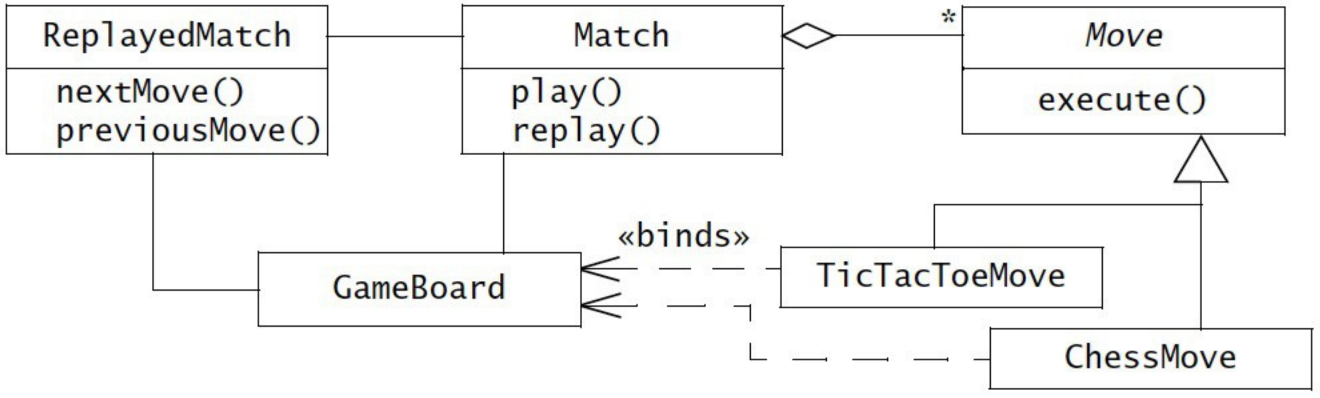


Figure 8-21 Applying the Command design pattern to Matches and ReplayedMatches in ARENA (UML class diagram).

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Encapsulating Hierarchies

- Composite design pattern:
  - it’s a solution for representing a recursive hierarchy, such as components and composites
  - example: UI toolkits, such as Java Swing

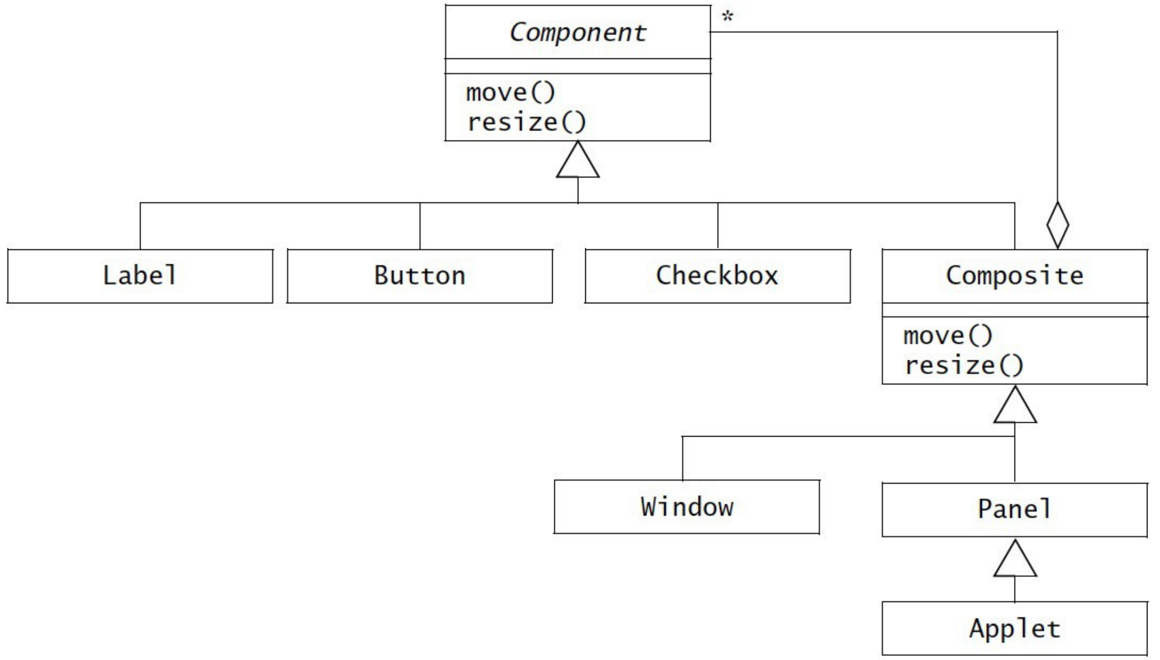


Figure 8-16 Applying the Composite design pattern to user interface widgets (UML class diagram). The Swing Component hierarchy is a `Composite` in which leaf widgets (e.g., `Checkbox`, `Button`, `Label`) specialize the `Component` interface, and aggregates (e.g., `Panel`, `Window`) specialize the `Composite`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

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Maintaining Consistency

- Observer design pattern:
  - it's a solution for propagating model changes across views
  - example: MVC architecture

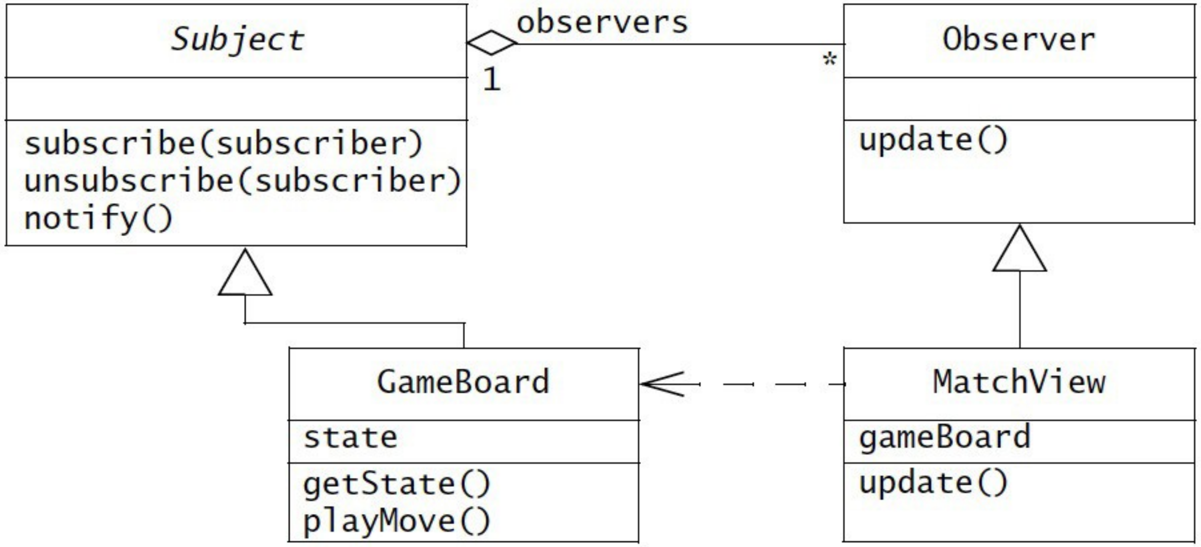


Figure 8-22 Applying the Observer design pattern to maintain consistency across MatchViews (UML class diagram).

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Heuristics for Selecting Design Patterns

- Abstract Factory
  - manufacturer independence
  - platform independence

- Adapter
  - compliance with existing interface
  - reuse of existing legacy component

- Bridge
  - support for future protocols

Heuristics for Selecting Design Patterns (cont.)

- Command
  - all commands should be logged
  - all commands should be un-doable
- Composite
  - support for aggregate structures
  - hierarchies of variable depth and width
- Strategy
  - decoupling of policy and mechanisms
  - interchanges of algorithms at runtime

Recap

- What we learned:
  - understand the need for code reuse
  - review the major design patterns
    - creational, structural, behavioural

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