Object Design

Software Engineering Languages and Tools
Spring 2011

1

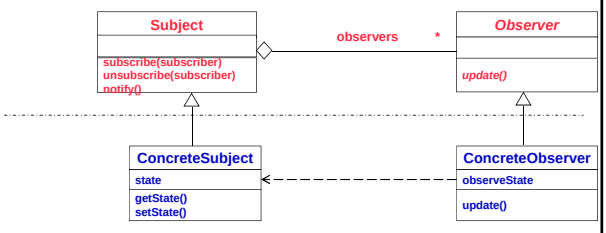
Object Design

- Purpose of object design:
  - Prepare for the implementation of the system model based on design decisions
  - Transform the system model (optimize it)
- Investigate alternative ways to implement the system model
  - Use design goals: minimize execution time, memory and other measures of cost.
- Object design serves as the basis of implementation.

2

Application Domain vs Solution Domain Objects

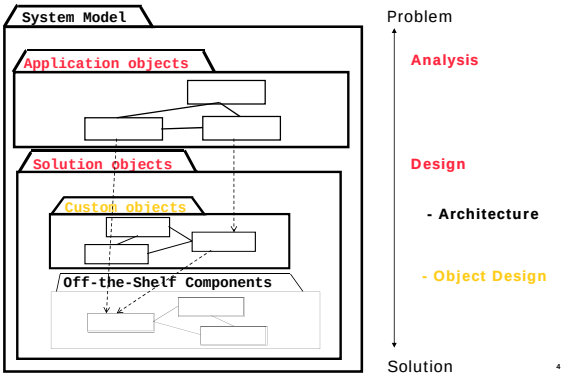
Requirements Analysis (Language of Application Domain)



Object Design (Language of Solution Domain)

3

System Development as a Set of Activities



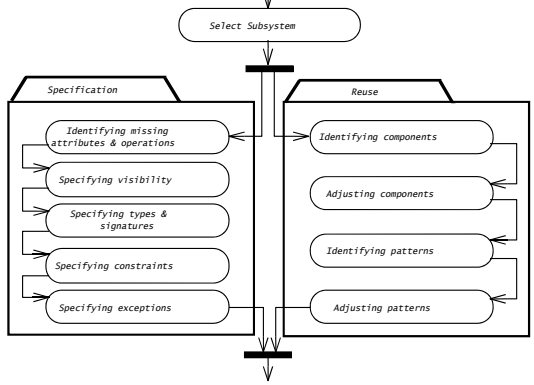
4

Object Design Activities

1. Object identification
   - Objects from application domain
   - New solution domain objects
   - Off-the-shelf components
   - Design patterns
2. Interface specification
   - Describes precisely each class interface
3. Object model restructuring
   - Transforms the object design model to improve its understandability and extensibility
4. Object model optimization
   - Transforms the object design model to address performance criteria such as response time or memory utilization.

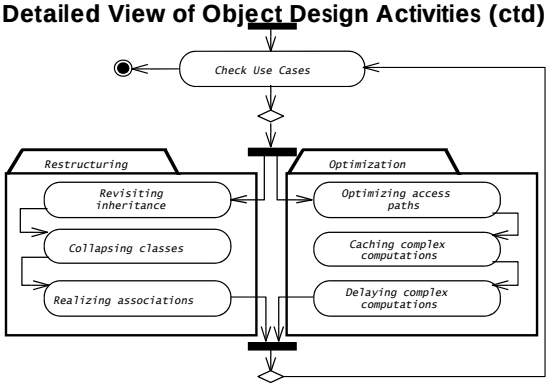
5

Object Design Activities



6

Detailed View of Object Design Activities (ctd)



7

Finding Objects

- The hardest problems in object-oriented system development are:
  - Identifying objects
  - Decomposing the system into objects
- Requirements Analysis focuses on application domain:
  - Object identification
- System Design (architectural design) addresses both, application and implementation domain:
  - Subsystem Identification
- Object Design focuses on implementation domain:
  - Additional solution objects

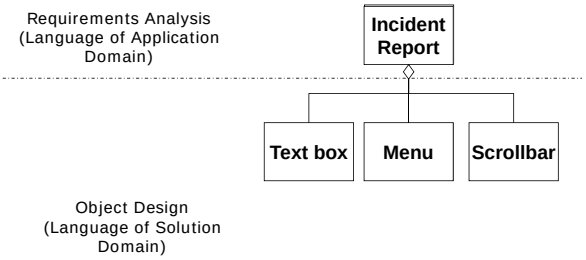
8

Techniques for Finding Objects

- Requirements Analysis
  - Start with Use Cases. Identify participating objects
  - Textual analysis of flow of events (find nouns, verbs, ...)
  - Extract application domain objects by interviewing client (application domain knowledge)
  - Find objects by using general knowledge
- System Design
  - Subsystem decomposition
  - Try to identify layers and partitions
- Object Design
  - Find additional objects by applying implementation domain knowledge

9

Identification of new Objects during Object Design



10

Other Reasons for new Objects

- The implementation of algorithms may necessitate objects to hold values
- New low-level operations may be needed during the decomposition of high-level operations
- Example: `EraseArea()` in a drawing program
  - Conceptually very simple
  - Implementation is complicated:
    - `Area` represented by pixels
    - We need a `Repair()` operation to clean up objects partially covered by the erased area
    - We need a `Redraw()` operation to draw objects uncovered by the erasure
    - We need a `Draw()` operation to erase pixels in background color not covered by other objects.

11

Another Source for Finding Objects : Design Patterns

- What are Design Patterns?
  - A design pattern describes a problem which occurs over and over again in our environment
  - Then it describes the core of the solution to that problem, in such a way that you can use the this solution a million times over, without ever doing it the same twice

12

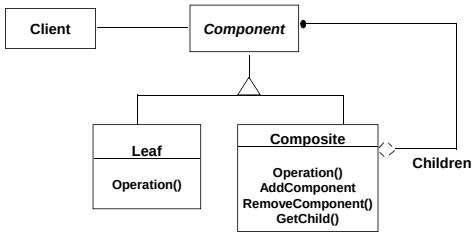
Design Patterns:

- Design patterns are partial solutions to common problems such as
  - such as separating an interface from a number of alternate implementations
  - wrapping around a set of legacy classes
  - protecting a caller from changes associated with specific platforms
- A design pattern consists of a small number of classes
  - uses delegation and inheritance
  - provides a modifiable design solution
- These classes can be adapted and refined for the specific system under construction
  - Customization of the system
  - Reuse of existing solutions.

13

Introducing the Composite 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



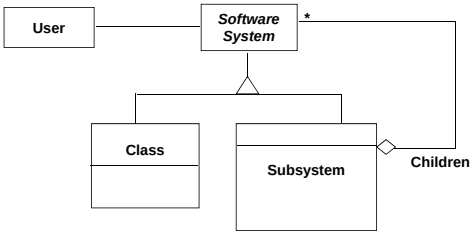
14

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 actitivies or collection of tasks
  - Composite: Activity (The software lifecycle consists of activities which consist of activities, which consist of activities, which….)
  - Leaf node: Task.

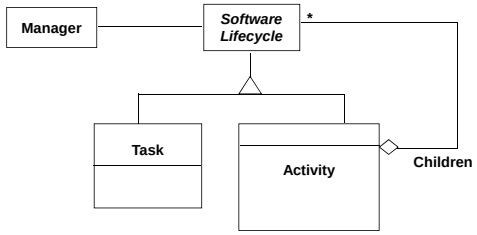
15

Modeling a Software System with a Composite Pattern



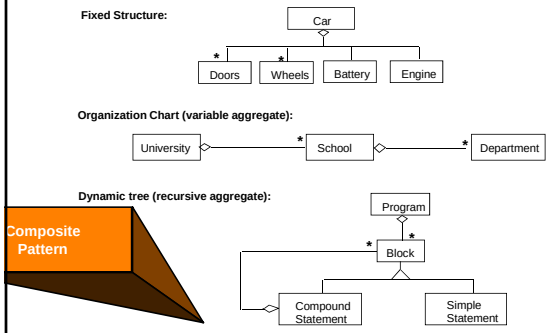
16

Modeling the Software Lifecycle with a Composite Pattern



17

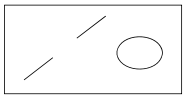
The Composite Patterns models dynamic aggregates

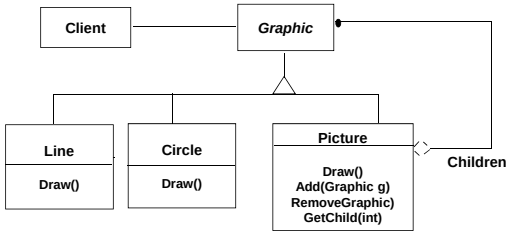


18

Graphic Applications also use Composite Patterns

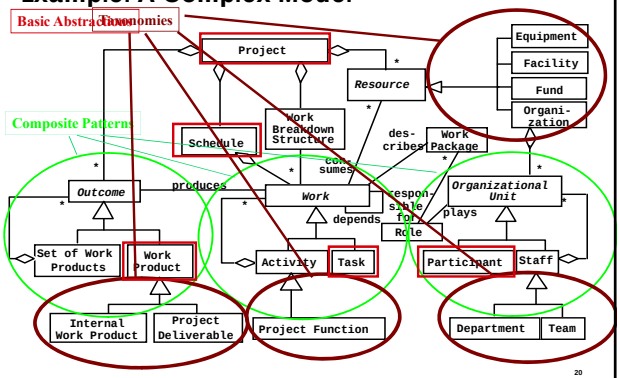
• The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)





19

Example: A Complex Model

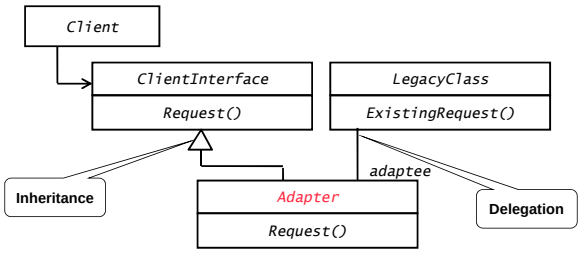


20

Many design patterns use a combination of inheritance and delegation

21

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 Adaptee**

22

Adapter Pattern

- 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
  - Object adapter:
    - Uses single inheritance and delegation
- Object adapters are much more frequent.
- We cover only object adapters (and call them adapters).

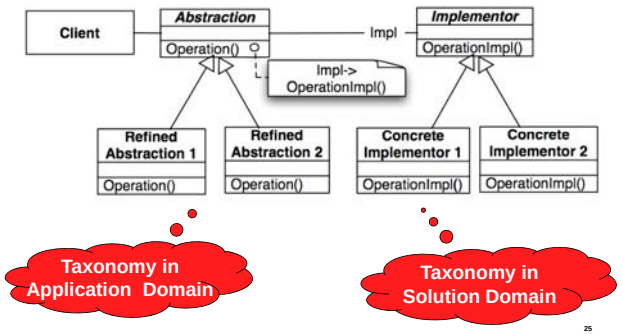
23

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.

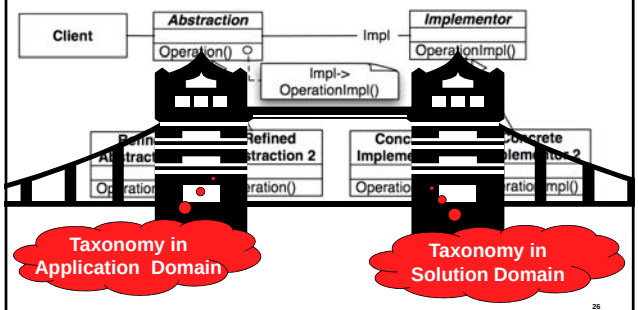
24

Bridge Pattern



25

Why the Name Bridge Pattern?



26

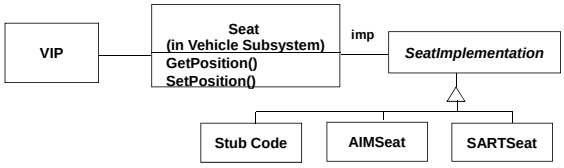
Motivation for the Bridge Pattern

- 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)

27

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:



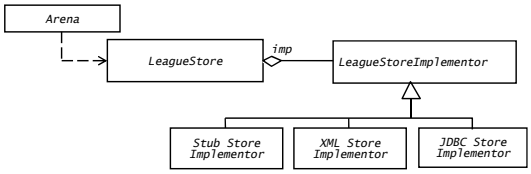
28

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
 }
 ...
}
```

29

Another use of the Bridge Pattern: Support multiple Database Vendors



30

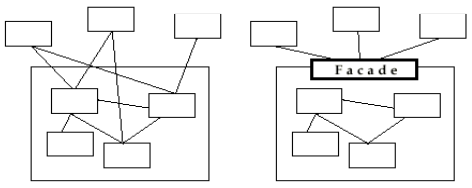
Adapter vs Bridge

- 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"

31

Facade Pattern

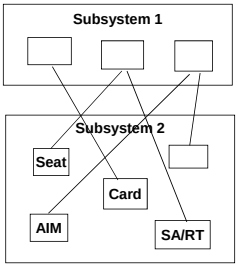
- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
-



32

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



33

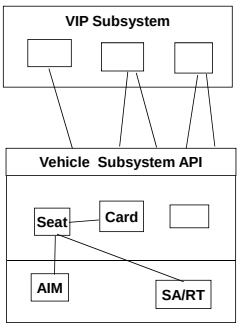
Subsystem Design with Façade, Adapter, Bridge

- 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!

34

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



35

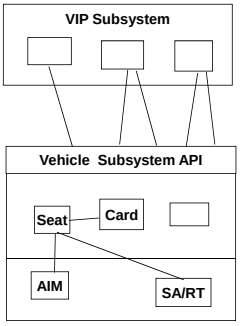
When should you use these Design Patterns?

- 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.

36

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



37

Summary of Design Patterns Discussed So Far

- Composite Pattern:
  - Models trees with dynamic width and dynamic depth
- Facade Pattern:
  - Interface to a subsystem
  - Distinguish between closed vs open architecture
- Adapter Pattern:
  - Interface to reality
- Bridge Pattern:
  - Interface to reality and prepare for future

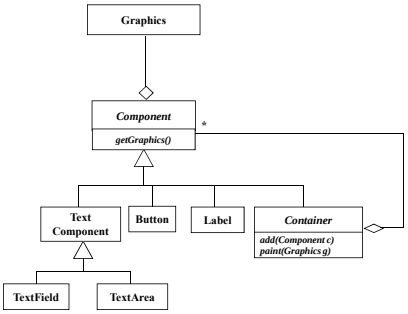
38

Additional Design Heuristics

- Never use implementation inheritance, always use interface inheritance
- A subclass should never hide operations implemented in a superclass
- If you are tempted to use implementation inheritance, use delegation instead

39

The Java's AWT library can be modeled with the component pattern



40

Notation used in the Design Patterns Book

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995
- Based on OMT (a precursor to UML). Notational differences between the OMT notation and UML:
  - Attributes come after the Operations
  - Associations are called acquaintances
  - Multiplicities are shown as solid circles ●
  - Dashed line: Instantiation Assocation (Class can instantiate objects of associated class) (In UML it denotes a dependency)
  - UML Note is called Dogear box (connected by dashed line to class operation): Pseudo-code implementation of operation.

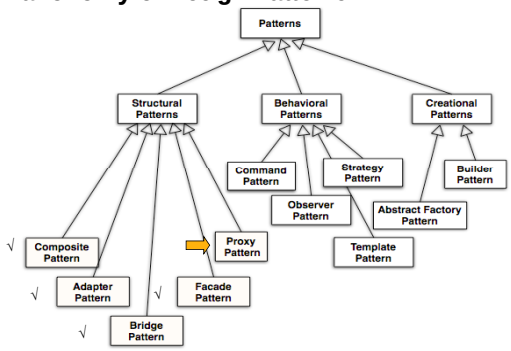
41

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).

42

A Taxonomy of Design Patterns

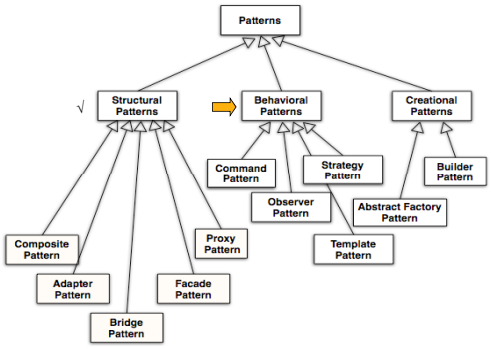


43

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.

44



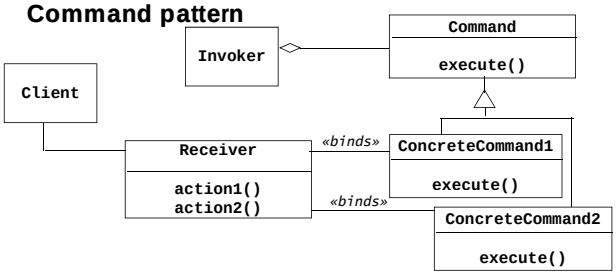
45

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.

46

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)

47

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.

48

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.

49

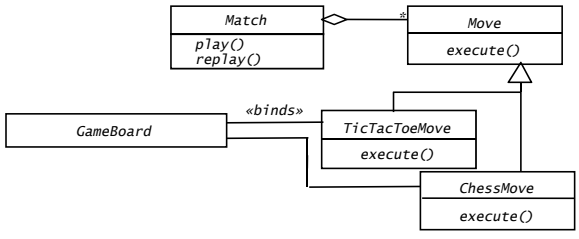
Command Pattern Applicability

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

- Uses:
  - Undo queues
  - Database transaction buffering

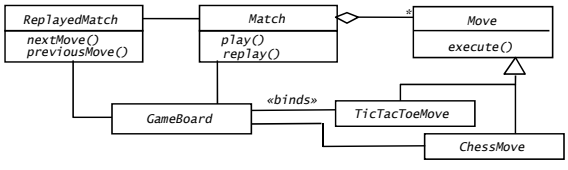
50

Applying the Command Pattern to Command Sets



51

Applying the Command design pattern to Replay Matches in ARENA

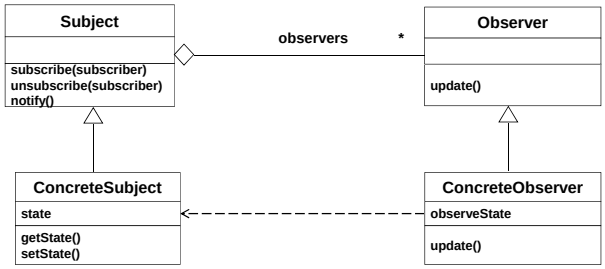


52

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



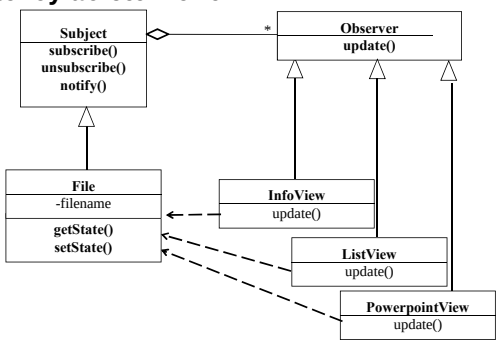
- The **Subject** ("Publisher") represents the entity object
- **Observers** ("Subscribers") attach to the Subject by calling **subscribe()**
- Each Observer has a different view of the state of the entity object
  - The **state** is contained in the subclass **ConcreteSubject**
  - The state can be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

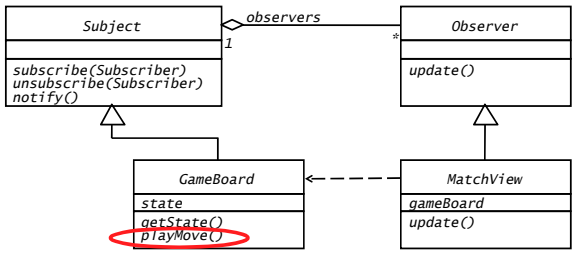
55

Applying the Observer Pattern to maintain Consistency across Views



56

Applying the Observer Design Pattern to maintain Consistency across MatchViews



Push, Pull or Push-Update Notification?

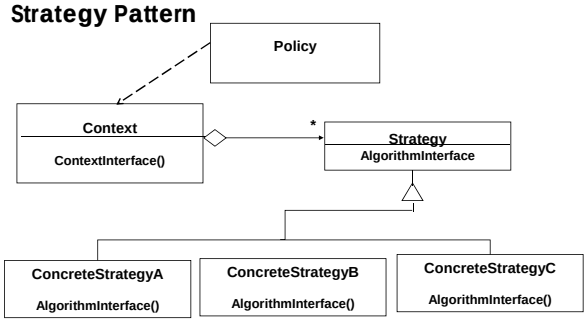
57

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.

58

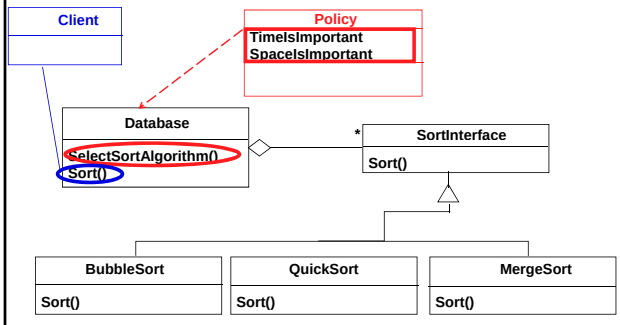
Strategy Pattern



Policy decides which ConcreteStrategy is best in the current Context.

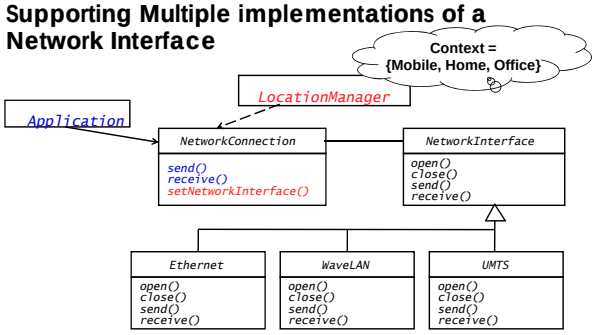
59

Using a Strategy Pattern to Decide between Algorithms at Runtime

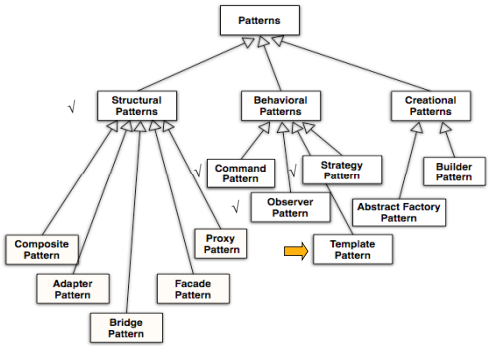


60

Supporting Multiple implementations of a Network Interface



61



62

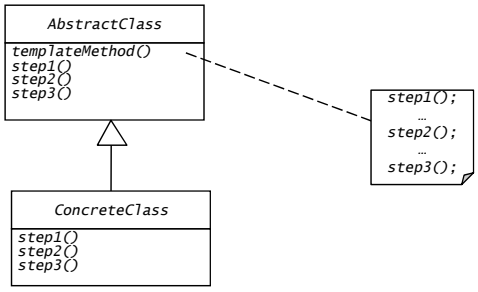
Template Method Motivation

- 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();
```

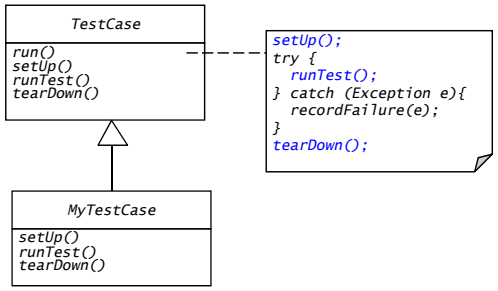
63

Template Method



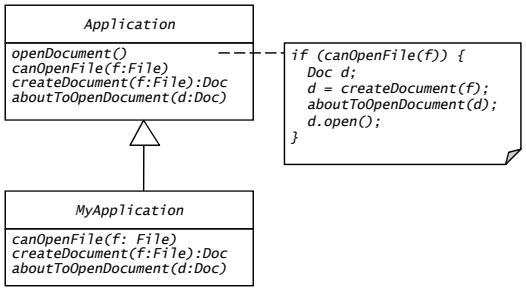
64

Template Method Example: Test Cases



65

Template Method Example: Opening Documents

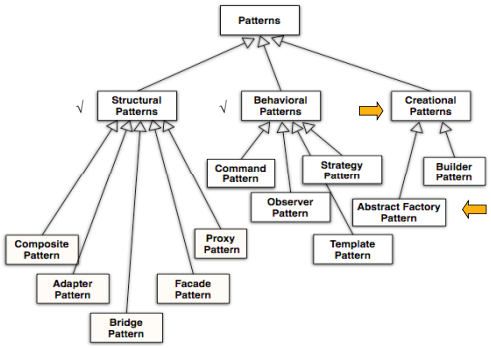


66

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”.

67



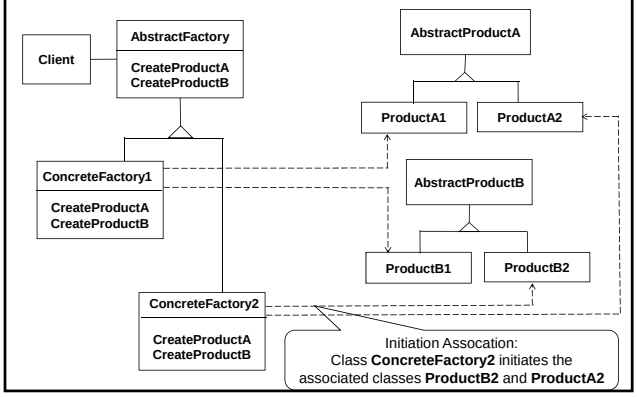
68

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?

69

Abstract Factory

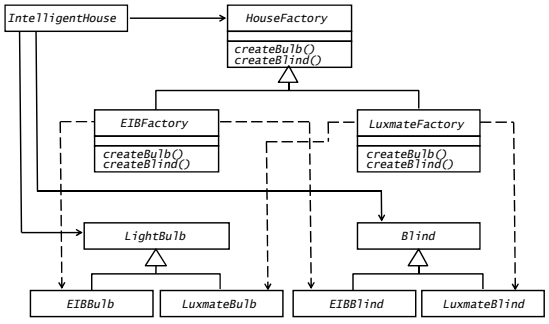


Applicability for Abstract Factory Pattern

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

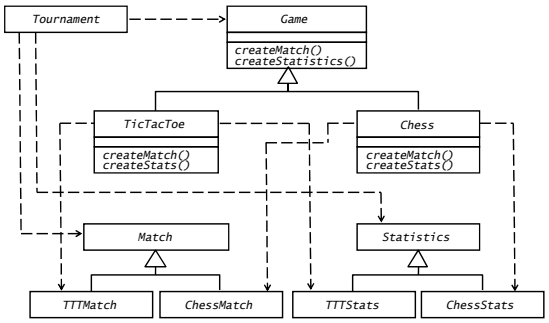
71

Example: A Facility Management System for a House



72

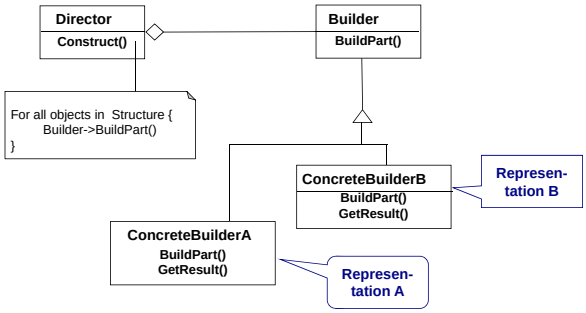
Applying the Abstract Factory Pattern to Games



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")

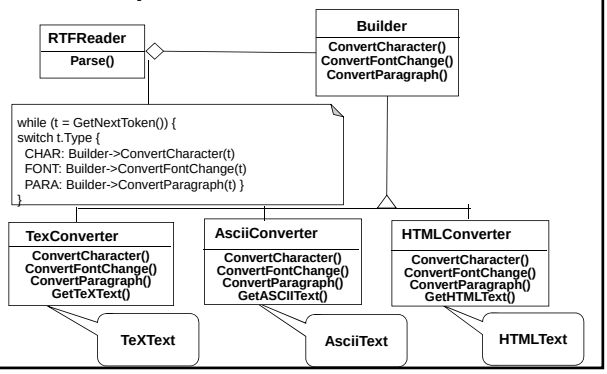
Builder Pattern



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

78

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

79

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

80

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

81

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

82