Lecture 4: Design Concepts For Responsibility-Driven Design

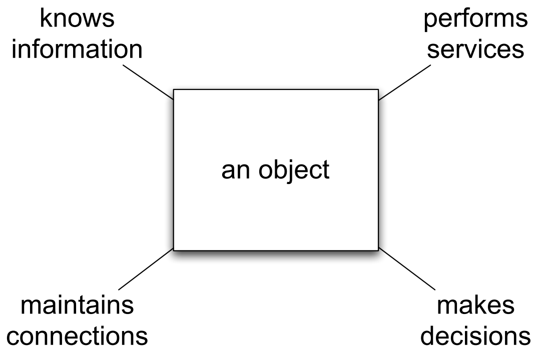
- Kenneth M. Anderson
- January 20, 2005

Introduction

- Chapter 1 of ***Object Design*** covers topics that aid understanding of Responsibility-Driven Design
  - Object Machinery
  - Roles
    - Role Stereotypes
    - Responsibilities and Collaborations
    - Object Contracts
      - Pre- and Post- Conditions
  - Domain Objects
  - Application-Specific Objects
  - Design Patterns
  - Frameworks
  - Architecture

  - Architectural Styles
  - Control Styles
  - Example: Layered Architecture

- Object Machinery
  - Software as a biological system
    - Like cells, software objects don't know what goes on inside one another (encapsulation) but they communicate (message passing) and work together to perform complex tasks (delegation and collaboration)
    - A software system's dynamic behavior emerges from the interactions of many objects
  - Object Responsibilities

- 

knows information

performs services

an object

maintains connections

makes decisions

- Common Object Design Terms
  - An ***application*** is a set of interacting objects
  - An ***object*** is an implementation of one or more roles
  - A ***role*** is a set of related responsibilities
  - A ***responsibility*** is an obligation to perform a task or know information
  - A ***collaboration*** is an interaction of objects or roles (or both)
  - A ***contract*** is an agreement outlining the terms of a collaboration

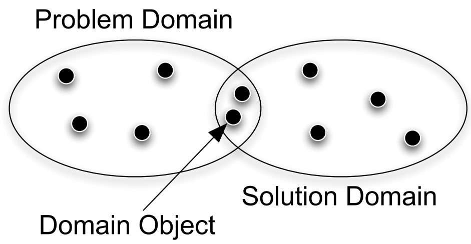
- Roles
  - Objects should have a specific purpose to play within a software system, i.e., a role
    - Roles are powerful because they allow objects that implement a role to be used interchangeably
      - Roles can be implemented using interfaces and composition
  - Role Stereotypes
    - Stereotypes are "purposeful oversimplifications" to give designers a target for defining roles
    - The following stereotypes have proven useful over time
      - Information Holder: knows and provides information
      - Structurer: maintains relationships between objects and information about those relationships
      - Service Provider: performs work
      - Coordinator: reacts to events by delegating tasks to others

    - Controller: makes decisions and closely directs the actions of other objects
    - Interfacer: transforms information and requests between distinct parts of a software system
  - Objects will often fit more than one stereotype, e.g., information holders will often provide services
    - A designers goal will be to decide what to emphasize and to strive to provide a clear cut role for each object
- Responsibilities and Collaborations
  - Responsibilities are assigned to roles
  - Roles are implemented by objects
    - If an object implements a role, it decides to accept the role's responsibilities
  - Objects work together to fulfill responsibilities
    - These object networks are called collaborations

  - A designer's task is to distribute responsibilities (roles) across a set of "intelligent" objects that can collaborate with each other such that all responsibilities are fulfilled
    - An "intelligent" object is one that has the right blend of information that it know about and services it can provide because of that information
      - you don't want to make any one object too powerful or too weak; the former tend to dominate designs in a bad way, reducing the use of encapsulation, inheritance, polymorphism, etc.; the latter tend not to provide much utility to the system overall
  - We will see specific examples of roles, responsibilities, and collaborations as we delve into responsibility-driven design over the next few weeks
- Object Contracts
  - Objects exist within an environment consisting of other objects
  - It is often helpful to specify during analysis and design the "contract" an object has with this environment

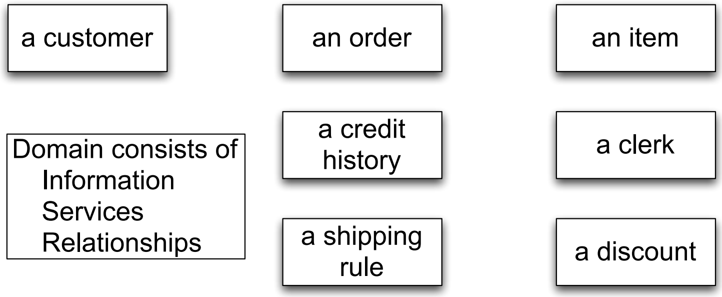
    - Here, a contract refers to specifying the conditions under which an object guarantees its work (pre-conditions) and the effects it leaves behind when its work is complete (post-conditions)
  - There are a number of ways this type of information can be documented; A particularly useful method is the use of "assert()" mechanisms in programming languages
    - At the start of each method, you place assert statements that specify the method's per-conditions; At the end of the method, you place assert statements that specify the post-conditions. At run-time, if an assert fails (evaluates to false), an exception is thrown

- Domain Objects
  - Designers and Users need a common vocabulary
    - This vocabulary allows designers to learn about the user's domain and to understand the requirements being provided by the user
    - This vocabulary often takes the form of a glossary but it can be much more useful if its incorporated into the object model of the system under design
      - This "vocabulary term as object" is considered a domain object
  - Problem Context
    - In the analysis and design of software systems, we face the following situation



- There will be plenty of objects in the problem domain
- There will be plenty of objects in the solution domain
- Our job is to find those objects from the problem domain that can be modeled in the solution domain that allow us to solve the problem posed to us by our users

Domain Objects
Inventory Control System



a customer

an order

an item

Domain consists of
Information
Services
Relationships

a credit history

a clerk

a shipping rule

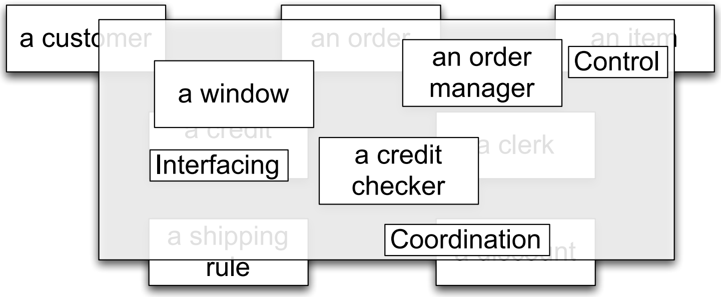
a discount

- There are many more objects possible in the domain of inventory control, but we may not need them!

Application-Specific Objects

- A software system also needs plenty of objects from the solution domain in order to function!
  - These objects are referred to as application-specific objects
    - They are typically added to a design AFTER the domain objects have been specified and are often driven by environmental and platform constraints (e.g. the choice of an application framework)
  - They consist of things like
    - user-interface elements
    - controller objects (more on that later)
    - collection classes
    - etc.

Application Objects
Inventory Control System



- The key to developing a successful software system is our ability as designers in merging domain objects into the solution domain, e.g. the world of the computer

Design Patterns

- Software designers will confront design problems as they develop a software system
  - Fortunately, many of these problems have been encountered (and solved!) before
- Design Patterns are descriptions of successful solutions to common design problems
  - They were made famous in the software development community by "the gang of four", the authors of the ***Design Patterns*** book that appeared in 1994
  - Good designers try to incorporate design patterns into their designs as much as possible; it allows them to focus on new problems or problems specific to their situation

- Design Patterns are often conveyed with a specific structure; our text book adopts the following
  - Name: Communicates the Intent of the Pattern
  - Problem: Describes a common design problem
  - Forces: Describes the tradeoffs you can make when applying this pattern
  - Context: Describes when the solution is appropriate
  - Solution: Describes how the problem can be solved
  - Consequences: Describes the impacts of using this solution in a software system
- Patterns provide the following benefits
  - Common Design Vocabulary: Raises the level of abstraction in analysis and design

- Expertise: Patterns are solutions that have worked before
  - Rule of Three: The design pattern book included only those patterns that had been deployed in at least three production systems
- Understanding: Developers can more readily understand the structure of a software system if it follows established design patterns

- Example: Double Dispatch
  - See code examples for two implementations of the "rock, paper, scissors" game; one that uses double dispatch and one that doesn't

Frameworks

- Frameworks are sets of objects designed to be extended and/or used to implement a common software service
  - Frameworks have been developed for
    - user interface services
    - persistence services
    - networking services
    - etc.
  - Typically, developers subclass framework objects to extend the framework for their particular situation
    - Alternatively, they provide objects which plug-in to the framework via composition by implementing a particular interface that governs when a framework will call the object and how; Think Adobe Photoshop plug-ins
- Frameworks offer a number of benefits:

  - Efficiency: frameworks reduce the amount of design and coding that you have to do
  - Richness: domain expertise is captured in the framework
  - Consistency: developers become use to the framework's structure; this allows them to develop new applications faster
  - Predictability: frameworks have been "stress tested" through frequent and wide use; they are hence more reliable and predictable than newly written software
- Frameworks come with disadvantages as well:
  - Complexity: frameworks often have steep learning curves
  - Blinders: developers may try to use a framework outside of its design constraints; "once you have a hammer, everything looks like a nail"

  - Performance: A framework can sometimes be slower than custom code because the framework's design goals may be flexibility and reusability and not performance
- Example Framework: Cocoa on MacOS X

- Architecture
  - A system's architecture consists of structure and behavior
    - Structure refers to the elements that appear in the software system and how they are arranged
      - typically these elements are "coarse" and refer to large subsystems;
      - but sometimes the elements are "fine grained" such as components, or even individual objects
      - it depends on the system
    - Behavior refers to the rules that govern how those elements interact
  - Architectural Styles
    - An architectural style is a predefined set of elements and behaviors; There are many types of architectural styles:
      - pipe and filter (Unix: "everything is a file")

      - message bus (pub/sub communication)
      - shared repository
      - layered abstract machines
    - Architectural styles are defined to provide benefits to the applications that follow them
      - layered architectures promote modularity and separation of concerns
      - pipe and filter promotes tool integration and consistency
      - message bus promotes loose coupling between system components
  - Control Styles
    - An important aspect of a software system's architecture is its control style (which we will examine later this semester)
    - Control style refers to how an application receives, processes, and responds to input events (from users and other tools)

  - A ***centralized control style*** involves the use of one object that processes and responds to all input events; such systems may use other objects but these tend to be just information holders with no logic of their own
  - A ***dispersed control style*** is located at the other end of the spectrum; logic for handling input events are spread across lots of objects; can lead to long sequences of method invocations to follow when debugging a single event
  - A ***delegated control style*** is a compromise between these two extremes; control is distributed across a number of object networks; each network handles one or a few input events, with the logic to process the event distributed evenly across each object in the network; dependencies between the various control centers are kept to an absolute minimum

- Example: Layered Architecture
  - Major responsibilities of a software system are distributed across a number of layers; layers on top are "closer" to the user (information presentation / event handling); layers on the bottom are "closer" to the machine (domain-related services / persistence / networking / ...)

  - Objects collaborate mostly within their layer
  - Messages across layers occur across well-defined connections with clients "above" servers. Thus,
    - requests flow down the layers
    - results flow up
  - Only the top and bottom layer are connected to the "outside" world

Summary

- So far, we have covered
  - fundamental OO concepts
    - objects and classes
    - encapsulation and abstraction
    - inheritance and composition
    - polymorphism
    - abstract classes and interfaces
    - object identity
  - Design concepts relevant to Responsibility-Driven Design
    - Roles
    - Domain and Application Specific Objects
    - Design Patterns, Frameworks
    - Application Architecture
- Next: Responsibility-Driven Design