LECTURE 7: CONTRACTS

Wirfs Brock et al., Designing Object-Oriented Software, Prentice Hall, 1990. (Chapter 6)

Outline

- Contacts: what and why?
- Contract documentation
  - Extension to CRC cards
  - More on collaboration graphs
- Guidelines for defining contracts
- Review: Module-View-Controller Architecture Pattern
Today’s Quote

- “I still haven’t graded your exams, and I’m so afraid to even look”
  
  *Dr. Salamah Salamah*

Review: Responsibility

- **Responsibility:** (what is this?)
Contract: What and Why

- **Responsibility**: something an object (server) does for other objects (clients)
- **Contract**: set of cohesive responsibilities that a client depends on
- Contract is not the same as responsibility
- Abstraction and analysis tool, e.g., for refining hierarchy and identifying subsystems

Client and Server View

- A **contract** defines a set of requests that a client can make of a server.
- The **server** is guaranteed to respond to the request.
- Responsibilities are the basis for determining contracts.
Example

- The class `Chart` has responsibilities for creating bar charts, pie charts, and line charts.

- The contract states that given a list of numbers and a chart type, the class will return a chart of the given type with the given numbers.

Classes and Contracts

- A class can support any number of contracts.

- A responsibility may be part of at most one contract.

- Not all responsibilities will be part of a contract
  - What do we call these?

- Contracts are used in a further formalize collaboration.
Collaboration Graphs

- Arrow from client to a contract supported by the server.
- One numbered semicircle for each contract.
- If two objects collaborate with a class by means of a contract, they point to the same semicircle.

Contract Documentation

- **Class**: <Class name>
- **Superclass**: <Parent class name>
- **Description**:
- **Contracts**:
  - <Contract #> <Contract Name>
  - <Contract Description>
  - <Responsibilities> <Collaborators>
- **Private Responsibilities**
  - <Responsibilities> <Collaborators>
Contract Documentation

11

Class: <Class name>
Superclass: <Parent class name>
Description:
Contracts:
   <Contract #> <Contract Name>
   <Contract Description>
   <Responsibilities> <Collaborators>
Private Responsibilities
   <Responsibilities> <Collaborators>

Number these sequentially for each class
Pay attention to inheritance;
Subclasses inherit from super classes

Example

12

Class: Form
Superclass: UserInteraction
Description: Provide human interface for data input
Contracts:
   1. Get Numeric Value from User
      Ask user for information DisplayScreen (7)
      Know if user responded Keypad
      Know user’s response
      Provide feedback to user
Private Responsibilities:
   Know form contents

Note the contract number for collaboration
Guidelines for Defining Contracts

- Ideas?
  - Group responsibilities used by the same clients
  - Maximize cohesiveness of classes
  - Minimize the number of contracts

Guideline 1

- Group responsibilities used by the same clients.
  - One way to find a contract, a set of cohesive responsibilities
  - When a set of responsibilities is used by the same client,
    - They may be detailed views of a single service
    - They may be part of one contract.
Example 1

- An Array class has two responsibilities likely to be used by the same client:
  - Return subset of elements that meet specified criterion
  - Return first element that meets specified criterion

- How many contracts?

- Contract: select elements based on specified criterion
Example 2

- An Employee class has two clients:
  - Client1: Record
    - needs general employee information
  - Client2: Payroll
    - needs salary-related information

- How many contracts?
  - Create two different contracts
    - Contract 1:
      - Provide general employee information
    - Contract 2:
      - Provide salary information

Guideline 2

Maximize cohesiveness of classes
- Contract is a cohesive set of responsibilities
- Class should be a cohesive set of contracts

- Make class hierarchy easier to refine by maximizing cohesiveness of contracts

- Look for server offering some abstract service to a variety of clients
  - Example: A Reader class can read from keyboard, file, network, …
Example

- Suppose that the class array supports a contract for element-wise arithmetic operations.
  - Q1: Appropriate behavior for arrays?
    - Any array containing elements for which multiplication is meaningless?
  - Q2: Solution?.

Remember?
Guideline 3

Minimize number of contracts

Why?

Guideline 3 (cont.)

- Provide a general interface so that clients do not have to know about implementations
- Look for similar responsibilities that can be generalized
- Use polymorphism
- Define contracts for classes at the top of hierarchy
- Add new contracts only for subclasses that add new functionality
Balance

- Conflicting Goals
  - Design small, easily understood and reusable classes
  - Design a small number of classes with easily understood relationships
  - Use your understanding of cohesion, coupling and inheritance to find the balance point.

Summary

- A contract is a set of cohesive responsibilities.
- A class should support a cohesive set of contracts.
  - Otherwise?
- Minimize number of contracts supported by each class.
What’s next

- Subsystem document.
  - Section 2 (Contracts)
  - Refine your CRC cards to group responsibilities into contracts.
  - Collaboration graph.
  - First Draft Due: Friday 03/02/208 by midnight.
  - Final Draft Due: Friday 03/09/208 by midnight.

OVERVIEW:
MVC ARCHITECTURE PATTERN
Recurring Problem

- Building User Interfaces
- How common is this?
- How often does it change?
- What do we do with something that changes a lot?
- Can you relate to this?

Motivation:

- Suppose we support both the command line and the GUI interfaces.
- What changes?
- What stays the same?
- This should sounds familiar
Motivation:

- Suppose we support both the command line and the GUI interfaces.
- What changes?
- What stays the same?
- How do you design to handle this?

Model-View Separation

- **Model**: The domain layer of objects.
  - (objects that contain data and operations).

- **View**: The presentation layer of objects
  - (windows, applets, reports).
Model-View Context

- **Context/Problem:** It is desirable to de-couple domain (model) objects from windows (views)
  - to support increased reuse of domain objects, and
  - minimize the impact of changes in the interface upon domain objects.

- **Solution:**
  - Define domain (model) classes that do not have direct coupling or visibility to the window (view) classes, and
  - Application data and functionality is maintained in domain classes, not window classes.

Model-View

**Goal:** Classes in Model should not have direct visibility to classes in View.

- **View**
  - Window
  - display()

- **Model**
  - Configuration
  - addEntry()
  - query()
Model-View Separation Motivation

- Focus more on the domain processes rather than on computer interfaces.
- Allow separate development of the model and user interface layers.
- Minimize the impact of changes in the interface upon the domain layer.
  - Which is one of the most common changes in software.
- Allow new views to be easily connected to an existing domain layer.

Problem

BAD

Model classes know about View classes.

View

Window
  display()

Configuration
  addEntry()
  query()

Model
displayMessage()
Problem

View

Window

display( )

query( )

-----

Configuration

addEntry( )

query( )

GOOD

View classes know about Model classes.

Model

Question

- Why is this better?
Question

- Why is this better???
  - Views change more often, this minimizes the impact of change.
  - We design for change.

Model-View Separation Pattern

The View Layer can be modified without requiring changes to the Model layer.

View

Old Window
- display()
- query()

New Window
- display()
- query()

Model

- Configuration
- addEntry()
- query()
Model-View Separation Pattern

**Problem:**
- Domain objects need to communicate with windows to cause a real-time ongoing display update as the state of information in the domain object changes.
  - Monitoring applications
  - Simulation applications.

**Solution:** Indirect Visibility
Model-View Separation Pattern with Indirect Visibility

- Named Publish-Subscribe *Architectural* Pattern
- Context/Problem:
  - A change in state (an event) occurs within a Publisher of the event
  - Other objects are dependent on or interested in this event (Subscribers to the event).
  - Publisher should not have direct knowledge of its subscribers.
- Solution:
  - Define an event notification system so that the Publisher can indirectly notify Subscribers.
  - Event Manager or Model View Controller (MVC).

### Diagram

```
Model
  addEntry() query()
  query()
```

```
View
  display()
```

```
Configuration
```

```
Window
  tells Window to Update
```

```
Model View Controller
  post Update Message
```

---

```
Configuration
```

```
Model View Controller
```

```
Window
  display()
```

```
View
  query()
```

```
Model
  addEntry()
```

---

```
Configuration
```

```
Model View Controller
```

```
Window
  display()
```

```
View
  query()
```

```
Model
  addEntry()
```

---

```
Configuration
```

```
Model View Controller
```

```
Window
  display()
```

```
View
  query()
```

```
Model
  addEntry()
```

---

```
Configuration
```

```
Model View Controller
```

```
Window
  display()
```

```
View
  query()
```

```
Model
  addEntry()
```
Why should students in 4311 care?

- Semester project deliverable:
  - GUI is due at midnight on Friday 02/09/2018
  - Client’s language of choice is Python
  - GUI should be built using the ideas of MVC
    - Or else you will run into lots of integration problems