Design and Model-View-Controller

CSC216

Design

• Goal: decide the structure of the software and the hardware configurations that support it.
• How individual classes and software components work together in the software system
  – Small programs: ~ 10 classes and interfaces
  – Medium programs: 1000s of classes and interfaces
• Design for single user applications different from web applications and mobile applications
• Software Artifacts: design documents, class diagrams, other UML diagrams
OO Design

- Discover classes (and their state)
  - What are the useful entities or concepts in your project requirements?
- Determine the responsibilities of each class
  - What should your class do? How does your class manipulate its data?
- Describe the relationships between the classes
  - Do classes have instances of other classes? Does a method of a class need to use an instance of another class?

Finding Candidate Classes and Methods

- Candidate Classes (and state)
  - Nouns
    - You will determine if the noun is an appropriate object or appropriate state
- Candidate Methods
  - Verbs
    - You will determine what class the method is most appropriate for.
- Use CRC cards to help identify candidate classes
University Registration System

- University XYZ has three types of students: undergraduates, masters, and PhDs. All students can register for courses using the Registration System.

UML

- UML: Unified Modeling Language
  - Models object-oriented software
  - Convergence from three earlier modeling languages
    - OMT (James Rumbaugh)
    - OOSE (Ivar Jacobson)
    - Booch (Grady Booch)

- Overseen by the Object Mentor Group (OMG): (www.omg.org)
Types of UML Diagrams

- Use Case Diagrams
- Class Diagrams
- Sequence Diagram
- State Chart Diagram

Use Case Diagrams

- How the system provides service(s) to the actors that will use it
Class Diagrams

- Classes and relationships between them

Sequence Diagram

- Sequence of actions (method calls) to complete a scenario
State Chart Diagrams

- How an object changes state when performing a task

Creating UML Diagrams

- Commercial or open-source tools
  - Microsoft Visio
  - Commercial Eclipse Plug-ins
  - Violet UML (open source)
  - Dia (freeware)
Class Diagrams in Depth

Classes

- **Student**
  - name: String
  - gpa: double
  - studentId: long
  - creditHoursCompleted: int
  - updateGPA(gradePoint:double,hours:int): void

Name

State

Behaviors
Classes

<table>
<thead>
<tr>
<th>Name</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- eventName: String</td>
</tr>
<tr>
<td></td>
<td>- meetingDays: String</td>
</tr>
<tr>
<td></td>
<td>- startTime: int</td>
</tr>
<tr>
<td></td>
<td>- endTime: int</td>
</tr>
<tr>
<td></td>
<td>+ Event(eventName: String, meetingDays: String, startTime: String, endTime: String)</td>
</tr>
<tr>
<td></td>
<td>+ isConflict(otherEvent: Event): Boolean</td>
</tr>
<tr>
<td></td>
<td>+ isConflict(otherCourse: Course): Boolean</td>
</tr>
</tbody>
</table>

Generalization (is-a)

- Generalization relationships show inheritance
- The child classes inherit the state and behavior of the parent class
  - An UndergraduateStudent is-a Student!
Composition (has-a)

- Composition is when one class has an instance of another class
  - Fields
  - Parameters
- Many ways to model in UML depending on type of composition relationship
- In assignments, requiring a composition relationship means there MUST be a has-a relationship in the diagram
  - It doesn’t have to use the composition UML connector of a solid diamond
  - It may be modeled using association (arrow) or aggregation (open diamond) connectors

Association Connectors

- Indicated by a solid arrow line from the source class to the target class
- Can be bi-directional represented by lines without arrow heads

Don’t usually put the association down as an attribute in the class
Properties: Attributes vs. Associations

• If attributes and associations are essentially the same thing, when should I use each one?
• Attributes
  – Small things (Strings, primitives, Dates, etc.)
  – Part of existing library
  – Immutable value objects
• Associations
  – Significant classes
  – Part of what will be implemented
  – Mutable references

Aggregation vs. Composition Connectors
Whole-Part Relationship

• Aggregation: stronger association (unidirectional)
  – Ex. A dept contains a set of employees
  – Ex. A faculty contains a set of teachers
  – A white diamond at the end of the association next to the aggregate class
  – The part can exist separate from the whole
• Composition: stronger aggregation (unidirectional)
  – Ex. Invoice– InvoiceLine
  – A black diamond on the end of association next to the composite class
  – The part cannot exist separate from the whole
Composition Relationship

```
Composition

Schedule

Event

Aggregation

Dependency

Association

Course

Schedule

CourseRecordIO

WolfSchedulerGUI
```

Relationships

```
Implement

Schedule

ScheduleItem

Course

Event

Interface

Abstract Method

Abstract Class

Inheritance

Dependency

CourseRecordIO

WolfSchedulerGUI
```

Evaluating a Class/Class Diagram

1. Intention-revealing naming: Does the name of the object convey its abstractions? Does the abstraction have a natural meaning and use in the domain?

2. Single Responsibility: Do the name, main responsibility statement data and functions align?

Sequence Diagrams

- How objects collaborate within a specified scenario
  - Behavior of the objects
  - Sequence of method calls
- An object’s lifetime is represented vertically
  - A rectangle (activation bar) represents when the object is active
    - Method is on the stack
Sequence Diagrams (2)

• Messages are passed between objects using a solid arrow
  – Messages may be passed within an object (self-call)
• Return messages point back to the calling object and are shown in dashed arrows
• Drawing sequence diagrams help developers understand how to implement a scenario.
Design Patterns

• “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”
  – Christopher Alexander

Design Patterns (2)

• Descriptions of communicating objects and classes
  – Participating classes
  – Roles and collaborations
  – Distribution of responsibilities
• Customized to solve a general design problem in a specific context
Why Design Patterns?

• Don’t need to start from scratch to solve a programming problem
  – Reuse previous **useful** ideas
• Design patterns are the closest thing we have to a Design Handbook for software engineering

• Design patterns are not a silver bullet!!!
  Sometimes the best designs do NOT require design patterns!!!

The Gang of Four

• THE book on Design Patterns
• 23 design patterns
• More in literature
• Book focuses on C++, but we have an online reference for Java

• Patterns are found, not created!
Pattern Families

- **Creational**: process of object creation
  - Singleton
- **Structural**: composition of classes or objects
- **Behavioral**: ways in which classes or objects interact and distribute responsibility
  - State/Strategy
  - Command

Model-View-Controller (MVC)

- **Isolates** business or domain logic from the input and presentation
- **Model**: data underlying the application
  - Changes in state require notifying the view
  - Model abstracts the data storage (DB, POJOs, etc.)
- **View**: UI
  - A model may have many views
- **Controller**: receives input and initiates response by calling model
MVC in Web Apps

• View: HTML or XHTML
• Controller: GET or POST input
  – Gives the GET or POST information to the objects
• Model: underlying domain logic

MVC in Java

• View / Controller: very tightly coupled
  – GUI Applications
    • View (how the object is presented) and controller (how events on the object are processed) belong to one UI object
    • Use generic components (i.e. JButton), which delegate to the underlying “look and feel” (i.e. the operating system)
  – Console Applications
    • View (console prompts) and controller (how user input is processed) typically belong to one (or more) UI objects.
• Model: underlying domain logic
Design Best Practices

• Classes should have **cohesion**
  – The extent to which the code for a class represents a single abstraction
  – Degree to which the members of a class are related to the general purpose of the class
  – Allows for reusability of the class in other programs

• Examples:
  – Student: only contains information relevant to a student
  – Courses: only contains information relevant to courses

Design Best Practices (2)

• A program should have low **coupling**
  – A connection between two classes is a dependency or coupling
    • Instance of an object in the class
    • Call another class to complete a task
  – Internal coupling: modifying another class’ data – avoid if possible!
  – Parameter coupling: using services provided by another class – unavoidable

• Highly coupled programs are difficult to write and maintain
Design Best Practices (3)

• Data and behavior should be **encapsulated** within a class or a package
  – Use packages to group together common functionality
    • Example: In an Android application, all Activities are part of the activity package
  – **Information hiding**: make data members private
  – Details about implementation are hidden within class and only exposed with public members

References

• Laurie Williams, UML Class Diagram Slides: http://agile.csc.ncsu.edu/SEMaterials/UMLClassDiagrams.pdf
• http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/