Adaptive Software Engineering
G22.3033-007

Session 8 – Sub-Topic 3 Presentation
Object-Oriented Design Guidelines

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Part I

Object-Oriented Design Guidelines
Object-oriented design

Goal: Want to express a design in terms of interacting objects
• What are classes and objects?
• How are classes used to formulate designs
• How is UML used in this process?

Object-oriented programming principles

• Encapsulation
• Inheritance
• Abstract data types
• Polymorphism
Two object oriented design processes

- Both: build the design first in UML diagrams
- Top-down (Sommerville) Start at architecture-component level, refine down to objects
- Use-case driven (Unified Process) Start with real-world entities that realize the use case, refine these to software objects

Good object-oriented design

Goals:
- Code should be easy to understand and easy to maintain
- A class has a simple, easily-stated purpose
- Class design should facilitate reuse
- Minimize interdependencies between different parts of the program

Two important metrics: coupling should be low and cohesion should be high
UML diagrams in the design process

• Use cases (not use-case diagrams!) drive the process
• Class diagrams: Identify the classes and how they are related
• Interaction diagrams (sequence and collaboration): Show how different objects interact (dynamically) to do the tasks needed.

Other UML diagrams

• State and activity diagrams are helpful in understanding the changing state of the system and its subsystems at various points during execution
• UML component, package, collaboration, deployment help us understand how large systems are assembled from constituent software and hardware pieces
Object-oriented analysis and design

See short summary on class website at:
http://www.nyu.edu/classes/jcf/g22.3033-007/handouts/g22_3033_007_h52.htm

Overview of object-oriented analysis and design

An iterative process (not necessarily in the following sequence):
• Develop requirements (in use-cases)
• Identify classes from real-world entities participating in use cases
• Draw interaction diagrams for the use cases, identifying responsibilities to be performed
• Draw class diagrams identifying the functions that will be performed in each class
• Use other diagrams to help understand and document the design
Sample Requirement statement

When an appointment is to be made which involves a number of people, the system finds a common slot in each of their diaries and arranges the appointment for that time. If no common slots are available, it interacts with the users to rearrange their personal diaries to make room for the appointment.

Use case: Schedule a meeting

- Precondition: A user has connected to the system, with info about a meeting to be scheduled
- Actors:
  - Initial user (who wants to schedule a meeting)
  - Workers (other users) who are invited to attend
  - E-mail system (to communicate with users)
Primary success scenario

1. User specifies proposed meeting (attendees, allowable days & times, duration, name & subject of meeting)
2. System finds earliest time slot that is available in all attendees’ diaries.
3. System inserts appointment in each diary
4. System sends e-mail to all attendees notifying them that they are scheduled to attend

Alternate scenario

If in step 2., no time slot is found:
3. System finds “best” time slot, i.e., least number of attendee conflicts
4. System sends e-mail to all users who have a conflict with that time, asking them to reschedule their other meeting (if possible).
5. After a reasonable wait for responses, if conflict remains, repeat from step 3 with next “best” slot etc.
6. Schedule meeting when time conflicts are resolved.
Initial conceptual classes
(real-world entities)

Drawn from application domain, or interfaces to secondary actors
• Worker
• Worker’s diary
• Meeting
• Appointment
• E-mail system interface

Other (software) classes

Computer-world entities that one reasonably can expect to need (others may still be added later)
• UI (needed to interact with the users)
• Scheduler (needed to perform the logic of the use case(s))
Interaction diagram: show all the interactions between classes

- Choose a level of detail appropriate for your level of understanding of the design
- More detail is added as the design becomes finalized
- Many choices, no one “best” answer
Design classes to perform the functions needed in use case

Can start with use cases, look for attributes, operations, and associations

- UI: provide panels to user
- Scheduler
  - Association to meeting
  - scheduleMeeting(meeting properties)
- Meeting
  - Attributes: meeting properties
  - Associated with worker
  - Create (meeting properties)
  - getNextTimeSlot(1..number of slots)
  - getAttendee(1..number of attendees)

More classes

- Worker
  - Attribute: diary
  - isAvailable(timeslot)
  - bookMeeting (meeting attributes)
- Diary
  - Associated with 1 worker, any number of appointments
  - isTimeSlotOpen(timeslot)
  - scheduleAppt(meeting attributes)
- Appointment
  - operations create(), add(attributes)
  - Attributes: time, date, name, subject, etc.
Construct Class Diagram

May need to add more information based on new awareness of functionality
Also will need to augment class diagram to consider all other use cases and alternate scenarios
Improve the object-oriented design

• Look at coupling and cohesion:
  – Do classes need to know too much about the internals of other classes?
  – Are the operations and attributes of a class all closely related?

• Is the design easily maintainable?
  – What changes are likely to happen as the product evolves?

Likely changes in product

• Different kinds of meetings: recurring, call-in, video
• Different worker types/preferences: onvacation, offsite, unschedulable
• Worker teams
• Different scheduling methods:
  – Invitees opt-in, or opt-out
  – Just insert into everyone’s diary
  – Mandatory
• Different e-mail systems (some may have better interface to diary)
Object-oriented design for expected changes

- Inheritance for different kinds of similar classes (e.g. meetings)
- Polymorphism for a method that acts differently in different situations (different scheduling algorithms)
- Different implementations can be hidden by high-level virtual methods (e-mail systems)
- Child classes can be aware of other child-classes (worker teams)
Iterate

• Keep adding detail until you are convinced you can perform all the use cases with the UML diagrams.

Convert to code

• Class definitions are derived directly from class diagrams
  – If you are doing your diagrams in Rational Rose, the tool will generate your class definitions (header files) and initial code for your methods
• Your method implementations come from your understanding of the functionality associated with each step of the use case.