UML and OOD

- What are good systems like?
- Most real systems are too big for a single developer to understand everything about the system ⇒ modules.
- How do we break a system into modules/components/classes?

These lectures notes are based, often literally without explicit quotation, on “Using UML: Software Engineering with Objects and Components” by P. Stevens and R. Pooley.

Some terms related to OOD

- Dependency
- Coupling
- Cohesion
- Interface
- Encapsulation
- Abstraction

Dependency

- Class A depends on class B if a change in B can necessitate a change in A.
- A is said to be a client of B.
- A circular dependence is bad because it hampers reuse. You can’t reuse one class in the circle without using all of them.

Coupling

- A system with many dependencies has high coupling (that’s bad).
- Low coupling implies changes in one part affect fewer other parts.
- How can you identify dependencies?

Interfaces

- An interface defines services exported from a class or module.
- The interface identifies the methods the clients depend on.
- Java checks syntactic dependencies (type checking and method existence).
- Java does not support semantic checking or enforcement of dependencies.
- For example, you cannot specify formally that a sort() method actually sorts.

Multiple Interfaces

- Implementing multiple interfaces allows for more accurate representation of dependencies.
- For example, by using the Painter interface, I documented that PenAdjuster depends on PaintListener3 have a setPenSize() method.
- Without this interface, it was not superficially obvious that PenAdjuster didn’t depend on other aspects of PaintListener3.
Cohesion

- A class/module with an intuitively simple interface that is in fact complex to implement has high cohesion.
- Its parts “belong” together.
- The interface provides an abstraction of the internally complex object.

Abstraction vs Encapsulation

- Abstraction: the client only needs to know the interface.
- Encapsulation: the client cannot see more than the interface.
- High cohesion and low coupling ⇒ easy to replace, easy to reuse, easy to understand/maintain/modify ⇒ Good!

UML: A Case Study

- You have been asked to build an audience response system for use in classrooms.
- We will use UML to work through part of the design.
- First we need to refine the requirements.

Requirement: Instructor Display

The server will include a display that shows a continuously updated histogram of student responses to “How is the lecture pace?” on a 5 point scale.
There should be a display that shows a histogram of responses to instructor stated questions. This histogram will also have 5 choices (in this case A through E). The question response histogram should only be updated in response to some action by the instructor such as clicking on a button.
There should be a display that shows text questions submitted by the students. Along with each question should be a number indicating how many students (including the author of the question) would like the question to be answered during class. Also there needs to be a mechanism for the instructor to delete questions.

Requirement: Server to Client Communication

The server is responsible for receiving the various data from the clients (i.e. pace, question responses, question submissions, and indications of support for a question) and broadcasting the question list including deletions and support count to the clients.

Requirement: Client Display

The client (student) system will include a GUI that includes four main features.
There should be a way to indicate how the pace of the class is going on a 5 point scale.
There should be a way to indicate a response to a multiple-choice question on a 5 point scale.
There should be a way to enter the text of a question and submit it to the instructor.
There should be a display list of current questions with a mechanism for indicating interest in having the question answered during class.
Use Case Model

- **Users** – actors in UML (usually people), e.g. Student, Instructor
- **Tasks** – use cases, e.g. Submit a question
- A use case is a task which a user must perform with the help of the system – e.g. submit a question.
- Write a detailed description, in third-person, active-voice, for each “use case.”

A Use Case

**Submit question** – A Student enters the text of a question and clicks the submit button. The client sends the question to the server. The server assigns a unique question id to the question and then broadcasts the question to all clients. The server also adds the question to a displayed list of questions. The clients when they receive the question add the question to a displayed list of questions.

Use Case Diagram

- **Student**
  - Submit a question
  - Support an existing question
  
- **Instructor**
  - Delete a question
  - Clear the response histogram

Use Case Descriptions

- You must write use case descriptions for each use case shown in the use case diagram.
- The goal is to document what the system “should do” not “how” it should do it.
- Do not invent requirements while writing these descriptions.

Identifying Classes: The noun identification technique

The server will include a display that shows a continuously updated histogram of student responses to “How is the lecture pace?” on a 5 point scale.

There should be a display that shows a histogram of responses to instructor stated questions. This histogram will also have 5 choices (in this case A through E). The question response histogram should only be updated in response to some action by the instructor such as clicking on a button.

There should be a display that shows a list of questions submitted by the students. Along with each question should be a number indicating how many students (including the author of the question) would like the question to be answered during class. Also there needs to be a mechanism for the instructor to delete questions.

The Initial Set of Classes

- After eliminating the “obviously” poor candidates we are left with the following possible classes:
  - Server
  - Display
  - Histogram
  - Histogram of Pace
  - Question Response Histogram
  - Question
  - Choice
  - List of Questions
CRC Cards

- At this point you could create CRC cards for the classes and begin to identify some responsibilities.
- We can also begin to identify some collaborations.
- In UML collaborations are called associations.

Class Diagrams and Associations

- Class diagrams are the UML equivalent of CRC cards.
- Associations describe how objects work together – the “collaboration” in CRC.
- Associations allow us to check coupling.

A Class Diagram

Class Diagrams

- Note the multiplicity notations.
- Note the lack of direction (navigability) on associations.
- We are not concerned at this point about which class depends on which.

Collaboration Diagrams

- Class diagrams show a static view of the system.
- Collaboration diagrams show a more dynamic view of the system in action.
- A collaboration diagram corresponds to a particular use case.
- Not all use cases will require collaboration diagrams.
Collaboration Diagram

- Note the numbering scheme for the messages.
- Messages imply associations and navigability in the class diagram.

Case Study 2: A Drawing Program

Write a program that can be used to create freehand drawing using the mouse. The user must be able to change the width of the pen used in the drawings. Allow the user to change the pen size using a menu. Drawing will occur while the mouse button is held down.

Use Case Diagram

A Class Diagram
Drawing with the Pen

theArtist: Artist
   ↓ dragMouse
theMouse: MouseButton
   ↓ 1:dragMouse
thePen: Pen

theDrawing: Drawing
   1.1:drawOval

Setting the Pen Size

theArtist: Artist
   ↓ select pen size
theMenu: Menu
   ↓ 1:set pen size
thePen: Pen

Drawing with the Pen: Revised

theArtist: Artist
   ↓ select pen size
   1:select pen size
thePen: Pen
   ↓ 1.1:set pen size
small: JMenuItem
   ↓ 1:selected
listener: PenAdjuster

Setting the Pen Size: Revised

theArtist: Artist
   ↓ select pen size
theMenu: Menu
   ↓ 1:set pen size
small: JMenuItem
   ↓ 1:selected
listener: PenAdjuster

Drawing with the Pen: Revised

theArtist: Artist
   ↓ dragMouse in
theDrawing: Drawing
   1:dragMouse
   ↓ 1.1:drawOval
thePen: Pen

A Class Diagram: Revised

PopupMenu
   connectivity
ActionListener
   adjusts
PenAdjuster
   adjusts
AdjustablePen
   draws on
Pen
   1.1:getImage
mouse
   1.1:getGraphics
MouseMotionListener
   moves
Drawing
   move
Image
   1.1:drawOval
Graphics

Coupling and The Law of Demeter
Law of Demeter
Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
In response to a message \( m \), an object \( O \) should send messages only to the following objects:
1. \( O \) itself
2. objects which are sent as arguments to the message \( m \)
3. objects which \( O \) creates as part of its reaction to \( m \)
4. objects which are \textit{directly} accessible from \( O \), that is, using values of attributes of \( O \).

### Coupling and The Law of Demeter

[Diagram of coupling and the Law of Demeter]

```
// Pen.java - paints on a Drawing
import java.awt.*;
import java.awt.event.*;
public class Pen implements MouseMotionListener,
AdjustablePen {
    public void mouseDragged(MouseEvent e) {
        Drawing canvas = (Drawing)e.getSource();
        canvas.fillOval(e.getX() - radius,
        e.getY() - radius,
        diameter, diameter);
    }
    public void mouseMoved(MouseEvent e) {};
}
```

```
public void setPenSize(String size) {
    if (size.equals("Small")) {
        radius = 0;
        diameter = 1;
    } else if (size.equals("Medium")) {
        radius = 3;
        diameter = radius * 2;
    } else if (size.equals("Large")) {
        radius = 6;
        diameter = radius * 2;
    }
}
protected int radius = 3;
protected int diameter = radius * 2;
```

```
// PenAdjuster.java - pass pen adjustment requests
// to the AdjustablePen
import java.awt.event.*;

class PenAdjuster implements ActionListener {
    private AdjustablePen pen;
    PenAdjuster(AdjustablePen thePen) {
        pen = thePen;
    }
    public void actionPerformed(ActionEvent e) {
        pen.setPenSize(e.getActionCommand());
    }
}
```

```
// Drawing.java - remember drawing operations,
// using an offscreen image, only supports fillOval()
import java.awt.*;
import javax.swing.*;

class Drawing extends JComponent {
    private static final int SIZE = 500;
    private Image offscreenImage;
    private Graphics offscreenGraphics;
    public void paint(Graphics g) {
        if (offscreenImage != null)
            g.drawImage(offscreenImage,0,0,SIZE,SIZE,null);
    }
}```
// draw on an offscreen image, displayed by paint()
public void fillOval(int left, int top,
        int width, int height)
{
    if (offscreenImage == null) {
        offscreenImage = createImage(SIZE, SIZE);
        offscreenGraphics = offscreenImage.getGraphics();
    }
    offscreenGraphics.fillOval(left, top, width, height);
    repaint();
}

public Dimension getMinimumSize() {
    return new Dimension(SIZE, SIZE);
}

public Dimension getPreferredSize() {
    return new Dimension(SIZE, SIZE);
}