CSC 309 Lecture Notes Weeks 6 and 7

Design Refinement

Introduction to Code Coverage Measures

I. **Administrative Matters**

A. Midterm moved to ***Monday 18 May.***

B. M4 due date also moved to ***Monday 18 May.***

C. Here is ***Week 7*** design review schedule:

Week 7 Design Reviews

Day	Time	Team
Wed 13 May	2:35-3:00	DJ Cars
	3:10-3:35	Fire Breathers
	3:35-4:00	Node
Fri 15 May	2:35-3:00	Team 0
	3:10-3:35	Team 1
	3:35-4:00	Token CSC

D. Here meeting schedule for ***Friday Week 6***:

Time	**Team**
2:10 - 2:28	DJ Cars
2:28 - 2:46	Token CSC
2:46 - 3:04	Team #1
3:04 - 3:22	Rubber Duckies
3:22 - 3:40	Node
3:40 - 3:58	Team 0

II. The "P" part of "MVP"

A. "P" is for "Process".

B. Define data and methods to support efficient imple'n of Model classes.

C. Process classes do not trace directly to user-level spec.

"P" is for "Process", cont'd

D. A lot of process classes come from a library, Java or other language.

   1. In particular, Java ***collection*** classes.

   2. Highly suitable for 309 use.

III. Example data structure refinement

A. See Calendar Tool example online.

B. Key refinement is choosing appropriate data rep for a user calendar.

C. Design based on need to access items by unique key, and in ordered sequences.

D. `TreeMap` is chosen.

E. See in particular `UserCalendar` and `ItemKey` classes.

IV. Using `java.io.File`

A. Key classes and methods:

1. `FileInputStream`, `FileOutputStream`
2. `ObjectInputStream`, `ObjectOutputStream`
3. `readObject`, `writeObject`

Using java.io.File, cont'd

B. Suppose we have

```
public class SomeModelClass
    extends Model {

    ...
}
```

C. To write out:

```
model = new SomeModelClass();
/* Put some data in model ... */
```

C. To write out:

```
model = new SomeModelClass();

/* Put some data in model ... */

FileOutputStream outFile =
    new FileOutputStream(
        "model.dat");
```

C. To write out:

```
model = new SomeModelClass();

/* Put some data in model ... */

FileOutputStream outFile =
    new FileOutputStream(
        "model.dat");

ObjectOutputStream outStream =
    new ObjectOutputStream(outFile);
```

C. To write out:

```
model = new SomeModelClass();

/* Put some data in model ... */

FileOutputStream outFile =
    new FileOutputStream(
        "model.dat");

ObjectOutputStream outStream =
    new ObjectOutputStream(outFile);

outStream.writeObject(model);
```

D. To read back in:

```
FileInputStream inFile =
    new FileInputStream("model.dat");

ObjectInputStream inStream =
    new ObjectInputStream(inFile);

model = (SomeModelClass)
    inStream.readObject();
```

Using java.io.File, cont'd

E. More details on pages 1-2 of notes.

F. For team members who implement file opening and saving.

V. Observer/Observable pattern.

A. Useful when multiple views change, based on changing model, e.g.,

V. Observer/Observable pattern.

A. Useful when multiple views change, based on changing model, e.g.,

1. ***CalTool:*** daily, weekly, monthly views

V. Observer/Observable pattern.

A. Useful when multiple views change, based on changing model, e.g.,

1. ***CalTool:*** daily, weekly, monthly views
2. ***Grader:*** gradebook, graphics views

V. Observer/Observable pattern.

A. Useful when multiple views change, based on changing model, e.g.,

1. ***CalTool:*** daily, weekly, monthly views
2. ***Grader:*** gradebook, graphics views
3. ***Scheduler:*** list and calendar views

V. Observer/Observable pattern.

A. Useful when multiple views change, based on changing model, e.g.,

1. ***CalTool:*** daily, weekly, monthly views
2. ***Grader:*** gradebook, graphics views
3. ***Scheduler:*** list and calendar views
4. ***TestTool:*** ques dialogs, ques bank views

B. Java's *Observer* interface.

```
interface Observer {

  public void update(
    Observable o,
    Object arg)

}
```

C. Java's *Observable* class.

```
class Observable {
  void addObserver(Observer o)
  void setChanged()
  boolean hasChanged()
  void notifyObservers()
  void notifyObservers(Object arg)
}
```

D. Typical usage

```
class model extends Observable { ... }
class View implements Observer { ... }

class UserCalendar extends Model {

     . . .

    public void add(ScheduledItem item) {

        . . .

        items.add(item);
        setChanged();
    }
}
```

```
public class OKScheduleEVentButtonListener
        implements ActionListener {

    public void actionPerformed() {

        . . .

        userCalendar.add(...);
        userCalendar.notifyObservers();
    }
}
```

```
public class MonthlyAgenda extends View {

    public MonthlyAgenda(
        UserCalendar userCalendar) {

        . . .

        userCalendar.addObserver(this)
    }

    public void update(Observable o,
            Object arg) {

        /* Get items from model ... */

    }
}
```

VI. Client/server pattern

A. Details in on page 4 of notes and 309/examples/rmi.

B. For team members who implement server-to-client processing using RMI.

VII. Coupling and cohesion -- a Couple Well-Established Terms

A. "Coupling" and "cohesion" denote connectedness and interconnectedness.

Coupling and Cohesion, cont'd

B. Measures of coupling:

Coupling and Cohesion, cont'd

B. Measures of coupling:

   1. number of classes inherited from

Coupling and Cohesion, cont'd

B. Measures of coupling:

   1. number of classes inherited from
   2. number of classes referenced

Coupling and Cohesion, cont'd

B. Measures of coupling:

1. number of classes inherited from
2. number of classes referenced
3. number of methods called

Coupling and Cohesion, cont'd

B. Measures of coupling:

1. number of classes inherited from
2. number of classes referenced
3. number of methods called
4. number of parameters in a method

Coupling and Cohesion, cont'd

C. Measures of cohesion:

Coupling and Cohesion, cont'd

C. Measures of cohesion:

1. class has logically-related functionality

Coupling and Cohesion, cont'd

C. Measures of cohesion:

1. class has logically-related functionality
2. method performs single specific function

Coupling and Cohesion, cont'd

C. Measures of cohesion:

1. class has logically-related functionality
2. method performs single specific function
3. coupling is reduced overall

Coupling and Cohesion, cont'd

D. Conclusion --

minimize coupling, maximize cohesion

VIII. Techniques to reduce coupling

VIII. Techniques to reduce coupling

A. Limit number of classes that communicate with each other.

VIII. **Techniques to reduce coupling**

A. Limit number of classes that communicate with each other.

B. Limit number of calls between classes that do communicate.

Reduce coupling, cont'd

1. E.g., button listeners talk directly to model:

```
OKScheduleEventListener.
                actionPerformed
```

calls

```
schedule.scheduleEvent
```

directly.

Reduce coupling, cont'd

2. More highly-coupled alternative, mediator / controller style:

```
SomeMediator.
  getSchedule().
    scheduleEvent(...)
```

Reduce coupling, cont'd

C. Limit public methods in each class.

Reduce coupling, cont'd

C. Limit public methods in each class.

   1. Make method public only if demanded.

Reduce coupling, cont'd

C. Limit public methods in each class.

1. Make method public only if demanded.

2. Don't add `get`, `set` methods "in case" they may be needed.

Reduce coupling, cont'd

C. Limit public methods in each class.

1. Make method public only if demanded.
2. Don't add `get`, `set` methods "in case" they may be needed.
3. E.g., ScheduleEventDialog.getSchedule is never used, and not provided.

Reduce coupling, cont'd

D. Limit method parameters, return vals to smallest type necessary.

Reduce coupling, cont'd

D. Limit method parameters, return vals to smallest type necessary.

   1. Consider `UserCalendar.getItems`.

Reduce coupling, cont'd

D. Limit method parameters, return vals to smallest type necessary.

   1. Consider `UserCalendar.getItems`.

   2. It could return full `UserCalendar` or simpler `ScheduledItem[]`.

Reduce coupling, cont'd

D. Limit method parameters, return vals to smallest type necessary.

   1. Consider `UserCalendar.getItems`.

   2. It could return full `UserCalendar` or simpler `ScheduledItem[]`.

   3. `ScheduledItem[]` is preferable, since callers of `getItems` don't need full `UserCalendar`.

Reduce coupling, cont'd

E. Design to limit change needed if data representations change.

Reduce coupling, cont'd

E. Design to limit change needed if data representations change.

1. When using external process classes (e.g., DB package), design *wrapper* classes.

Reduce coupling, cont'd

E. Design to limit change needed if data representations change.

1. When using external process classes (e.g., DB package), design *wrapper* classes.

2. With properly designed wrappers, little or no change to Model is necessary when changing Process classes.

Reduce coupling, cont'd

F. Use exception handling wisely.

Reduce coupling, cont'd

F. Use exception handling wisely.

1. Consider use of exception handling to communicate between models and views.

Reduce coupling, cont'd

F. Use exception handling wisely.

1. Consider use of exception handling to communicate between models and views.

2. The `PrecondViolation` exception classes provide generic, uniform error communication.

IX. Techniques to increase cohesion

IX. Techniques to increase cohesion

A. Coupling easier to pinpoint.

IX. Techniques to increase cohesion

A. Coupling easier to pinpoint.

B. Less coupling generally increases cohesion.

IX. **Techniques to increase cohesion**

A. Coupling easier to pinpoint.

B. Less coupling generally increases cohesion.

C. Addressing cohesion directly means each class does ***"one thing"***.

Increase cohesion, cont'd

D. Many design patterns promote cohesion.

Increase cohesion, cont'd

D. Many design patterns promote cohesion.

E. Simply limiting size of methods and classes increases cohesion.

Increase cohesion, cont'd

D. Many design patterns promote cohesion.

E. Simply limiting size of methods and classes increases cohesion.

F. I.e., it's harder to do too much in small classes and functions.

NOTE:

Handout on design and implementation conventions says:

Increase cohesion, cont'd

- No method longer than 50 lines
- No more that 25 public, 25 protected methods per class (50 total)
- No more than 50 data fields

X. Ease of debugging -- a major rationale for limiting coupling

X. Ease of debugging -- a major rationale for limiting coupling

A. "If something goes wrong in module X,

where do I look?"

X. Ease of debugging -- a major rationale for limiting coupling

A. "If something goes wrong in module X,

where do I look?"

B. First, look in module X.

X. Ease of debugging -- a major rationale for limiting coupling

A. "If something goes wrong in module X, ***where do I look?"***

B. First, look in module X.

C. Then look at other modules coupled to X.

X. Ease of debugging -- a major rationale for limiting coupling

A. "If something goes wrong in module X, ***where do I look?"***

B. First, look in module X.

C. Then look at other modules coupled to X.

D. Repeat until problem located.

X. Ease of debugging -- a major rationale for limiting coupling

A. "If something goes wrong in module X, ***where do I look?"***

B. First, look in module X.

C. Then look at other modules coupled to X.

D. Repeat until problem located.

E. This is much easier with less coupling.

XI. Milestone 4 Testing Details

A. We will ***NOT*** use Spest to generate tests.

B. Do use Spest for specs,
to aid hand-written testing:

1. preconds define testing input ranges

2. postconds define testing oracles

Milestone 4 Testing Details, Cont'd

C. Don't forget HOW-TO-RUN-TESTS.html.

D. Describe in detail the following:

1. Exactly what's tested per team member.
2. How to execute tests.
3. Where results appear.

XII. **What is code coverage?**

A. What's covered during program execution.

B. Typically measured at lines of code.

C. Coverage measure is percentage of *program lines run.*

D. All lines covered => 100%.

XIII. **How code goes "uncovered".**

A. Reasons include:

1. *Uninvoked functions*
2. *Untaken conditional branches*
3. *Unexecuted loop bodies*

How code goes uncovered, cont'd

B. During testing, uncovered code means ***there are insufficient test cases.***

XIV. Coverage Tool Resources

A. See the `309/doc/` page.

B. Note that full code coverage is NOT required for Milestone 4, but is for final project.

C. M4 requires ***selection of which coverage tool to use*** + initial application.

XV. **Where code coverage fits into testing.**

A. Ensure black box tests are adequate.

B. Different levels of coverage exist.

C. Good tests must ensure
some measure of coverage.

Where code coverage fits into testing, cont'd

D. Coverage measures made during testing

E. Following discussion is of different coverage measures, from weakest to strongest.

XVI. Code coverage measures.

A. Function (method) coverage.

B. Statement coverage

C. Branch coverage

D. Decision coverage

Code coverage measures, cont'd

E. Loop coverage

F. Define-use (d-u) coverage

G. All path coverage

H. Exhaustive coverage