Lecture 6

OCL

Use cases: managing requirements

Testing

Reusable Software

Object-Based Software

Object-Oriented Software

Collaborations

Testing Object-Based Software: chapter 21

- OO without inheritance and virtual functions
- classes with data and function members
- pre- and post-conditions for methods, class invariants (design by contract: book is not precise here!)

Subsystem Size and Test Implementation

- Class: smallest reasonable subsystem
  - testing methods in isolation: would require stubs
  - but we are looking for interaction errors
  - class too small a test unit

Groups of classes

- Often designed to work together
- Test them together
- Often have their own collective invariant

- Three classes
  - Object_to_be_displayed, Visual_representation,
- Invariant
  - any object to be displayed has at most one

    keyboard controllers
- Makes no sense to test in isolation

Testing a group of methods of a

- Will often still test group of classes since methods use other classes
- Create pairs of sequences which are supposed to give same object
  - example: o1 and o2 should be the same

Class Requirement Catalog (21.3)

- Integration requirements should be listed once and for all in a class requirement catalog for “important” classes

Organization of class requirement catalog

- Object use requirements
- State machine requirements
- Member function requirements

Organization of class requirement catalog

from class invariant: useful independent of member function called

if class has state machine or statechart

- Member function requirements
  - preconditions,
- Collaborations of objects

Object-Oriented Software 1: Inheritance (chapter 22)

- Inheritance structure of object-oriented software leads to inheritance structure in test design documents
  - Class A => Extended Class A1
  - A test requirements => A1 test requirements
  - A test specifications => A1 test specifications

What needs to be retested in subclasses?

- Code which was tested in superclass, does it need to be retested in subclass?
- Example:

```
class Refrigerator {
  public:
      void set_desired_t(int t);
      int get_t();
      void calibrate();
   private
      int t;}
```

What needs to be retested in subclasses?

- `set_desired_t` allows the temperature to be between 5 and 20 degrees C. `calibrate()` puts the actual refrigerator through several cooling cycles and uses sensor readings to calibrate the cooling unit

```
class Refrigerator {
  public:
      void set_desired_t(int t);
      int get_t();
      void calibrate();
   private
      int t;}
```

What needs to be in subclasses?

allows the temperature to be between -5 and 20 degrees C. `calibrate()` is unchanged: do we have to retest calibrate? YES!

```
class Refrigerator {

       void set_desired_t(int
       int get_t();

   private
       int
```

What needs to be retested in subclasses?

- Assume that calibrate works by dividing sensor In the improved refrigerator we might get a division by zero error

  code. HAVE TO RETEST ALSO INHERITED, UNCHANGED CODE.

Test requirement checklist for subclass

- B inherits from A

```
class A {
public:
       virtual void member1();
       virtual void member2();
protected:
       Count count1;
       Count count2;}
class B : public A {
public:
       void member1();
       void member2()
protected:
       Count count3;}
```

Test requirement checklist for subclass

- To test B, create three test requirement checklists, one for each member function.
- Member1 and member2 are new code, their checklist must be complete
- Member 2 is unchanged: contains requirements that tests interactions with member1 and member3

What is gained?

- New checklist for B contains only requirements for new code and inherited code affected by the change: might require much less work than testing B from scratch

Reuse tests for superclass

Tests for A may satisfy some of the new test requirements: saving test specifications

Can be a cheap way to exercise B thoroughly

Design for Testability

- Implementation of concrete class must obey constraints described by its abstract superclass.
  - Constraint checks
    - self-check functions
  - Trace checks
    - check effects of a sequence of operations

Constraint check

Are all cardinality constraints satisfied?

–

- Self-check functions are also very useful during debugging: Demeter/C++

Trace checks

- Check effect of sequences of operations
  - Constant traces
    - o.f().g().h() does not change o
  - Equivalent traces
    - o.f().g().h() = o.a().b().c()
  - Simulation traces
    - design simulated objects (ATM customer)
      - generate tests from use cases
        - » use random generation

Testing collaborations

- Problem: tangled with other code
- White-box testing needs to recover the collaborations from the tangled code
- Identify roles played by the classes
- Same abstract collaboration may be present several times. Develop tests for abstract collaboration and adapt them

Testing collaborations

- Testing as an opportunity to improve the design
- How to describe the improved the design which is used to develop the test requirements and the test specifications
- This is a common theme: have to develop a model of the software under test

Testing collaborations

- Finding the abstract collaborations
  - Identify participants
    - important
      - they do real work
    - unimportant
      - they are only used as transmitters of control and information

Modeling collaborations

- Write down group of collaborating classes
- Separate code for pure navigation through classes from code which does interesting work
- Write test requirements for navigation
  - does code go to right objects in correct order?
- Write test requirements for interesting code

Modeling collaborations

- Do similar collaborations occur for similar sets of collaborating classes? They may have completely different names and different detailed connections but the connectivity relationships at a suitable level of abstraction are identical
- Study paper on modeling collaborations