CS 407
Design Patterns

Chain of responsibility example
Testing

Dr. Chad Williams
Central Connecticut State University

Group work

- A bank has a very sophisticated system to give their customers protection against bouncing checks by taking advantage of the number of accounts they hold as well as credit options.
  - Customers have 1 checking account, then optionally a savings account, and optionally 1 or more credit cards with different interest rates
  - When the CheckProcessor processs a customer's check it first checks if there are sufficient funds in the checking account if so stops, if not if the person has a savings account that is checked, otherwise try each of the customers many credit cards (lowest interest first)
- Create UML
- Create sequence diagrams for situations: checking acct sufficient; savings acct present but must go to credit; no savings but 3 credit cards none of which are sufficient

Contracts and invariants

Contracts and invariants

- Each interface or class defines a set of services based on their public methods that will be implemented
  - For interfaces all classes that realize the interface must implement the services
  - For classes the class itself must implement the services as well as all of its child classes
- Methods signatures only specify the type of method not the behavior

Design by contract

- Realizing interfaces and extending classes
  - Because polymorphism allows all classes that extend or implement a class to be treated the same it is critical that these subclasses behave in a way consistent with their parent's contracts

1) Preconditions of subclass' contract should be *no stronger* than that of the parent class
2) Post condition of subclass must be *no weaker* that that of the parent

  - Only when both of these rules are followed is polymorphism maintained

Contracts

- A contract of a method specifies its behavior – what service is provided
- Potential problems without contract
  - Incompleteness on some aspect of the behavior
  - Ambiguity and multiple interpretations
  - Contradictions with other contracts

Advantages

- Formal contracts resemble mathematical functions
- Advantages of formal contracts
  - Precision and unambiguous
  - Facilitates reasoning about the behavior (implementation and service users)

Contract features

- Contract specifies *preconditions* and *postconditions*
  - Precondition – boolean expression that must be true when the method is invoked. ie. If false it shouldn't be invoked
  - Postcondition – boolean expression that must be true when the method invocation returns

Formally documenting contracts

Generally recommended for complex public interfaces

Documenting conditions

- Within javadoc pre and post conditions can be documented using the tags @pre and @post
- Multiple @pre/@post indicate multiple sub conditions where each must be true for full pre/post conditions to be true
  - Note these tags are not defined in the standard javadoc but can be added
    - In NetBeans right click the project
    - Click on properties
    - Click on documenting
    - Add “-tag pre -tag post” to the additional javadoc options

Condition examples cont. List interface

```
/**
 * Returns the i-th element in the list.
 * @pre i >= 0 && i<size()
 * @post @result = element(i)
 * @post @nochange
 */
public Object element(int i);
```

- Precondition means the method can only be called when the precondition is met or it

More examples

```
/**
 * Returns first item in the list.
 * @pre !isEmpty()
 * @post @result == element(0)
 * @post @nochange
 */
public Object head();

/**
 * Returns last item in the list.
 * @pre !isEmpty()
 * @post @result == element(size()-1)
 * @post @nochange
 */
public Object last();
```

Invariants

- Object said to be *transient* if it is currently being manipulated and *stable* it has been initialized and is not being manipulated
- An *invariant* is a condition that must hold true for any well-formed stable state through the life of the instance
- Invariants define assumptions that can be made when interacting with an instance if the instance meets all its invariants it is *well-formed*

Invariant example
Doubly linked list

- If the list is empty, both `head` and `tail` should be null
- If the list is not empty `head` points to the first element in the list and `tail` points to the last
- `Count` should equal the number of elements in the list
- For each node the `next` should point to the successive node in the list and `prev` should point to the previous node in the list
- The first node/`head` node's `prev` should be null and the last/`tail`'s node `next` should be null

Testing

Testing

- For production system multiple phases of testing
  - **Unit testing** – Test each unit, independently before the units are integrated into the whole system
  - **Integration and system testing** – Integrate all of the components of a system to test the system as a whole
  - **Acceptance testing** – Validate that the system functions and performs as expected by the customers or the users

Unit testing

- Critical in making sure overall application is robust
  - For the whole system to function properly requires all individual units to function properly
  - Much easier to find errors at unit level than once multiple components combined

Developer's responsibility

- Other phases of testing responsibility of testing is by a group, unit testing is developer's responsibility
- Only phase where all paths of code can be tested
- Retest all functionality any time a change is made to ensure related functionality isn't broken

Testing coverage criteria

- Systematically test **all** aspects of the implementation
- Automatically check the correctness of the test results
- Systematic testing approaches
  - Black box testing
  - White box testing

Black box testing

- Derive test cases based on the specifications of component alone.
- Tests can be created independently of implementation
- In XP you create automated black box test conditions before developing implementation

Black box methodology

- Tests should include both valid and invalid elements
  - Test all exceptions (and conditions to produce)
  - Test valid – null,0,1,many
    - Pre/post conditions
  - Test equality/hash (more to come here)

White box testing

- Derives test cases based on the structure of the code implementing the functionality
- Tests cannot be fully created prior to completion of code
- Only way to fully ensure robustness of code

White box methodology

Test cases written to satisfy:

- **Statement coverage** – Every statement in the program must be executed at least once
- **Branch coverage** – Every branch of the program must be executed at least once

White box methodology cont.

- **Condition coverage** – Every boolean condition in the control statements must be evaluated to true and false at least once
- **Combination condition coverage** – For every compound boolean condition in the control statements of the program, every combination of truth value of the individual boolean condition must be excercised

Completeness of testing

- To test code thoroughly requires both black and white box testing
- To verify unit is fully tested requires testing every test case of both types of testing anytime a code change is made
- Result is unit testing needs to be automated for regression testing

Automating unit testing

- Identify each test case and expected output
- Write code to execute test and compare actual output vs. expected output
- Execute all test cases anytime releasing code

- **Enter JUnit**

JUnit testing overview

- Creates mirror of package structure allowing test to be created that can reference public, package, and protected methods…without adding a bunch of junk to production code!!
- For unit test one test class per actual class and executes both white and black box testing
- Execute all test before/during build

JUnit Annotations

- Annotations added before first line of method

Annotation	Description
@Test	The `@Test` annotation identifies a method as a test method.
@Before	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
@After	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.

JUnit Assertions

- Test that matches expected otherwise test fails

Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The String parameter is optional.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertFalse([message], boolean condition)	Checks that the boolean condition is false.
assertEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.

Junit Demo problem

Create 3 classes A, B, C (you can assume the Exception classes are already defined). Class A has a method that takes two arguments (doubles) *a* and *b* and returns a double. The function should calculate the (square root of *a)*/*b*. If *a* is negative it should throw NegAException, if *b* is zero it should throw BZeroException. Class B should have a method that calls the method on Class A and catches just the NegAException and prints a message indicating *a* can't be negative. Class C should call Class B's method and if BZeroException is thrown it should output stack debug information.

(Math function is “sqrt”)

Group work

- Blackbox test cases for Stack implementation
  - Methods on Stack
    - size()
    - push(Object obj)
    - Object pop() throws NoMoreElementsException
    - toString()

Group work

- Blackbox test cases for Set implementation
  - Methods on Set - Ensure unique set of objects in Set
  - size()
  - add(Object)
  - remove(Object)
  - Object[] toArray()

Group work

- White box test cases for following implementation

```
public boolean doSomething(int a, int b,
boolean c) throws BadException{
   if (((a<b)&&(c==true))||((b>c)&&(c==false)){
      return otherCall(c)  //returns true or false
   based on c
   }else if((a<b)&&(b>5)){
      throw new BadException();
   }else{
      return true;
   }
}
```