Data Structures

Software Lifecycle

CS284

Software Lifecycles

- A software development life cycle (SDLC), is a structure imposed on the development of a software product
- Describes the phases and tasks that take place during the software development process
- Several models exist
- "Waterfall" is the simplest
  - It is not used nowadays
  - Some of its ideas have been integrated to more modern models

General Lifecycle Model

- The phases are the same in most models:
  1. Requirements specification
  2. Design
  3. Implementation
  4. Testing
  5. Deployment and maintenance

Requirements

- A complete description of the behavior of a system to be developed
- It includes a set of use cases that describe all the interactions the users will have with the software
- However,
  - Customers typically have an abstract idea of what they want, but not what software should do
  - Incomplete, ambiguous, or even contradictory requirements are often encountered

Design

- A high-level solution to the problem
- Includes tasks such as
  - Division of the problem into modules
  - Data structure, algorithm and programming language selection
- Includes tools such as:
  - Flowcharts
  - Storyboards (if UI is important part of the software)
- Considerations:
  - Compatibility, Extensibility, Maintainability, Modularity, Reliability, Robustness, Security, Usability

Implementation

- Obvious
- Includes integration
  - Vertical: integration of subsystems to create functional entities (end-to-end systems)
  - Star: each subsystem is connected to all existing subsystems
  - Horizontal: specialized subsystem dedicated to communication between subsystems

Testing

- Runs a program or part of a program under controlled conditions to verify that results are as expected
- Detects program defects after the program compiles (all syntax errors have been removed)
- While extremely useful, testing cannot detect the absence of all defects in complex programs

Testing Levels

- Unit testing: tests the smallest testable piece of the software, often a class or a sufficiently complex method
  - We'll focus on this level
- Integration testing: tests integration among units
- System testing: tests the whole program in the context in which it will be used
- Acceptance testing: system testing designed to show that a program meets its functional requirements

Types of Testing

- Black-box (or closed-box or functional) testing:
  - Tests the item (method, class, or program) based on its interfaces and functional requirements
  - Is accomplished by varying input parameters across the allowed range and outside the allowed range, and comparing with independently calculated results
- White-box (glass-box, open-box, or coverage) testing:
  - tests the item (method, class, or program) with knowledge of its internal structure
  - exercises as many paths through the element as possible
  - provides appropriate coverage
    - statement – ensures each statement is executed at least once
    - branch – ensures each choice of branch (if, loops, etc.) is taken
    - path – tests each path through a method

Preparations for Testing

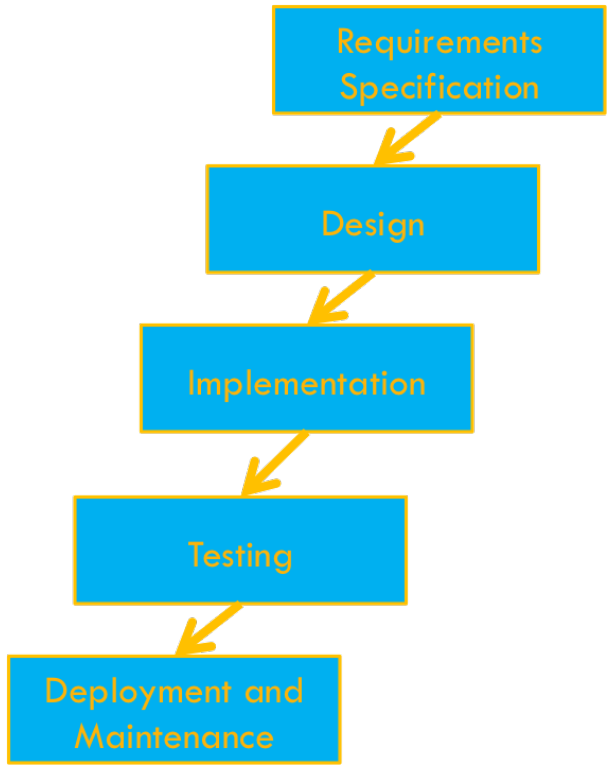
- A test plan should be developed early in the design stage – the earlier an error is detected, the easier and less expensive it is to correct it
- Aspects of test plans include deciding:
  - how the software will be tested
  - when the tests will occur
  - who will do the testing
  - what test data will be used

Final Phases

- Installation
- Maintenance

The Waterfall Model

- Sequential design process
- After each phase is finished, next one begins
- Advantage: bugs found early cost less (money, effort, time)
- Disadvantage: inflexibility
- Impossible to finish each phase perfectly without ever having to go back
- Nowadays other models are used (that build on some ideas of this one)
  - More in CS 347 Software Development Process



Basic Testing Tips

- Documenting Your Code
- Preconditions and Postconditions
- Developing Test Data

Documentation

- Carefully document method operation, parameter, and class attributes using comments
- Use `/**... */` and follow Javadoc conventions
- For example:

```
/**
 * Person is a class that represents a human being.
 * @author Koffman and Wolfgang
 * @version 1.2
 */
public class Person { ... }
```

Javadoc Tags

- Keywords recognized by Javadoc which define the type of information that follows.
- Come after the description (separated by a new line).
- Some common pre-defined tags:
  - `@author [author name]` identifies author(s) of a class or interface.
  - `@version [version]` version info of a class or interface.
  - `@param [arg. name] [arg. descrip.]` describes an argument of method or constructor.
  - `@return [descrip. of return]` describes data returned by method (unnecessary for constructors and void methods).
  - `@exception [exception thrown] [exception descrip.]` describes exception thrown by method.
  - `@throws [exception thrown] [exception descrip.]` same as `@exception`.

Javadoc Tags

```
public class Person {

    /** The age at which a person can vote */
    private static final int VOTE_AGE = 18;

    /**
     * Determines whether a person can vote.
     * @param year The current year
     * @return true if the person's age is greater than or equ
     *         the voting age
     */
    public boolean canVote(int year) { ...}
}
```

Preconditions and Postconditions

- A precondition is a statement of any assumptions or constraints on the input parameters or the state of the recipient object before a method begins execution
- It is a requirement that must be met by the caller of the method
- It is the responsibility of the caller never to call the method if the requirement is violated
- Eg. In a `deposit` method of a `BankAccount` calls, the amount to be deposited should be non-negative

Preconditions and Postconditions

```
/**
    Deposits money into this account
    @param amount the amount of money to deposit
    (Precondition: amount>=0)
*/
public void deposit(int amount) {
 ...
}
```

- The method is free to do anything if the precondition is not fulfilled

Preconditions and Postconditions

```
/**
    Deposits money into this account
    @param amount the amount of money to deposit
    (Precondition: amount>0)
*/
public void deposit(int amount) {
 ...
}
```

- Should the code check that the precondition is met?
- If so, what should be done if the condition is not met?
- What do you think of this?

```
public void deposit() {
 if (amount < 0) return;
 ...
}
```

Preconditions and Postconditions

```
/**
    Deposits money into this account
    @param amount the amount of money to deposit
    (Precondition: amount>0)
*/
public void deposit(int amount) {
 ...
}
```

- Should the code check that the precondition is met?
  - May be inefficient if may checks have to be made
- Or should it assume that the precondition is met?
  - May be dangerous
- Convenient compromise: assertion checking

Preconditions and Postconditions

```
%[
%   linebackgroundcolor={%
%    \btLstHL<1>{7}%
 % }]
/**
    Deposits money into this account
    @param amount the amount of money to deposit
    (Precondition: amount>0)
*/
public void deposit(int amount) {
    assert amount>0;
    balance = balance + amount:
}
```

- Assertion: condition believed true at a particular location in the program
- Failure of assertions will be notified to the programmer (if assertion checking is enabled)
- Assertions can be enabled or disabled

Preconditions and Postconditions

- A postcondition describes the result of executing the method, including any change to the object's state
- A method's preconditions and postconditions serve as a contract between a method caller and the method programmer

```
/**
    Deposits money into this account
    @param amount the amount of money to deposit
    (Precondition: amount>0)
    (Postcondition: adds amount to balance)
*/
public void deposit(int amount) {
  ...
}
```

JUnit

- JUnit is a unit testing framework for Java (see Appendix C)
  - A unit generically refers to a function, method, module, package, etc.
- To install download .jar file from `http://junit.org` and then import it into your project

Example

```
package BinaryArithmetic;
import static org.junit.Assert.*;
import org.junit.Test;

public class BinaryNumberTest {
  @Test
  public void testAdd() {
    BinaryNumber b1 = new BinaryNumber("1010");
    BinaryNumber b2 = new BinaryNumber("1100");
    // assert statements
    assertEquals("1010 + 1100 must be 0001", "0001", b1.add(b2
  }
}
```

Another Example: Testing Array Search

```
public class Search {
   private int[] x;

   Search() {
       x = new int[]{5, 12, 15, 4, 8, 12, 7};
   }

   Search(int[] y) {
       x=y;
   }

   public int search(int target) {
      for (int i = 0; i < x.length; i++) {
                if (x[i] == target)
                        return i;
      }
      return -1;
   }
}
```

Testing Array Search

```
package Search;
import static org.junit.Assert.*;
import org.junit.Test;

public class SearchTest {

    @Test
    public void testForNonExtantElement() {
        Search s = new Search();
        // assert statements
        assertEquals(-1, s.search(2));
    }

    @Test
    public void testForTargetAsFirstElement() {
        Search s = new Search();
        // assert statements
        assertEquals(0, s.search(5));
    }
```

Testing Array Search

```
@Test
public void testForTargetAsLastElement() {
    Search s = new Search();
    // assert statements
    assertEquals(6, s.search(7));
}

@Test
public void testForTargetForMultipleOccurrenceOfTarget() {
    Search s = new Search();
    // assert statements
    assertEquals(1, s.search(12));
}
```

Testing Array Search

```
    @Test
    public void testForTargetSomewhereInTheMiddle() {
        Search s = new Search();
        // assert statements
        assertEquals(3, s.search(4));
    }

    @Test
    public void testFor1ElementArray() {
        int[] y = {10};
        Search s = new Search(y);
        // assert statements
        assertEquals(0, s.search(10));
    }
```