

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
- ▶ Existing models:
 - ▶ Waterfall model
 - ▶ Unified process model
 - ▶ Agile development model
 - ▶ ...

The Waterfall Model

Finish one phase before another phase begin:



The Waterfall Model

- ▶ **Requirements specification**

- ▶ Program manager gather customers' needs

- ▶ **Design**

- ▶ Engineers develop high-level design

- ▶ **Implementation**

- ▶ Classes and methods are coded into the target programming language

- ▶ **Testing**

- ▶ Verify the software meets the requirements

- ▶ **Deployment and maintenance**

- ▶ Modifying the software to improve performance or correct faults

Stage 1: Requirements Specification

- ▶ Program manager discusses with subject matter experts (SME) and stake holders to decide:
 - ▶ Customers' needs
 - ▶ Problem the software will resolve
 - ▶ Functionality the customer desires
- ▶ Requirements documents
 - ▶ Functional requirements, usability requirements, interaction requirements, security requirements
- ▶ Requirements vs. use cases
 - ▶ Requirements: *Shift all digits in a binary number any number of places to the left or right*
 - ▶ Use case: *If the amount of shift $>$ length of binary number and direction = right, should return 0*

Requirements Document

SAMPLE Project Requirements Document – Library Blog

Stakeholders: John Doe (Web Services), Jane Smith (Programmer), Peter Rabbit (Public Services, Sponsor), Raggedy Ann (Administration)

Task Force: John Doe, Jane Smith, Peter Rabbit, Raggedy Ann, Raggedy Andy.

Document Modification History

Version	Date	Author	Description
1.0	05/16/2011	Raggedy Andy	Initial Version
1.1	05/18/2011	Peter Rabbit	Added changes from stakeholders meeting
1.2	05/23/2011	Jane Smith	Added technical documentation details

Project Description

The library will have a public-facing blog that will serve to communicate library news, events and resources, as well as providing the library's user community with the ability to comment on posts.

Service Need

The University of Awesomeness Library has a need for communicating changes to policies, procedures, and resources to its users. Library blogs have been shown to be an effective method of communicating with library users.¹ We currently have a home-grown blog in place, but it lacks many of the functionalities we require.

Requirements Elicitation

However:

- ▶ Customers typically have an vague idea of what they want, not what software should do
- ▶ Incomplete, ambiguous, or even contradictory requirements are often encountered

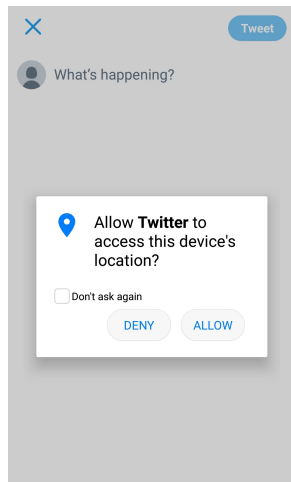
Requirements clarification/elicitation

- ▶ *What is the format of the binary number? String or int array?*
- ▶ *What if the binary number is negative?*
- ▶ *What is the output format? String or int array?*
- ▶ *What if amount > 64?*

Requirements Engineering

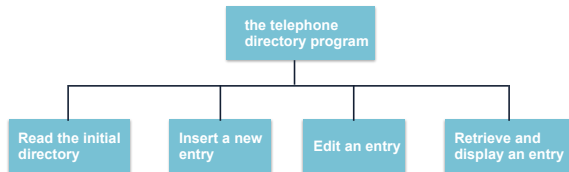
Requirements engineering:

- ▶ A scalable and efficient way of gathering software requirements
- ▶ Mining requirements from massive text data
 - ▶ Bug report, e.g., *search bar not working*
 - ▶ User reviews, e.g., *After the update, I could not find xxx...*
 - ▶ Software documentation, e.g., mining security requirements [Liu et al. 2018]



Stage 2: Design

- ▶ Two stages of the design process:
 - ▶ Logical design: abstract representation of the software data flow, inputs, outputs
 - ▶ Physical design: hardware, e.g., storage, turning the logical design into reality
- ▶ Approaches:
 - ▶ Top-down design: Division of the problem into modules
 - ▶ Object-oriented design: identify a set of objects and their interactions



Stage 2: Design

- ▶ Tools:
 - ▶ UML Diagram
 - ▶ Storyboards (if UI is important part of the software)
- ▶ Considerations:
 - ▶ Compatibility, Extensibility, Maintainability, Modularity, Reliability, Robustness, Security, Usability

Stage 3: Implementation

- ▶ 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

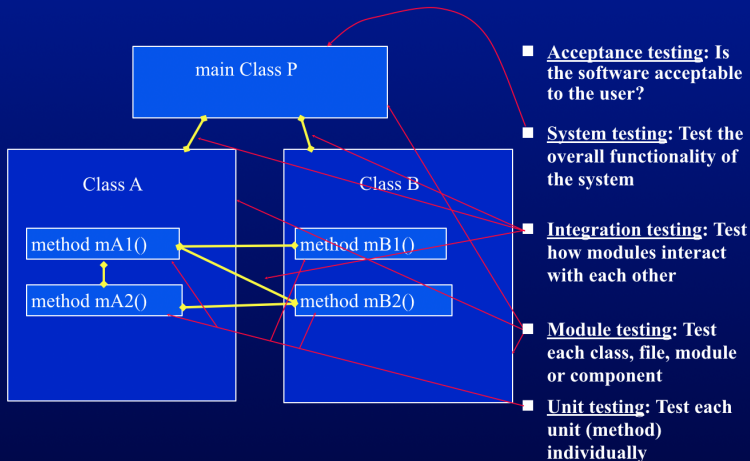
Stage 4: Testing

- ▶ 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 to make sure they work together and meet the requirements
- ▶ **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

Testing at Different Levels



Source: UIUC CS498TX Software testing, Fall 2017

Types of Testing

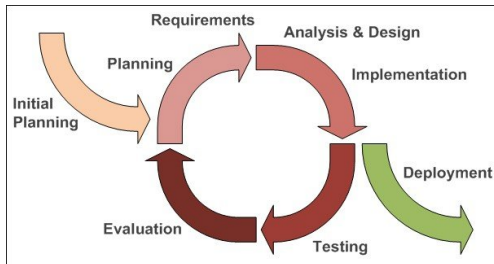
- ▶ Black-box (or closed-box or functional) testing:
 - ▶ Tests the item (method, class, or program) based on its interfaces and functional requirements
 - ▶ Is accomplished by varying input parameters across the allowed range and outside the allowed range, and comparing with independently calculated 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

Other SDLC Models

- ▶ Water fall: sequential design process
 - ▶ 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 (More in CS 347 Software Development Process)



Software Lifecycles

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 equal  
     *         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 many checks have to be made
- ▶ Or should it assume that the precondition is met?
 - ▶ May be dangerous
- ▶ Convenient compromise: assertion checking

Preconditions and Postconditions

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

More on Software Testing

Testing vs. debugging

- ▶ Testing: Finding inputs that cause the software to fail
- ▶ Debugging: The process of finding a fault given a failure

Test case generation:

- ▶ Discovering code in the program that is not being exercised by test cases


2. Logic – Active Clause Coverage

$((a > b) \text{ or } G) \text{ and } (x < y)$

With these values
for G and $(x < y)$,
 $(a > b)$ determines
the value of the
predicate

1	T	F	T
2	F	F	T
3	F	T	T
4	F	F	T
5	T	T	T
6	T	T	F

duplicate



Source: UIUC CS498TX Software testing, Fall 2017