Objectives

- Testing

No Silver Bullet: Essence and Accidents of Software Engineering *by Frederick P. Brooks, Jr., 1986*

"Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.

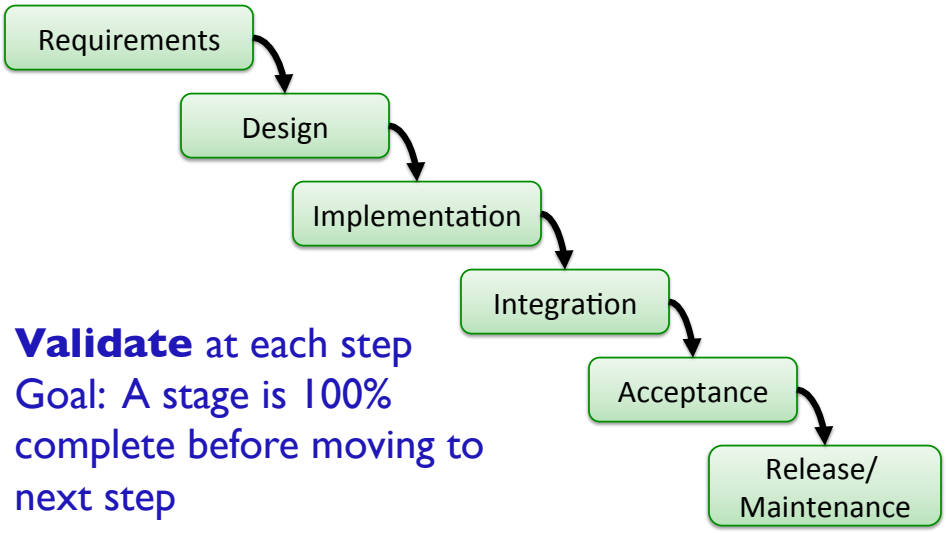
"The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet--something to make software costs drop as rapidly as computer hardware costs do.

"But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity. In this article, I shall try to show why, by examining both the nature of the software problem and the properties of the bullets proposed."

Who is Fred Brooks?

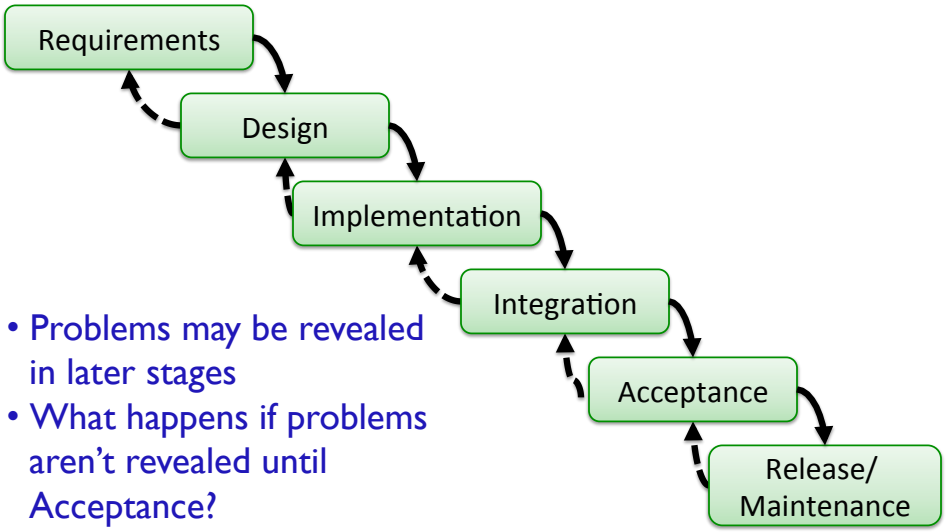
- UNC Professor
- Turing Award winner
- "The most important single decision I ever made was to change the IBM 360 series from a 6-bit byte to an 8-bit byte, thereby enabling the use of lowercase letters. That change propagated everywhere."

Traditional Software Engineering Process: Waterfall Model



Validate at each step
Goal: A stage is 100% complete before moving to next step

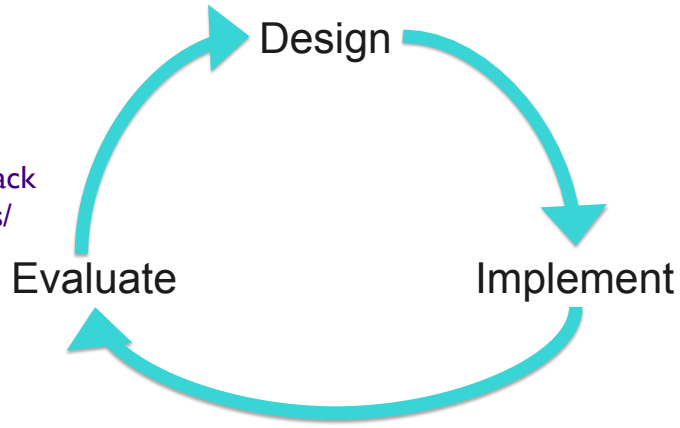
Feedback in Waterfall Model



- Problems may be revealed in later stages
- What happens if problems aren't revealed until Acceptance?

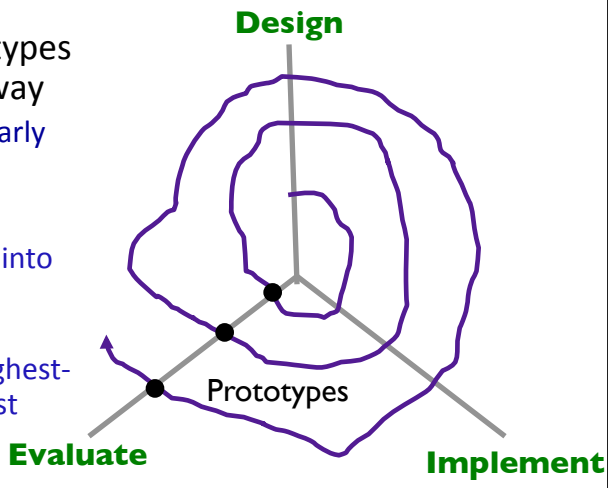
Iterative Design

Get feedback from users/ clients



Spiral Model

- Idea: smaller prototypes to test/fix/throw away
  - Finding problems early costs less
- In general…
  - Break functionality into smaller pieces
  - Implement most depended-on or highest-priority features first

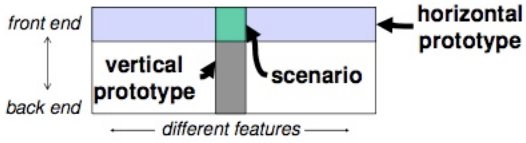


[Boehm 86]

Radial dimension: cost

Prototypes

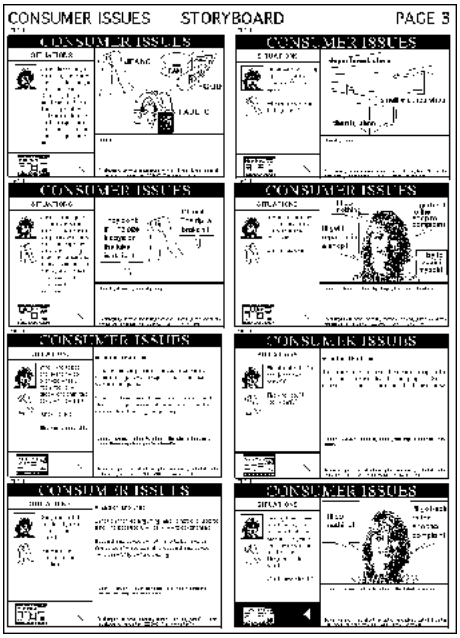
- Purpose/Dimensions
  - Functionality
  - Interaction
  - Implementation
- Fidelity:
  - Low: omits details
  - High: closer to finished project
  - Multi-dimensional
    - Breadth: % of features covered
      - Only enough features for certain tasks
    - Depth: degree of functionality
      - Limited choices, canned responses, no error handling



From Nielsen, *Usability Engineering*

Low Fidelity Prototypes

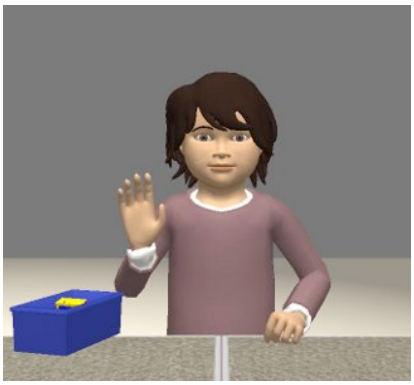
- Media: Paper
- Examples: storyboard, sketches, flipbook, flow diagram



High Fidelity Prototypes

- Media: Flash, HTML (non-interactive), PowerPoint, Video
- Examples: Mockups, Wizard of Oz

Virtual Peer for Autistic Children



http://www.articulab.justinecassell.com/projects/samautism/index.html

How to Implement an Effective Solution

- Understand the problem (interact with ***people***)
- Understand external constraints (interact with ***people***)
- Design an effective solution to the problem
- While designing the solution, design some ***tests*** to verify that the problem is solved (and remains solved)
- Code the effective solution to the problem
- Teach other team members about your solution to the problem (interact with ***people***)

Spiral Model Steps

- Design a {method, class, package}
- Implement the {method, class, package}
- Test the {method, class, package}
- Fix the {method, class, package}
- Deploy the {method, class, package}
- Get feedback
  - Probably will require modifications to design
  - May even need to rollback a previous version
- Repeat

Agile Development Framework: Scrum

- **The Scrum framework in 30 seconds**
  - Product owner creates prioritized wish list, *a product backlog*
  - Team works in a *sprint*, usually 2-4 weeks
    - During planning, team picks a subset of wish list, *a sprint backlog*, and decides how to implement those pieces
    - Daily Scrum: team meets daily to assess its progress
      - ScrumMaster keeps the team focused on its goal
    - At end of sprint, work should be potentially shippable:
      - ready to hand to a customer, put on a store shelf, or show to a stakeholder
    - The sprint ends with a sprint review and retrospective
  - Repeat sprint

`https://www.scrumalliance.org/why-scrum`

SOFTWARE TESTING PROCESS

A Bad Role Model

 http://imgur.com/HBSbn

Microsoft Windows Vista Testing

- Beyond their internal testing …
  - 5 million people beta tested
  - 60+ years of performance testing
  - 1 Billion+ Office 2007 sessions
- Still, users found correctness, stability, robustness, and security bugs

Type 1 Bugs: Compile-Time

- Syntax errors
  - Missing semicolon, parentheses
- Compiler notifies of error
- Cheap, easy to fix

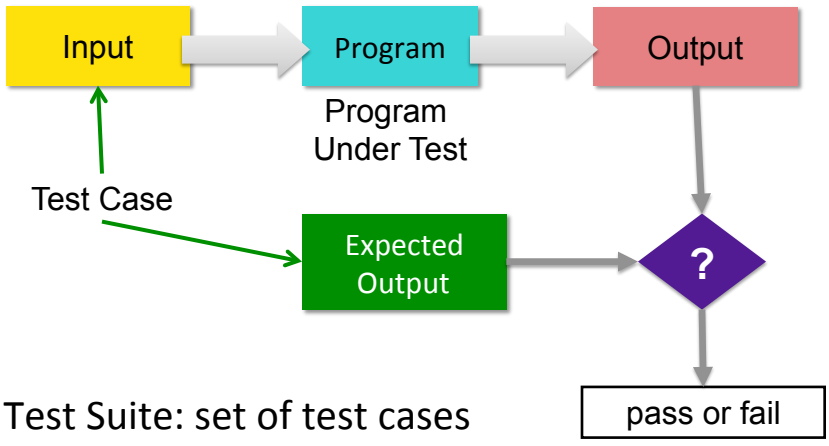
Type 2 Bugs: Run-Time

- Usually logic errors
- Expensive to locate, fix

Aside: Objections to "Bug" Terminology

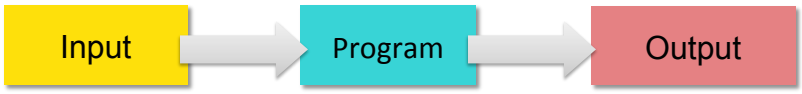
- "Bug"
  - Sounds like it's just an annoyance
    - Can simply swat away
  - Minimizes potential problems
  - Hides programmer's responsibility
- Alternative terms
  - **Defect**
  - **Fault**

Software Testing Process



- Test Suite: set of test cases

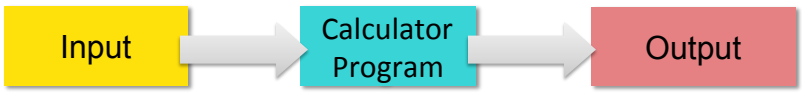
Software Testing Process



- Tester plays devil's advocate
  - ***Hopes*** to reveal problems in the program using "good" test cases
  - Better tester finds than a customer!

How is **testing** different from **debugging**?

How Would You Test a Calculator Program?



Operands, operators, expected output	adds, subtracts, multiplies, divides	Numerical Answer

- What test cases: input and expected output?

Example Test Cases for Calculator Program

- Basic Functionality
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Order of operations
- Invalid Input
  - Letters, not-operation characters (&,$, …)
- "Tricky" Cases
  - Divide by 0
  - Negative Numbers
  - Long sequences of operands, operators
  - VERY large, VERY small numbers

Types of Testing
(Non-Exhaustive)

- Black-box testing
- White-box testing
- Non-functional testing
- Acceptance testing

Ideas or definitions of any of these?

Types of Testing
(Non-Exhaustive)

- Black-box testing
  - Test *functionality* (e.g., the calculator)
  - No knowledge of the code
  - Examples of testing: boundary values
- White-box testing
  - Have access to code
  - **Goal**: execute *all* code
- Non-functional testing
  - Performance testing
  - Usability testing (HCI)
  - Security testing
  - Internationalization, localization
- Acceptance testing
  - Customer tests to decide if accepts product

Levels of Testing

- Unit
  - Tests minimal software component, in isolation
  - For us, Class-level testing
  - Web: Web pages (Http Request)
- Integration
  - Tests interfaces & interaction of classes
- System
  - Tests that completely integrated system meets requirements
- System Integration
  - Test system works with other systems, e.g., third-party systems

UNIT TESTING

Why Unit Test?

- Verify code works as intended in isolation
- Find defects ***early*** in development
  - Easier to test small pieces
  - Less cost than at later stages

Why Unit Test?

- Verify code works as intended in isolation
- Find defects ***early*** in development
  - Easier to test small pieces
  - Less cost than at later stages
- As application evolves, new code is more likely to break existing code
  - Suite of (small) test cases to run after code changes
  - Also called **regression** testing

Some Approaches to Testing Methods

- Typical case
  - Test typical values of input/parameters
- Boundary conditions
  - Test at boundaries of input/parameters
  - Many faults live "in corners"
- Parameter validation
  - Verify that parameter and object bounds are documented and checked
  - Example: pre-condition that parameter isn't null

➥ All black-box testing approaches

Another Use of Unit Testing: Test-Driven Development

- A development style, evolved from Extreme Programming
- Idea: write tests first *without code bias*
- The Process:
  1. Write tests that code/new functionality should pass
     - Like a specification for the code (pre/post conditions)
     - All tests will initially *fail*
  2. Write the code and verify that it passes test cases
     - Know you're done coding when you pass **all** tests

How do you know you're "done" in traditional development?

What assumption does this make?

Software Testing Issues

- How should you test? How often?
  - Code may change frequently
  - Code may depend on others' code
  - A lot of code to validate
- How do you know that an output is correct?
  - Complex output
  - Human judgment?
- What caused a code failure?

➥ Need a *systematic, automated, repeatable* approach

Characteristics of Good Unit Testing

- **Automatic**
- **Thorough**
- **Repeatable**
- **Independent**

Why are these characteristics of good (unit) testing?

Characteristics of Good Unit Testing

- **Automatic**
  - Since unit testing is done frequently, don't want humans slowing the process down
  - Automate executing test cases and evaluating results
  - Input: in test itself or from a file
- **Thorough**
  - Covers all code/functionality/cases
- **Repeatable**
  - Reproduce results (correct, failures)
- **Independent**
  - Test cases are independent from each other
  - Easier to trace fault to code

JUNIT

JUnit Framework

- A framework for unit testing Java programs
  - Supported by Eclipse and other IDEs
  - Developed by Erich Gamma and Kent Beck
- Functionality
  - Write tests
    - Validate output, automatically
  - Automate execution of test suites
  - Display pass/fail results of test execution
    - Stack trace where fails
  - Organize tests, separate from code

But, you still need to come up with the tests!

Aside: Framework

A ***framework*** is a basic conceptual structure used to solve or address complex issues.

This very broad definition has allowed the term to be used as a buzzword, especially in a software context.

Testing with JUnit

```
tests
   └ CDTest
     DVDTest
     MediaItemTest
```

- Typical organization:
  - Set of testing classes
  - Testing classes packaged together in a **`tests`** package
    - Separate package from code testing
- A test class typically
  - Focuses on a specific class
  - Contains methods, each of which represents another test of the class

Structure of a JUnit Test

1. Set up the test case (optional)
   - Example: Creating objects
2. Exercise the code under test
3. Verify the correctness of the results
4. Teardown (optional)
   - Example: reclaim created objects

Annotations

- Testing in JUnit 4: uses **annotations**
- Provide data about a program that is not part of program itself
- Have no direct effect on operation of the code
- Example uses:
  - @Override: method declaration is intended to override a method declaration in parent class
    - If method does not override parent class method, compiler generates error message
  - Information for the compiler to suppress warnings (@SupressWarnings)

Tests are Methods

- Mark your testing method with `@Test`
  - From `org.junit.Test`

```
public class CalculatorTest {

      @Test
      public void addTest() {
            …
      }
}
```

Class for testing the Calculator class

A method to test the "add" functionality

- Convention: Method name describes what you're testing

Assert Methods

- Variety of assert methods available
- If fail, throw an exception
- Otherwise, test keeps executing
- All `static void`
- Example:
  `assertEquals(Object expected, Object actual)`

```
@Test
public void addTest() {
      …
      assertEquals(4, calculator.add(3, 1));
}
```

Assert Methods

- To use asserts, need *static* import:

```
import static org.junit.Assert.*;
```

  - `static` allows us to not have to use classname
- More examples
  - `assertTrue(boolean condition)`
  - `assertSame(Object expected, Object actual)`
    - Refer to same object
  - `assertEquals(double expected, double actual, double delta)`

Example Uses of Assert Methods

```
@Test
public void testEmptyCollection() {
    Collection collection = new ArrayList();
    assertTrue(collection.isEmpty());
}
```

`assertEquals(double expected, double actual, double delta)`

```
@Test
public void testPI() {
    final double ERROR_TOLERANCE = .01;
    assertEquals(Math.PI, 3.14, ERROR_TOLERANCE);
}
```

Will fail if ERROR_TOLERANCE = .001

Set Up/Tear Down

- May want methods to set up objects for every test in the class
  - Called **fixtures**
  - If have multiple, no guarantees for order executed

```
@Before
public void prepareTestData() { ... }

@Before
public void setupMocks() { ... }

@After
public void cleanupTestData() { ... }
```

Executed before **each** test method

Example Set Up Method

```
private CD testCD;

@Before
public void setUp() {
     testCD = new CD("CD title", 100, 1997,
                     "CD Artist", 11);
}
```

`@Before` Executed before **each** test method
Can use `testCD` in test methods

Expecting an Exception

- Handling Error Cases
  - Sometimes an exception *is* the expected result

Add an "expected" attribute:

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
   ArrayList emptyList = new ArrayList();
   Object o = emptyList.get(0);
}
```

Test case passes iff exception thrown

Set Up/Tear Down For Class

- May want methods to set up objects for set of tests
  - Executed once before any test in class executes

```
@BeforeClass
public static void
setupDatabaseConnection() { ... }

@AfterClass
public static void
teardownDatabaseConnection() { ... }
```

JUNIT IN ECLIPSE

Using JUnit in Eclipse

- Eclipse can help make our job easier
  - Automatically execute tests (i.e., methods)
  - We can focus on coming up with tests

Using JUnit in Eclipse

- In Eclipse, go to your `MediaItem` project
- Create a new JUnit Test Case (under Java)
  - Use JUnit **4**
    - Add junit to build path
  - Put in package `media.tests`
  - Name: `DVDTest`
  - Choose to test `DVD` class
    - Select `setUp` and `tearDown`
    - Select methods to test
- Run the class as a JUnit Test Case

Example

- Test method that gets the length of the DVD
  - Revise: Add code to `setUp` method that creates a DVD
- Notes
  - Replaying all the test cases: right click on package
  - FastView vs Detached
  - Hint: CTL-Spacebar to get auto-complete options

Unit Testing & JUnit Summary

- Unit Testing: testing smallest component of your code
  - For us: class and its methods
- JUnit provides framework to write test cases and run test cases automatically
  - Easy to run again after code changes
- JUnit Resources available from Course Page's "Resource" Link, under Java
  - API
  - Tutorials

Project 1: Testing Practice

- Given: a `Car` class that only has enough code to compile
- Your job: Create a ***good*** set of test cases that ***thoroughly***/***effectively*** test `Car` class
  - Find faults in my faulty version of `Car` class
  - Start: look at code, think about how to test, set up JUnit tests
  - Written analysis of process

Project 1: Testing Practice

- 1st: Email me and your teammate with the name of your team
  - I will create a repository that the pair can work on together

Looking Ahead

- More Testing!
- Extra credit assignment