Test-Driven Development

SC12 Educator's Session

November 13, 2012

Introduction

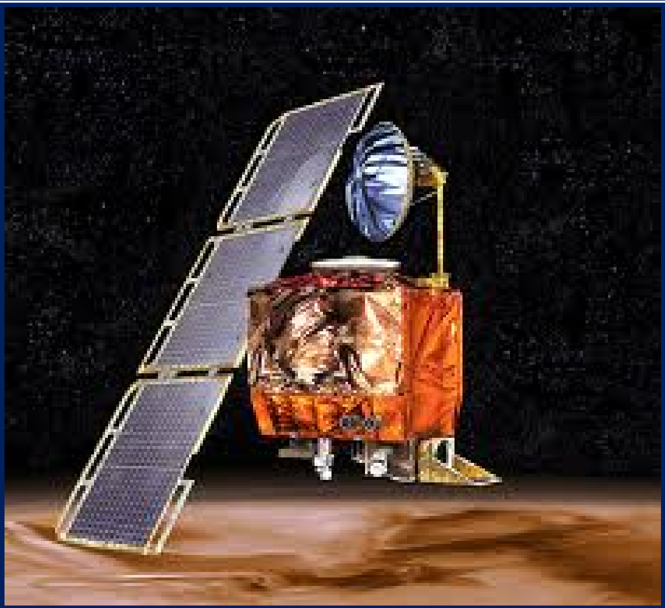
- Name
- Institution
- Knowledge of Software Engineering
- What you want to get out of today's tutorial

Slides Available

- SC12 Website
  - Communities-> HPC Educators -> HPC Educator’s Program -> 13

Motivating Example: Mars Climate Orbiter

- $125M satellite
- Goal: Help scientists understand Mars water history and potential for life
- Lost because of metric to English measurement conversion



Motivating Example:
Climategate

- Correctness of science called into question because of software quality factors
- Scientists do not always use well-documented practices
- Resulted in call for better transparency into software development processes

Outline

- Software Quality
- Overview of Testing
- Automated Testing Tools
- Test-Driven Development

SOFTWARE QUALITY

Software Quality

- Multiple Definitions
- Developer’s View vs. User’s View
  - Developers = Correctness
  - Users = Reliability

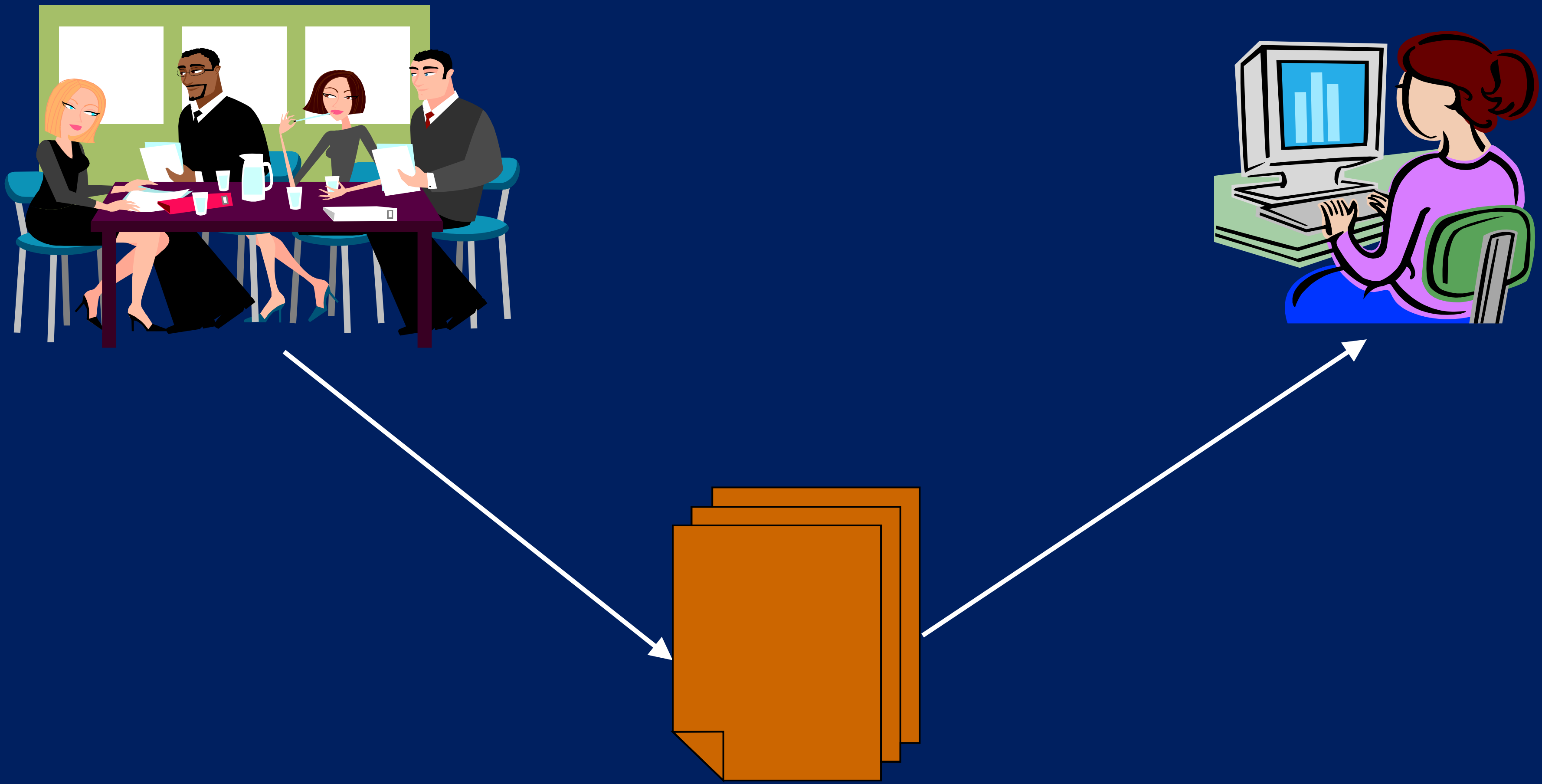
Definitions

- Software must do the right things
  - Perform the right functions
  - Often referred to as Validation
- Software must do things right
  - Perform intended functions without problems
  - Often referred to as Verification
- Together referred to as V&V

Quality Definitions: Defects

- Failure
  - “Inability of the system to perform its required function within specified performance requirements”
  - *Something goes wrong at execution*
- Fault
  - “An incorrect step, process, or data definition in a computer program”
  - *A mistake written down in a document*
- Error
  - “A human action that produces an incorrect result”
  - *The misunderstanding on the part of the human*

Quality Definitions:
Defects



Quality Focus

- Customers/Users
  - External
  - Failures – which ones, likelihood, severity, etc.

- Developers
  - Internal
  - Faults – which ones, what type, severity, etc.

QA Activities:
Types

- Defect Prevention
  - Error blocking
  - Error source removal
- Defect Reduction
  - Inspection
  - Testing
- Defect Containment
  - Fault-tolerance techniques to localize failure
  - Failure containment to avoid catastrophic failure

DEFECT PREVENTION

Defect Prevention: Introduction

- Reduce the chance of fault injection
- Approach depends on source
  - Human misconceptions
    - Education and Training
  - Imprecise design and implementation
    - Formal methods
  - Non-conformance to processes or standards
    - Process conformance or standard enforcement
- There may be specific tools or technologies that can also help
- Important to establish the correct root-cause

Defect Prevention: Education and Training

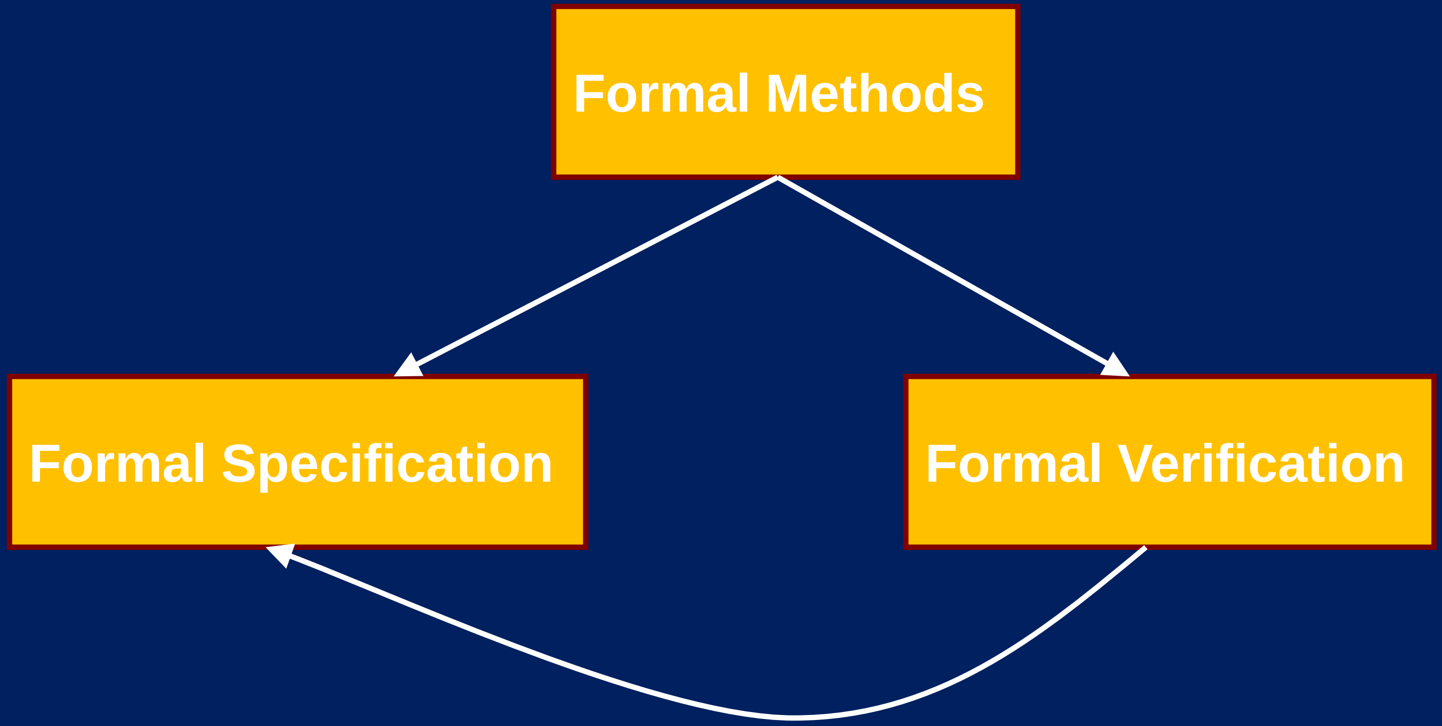
- People are most important factor in quality and success
- Education and Training can improve the quality of the work done by practitioners
- Elimination of misconceptions will reduce the probability of defect injection

Defect Prevention: Education and Training

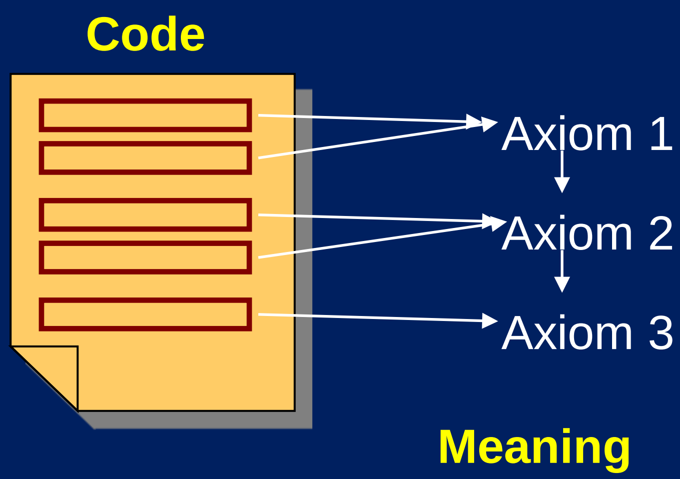
- Product and Domain Specific Knowledge
  - Unfamiliarity could lead to misunderstandings
- Software Development Expertise
  - Poorly written requirements/design can lead to problems
- Knowledge about tools, methods, techniques
  - Lack of knowledge could lead to misuse
- Development Process Knowledge
  - Hard to properly implement the process if developers do not understand it

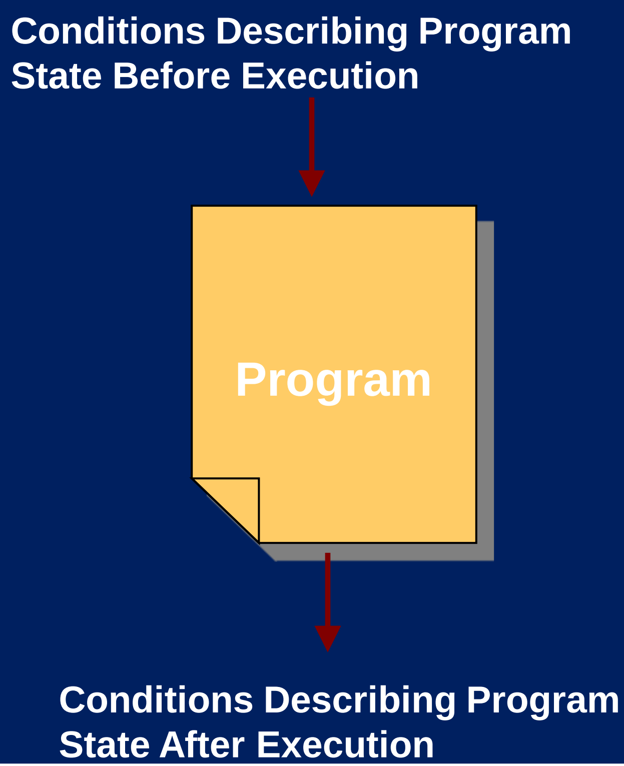
Defect Prevention: Formal Methods

Help eliminate error sources and verify the absence of faults



Formal Methods:
Axiomatic Approach



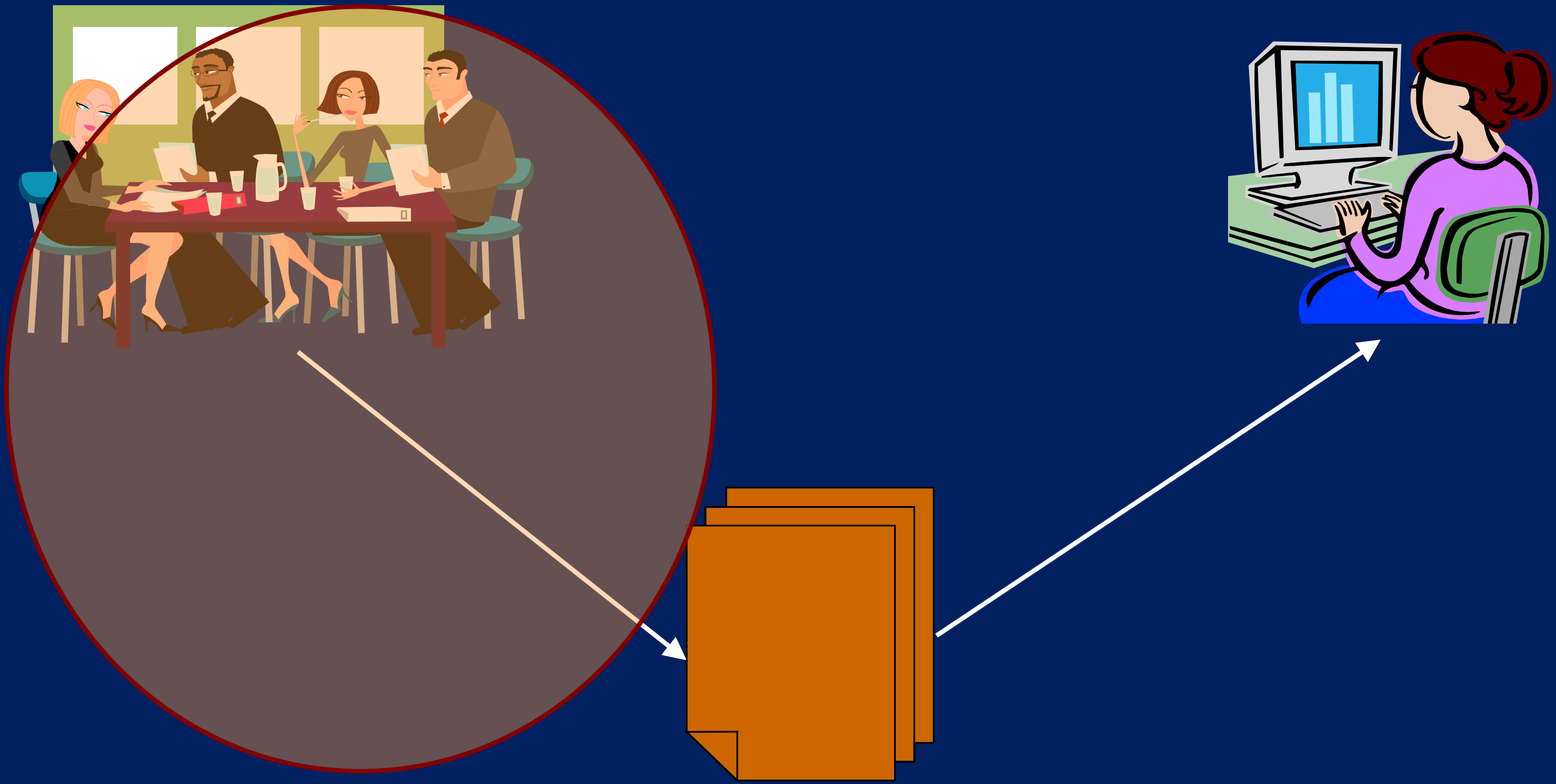


Biggest obstacle to use of formal methods:
Cost

Defect Prevention: Other Techniques

- Use of additional software development methodologies (besides Formal Methods)
  - Prevent extra functionality
  - Reduce complexity
- Better management
  - Concrete process definition
  - Enforcement of standards
- Use of specific tools
  - Enforce coding standards

QA Activities:
Defect Prevention



DEFECT REDUCTION

Defect Reduction: Introduction

- Unrealistic to expect Defect Prevention step to stop all defects

- Different approaches
  - Inspection
  - Testing
  - Other techniques

Defect Reduction: Testing

- Execution of software and checking results
  - Locates failures
  - Isolate and fix the fault(s) that led to the failure
- When to test
  - Need some executable
  - Unit tests of components through acceptance test of entire system
  - Can also use prototypes

Defect Reduction: What to Test

- Functional (black box)
  - External behavior
  - User observable behavior
  - Focus: reducing the chances of a target user encountering a functional problem
- Structural (white box)
  - Internal structure
  - Correct implementation
  - Focus: reduce internal faults so the software is less likely to fail in an unknown situation

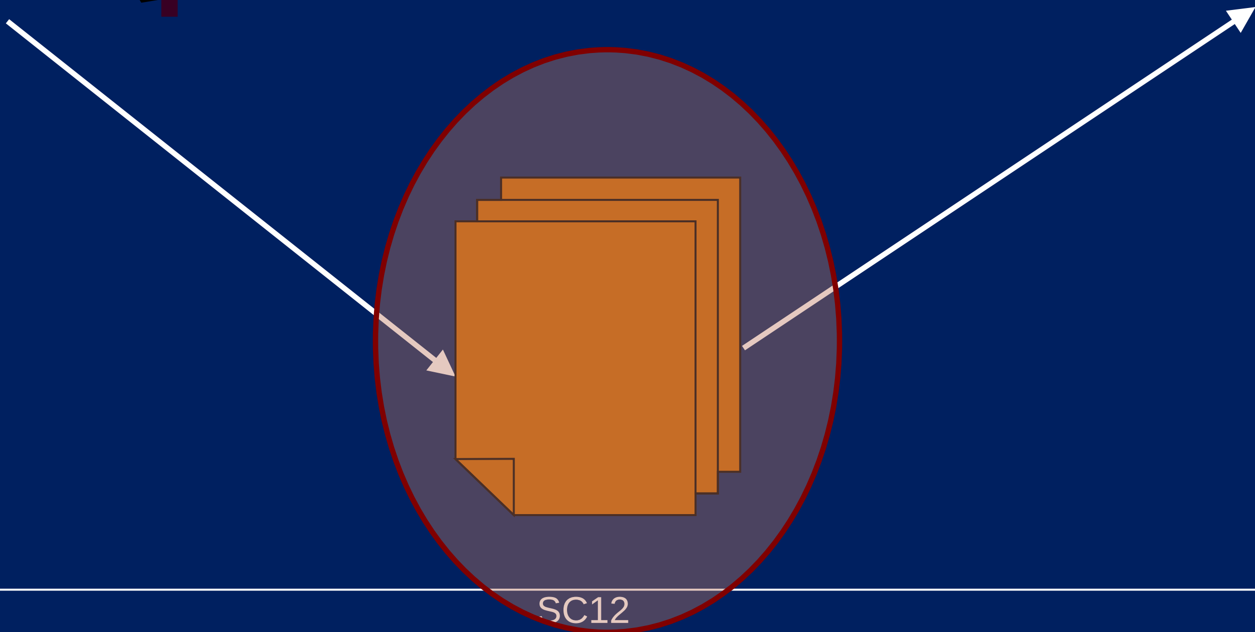
Defect Reduction: When to Stop Testing

- Can use coverage criteria
  - Assumption: higher coverage → fewer remaining defects
  - Functional or Structural
- Reliability goals
  - More objective
  - Measures what users are likely to encounter
  - Can be tailored for anticipated user groups

Defect Reduction: Observations

- Many other techniques available
- In-field measurement and repair not normally considered part of QA
- Important to determine risky components
  - Typically 80% of faults occur in 20% of components
  - Often these components can be identified with appropriate metrics (i.e. size, complexity)

QA Activities:
Defect Reduction



DEFECT CONTAINMENT

Defect Containment: Introduction

- Important for systems where the impact of failures is substantial
- Not all faults can be eliminated (cost, time)
- Rather than comprehensively removing all failures, find ways to isolate the impact of those that remain in the software

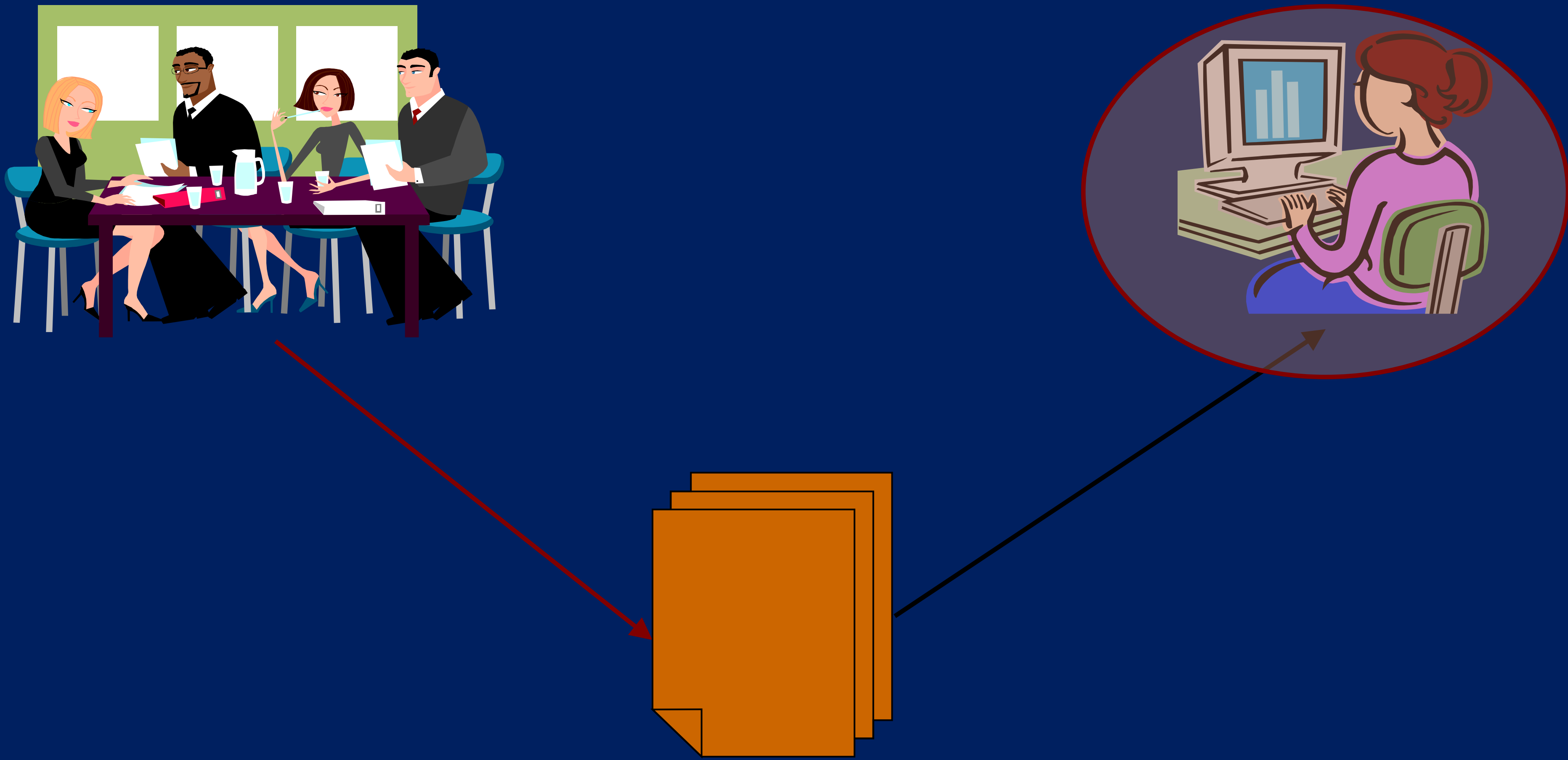
Defect Containment: Fault Tolerance

- Different from manufacturing
- Approaches
  - Recovery Blocks
    - Repeated execution
    - If a failure is discovered, portion of execution is repeated
  - N-Version Programming
    - N versions of the software perform the same functionality
    - Execute in parallel
    - Overall algorithm prevents failures from propagating
- Does not focus on identifying and removing the faults that cause the failures

Defect Containment: Safety Assurance

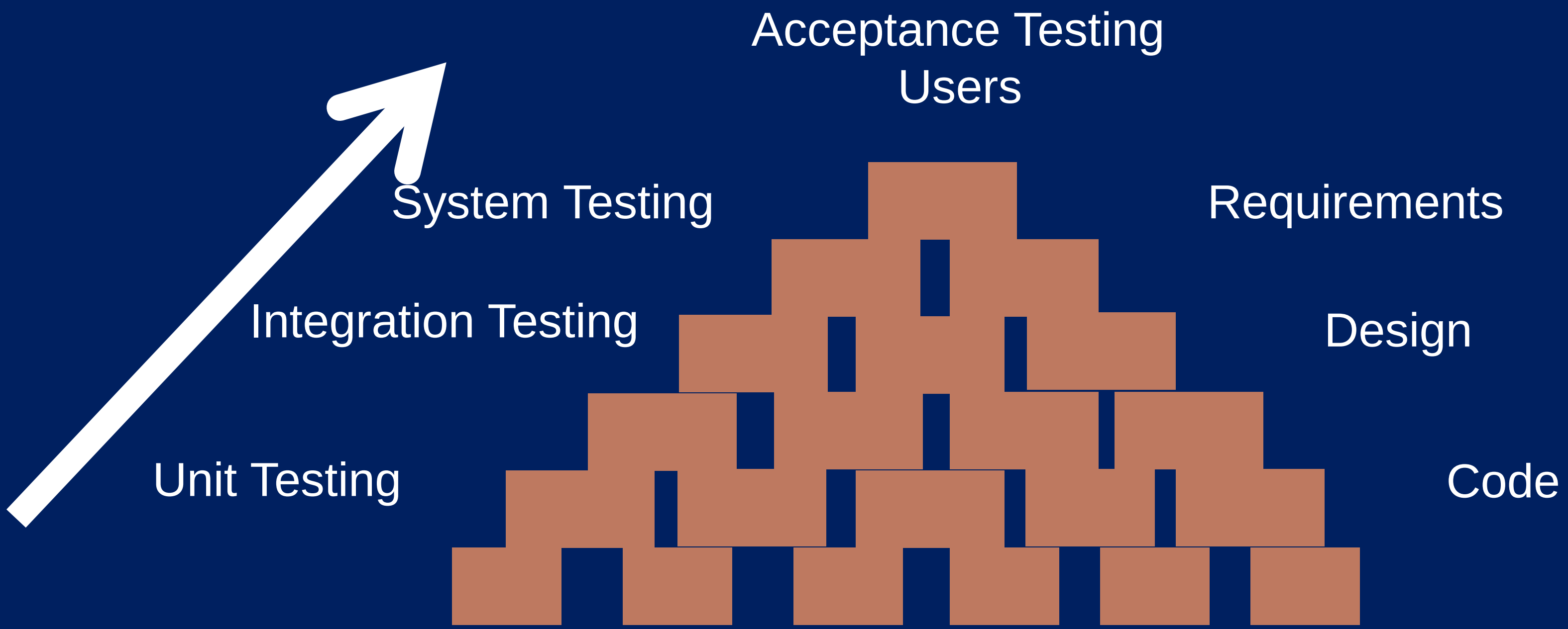
- Safety-critical systems: failure → accident
- Address even low probability failures
- Safety Assurance techniques
  - Hazard elimination – similar to defect prevention but focused on safety critical issues
  - Hazard reduction – similar to fault tolerance
  - Hazard control – reduce severity or impact of failures
  - Damage control – reduce severity of accidents

QA Activities:
Defect Containment



INTRODUCTION TO TESTING

Types of Testing



Unit Testing

- What?
  - Code units
  - Varies with programming language
- Who?
  - Developer
- What is the focus?
  - Correctness of implementation
  - Executable statements, control flow, data flow
- What type of testing techniques do we use?
  - White-box
  - Ad hoc (coverage tools)
  - Input domain partitioning
  - Control Flow / Data Flow

Integration Testing

- What?
  - Collection of components
- Who?
  - Professional testers
- What is the focus?
  - Integrating components to work together to accomplish functionality
  - Each component is a black box
  - Interfaces are tested
- What type of testing techniques do we use?
  - White box – units are the components rather than statements
  - FSM – model control passing between components
- Merged with System Testing?

System Testing

- What?
  - Entire system

- Who?
  - Professional testers

- What is the focus?
  - Overall function from customer's point of view
  - System is black box – external functions tested

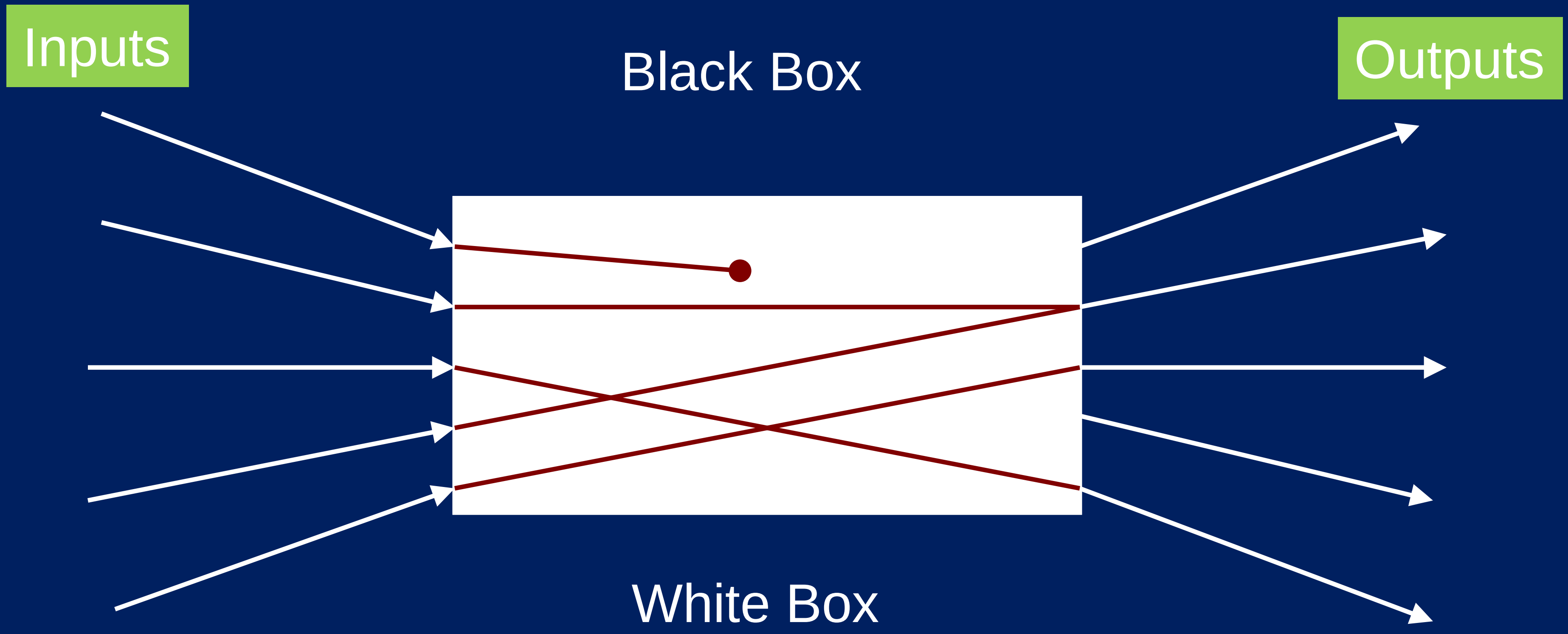
- What type of testing techniques do we use?
  - Function checklist
  - FSM representing system functions
  - Operational profiles

- Embedded systems?

Acceptance Testing

- What?
  - System
- Who?
  - Professional testers
- What is the focus?
  - Is the system reliable enough to release?
  - What support will have to be provided?
  - Not focused on fixing problems
- What type of testing techniques do we use?
  - Usage-based statistical testing

Functional vs. Structural



Functional vs. Structural

- Individual elements
  - Statements
  - Functions
  - Components
- Interactions of elements
  - Sub-system
  - System
- Inputs and outputs – functional

Black Box (Functional) Testing: Overview

- Observes external behavior of software

- What are some approaches we could take?
  - Ad hoc
  - User scenarios
  - Checklist
  - Formal Models

Black Box (Functional) Testing: Process

- Planning
  - Identify external functions to test
  - Derive inputs and outputs
  - Set quality goals
    - Exit criteria
    - Completion of test cases
- Execution
  - Observe behavior
  - Record problems
  - Note execution information to aid in repair
- Analysis
  - Compare results to expectations
  - Testing *oracle* problem
  - Leads to follow-up action to correct the problem

White Box (Structural) Testing: Overview

- Verifies correct implementation of software units

- What are some approaches?
  - Ad hoc
  - Results of functional tests (defects)
  - Coverage

- What knowledge is needed?
  - Programming (general)
  - Tools
  - Specifics of the code

Stopping Criteria: Coverage-Based

- Ensures some item has been covered
- Assumes that higher coverage equals higher quality
- Approaches
  - Checklist
  - Partitions
  - Finite State Machines

Coverage Based Testing: Process

- Define the model
- Check the model elements
- Define the coverage criteria
- Derive the test cases

Test Activities

- Test Planning
- Test Execution
- Analysis and Follow-up

Test Planning

- High level goal: Determine the test strategy
  - Identify the types of testing
  - Set the exit criteria
- Make the following decisions
  - Overall objectives and goals
  - Objects to be tested and focus
- Have to account for personnel

Test Planning:
Test Case Creation

- What is needed?
  - Inputs
  - Outputs
  - Dependencies

- How are they generated?
  - Using inputs and outputs
  - Replay of actual user scenarios

Test Planning:
Test Suite Preparation

- What is a test suite?

- How are they created?

- Expensive → should be maintained for future use

Test Planning: Preparation of Procedure

- Ordering of test cases
  - Dependencies
  - Defect detection
  - Problem diagnosis
  - Natural groupings
- One test case should leave the system ready to execute the next
- Assignment of personnel

Test Execution: Overview

- Major steps:
  - Allocation of time and resources
  - Running tests
  - Analyzing results
- Prevent failed test cases from halting execution
- Environment

Specific Approaches to Testing

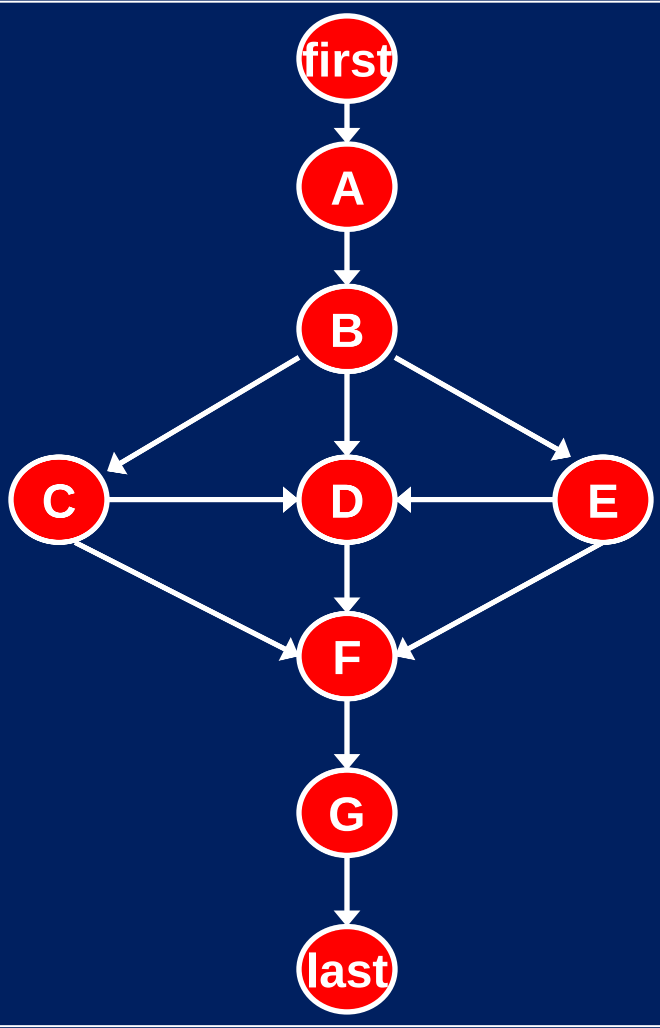
- Control Flow Testing
- Partition-based Testing
- Usage-based Testing
- Data-flow Testing

Systematic Testing

- Drawbacks to *ad hoc* testing
  - Lack of structure
  - Likely to repeat
  - Likely to miss
- One way to structure is to build a checklist

Control Flow Testing: Overview

- Model of the software is a graph
  - Nodes (entry, exit, decision, junction, processing)
  - Links (outlinks, inlinks)
  - Paths
- Use
  - Build graph
  - Define paths
  - Choose inputs
  - Check results



Control Flow Testing: Model Construction

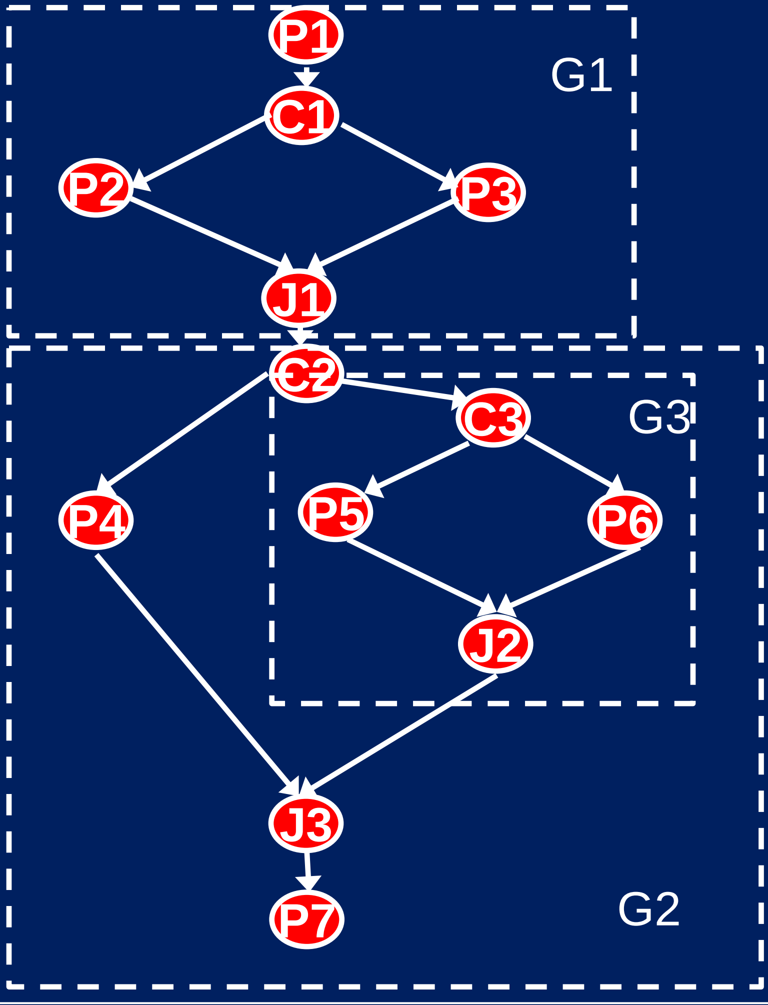
- Using program code, build graph
- Processing nodes
  - Assignment or function calls
- Decision
  - If-then / if-then-else
  - Loops
- Entry/Exit – first and last statements
- Creates a large number of nodes. How can we deal with this?

Control Flow Testing: Model Construction

- Can also be done with black box testing
  - How?
- Elements
  - Processing nodes
    - Some described action
  - Branching nodes
    - Some decision
  - Entry/Exit
    - First and last items

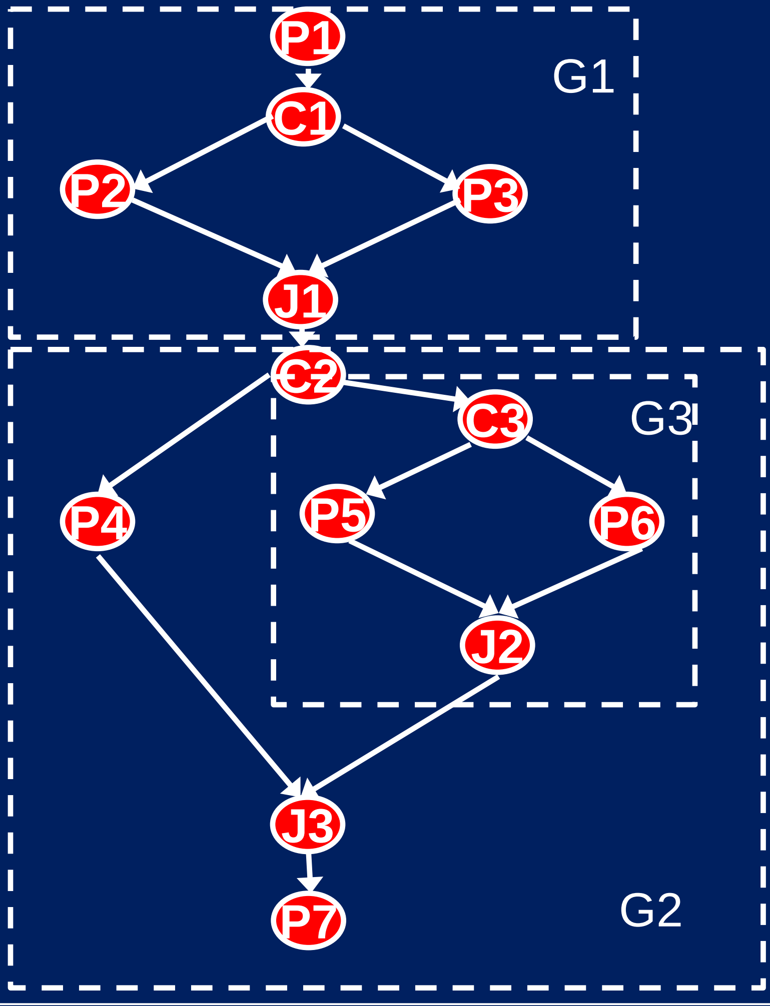
Control Flow Testing:
Path Selection

- Structured CFG
  - Only sequential concatenation and nesting allowed (no go-tos)
  - Unique entry and unique exit
- Can be decomposed into subgraphs – each subgraph is a proper CFG
  - G = G1 ∘ G2 (-,G3)



Control Flow Testing:
Path Selection

- With two graphs (G1, G2); G1 has *M* paths and G2 has *N* paths.
- Sequential combination [G = G1 ◦ G2]
  - *M* x *N* paths
- For nesting [G = G2 (G3)]
  - *M* + *N* -1 paths
- Start with the prime CFGs and work up



Control Flow Testing: Model Construction

```
L1:  input(a,b,c)
L2:  d ← b*b – 4*a*c
L3:  If (d>0) then
L4:        r ← 2
L5:  else_if (d=0)
L6:        r ← 1
L7:  else_if (d<0)
L8:        r ← 0
L9:  output (r)
```

Control Flow Testing: Creating Test Cases

- If each decision is based on an independent variable, then just choose appropriate values
  - Can use the idea of equivalence classes
- If decisions are not independent, some branches may be eliminated as infeasible
- Some decisions may be based on processing between decisions nodes – may be hard to develop test cases

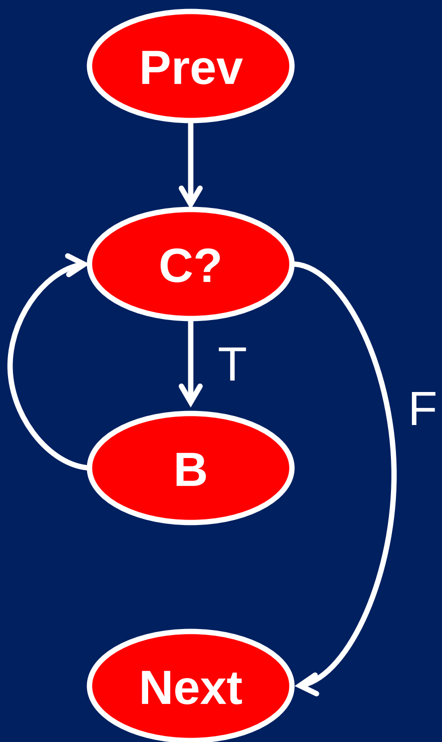
Loops

- Loops complicate the CFT idea. Why?
  - Could result in a lot of test cases
  - May be unpredictable

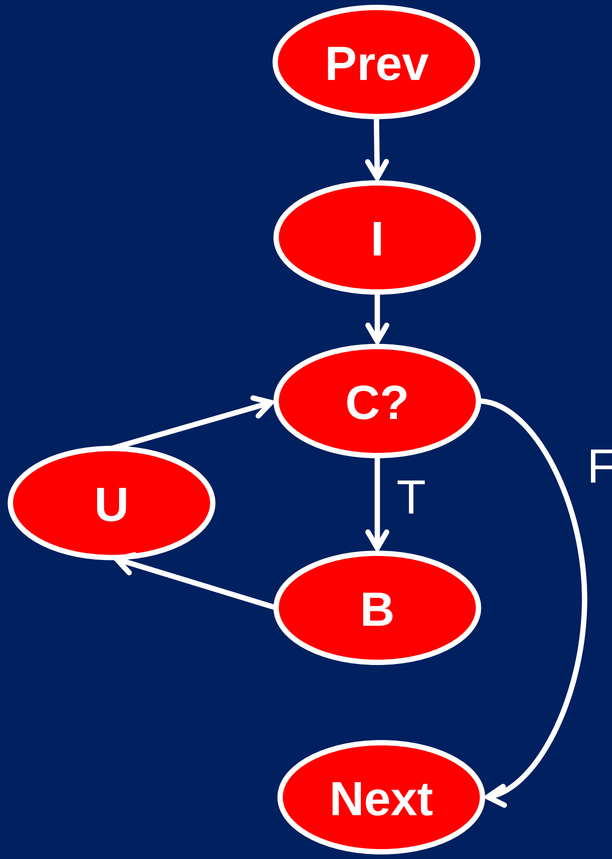
- Parts of a loop:
  - Loop body – accomplishes something; repeated a number of times – represented by a node or a nested CFG
  - Loop control – decision point – represented by a decision node
  - Loop entry/exit – usually have only one – often are the decision point
  - Can be combined through nesting

Loops

While (C) do {B}



For (I; C; U) do {B}



Loop Testing:
Difficulties

- When loops are nested, number of paths quickly grows unmanageable

- Complete path coverage not possible, have to be selective

- Where do most problems occur?
  - Loop boundaries
  - Use equivalence class concepts

- What types of test cases do we need and why?
  - Bypass the loop
  - Once through the loop
  - Twice through the loop
  - Typical cases

Loop Testing: Difficulties

- Concatenation/Nesting of loops
  - 7 test cases for each loop (bypass, once, twice, typical, max-1, max, max+1)
  - 7^n for n concatenated loops
- How can we reduce number of test cases?
  - Test the inner loops with all 7 cases
  - Fix the inner loop with only 1 case and move up the hierarchy (or randomly select one case each time)

Specific Approaches to Testing

- Control Flow Testing
- Partition-based Testing
- Usage-based Testing
- Data-flow Testing

Partition Based Testing

- Benefits
  - Increased coverage
  - Reduced overlap

- Examples:
  - Solve for root of $ax^2 + bx + c = 0$
  - Thermostat

Partition Testing: Theory

- A set S contains a list of unique elements
- Partition of S creates subsets G_1, G_2, … G_n such that
  - Sets are mutually exclusive
  - Sets are collectively exhaustive
- $G_1 .. G_n$ are equivalence classes if created based on some definition of equality
- Properties
  - Symmetric
  - Transitive
  - Reflexive

AUTOMATED TESTING TOOLS

Automated Testing Frameworks

- Enable set of tests to be executed repeatedly

- Family of tools
  - jUnit
  - cUnit
  - ... (xUnit)

- Demo

jUnit

- Plug-in for Eclipse

- Demo of how to use:
  - Create project
  - In ‘src’ folder, create package
  - Create new ‘source folder’ “test”
  - In ‘test’ folder, create package
  - Create new jUnit Test
  - Run test

cUnit

- http://cUnit.sourceforge.net
- Framework to create and execute tests
- Assertions
  - CU_ASSERT
  - CU_ASSERT_TRUE
  - CU_ASSERT_FALSE
  - CU_ASSERT_EQUAL
  - CU_ASSERT_NOT_EQUAL
  - ...

cUnit:
Test Registry

- Repository of test suites and tests
- Using the test registry
  - Create
  - Clean up
- Adding tests
  - Create a test suite
  - Add tests to the test suite

cUnit:
Running Tests

- Can run:
  - All tests
  - Individual suites
  - Individual tests

- Modes
  - Automated – non-automated / XML output
  - Basic – non-automated / stdout output
  - Console – interactive console under user control

cUnit:
Demo

- Install cUnit
  - ./configure
  - make
  - make install
  - Rename library to ‘cunit’ and place in path
  - Link the ‘cunit’ library in when compiling code

- Perform Unit Testing
  - Write tests
  - Execute tests
  - Examine results (XML)

TEST-DRIVEN DEVELOPMENT

Test-Driven Development: Introduction & Background

- Basic idea:
  - Write automated tests
  - Prior to developing functional code
  - Small rapid iterations

- Part of the *agile* software development approach
  - Short iterations
  - Little up-front design
  - Lightweight documentation
  - Refactoring
  - Pair programming

Test-Driven Development: Overview

- Focus on unit tests
  - Traditionally written after code is completed
  - In TDD tests are written before code
- Often require
  - Test drivers
  - Test stubs
- Can be automated or manual
- Can be performed by developers or testers

Test-Driven Development: Motivation

- Programming practice that instructs developers to:
  - Write code only if a test has failed
  - Eliminate duplication
- Test-Driven Development
  - Leads to analysis, design and programming decisions
  - Writing a test is one of the first steps in deciding what a program should do (analysis)

Test-Driven Development: Definition

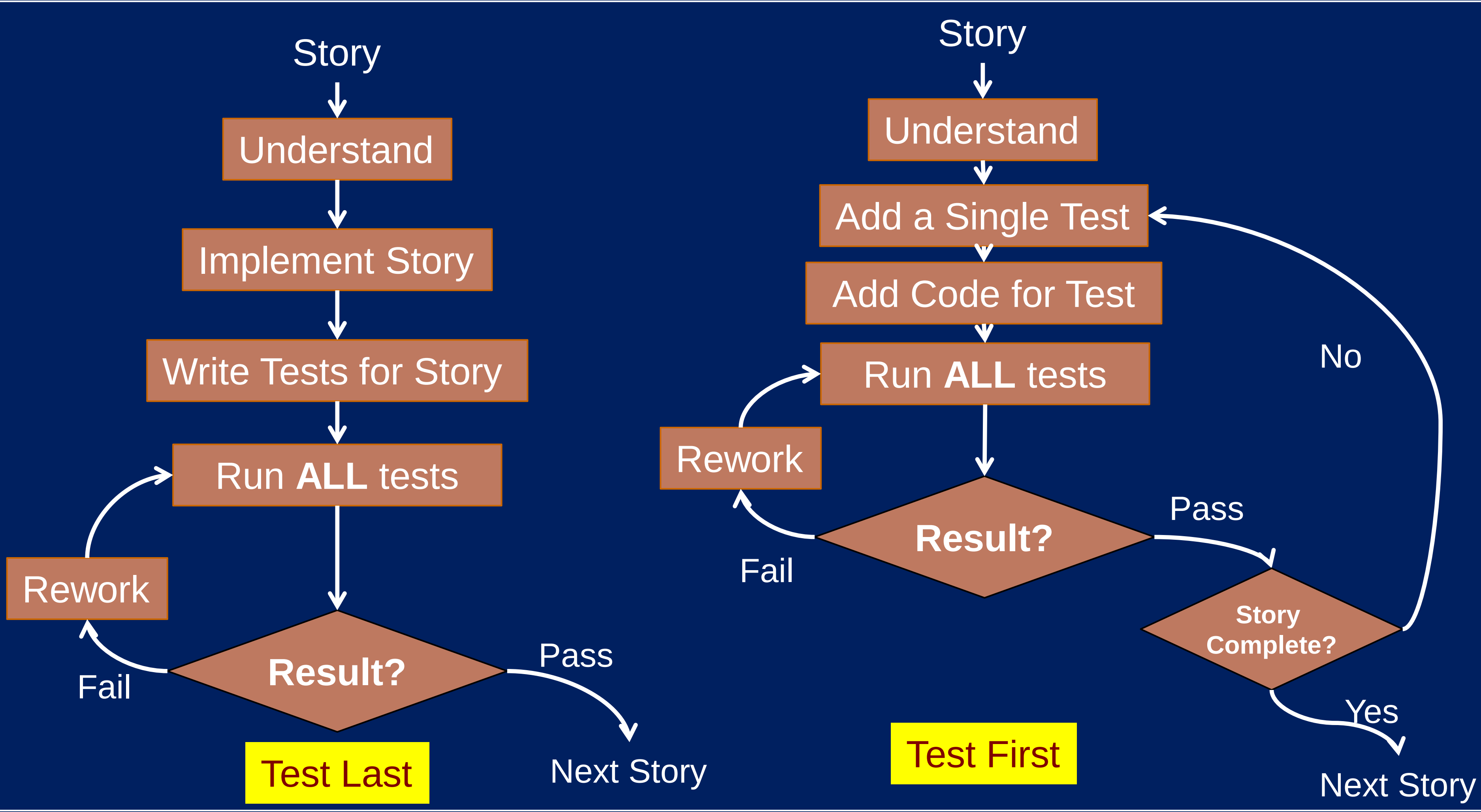
- From the Agile Alliance

  Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code.

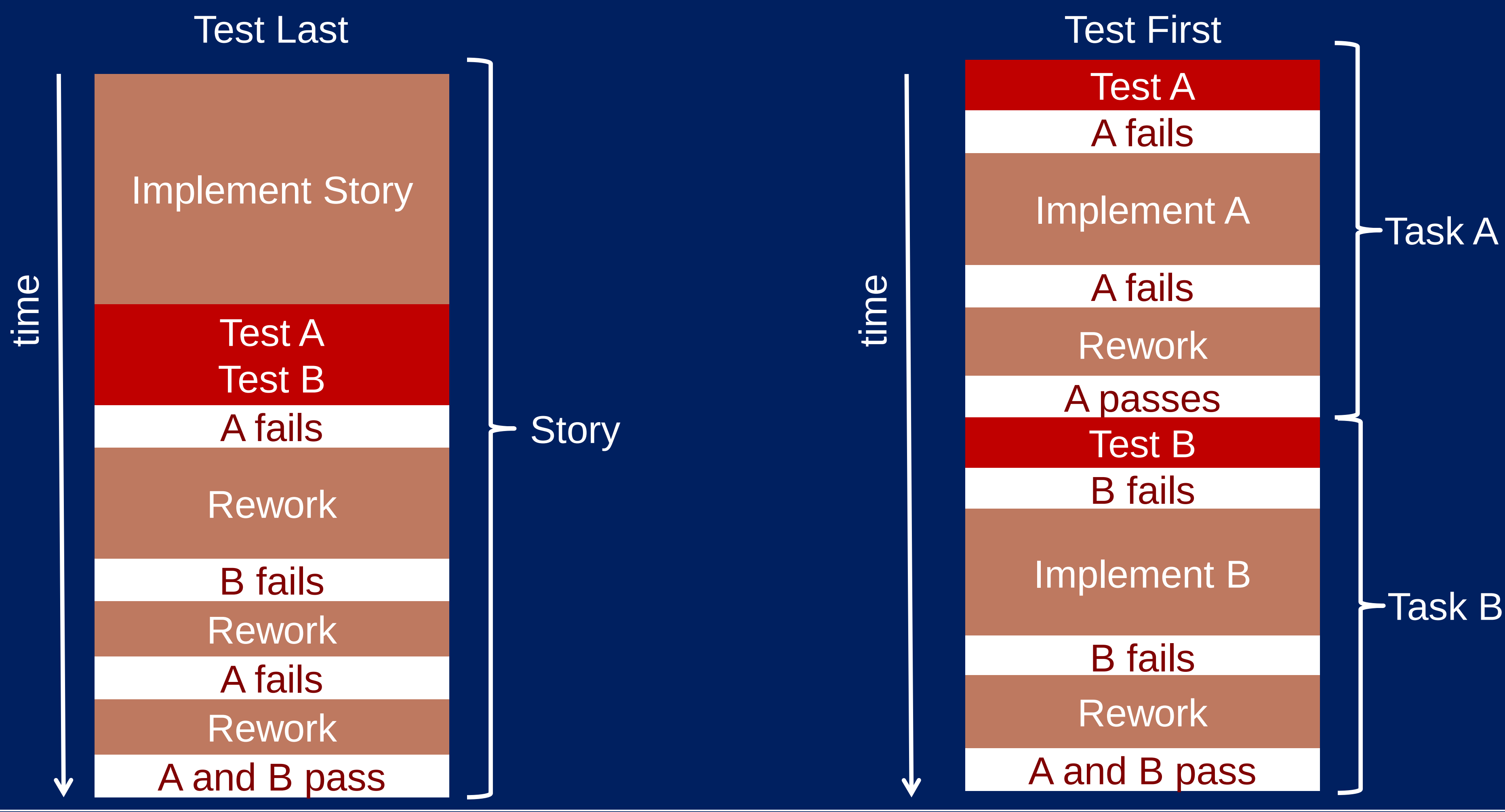
Test-Driven Development: Additional Thoughts

- Refactoring
  - Additional step after coding
  - Code becomes complex
  - Tests still pass, but code is simpler
- Not a software development methodology
- Provides automated test
  - Not thrown away
  - Become part of the development process
  - If a change breaks something that worked before, developer knows immediately

Test-Driven Development: Process



Test-Driven Development: Example Story



Test-Driven Development: Automated Testing

- TDD assumes the presence of an automated testing framework
- Test Harnesses
- xUnit
  - Lets users write tests to initialize, execute, and make assertions about code being tested
  - Tests can serve as documentation

Test-Driven Development: Evaluation

- Performed in Industry and Academia

- Industrial studies
  - 4 studies in small companies
  - Measured defect density
  - Results
    - Programmers using TDD produced code that passed 18% - 50% more tests
    - TDD programmers spent less time debugging
    - TDD *decreased* productivity – but they wrote more test cases

Test-Driven Development: Challenges to Adoption

- Requires discipline by programmers
- TDD is misunderstood – many think it addresses only testing and not design
- Does not fit every situation

Test-Driven Development: Example

- Design a system to perform financial transactions with money that may be in different currencies

- For example –
  - If the exchange rate from My New Currency to US Dollars is 2 to 1, then we can calculate
    - 5 USD + 10 MNC = 10 USD
    - 5 USD + 10 MNC = 20 MNC

Example:
Starting Point

- How do we start?
- Write a list of things we want to test
- List can be any format, just keep it simple
- Example
  - 5 USD + 10 MNC = 10 USD if rate is 2:1
  - 5 USD * 2 = 10 USD

Example:
First Test

- Second item is easier, start there
  - 5 USD * 2 = $10

- First write a test case

Example:
Test Case Discussion

- What benefits does this provide?

- Target class plus some of its interface
  - Design the interface of the Dollar class by thinking about how we would want to use it

- Testable assertion about the state of the Dollar class after a particular sequence of operations

Example:
Next Step

- Test case revealed some issues with the Dollar class that must be cleaned up
  - The amount is represented as an integer, making it difficult to handle things like 1.5 USD; how do we handle rounding of fractions?
  - Dollar.amount is public; violates encapsulation
  - Side effects?
    - We first declared our variable as “five”, but after we performed the multiplication, it equals “ten”
- Update Test List

Example:
First Version of Dollar Class

- Our test will not compile
  - What compile errors will we encounter?
  - Fix compile errors
  - Create skeleton of Dollar class

- Now our test compiles and fails

Example:
Too Slow?

- Process
  - Do the simplest thing to get the test to compile
  - Now do the simplest thing to get the test to pass
- Is this process too slow?
  - Yes
    - As you get familiar with the TDD lifecycle you will gain confidence and make bigger steps
  - No
    - Small simple steps help avoid mistakes
    - Beginning programmers try to code too much before compiling
    - Spend the rest of their time debugging!

Example:
Make the Test Case Pass

- First do simplest thing to get test case to pass

- The test now passes

- Now, we need to refactor to remove duplication
  - Where is the duplication?
  - Hint: Its between the Dollar class and the test case

Example: Refactoring

- To remove the duplication of the test data and the hard-wired code of the times method, we think the following
  - *I am trying to get at 10 at the end of my test case. I've been given a 5 in the constructor and a 2 was passed as a parameter to the times method*
- Let's connect things

Example:
First Version of Dollar Class

- Refactor Dollar class
- Now our test compiles and passes, and we didn’t have to cheat!
- One TDD loop complete
  - Update testing list
  - Move on to next item

Example:
Second Loop

- Address the “Dollar Side-Effects” item

- Next test case
  - When we called the times operation on our variable, “five” was pointing at an object whose amount equaled “ten”; not good
    - The times operation had a side effect which was to change the value of a previously created “value object”
    - This doesn’t make sense, you can’t change a $5 bill into a $10 bill; the $5 bill remains the same throughout transactions
  - Rewrite test case

Example: Test Fails

- Won’t compile
  - How do we fix this problem?

- Change the signature of the times method; previously it returned void and now it needs to return Dollar

- The test compiles but still fails – progress
  - How do we fix this problem?

Example:
Test Passes

- To make the test pass, we need to return a new Dollar object whose amount equals the result of the multiplication
- Test passes, cross “Dollar Side-Effects” off of the testing list.
- No need to refactor here
- Move on to next test item

C Example

- Write a program to calculate bowling score using TDD

- Description
  - 10 frames, in each frame 2 balls to knock down 10 pins
  - Spare – all 10 pins with two balls
    - Bonus – get to add number of pins on next ball
  - Strike – all 10 pins with one ball
    - Bonus – get to add number of pins on next 2 balls
  - 10^{th} frame – spare or strike entitles additional ball rolls (no more than 3 total in the frame)

C Example:
Test Cases

- Test a “gutter game” – all 0s
- Test “all ones” – hitting 1 pin with all balls
- Test “one spare” – 1 spare, plus bonus, the rest 0s
- Test “one strike” – 1 strike, plus bonus, the rest 0s
- Test “perfect game” – all strikes (12)

HANDS-ON TIME

Materials

- Updated slides and handouts available on the web

- http://carver.cs.ua.edu/SC12_Tutorial/

References

- Jansen, D. and Saiedian, H. “Test-Driven Development: Concepts, Taxonomy, and Future Direction.” *Computer*. Sept. 2005. p. 43-50
- Erdogmus, H., Morisio, M., and Torchiano, M. “On the Effectiveness of the Test-First Approach to Programming.” *IEEE Transactions on Software Engineering*. 31(5): 226-237. March 2006.
- Currency example taken from
  - Kenneth Anderson, Univ. of Colorado, Boulder
- Bowling example taken from
  - http://www.slideshare.net/amritayan/test-driven-development-in-c