SOEN 6431 Software Maintenance and Program Comprehension

Week 6:
Impact Analysis and Regression Testing

Characteristics of change

- Software requirements change to accommodate user expectations, new environments etc.
  - potential high impact
  - invalidate assumptions
  - may change system design
- Long development cycles
  - few people stay with the project
  - initial requirements or their rationale forgotten
- Large teams, multiple version/ configurations

General Idea Change Impact

The maintenance process starts by performing impact analysis.

Impact analysis basically refers to identifying the components that are impacted by the Change Request (CR).

Impact of the changes are analyzed for the following reasons:

- to **estimate the cost** of executing the change request.
- to **determine** whether some critical **portions** of the system are going to be **impacted** due to the requested change.
- to **record the history** of change related information for future evaluation of changes.
- to understand **how** items of **change** are **related** to the structure of the software.
- to determine the **portions** of the software that need to be subjected to **regression testing** after a change is effected.

General Idea - Traceability

Implementation of change requests impact all kinds of **artifacts**, including the source code, requirements, design documentation, and test scenarios.

Therefore, impact analysis traceability information can be used in performing impact analysis.

Gotel and Finkkelstein define traceability as the **ability to describe and follow** the life of an artifact in both the forward and backward directions.

T**raceability** helps software developers understand the relationships among all the software artifacts in a project.

General Idea – Ripple Effect Analysis

A topic related to impact analysis is ripple effect analysis.

Ripple effect means that a modification to a single variable may require several parts of the software system to be modified.

Analysis of ripple effect reveals what and where changes are occurring.

- (i) between successive versions of the same system, measurement of ripple effect will tell us how the software's complexity has changed.
- (ii) measurement of ripple effect can tell us how the software's complexity has changed because of the addition of the new module.

Impact analysis: identifying the potential consequences (impacts) of a change, or estimating what needs to be modified to accomplish a change

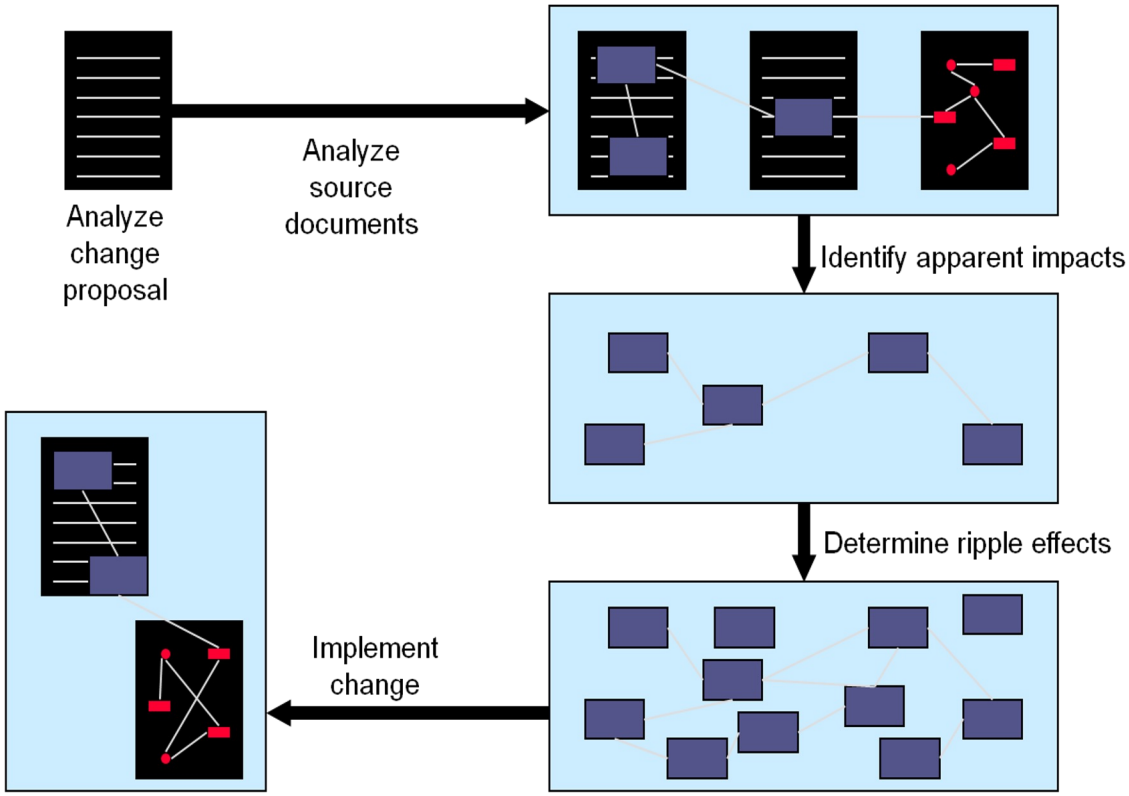
A ***side effect*** is an error or other undesirable behavior that occurs as the result of a change

Ripple effect: Small change to a system affects many other parts of it

- Involves transitive closure
- Ripple effect analysis emphasis changes of source code
- Other ripple effect types:
  - Requirements: operation of the system is influenced
  - Interface: changes in using module needed
  - Environment: new programming language or programming environment
  - Management and logistics: Impacts resources available, schedule, contracts, training, deployment

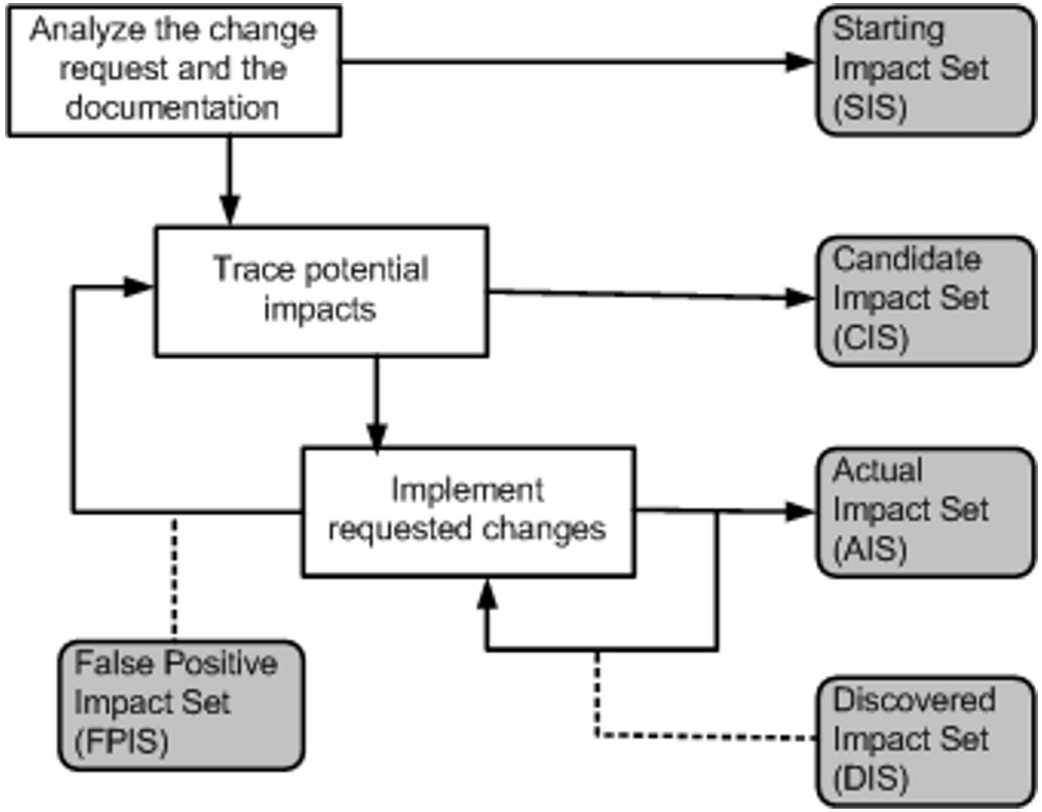
Impact-analysis process

- Input: Set of software life-cycle objects (SLO)
- Analyze SLOs with respect to the change
- Output: List of items that should be addressed during the change process



A more detailed look:
Impact Analysis Process

2/13/2024



Impact Analysis Process

- **Starting Impact Set (SIS):** The initial set of objects (or components) presumed to be impacted by a software CR is called SIS.
- **Candidate Impact Set (CIS):** The set of objects (or components) estimated to be impacted according to a certain impact analysis approach is called CIS.
- **Discovered Impact Set (DIS):** DIS is defined as the set o new objects (or components), not contained in CIS, discovered to be impacted while implementing a CR.
- **Actual Impact Set (AIS):** The set of objects (or components) actually changed as a result of performin a CR is denoted by AIS.
- **False Positive Impact Set (FPIS):** FPIS is defined as the set of objects (or components) estimated to be impacted by an implementation of a CR but not actuall impacted by the CR. Precisely, FPIS = (CIS U DIS) \ AIS.
  - where U denotes set union and \ denotes set difference.
- In the process of impact analysis it is important to minimize the differences between AIS and CIS, by eliminating false positives and identifying true impacts

Major areas

Dependency analysis

- examining detailed dependency relationships among program entities
  (often: source code entities)
- intra-product dependencies
- often hard-coded into tools

Traceability analysis

- relationship among all types of SLOs
- inter-product dependencies
- often: not very detailed

Identifying the Starting Impact Set (SIS)

There are several methods to identify concepts, or features, in source code.

The "grep" pattern matching utility available on most Unix systems and similar search tools are commonly used by programmers.

However, grep has some deficiencies:

- it is based on the correspondence between the name for the concept assigned by the programmer and an identifier in the code.

The technique often fails when the concepts are hidden in the source code, or when the programmer fails to guess the program identifiers.

Identifying the Starting Impact Set (SIS)

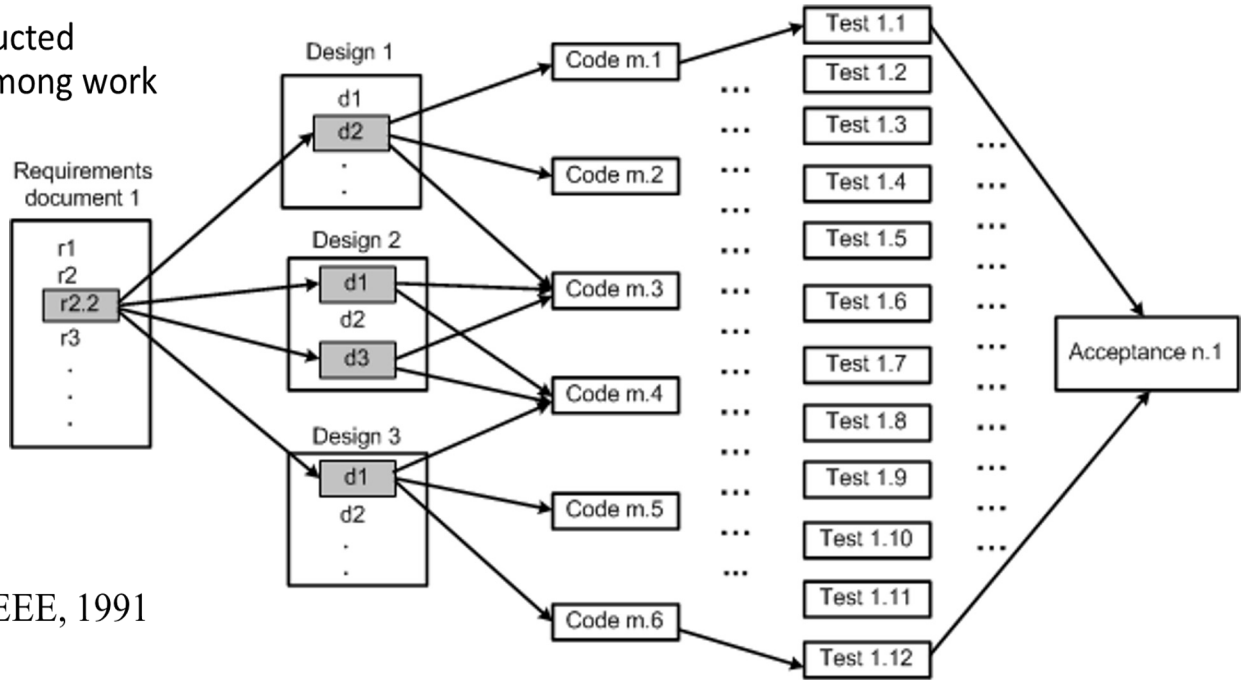
The **software reconnaissance** methodology proposed by Wilde and Scully is based on the idea that some programming concepts are selectable, because their execution depends on a specific input sequence.

Selectable program concepts are known as features. By executing a program twice, one can often find the source code implementing the features:

- (i) execute the program once with a feature and once without the feature.
- (ii) mark portions of the source code that were executed the first time but not the second time.
- (iii) the marked code are likely to be in or close to the code implementing the particular feature.

Analysis of Traceability Graphs

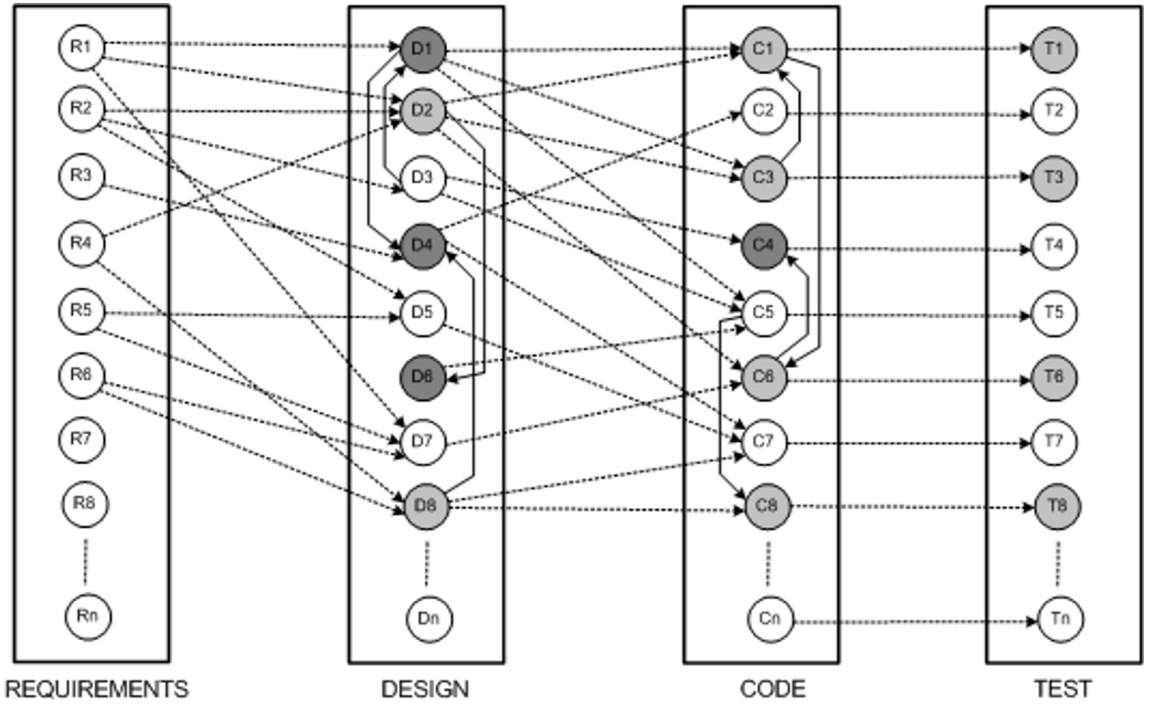
- The graph shows the horizontal traceability of the system.
- The graph that is so constructed reveals the relationships among work products.



Traceability in software work products ©IEEE, 1991

Analysis of Traceability Graphs

- The graph has four categories of nodes: requirements, design, code, and test.
- The edges within a silo represent vertical traceability for the kind of work product represented by the silo.



Traceability in software work products ©IEEE, 1991

Video

Identify the Candidate Impact Set (CIS)

A CIS is identified in the next step of the impact analysis process.

The SIS is augmented with software lifecycle objects (SLOs) that are likely to change because of changes in the elements of the SIS.

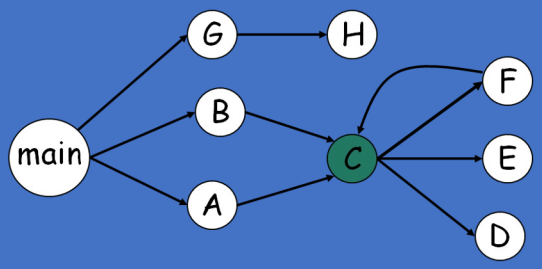
Changes in one part of the software system may have direct impacts or indirect impacts on other parts.

Both direct impact and indirect impact are explained in the following.

Direct impact: A direct impact relation exists between two entities, if the two entities are related by a fan-in and/or fan-out relation.

Indirect impact: If an entity A directly impacts another entity B and B directly impacts a third entity C, then we can say that A indirectly impacts C.

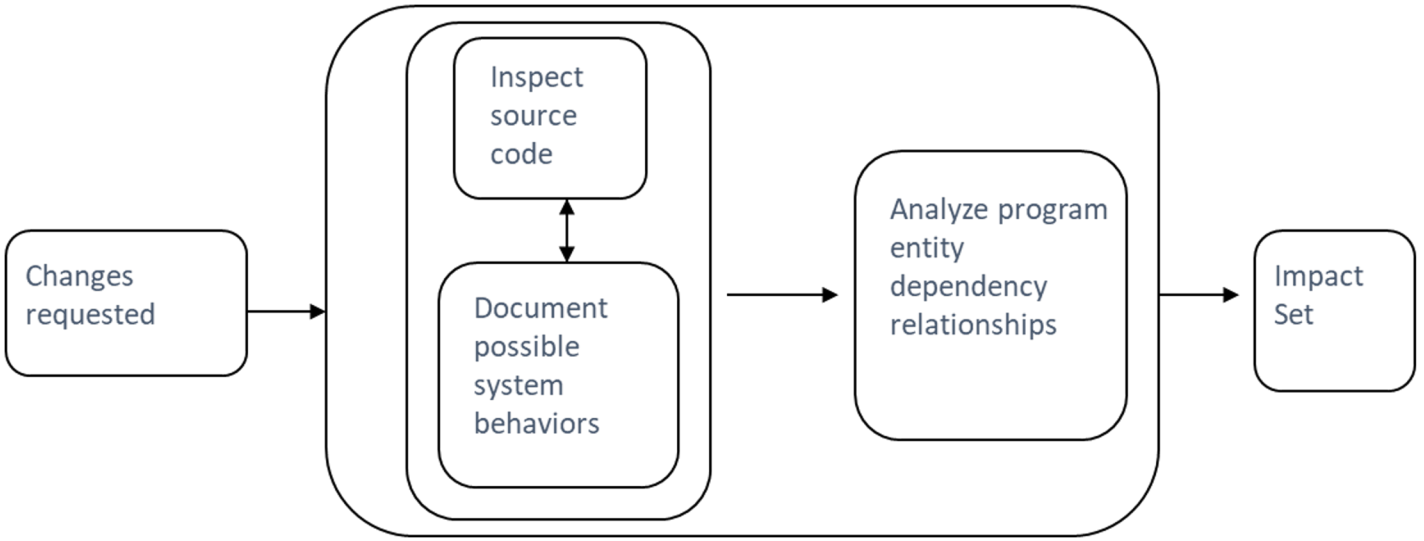
Call-Graph Based Approach



Objective:

- *If I change function C, what other functions could be affected (impacted)?*
  - Depends on analysis of source code
  - Based on **assumptions** of all possible software runtime behaviors
  - Results can include most of the software system in the impact set
- The call graph-based approach to impact analysis suffers from **many disadvantages** as follows:
  - Impact analysis based on call graphs can produce an imprecise impact set. For example (Figure) one cannot determine the conditions that cause impacts of changes to propagate from M to other procedures.
  - Impact propagations due to procedure returns are not captured in the call graph-based technique. Suppose that, in Figure, *E* is modified and control returns to C.. Now, following the return to C, it cannot be inferred whether impacts of changing E propagates into none, both, A or B.

Static Impact Analysis



Perform analysis on source code

- Call graph traversals
- Slicing

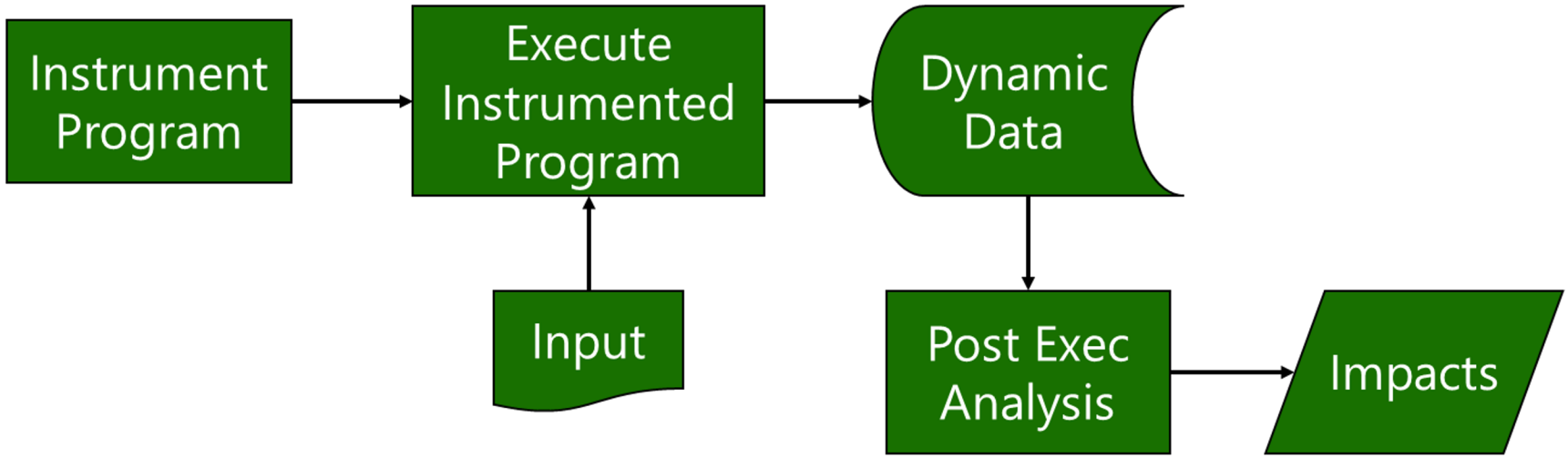
E.g., Static Forward slicing !

Obtain conservative results

- Accounts for all possible inputs and behaviors
- Can give very large impact sets

Dynamic Impact Analysis

- *Not as conservative -- results depend on input*
- *Give impacts related to program use*

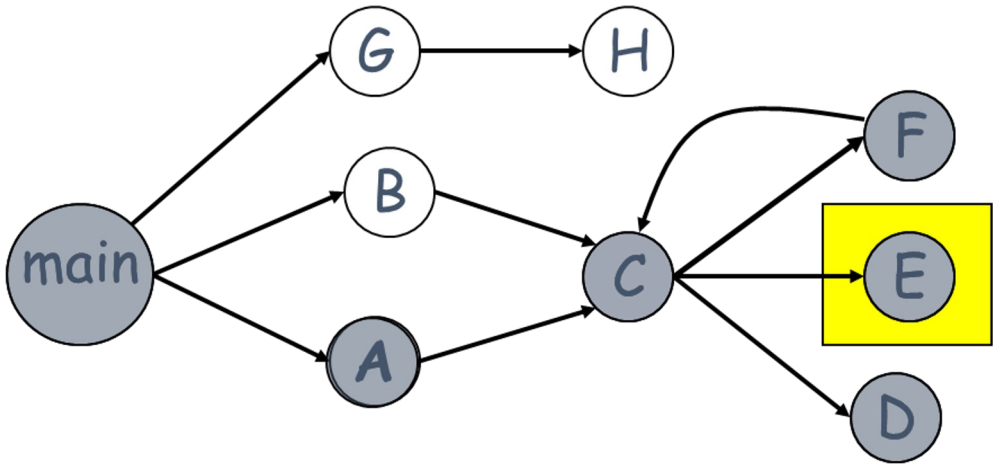


Example

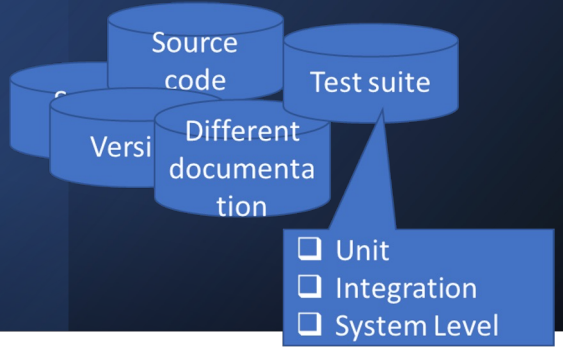
- Let us consider an execution trace as shown in below.

  M B r A C D r E r r r r x. Where **r** and **x** represent function returns and program exits.

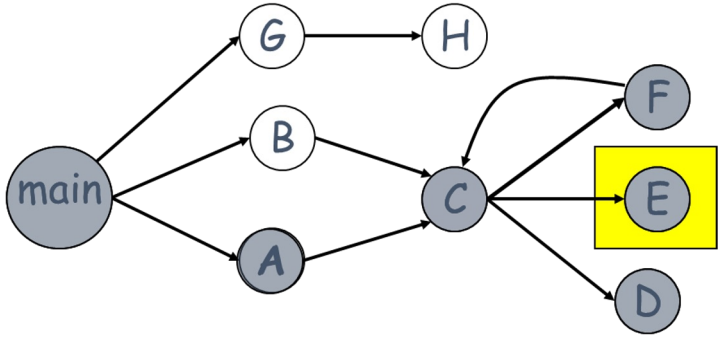
- The impact of the modification of M with respect to the given trace is computed by forward searching in the trace to find:
  - procedures that are indirectly or directly invoked by E; and
  - procedures that are invoked after E terminates.
- One can identify the procedures into which E returns by performing backward search in the given trace.
- For example, in the given trace, E does not invoke other entities, but it returns into M, A, and C.
- Due to a modification in E, the set of potentially impacted procedures is {M,A,C, E}.



Review



1. Identify starting point of the change => E
2. Identify impacted code and other artifacts (static, dynamic)
3. Implement the actual change => Module E, C, F…?
4. Re-test – what should we be re-testing and what would be the purpose of re-testing?



Software Regression Testing

- What is Software Regression Testing?
- Basic Software Regression Problems
- Software Regression Testing Process
- Regression Strategies for Traditional Software
- Basic Solutions to Software Regression Problems
- Regression Strategies for Object-Oriented Software

What is Software Regression Testing?

- **What is Software Regression Testing?**
  - Testing activities occur **after** software changes.
  - Regression testing **usually** refers to testing activities during software maintenance phase.

- **Major testing objectives:**
  - Retest changed components (or parts)
  - Check the affected parts (or components)

- **Regression testing at different levels:**
  - Regression testing at the unit level
  - Re-integration
  - Regression testing at the function level
  - Regression testing at the system level

What is Software Regression Testing?

Who performs software Regression:

- Developers - regression testing at the unit level or integration
- Test engineers - regression testing at the function level
- QA and test engineers - regression testing at the system level

What do you need to perform software regression testing?

- Software change information (change notes).
- Updated software REQ and Design specifications, and user manuals.
- Software regression testing process and strategy.
- Software regression testing methods and criteria.

What Is Regression Testing?

A system under test (SUT) is said to *regress* if

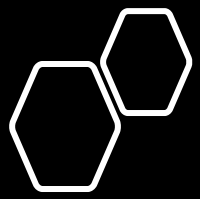
- A **modified** component **fails**, or
- **A new component**, **when used with unchanged components**, causes failures in the unchanged components by generating side effects or feature interactions.

Baseline version

- The version of a component (system) that has passed a test suite.

Delta version

- A changed version that has not passed a regression test



What Is Regression Testing? - cont

- **Delta build**
  - An executable configuration of the SUT that contains all the delta and baseline components
- **Regression test case**
  - A test case that the baseline has passed and which is expected to pass when rerun on a delta build
- **Regression test suite**
  - A set of regression test cases
- **Regression fault**
  - A fault revealed by a test case that has previously passed but no long passes

Challenges in Software Regression Testing

Major challenges in software regression testing:

- How to **identify software changes** in a systematic way?

  - REQ. specification changes
  - Design specification changes
  - Implementation (or source code) changes
  - User manual changes
  - Environment or technology changes
  - Test changes

- How to **identify** software **change impacts** in a systematic way?

  - REQ impacts
  - Design impacts
  - Implementation impacts
  - User impacts
  - Test impacts

- How to **minimize re-testing** efforts, and achieve the adequate testing coverage?

When Is Regression Testing Done?

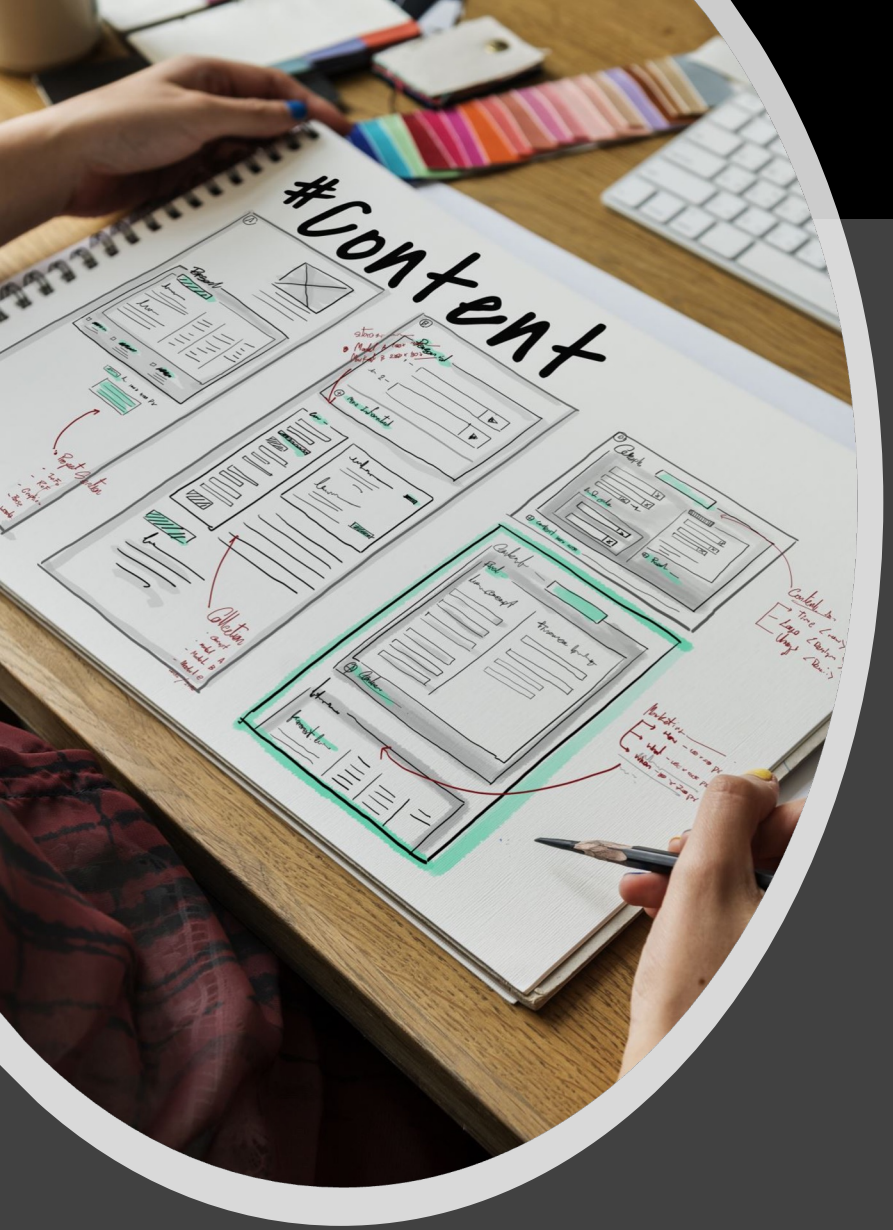
During software maintenance

- *Corrective maintenance* – changes made to correct a system after a failure has been observed (usually after general release)
- *Adaptive maintenance* – changes made to achieve continuing compatibility with the target environment or other systems.
- *Perfective maintenance* – changes designed to improve or add capabilities.
- *Preventive maintenance* – changes made to increase robustness, maintainability, portability, and other features.

Software Regression Process

Software Regression Process:

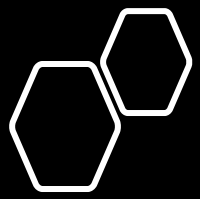
- Step #1: Software Change Analysis
  Understand and analyze various software changes.
- Step #2: Software Change Impact Analysis
  Understand and analyze software change impacts
- Step #3: Define Regression Test Strategy and Criteria
- Step #4: Define, select, and reuse test cases to form a regression test suite
- Step #5: Perform re-testing at the different levels.
  - re-testing at the unit level
  - re-testing at integration level
  - re-testing at the function level
  - re-testing at the system level
- Step #6: Report and analyze regression test results



Software Regression Strategy

What is a software Regression strategy?

- Software test strategy provides the basic strategy and guidelines to test engineers to perform software regression testing activities in a rational way.
- Software Regression strategy usually refers to
  a rational way to define regression testing scope, coverage criteria, re-testing sequence (or order) and re-integration orders.
- Software regression test models are needed to support the definition of software regression test strategy, test cases, and coverage criteria.



How Is Regression Testing Done?

- **Remove broken test cases** from the original test suite;
  - A broken test case is one that cannot run on the delta build.
- Choose a **full** regression test suite **or** a **reduced** regression test suite
- **Set up** the test configuration;
- **Run** the (reduced) **regression** test suite;
- Take appropriate **action** for no pass tests;

Automation: Required Capabilities

Version control

The test suites and the SUT must be under configuration management control

Modular structure

Baseline test suites grow with successive increments and releases.

Organizing test suites into small modules facilitates selection and maintenance of tests.

Compare baseline and delta results

Automation: Required Capabilities – cont.

Smart comparator

- The ability to ignore some output fields (e.g. time stamps) while comparing expected and actual results.

Shuffling

- Automatically shuffles the order of the test suite , in order to reveal sequence-sensitive bugs

Built-in test

- Test drivers packaged with a component can reduce the cost of test maintenance

Automated Testing

Test Suite Maintenance

A regression suite can become large after several release cycles.

- Additions and changes to the SUT call for corresponding changes in its test suite.

Test suite decay

- Some tests simply fail to run due to code change
- Tests may become obsolete as requirements change / evolve
  - If 1<=x<=1000 has changed to 1<=x<=10000, the baseline tests still pass, but do not probe the boundary
- Redundant test cases can easily occur.
- Testers are often reluctant to remove them because of the risk of throwing away a test that would reveal some current or future fault

Test Suite Maintenance - cont

How to avoid decay:

- Pay close attention to decreases in coverage. Use missing coverage as a pointer to missing clues.
- Insist on tests with variety and complexity.
- Discard unnecessary tests

Reducing a Test Suite- Unsafe Reduction

Unsafe reduction

- Can omit test cases that might reveal a regression bug

Systematic sampling

- Select every n^{th} baseline test case automatically.
- A simple way to control the scope of testing.

Random sampling

- Select baseline test cases automatically so that, given n, each baseline case has a 1/n chance of being selected.
- Almost produces a different test suite each time.

Coverage-based filtering

- Coverage analyzers can identify tests that exercise the same components and offer this ability as a feature for reducing regression test suites.

Retest Risky Use Cases

Choose baseline tests to rerun by risk heuristics

- Skip enough non-critical, low priority, or highly stable use cases to meet a deadline or budget constraint.

Risk criteria

- *Suspicious use cases* – use cases that depend on components, middleware, or other resources that
  - are individually unstable or unproven;
  - have not been shown to work together before or are unstable;
  - implement complex business rules;
  - have a complex implementation (e.g. 10 times the code size of all other components), or
  - were subject to high churn during development.

Retest Risky Use Cases - cont

Risk criteria - cont

- *Critical use cases*: select tests for use cases that are necessary for safe, effective operation.
  - For each case, try to identify the worst-case effects of failure.

Consequences

- Moderate risk of missing a regression fault and generally a low cost of analysis and setup.

Known Use

- Reducing test suites by focusing on risks and customer priorities is a common practice.

Retest Risky Use Cases - Consequences

Inclusiveness

- **Unsafe** because of no analysis of dependencies

Precision

- Some tests that could be skipped may be repeated

Efficiency

- The time and cost are constrained.
- Analysis can be done without code analyzers or in-depth technical knowledge of the SUT because selection is based on use cases.

Generality

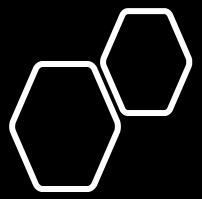
- Can be applied in nearly and circumstance and at any scope

Retest by Profile

Use a budget-constraint operational profile

Test strategy

- Determine how many of the test cases can be run?
  - *Average time needed to run each test case*
  - *Expected bug rate*
    - Hopefully reduced, compared with that of the baseline testing
  - *Average debugging time*
    - Probably will not be reduced
- Determine how many cases will be used for each use case
  - Distributed in proportion to the relative frequency of each use case - defined in profile.



How Many Test Cases Can Be Run? (Example)

Example: For an ATM system

- Baseline test suite has 20,000 test cases.
- 6000 minutes (100 hours) available.
- An average of 5 min to run one test case.
- A test reveals a bug 0.5 percent of the time.
- A bug fix requires 200 minutes.

How many test cases can we re-run within the given constraints?

- $5x + (0.005x * 205) = 6000$
- $5x + 1.025x = 6000$
- $6.025x = 6000$
- $x \cong 1000$

Allocation of Regression Test Time

By use case frequency

Use case	Probability	#Tests
Cash withdrawal	0.53	2650
Checking deposit	0.15	750
Savings deposit	0.14	700
Funds transfer	0.08	400
Balance inquiry	0.06	300
Restock	0.02	100
Collect deposit	0.02	100
Total	1	5000

Retest by Profile – Consequence

Moderate risk of missing a regression fault

- ***Inclusiveness***
  - Unsafe
- ***Precision***
  - Some tests that could be skipped may be repeated
- ***Efficiency***
  - Time and cost to run the tests are constrained.
  - Done without code analyzers or in-depth knowledge of the SUT
- ***Generality***
  - Best suited to system scope testing of applications for which an accurate operational profile can be developed

Reducing a Test Suite

Unsafe Reduction

Retest by profile

- use a budget-constrained operational profile
- e.g., all use cases are exercised and in the same relative frequency as the full test suite, but with fewer test per use case.

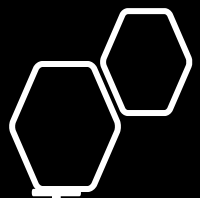
Retest risky use cases

- use risk-based heuristic to select a partial suite
- e.g., suspicion about which new components are likely to cause trouble or which components depend on some others.

- …

Retest Within Firewall

- Use code dependency analysis to select a partial test suite.
- A firewall is a set of component whose test cases will be included in a regression test
  - Identified by analysis of changes to each component and its dependencies on other components
    - Contract change
    - Implementation change
- **Low risk** of missing a regression fault.
- The direct cost of analysis and setup can be low if the analysis is automated



Traditional Software Regression Strategy Based on The Firewall Concept

A Module-Based Firewall Concept for Software Regression Testing:

A module firewall refers to a changed software module and a closure of all possible affected modules and related integration links in a program based on a control-flow graph.

With this firewall concept, we can reduce the software regression testing to a smaller scope All modules and related integration links inside the firewall.

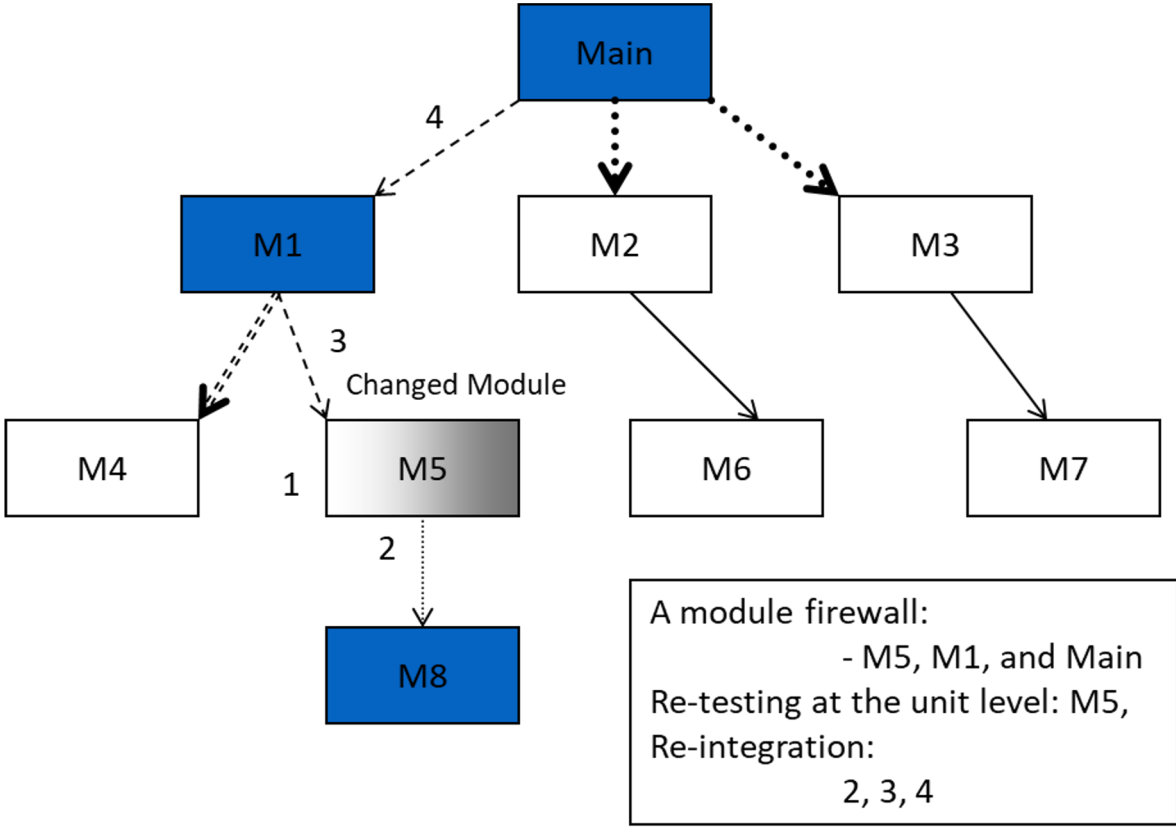
This implies that:

- ❑ re-test of the changed module and its affected modules
- ❑ re-integration for all of related integration links
- ❑ Similarly, we can come out different kinds of firewalls based on various test models.

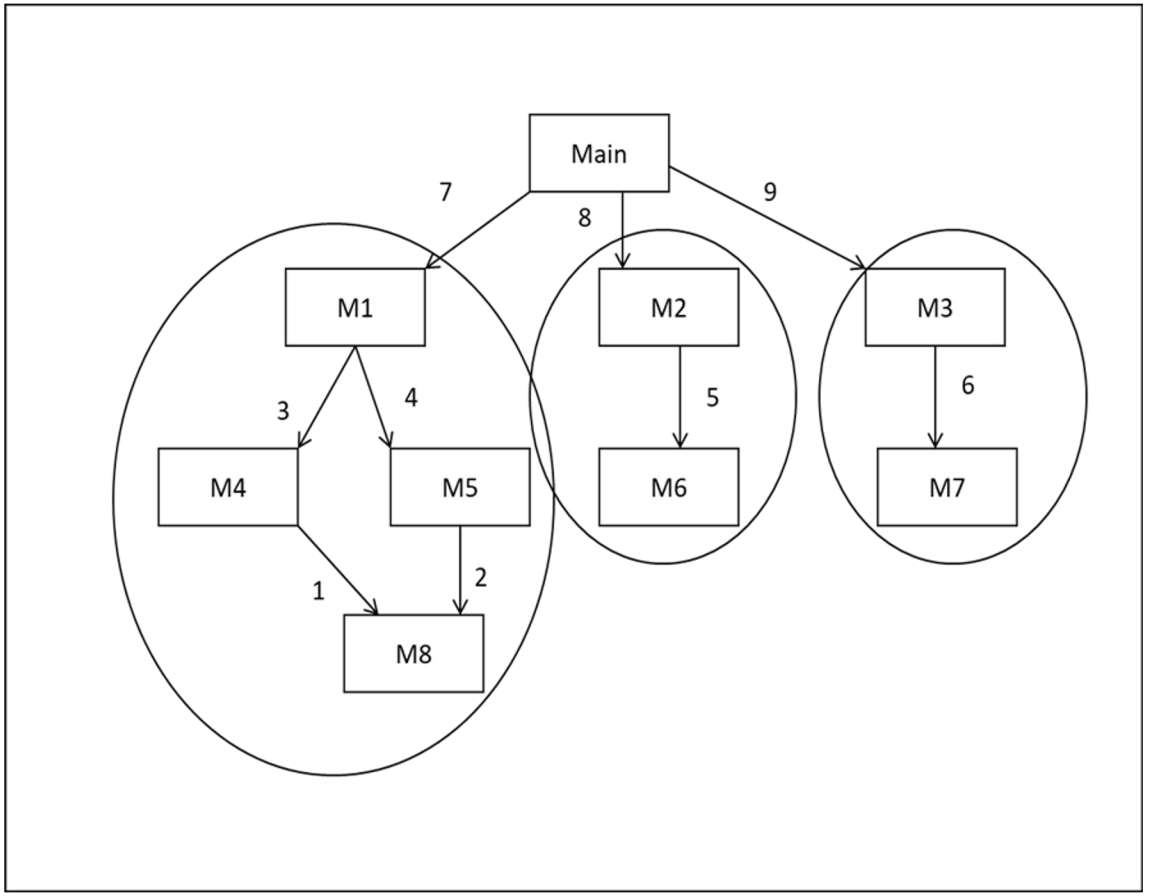
  data firewall, function firewall

  class firewall, state/transaction firewall

A Changed Module Firewall For Regression Testing



Example of Firewalls within a larger system



The Class Firewall Concept

A class firewall concept in OO Software is very useful for OO regression testing.

What is a class firewall?

- A class firewall **is a closure set** of all classes that are directly or indirectly dependent on the changed class in an OO program.
- The class firewall provides **the safe scope** of regression testing for an OO software after changing a class(es).

Class Firewall Application:

- With this class firewall concept, we can narrow down the class regression testing scope, including unit re-testing, and re-integration.
- Based on the class firewall and changed information, we can select, define, and reuse class test cases for regression testing.

Retest Within Firewall - Consequences

- ***Inclusiveness***
  - Safe. Implicit dependencies are considered
- ***Precision***
  - A few tests that could be skipped may be included
- ***Efficiency***
  - In the worst case, the selection computation is on the order of *the size of the baseline test suite* times *the number of components* in the larger of the baseline or delta component.
  - It does not guarantee any particular time and cost deduction – constraints may not be met
- ***Generality***
  - Applicable at cluster scope. Requires the programmer to have skills in code analysis and related tool.

Providing Test Input

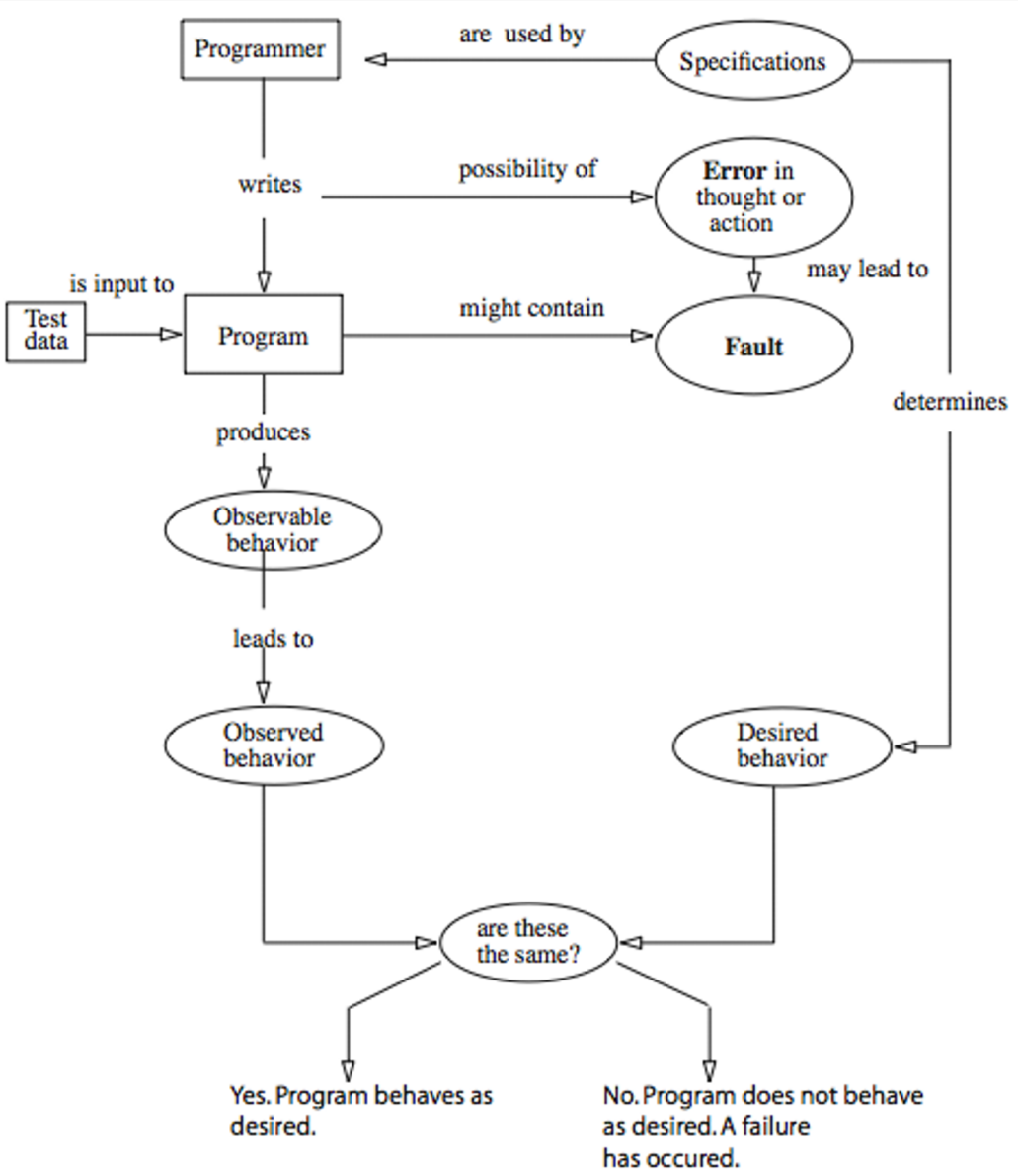
- Hardwire test inputs into the test harness
  - Simply execute the test harness whenever you fix a bug in the class that is being tested
- Place inputs on a file instead
  - Input redirection:
    - E.g. < test.in
    - Some IDEs do not support input redirection - use command window (shell).
- Generate test inputs automatically
  - For few possible inputs, feasible to run through (representative) number of them with a loop
- Generate test inputs randomly

Test Case Evaluation

How do you know whether the output is correct?

- Calculate correct values by hand
  - For a payroll program, compute taxes manually
- Supply test inputs for which you know the answer
  - square root of 4 is 2 and square root of 100 is 10
- Verify that the output values fulfill certain properties
  - square root squared = original value
- Use an Oracle
  - A *reliable* existing method to compute a result
  - E.g. `Math.pow` for $x^{1/2}$

Programming, Faults & Testing



What Is Automated Testing?

- **Test automation is software that automates any aspect of testing an application system**
  - Generate test inputs and expected results
  - Run test suites without manual intervention
  - Evaluate pass/no pass
- **Extent of automated testing depends on:**
  - Testing goals and budget
  - Development process
  - Kind of application under development
  - Particulars of the development and target environment

Automated Testing: Advantages

Improved Productivity:

- Repeatable tests
- Long and complex tests
- More time to design tests achieving greater coverage
- The only repeatable and efficient way to evaluate a large quantity of output.
- Quick and efficient verification of bug fixes
- Consistent capture and analysis of test results

Reduced Costs

- *What about the cost of test automation?*
  - It is typically recovered after two or three development cycles from increased productivity and the avoided costs with buggy software.

Automated Testing: Limitations

- A skilled tester using knowledge of product features to improvise tests can be effective.
- If no need to repeat the tests is evident, the cost of automation is probably not justified.
- Test suites and harness must be maintained, which is no less difficult or expensive than maintaining application software.

Combining automated and manual testing is a common and effective practice.