STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

SSW-540: Fundamentals of Software Engineering

Software Testing and Evolution

Roberta (Robbie) Cohen, Ph.D.
Industry Professor
School of Systems and Enterprises

Testing, verification and validation

- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- Testing reveals the **presence** of errors, NOT their absence.
  - Testing builds confidence in correctness
  - Testing, in practice, can't cover every possible behavior
- Testing is part of a more general verification and validation process, which includes static validation techniques as well as active testing of executing code.

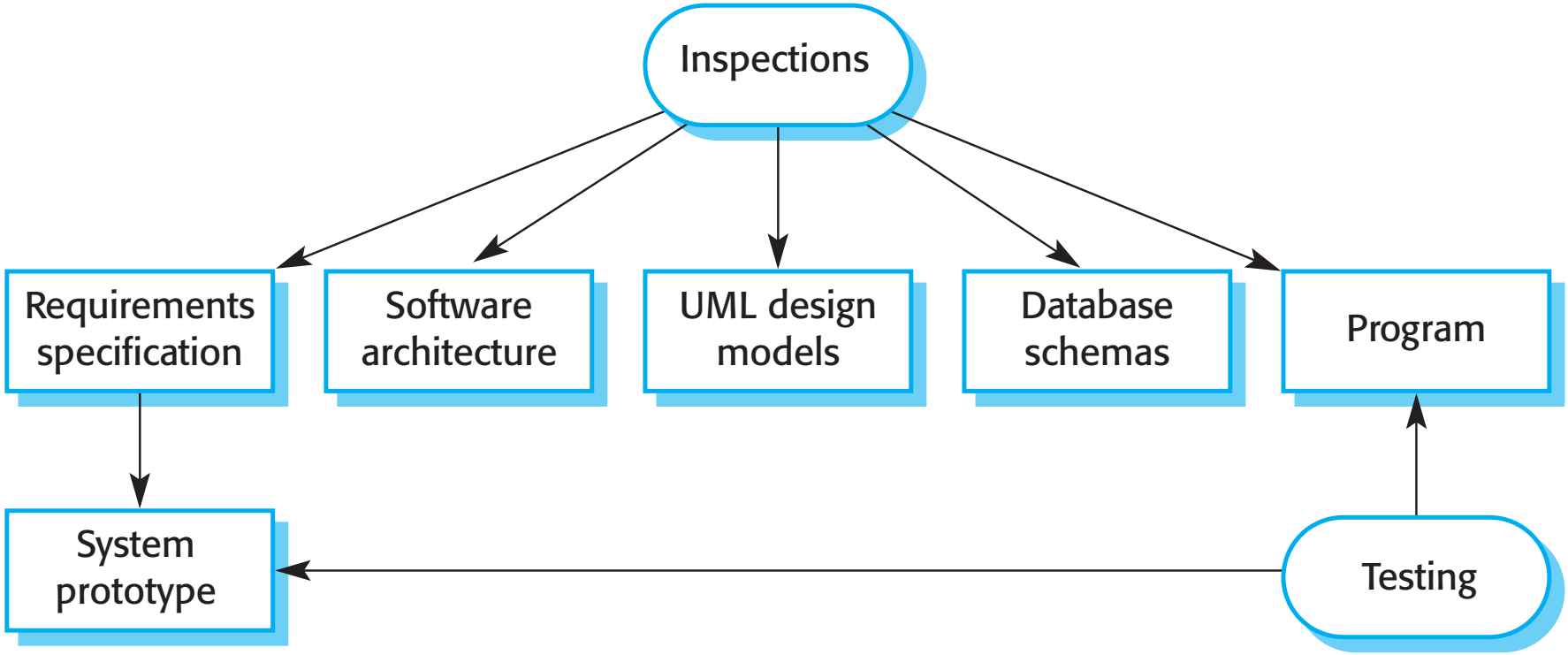
Testing process goals

- Validation testing
  - To demonstrate to the developer and the system's customer that the software meets its requirements
  - A successful test shows that the system operates as intended.
  - The aim is to establish confidence that the software system is "fit for purpose" (meets user expectations and the market environment)
- Defect testing
  - To discover faults or defects in the software where the software's behavior is incorrect or not in conformance with its specification
  - A test that makes the system perform incorrectly and so exposes a defect in the system is the goal.

Static validation

- Code reviews (aka inspections; a form of white box testing)
  - Highly effective, but very time consuming
    - Informal reviews are quite doable
    - Even formal reviews a worthwhile for new coders and for volatile code
  - Achieved in Agile projects via pair programming and common *ownership* of code
- Static analysis (aka Source Code Analysis, white box testing)
  - Analyzes code without executing it to find defects and/or determine conformance to coding guidelines
  - Often done as an integrated part of a build process
  - Usually tool-based; tools are language-specific
  - Can be used throughout development

Inspections are valuable for many software work products

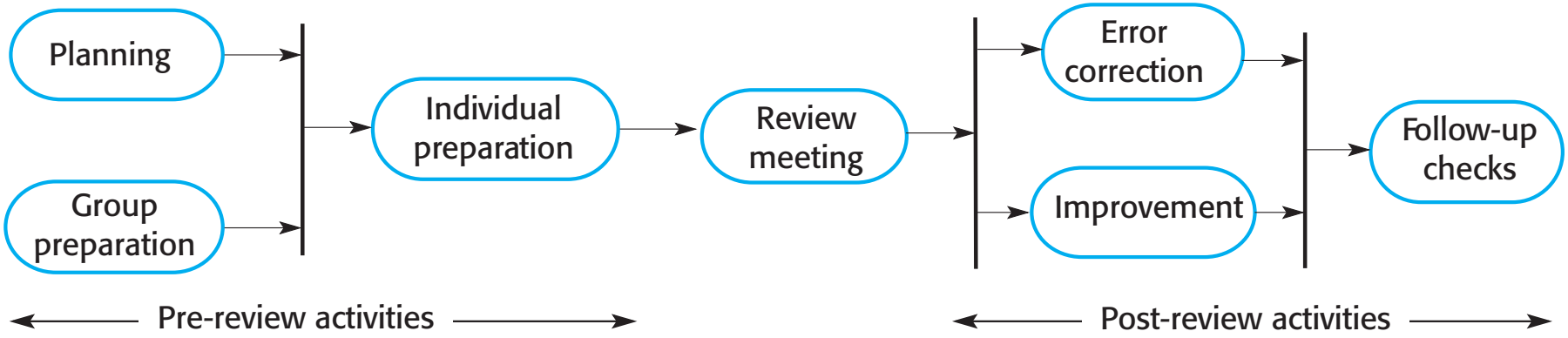


Reviews and inspections

- A group examines part or all of a process or system and/or its documentation to find potential problems.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- There are different types of review with different objectives
  - Inspections for defect removal (product);
  - Reviews for progress assessment (product and process);
  - Quality reviews (product and standards).
    - Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- Reviews may be local or distributed.

Phases in the review process

- Pre-review activities -- concerned with review planning and review preparation
- The review meeting -- an author of the document or program being reviewed 'walks through' the material with the review team.
- Post-review activities -- addressing the problems and issues that have been raised during the review meeting.



Inspections and checklists

- Inspections do not require execution of a system so may be used before implementation.
- They have been shown to be an effective technique for discovering program errors.
- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialization, constant naming, loop termination, array bounds, etc.

An inspection checklist (page 1 of 2)

Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?

An inspection checklist (page 1 of 2)

Fault class	Inspection check
Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

Inspections plusses and minuses

- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, specialized test harnesses are needed to dynamically test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, security, portability and maintainability.
- But inspections cannot test non-functional implementation characteristics such as performance, usability, reliability, etc.

Static program analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Automated static analysis checks

Fault class	Static analysis check examples
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter-type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic Memory leaks

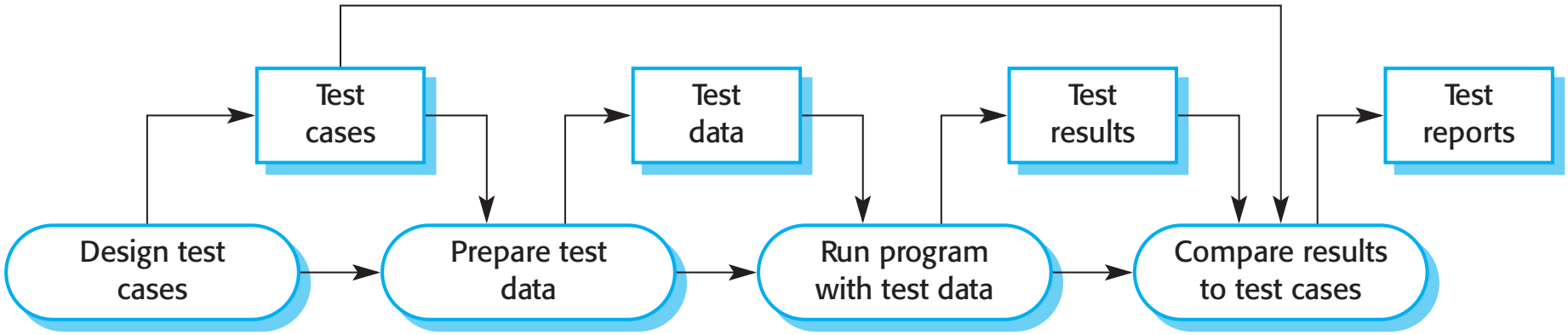
Levels of static analysis

- Characteristic error checking
  - The static analyzer can check for patterns in the code that are characteristic of errors made by programmers using a particular language.
- User-defined error checking
  - Users of a programming language define error patterns, thus extending the types of error that can be detected. This allows specific rules that apply to a program to be checked.
- Assertion checking
  - Developers include formal assertions in their program and relationships that must hold. The static analyzer symbolically executes the code and highlights potential problems.

Use of static analysis

- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler.
- Particularly valuable for security checking – the static analyzer can discover areas of vulnerability such as buffer overflows or unchecked inputs.
- Static analysis is now routinely used in the development of many safety and security critical systems.

Dynamic software testing process



- Performed multiple times at various stages for large systems:
  - Development testing, where the system is tested during development to discover defects.
  - Release testing, where a separate testing team tests a complete version of the system before it is released to users.
  - User testing, where users or potential users of a system test the system in their own environment.

Dynamic development testing

- Unit testing to defect test individual objects and methods; focused on testing functionality.
- Component testing (aka integration testing) to test related groups of objects (composite components); focused on testing component interfaces.
- System testing to test partial or complete software systems; focused on testing component interactions.
- Regression testing to verify that code changes haven't broken any code.

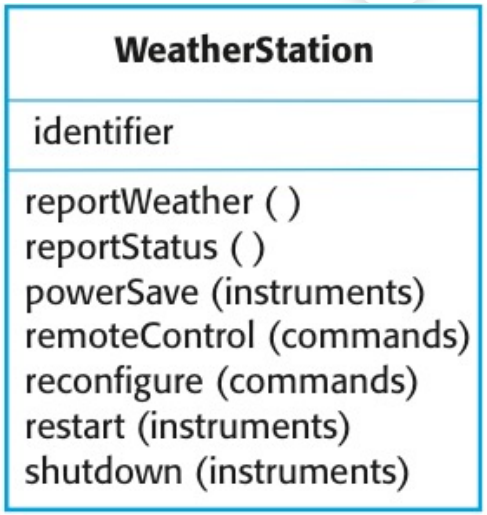
Unit Testing
- Test new features

Component Test
- Combine and test code from multiple developers
- Expose defects in interfaces between integrated components and systems

Regression Test
- Verify that new changes haven't broken previously working code

System Test
- Test the complete system
  - Functionality
  - Performance
  - Stress
  - Security

Unit test example: object class testing



- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localized.
- Example: weather station testing
  - Need to define test cases for reportWeather, reportStatus, etc.
  - Need to identify sequences of state transitions to be tested and the event sequences to cause these transitions
  - E.g.: Shutdown→Running→Shutdown & Configuring→Running→Testing→Transmitting→Running

Automated tests

- Wherever possible, we write automated tests so the tests are run and checked without manual intervention.
- The tests are embedded in a program that can be run every time a change is made to a system.
- Unit testing frameworks (like JUnit) provide generic test classes that you extend to create specific test cases. They run the tests and often report on their results as well.
- Each automated test includes
  - A setup, where you initialize the test with the inputs and expected outputs.
  - A call, where you call the object or method to be tested.
  - An assertion, where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

Test case selection

- The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
- If there are defects in the component, these should be revealed by test cases. So test cases should try to ‘break’ the software by looking for the kinds of defects found in other systems.
- This leads to **2 types of unit test cases**:
  - The first of these should reflect normal operation of a program and should show that the component works as expected.
  - The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

Unit testing strategies

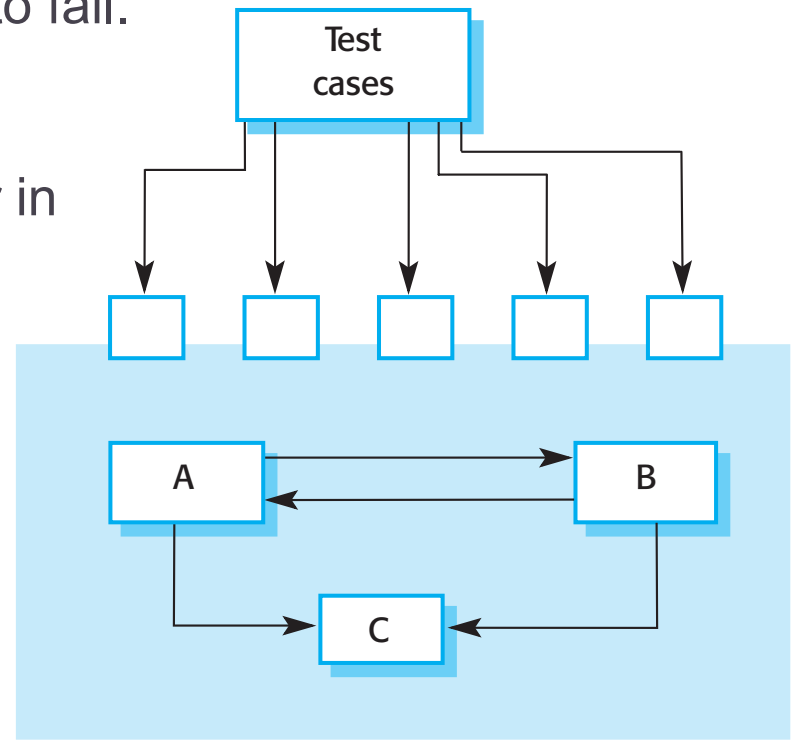
- Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - Input data and output results often fall into different classes where all members of a class are related.
  - Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
  - Test cases should be chosen from each partition & include boundaries
- Guideline-based testing, where test cases are drawn from common error areas.
  - Choose inputs that force the system to generate all error messages
  - Design inputs that cause input buffers to overflow
  - Repeat the same input or series of inputs numerous times
  - Force invalid outputs to be generated
  - Force computation results to be too large or too small.

Component testing - interfaces

- Also known as Integration Testing since it tests integrations of the unit tested work of multiple developers
- Objective is to detect faults due to interface errors or invalid assumptions about **interfaces**.
- Interface errors
  - Interface misuse - A calling component calls another component and makes an error using the interface e.g. parameters in the wrong order.
  - Interface misunderstanding - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
  - Timing errors - The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the boundaries of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Stress test message passing systems.
- In shared memory systems, vary the order in which components are activated.
- Testing is important for all types of systems, especially message passing systems



System testing

- System testing during development involves integrating all the components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the **substantive** interactions (not the interfaces) between components.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces so as to fulfil the needed function and non-functional requirement(s).
- System testing tests the emergent behavior of a system.
- The use-cases developed to identify system interactions can be used as a basis for system testing.

Regression Testing

- Confirm that an addition or change in the code or system has not adversely impacted existing features
- Usually involves rerunning already executed test cases
- Find a bug; fix a bug; regression test all potentially affected code
- Options:
  - Retest all
  - Select regression tests cases – including some from unaffected code
  - Prioritize test cases by business impact, criticality, or frequency of function use

Selecting regression test cases

There is something of an art to selecting regression test case:

- Test cases which have frequent defects
- Functionalities which are more visible to the users
- Test cases which verify core features of the product
- Test cases of functionalities which have undergone more and recent changes
- All integration test cases
- All complex test cases
- Boundary value test cases
- A sample of successful test cases
- A sample of failure test cases

Use of test automation tools eases this process

System testing policies

- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
  - All system functions that are accessed through menus should be tested.
  - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - Where user input is provided, all functions must be tested with both correct and incorrect input.

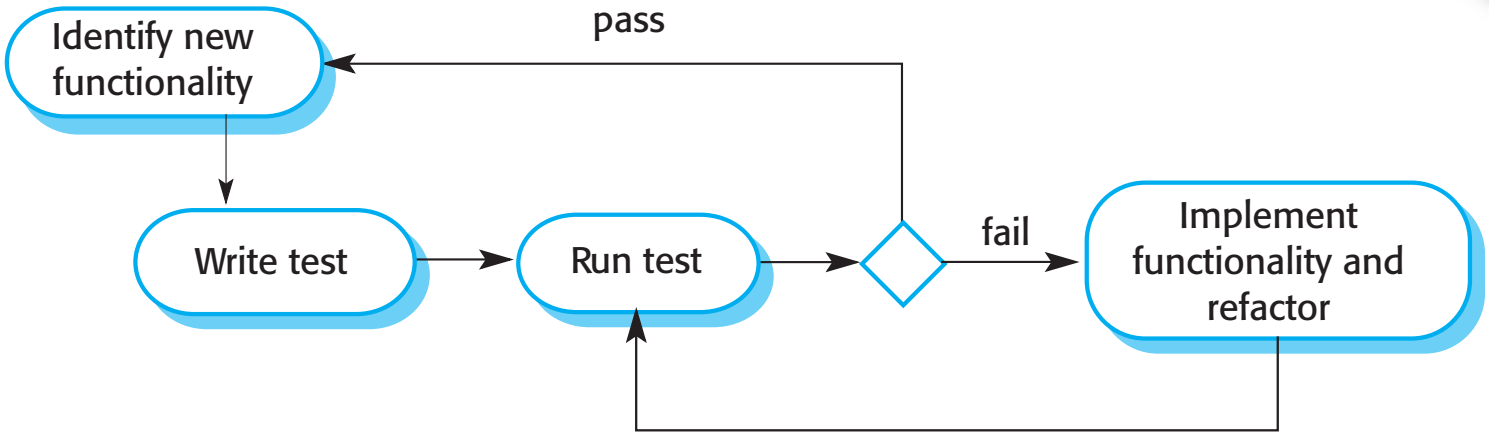
Why do we say that exhaustive system testing is impossible?

7 Testing Principles from video

(https://www.youtube.com/watch?v=rFaWOw8bIMM)

1. Testing shows presence of defects
2. Exhaustive testing is impossible
3. Early testing
4. Defect clustering
5. Pesticide paradox
6. Testing is context dependent
7. Absence of errors - fallacy

Test-driven development



- Test-driven development (TDD) is an approach to development in which we inter-leave testing and code development.
- Tests are written before code and ‘passing’ the tests is the critical driver of development.
- We develop code incrementally. We don’t move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming, but it is used in plan-driven development too.

Benefits of test-driven development

- Code coverage
  - Every code segment that you write has at least one associated test
- Regression testing
  - A regression test suite is developed incrementally as the software is developed.
- Simplified debugging
  - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing.

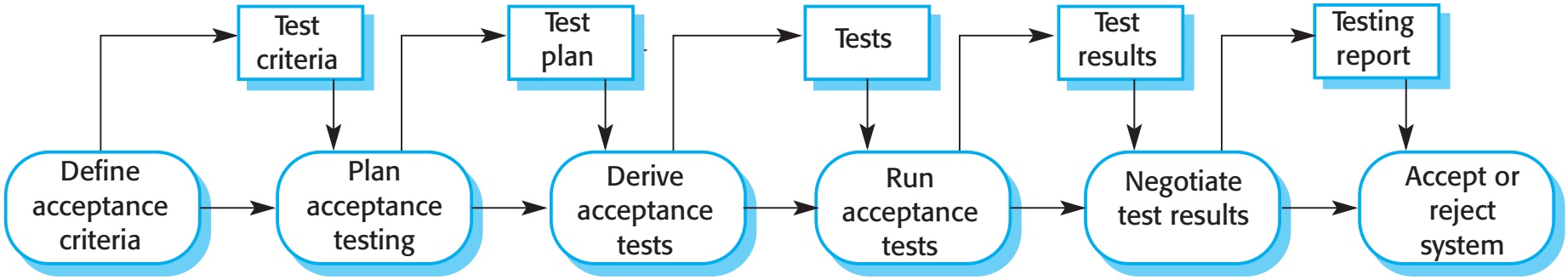
Release testing

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
  - Release testing has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a *black-box* testing process where tests are only derived from the system specification.
  - Such testing is aimed at validating the system, not discovering defects (defect testing was what system test did!)
  - A separate team, one not involved in the system's development, should be responsible for release testing.

Scenario testing often used in release testing

- Just as use cases are often used in system testing, release test cases often derive from typical usage scenarios.
- The features of the software system get tested in the sequences in which they will often be called.
- Part of release testing involve testing emergent properties such as performance and reliability.
- These tests should reflect the profile of use of the system—not a specific scenario but a profile representing the frequencies of scenario occurrences.
- Such a profile is called an **operational profile**.

(User) Acceptance testing



- A testing process where the aim is for the **user** to decide if the software is good enough to be deployed and used in its operational environment.
- In agile methods,
  - the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
  - Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
  - There is no separate acceptance testing process.
  - Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.

Test Plan Document(s)

- Identifies the testing that will be performed for each level and type of testing and who will perform those tests
- E.g.:
  - Unit tests
  - Integration tests
  - System tests
  - Acceptance tests
  - Specific testing
    - Performance testing
    - Security testing
    - Others, as needed

Detailed Contents

- Features to be tested
- Features not to be tested
- Approach
- Item pass/fail criteria
- Suspension criteria & resumption requirements
- Test deliverables (test cases, scripts, logs, etc.)
- Test environment
- Estimate of costs/effort
- Schedule
- Staffing and training needs
- Responsibilities
- Risks
- Assumptions and Dependencies

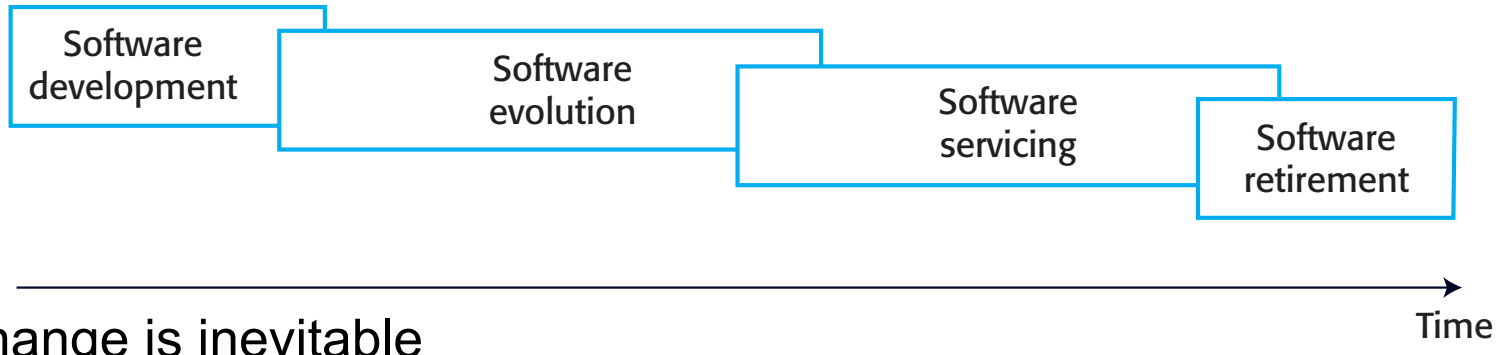
Testing, in sum…

- Testing shows the presence of errors—not their absence.
- Development testing includes unit, component and system test.
- Release testing should be done by a separate team.
- Acceptance testing should be user testing.

Before we move on to software evolution and change…

Watch https://www.youtube.com/watch?v=oLc9gVM8FBM video describing testing in the age of Agile.

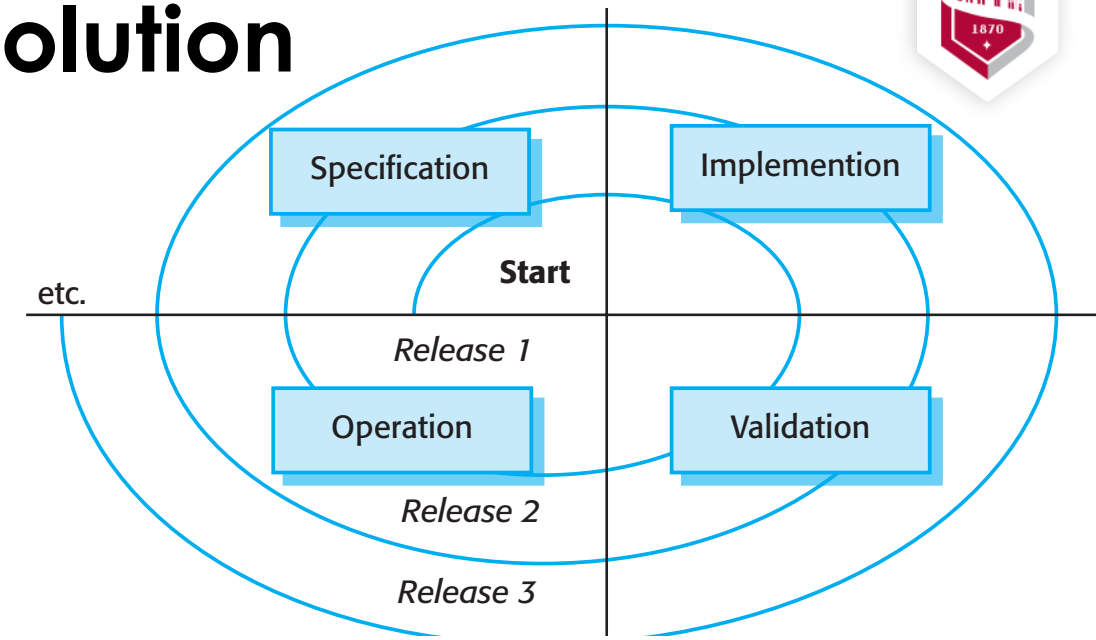
And just when you think you are done…



- Software change is inevitable
  - New requirements emerge when the software is used;
  - The business environment changes;
  - Errors must be repaired;
  - New computers and equipment is added to the system;
  - The performance or reliability of the system may have to be improved.
- A key problem for all organizations is implementing and managing change to their existing software systems.

Importance of evolution

A spiral model of development and evolution



- Organizations have huge investments in their software systems - they are critical business assets.
- To maintain the value of these assets to the mission/business, they must be changed and updated.
- The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

Evolution and services

- Evolution

  The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.

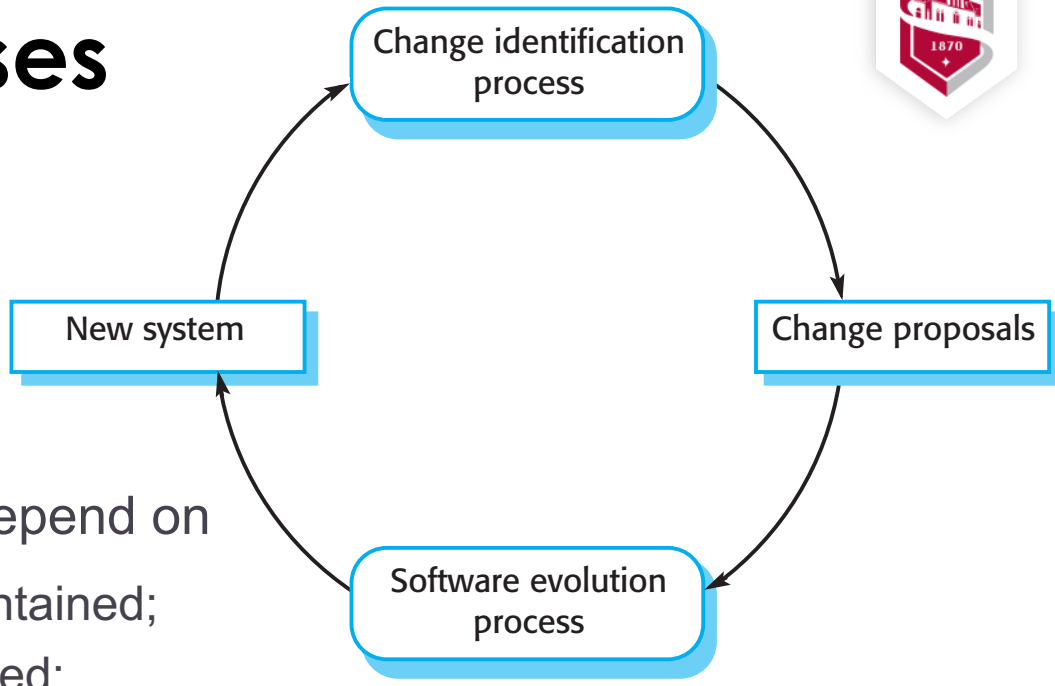
- Servicing

  At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software's environment. No new functionality is added.

- Phase-out

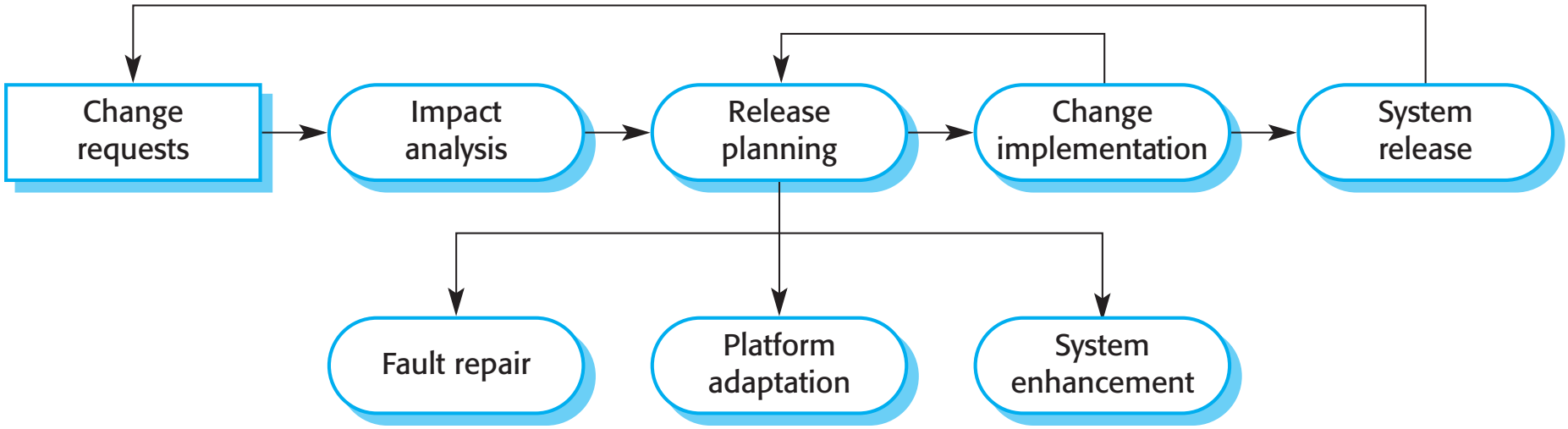
  The software may still be used but no further changes are made to it.

Evolution processes

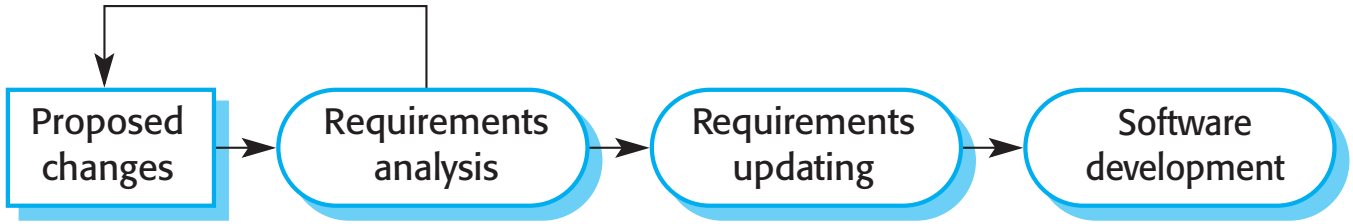


- Software evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- Proposals for change are the driver for system evolution.
  - Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- Change identification and evolution continues throughout the system lifetime.

The software evolution process



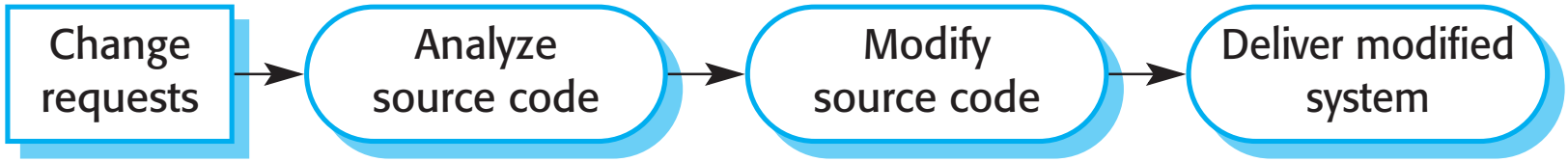
Understanding the existing software is a critical first step.



Change Implementation

Urgent change requests

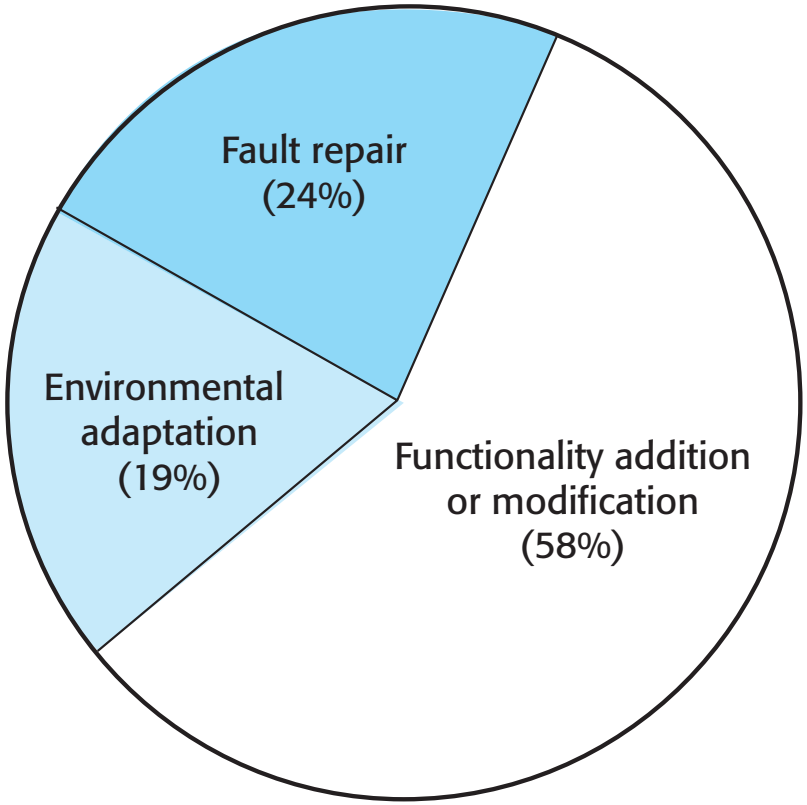
- Urgent changes may have to be implemented without going through all stages of the software engineering process
  - If a serious system fault has to be repaired to allow normal operation to continue;
  - If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
  - If there are business changes that require a very rapid response (e.g. the release of a competing product).



Agile methods and evolution

- Agile methods are based on incremental development so the transition from development to evolution is a seamless one.
  - Evolution is simply a continuation of the development process based on frequent system releases.
- Automated regression testing is particularly valuable when changes are made to a system.
- Changes may be expressed as additional user stories.
- Code changes are eased by the previous efforts toward simplification of code (refactorings)
- But if the system has grown, lack of documentation may hinder understanding.

Software evolution via maintenance



Maintenance effort distribution

- Changing software after it has been put into use.
- The maintenance term is mostly used for changing custom software. Generic software products are said to evolve to create new versions.
- Maintenance does not normally involve major changes to the system's architecture.
- Changes are implemented by modifying existing components and adding new components to the system.

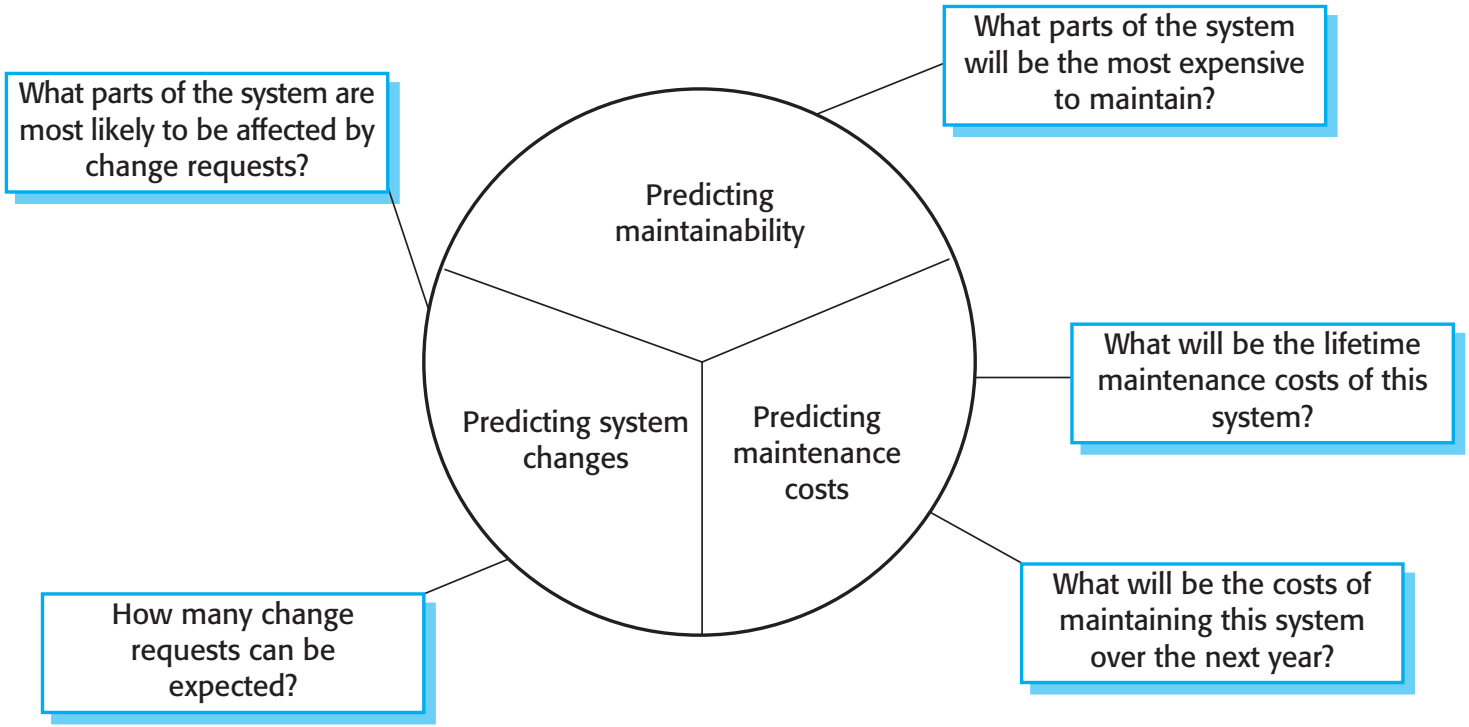
Software maintenance costs

- Usually exceed development costs (2x to 100x) for custom software.
- Affected by both technical and non-technical factors.
- Costs increase as software is maintained. Maintenance corrupts the software structure which makes further maintenance more difficult.
- Aging software can have high support costs (e.g. old languages, compilers).
- It's usually more expensive to add new features during maintenance than to add the same features during development
  - A new team must understand the programs being maintained
  - Separate maintenance and development teams means there is no incentive for the development team to write maintainable software
  - Program maintenance work is unpopular
    - Maintenance staff are often inexperienced with limited domain knowledge.
    - They need more time to do what the experienced original programmer can do quickly.

Maintenance prediction

Predicting maintenance activities and costs supports effective resource planning.

- Tightly coupled systems require changes whenever the environment is changed.
- Complexity drives maintenance costs, too.
  - Most maintenance effort is spent on a relatively small number of components.
  - We can measure complexity of our software.



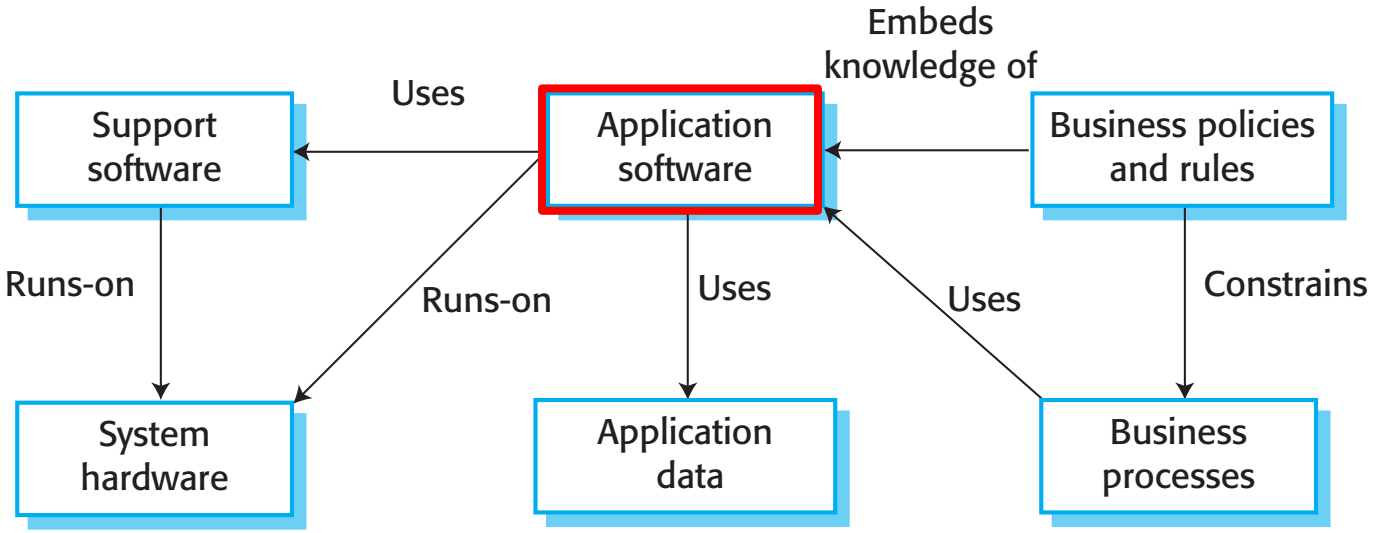
Complexity metrics

- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- Complexity depends on
  - Complexity of control structures;
  - Complexity of data structures;
  - Object, method (procedure) and module size.

Process metrics

- Process metrics may be used to assess maintainability
  - Number of requests for corrective maintenance;
  - Average time required for impact analysis;
  - Average time taken to implement a change request;
  - Number of outstanding change requests.
- If any or all of these is increasing, this may indicate a decline in maintainability.

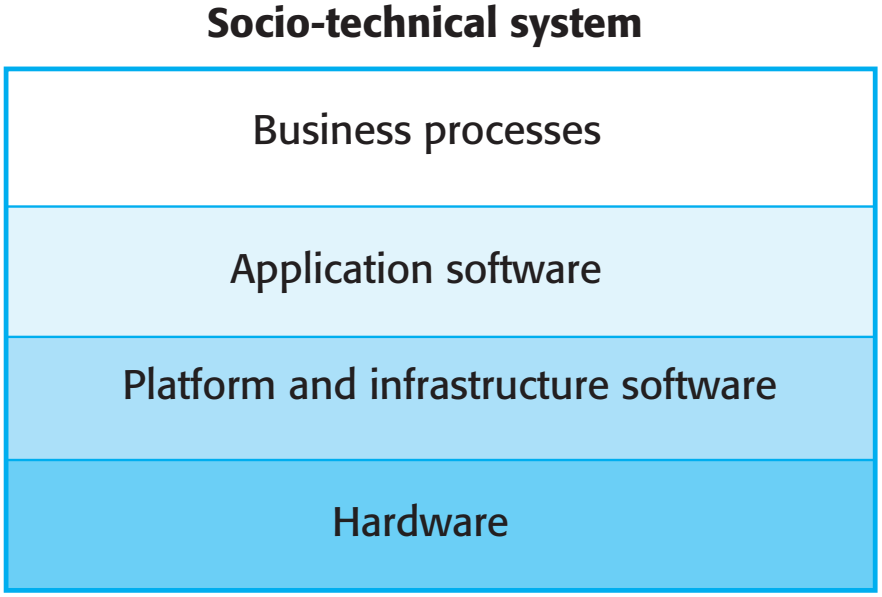
Legacy systems (*brownfield systems*)



- Legacy systems are older systems that often rely on languages and technology that are no longer used for new systems development.
- Legacy software may be dependent on older hardware, such as mainframe computers.
- Legacy systems are not just software systems but are broader socio-technical systems that include hardware, software, libraries and other supporting software and business processes.

Legacy system replacement

- Legacy system replacement is risky and expensive so businesses continue to use these systems
- System replacement is risky for a number of reasons
  - Lack of complete system specification
  - Tight integration of system and business processes
  - Undocumented business rules embedded in the legacy system
  - New software development may be late and/or over budget



Legacy system change

- Legacy systems are expensive to change for a number of reasons:
  - No consistent programming style
  - Use of obsolete programming languages with few people available with these language skills
  - Inadequate system documentation
  - System structure degradation
  - Program optimizations may make them hard to understand
  - Data errors, duplication and inconsistency

What you do about legacy systems

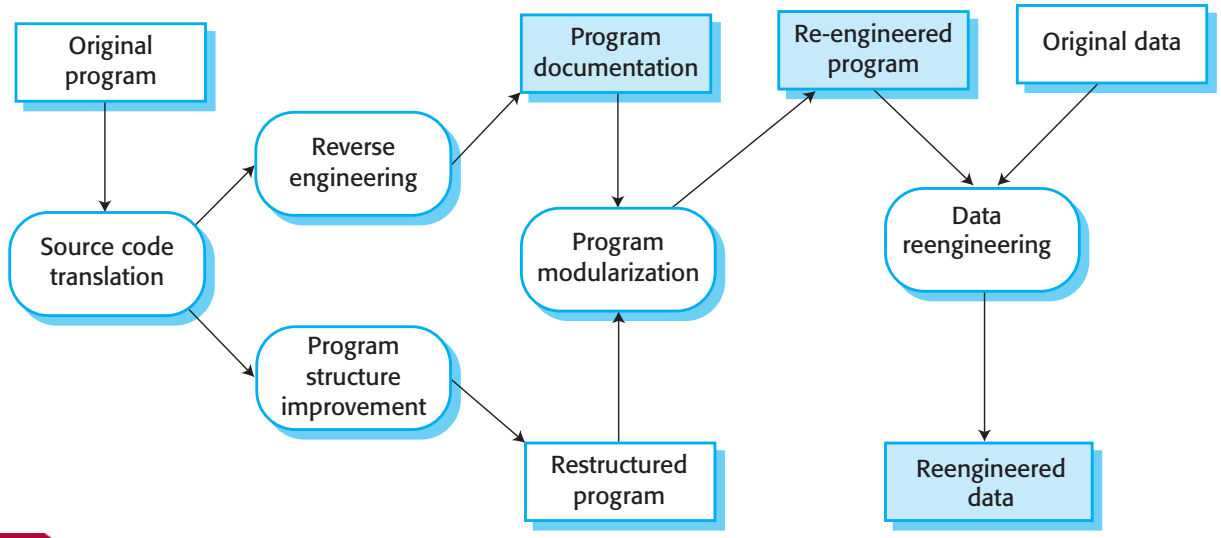
- Depends on the system's value to the business
  - Its uses
  - The business processes that are supported
  - System dependability
  - The system outputs
- Depends on the system's quality
  - How well does the implemented business process support the current goals of the business?
  - How effective is the system's environment and how expensive is it to maintain?
  - What is the quality of the application software system?

System measurement

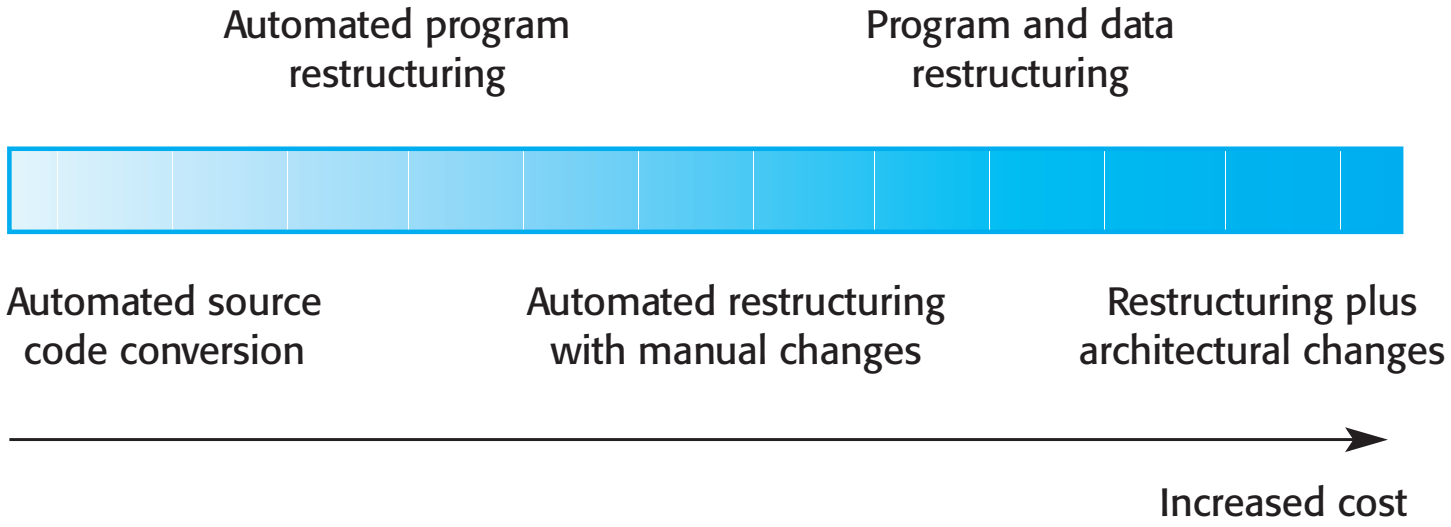
- For example, use quantitative data to assess the quality of the application system
  - The number of system change requests; The higher this accumulated value, the lower the quality of the system.
  - The number of different user interfaces used by the system; The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.
  - The volume of data used by the system. As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data.
  - Cleaning up old data is a very expensive and time-consuming process

Making maintenance easier: Software reengineering

- Restructuring or rewriting part of a legacy system without changing its functionality.
- Applicable where some sub-systems of a larger system require frequent, expensive maintenance.
- Reengineering involves adding effort to make software easier to maintain. The system may be re-structured and re-documented.
- Software reengineering is less risky than developing new software and costs significantly less.
- But it's still costly!



Reengineering costs



- Cost factors
  - The quality of the software to be reengineered.
  - The tool support available for reengineering.
  - The extent of the data conversion which is required.
  - The availability of expert staff for reengineering.
    - This can be a problem with old systems based on technology that is no longer widely used.

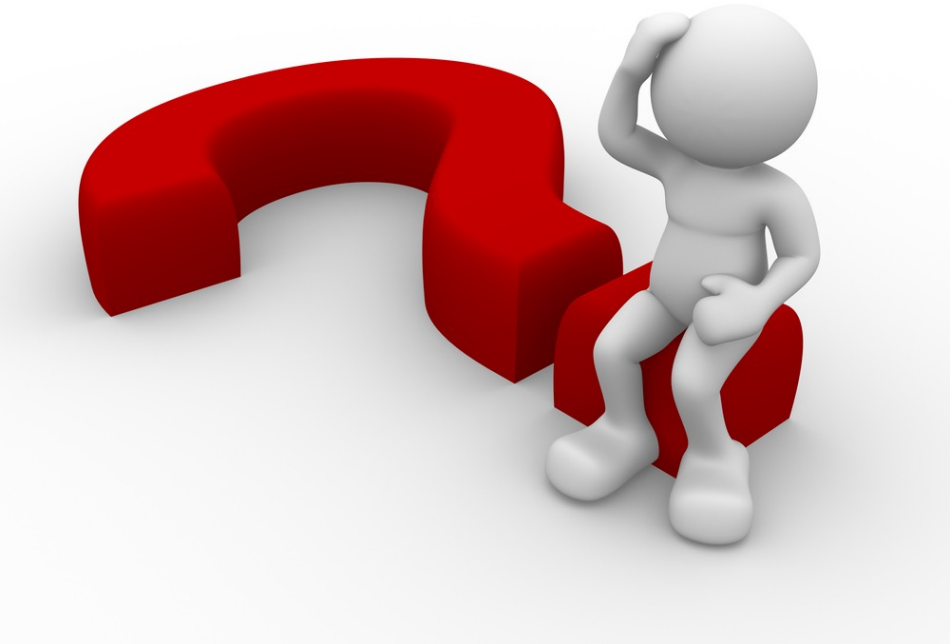
Refactoring systems to avoid costly maintenance

- **Reengineering** takes place after a system has been maintained for some time and maintenance costs are increasing.
- **Refactoring** is the continuous process of making improvements to a program to slow down change-driven degradation.
- You can think of refactoring as 'preventative maintenance' that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- When you refactor a program, you don't add functionality but rather concentrate on program improvement.
- Refactoring is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

In sum…

- We test to find defects and to assure requirements have been met.
- Testing can be static or dynamic
- Development testing includes unit, component/integration, system and regression testing; followed by release and user testing.
- Testing gets you into the game. Software evolution keeps you there.
- Software maintenance includes fixing bugs, modifying software to work in new environments, and implemented new requirements. It is driven by change requests.
- Reengineering is restructuring and redocumenting software to make it easier to understand and maintain.
- Refactoring is preventative maintenance—small changes continuously applied.

Questions?



Individual Exercise this week

In the previous Python assignment, you wrote a script for the Guess a Number game, perhaps with a function that compared each guess to the randomly generated target number.

1. What kind of tests would you run on this program? For what would you test?
2. What testing partitions would be appropriate for testing the function comparing guesses to the target number?
3. What testing of ATM software would be analogous to the tests and partitions used for your Guess a Number game?