# SSW-540: Fundamentals of Software Engineering

*Software Testing and Evolution*

Dr. Richard Ens
School of Systems and Enterprises

# A Python Pearl (of wisdom)

If you haven’t figured it out yet, to grade your Python scripts, I test them.

1. I test for the nominal case(s) that are “sunny day” scenarios.
2. I test for the boundary case(s) where input or output to the script varies from the nominal.
3. I test for the exceptional case(s) where incorrect input is provided (like bad filenames).

Experience teaches that the best programs are those that ascertain that data they have to work with are the appropriate data. We call that “bounding the input”.

## Python Pointers: Exceptions and boundary conditions

- Exceptions are anomalous occurences
  - They disrupt the normal flow of a program or script.
  - They require special processing.
  - We use specialized handling constructs for exceptions (in Python we use the **try/except** construct).
- If a data value falls outside the boundaries we have set (say 25 is entered in response to a request for a number between 1 and 20), should we throw an exception?
  - Probably not.
  - Why? Because our normal process should include bounding our data.
  - Best practice would be to handle boundaries via a conditional statement like **if**.

# 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.

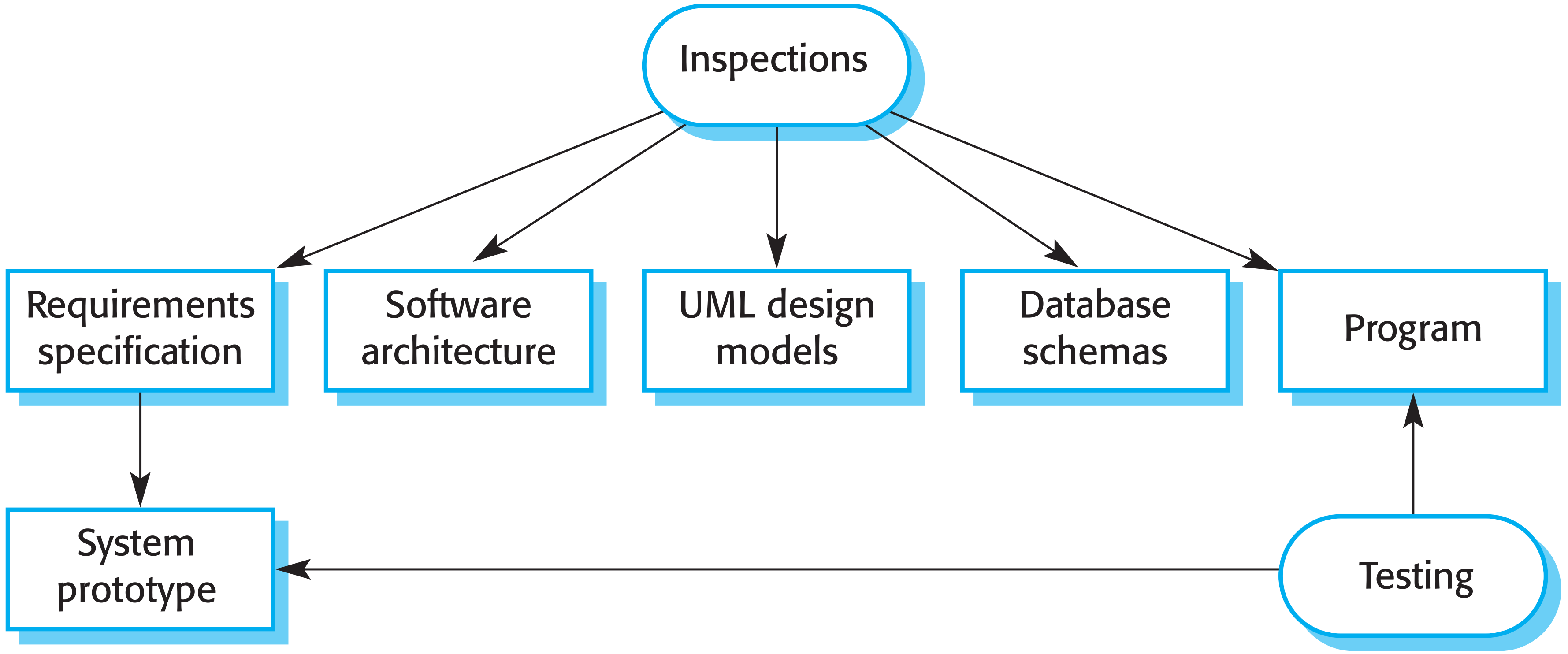
# Testing process goals

- 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

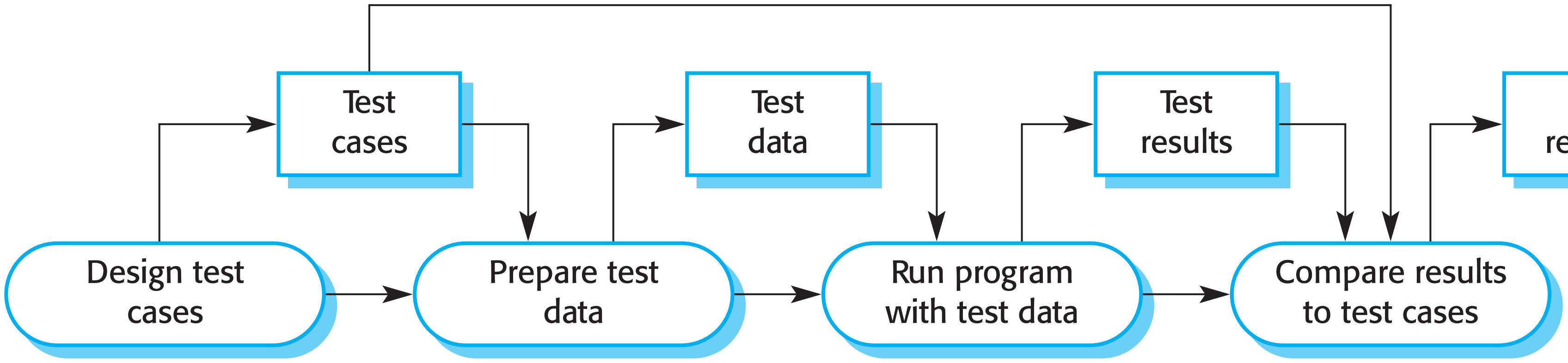
# Inspections are valuable for many software work products



## 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 actively 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.

# Dynamic software testing process



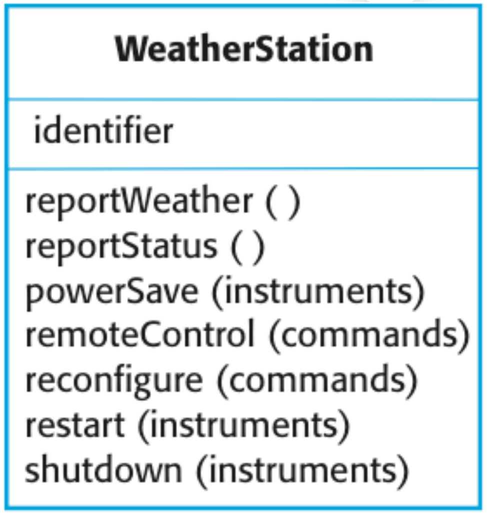
✧ Performed multiple times at various stages for large systems:

- 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.

## Development testing

- A responsibility of the development team.
- Includes
  - 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.

## Unit test example: object class testing



WeatherStation

identifier

reportWeather ( )
reportStatus ( )
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

- 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, you should 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 case:
  - 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

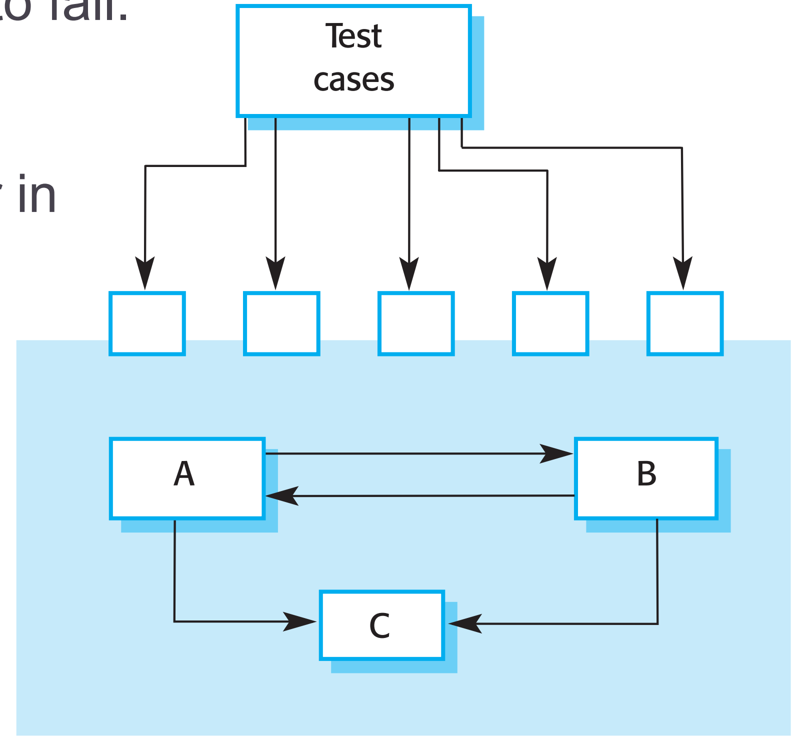
- Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - Input data and output results often fall into different classes where all members of a class are related.
  - Each of these classes is an equivalence partition or domain 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

- 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

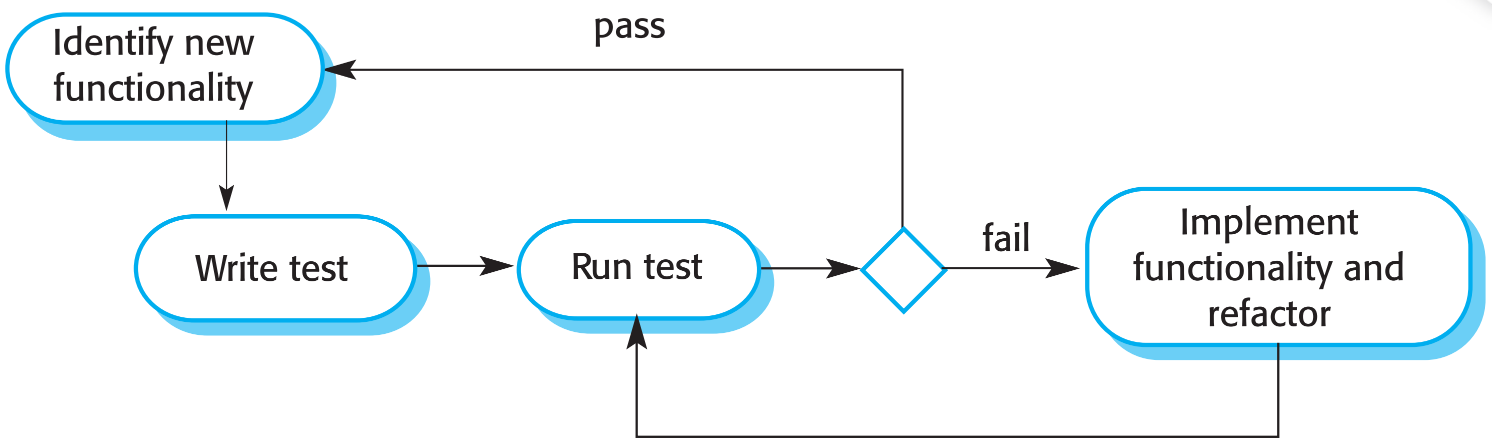
- System testing during development involves integrating all of the components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the interactions between components.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- 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.

## 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?*

# Test-driven development



Identify new functionality
pass
Write test
Run test
fail
Implement functionality and refactor

- Test-driven development (TDD) is an approach to development in which you inter-leave testing and code development.
- Tests are written before code and ‘passing’ the tests is the critical driver of development.
- You develop code incrementally. You 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 so all code written has at least one 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.

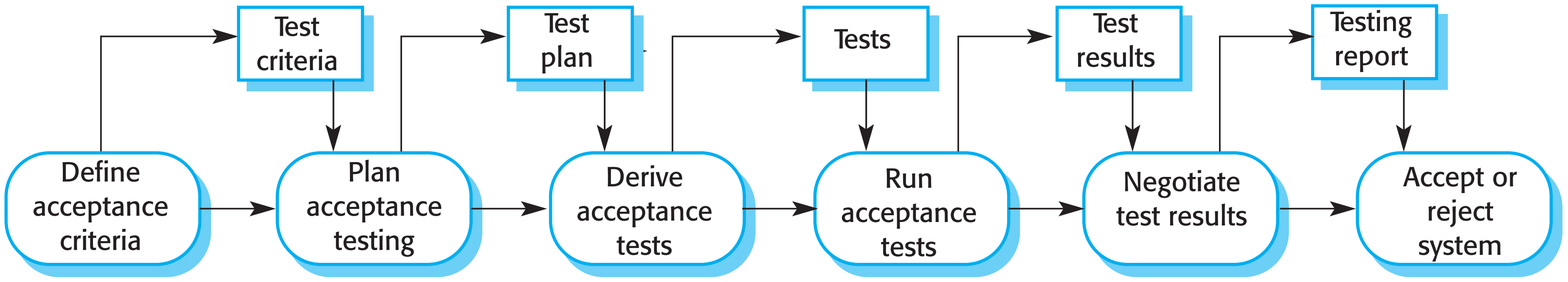
## Release testing

- 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, therefore, 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

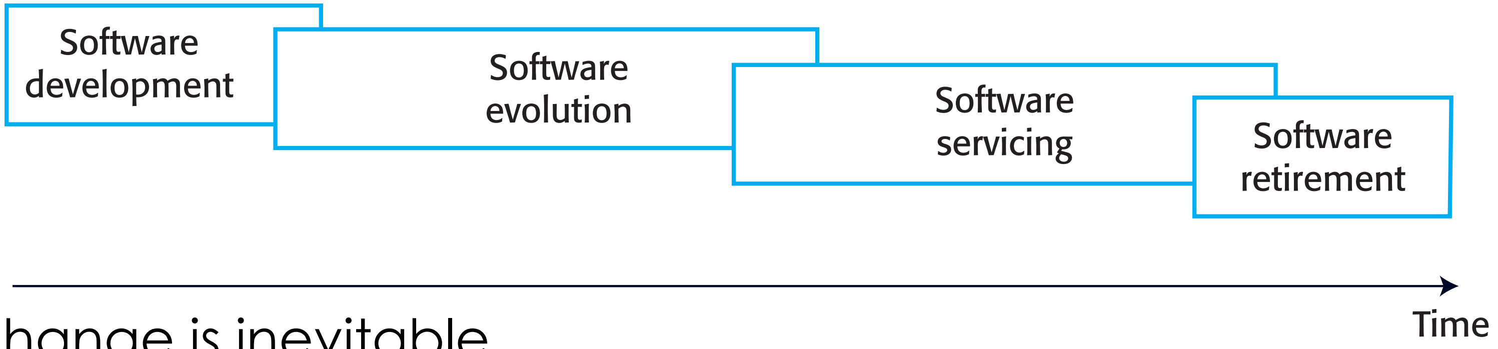
- Just as use cases are often used in system testing, release test cases often derive from typical usage scenarios.
- The features of the software system get tested in the sequences in which they will often be called.
- Part of release testing involve testing emergent properties such as performance and 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**.

# Acceptance testing



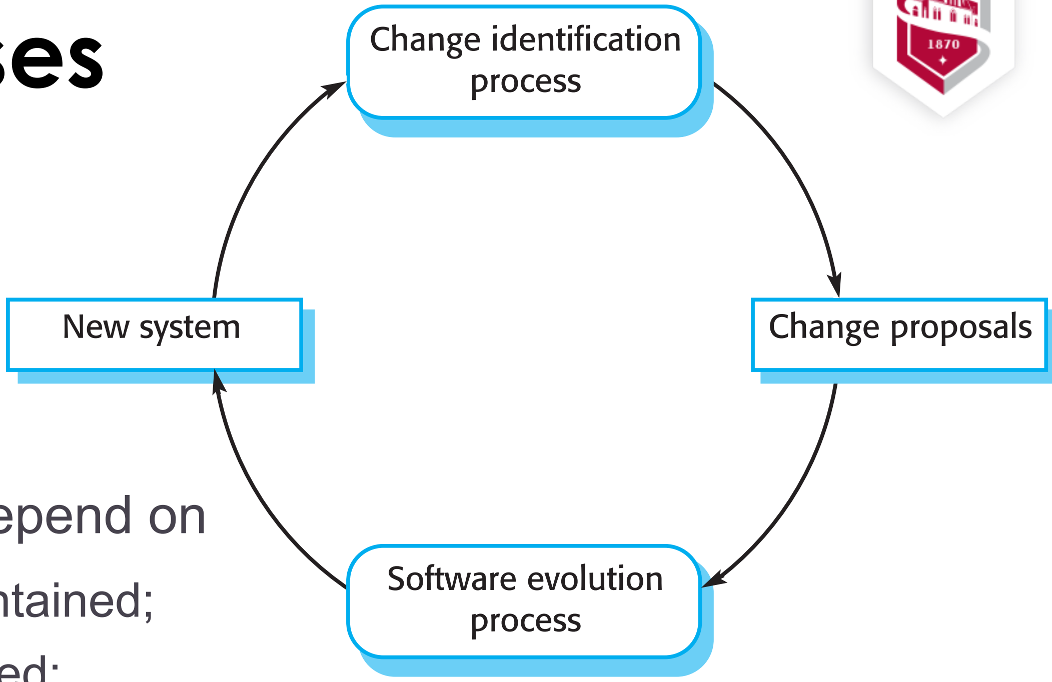
- A testing process where the aim is for the user to decide if the software is good enough to be deployed and used in its operational environment.
- In agile methods,
  - the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
  - 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.

# And just when you think you are done…



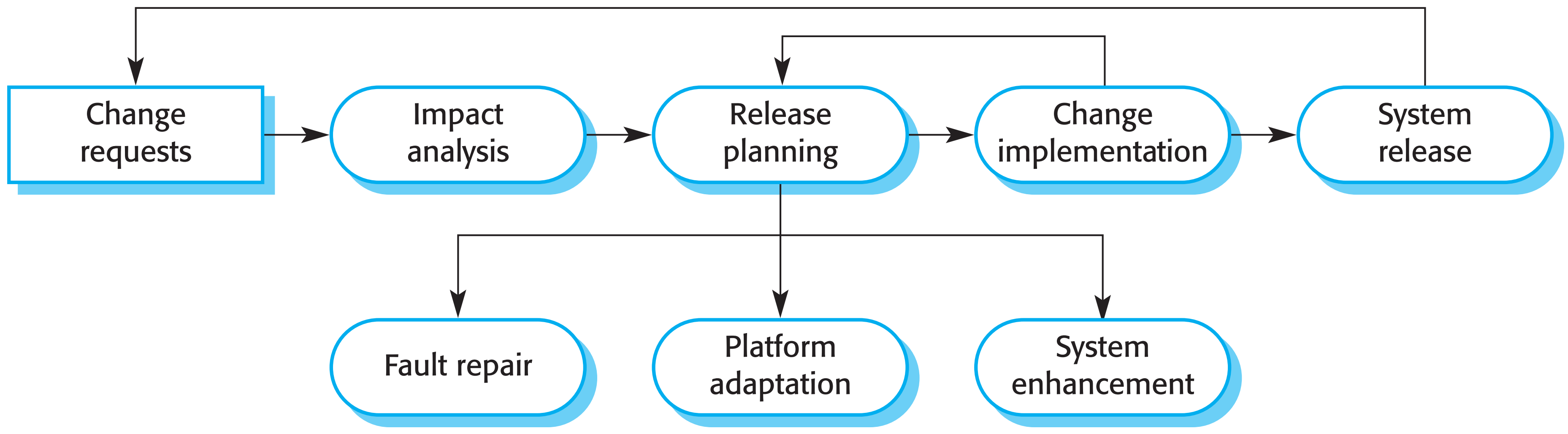
- Software change is inevitable
  - New requirements emerge when the software is used;
  - The business environment changes;
  - Errors must be repaired;
  - New computers and equipment is added to the system;
  - The performance or reliability of the system may have to be improved.
- A key problem for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

# Evolution processes

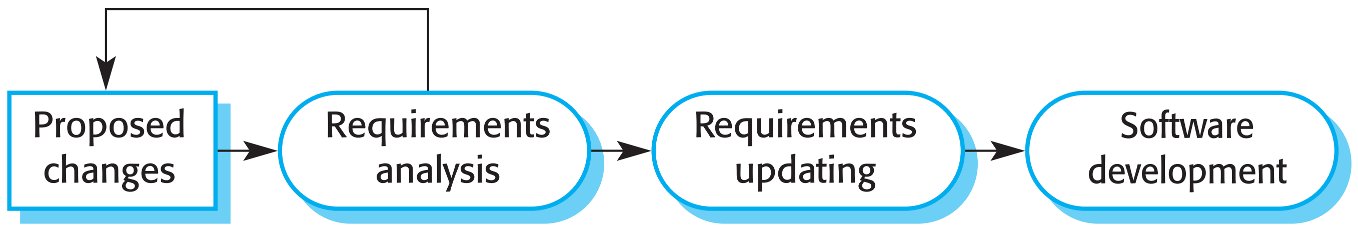


- Software evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- Proposals for change are the driver for system evolution.
  - Should be linked with components that are affected by the change, 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



Change Implementation:

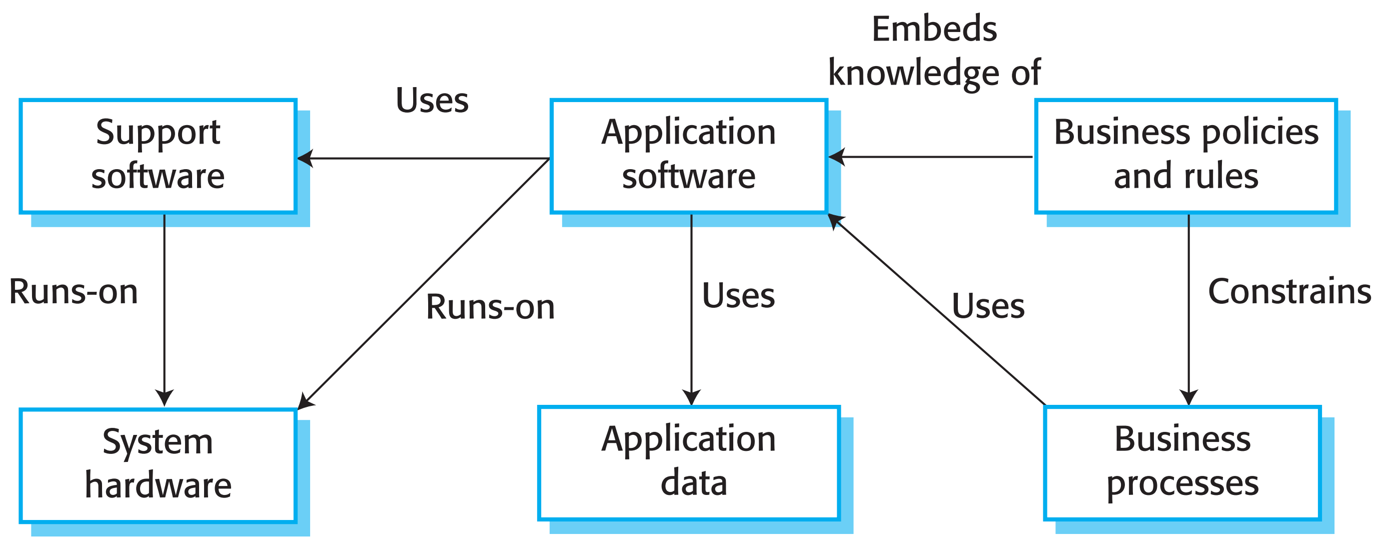


Understanding the existing software is a critical first step.

## 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.

# 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.

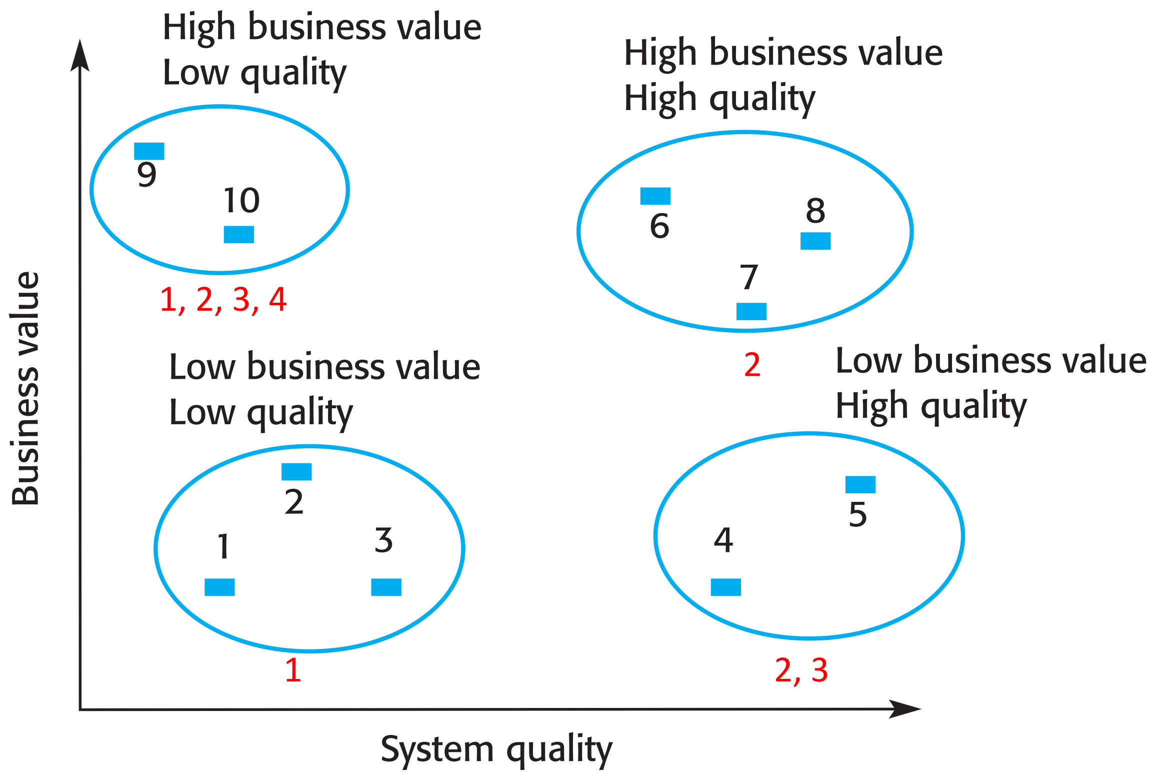
## Legacy system replacement and change

- Legacy system replacement is risky due to:
  - 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 systems are expensive to change because:
  - No consistent programming style
  - Use of obsolete programming languages with few people who have those language skills
  - Inadequate system documentation
  - System structure degradation
  - Program optimizations may make the software hard to understand
  - Data errors, duplication and inconsistency

# Strategies for legacy system management

Choices:

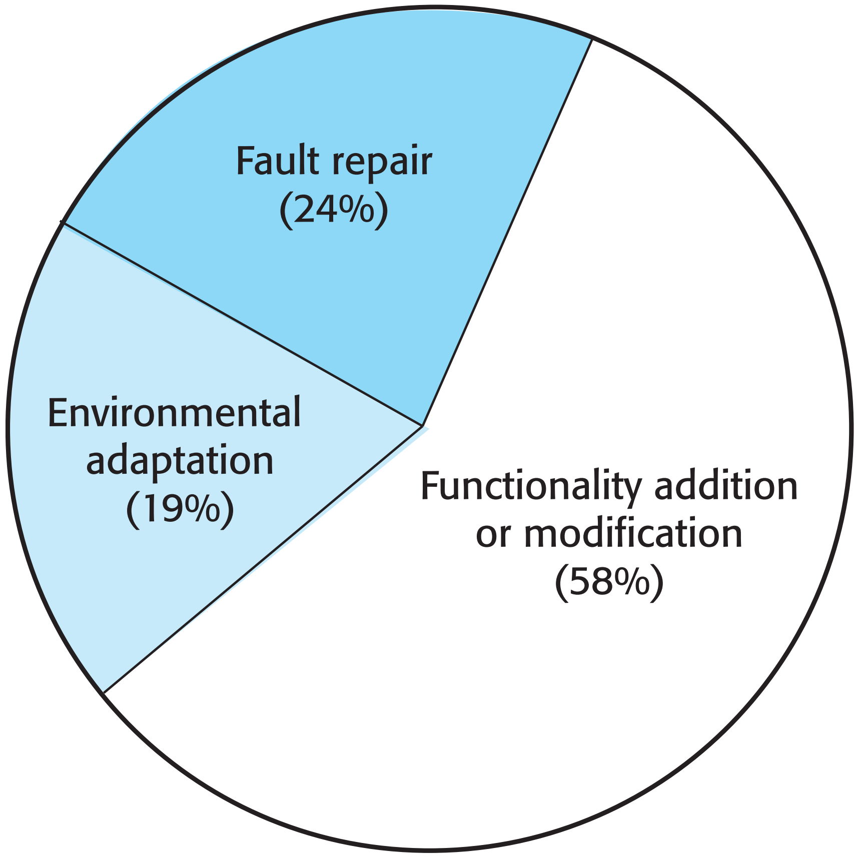
1. Scrap the system completely and modify business processes so that it is no longer required;
2. Continue maintaining the system;
3. Transform the system by re-engineering to improve its uses & maintainability;
4. Replace the system with a new system.



# System quality assessment

- Business process assessment
  - How well does the business process support the current goals of the business?
- Environment assessment
  - How effective is the system's environment and how expensive is it to maintain?
- Application assessment
  - What is the quality of the application software system?
  - You may use quantitative data to make an assessment of the quality of the application system
    - The number of system change requests.
    - The number of different interfaces & user interfaces used by the system.
    - The volume of data used by the system. (Cleaning up old data is a very expensive and time-consuming process.)

# Software maintenance



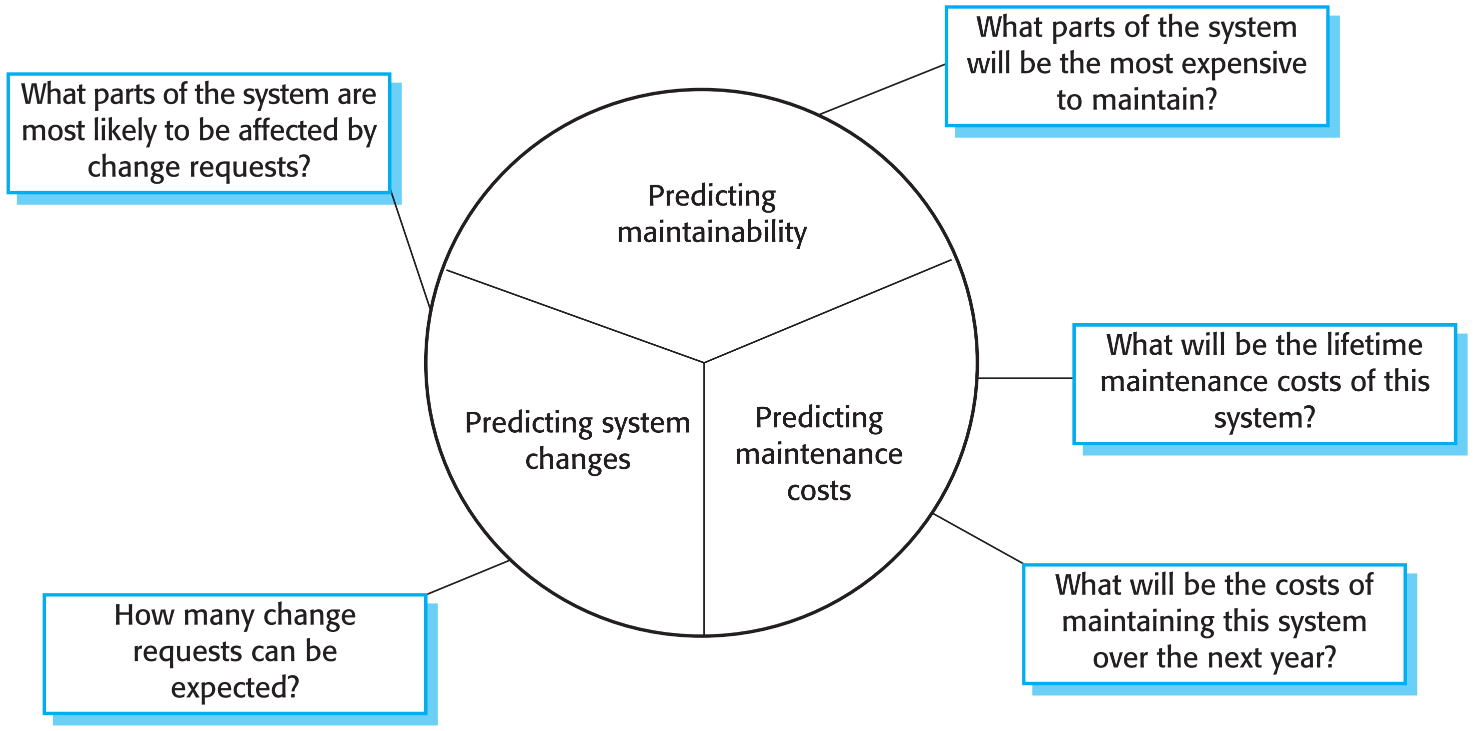
Maintenance effort distribution

- Modifying a program after it has been put into use.
- The 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.

## Maintenance costs

- Usually exceed development costs (2x to 100x).
- 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 etc.).
- It's usually more expensive to add new features to a system during maintenance than it is to add the same features during development
  - A new team has to understand the programs being maintained
  - Separating maintenance and development 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.

# Maintenance prediction

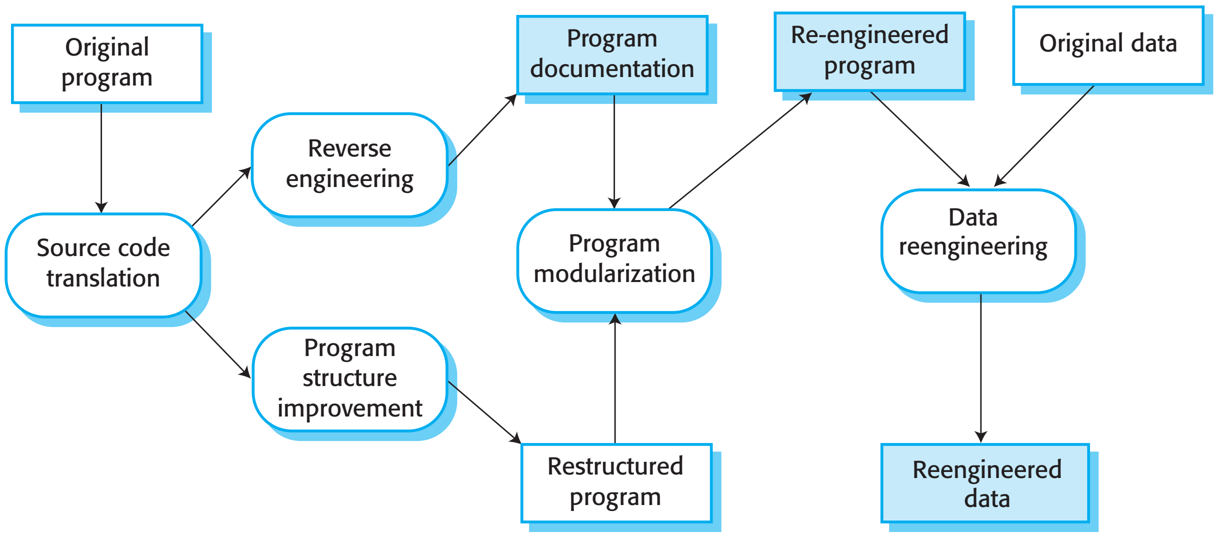

What parts of the system are most likely to be affected by change requests?
What parts of the system will be the most expensive to maintain?
Predicting maintainability
What will be the lifetime maintenance costs of this system?
Predicting system changes
Predicting maintenance costs
How many change requests can be expected?
What will be the costs of maintaining this system over the next year?

Predicting maintenance activities and costs supports effective resource planning.

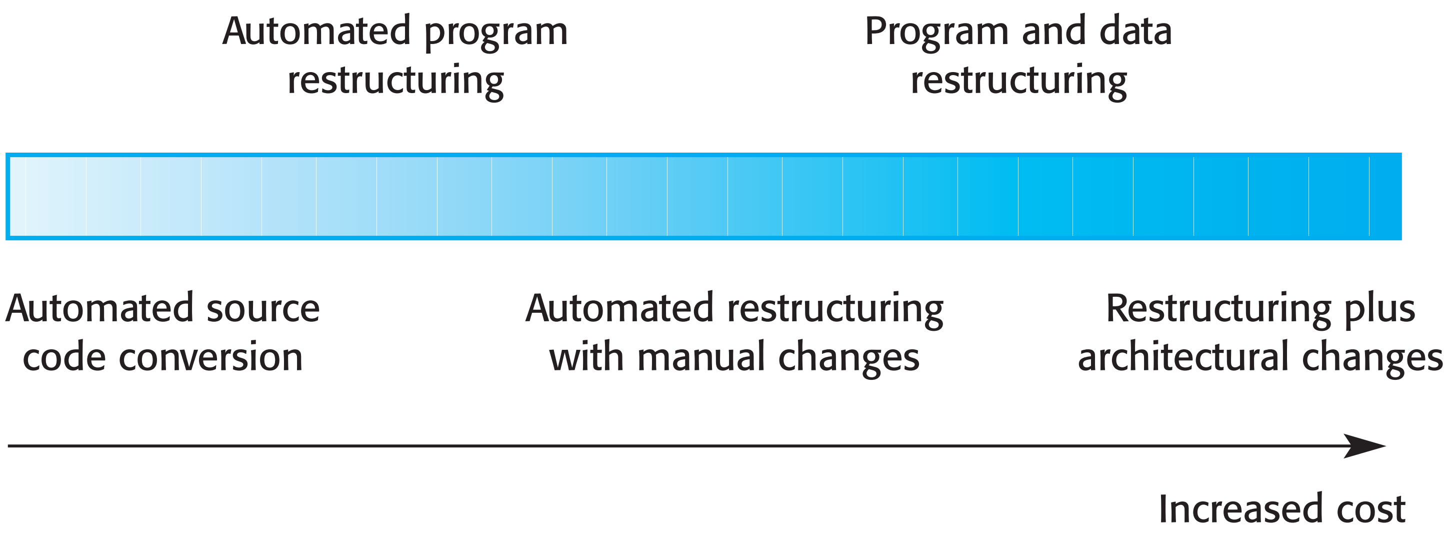
- Tightly coupled systems require changes whenever the environment is changed.
- Complexity drives maintenance costs, too.
  - Most maintenance effort is spent on a relatively small number of components.
  - We can measure complexity of our software.

# Software reengineering

- ✧ Restructuring or rewriting part or all of a legacy system without changing its functionality.
- ✧ Applicable where some but not all 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.



# Reengineering costs


Automated program restructuring
Program and data restructuring
Automated source code conversion
Automated restructuring with manual changes
Restructuring plus architectural changes
Increased cost

- 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 *fielded* systems

- Re-engineering 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.

# Classroom Activity – within your EMSS project teams

Activity:

Develop the high-level test plan for the EMSS noting what kind of tests will be performed at what point in the life-cycle of the software project. Also develop the high-level field maintenance plan for the EMSS noting how the software will be supported and evolved once it is released.

Submission to complete the Canvas Assignment EMSS Week 7:

Submit your test and evaluation plans and well as your post-release maintenance plan.