Software Quality

Week 10: Software Testing and Evolution

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING
& COMPUTER
SCIENCE

Characteristics of Software Quality

- Software has both external and internal quality characteristics. External characteristics are characteristics that a user of the software product is aware of, including the following:

How focusing on the factor below affects the factor to the right	Correctness	Usability	Efficiency	Reliability	Integrity	Adaptability	Accuracy	Robustness
Correctness	↑		↑	↑			↑	↓
Usability		↑				↑	↑	
Efficiency	↓		↑	↓	↓	↓	↓	
Reliability	↑			↑	↑		↑	↓
Integrity			↓	↑	↑			
Adaptability					↓	↑		↑
Accuracy	↑		↓	↑		↓	↑	↓
Robustness	↓	↑	↓	↓	↓	↑	↓	↑

Helps it ↑

Hurts it ↓

Characteristics of Software Quality

- External characteristics of quality are the only kind of software characteristics that users care about.
  - Users care about whether the software is easy to use, not about whether it's easy for you to modify.
  - They care about whether the software works correctly, not about whether the code is readable or well structured.
  - But programmers care about the internal characteristics of the software as well as the external ones.

Internal Quality Characteristics

- Programmers should care about internal as well as external quality characteristics:

1. **Maintainability**—The ease with which you can modify a software system to change or add capabilities, improve performance, or correct defects.
2. **Flexibility**—The extent to which you can modify a system for uses or environments other than those for which it was specifically designed.
3. **Portability**—The ease with which you can modify a system to operate in an environment different from that for which it was specifically designed.
4. **Reusability**—The extent to which and the ease with which you can use parts of a system in other systems.

Internal Quality Characteristics

5. **Readability**—The ease with which you can read and understand the source code of a system, especially at the detailed-statement level.
6. **Testability**—The degree to which you can unit-test and system-test a system; the degree to which you can verify that the system meets its requirements.
7. **Understandability**—The ease with which you can comprehend a system at both the system and statement levels. Understandability has to do with the coherence of the system at a more general level than readability does.

Techniques for Improving Software Quality

- Software quality assurance (QA) is a planned and systematic program of activities designed to ensure that a system has the desired characteristics.

Techniques for Improving Software Quality

- Software-quality objectives
  - One powerful technique for improving software quality is setting explicit quality objectives from among the external and internal characteristics described previously.
  - Without explicit goals, programmers might work to maximize characteristics different from the ones you expect them to maximize.
  - You might wonder whether, if you set explicit quality objectives, programmers will actually work to achieve them?

Setting Software-Quality Objectives

- Gerald Weinberg and Edward Schulman conducted a fascinating experiment to investigate the effect on programmer performance of setting quality objectives (1974).
  - They had five teams of programmers work on five versions of the same program.
  - The same five quality objectives were given to each of the five teams, and each team was told to optimize a different objective.
    - Minimize the memory required.
    - Produce the clearest possible output.
    - Build the most readable code.
    - Use the minimum number of statements.
    - Complete the program in the least amount of time possible.

Objective team was told to optimize	Minimum memory use	Most readable output	Most readable code	Least code	Minimum programming code
Minimum Memory	1	4	4	2	5
Output readability	5	1	1	5	3
Program readability	3	2	2	3	4
Least code	2	5	1	3	3
Minimum programming code	4	3	5	4	1

Source: Adapted from *Goals and Performance in Computer Programming* (Weinberg and Schulman 1974).

Setting Software Quality Objectives

- The surprising implication is that people actually do what you ask them to do.
  - Programmers have high achievement motivation.
  - They will work to the objectives specified, but they must be told what the objectives are.
- The second implication is that, as expected, objectives conflict, and it's generally not possible to do well on all of them.

Techniques for Improving Software Quality

- Explicit quality-assurance activity
  - One common problem in assuring quality is that quality is perceived as a secondary goal.
    - Indeed, in some organizations, quick and dirty programming is the rule rather than the exception.
    - In such organizations, it shouldn't be surprising that programmers don’t make quality their first priority.
    - The organization must show programmers that quality is a priority.

Techniques for Improving Software Quality

- Testing strategy
  - Testing can provide a detailed assessment of a product's reliability.
  - Part of quality assurance is developing a test strategy in conjunction with the product requirements, architecture, and design.
  - Developers on many projects rely on testing as the primary method of both quality assessment and quality improvement.

Techniques for Improving Software Quality

- Formal technical reviews
  - One part of managing a software-engineering process is catching problems at the “lowest-value” stage.
  - To achieve such a goal, developers use **quality gates**, periodic tests or reviews that determine whether the quality of the product at one stage is sufficient to support moving on to the next.
    - Quality gates are usually used to transition between requirements development and architecture, architecture and construction, and construction and system testing.
    - The “gate” can be an inspection, a peer review, a customer review, or an audit.

Quality Gates

- A “gate” does not mean that architecture or requirements need to be 100 percent complete or frozen.
  - It does mean that you will use the gate to determine whether the requirements or architecture are good enough to support downstream development.
    - **Good enough** might mean that you’ve sketched out the most critical 20 percent of the requirements or architecture, or it might mean you've specified 95 percent in excruciating detail.
    - Which end of the scale you should aim for depends on the nature of your specific project.

Techniques for Improving Software Quality

- Change-control procedures
  - Uncontrolled requirements changes can result in disruption to design and coding.
  - Uncontrolled changes in design can result in code that doesn't agree with its requirements and inconsistencies in the code.
  - Uncontrolled changes in the code itself can result in internal inconsistencies and uncertainties about which code has been fully reviewed and tested and which hasn't.
  - The natural effect of change is to destabilize and degrade quality, so handling changes effectively is a key to achieving high quality levels.

Techniques for Improving Software Quality

- Prototyping
  - Prototyping is the development of realistic models of a system's key features/functions.
  - A developer can prototype parts of a user interface to determine usability, critical calculations to determine execution time, or typical datasets to determine memory requirements.
  - A survey of 16 published and eight unpublished case studies revealed that prototyping can lead to better designs, better matches with user needs, and improved maintainability (Gordon and Bieman 1991).

Techniques for Improving Software Quality

- Measurement of results
  - Unless results of a quality-assurance plan are measured, you'll have no way to know whether the plan is working.
    - Measurement tells you whether your plan is a success or a failure and also allows you to vary your process in a controlled way to see how it can be improved.
    - You can also measure quality attributes themselves: correctness, usability, efficiency, …

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Software Testing Overview

Week 10: Software Testing and Evolution

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Program Testing

- 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.
  - When you test software, you execute a program using artificial data.
  - You check the results of the test run for errors, anomalies, or information about the program's nonfunctional attributes.
  - "Testing can only show the presence of errors, not their absence"—Edsger Dijkstra (Dijkstra et al. 1972).

Verification vs. Validation

- Testing is part of a more general verification and validation (V&V) process, which also includes **static validation techniques**.
- Validation: "Are we building the right product?”
  - The software should do what the user really requires.
- Verification: "Are we building the product right?”
  - The software should conform to its specification.

Program Testing Goals

1. To demonstrate to the developer and the customer that the software meets its requirements
   - For custom software, this means that there should be at least one test for every requirement in the requirements document.
   - For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification

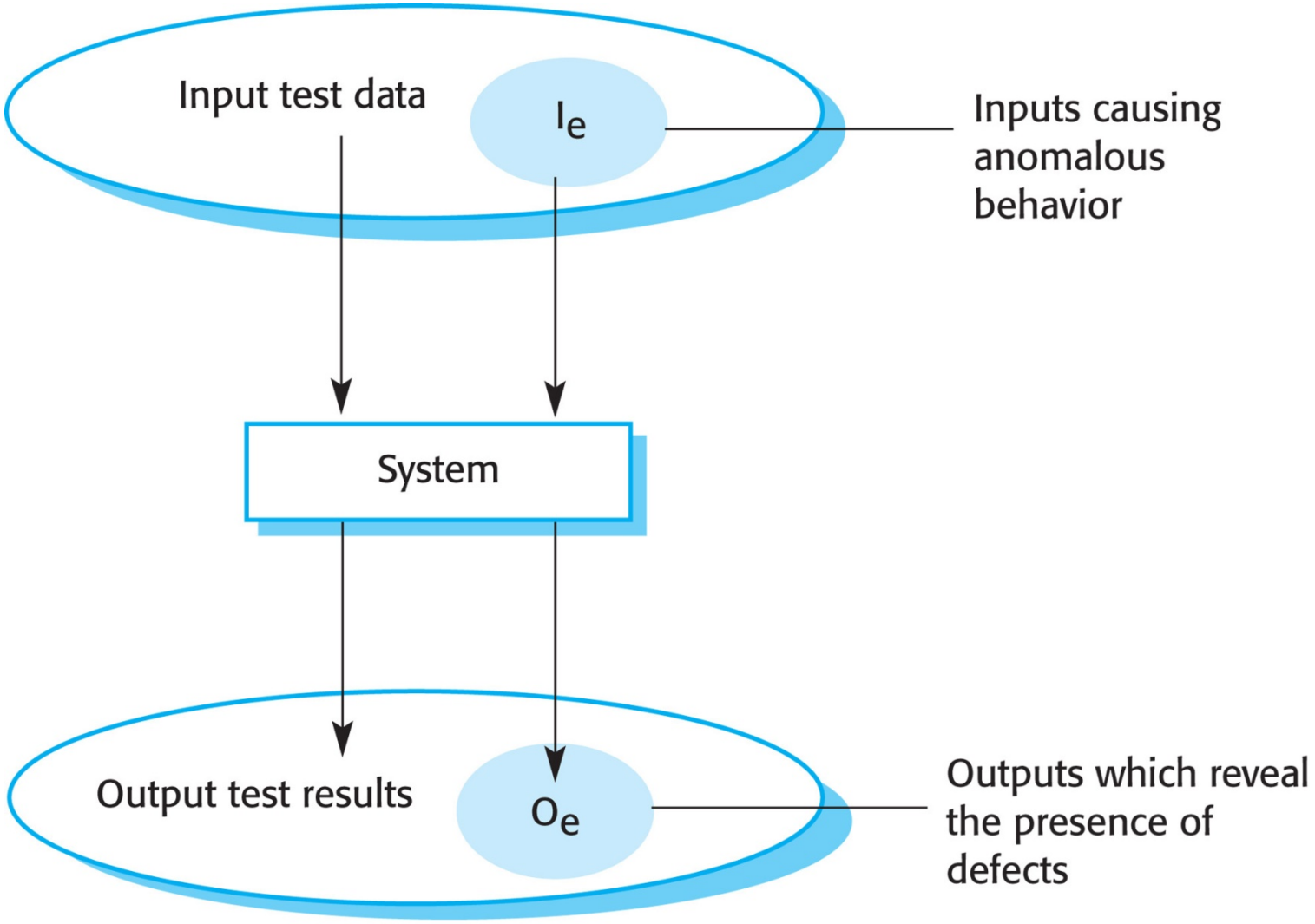
Validation vs. Defect Testing

- The first goal leads to **validation testing**.
  - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
  - To demonstrate to the developer and the customer that the software meets its requirements
  - A successful test shows that the system operates as intended.

Validation vs. Defect Testing

- The second goal leads to **defect testing**.
  - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.
  - The test cases are designed to expose defects.
  - The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

An I/O Model of Program Testing



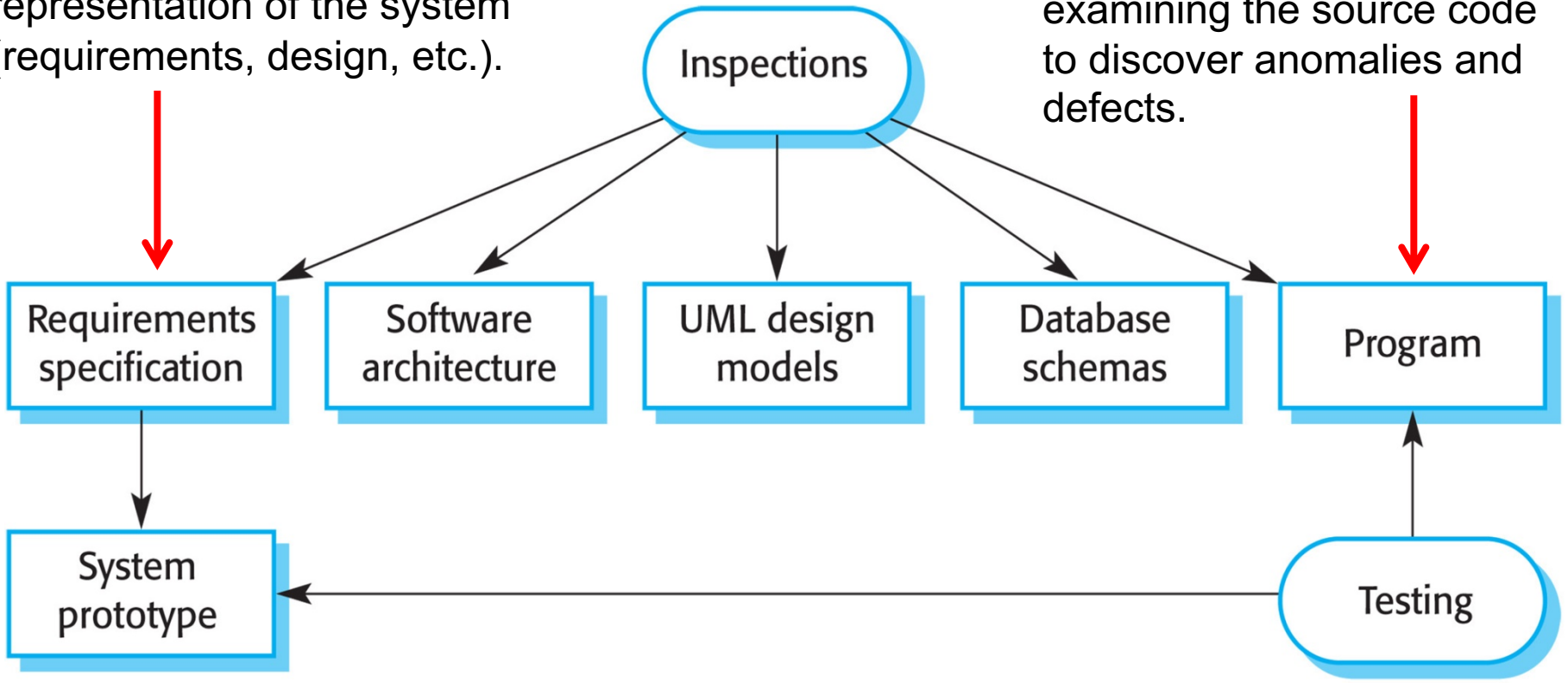
Inspections vs. Testing

- **Software inspections** (code reviews)
  - Concerned with analysis of the static system representation to discover problems—**static verification**
    - May be supplemented by tool-based document and code analysis
- **Software testing**
  - Concerned with exercising and observing product behavior—**dynamic verification**
    - The system is executed with test data and its operational behavior is observed.

Inspections vs. Testing

These may be applied to any representation of the system (requirements, design, etc.).

These involve people examining the source code to discover anomalies and defects.



Inspections have been shown to be an effective technique for discovering program errors.

Advantages of Inspections

- 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, then you need to develop specialized test harnesses to test the parts that are available.
  - As well as searching for program defects, an inspection can also consider broader quality attributes of a program.
    - You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

Advantages of Inspections

- Fagan (1986) reported that more than **60 percent** of the errors in a program can be detected using informal program inspections.

- In the cleanroom process (Prowell et al. 1999), it is claimed that more than **90 percent** of defects can be discovered in program inspections.

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item

Cleanroom software engineering

From Wikipedia, the free encyclopedia

This article is about the software development methodology. For the method used to avoid copyright infringement, see Clean room design.

The **cleanroom software engineering** process is a software development process intended to produce software with a certifiable level of reliability. The cleanroom process was originally developed by Harlan Mills and several of his colleagues including Alan Hevner at IBM.[1] The focus of the cleanroom process is on defect prevention, rather than defect removal. The name "cleanroom" was chosen to invoke the cleanrooms used in the electronics industry to prevent the introduction of defects during the fabrication of semiconductors. The cleanroom process first saw use in the mid to late 80s. Demonstration projects within the military began in the early 1990s.[2] Recent work on the cleanroom process has examined fusing cleanroom with the automated verification capabilities provided by specifications expressed in CSP.[3]

Software development process

Core activities
Requirements · Design · Construction · Testing · Debugging · Deployment · Maintenance

Paradigms and models
Software engineering · Waterfall · Prototyping · Incremental · V-Model · Dual Vee Model · Spiral · IID · Agile · Lean · DevOps

Methodologies and frameworks
Cleanroom · TSP · PSP · RAD · DSDM · MSF · Scrum · Kanban · UP · XP · TDD · ATDD · BDD · FDD · DDD · MDD · SAFe

Supporting disciplines
Configuration management · Infrastructure as Code · Documentation · Software Quality assurance (SQA) · Project management · User experience

Disadvantages of Inspections

- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check nonfunctional characteristics such as performance, usability, and so on.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Development Testing

Week 10: Software Testing and Evolution

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Stages of Software Testing

- **Development testing**, where the system is tested during development to discover bugs and 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.

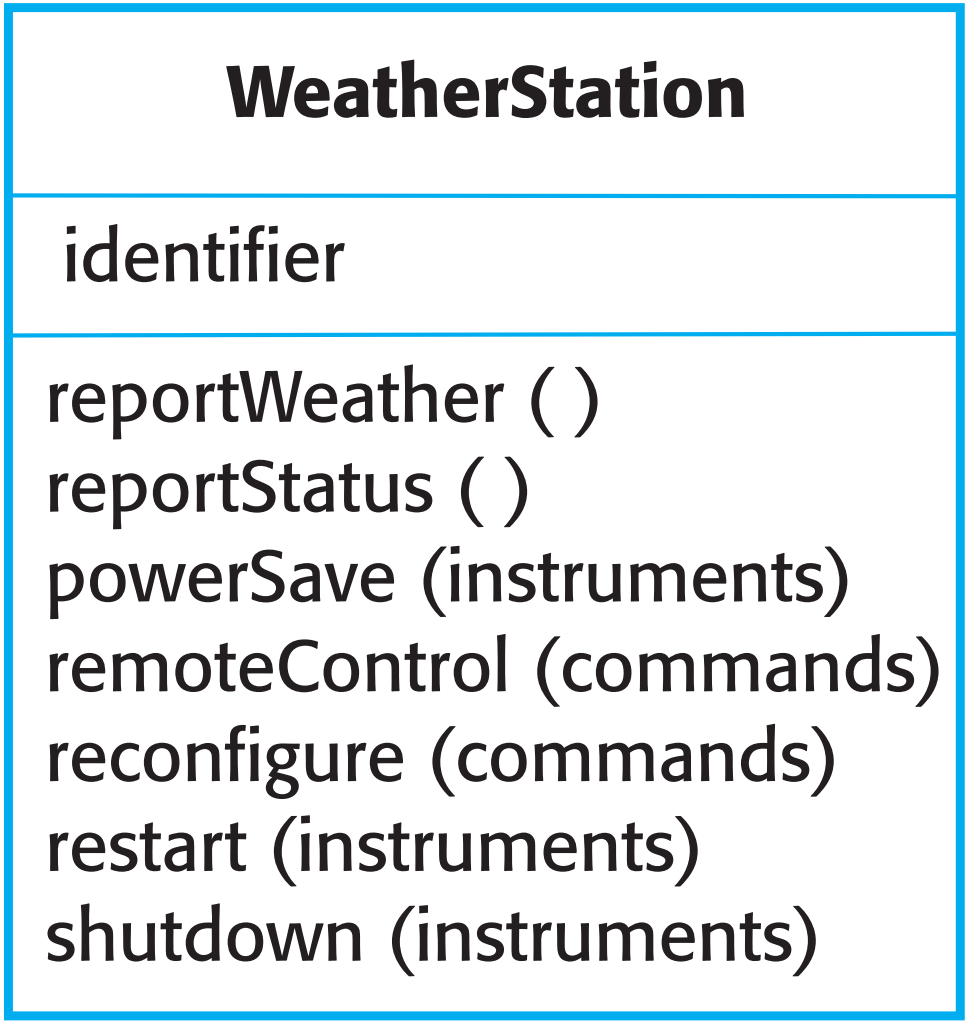
Development Testing

- Development testing includes all testing activities that are carried out by the development team.
  - **Unit testing**
    - Individual program units or object classes are tested.
    - Unit testing should focus on testing the **functionality of objects or methods**.
  - **Component testing**
    - Several individual units are integrated to create composite components.
    - Component testing should focus on testing **component interfaces**.
  - **System testing**
    - Some or all of the components in a system are integrated, and the system is tested as a whole.
    - System testing should focus on testing **component interactions**.

Unit Testing

- Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite objects with defined interfaces used to access their functionality
- 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.

The Weather Station Object Interface



Weather Station Testing

- Need to define test cases for reportWeather, calibrate, test, startup, and shutdown.
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions.
- For example:
  - Shutdown → running → shutdown
  - Configuring → running → testing → transmitting → running
  - Running → collecting → running → summarizing → transmitting → running

Automated Testing

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
  - In automated unit testing, you make use of a test-automation framework (such as JUnit) to write and run your program tests.
  - Unit testing frameworks provide generic test classes that you extend to create specific test cases.
  - They can then run all of the tests that you have implemented and report, often through some GUI, on the success or otherwise of the tests.

Unit Test Effectiveness

- 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.
- This leads to two 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.

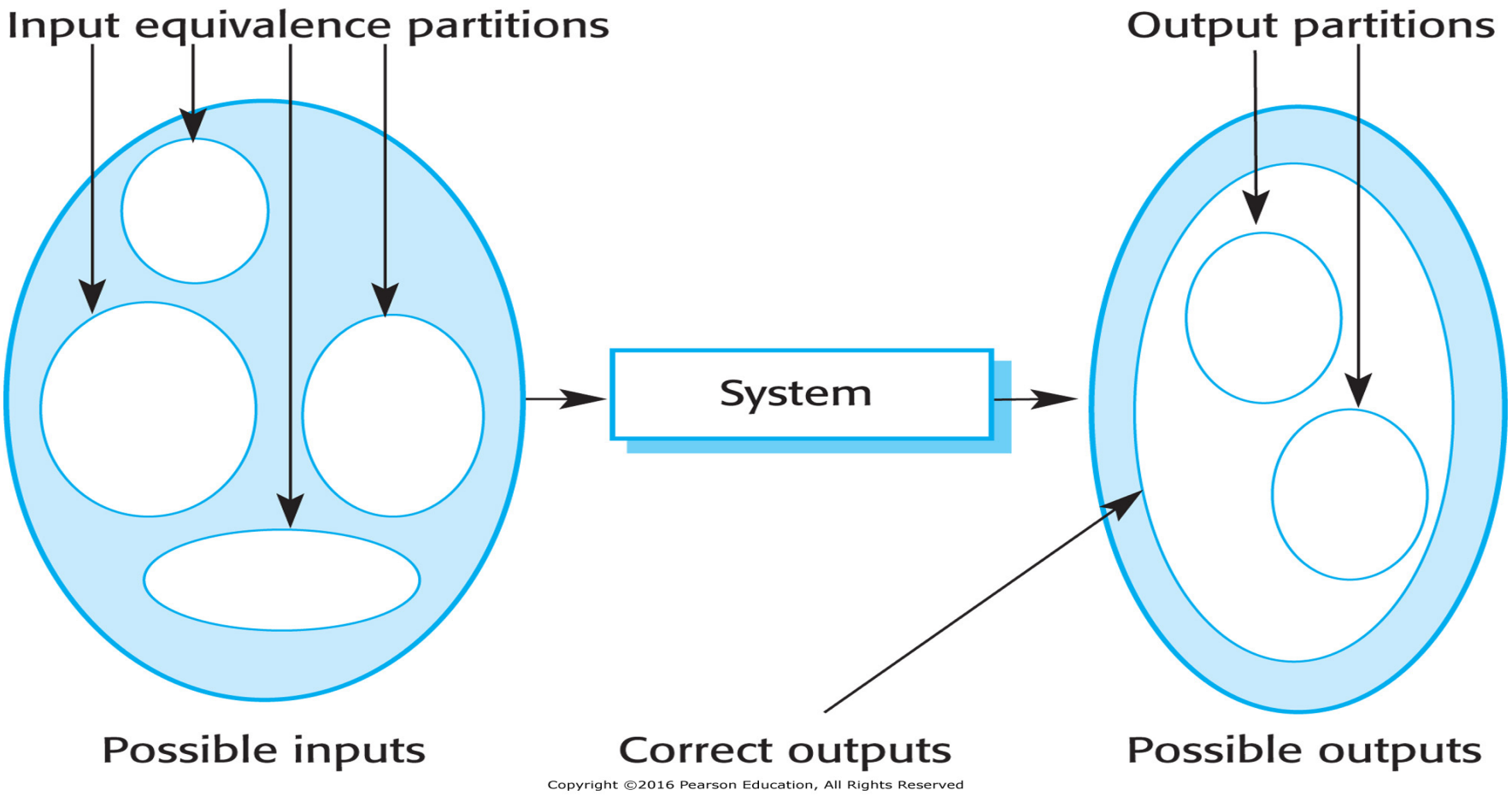
Testing Strategies

- **Partition testing**
  - You identify groups of inputs that have common characteristics and should be processed in the same way.
  - You should choose tests from within each of these groups.
- **Guideline-based testing**
  - You use testing guidelines to choose test cases.
  - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Partition Testing

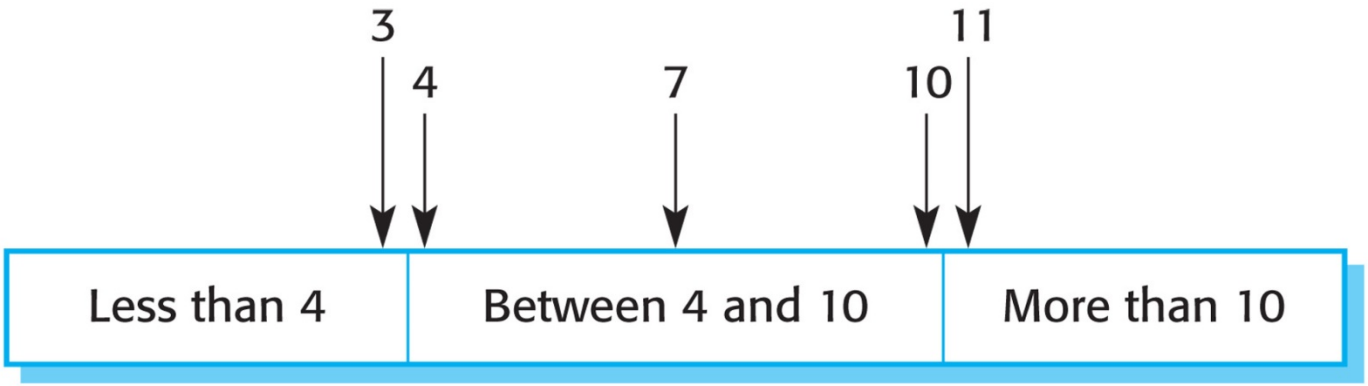
- Input data and output results often fall into different classes where all members of a class are related.
  - Each of these classes is an **equivalence partition** where the program behaves in an equivalent way for each class member.
  - Test cases should be chosen from each partition.

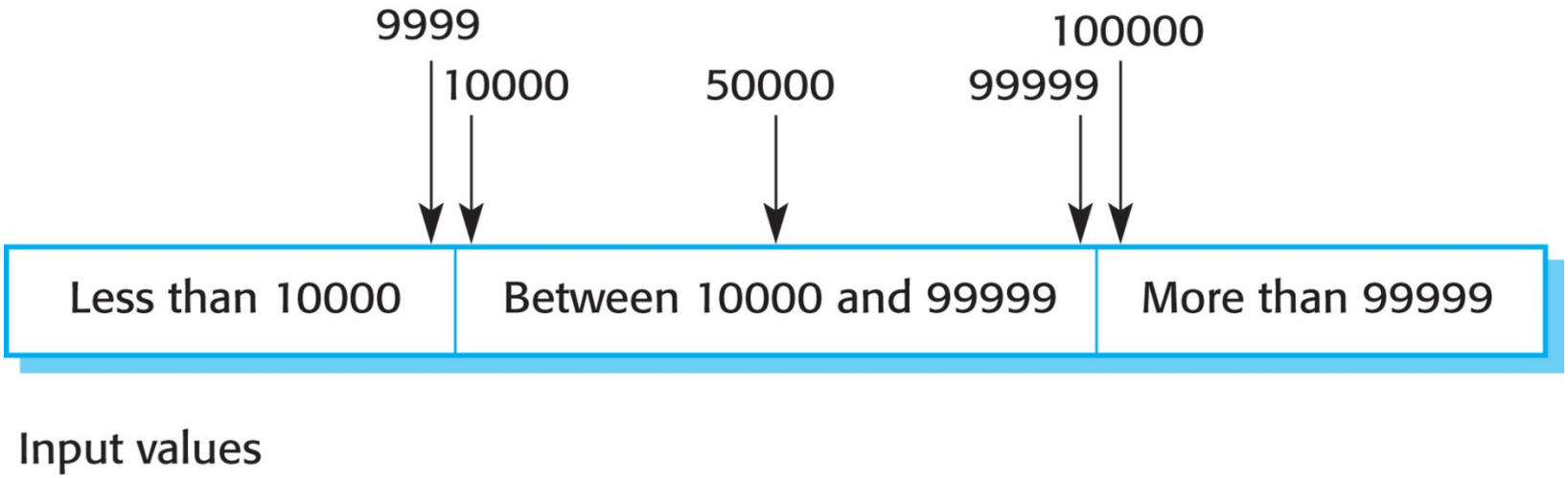
Equivalence Partition



Test cases should be chosen from each partition.

Equivalence Partition



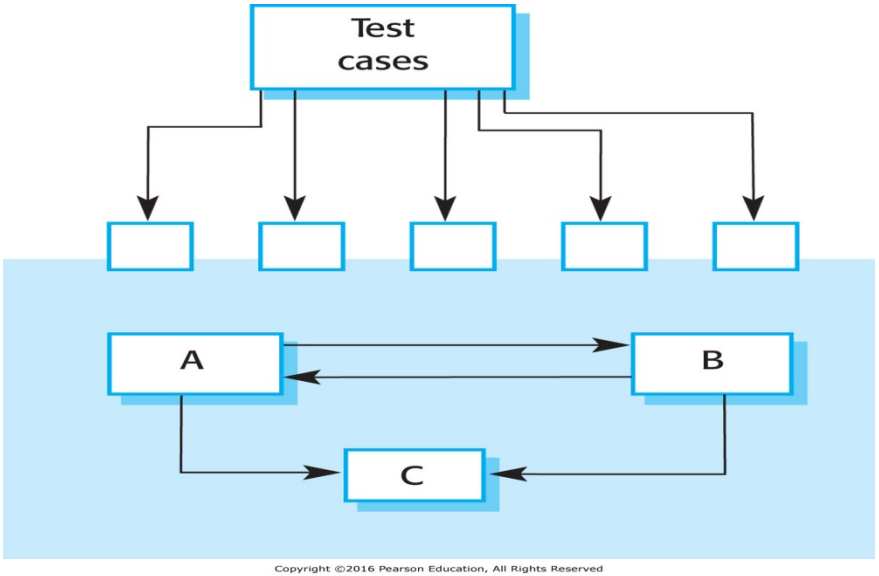


General Testing Guidelines

- Whittaker's book, ***How to Break Software: A Practical Guide to Testing*** (2002), includes many examples of guidelines that can be used in test case design.
  - 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

- Software components are often composite components that are made up of several interacting objects.
- You access the functionality of these objects through the defined component interface.



- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - You can assume that unit tests on the individual objects within the component have been completed.

Interface Testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Interface types
  - **Parameter interfaces**: Data passed from one method or procedure to another.
  - **Shared memory interfaces**: Block of memory is shared between procedures or functions.
  - **Procedural interfaces**: Subsystem encapsulates a set of procedures to be called by other subsystems.
  - **Message passing interfaces**: Subsystems request services from other subsystems.

Common Interface Errors

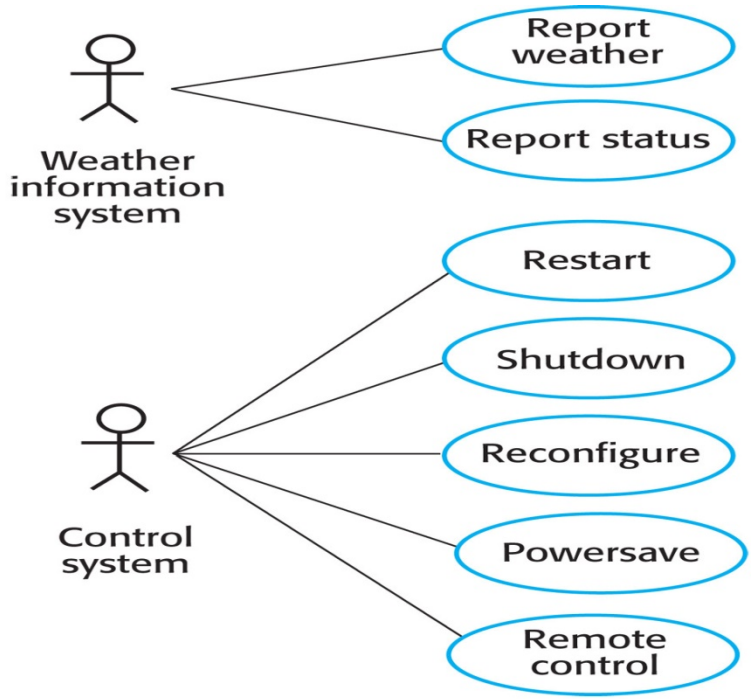
- **Interface misuse**
  - A calling component calls another component and makes an error in its use of its interface, e.g., parameters in the wrong order.
- **Interface misunderstanding**
  - A calling component embeds assumptions about the behavior of the called component that are incorrect, e.g., a binary search method may be called with a parameter that is an unordered array. The search would then fail.
- **Timing errors**
  - The called and the calling component operate at different speeds and out-of-date information is accessed.

System Testing

- System testing during development involves integrating 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.

System Testing: Use-Case Testing

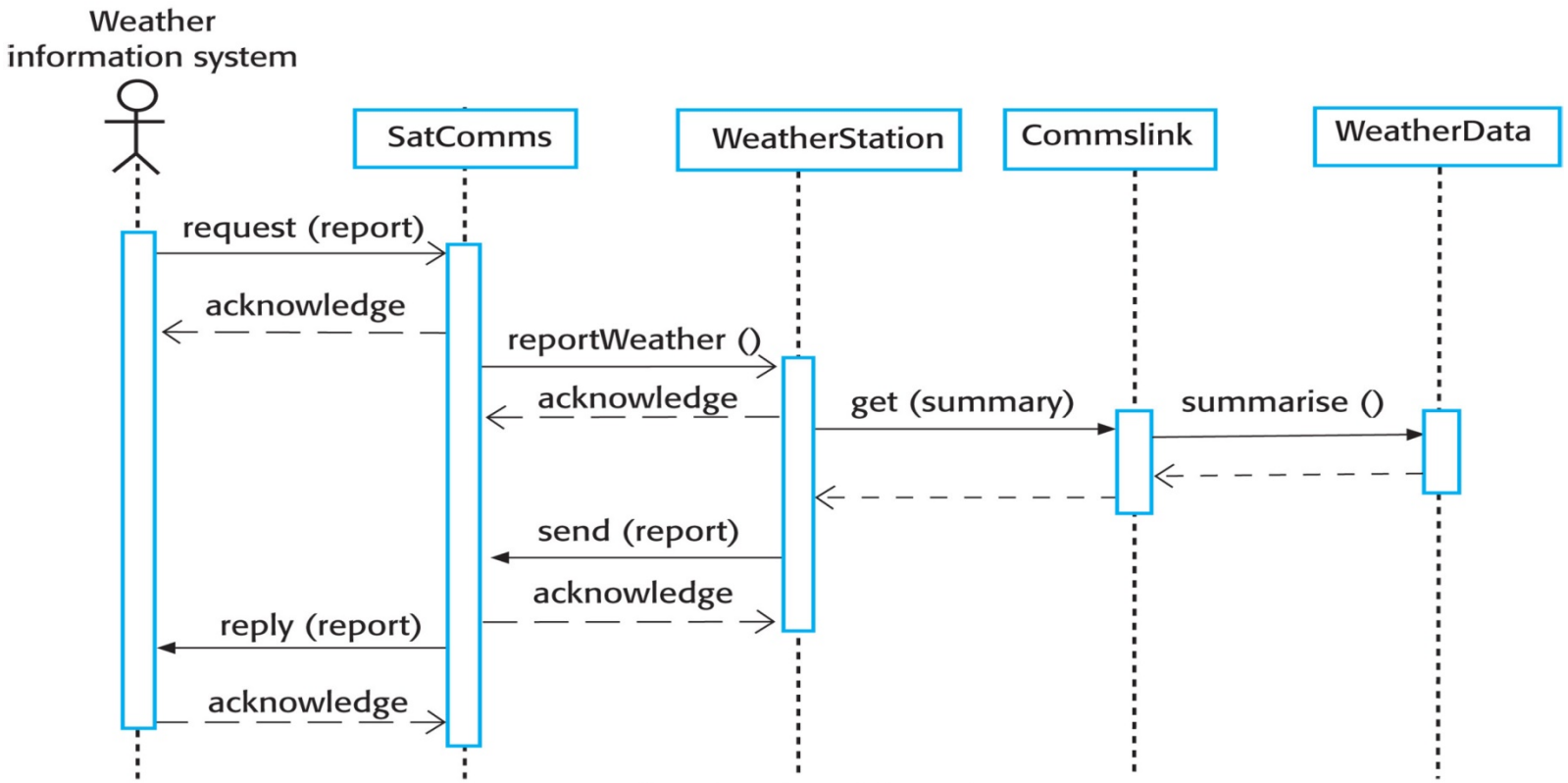
- The **use-case diagrams** developed to identify system interactions can be used as a basis for system testing.
  - Each use case usually involves several system components so testing the use case forces these interactions to occur.



System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future.

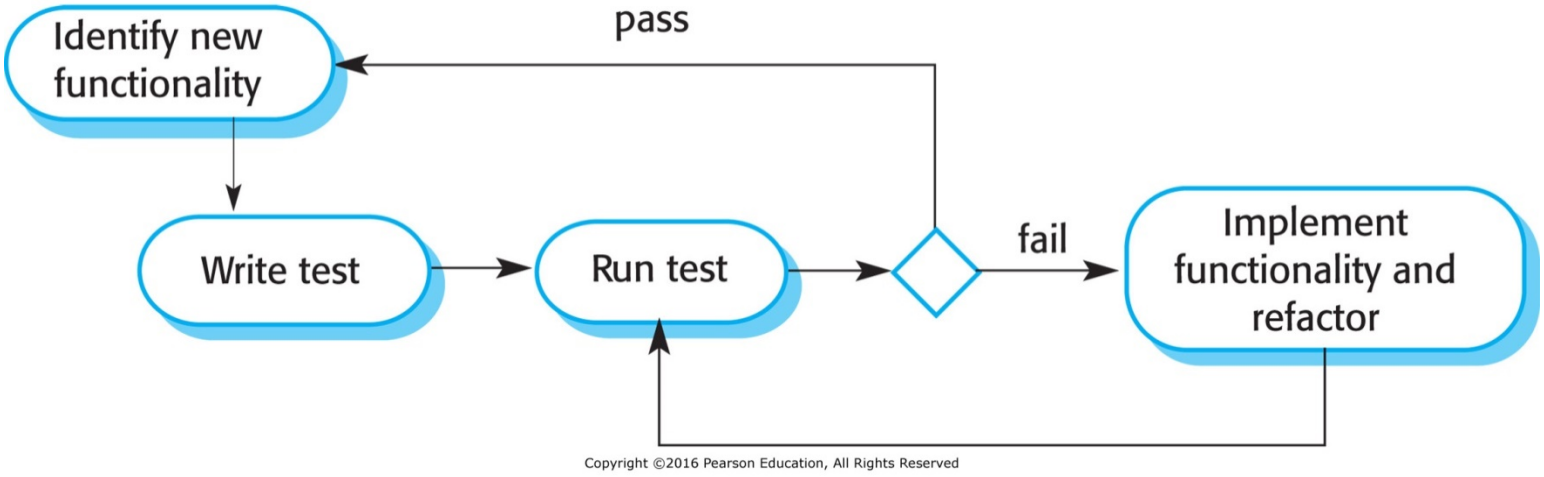
System Testing: Use-Case Testing

- The **sequence diagrams** associated with the use-case diagrams can be used as a basis for system testing as well. The components and interactions in the sequence diagrams can be tested.



Test-Driven Development

- Test-driven development (TDD) is an approach to program development in which you interleave testing and code development.
  - Tests are written **before** code and "passing" the tests is the critical driver of development.
  - TDD was introduced as part of agile methods such as **extreme programming**, but it can also be used in plan-driven development processes.



Regression Testing

- Regression testing is testing the system to check that changes have not “broken” previously working code.
  - In a manual testing process, regression testing is expensive, but with automated testing, it is simple and straightforward.
    - All tests are rerun every time a change is made to the program.
  - Tests must run “successfully” before the change is committed.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Release Testing and User Testing

Week 10: Software Testing and Evolution

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING
& COMPUTER
SCIENCE

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 sponsor 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 derived only from the system specification.

White-box testing

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.

(February 2013)

White-box testing (also known as **clear box testing**, **glass box testing**, **transparent box testing**, and **structural testing**) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT). White-box testing can be applied at the unit, integration and system levels of the software testing process. Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently today. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it has the potential to miss unimplemented parts of the specification or missing requirements.

White-box test design techniques include the following code coverage criteria:

- Control flow testing
- Data flow testing
- Branch testing
- Statement coverage
- Decision coverage
- Modified condition/decision coverage
- Prime path testing
- Path testing

Release Testing vs. System Testing

- Release testing is a form of system testing.
- Important differences:
  - A separate team that has not been involved in the system development should be responsible for release testing.
  - System testing by the development team should focus on discovering bugs in the system (defect testing).
  - The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

Performance Testing

- Part of release testing may involve testing the emergent (nonfunctional) properties of a system, such as **performance** and **reliability**.
  - Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable. → stress testing
    - **Stress testing** is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

User Testing

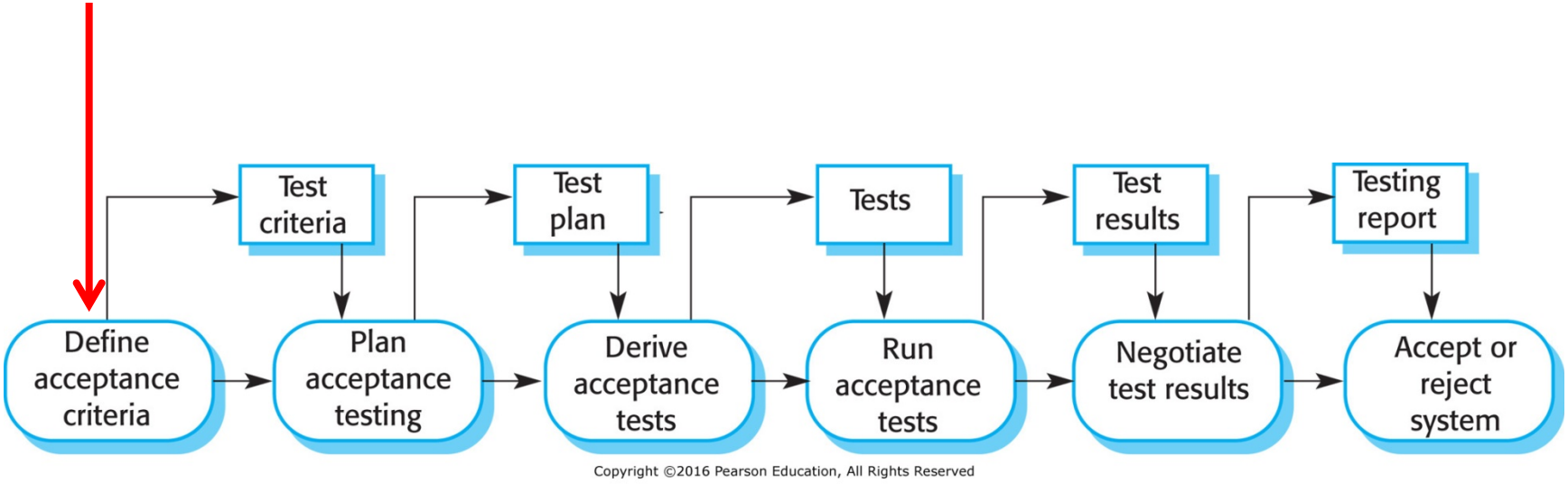
- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
  - User testing is essential, even when comprehensive system and release testing have been carried out.
    - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability, and robustness of a system. These cannot be replicated in a testing environment.

Types of User Testing

- **Alpha testing**
  - Users of the software work with the development team to test the software at the developer's site.
- **Beta testing**
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- **Acceptance testing** (next slides)
  - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.
    - Primarily for custom systems

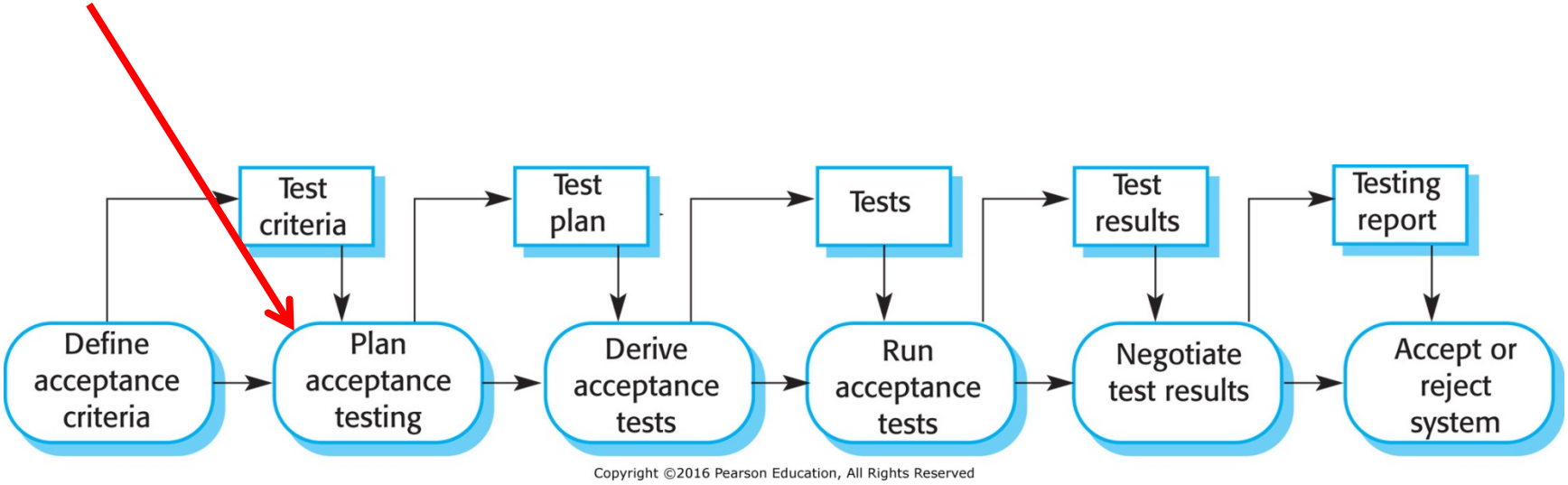
Acceptance Testing

This stage should, ideally, take place before the contract for the system is signed. The acceptance criteria should be part of the system contract.



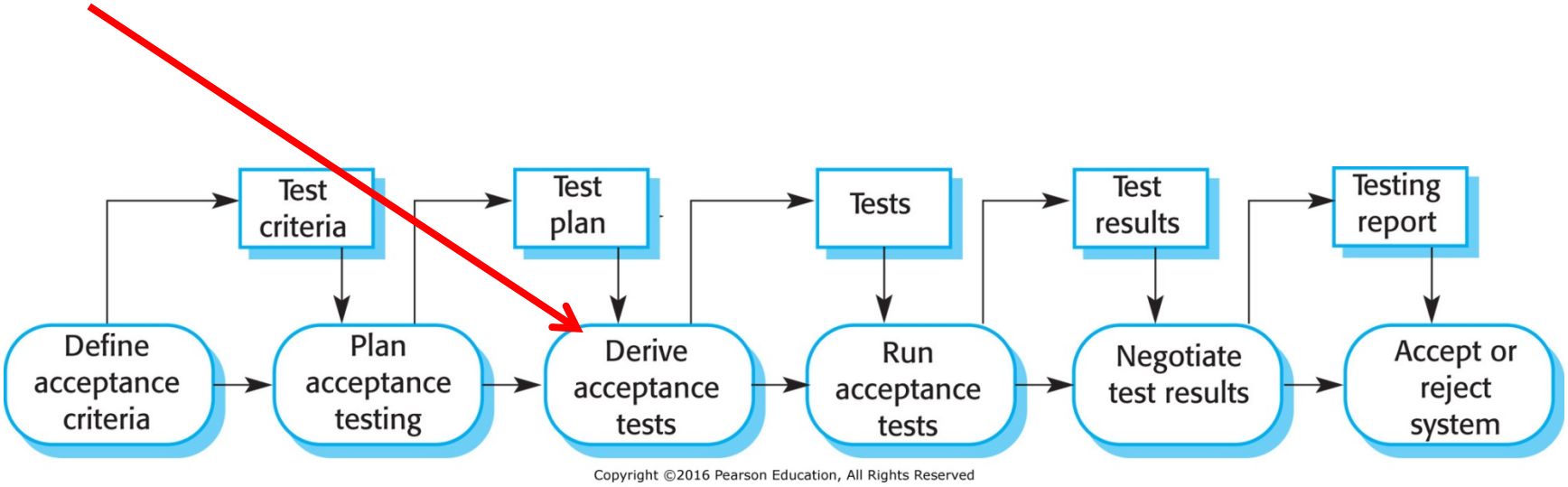
Acceptance Testing

This involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule.



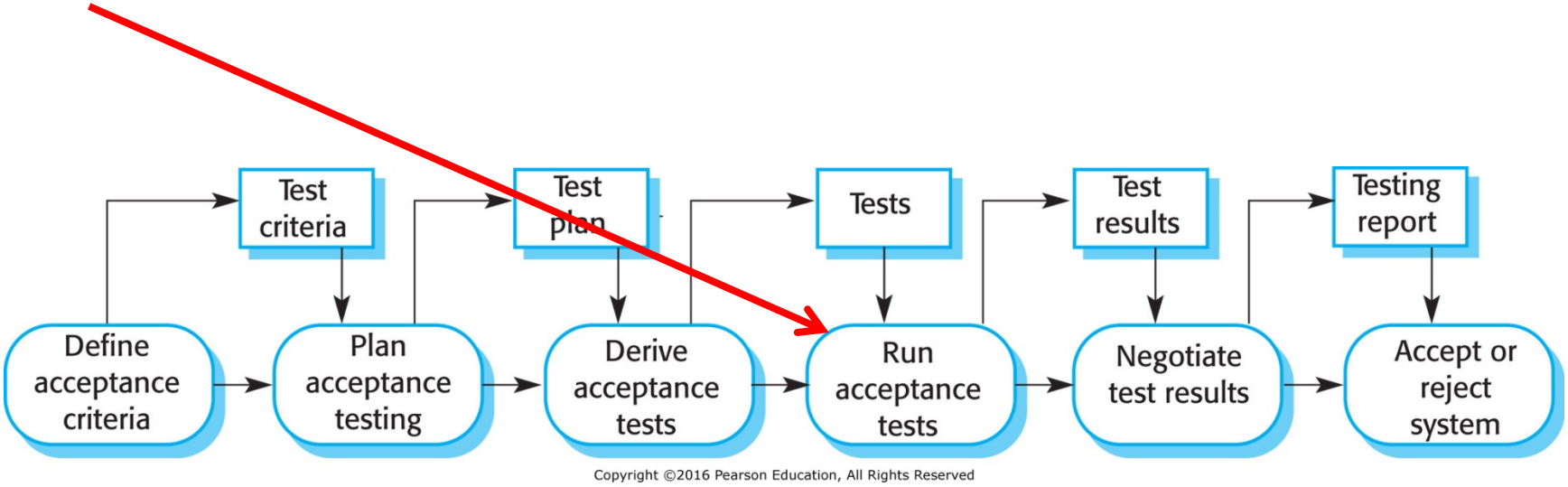
Acceptance Testing

Acceptance tests should aim to test both the functional and nonfunctional characteristics of the system and should, ideally, provide complete coverage of the system requirements.



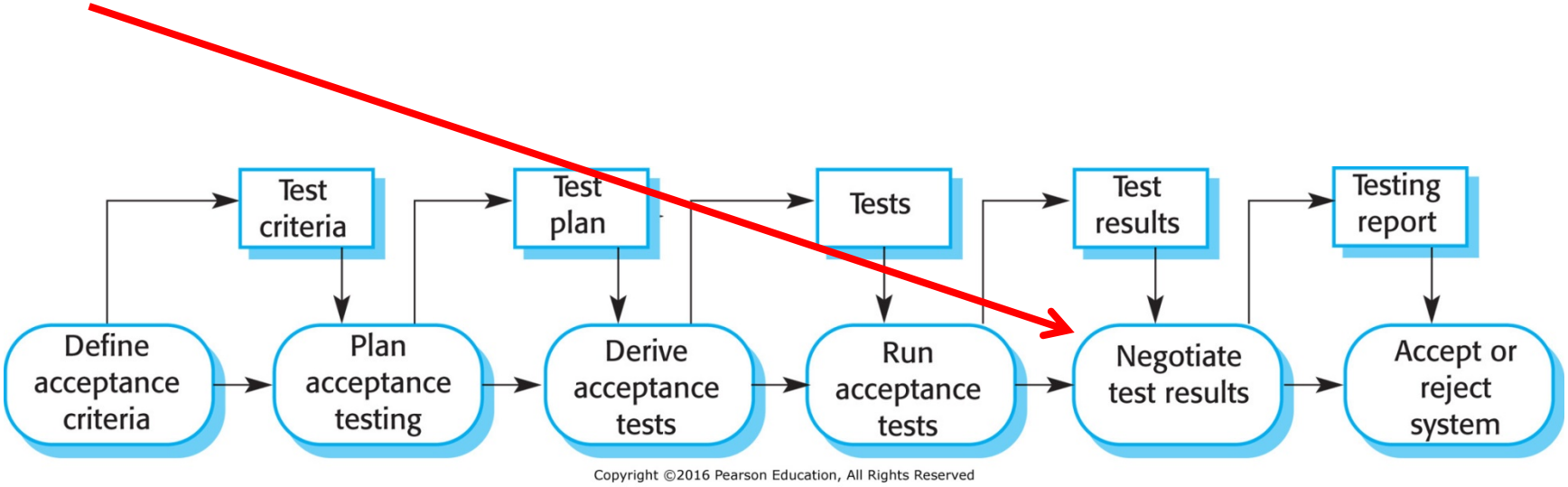
Acceptance Testing

Ideally, this should take place in the actual environment where the system will be used, but this may be disruptive and impractical.



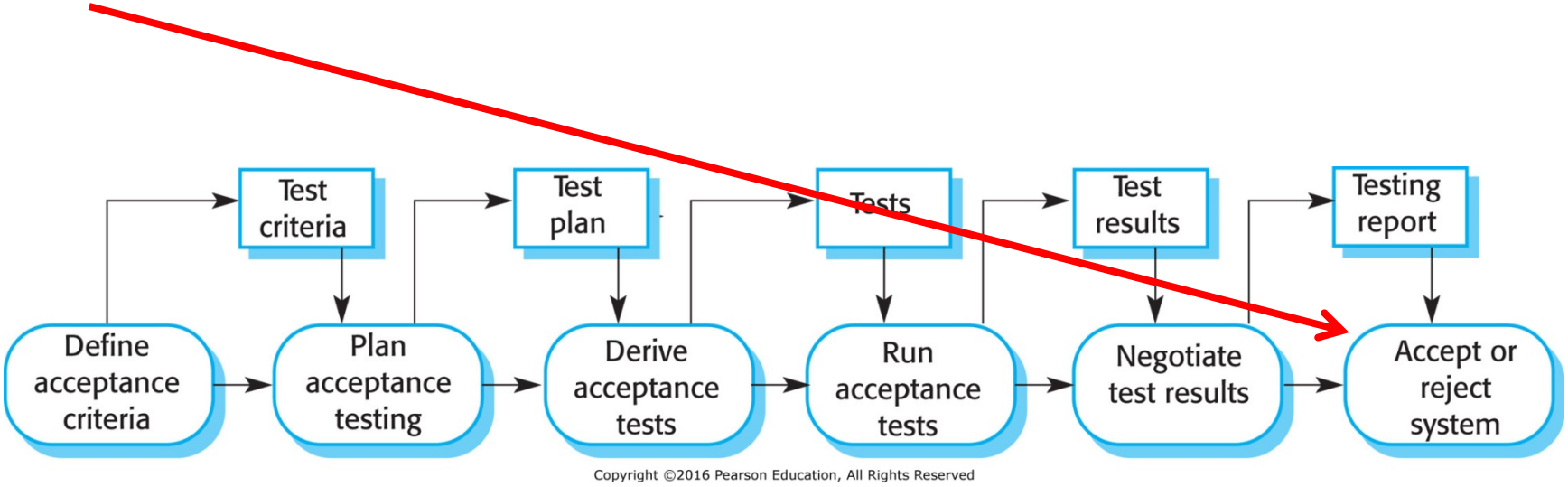
Acceptance Testing

The developer and the customer have to negotiate to decide if the system is good enough to be put into use.



Acceptance Testing

This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted.



SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Effectiveness of Quality Techniques

Week 10: Software Testing and Evolution

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

Effectiveness of Quality Techniques

- Many QA techniques have been studied, and their **effectiveness** at detecting and removing defects is known.
- One way to evaluate defect-detection methods is to determine the **percentage of defects** they detect out of the total defects that exist at that point in the project.
- Some practices are better at detecting defects than others, and different methods find different kinds of defects.

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

The most interesting fact that this data reveals is that the modal rates don't rise above **75** percent for any single technique and that the techniques average about 40 percent.

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

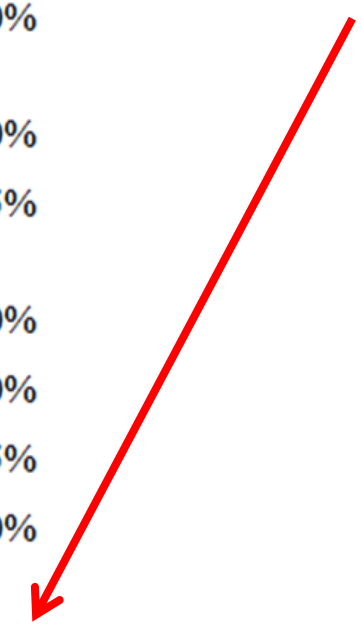
Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

For the most common kinds of defect detection—unit testing and integration testing—the modal rates are only 30 percent to 35 percent.

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

The typical organization uses a test-heavy defect-removal approach and achieves only about **85** percent.



Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

Leading organizations use a wider variety of techniques and achieve defect-removal efficiencies of **95 percent** or higher (Jones 2000).

Effectiveness of Quality Techniques

- Moral of the story
  - If project developers are striving for a higher defect-detection rate, they need to use a combination of techniques.

Effectiveness of Quality Techniques

- Studies at NASA's Software Engineering Laboratory, Boeing, and other companies have reported that different people tend to find different defects.
  - Only about **20** percent of the errors found by inspections were found by more than one inspector

    (Kouchakdjian, Green, and Basili 1989; Tripp, Struck, and Pflug 1991; Schneider, Martin, and Tsai 1992).

Effectiveness of Quality Techniques

- These studies can also be used to understand why programmers who begin working with a disciplined defect-removal technique such as **extreme programming** experience higher defect-removal levels than they have experienced previously.

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews (pair programming)	25%	35%	40%
Informal code reviews (pair programming)	20%	25%	35%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
Expected cumulative defect-removal efficiency	~74%	~90%	~97%

Effectiveness of Quality Techniques

- Although some people have linked this effectiveness to synergy among XP's practices, it is really just a predictable outcome of using these specific defect-removal practices.
- Other combinations of practices can work equally well or better, and the determination of which specific defect-removal practices to use to achieve a desired quality level is one part of effective project planning.

The General Principle of Software Quality

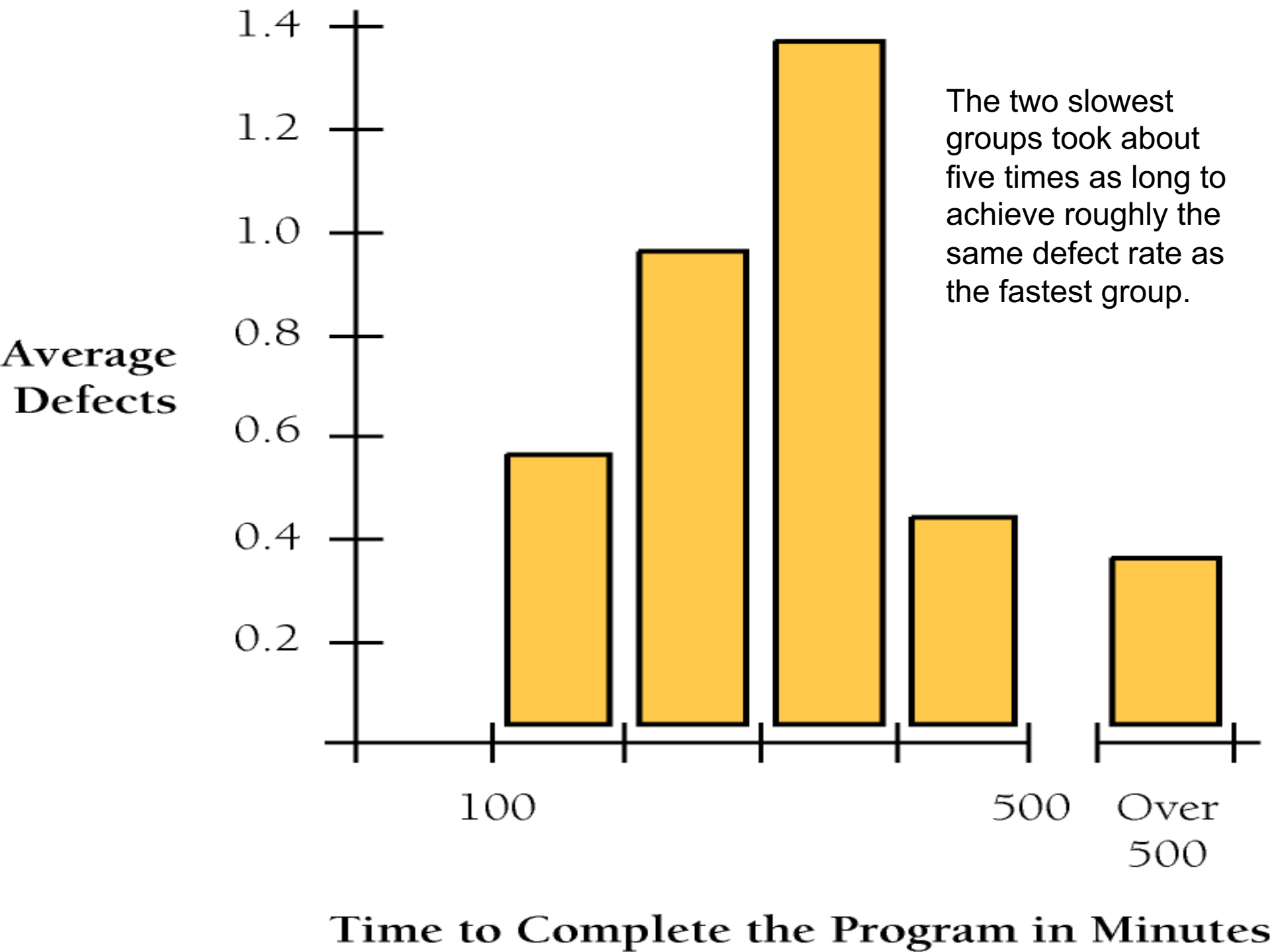
- The single biggest activity on most projects is debugging/bug fixing.
  - Debugging and associated **refactoring** and other rework consume about **50** percent of the time using a traditional software development process.
  - Hence, the most obvious method of shortening a software development schedule is to decrease the amount of time spent debugging and reworking the software.

The General Principle of Software Quality

- In a review of 50 development projects involving over 400 work-years of effort and almost 3 million lines of code, a study at NASA's Software Engineering Laboratory found that increased quality assurance was associated with decreased error rate but did not increase overall development cost.

The General Principle of Software Quality

- The same effect holds true at the small end of the scale.
  - In a 1985 study, 166 professional programmers wrote programs from the same specification.
    - The resulting programs averaged 220 lines of code and a little under five hours to write.
    - The fascinating result was that programmers who took the median time to complete their programs produced programs with the greatest number of errors.
    - The programmers who took more or less than the median time produced programs with significantly fewer errors (DeMarco and Lister 1985).


1.4
1.2
1.0
0.8
0.6
0.4
0.2
Average Defects
100
500
Over 500
Time to Complete the Program in Minutes
The two slowest groups took about five times as long to achieve roughly the same defect rate as the fastest group.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Software Change

Week 10: Software Testing and Evolution

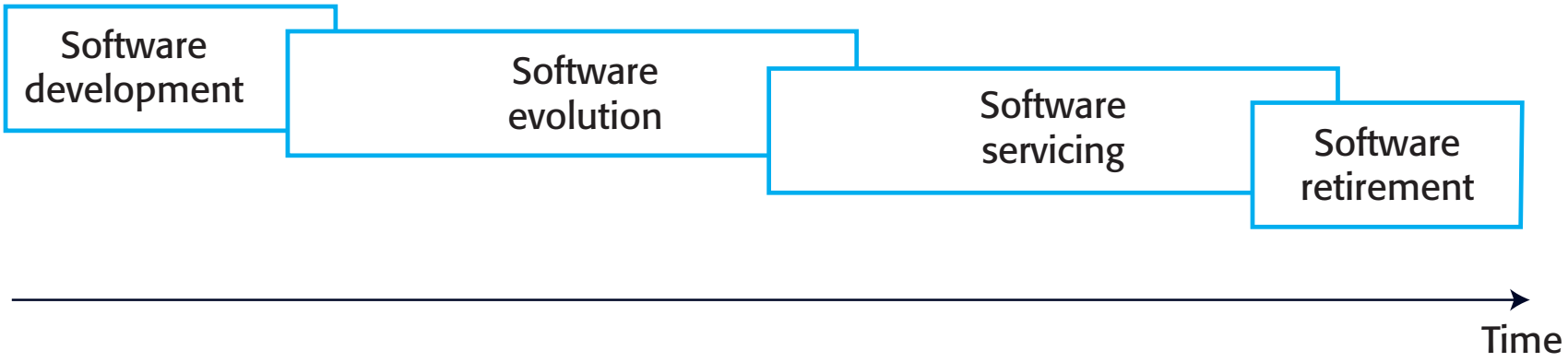
Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Software Change

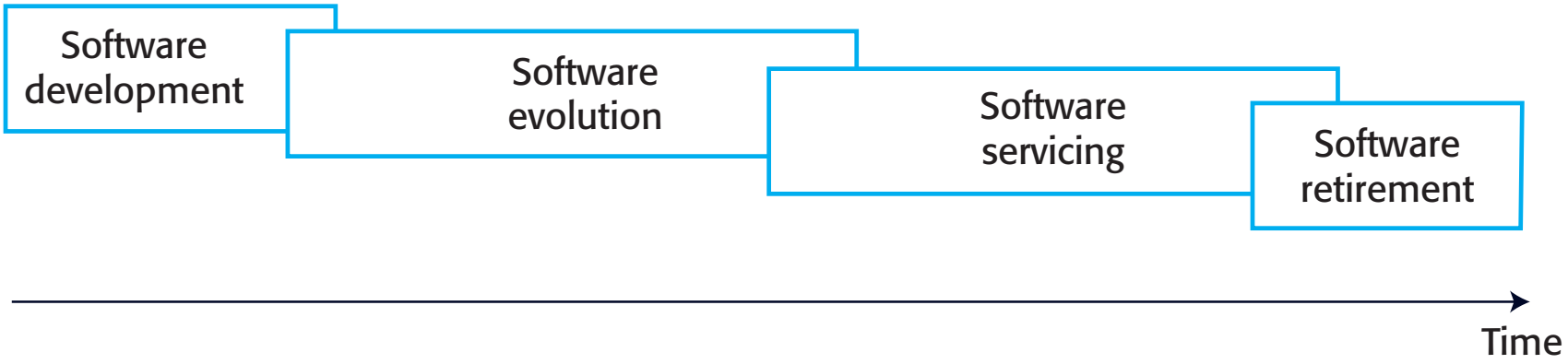
- Software change is inevitable.
  - New requirements emerge when the software is used.
  - The business environment changes.
  - Errors must be repaired.
  - New computers and equipment are added to the system.
  - The performance or reliability of the system may have to be improved.
- A key problem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

Evolution and Servicing



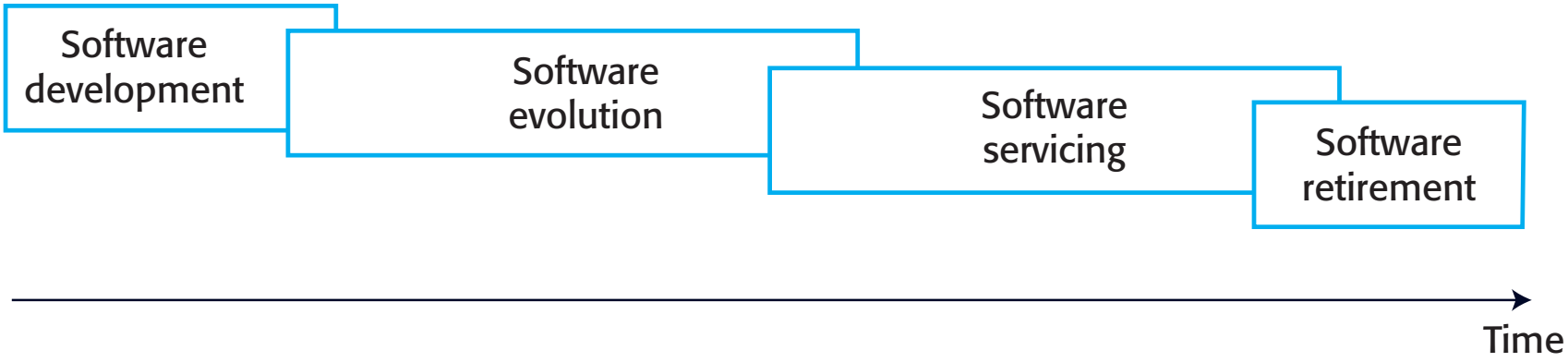
- Evolution
  - The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.

Evolution and Servicing



- 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.

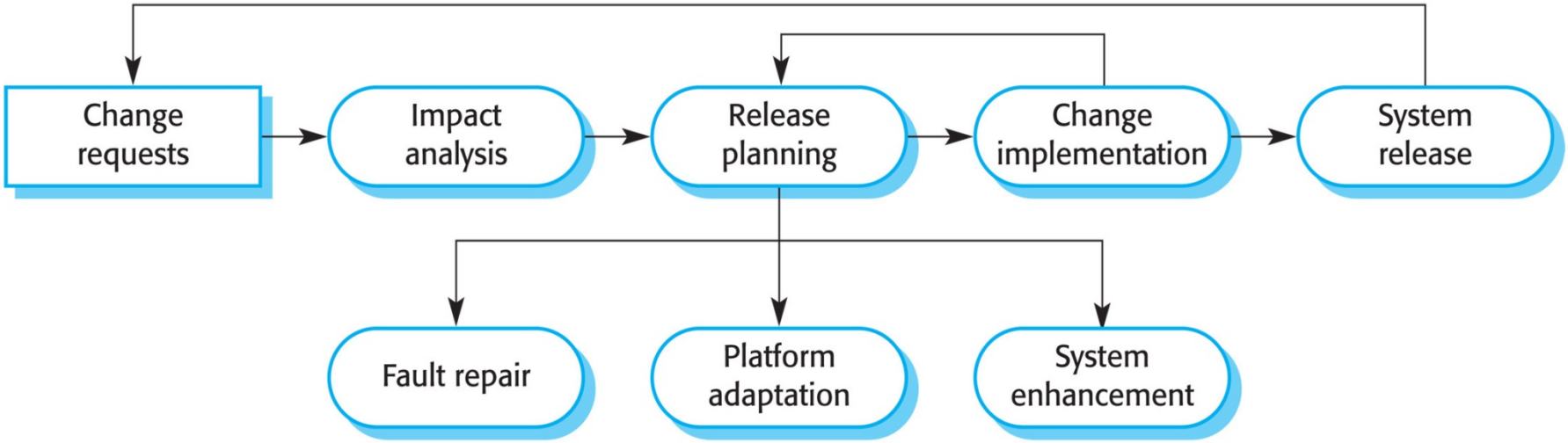
Evolution and Servicing



- Phase-out
  - The software may still be used, but no further changes are made to it.

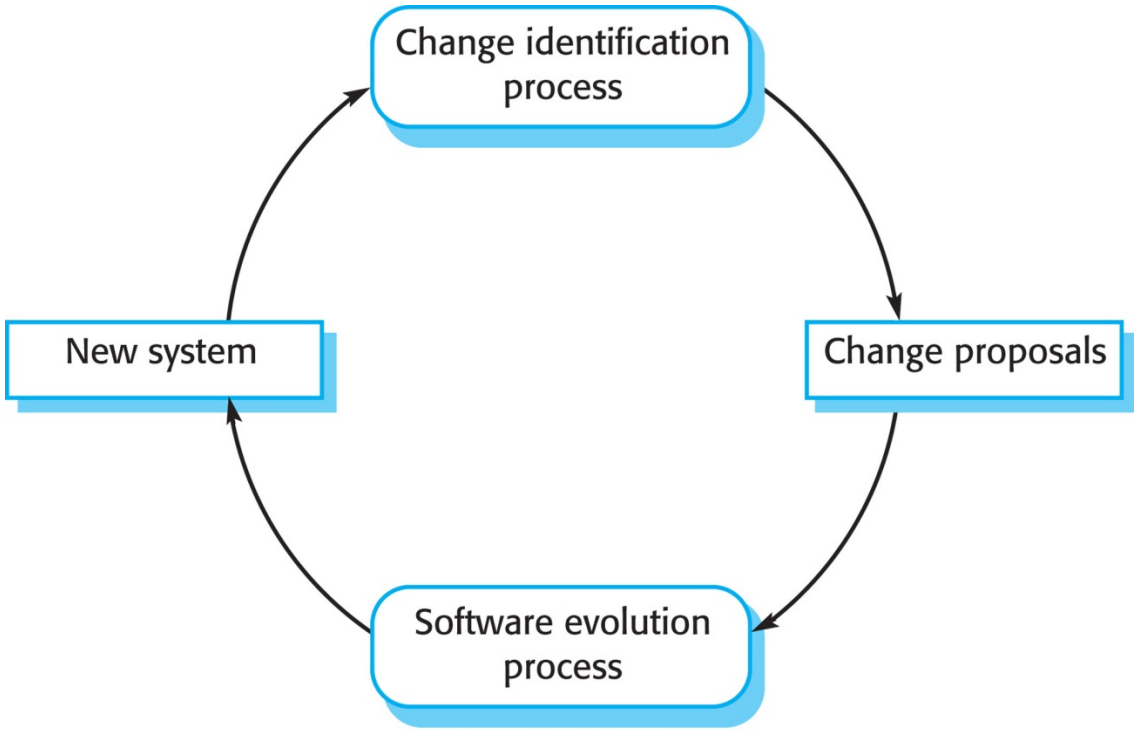
Evolution Processes

- Software evolution processes depend on
  - The type of software being maintained
  - The development processes used
  - The skills and experience of the people involved



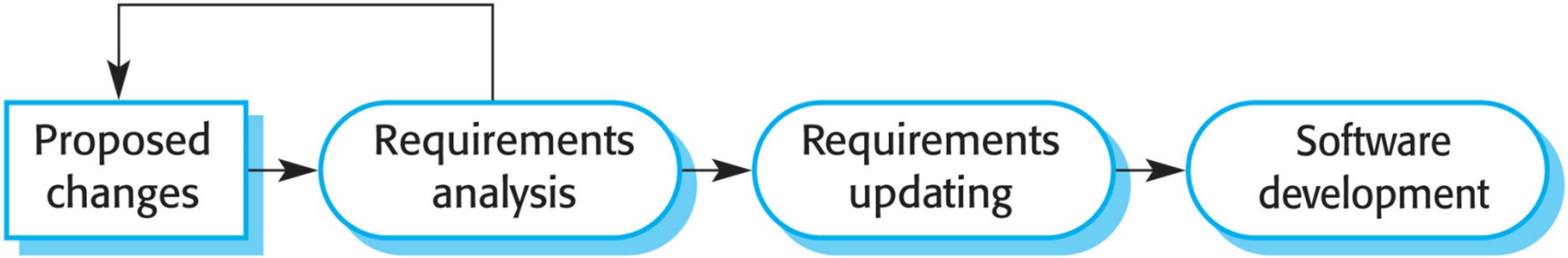
Evolution Processes

- 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 Implementation

- The first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
- During the program-understanding phase, you have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program.



SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Software Maintenance

Week 10: Software Testing and Evolution

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

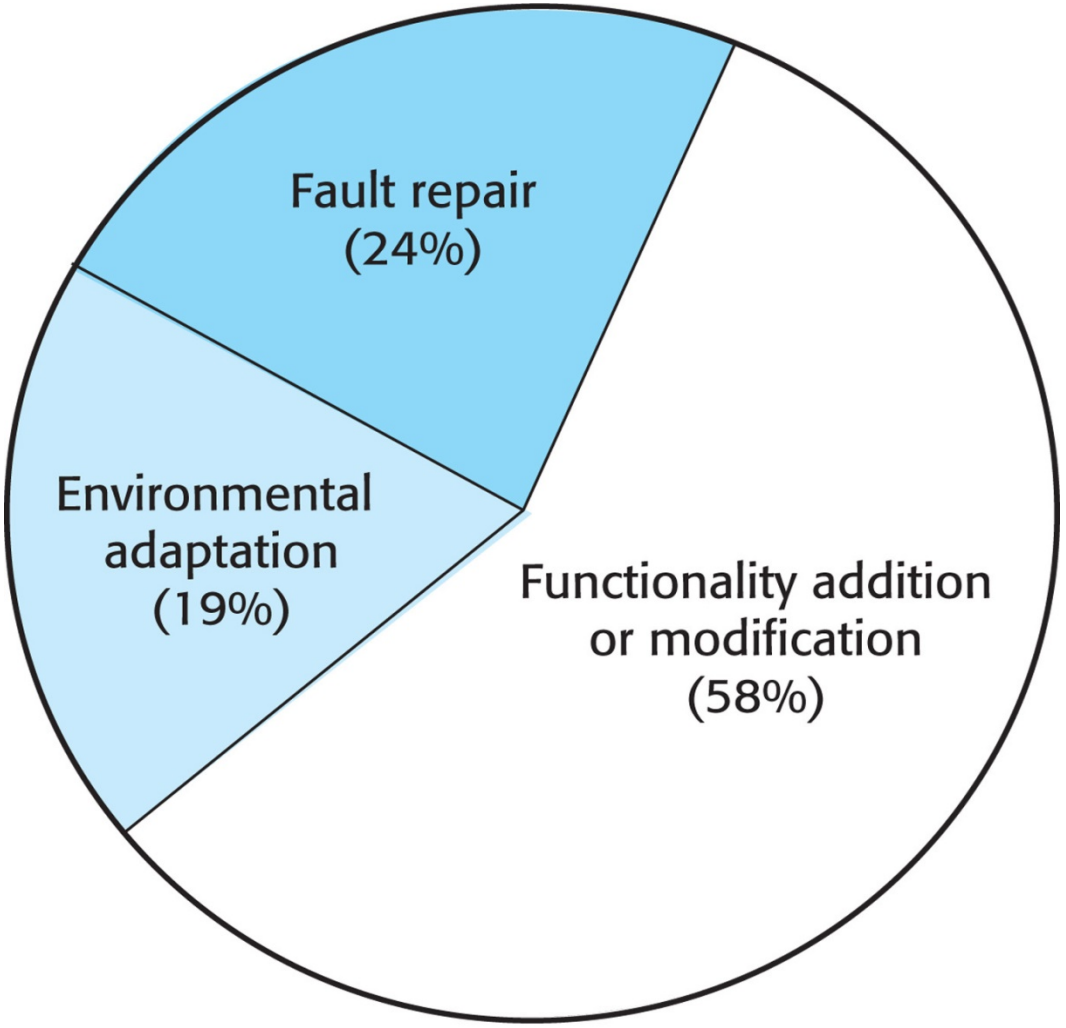
Software Maintenance

- Refers to 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.

Types of Maintenance

- Fault repairs
  - Changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements
- Environmental adaptation
  - Maintenance to adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation
- Functionality addition and modification
  - Modifying the system to satisfy new requirements

Maintenance Effort Distribution



Maintenance Costs

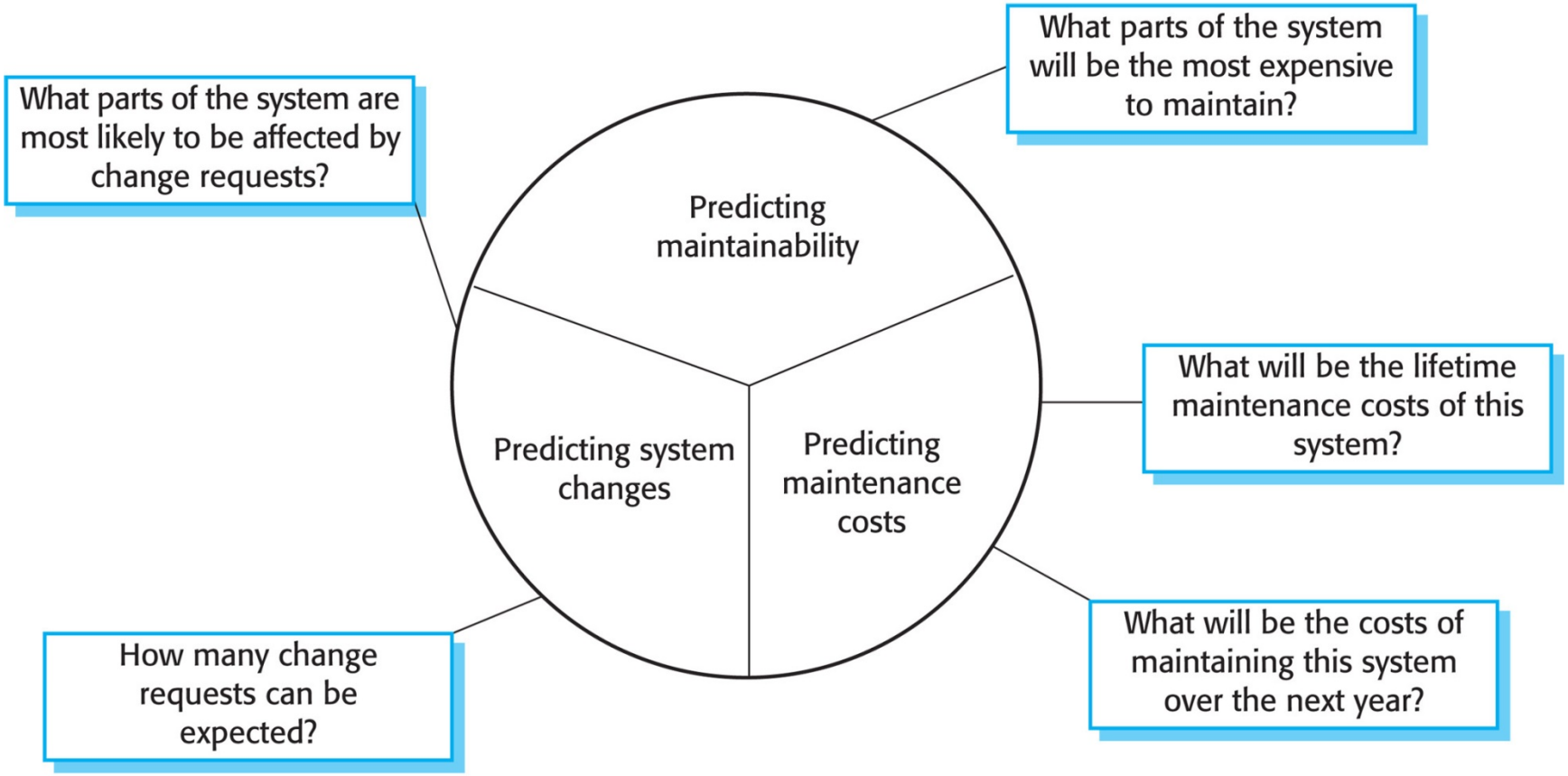
- Usually greater than development costs (2 times to 100 times, depending on the application).
- Affected by both technical and nontechnical factors.
- Increases as software is maintained. Maintenance corrupts the software structure, so makes further maintenance more difficult.
- Aging software can have high support costs (e.g., old languages, compilers etc.).

Maintenance Costs

- It is 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 and have limited domain knowledge.
  - As programs age, their structure degrades, and they become harder to change.

Maintenance Prediction

- Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs.



Change Prediction

- Predicting the number of changes requires an understanding of the relationships between a system and its environment.
- Tightly coupled systems require changes whenever the environment is changed.
- Factors influencing this relationship are
  - Number and complexity of system interfaces
  - Number of inherently volatile system requirements
  - The business processes where the system is used

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

Refactoring

- Refactoring is the process of making improvements to a program to slow down degradation through change.
- You can think of refactoring as “preventive 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 should not add functionality but rather concentrate on program improvement.

“Bad Smells” in Program Code

- Duplicate code
  - The same or very similar code may be included at different places in a program.
  - Duplicated code almost always represents a failure to fully factor the design in the first place.
  - Duplicate code sets you up to make parallel modifications: whenever you make changes in one place, you have to make parallel changes in another place.
  - It also violates what Andrew Hunt and Dave Thomas refer to as the “**DRY principle**”: Don't Repeat Yourself.
  - This can be removed and implemented as a single method or function that is called as required.

“Bad Smells” in Program Code

- Long methods
  - In object-oriented programming, methods longer than a screen are rarely needed and usually represent the attempt to force-fit a structured programming foot into an object-oriented shoe.
  - One way to improve a system is to increase its **modularity**—increase the number of well-defined, well-named methods that do one thing. →the single-responsibility principle
    - If you find a class that takes ownership of unrelated responsibilities, that class should be broken up into multiple classes, each of which has responsibility for a cohesive set of responsibilities.

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Български
Deutsch
Español
Français
한국어
हिन्दी
Italiano

Contents [hide]

Overview [edit]

Initial	Stands for	Concept
S	SRP [4]	**Single responsibility principle** a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
O	OCP [5]	**Open/closed principle** "software entities ... should be open for extension, but closed for modification."
L	LSP [6]	**Liskov substitution principle** "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.
I	ISP [7]	**Interface segregation principle** "many client-specific interfaces are better than one general-purpose interface."[8]
D	DIP [9]	**Dependency inversion principle** one should "Depend upon Abstractions. Do not depend upon concretions."[8]

“Bad Smells” in Program Code

- A parameter list has too many parameters.
  - Well-factored programs tend to have many small, well-defined routines that don't need large parameter lists. → **the magic number 7**
  - A long parameter list is a warning that the abstraction of the routine interface has not been well thought out.

“Bad Smells” in Program Code

- A subclass uses only a small percentage of its parents' routines.
  - Typically this indicates that that subclass has been created because a parent class happened to contain the routines it needed, not because the subclass is logically a descendent of the superclass.
  - Consider achieving better encapsulation by switching the subclass's relationship to its superclass from an **is-a** relationship to a **has-a** relationship.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE