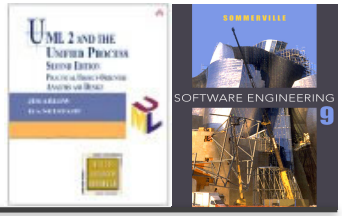


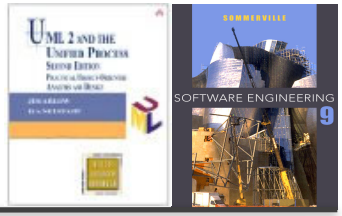
Lecture 9

TESTING, VERIFICATION AND VALIDATION

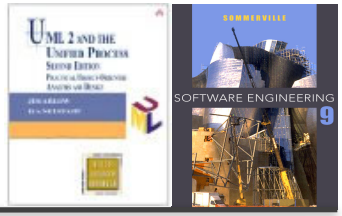
PB007 Software Engineering I
Faculty of Informatics, Masaryk University

Fall 2017

Outline



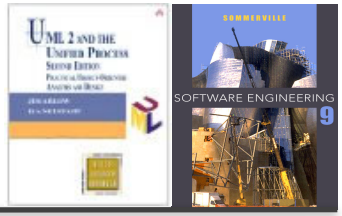
- ✧ Validation and verification
- ✧ Static analysis
- ✧ Testing and its stages
- ✧ Testing of non-functional properties



Validation and Verification

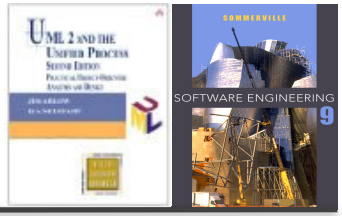
Lecture 9/Part 1

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 non-functional attributes.
- Can reveal the presence of errors **NOT their absence**.
- Testing is part of a more general verification and validation process, which also includes static V&V techniques.

Testing process goals



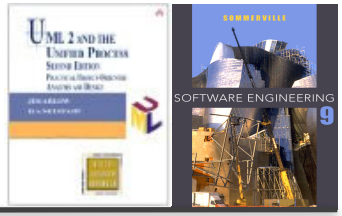
✧ Validation testing

- To demonstrate to the developer and the system customer that the software **meets its requirements**
- This means that there should be at least one test for every requirement or system feature.
- A **successful test** shows that the system operates as intended.

✧ Defect testing

- To **discover faults or defects** in the software where its behaviour is incorrect or not in conformance with its specification
- A **successful test** is a test that makes the system perform incorrectly and so exposes a defect in the system.

Validation vs. Verification



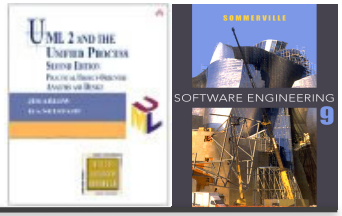
✧ **Validation:**
"Are we building the **right product**".

✧ The software should meet user expectations, conform to user requirements (high-level user specification).

✧ **Verification:**
"Are we building the **product right**".

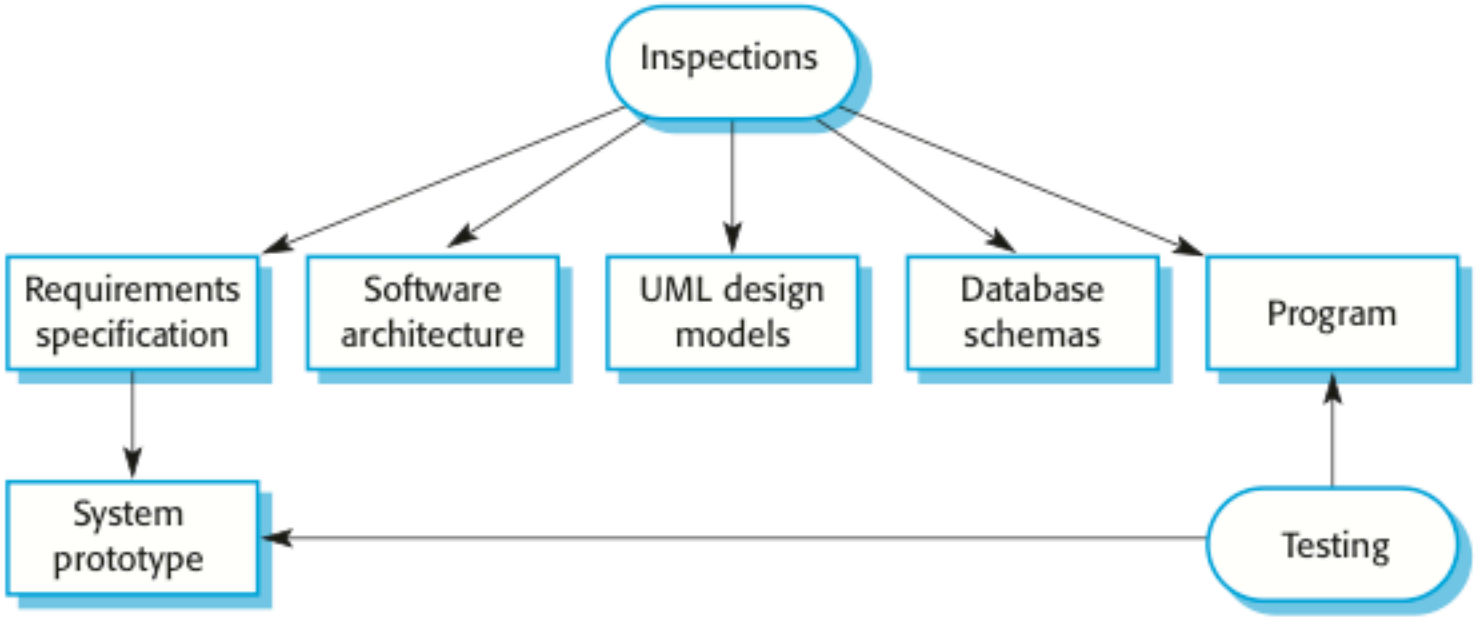
✧ The software should be defect-free, conform to low-level programmer specification (e.g. test cases).

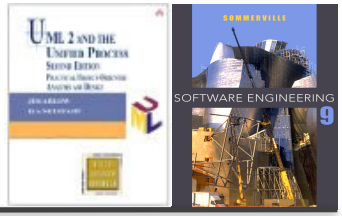
Static and dynamic verification



- **Static verification.** Concerned with analysis of the static system representation to discover problems – e.g. software inspections
  - May be supplement by tool-based document and code analysis.
- **Dynamic verification.** Concerned with exercising and observing product behaviour – e.g. software testing
  - The system is executed with test data and its operational behaviour is observed.

Inspections and testing

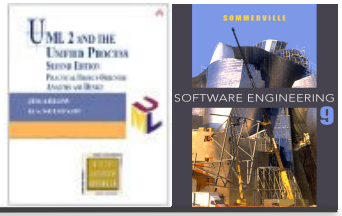




Static Analysis

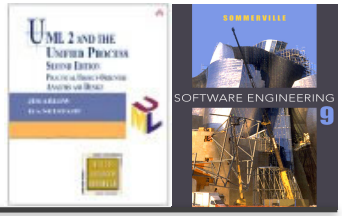
Lecture 9/Part 2

Static analysis



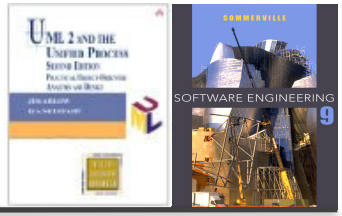
- Static analysis techniques are system verification techniques that **don't involve executing a program**.
- Static analysis includes techniques such as
  - Inspections and reviews
  - Automated program analysis
  - Formal verification (model checking)
- Static analysis has its value whenever it is **cheaper to find and remove faults than to pay for system failure** – in critical systems namely.

Software inspections



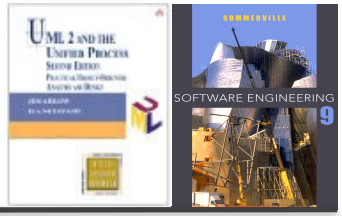
- These involve **people examining the source representation** with the aim of discovering anomalies and defects.
- Inspections **do not require execution** of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc).
- They have been shown to be an effective technique for discovering program errors.

Advantages of inspections



- 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, 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, such as **compliance with standards, portability and maintainability**.

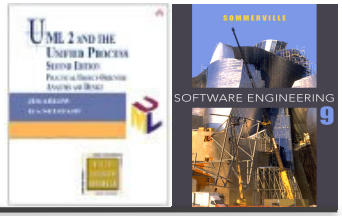
Automated static analysis



- 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.

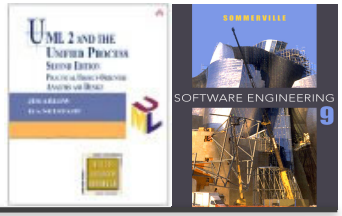
Do you use automated static analysis yourself?

Automated static analysis checks



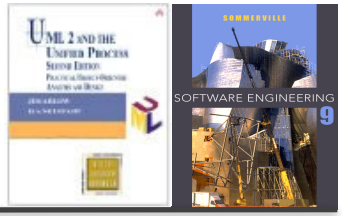
Fault class	Static analysis check
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output 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 automated static analysis



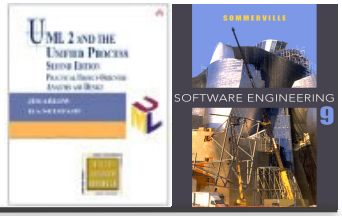
- ✧ Characteristic error checking
  - The static analyzer can check for **patterns in the code that are characteristic of errors** made by programmers using a particular language.
- ✧ User-defined error checking
  - **Users** of a programming language **define error patterns**, thus extending the types 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 automated static analysis



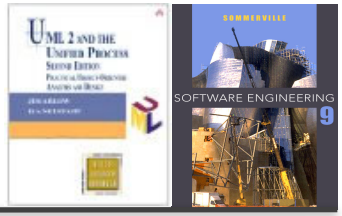
- Particularly valuable when a language such as C is used which has **weak typing** and hence many errors are undetected by the compiler.
- Particularly valuable for **security checking** – the static analyzer can discover areas of vulnerability such as buffer overflows or unchecked inputs.
- Static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

Verification and formal methods



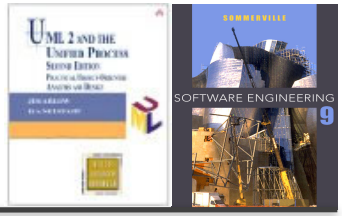
- Formal methods can be used when a **mathematical specification** of the system is produced.
- They are the ultimate static verification technique that may be used at different stages in the development process:
  - A formal specification may be developed and mathematically analyzed for consistency. This helps discover specification errors and omissions.
  - Formal arguments that a program conforms to its mathematical specification may be developed. This is effective in discovering programming and design errors.

Arguments for formal methods



- Producing a mathematical specification requires a **detailed analysis of the requirements** and this is likely to uncover errors.
- Concurrent systems can be analysed to discover **race conditions that might lead to deadlock**. Testing for such problems is very difficult.
- They can detect implementation errors before testing when the program is analyzed alongside the specification.

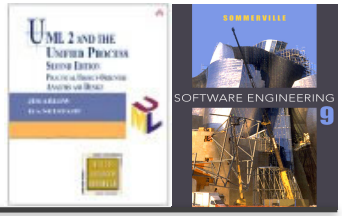
Arguments against formal methods



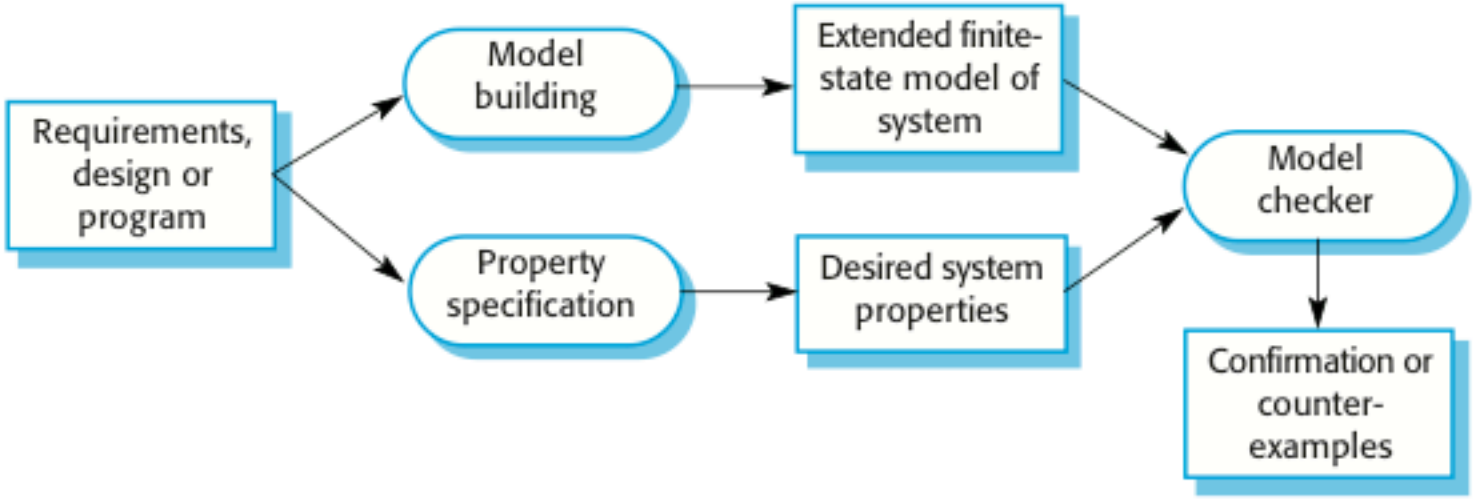
- Require specialised notations that are hard to understand for domain experts.
- **Very expensive to develop a specification** and to show that a program meets that specification.
- Proofs may contain errors.
- It may be possible to reach the same level of confidence in a program **more cheaply** using other V & V techniques.

What do you think, is formal verification used in practice?

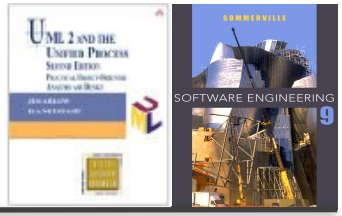
Model checking



- Involves creating an extended finite state model of a system and, using a specialized system (a model checker), checking that model for errors.
- Checks formal model against formal specification.

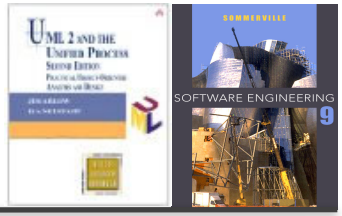


Model checking

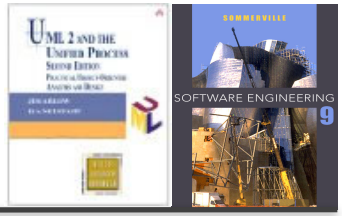


- The model checker explores all possible paths through the model and checks that a user-specified property is valid for each path.
- Model checking is particularly valuable for verifying concurrent systems, which are hard to test.
- Although model checking is computationally very expensive, it is now practical to use it in the verification of small to medium sized critical systems.

Key points



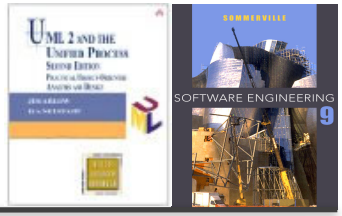
- **Inspections and reviews** involve people examining the source representation with the aim of discovering anomalies and defects.
- **Static analysis** is an approach to V&V that examines the source code (or other representation) of a system, looking for errors and anomalies.
- **Model checking** is a formal approach to static analysis that exhaustively checks all states in a system for potential errors.



Testing and its Stages

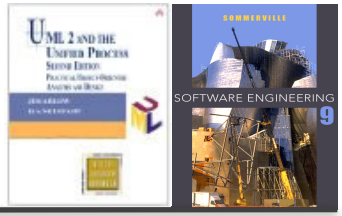
Lecture 9/Part 3

Topics covered



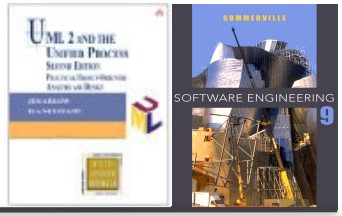
- ✧ Development testing
  - Unit testing
  - Component testing
  - System testing
- ✧ Release testing
- ✧ User testing
  - Alpha testing
  - Beta testing
  - Acceptance testing

Stages of testing



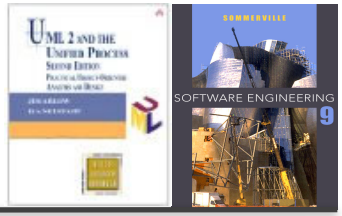
- **Development testing**, where the system is tested during development to discover bugs and defects.
- **Release testing**, where a separate testing team test a complete version of the system before it is released to users.
- **User testing**, where users or potential users of a system test the system in their own environment.

Development testing



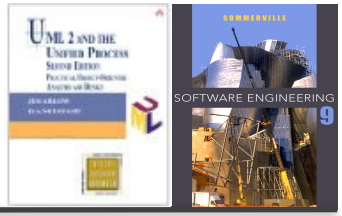
- Development testing includes all testing activities that are carried out by the team developing the system.
  - **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  - **System testing**, where 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



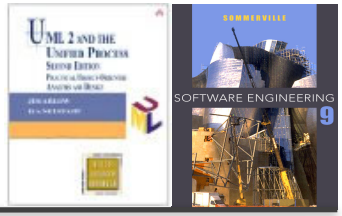
- Unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Units may be:
  - **Individual functions** or methods within an object
  - **Object classes** with several attributes and methods
  - **Composite components** with defined interfaces used to access their functionality.

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 localised.

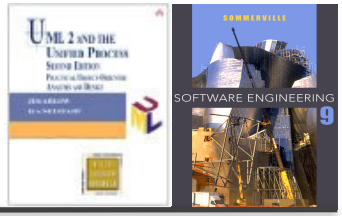
Example: Weather station testing



- Define test cases for each method and attribute usage **in isolation** (e.g. reportWeather()) or interaction if needed (e.g. shutdown() and restart()).
- 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
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

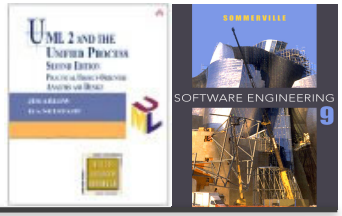
WeatherStation
identifier
reportWeather ( ) reportStatus ( ) powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

Automated testing



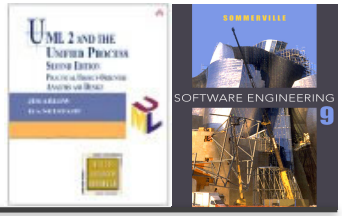
- Whenever possible, unit testing **should be automated** so that tests are run and checked without manual effort.
- Test automation frameworks (such as JUnit) can be used to write and run program tests.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests and report, often through some GUI, on the success of the tests.
- **Test structure:** setup – call – assertion evaluation, which compares the result of the call with the expected result

Testing strategies

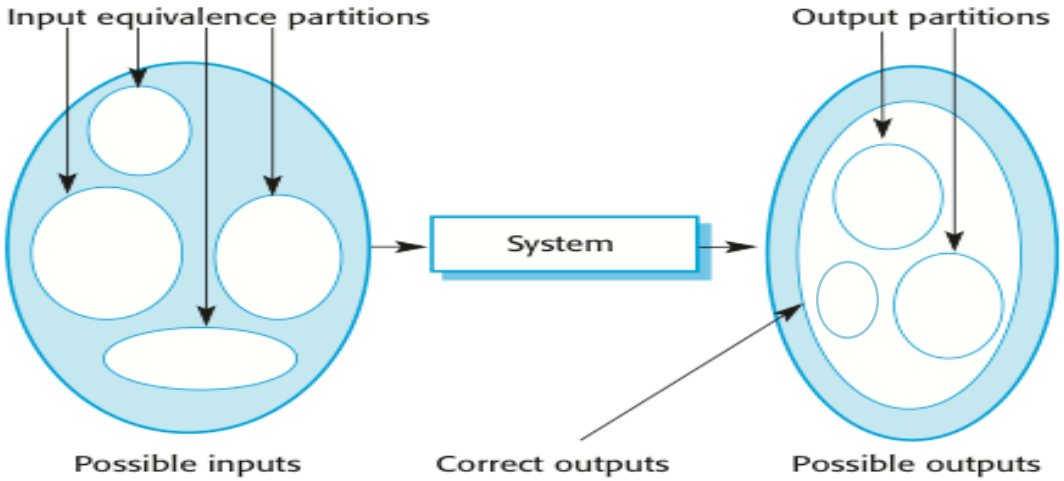


- **Partition testing**, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - You should choose tests from within each of these groups.
  - Focus on both the expected inputs (verifying normal execution) and abnormal inputs (localizing defects).
- **Guideline-based testing**, where 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.

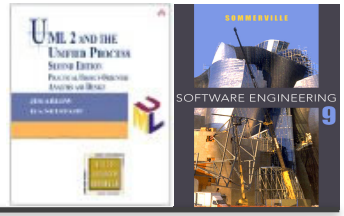
Partition testing



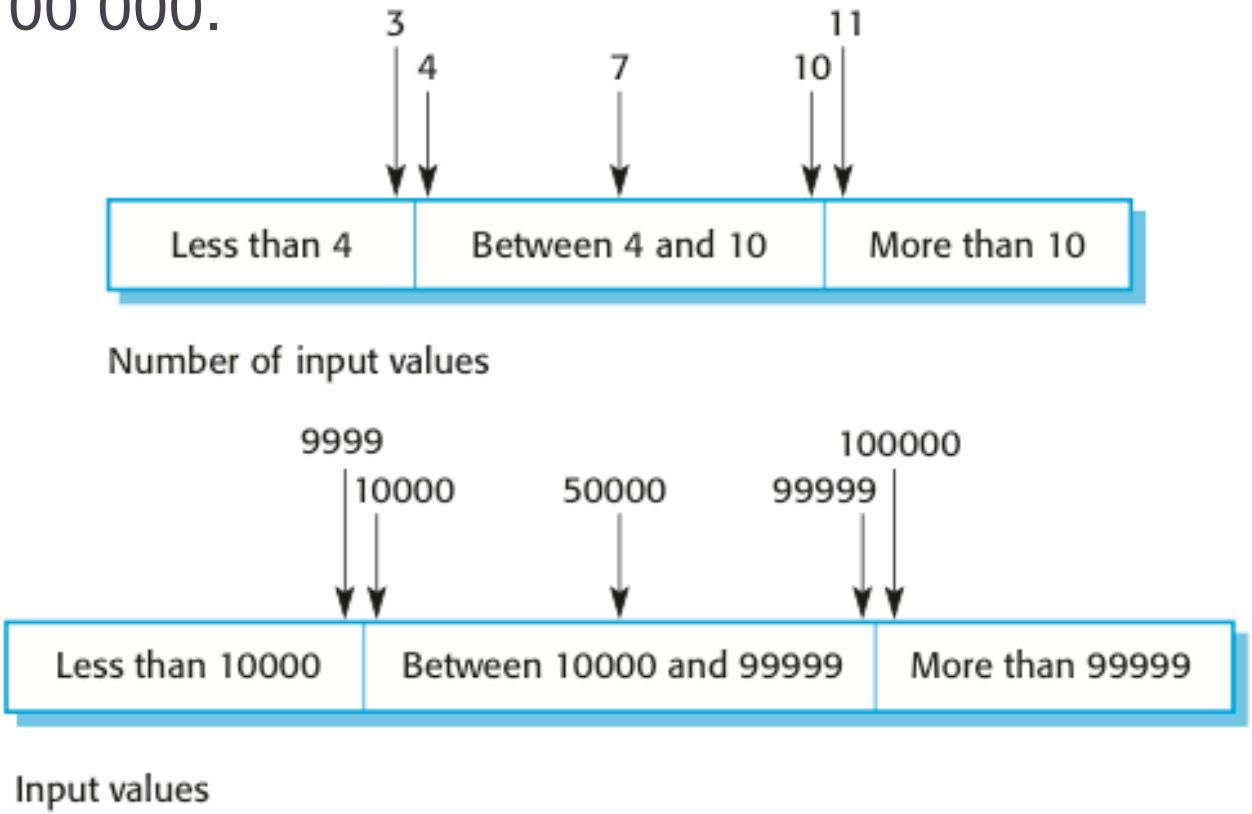
- Input data and output results often fall into different classes where all members of a class are related.
- The classes form **equivalence partitions** where the program behaves in an equivalent way for each member.
- Test cases should be chosen from each partition.



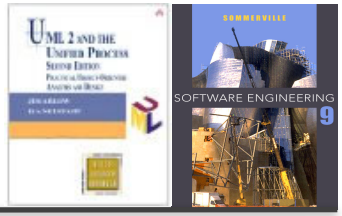
Equivalence partitions



- Consider a program that accepts 4 to 10 inputs which are five-digit integers greater than 10 000, but smaller than 100 000.

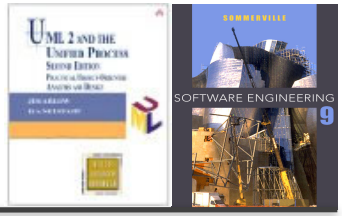


Testing guidelines



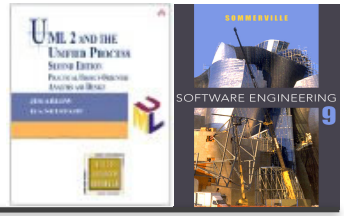
- Choose inputs that force the system to **generate all error messages**.
- Design inputs that cause **input buffers to overflow**.
- Repeat the same input or **series of inputs numerous times**.
- Force **invalid outputs** to be generated
- Force computation results to be **too large or too small**.

Component testing

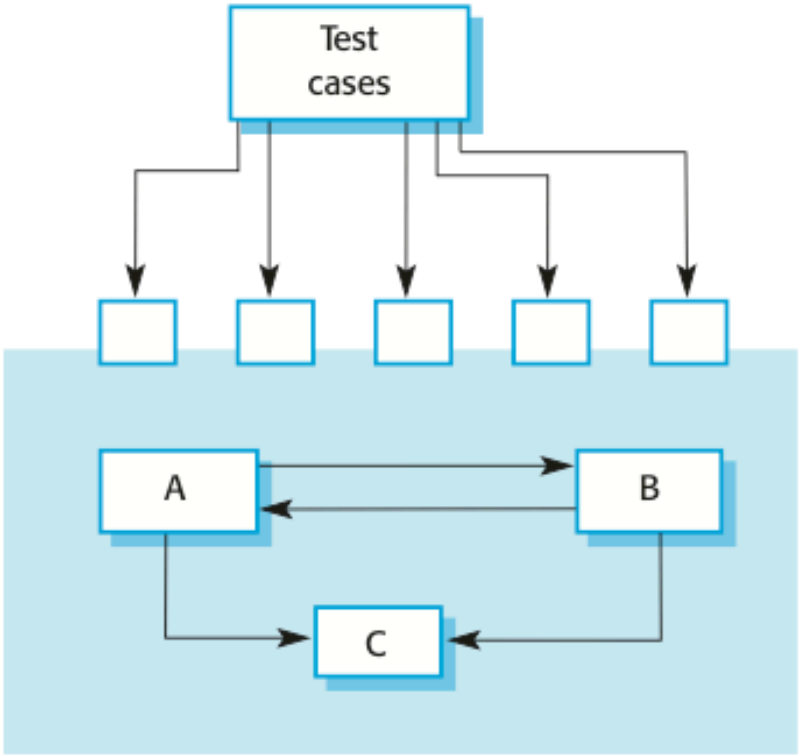


- Software components are often composite components that are made up of several interacting objects.
  - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects **through 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.

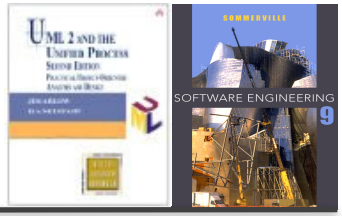
Interface testing



- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.

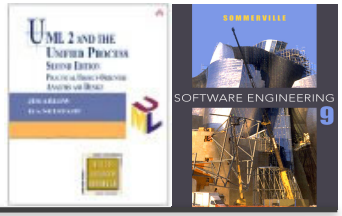


Interface errors



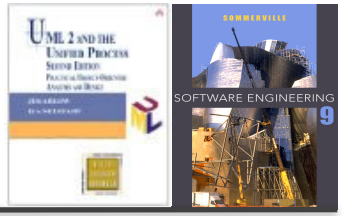
- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface e.g. **parameters in the wrong order**.
- Interface misunderstanding
  - A calling component embeds **assumptions about the behaviour** of the called component which are **incorrect**.
- Timing errors
  - The called and the calling component operate at different speeds and **out-of-date information is accessed**.
  - The caller is **violating the protocol** (assumed ordering of service calls) of the calling component.

Interface testing guidelines



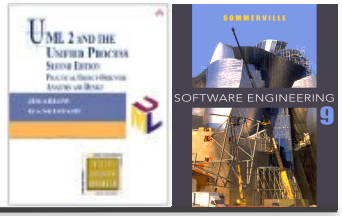
- Design tests so that parameters to a called procedure are at the **extreme ends of their ranges**.
- Always test pointer parameters with **null pointers**.
- Design tests which cause the component to fail.
- Use **stress testing** in message passing systems.
- In **shared memory systems**, vary the order in which components are activated.

System testing



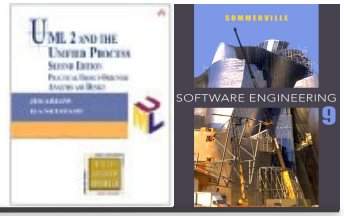
- **System testing** checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- The focus in system testing is testing the **interactions between components**.
- System testing during development involves **integrating components** to create a version of the system and then testing the integrated system.
  - Components developed by **different teams** and **third-party components** are integrated at this stage. System testing is a collective rather than an individual process.

Testing policies

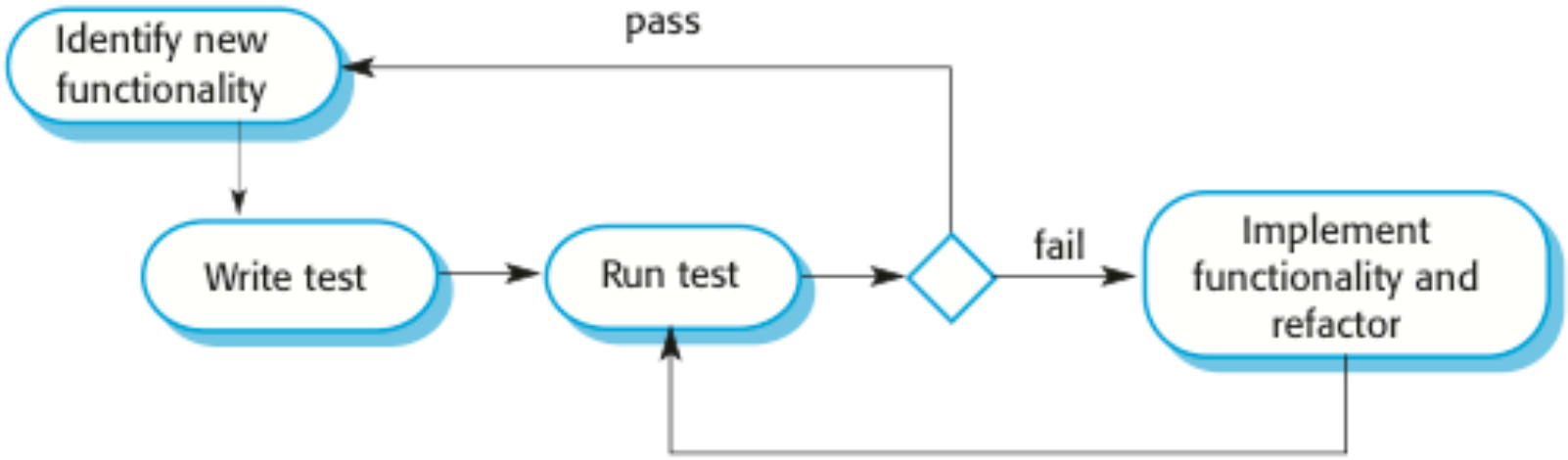


- **Exhaustive system testing is impossible** so testing policies which define the required system **test coverage** may be developed.
- Examples of testing policies:
  - All system functions that are **accessed through menus** should be tested.
  - Combinations of functions (e.g. text formatting) that 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.
  - All **use cases** must be executed during testing.

Test-driven development

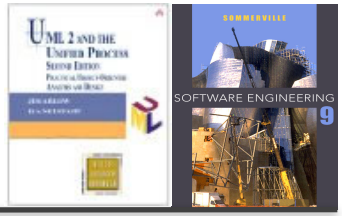


- Test-driven development (TDD) is an approach to program development in which tests are written before code and ‘passing’ the tests is the driver of development.



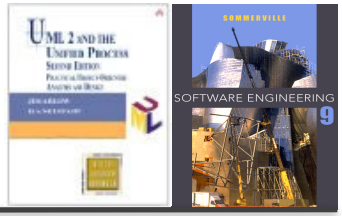
Is TDD connected with a specific process method?

Benefits of test-driven development



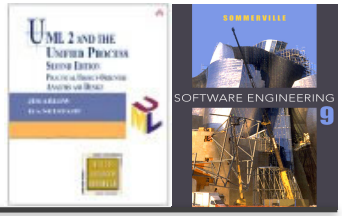
- Code coverage
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- Regression testing
  - A regression test suite is developed incrementally as a program is developed.
- Simplified debugging
  - When a test fails, it should be obvious where 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.

Regression testing



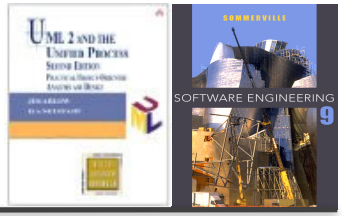
- Regression testing is testing the system to check that **changes have not 'broken' previously working code**.
- In a **manual testing** process, regression testing is expensive but, with **automated testing**, it is simple and straightforward. All tests are rerun every time a change is made to the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

Release testing



- Release testing is the process of testing a particular release of a system that is intended for use.
- 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.**
  - The use-cases developed to identify system requirements can be used as a basis for release testing.
- Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release testing and system testing

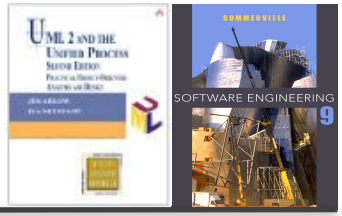


✧ Release testing is a form of system testing.

✧ Important differences:

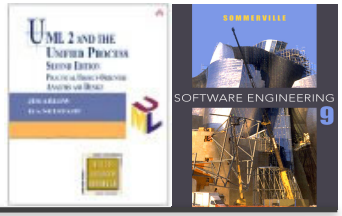
- **A separate team** that has not been involved in the system development, should be responsible for release testing.
- System testing by the development team should focus on **discovering bugs** in the system (defect testing).
  The objective of release testing is to check that the system **meets its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

Requirements based testing



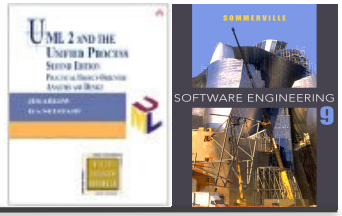
- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- MHC-PMS requirement:
  - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
  - Tests:
    - Patient record with no allergies. Prescribe medication. Check that a warning message is not issued by the system.
    - Patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued.
    - Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.

User testing



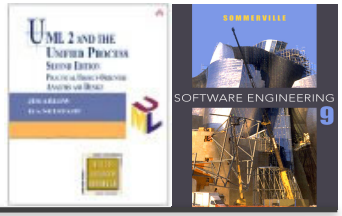
- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system.

Types of user testing

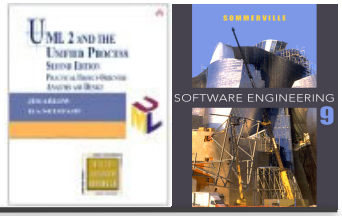


- ✧ Alpha testing
  - Users of the software work with the development team to test the software **at the developer's site**.
- ✧ Beta testing
  - A release of the software is **made available to users** to allow them to experiment and to raise problems that they discover with the system developers.
- ✧ Acceptance testing
  - Customers test a system to decide whether it is ready to be accepted from the developer and deployed in their environment.
  - Systematic activity with defined test criteria, test plan and result reports.

Key points



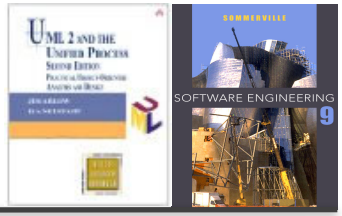
- Development testing
  - **Unit testing** – methods, classes, components
  - **Component testing** – interface testing
  - **System testing** – integration testing
  - Test Driven Development
  - Regression testing
- Release testing
- User testing
  - Alpha testing
  - Beta testing
  - Acceptance testing



Testing of Non-Functional Properties

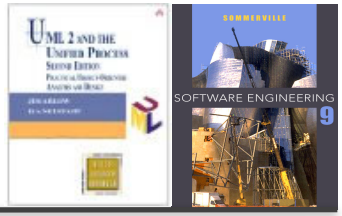
Lecture 9/Part 4

Topics covered



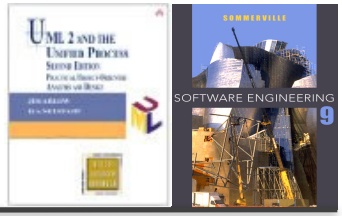
- ✧ Performance testing
- ✧ Reliability testing
- ✧ Security testing

Performance testing

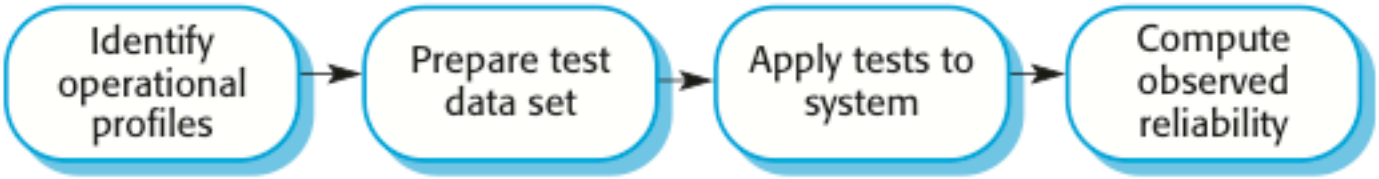


- **Release testing** may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the **profile of use** of the system.
- Performance tests usually involve planning a series of tests where the **load is steadily increased** until the system performance becomes unacceptable.
- **Stress testing** is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.

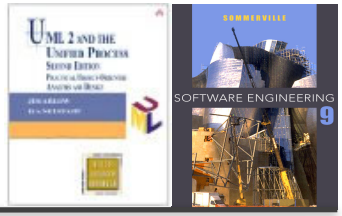
Reliability testing



- Reliability validation involves exercising the program to assess whether or not it has reached the required level of reliability.
- Reliability measurement requires a specially designed data set that reflects **expected system usage** (unlike in defect testing where atypical usage receives higher attention), i.e. **replicates expected input patterns**.

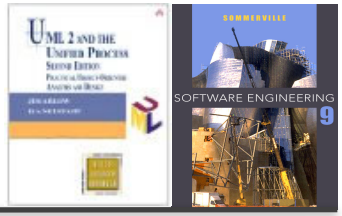


Reliability measurement problems



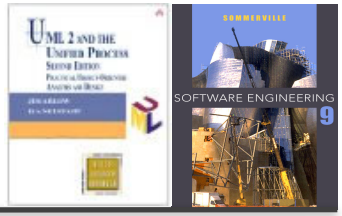
- Operational profile uncertainty
  - The operational profile may not be an accurate reflection of the real use of the system.
- Statistical uncertainty
  - You need a statistically significant number of failures to compute the reliability but highly reliable systems will rarely fail.
- Recognizing failure
  - It is not always obvious when a failure has occurred as there may be conflicting interpretations of a specification.
- High costs of test data generation
  - Costs can be very high if the test data for the system cannot be generated automatically.

Security testing



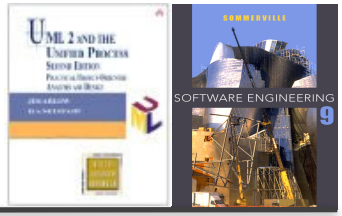
- Testing the extent to which the system can protect itself from external attacks.
- Problems with security testing
  - Security requirements are **'shall not' requirements** i.e. they specify what should not happen. It is not usually possible to define security requirements as simple constraints that can be checked by the system.
  - The **people attacking a system are intelligent** and look for vulnerabilities. They can experiment to discover weaknesses and loopholes in the system.
- Static analysis may be used to guide the testing team to areas of the program that may include errors and vulnerabilities.

Security validation



- ✧ Experience-based validation
  - The system is reviewed and analysed against the types of attack that are known to the validation team.
- ✧ Tiger teams
  - A team is established whose goal is to breach the security of the system by simulating attacks on the system.
- ✧ Tool-based validation
  - Various security tools such as password checkers are used to analyse the system in operation.
- ✧ Formal verification
  - The system is verified against a formal security specification.

Key points



- **Performance testing** tests system performance properties.
- **Reliability testing** relies on testing the system with a data set that reflects the operational profile of the software.
- **Security validation** may be carried out using experience-based analysis, tool-based analysis or ‘tiger teams’ that simulate attacks on a system.