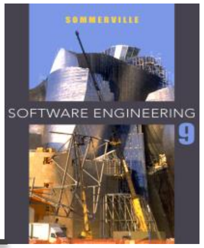
Forelesning 18 - IMT2243

Tema : Testing , Idriftssetting og Vedlikehold

- Hvorfor tester vi i programvareutviklingsprosjekter ?
- Hvordan skal vi legge opp testingen ?
  - V-modellen, strategier (TD, BU), metoder (BB,WB) og teknikker (BPT, EP) for testing
- Testaktiviteten i ulike omgivelser (TDD, OSSD)
- Hva er inspeksjoner ? Sammenheng med testing ?
- Idriftsetting og Vedlikehold

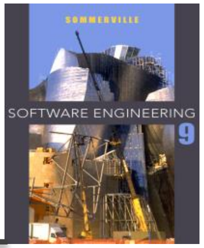
Pensum : Sommerville kap 2.2.3, kap 8, kap 9.3 og kap 24.3

Program testing



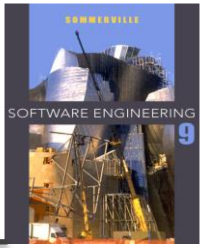
- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- When you test software, you execute a program using artificial data.
- You check the results of the test run for errors, anomalies or information about the program's 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 validation techniques.

Program testing goals



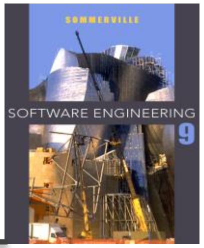
- ✧ To demonstrate to the developer and the customer that the software meets its requirements.
  - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- ✧ To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

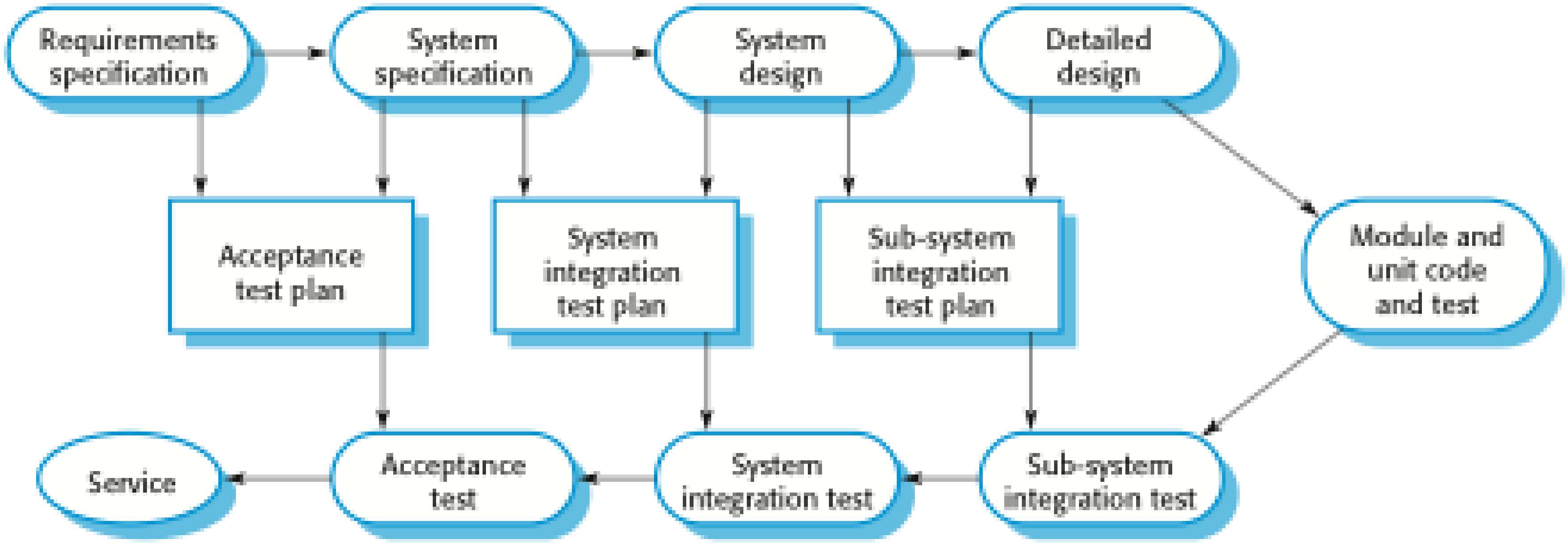
Verification vs validation



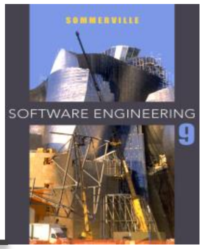
- ✧ Verification:
  "Are we building the product right”.
- ✧ The software should conform to its specification.
- ✧ Validation:
  "Are we building the right product”.
- ✧ The software should do what the user really requires.

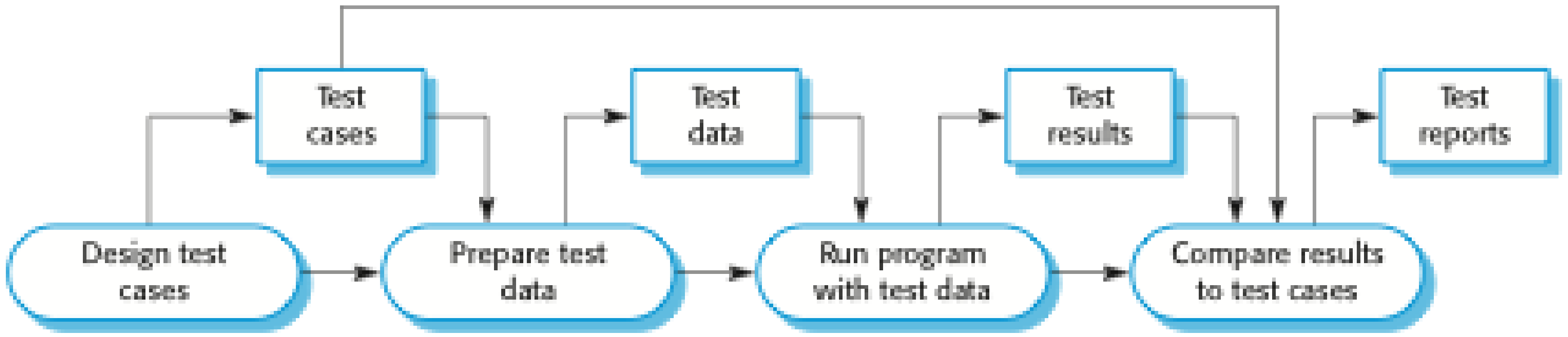
Testing phases in a plan-driven software process



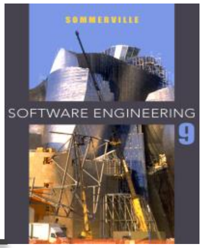

Requirements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and test
Service
Acceptance test
System integration test
Sub-system integration test

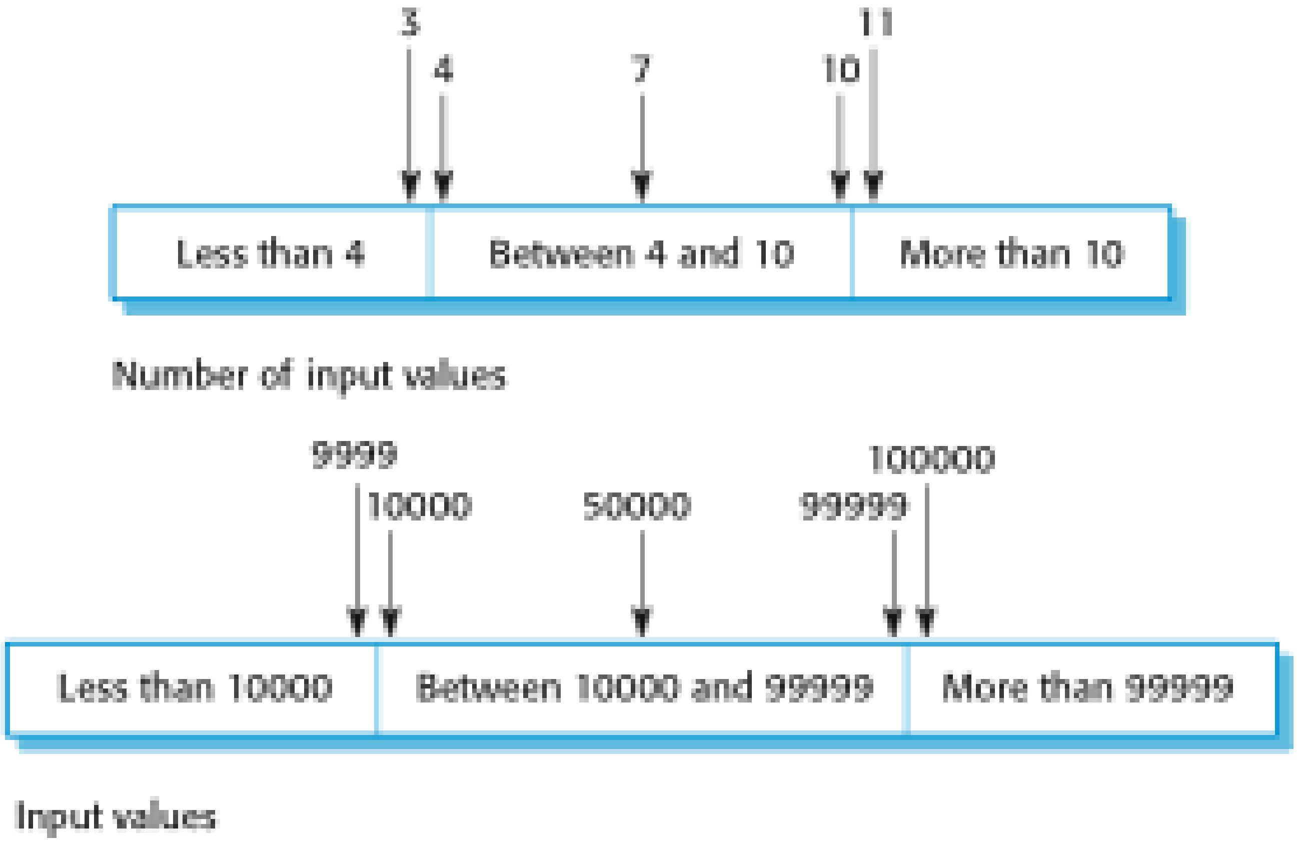
A model of the software testing process



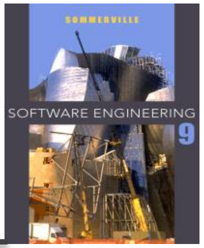


Equivalence partitions





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.

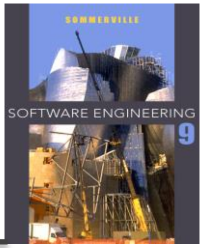
✧ Interface misunderstanding

- A calling component embeds assumptions about the behaviour of the called component which are incorrect.

✧ Timing errors

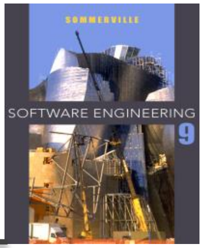
- The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines



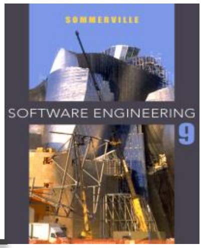
- ✧ Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- ✧ Always test pointer parameters with null pointers.
- ✧ Design tests which cause the component to fail.
- ✧ Use stress testing in message passing systems.
- ✧ In shared memory systems, vary the order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

System testing



- ✧ System testing 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 behaviour of a system.

Performance testing



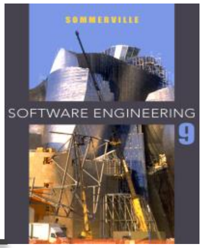
✧ Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.

✧ 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.

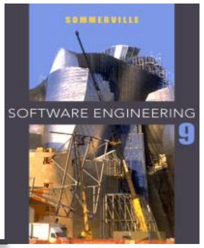
✧ Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.

Test-driven development



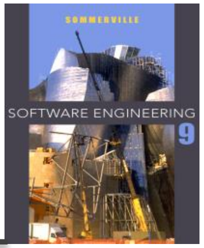
- ✧ Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- ✧ Tests are written before code and ‘passing’ the tests is the critical driver of development.
- ✧ You develop code incrementally, along with a test for that increment. You 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. However, it can also be used in plan-driven development processes.

TDD process activities



- ✧ Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- ✧ Write a test for this functionality and implement this as an automated test.
- ✧ Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- ✧ Implement the functionality and re-run the test.
- ✧ Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of test-driven development



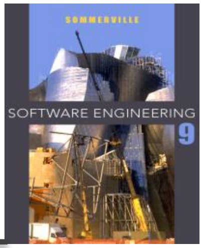
- ✧ Code coverage
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- ✧ Regression testing
  - A regression test suite is developed incrementally as a program is developed.
- ✧ Simplified debugging
  - When a test fails, it should be obvious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

Testfilosofi / tilnærming innen OpenSource Software Dev.

Linus's Law as described by Raymond is a claim about software development, named in honor of Linus Torvalds and formulated by Raymond in his essay and book "The Cathedral and the Bazaar" (1999).[1][2]

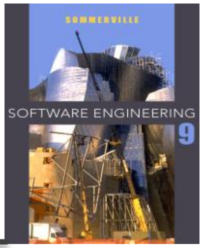
The law states that "**given enough eyeballs, all bugs are shallow**"; or more formally: "Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone."

Inspections and testing

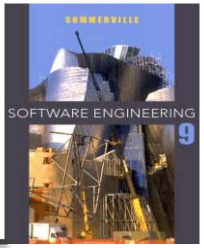


- ✧ Software inspections Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis.
- ✧ Software testing Concerned with exercising and observing product behaviour (dynamic verification)
  - The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

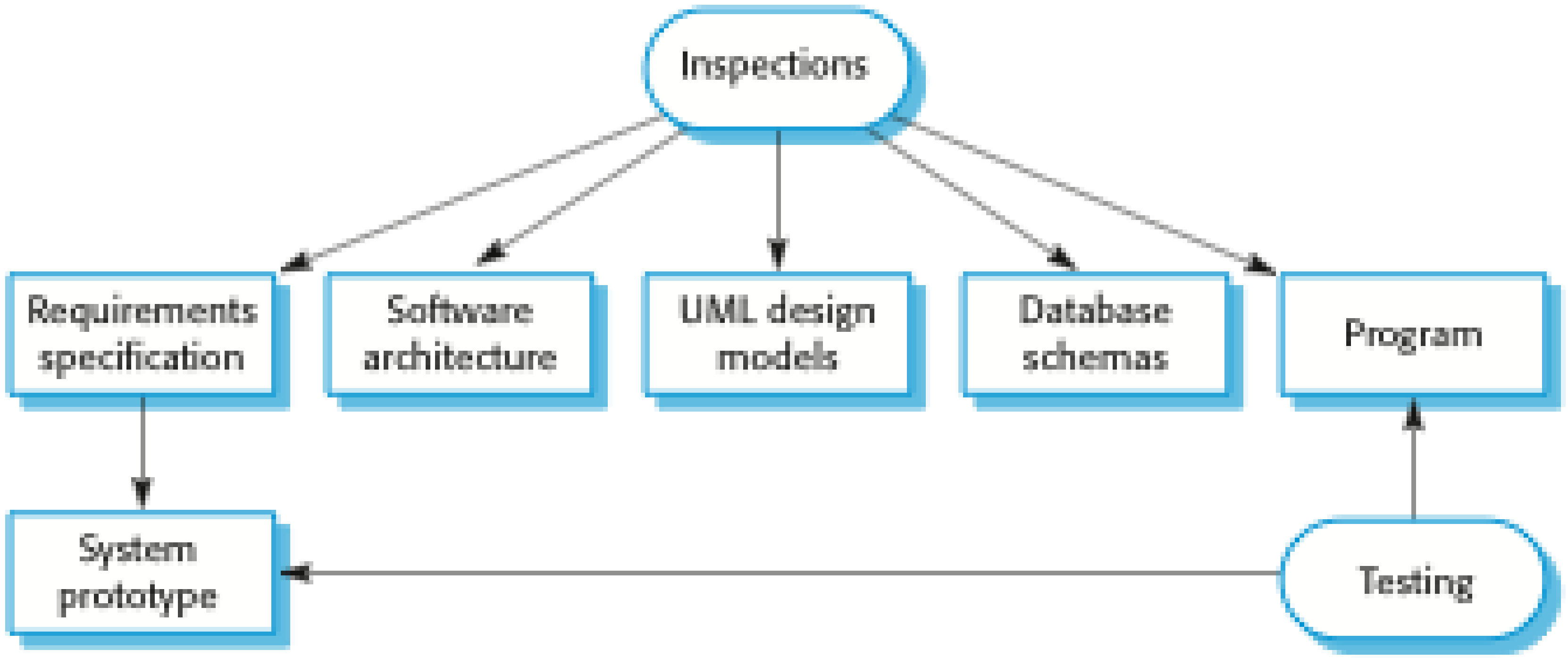
Reviews and inspections



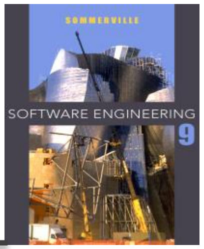
- ✧ A group examines part or all of a process or system and its documentation to find potential problems.
- ✧ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- ✧ There are different types of review with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

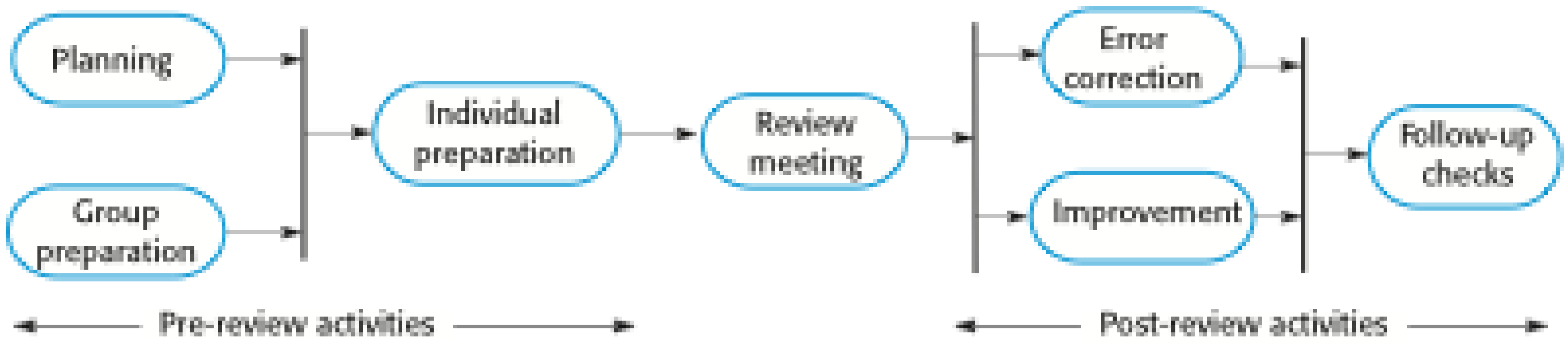


Inspections and testing

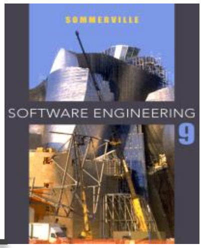


The software review process



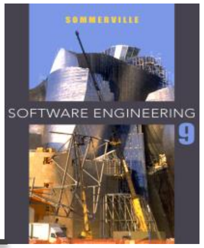


An inspection checklist (a)



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

An inspection checklist (b)



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

Idriftssetting – ulike strategier

Nyere trend :

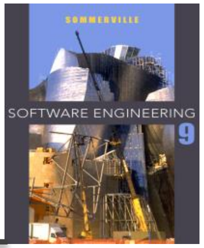
DevOps – se tidligere forelesning – Kyrre Begum
Cloud-teknologi nøkkelfaktor for å muliggjøre DevOps

4 ulike strategier ved mer tradisjonell idriftssetting :

Direkte idriftssetting av samlet løsning («Big Bang»)
Paralellkjøring med gammel løsning
Stegvis idriftssetting (ved Inkrementelt utviklet progvare)
Lokasjonsbasert idriftssetting

Viktige aspekter å ivareta rundt dennefasen :

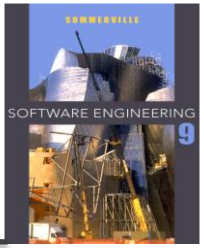
Opplæring og Dokumentasjon av prosedyrer
Risikoanalyse og reelle «retrettuligheter» som er testet
Migrering av historiske data
Timing (når er det hensiktsmessig for organisasjonen?)
Avklaring av Godkjenningsperiode



Chapter 9 – Software Evolution

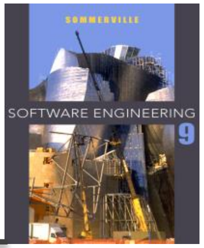
Lecture 2

Software maintenance



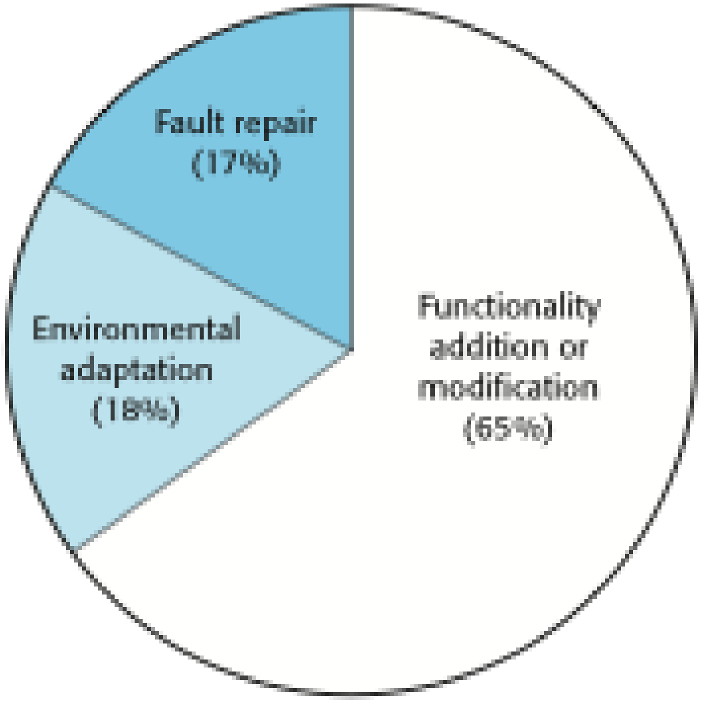
- ✧ Modifying a program after it has been put into use.
- ✧ The term is mostly used for changing custom software. Generic software products are said to evolve to create new versions.
- ✧ Maintenance does not normally involve major changes to the system’s architecture.
- ✧ Changes are implemented by modifying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

Types of maintenance



- ✧ Maintenance to repair software faults
  - ▪ Changing a system to correct deficiencies in the way meets its requirements.
- ✧ 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.
- ✧ Maintenance to add to or modify the system’s functionality
  - ▪ Modifying the system to satisfy new requirements.

Figure 9.8 Maintenance effort distribution


Fault repair (17%)
Environmental adaptation (18%)
Functionality addition or modification (65%)