Software Engineering – Lecture 8
Verification and Validation (V & V)

Adapted after ©**Ian Sommerville**
Software Engineering, 2010, chapter **8** ;
Engineering Software Products, 2019, chapter **9**

Topics covered

Verification and validation

Testing

Functional testing

Test-driven development

Quality testing

The V & V process

V&V process is a *whole life-cycle process* - V & V must be applied at each stage in the software process.

Objectives:

- The discovery of *defects* in a system (verification);

  "Are we building the product right ?".

- The assessment of whether or not the system is *useful* and *useable* in an operational situation (validation).

  "Are we building the right product ?".

- Static V&V – software inspections
- Dynamic V&V – testing

Software testing

Testing - part of V&V.

- Intended to show that a program does what it is intended to do (validation) and to discover program defects (verification) before it is put into use.
- Testing implies *executing* a program using artificial data.
- The results of the test run are checked for errors, anomalies or information about the program’s extra-functional attributes.
- Can reveal the presence of errors NOT their absence.

V&V testing

Validation testing

Objective:To demonstrate to the developer and the system customer that the software meets its requirements.

- The system is expected to perform correctly using a given set of test cases that reflect the system's expected use.
- A successful test shows that the system operates as intended.

Verification (defect) testing

Objective: To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification

- 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.

Software inspections

- Software inspections 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, code, test data, etc.).

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

Advantages of inspections

- *Many different defects* may be discovered in a single inspection. In testing, one defect may mask another, so several executions are required.
- Incomplete versions of a system can be inspected. If a program is incomplete - specialized test harnesses to test the parts that are available must be developed.
- Inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

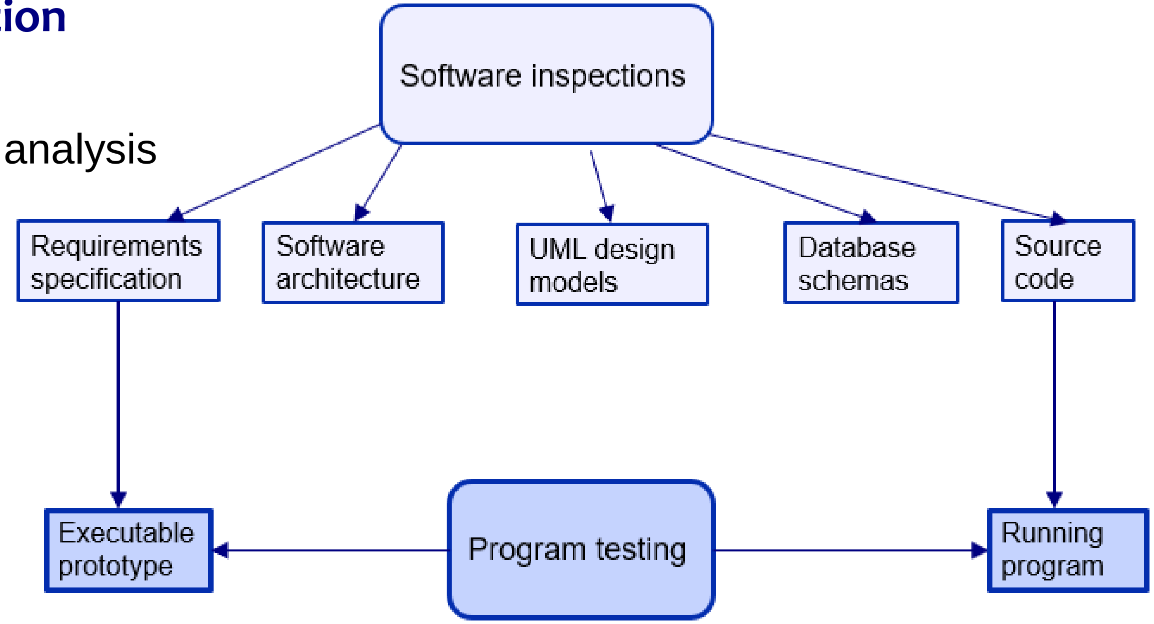
Inspections and testing

- ***Software inspections***. Concerned with analysis of the *static system representation* to discover problems

Static verification and validation

May be supplemented by

tool-based document and code analysis

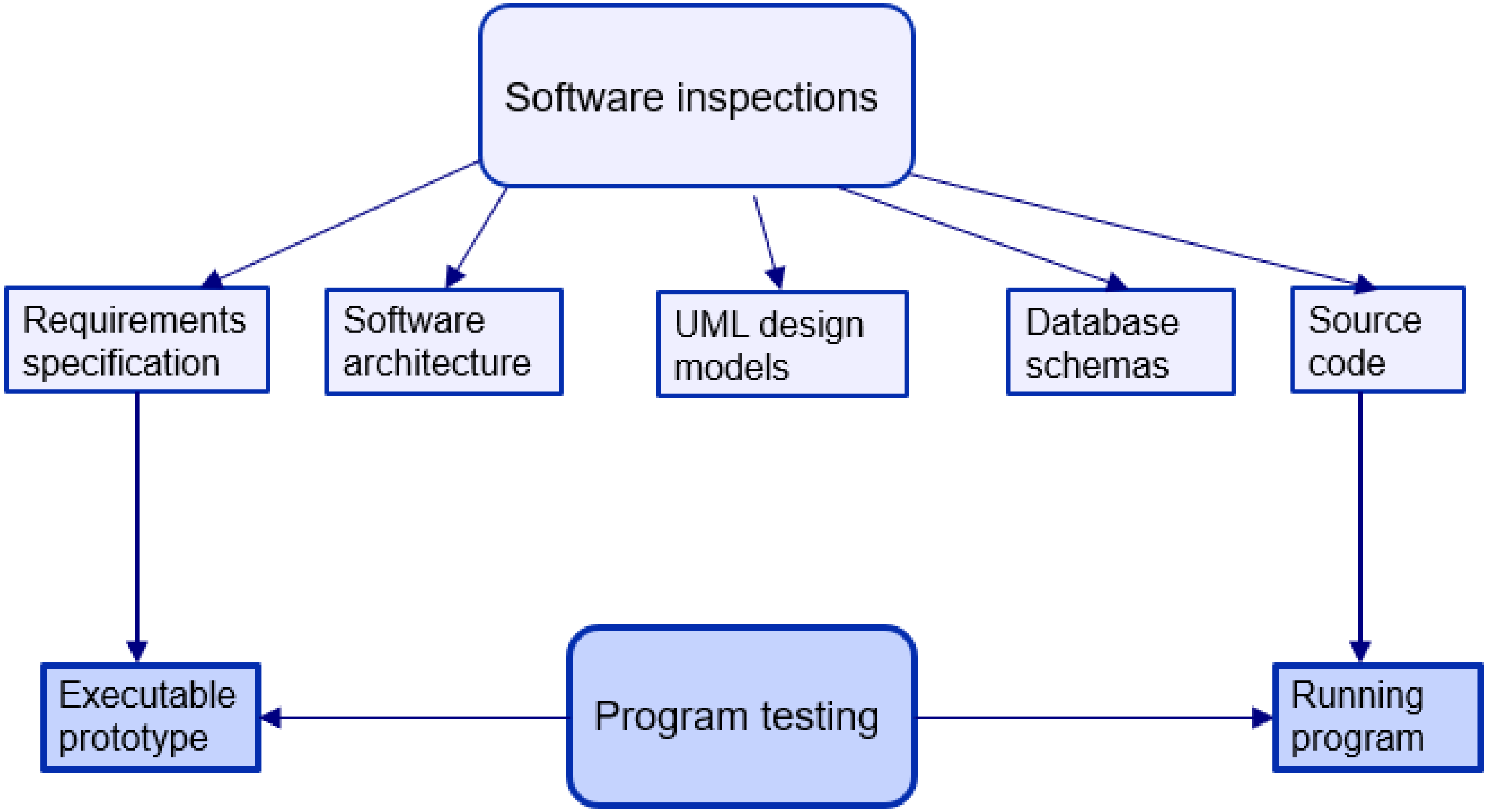


- ***Software testing***. Concerned with exercising and observing *product behaviour*

Dynamic verification and validation

The system is executed with test data and its operational behaviour is observed.

Inspections and testing



Inspections and testing

- Inspections and testing are complementary V&V techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer’s real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Code reviews (inspections of the code)

- Code reviews are effective in finding bugs that arise through *misunderstandings* and bugs that may only arise when *unusual sequences of code* are executed.
- Many software companies insist that all code has to go through a process of code review before it is integrated into the product codebase.

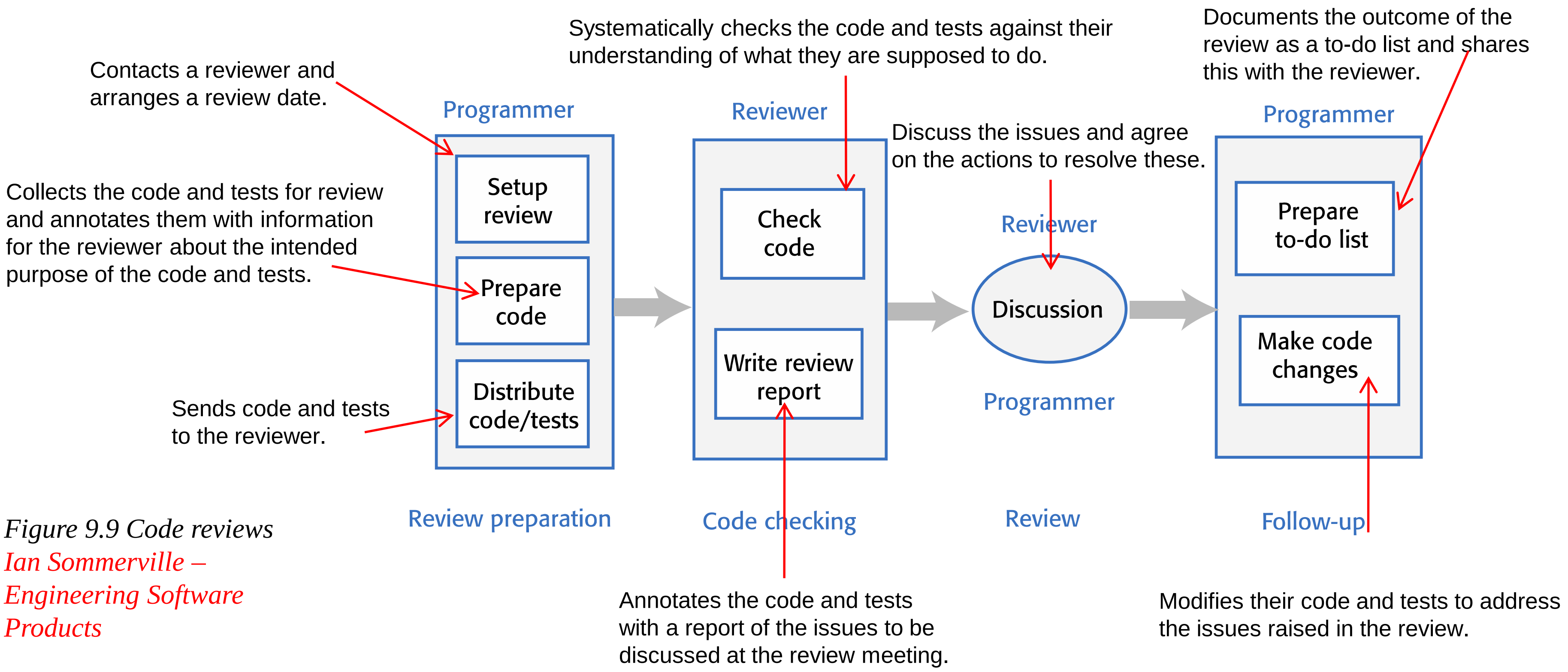

Contacts a reviewer and arranges a review date.
Collects the code and tests for review and annotates them with information for the reviewer about the intended purpose of the code and tests.
Sends code and tests to the reviewer.
Programmer
Setup review
Prepare code
Distribute code/tests
Review preparation
Systematically checks the code and tests against their understanding of what they are supposed to do.
Reviewer
Check code
Write review report
Code checking
Annotates the code and tests with a report of the issues to be discussed at the review meeting.
Discuss the issues and agree on the actions to resolve these.
Reviewer
Discussion
Programmer
Review
Documents the outcome of the review as a to-do list and shares this with the reviewer.
Programmer
Prepare to-do list
Make code changes
Follow-up
Modifies their code and tests to address the issues raised in the review.

Figure 9.9 Code reviews
Ian Sommerville – Engineering Software Products

Part of a check list for a Python code review

- ***Are meaningful variable and function names used? (General)***
  Meaningful names make a program easier to read and understand.
- ***Have all data errors been considered and tests written for them? (General***)
  It is easy to write tests for the most common cases but it is equally important to check that the program won’t fail when presented with incorrect data.
- ***Are all exceptions explicitly handled? (General)***
  Unhandled exceptions may cause a system to crash.
- ***Are default function parameters used? (Python)***
  Python allows default values to be set for function parameters when the function is defined. This often leads to errors when programmers forget about or misuse them.
- ***Are types used consistently? (Python)***
  Python does not have compile-time type checking so it it is possible to assign values of different types to the same variable. This is best avoided but, if used, it should be justified.
- ***Is the indentation level correct? (Python)***
  Python uses indentation rather than explicit brackets after conditional statements to indicate the code to be executed if the condition is true or false. If the code is not properly indented in nested conditionals this may mean that incorrect code is executed.

Formative evaluation

1. When is considered to be successful a validation test and when a verification test ?

2. What is the method to statically realize verifications and validations and what artifacts, obtained during software development process, are verified and validated in this way ?

3. Which are the advantages offered by source code inspections compared with code testing ?

https://forms.gle/Vbx7b7kCcZkutMWL9

Topics covered

Verification and validation

Testing

Functional testing

Test-driven development

Quality testing

Types of testing

Functional testing

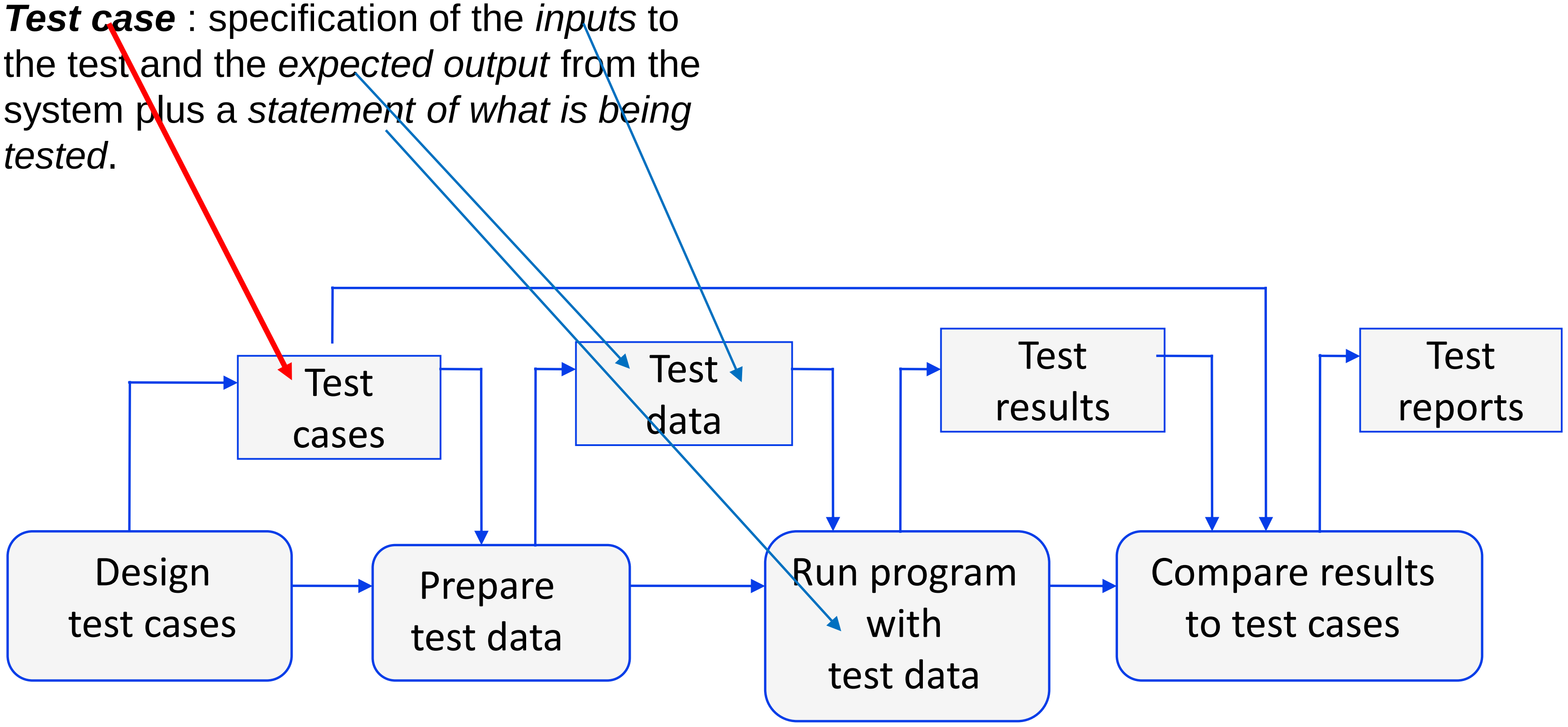
Test the functionality of the overall system. The goals of functional testing are to *discover* as many *bugs* as possible in the implementation of the system and to provide convincing evidence that the *system is fit for its intended purpose*.

Quality testing

- ***Usability testing***
  Test that the software product is *useful* to and *usable* by end-users. You need to show that the features of the system help users do what they want to do with the software. You should also show that users understand how to access the software's features and can use these features effectively.
- ***Performance and load testing***
  Test that the software works *quickly* and can handle the *expected load* placed on the system by its users. You need to show that the response and processing time of your system is acceptable to end-users. You also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases.
- ***Security testing***
  Test that the software maintains its *integrity* and can *protect user information* from theft and damage.

A model of the software testing process

Test case : specification of the *inputs* to the test and the *expected output* from the system plus a *statement of what is being tested*.



Stages of testing

Development testing - the system is tested during development to discover bugs and defects.

Levels of development testing :

- *Unit* testing : units of code are tested in isolation
- *Feature* testing : system features are tested
- *System* testing : system is tested as a whole

Release testing - a separate testing team tests a complete version of the system before it is released to users.

User testing - users of a system test the system in their own environment.

User testing

- User (customer) testing - stage in the testing process of *project-based software* 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.

Reason : 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.

Stages 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* (testing stage for custom systems) **-** *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.

Test automation

Test automation is based on the idea that *tests should be executable*.

The tests are *embedded in a program* that can be *run every time a change is made* to a system.

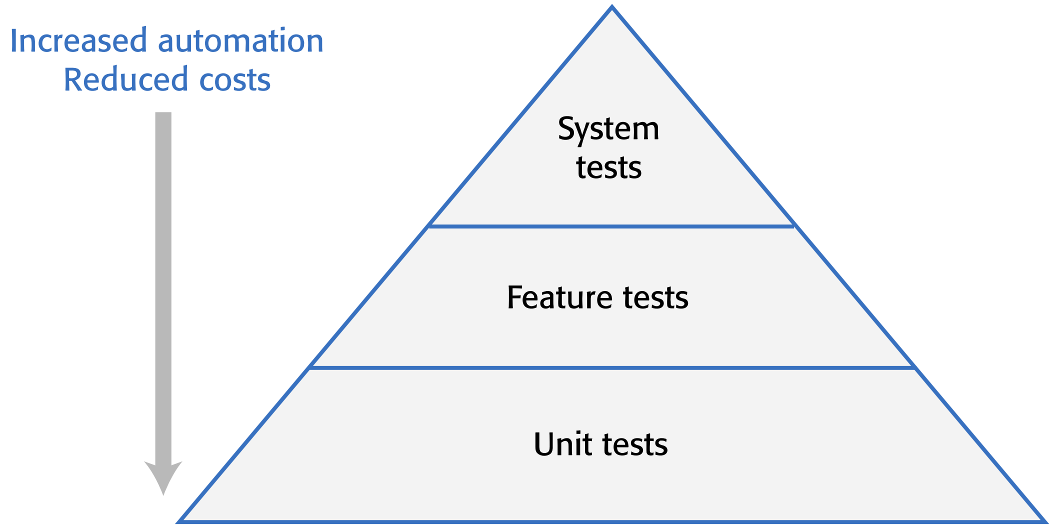


Figure 9.5 The testing pyramid
Ian Sommerville – Engineering Software Products

Topics covered

Verification and validation

Testing

Functional testing

Test-driven development

Quality testing

Key points

V&V must be applied at each stage in the software process. Verification is concerned with the discovery of *defects* in a system and validation is concerned with the assessment of *usefulness* and *usability* of the system in an operational situation.

Software inspections (static V&V) implies analysis of the *static system representation* to discover problems. Software testing (dynamic V&V) implies *executing a program* using artificial data and observing its behaviour.

Testing *stages* are : development testing (unit testing, feature testing, system testing), release testing and user testing. Both software *functionality* and software *qualities* must be tested.

Test automation is based on the idea that tests should be executable. The tests are embedded in a program that can be run every time a change is made to a system.

Functional testing

Functional testing involves developing a large set of program tests so that, ideally, all of a program's code is executed at least once.

The number of tests needed depends on the size and the functionality of the application.

Functional testing is a staged activity :

- individual units of code testing,
- integrated code units testing,
- final system testing.

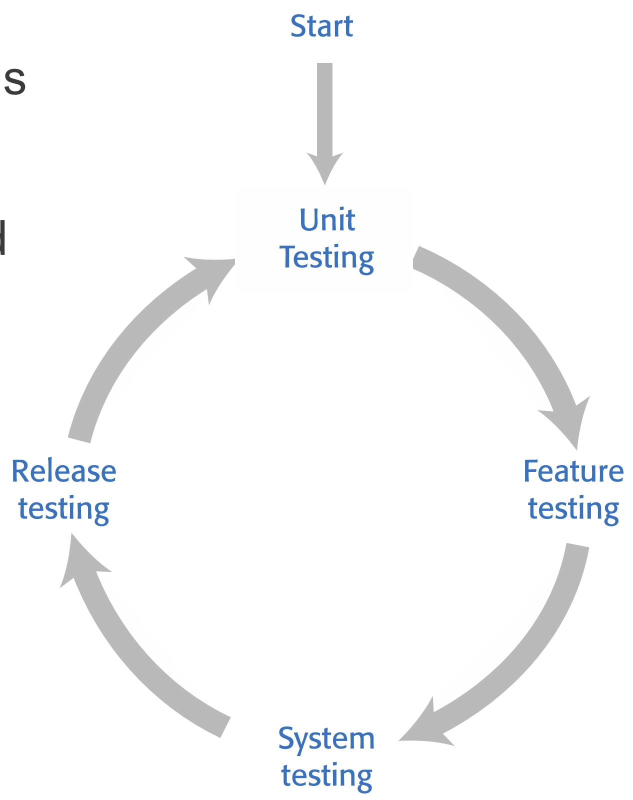


Figure 9.2 Functional testing
Ian Sommerville – Engineering Software Products

Functional testing processes

Process stage	What is tested	Who is testing
Unit testing	Program units in isolation. All of the code in a unit should be executed at least once.	Programmer who developed that code.
Feature testing	Code units are integrated to create features. All aspects of a feature are tested.	All of the programmers who contribute code units
System testing	A working (perhaps incomplete) version of a system. Checks that there are no unexpected interactions between the features in the system. May also involve checking the responsiveness, reliability and security of the system.	Dedicated testing team in large companies. Product developers in small companies.
Release testing	Packaged system, ready for release. Check that it operates as expected in specific environments (ex. cloud, computer, mobile device).	A separate team. The development team, if DevOps is used

Unit testing

Unit testing is the process of testing ***individual units in isolation***.

- It is a *defect* testing process.

Code unit = anything that has a clearly defined responsibility,

and may be:

  - Individual *function* or method within an object;
  - Object *class* with several attributes and methods;
  - *Module*, with defined interfaces used to access its 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*.

Obs. Inheritance makes it more difficult to design object class tests as the information to be tested is not localized.

Because – all elements of the class must be tested, including the inherited ones, in the particular context of the tested class.

Unit testing effectiveness

Test cases are defined for the developed code.

Test case = specification of the inputs to the test and the expected output from the system plus a statement of what is being tested.

Test cases objectives:

- show that, when used as expected, the tested unit does what it is supposed to do.
- reveal defects in the unit.

$\Downarrow$

Types of unit test case:

- reflect normal operation of a program and should show that the unit works as expected.
- use abnormal inputs to check that these are properly processed and do not crash the unit.

Testing startegies

- **Partition testing** - identify *groups of inputs* that have *common characteristics* and should be processed in the same way.
  - Tests are chosen from within each of these groups.

- **Guideline-based testing** - use testing guidelines to choose test cases.
  - Guidelines reflect previous experience of the kinds of errors that programmers often make when developing software.

Partition testing

General principle:

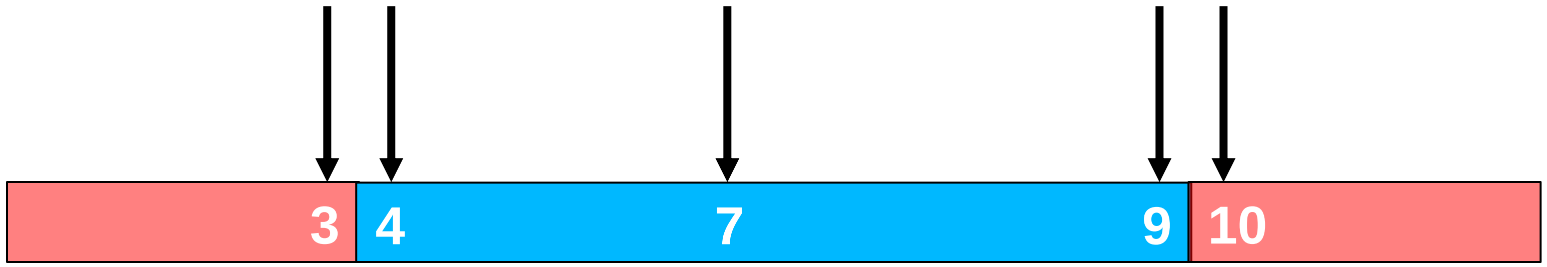
If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for a larger set whose members share these characteristics.

- Input data and output results generally fall into different groups, where all members of a group are related as regards the behavior of the program.
- Each of these groups is an ***equivalence partition*** (or *domain*) where *the program behaves in an equivalent way for each group member*.
- Test cases should be chosen from *each* partition.

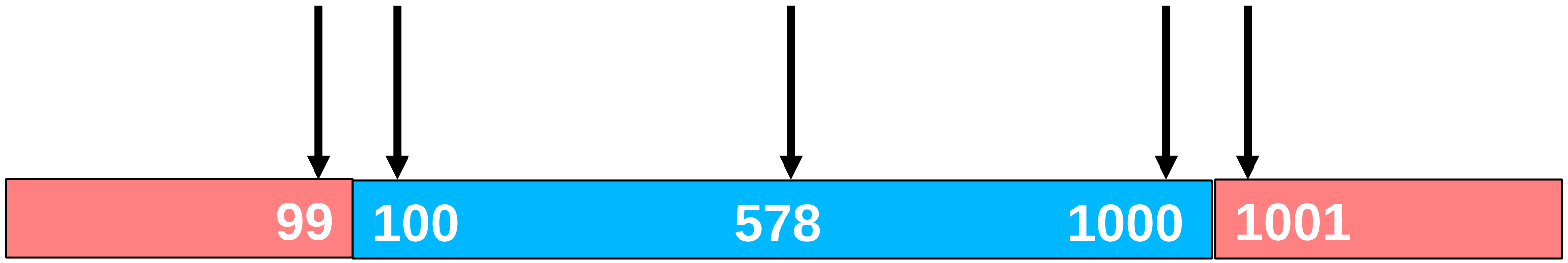
Procedure:

- Identification of the equivalence partitions as sets of inputs that will be treated in the same way in the code.
- Identification of ‘incorrectness partitions’, where the inputs are deliberately incorrect.

Example: Equivalence partitions



number of input values : *ni* > 3 and *ni* < 10



input value : *iv* ≥ 100 and *iv* ≤ 1000

Equivalence partitions for a name checking function

- **Unit testing**
- Feature testing
- System testing
- Release testing

```
 def namecheck (s):
# Checks that a name only includes
# alphabetic characters, - or a single quote.
# Names must be between 2 and 40
# characters long
# quoted strings and -- are disallowed


  namex = r"^[a-zA-Z][a-zA-Z-'']{1,39}$"
  if re.match (namex, s):
   if re.search ("'.*'", s) or re.search ("--", s):
     return False
   else:
     return True
  else:
   return False
```

Correct names 1 : The inputs only includes alphabetic characters and are between 2 and 40 characters long.

Correct names 2 : The inputs only includes alphabetic characters, hyphens or apostrophes and are between 2 and 40 characters long.

Incorrect names 1 : The inputs are between 2 and 40 characters long but include disallowed characters.

Incorrect names 2 : The inputs include allowed characters but are either a single character or are more than 40 characters long.

Incorrect names 3 : The inputs are between 2 and 40 characters long but the first character is a hyphen or an apostrophe.

Incorrect names 4 : The inputs include valid characters, are between 2 and 40 characters long, but include either a double hyphen, quoted text or both.

Example: Search routine specification

```
procedure Search (Key : ELEM ; T: SEQ of ELEM;
        Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
-- the sequence has at least one element
T'FIRST <= T'LAST
Post-condition
-- the element is found and is referenced by L
( Found and T (L) = Key)
or
-- the element is not in the array
( not Found and
                not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

Example: Search routine

Input partitions

- Inputs which conform to the pre-conditions.
- Inputs where a pre-condition does not hold.
- Inputs where the key element is a member of the array.
- Inputs where the key element is not a member of the array.

Testing guidelines for sequences

- Test software with sequences which have *only a single value (length =1)*.
- Use sequences of *different sizes* in different tests.
- Derive tests so that the *first*, *middle* and *last* elements of the sequence are accessed.
- Test with sequences of *zero length*.

Example: Search routine - input partitions

Sequence	**Element**	**Input sequence** (T)	**Key** (Key)	**Output element** (Found, L)
Single value	In sequence	17	17	True, 1
Single value	Not in sequence	17	0	False, ??
More then 1 value	First element in sequence	17,29,21,23	17	True, 1
More then 1 value	Last element in sequence	41,18,9,3,30,16,45	45	True, 7
More then 1 value	Middle element in sequence	17,18,21,23,29,41,38	23	True, 4
More then 1 value	Not in sequence	21,23,29,33,38	25	False, ??
Empty	-	-	10	False, ??

Unit testing guidelines (1)

Testing guidelines are hints for the testing team to help them ***choose tests*** that will reveal defects in the system:

Test edge cases

If your partition has upper and lower bounds (e.g. length of strings, numbers, etc.) choose inputs at the edges of the range.

Force errors

Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.

Fill buffers

Choose test inputs that cause all input buffers to overflow.

Repeat yourself

Repeat the same test input or series of inputs several times.

Unit testing guidelines (2)

Overflow and underflow
If your program does numeric calculations, choose test inputs that cause it to calculate very large or very small numbers.

Don’t forget null and zero
If your program uses pointers or strings, always test with null pointers and strings. If you use sequences, test with an empty sequence. For numeric inputs, always test with zero.

Keep count
When dealing with lists and list transformation, keep count of the number of elements in each list and check that these are consistent after each transformation.

One is different
If your program deals with sequences, always test with sequences that have a single value.

Automated unit testing

- Automated unit testing ⇒ executable tests that run and check without manual intervention.
- A test automation framework e.g. PyUnit, JUnit) is used to *write* and *run* the program tests.
- Unit testing frameworks
  - provide *generic test classes* that must be *extended* to create *specific test cases*.
  - *run all of the tests* that have been implemented and *report test results*.

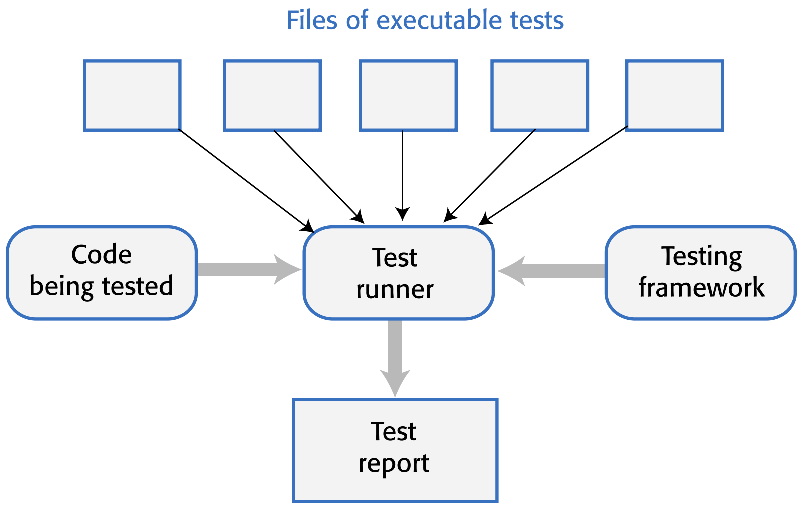


Figure 9.4 Automated testing
Ian Sommerville – Engineering Software Products

Automated unit test structure

Automated tests are structured into three parts:

- **Setup (arrange)** part

System initialization with the test case, namely the inputs and expected outputs. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.

- **Call (action)** part

Call the unit that is being tested with the test parameters.

- **Assertion** part

Compare the result of the call with the expected result.

If the assertion evaluates to true, the test has been successful if false, then it has failed.

Obs.

1. If equivalence partitions are used, several tests based on correct and incorrect inputs from each partition should be designed.
2. Normally, thousands of executable tests are developed for an industrial software product.

- **Unit testing**
- Feature testing
- System testing
- Release testing

Test automation

Example : Test methods for an interest calculator

```
 # TestInterestCalculator inherits attributes and methods from the class TestCase in the testing framework
# unittest (PyUnit)

 class TestInterestCalculator (unittest.TestCase):
        # Define a set of unit tests where each test tests one thing only
        # Tests should start with test_ and the name should explain what is being tested
        def test_zeroprincipal (self):
                #Arrange - set up the test parameters
                p = 0; r = 3; n = 31
                result_should_be = 0
                #Action - Call the method to be tested
                interest = interest_calculator (p, r, n)
                #Assert - test what should be true
                self.assertEqual (result_should_be, interest)

        def test_yearly_interest (self):
                #Arrange - set up the test parameters
                p = 17000; r = 3; n = 365
                result_should_be = 270.36
                #Action - Call the method to be tested
                interest = interest_calculator (p, r, n)
                #Assert - test what should be true
                self.assertEqual (result_should_be, interest)
```

- **Unit testing**
- Feature testing
- System testing
- Release testing

Automated unit tests

Example : Executable tests for namecheck function (1)

```
import unittest
from RE_checker import namecheck
class TestNameCheck (unittest.TestCase):

    def test_alphaname (self):
            self.assertTrue (namecheck ('Sommerville'))

    def test_doublequote (self):
            self.assertFalse (namecheck ("Thisis'maliciouscode'"))

    def test_namestartswithhyphen (self):
            self.assertFalse (namecheck ('-Sommerville'))

    def test_namestartswithquote (self):
            self.assertFalse (namecheck ("'Reilly"))

    def test_nametoolong (self):
            self.assertFalse (namecheck ('Thisisalongstringwithmorethen40charactersfrombeginningtoend'))

    def test_nametooshort (self):
            self.assertFalse (namecheck ('S'))

    def test_namewithdigit (self):
            self.assertFalse (namecheck('C-3PO'))
```

Automated unit tests

Example : Executable tests for namecheck function (2)

```
 def test_namewithdoublehyphen (self):
                 self.assertFalse (namecheck ('--badcode'))

 def test_namewithhyphen (self):
                 self.assertTrue (namecheck ('Washington-Wilson'))

def test_namewithinvalidchar (self):
                 self.assertFalse (namecheck('Sommer_ville'))

def test_namewithquote (self):
                 self.assertTrue (namecheck ("O'Reilly"))

def test_namewithspaces (self):
                 self.assertFalse (namecheck ('Washington Wilson'))

def test_shortname (self):
                 self.assertTrue ('Sx')

def test_thiswillfail (self):
                 self.assertTrue (namecheck ("O Reilly"))
```

Automated unit tests

Code to run unit tests from files

```
import unittest

loader = unittest.TestLoader()

#Find the test files in the current directory

tests = loader.discover('.')

#Specify the level of information provided by the test runner

testRunner = unittest.runner.TextTestRunner(verbosity=2)
testRunner.run(tests)
```

Formative evaluation

1. Why automated testing is useful ?
2. What is the role of equivalence partitions in unit testing ?
3. Give examples of criteria for selecting tests that reveal defects in the system.
4. Which are the elements of the automated unit test ?

https://forms.gle/8aYS96jjTajvPN47A

Feature (requirements) testing

Features have to be tested to show that the functionality is implemented as expected and that the functionality meets the real needs of users.

For example, if a product has a feature that allows users to login using their Google account, then one has to check that this registers the user correctly and informs them of what information will be shared with Google.

It may be also checked that it gives users the option to sign up for email information about the product.

Normally, a feature that does several things is implemented by multiple, interacting, program units.

These units may be implemented by different developers and all of these developers should be involved in the feature testing process.

Types of feature test

Interaction tests :

Test

- the interactions between the units that implement the feature.

Reveal :

- Different understandings, by the developers of the units that are combined to make up the feature, about of what is required of that feature.
- Bugs in program units, which were not exposed by unit testing.

Usefulness tests :

Test

- Conformance of the feature implementation with what users are likely to want.

For example, the developers of a login with Google feature may have implemented an opt-out default on registration so that users receive all emails from a company. They must expressly choose what type of emails that they don’t want.

What might be preferred is an opt-in default so that users choose what types of email they do want to receive.

Feature test

Example : User stories for the sign-in Google feature

User registration

As a user, **I want to** be able to login without creating a new account **so that** I don't have to remember another login id and password.

Information sharing

As a user**, I want t**o know what information you will share with other companies. **I want to** be able to cancel my registration if I don't want to share this information.

Email choice

As a user, **I want to** be able to choose the types of email that I'll get from you when I register for an account.

Guideline: Design a set of tests including valid and invalid inputs.

Feature test

Example : Tests for the sign-in Google feature

Initial login screen

- Test that the screen displaying a request for Google account credentials is correctly displayed when a user clicks on the ‘Sign-in with Google’ link.
- Test that the login is completed if the user is already logged in to Google.

Incorrect credentials

- Test that the error message and retry screen is displayed if the user inputs incorrect Google credentials.

Shared information

- Test that the information shared with Google is displayed, along with a cancel or confirm option.
- Test that the registration is cancelled if the cancel option is chosen.

Email opt-in

- Test that the user is offered a menu of options for email information and can choose multiple items to opt-in to emails.
- Test that the user is not registered for any emails if no options are selected.

Automated feature testing

Design the product so that its *features* can be *directly accessed through an API.*

$\Downarrow$

The *feature tests* can access features directly through the API.

Benefits:

- Less expensive than to automated testing the GUI
- Possibility to re-implement the GUI without changing the functional components of the software.

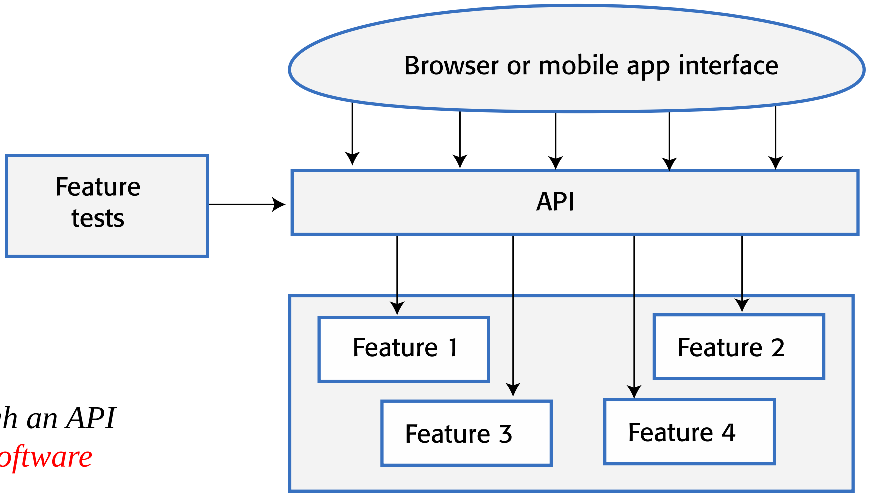


Figure 9.6 Feature editing through an API
Ian Sommerville – Engineering Software Products

System testing

System testing during development involves :

- integrating components to create a version of the system
- testing the integrated system.

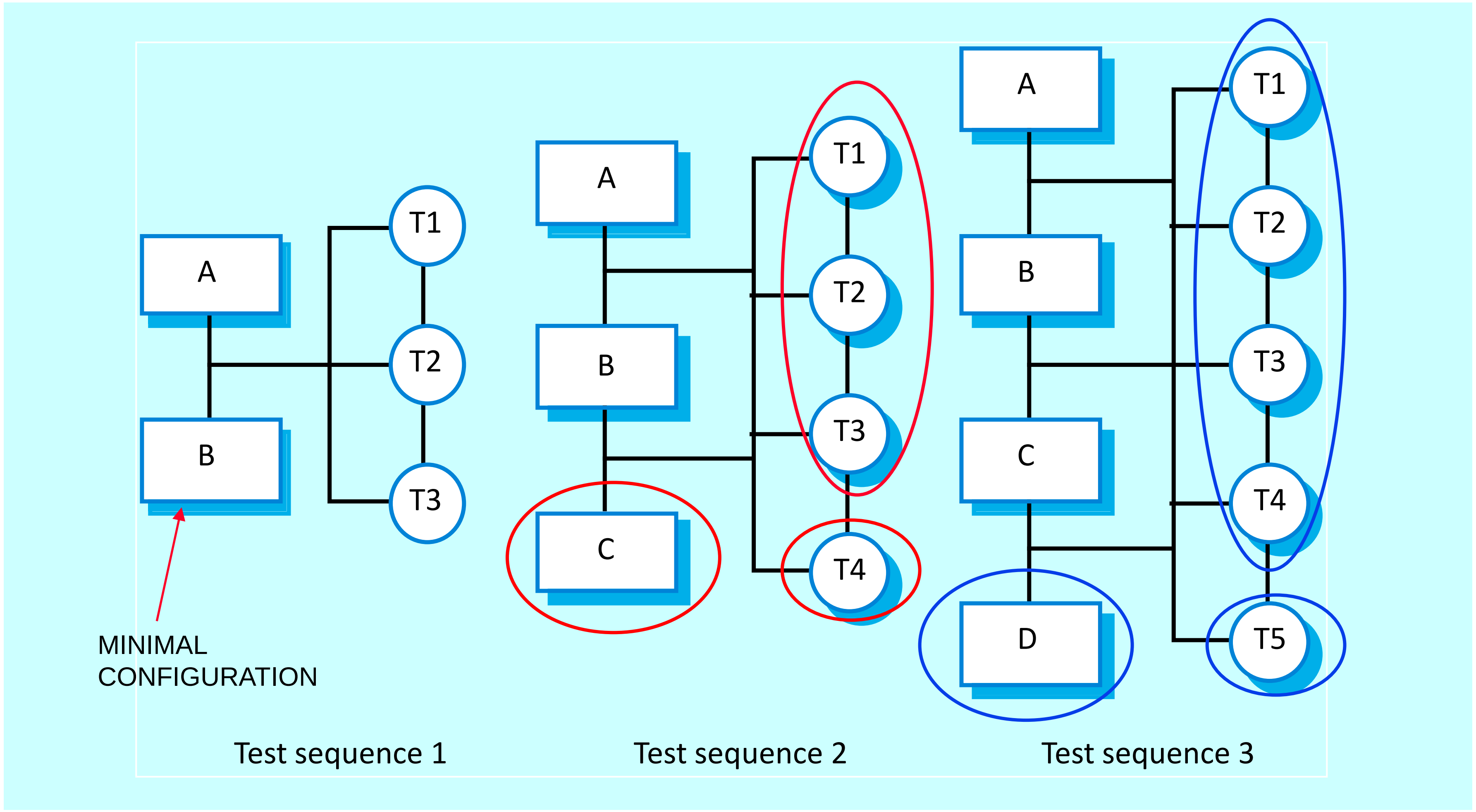
Obs. May involve testing an increment to be delivered to the customer.

- Focus - testing the *interactions* between components.
- Check - components *compatiblity*, components *interaction* and correctness of *data transfer* (the right data at the right time across the interfaces).
- Test - the *emergent* behaviour and qualities of the system.

- Unit testing
- Feature testing
- **System testing**
- Release testing

Incremental integration testing

IMPORTANT! : To simplify error localization,systems should be incrementally integrated.



Regression testing

Software regression **: software bug or quality altering, as a consequence of a software change.**

- Software regression testing - rerunning an existing set of tests.

$$\Downarrow$$

- When problems appear check for their source in:
  - added increment of functionality.
  - the previous increment.

Automated testing frameworks (ex. JUnit) allows automatically rerun of the tests.

All tests are rerun every time a change is made to the program.

Tests must run ‘successfully’ before the change is committed.

System testing

System testing also implies testing the *emergent* behaviour and qualities of the system.

- Testing to discover if there are unexpected and unwanted interactions between the features in a system.
- Testing to discover if the system features work together effectively to support what users really want to do with the system.
- Testing the system to make sure it operates in the expected way in the different environments where it will be used.
- Testing the responsiveness, throughput, security and other quality attributes of the system.

Scenario-based testing

To systematically test a system :

- start with a set of scenarios that describe possible uses of the system
- work through these scenarios each time a new version of the system is created.

Using the scenario, identify a set of end-to-end pathways that users might follow when using the system.

End-to-end pathway = sequence of actions from starting to use the system for the task, through to completion of the task.

- Unit testing
- Feature testing
- **System testing**
- Release testing

Scenario-based testing

Example : Choosing a holiday destination

Andrew and Maria have a two year old son and a four month old daughter. They live in Scotland and they want to have a holiday in the sunshine. However, they are concerned about the hassle of flying with young children. They decide to try a family holiday planner product to help them choose a destination that is easy to get to and that fits in with their childrens’ routines.

Maria navigates to the holiday planner website and selects the ‘find a destination’ page. This presents a screen with a number of options. She can choose a specific destination or can choose a departure airport and find all destinations that have direct flights from that airport. She can also input the time band that she’d prefer for flights, holiday dates and a maximum cost per person.

Edinburgh is their closest departure airport. She chooses ‘find direct flights’. The system then presents a list of countries that have direct flights from Edinburgh and the days when these flights operate. She selects France, Italy, Portugal and Spain and requests further information about these flights. She then sets a filter to display flights that leave on a Saturday or Sunday after 7.30am and arrive before 6pm.

She also sets the maximum acceptable cost for a flight. The list of flights is pruned according to the filter and is redisplayed. Maria then clicks on the flight she wants. This opens a tab in her browser showing a booking form for this flight on the airline’s website.

Scenario-based testing

Example : End-to-end pathways

1. User inputs departure airport and chooses to see only direct flights. User quits.

2. User inputs departure airport and chooses to see all flights. User quits.

3. User chooses destination country and chooses to see all flights. User quits.

4. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User quits.

5. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User selects a displayed flight and clicks through to airline website. User returns to holiday planner after booking flight.

Automation of system testing

System testing involves testing the system as a *surrogate user*.

Actions involved :

- select items from menus,
- make screen selections,
- input information from the keyboard, etc.

Objective

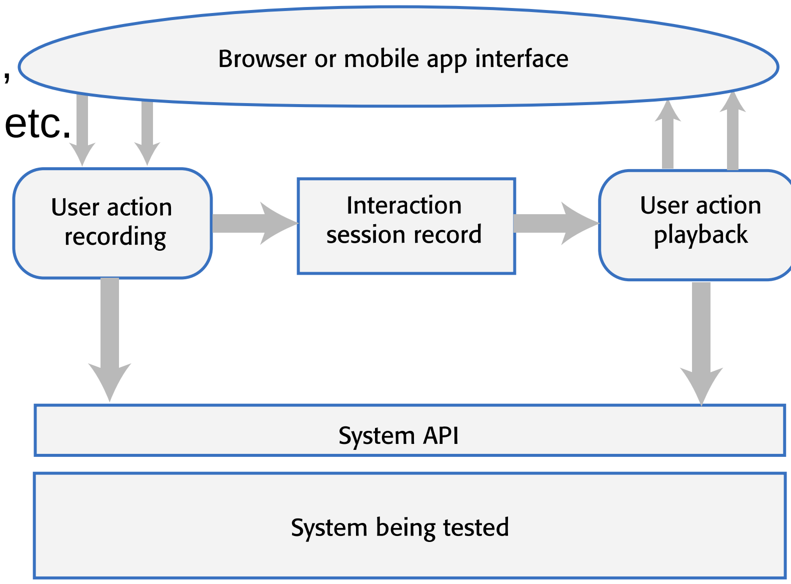
- find interactions between features that cause problems,
- find sequences of actions that lead to system crashes, etc.

Manual system testing is boring and error-prone.

In some cases, the timing of actions is important and is practically impossible to repeat consistently.

Specific testing tools can *record* a series of actions and automatically replay these when a system is retested.

Figure 9.7 Interaction recording and playback
Ian Sommerville – Engineering Software Products



Release testing

Release testing is *a type of system testing* where a system that's intended for release to customers is tested.

Fundamental differences between release testing and system testing:

- Release testing tests the system in its real operational environment rather than in a test environment. Problems commonly arise with real user data, which is sometimes more complex and less reliable than test data.
- The aim of release testing is to decide if the system is good enough to release, not to detect bugs in the system. Therefore, some tests that 'fail' may be ignored if these have minimal consequences for most users.

Preparing a system for release involves :

- packaging that system for deployment (e.g. in a container if it is a cloud service)
- installing software and libraries that are used by the product
- defining configuration parameters (e.g. the name of a root directory, the database size limit per user, etc.).

Formative evaluation

1. What do we need to consider when we design the software product, so that we can automatically and efficiently test its features ?

https://forms.gle/vjtYQAvpK8yC7eLM9

Topics covered

Verification and validation

Testing

Functional testing

Test-driven development

Quality testing

Test-driven development

- Test-driven development (TDD) - approach to program development in which testing and code development are inter-leaved.
- Tests are written before code and ‘passing’ the tests is the critical driver of development.
- The code is developed incrementally, along with a test for that increment. The next increment starts after the code of the current increment has passed 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.
- Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.
- Even the strongest advocates of TDD accept that it is challenging to use this approach when developing and testing systems with graphical user interfaces.

TDD process activities

Identify new unit (part) of functionality
Break down the functionality required into smaller units. Choose one of these units for implementation.

Write unit tests
Write one or more automated tests for the unit chosen for implementation. The unit should pass these tests if it is properly implemented.

Write a code stub that will fail test
Write incomplete code that will be called to implement the unit. This will fail.

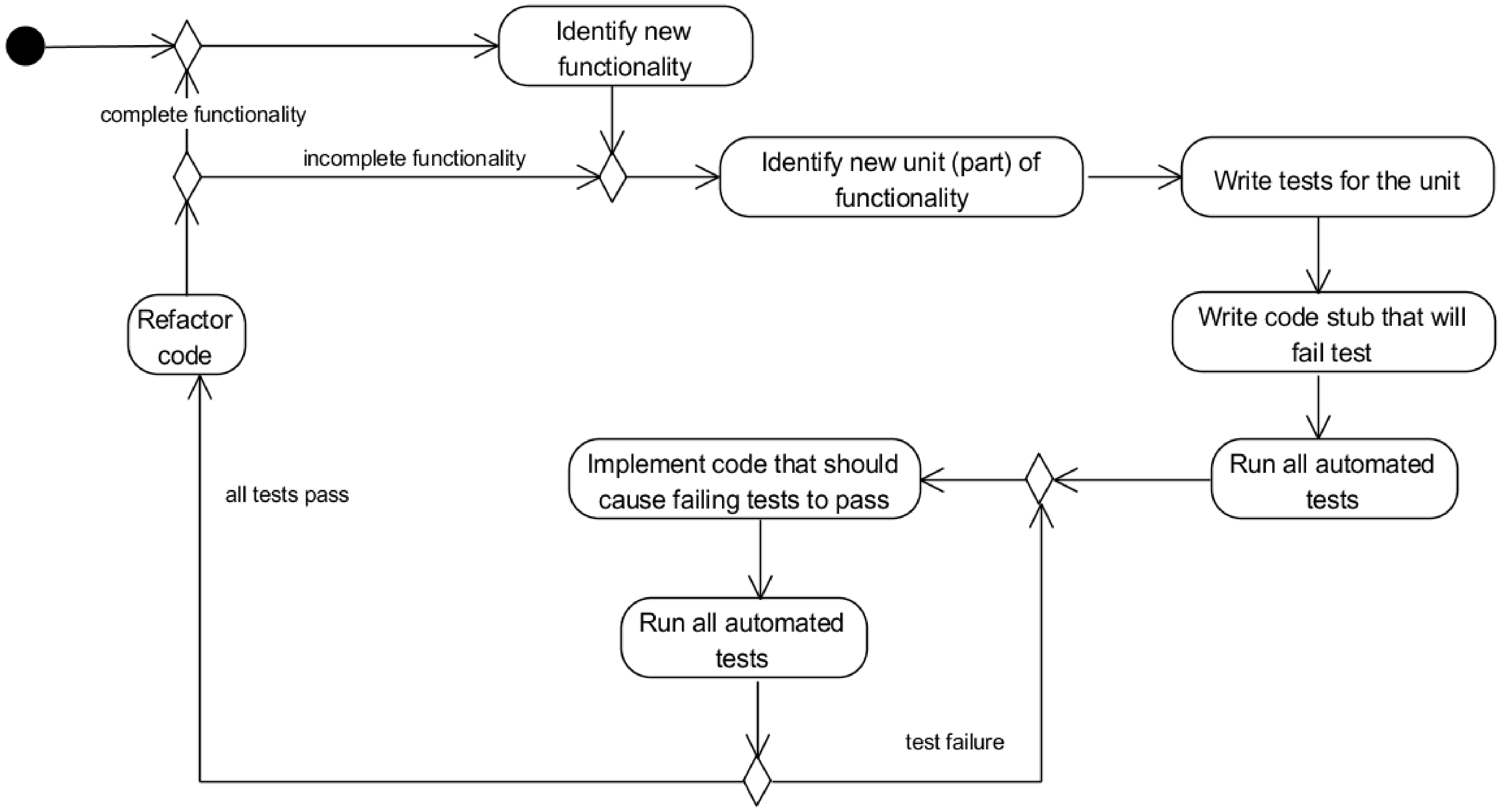
Run all existing automated tests
All previous tests should pass. The test for the incomplete code should fail.

Implement code that should cause the failing test to pass
Write code to implement the unit, which should cause it to operate correctly.

Rerun all automated tests
If any tests fail, the code is probably incorrect. Change it until all tests pass.

Refactor code
Look for ways of improving the code without changing its functionality.

Test-driven process


Identify new functionality
complete functionality
incomplete functionality
Identify new unit (part) of functionality
Write tests for the unit
Write code stub that will fail test
Run all automated tests
Implement code that should cause failing tests to pass
Run all automated tests
test failure
all tests pass
Refactor code

Benefits of TDD

- Code coverage
  - Every written code segment 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.

Some problems with TDD

TDD discourages radical program refactoring.

Developer tend to focus on the tests rather than the problem to be solved.

Too much time dedicated to implementation details rather than to the programming problem.

It is practically impossible to anticipate all of the data problems that might arise and write tests for these in advance.

Topics covered

Verification and validation

Testing

Functional testing

Test-driven development

Quality testing

Performance testing

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

- Performance tests usually involve designing a series of tests where *the load is steadily increased* until the system performance becomes unacceptable.

Stress testing

- Exercises the system *beyond its maximum design load*.
- Stressing the system often causes defects to come to light.
- Stressing the system will test failure behaviour. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.

Security testing

Security testing aims to find vulnerabilities that may be exploited by an attacker and to provide convincing evidence that the system is sufficiently secure.

The tests should demonstrate that the system can resist attacks like:

- malware injection
- users’ data and identity corruption
- users’ data and identity steal.

Comprehensive security testing requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities.

Risk-based security testing

A risk-based approach to security testing involves

- identifying common risks
- developing tests to demonstrate that the system protects itself from these risks.

It may be possible to construct automated tests for some of these checks, but others inevitably involve manual checking of the system’s behaviour and its files.

Automated tools, that scan the system to check for known vulnerabilities (e.g. detect unused HTTP ports being left open), exist.

Examples of security risks

- *Unauthorized* attacker gains access to a system *using authorized credentials*
- *Authorized* individual accesses *resources that are forbidden* to them
- *Authentication system fails* to detect unauthorized attacker
- Attacker gains access to database using *SQL poisoning attack*
- Improper management of *HTTP session*
- HTTP session *cookies revealed* to attacker
- Confidential data are *unencrypted*
- Encryption *keys are leaked* to potential attackers

Risk analysis and testing

1. *Analyze* identified risks to assess how they might arise.

Example

Risk : Unauthorized attacker gains access to a system using authorized credentials

Causes :

- The user has set *weak passwords* that can be guessed by an attacker.
- The system's *password file* has been *stolen* and passwords discovered by attacker.
- The user has *not* set up *two-factor authentication*.
- An attacker has *discovered credentials* of a legitimate user through social engineering techniques.

2. *Develop tests* to check some of these possibilities.

Example: A test to check that the code that allows users to set their passwords always checks the *strength of passwords*.

Formative evaluation

1. Explain the process of TDD.

2. What is the difference between load testing and stress testing ?

3. Suppose it has been identified the risk that confidential data remain unencrypted. Analyze this risk to identify possible causes. Give examples of tests to check that this risk is avoided.

https://forms.gle/rbGXA5EkKyTy6tiP9

Key points (1)

V&V must be applied at each stage in the software process. Verification is concerned with the discovery of *defects* in a system and validation with the assessment of whether or not the system is *useful* and *useable* in an operational situation.

Software inspections (static V&V) implies analysis of the *static system representation* to discover problems. Software testing (dynamic V&V) implies *executing a program* using artificial data and observing its behaviour.

Both software *functionality* and software *qualities* must be tested.

Testing *stages* are : unit testing, feature testing, system testing, release testing and user testing.

Unit testing involves testing program units such as functions or class methods that have a single responsibility. *Feature testing* focuses on testing individual system features. *System testing* tests the system as a whole to check for unwanted interactions between features and between the system and its environment.

Key points (2)

Identifying *equivalence partitions*, in which all inputs have the same characteristics, and choosing test inputs at the boundaries of these partitions, is an effective way of finding bugs in a program.

User stories may be used as a basis for deriving feature tests.

Test automation is based on the idea that tests should be executable. Wherever possible, write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.

Test-driven development is an approach to development where executable tests are written before the code. Code is then developed to pass the tests.

Performance tests involve steadily increased load until the system performance becomes unacceptable, while *stress testing* exercises the system beyond its maximum design load to test failure behaviour.

Security testing may be risk driven where a list of security risks is used to identify tests that may identify system vulnerabilities.