Testing

Software testing

- Software testing is a process in which you execute your program using data that simulates user inputs.
- You observe its behaviour to see whether or not your program is doing what it is supposed to do.
  - Tests pass if the behaviour is what you expect. Tests fail if the behaviour differs from that expected.
  - If your program does what you expect, this shows that for the inputs used, the program behaves correctly.
- If these inputs are representative of a larger set of inputs, you can infer that the program will behave correctly for all members of this larger input set.

Program bugs

- If the behaviour of the program does not match the behaviour that you expect, then this means that there are bugs in your program that need to be fixed.

- There are two causes of program bugs:

  - ***Programming errors*** You have accidentally included faults in your program code. For example, a common programming error is an ‘off-by-1’ error where you make a mistake with the upper bound of a sequence and fail to process the last element in that sequence.

  - ***Understanding errors*** You have misunderstood or have been unaware of some of the details of what the program is supposed to do. For example, if your program processes data from a file, you may not be aware that some of this data is in the wrong format, so your program doesn’t include code to handle this.

Table 9.1 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.

User 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.

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 obviously depends on the size and the functionality of the application.
- For a business-focused web application, you may have to develop thousands of tests to convince yourself that your product is ready for release to customers.
- Functional testing is a staged activity in which you initially test individual units of code. You integrate code units with other units to create larger units then do more testing.
- The process continues until you have created a complete system ready for release.

Figure 9.2 Functional testing

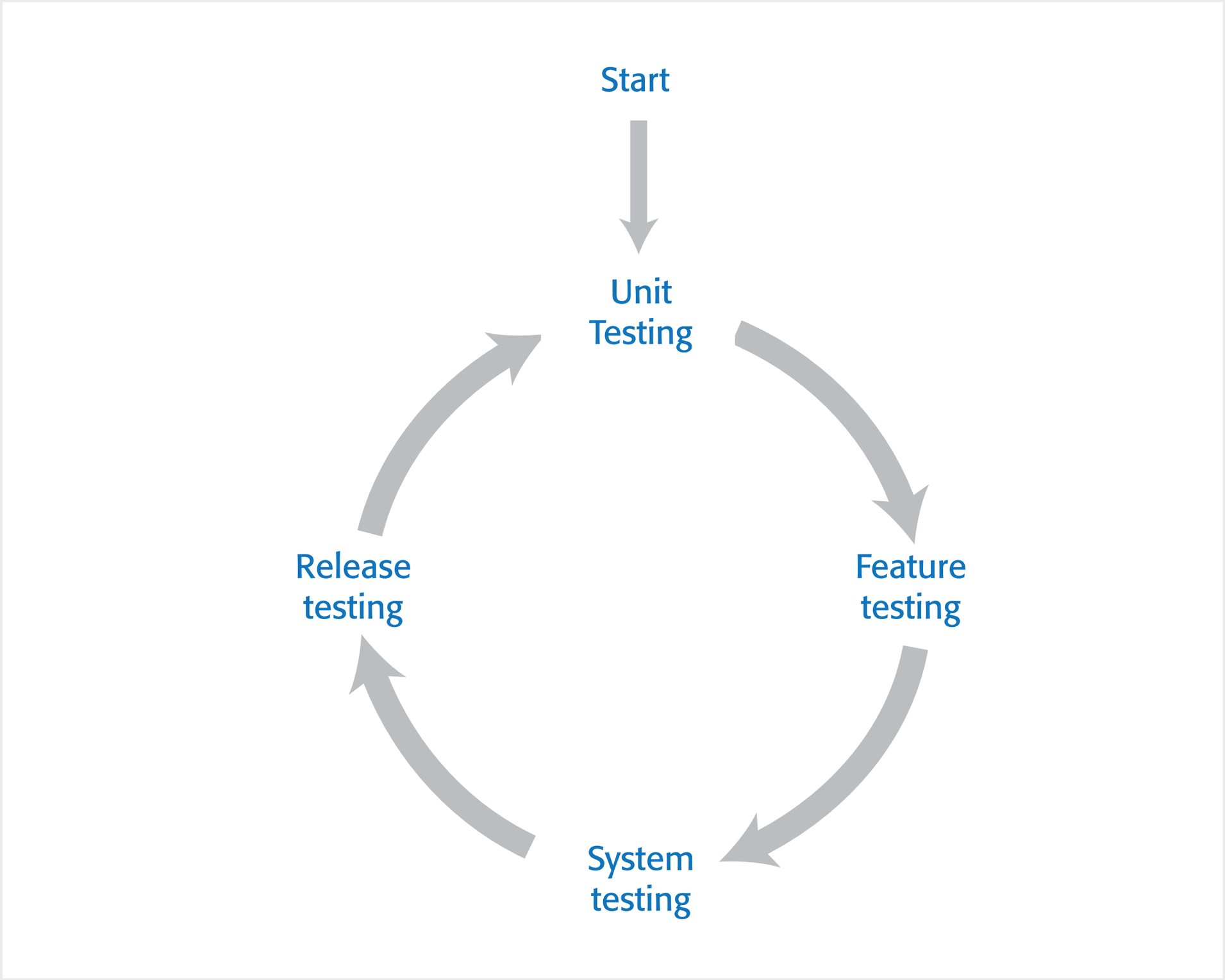


Table 9.2 Functional testing processes

Unit testing
The aim of unit testing is to test program units in isolation. Tests should be designed to execute all of the code in a unit at least once. Individual code units are tested by the programmer as they are developed.

Feature testing
Code units are integrated to create features. Feature tests should test all aspects of a feature. All of the programmers who contribute code units to a feature should be involved in its testing.

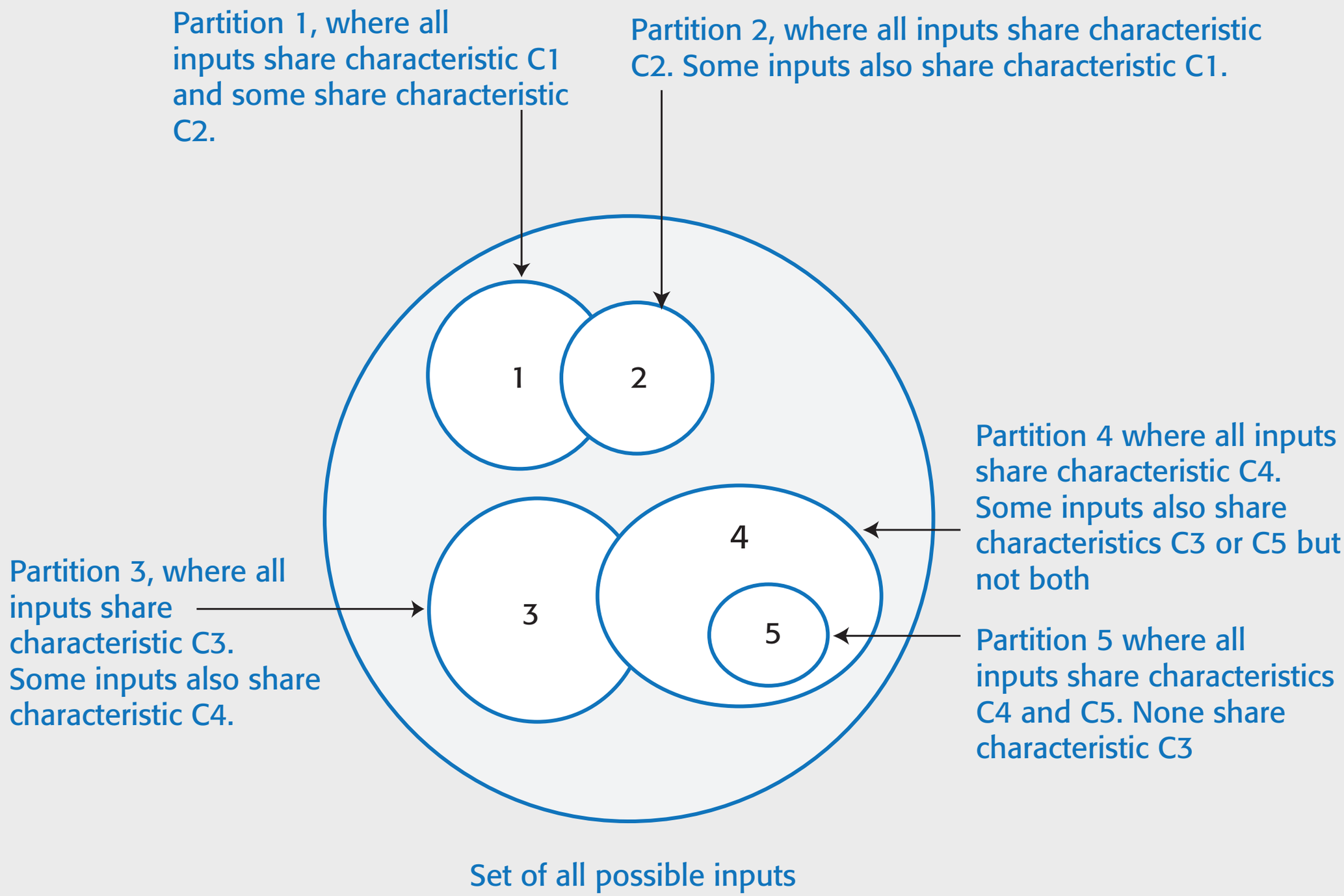
System testing
Code units are integrated to create a working (perhaps incomplete) version of a system. The aim of system testing is to check that there are no unexpected interactions between the features in the system. System testing may also involve checking the responsiveness, reliability and security of the system. In large companies, a dedicated testing team may be responsible for system testing. In small companies, this is impractical, so product developers are also involved in system testing.

Release testing
The system is packaged for release to customers and the release is tested to check that it operates as expected. The software may be released as a cloud service or as a download to be installed on a customer's computer or mobile device. If DevOps is used, then the development team are responsible for release testing otherwise a separate team has that responsibility.

Unit testing

- As you develop a code unit, you should also develop tests for that code.
- A code unit is anything that has a clearly defined responsibility. It is usually a function or class method but could be a module that includes a small number of other functions.
- Unit testing is based on a simple 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.
- To test a program efficiently, you should identify sets of inputs (equivalence partitions) that will be treated in the same way in your code.
- The equivalence partitions that you identify should not just include those containing inputs that produce the correct values. You should also identify ‘incorrectness partitions’ where the inputs are deliberately incorrect.

Figure 9.3 Equivalence partitions



```
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
```

Program 9.1
A name checking function

Table 9.3 Equivalence partitions for the name checking function

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.

Table 9.4 Unit testing guidelines (1)

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.

Table 9.4 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.

Feature 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 your product has a feature that allows users to login using their Google account, then you have to check that this registers the user correctly and informs them of what information will be shared with Google.
  - You may want to check that it gives users the option to sign up for email information about your 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
  - These test the interactions between the units that implement the feature. The developers of the units that are combined to make up the feature may have different understandings of what is required of that feature.
  - These misunderstandings will not show up in unit tests but may only come to light when the units are integrated.
  - The integration may also reveal bugs in program units, which were not exposed by unit testing.
- Usefulness tests
  - These test that the feature implements 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.

Table 9.5 User stories for the sign-in with 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 to 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.

Table 9.6 Feature tests for sign-in with Google

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.

System and release testing

- System testing involves testing the system as a whole, rather than the individual system features.
- System testing should focus on four things:
  - 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

- The best way to systematically test a system is to start with a set of scenarios that describe possible uses of the system and then work through these scenarios each time a new version of the system is created.
- Using the scenario, you identify a set of end-to-end pathways that users might follow when using the system.
- An end-to-end pathway is a sequence of actions from starting to use the system for the task, through to completion of the task.

Figure 9.7 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.

Table 9.8 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.

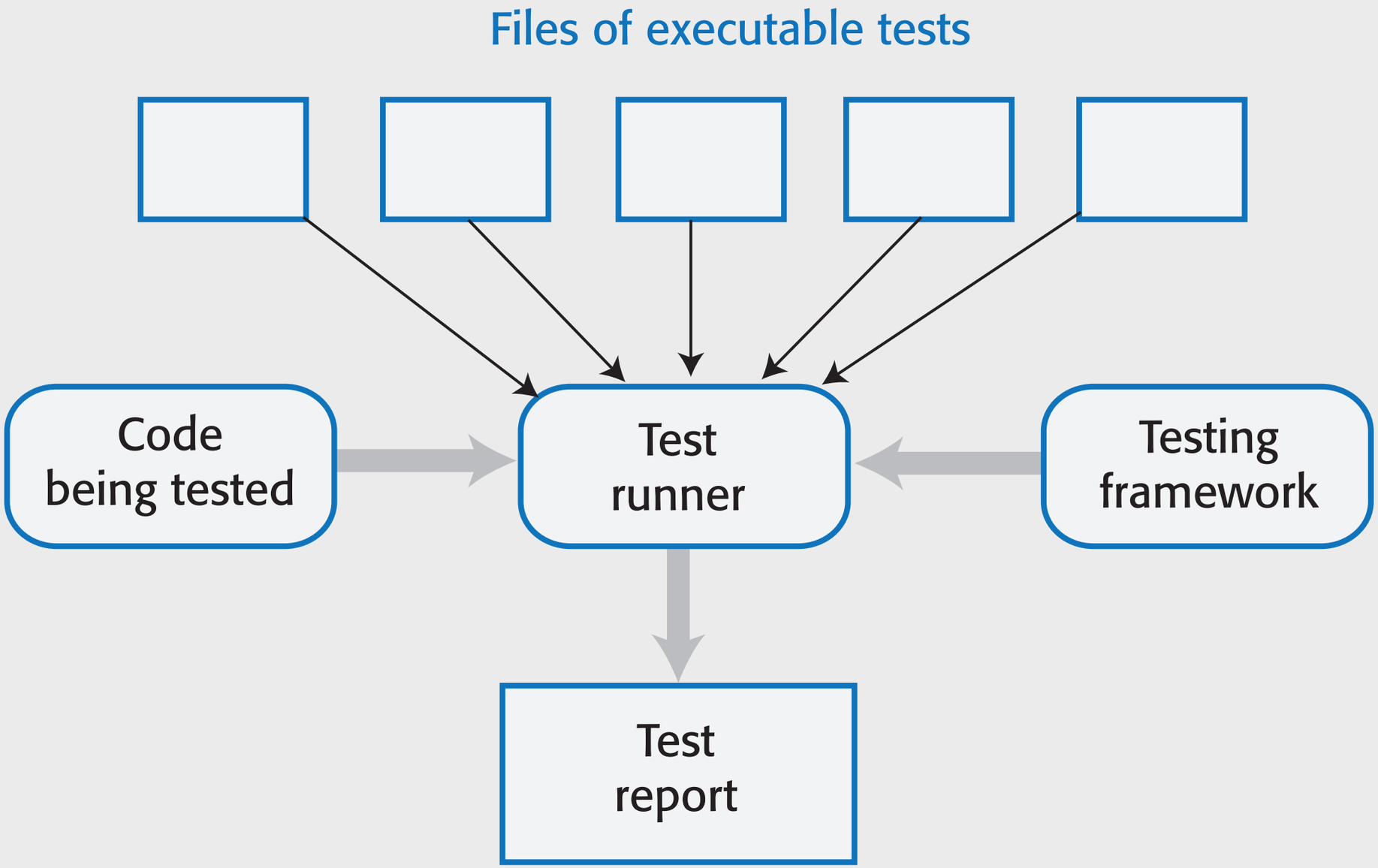
Release testing

- Release testing is a type of system testing where a system that's intended for release to customers is tested.
- The fundamental differences between release testing and system testing are:
  - 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) and installing software and libraries that are used by your product. You must define configuration parameters such as the name of a root directory, the database size limit per user and so on.

Test automation

- Automated testing is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result and a check that the unit returns the expected result.
- You run the test and the test passes if the unit returns the expected result.
- Normally, you should develop hundreds or thousands of executable tests for a software product.

Figure 9.4 Automated testing



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

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
    #Action - Call the method to be tested
    result_should_be = 270.36
    interest = interest_calculator (p, r, n)
    #Assert - test what should be true
    self.assertEqual (result_should_be, interest)
```

Program 9.2 Test methods for an interest calculator

Automated tests

- It is good practice to structure automated tests into three parts:
  - **Arrange** You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.
  - **Action** You call the unit that is being tested with the test parameters.
  - **Assert** You make an assertion about what should hold if the unit being tested has executed successfully. In Program 9.2, I use AssertEquals, which checks if its parameters are equal.
- If you use equivalence partitions to identify test inputs, you should have several automated tests based on correct and incorrect inputs from each partition.

```
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'))

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

Program 9.3 (1)
Executable tests for the namecheck function

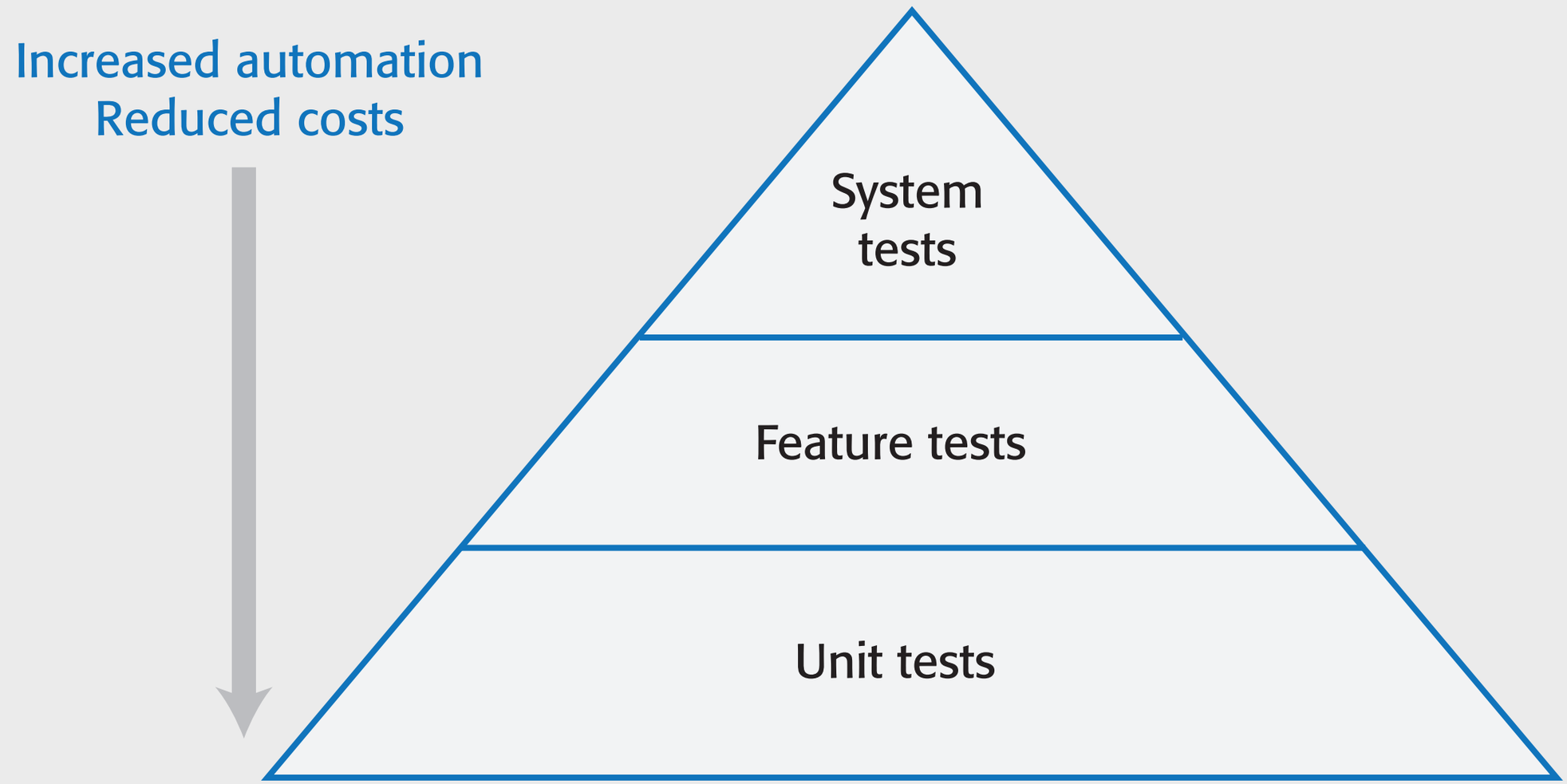
```
def test_namewithhyphen (self):
        self.assertTrue (namecheck ('Washington-Wilson'))

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"))
```

Program 9.3 (2)
Executable tests for the namecheck function

```
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)
```

Program 9.4 Code to run unit tests from files

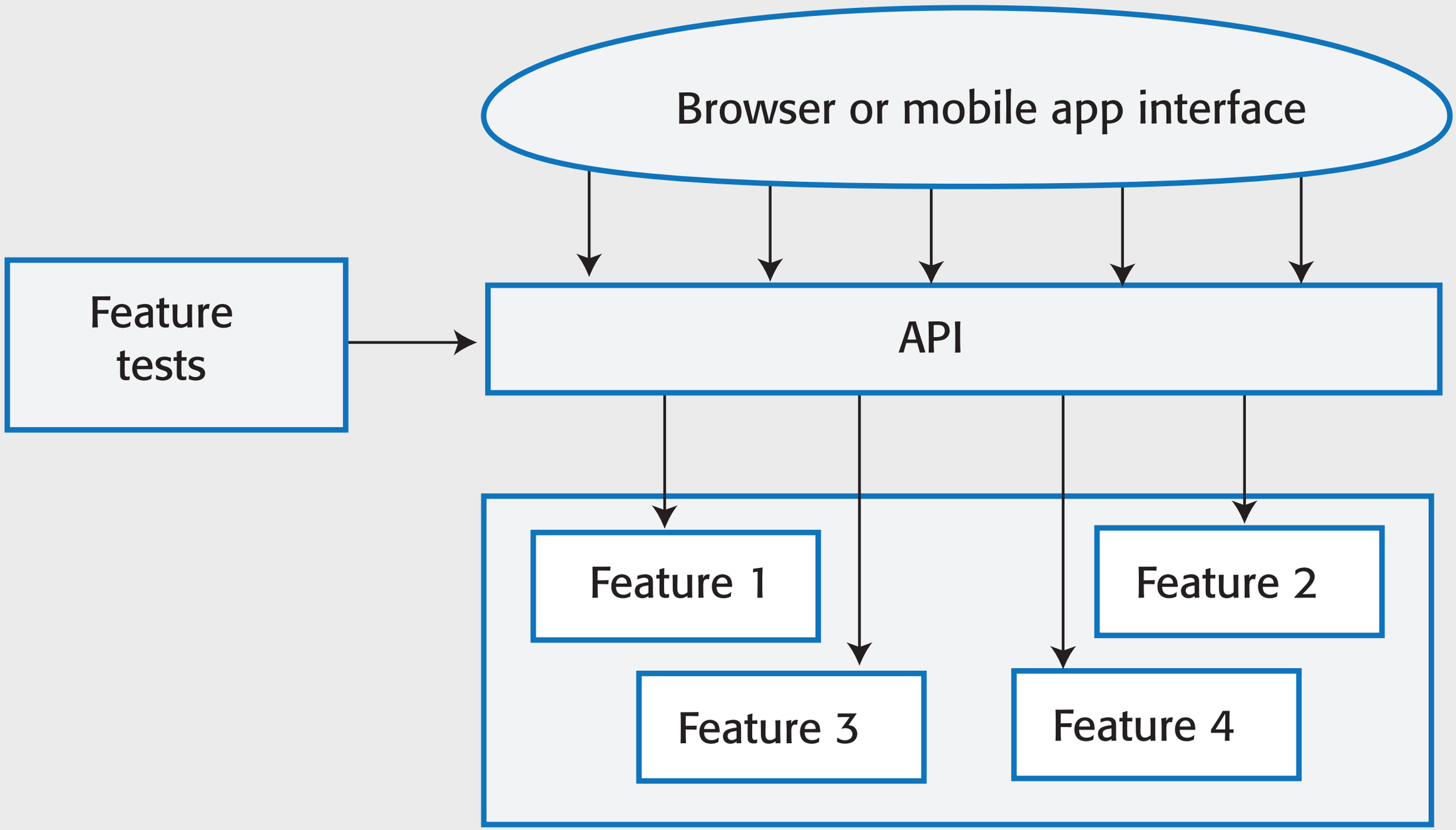
Figure 9.5 The test pyramid



Automated feature testing

- Generally, users access features through the product's graphical user interface (GUI).
- However, GUI-based testing is expensive to automate so it is best to design your product so that its features can be directly accessed through an API and not just from the user interface.
- The feature tests can then access features directly through the API without the need for direct user interaction through the system's GUI.
- Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software.

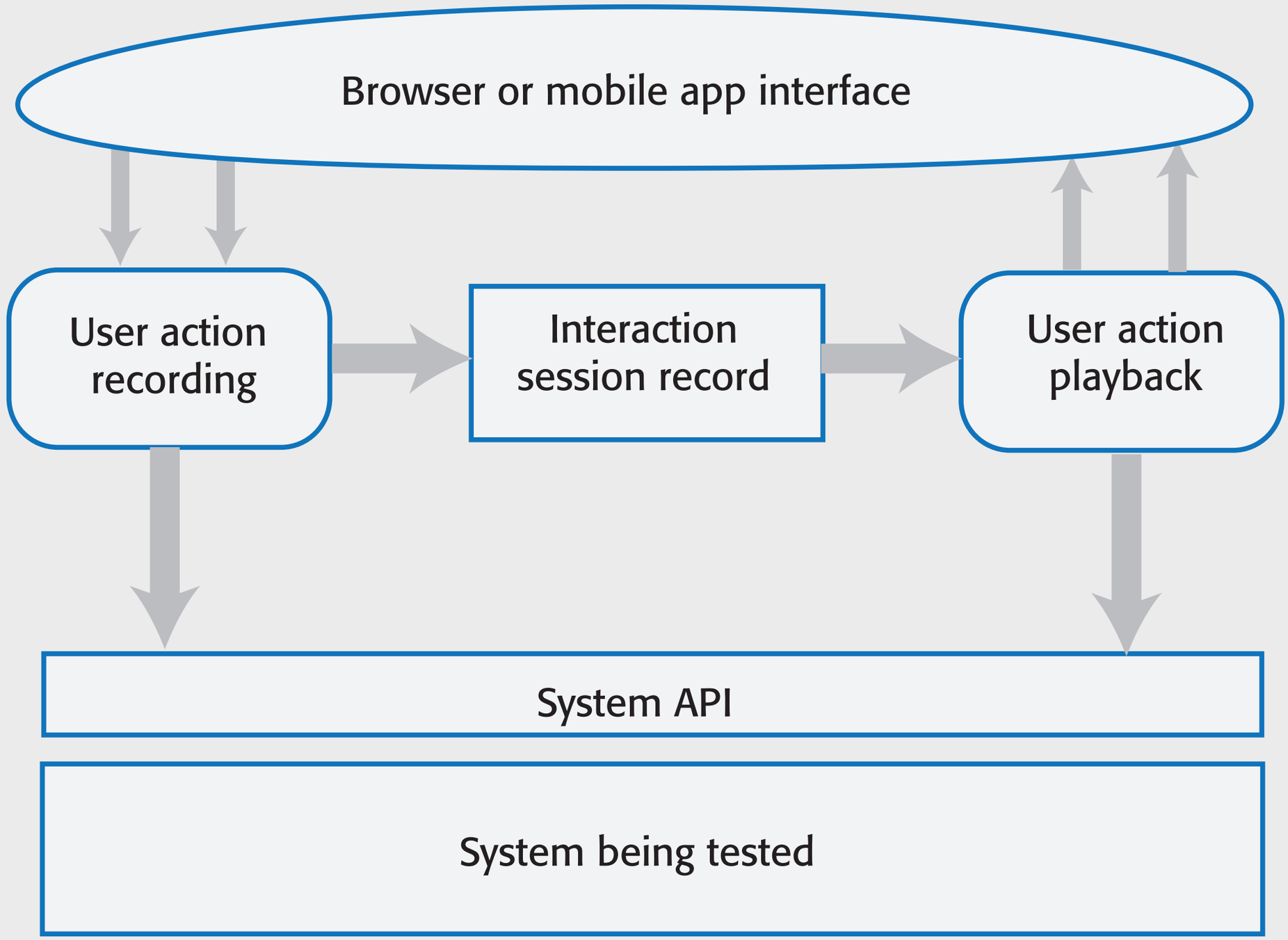
Figure 9.6 Feature editing through an API



System testing

- System testing, which should follow feature testing, involves testing the system as a surrogate user.
- As a system tester, you go through a process of selecting items from menus, making screen selections, inputting information from the keyboard and so on.
- You are looking for interactions between features that cause problems, sequences of actions that lead to system crashes and so on.
- Manual system testing, when testers have to repeat sequences of actions, is boring and error-prone. In some cases, the timing of actions is important and is practically impossible to repeat consistently.
    - To avoid these problems, testing tools have been developed that can record a series of actions and automatically replay these when a system is retested

Figure 9.7 Interaction recording and playback



Test-driven development

- Test-driven development (TDD) is an approach to program development that is based around the general idea that you should write an executable test or tests for code that you are writing before you write the code.
- It was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.
- 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 you are developing and testing systems with graphical user interfaces.

Figure 9.8 Test-driven development

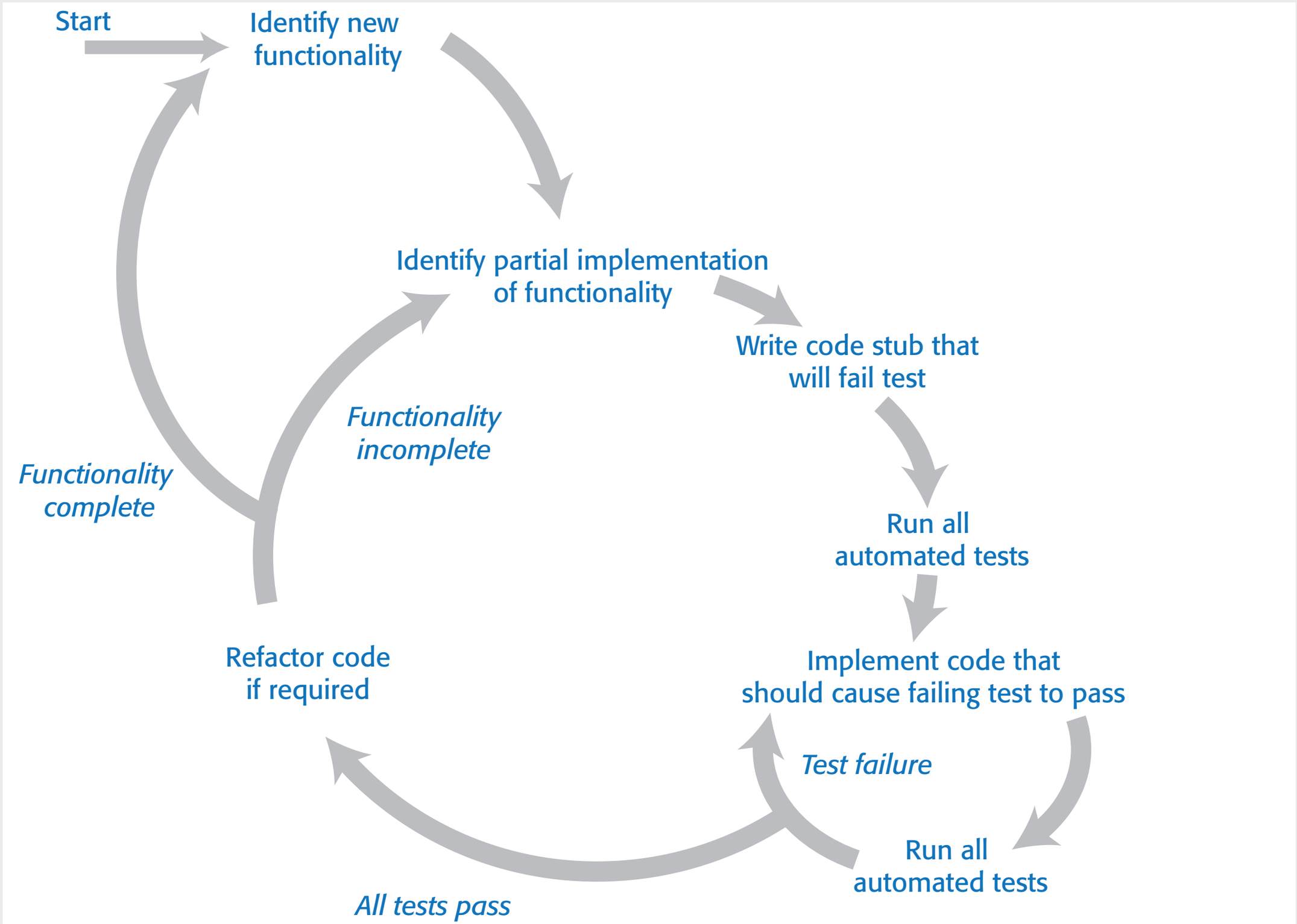


Table 9.9 Stages of test-driven development (1)

Identify partial implementation
Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.

Write mini-unit tests
Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-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 mini-unit. You know this will fail.

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

Table 9.9 Stages of test-driven development (2)

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

Rerun all automated tests
If any tests fail, your code is probably incorrect. Keep working on it until all tests pass.

Refactor code if necessary
If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

Benefits of test-driven development

- It is a systematic approach to testing in which tests are clearly linked to sections of the program code.
    - This means you can be confident that your tests cover all of the code that has been developed and that there are no untested code sections in the delivered code. In my view, this is the most significant benefit of TDD.
- The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests.
- Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system.
- It is argued that TDD leads to simpler code as programmers only write code that's necessary to pass tests. They don't over-engineer their code with complex features that aren't needed.

Table 9.10 My reasons for not using TDD

TDD discourages radical program change
I found that I was reluctant to make refactoring decisions that I knew would cause many tests to fail. I tended to avoid radical program change for this reason.

I focused on the tests rather than the problem I was trying to solve
A basic principle of TDD is that your design should be driven by the tests you have written. I found that I was unconsciously redefining the problem I was trying to solve to make it easier to write tests. This meant that I sometimes didn't implement important checks, because it was difficult to write tests in advance of their implementation.

I spent too much time thinking about implementation details rather than the programming problem
Sometimes when programming, it is best to step back and look at the program as a whole rather than focusing on implementation details. TDD encourages a focus on details that might cause tests to pass or fail and discourages large-scale program revisions.

It is hard to write 'bad data' tests
Many problems involving dealing with messy and incomplete data. It is practically impossible to anticipate all of the data problems that might arise and write tests for these in advance. You might argue that you should simply reject bad data but this is sometimes impractical.

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 on its availability, attacks that try to inject malware and attacks that try to corrupt or steal users' data and identity.
- 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 and developing tests to demonstrate that the system protects itself from these risks.
- You may also use automated tools that scan your system to check for known vulnerabilities, such as unused HTTP ports being left open.
- Based on the risks that have been identified, you then design tests and checks to see if the system is vulnerable.
- 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.

Table 9.11 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

- Once you have identified security risks, you then analyze them to assess how they might arise. For example, for the first risk in Table 9.11 (unauthorized attacker) there are several possibilities:
  - 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.
- You can then develop tests to check some of these possibilities.
  - For example, you might run a test to check that the code that allows users to set their passwords always checks the strength of passwords.

Code reviews

- Code reviews involve one or more people examining the code to check for errors and anomalies and discussing issues with the developer.
- If problems are identified, it is the developer's responsibility to change the code to fix the problems.
- Code reviews complement testing. They 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.

Figure 9.9 Code reviews

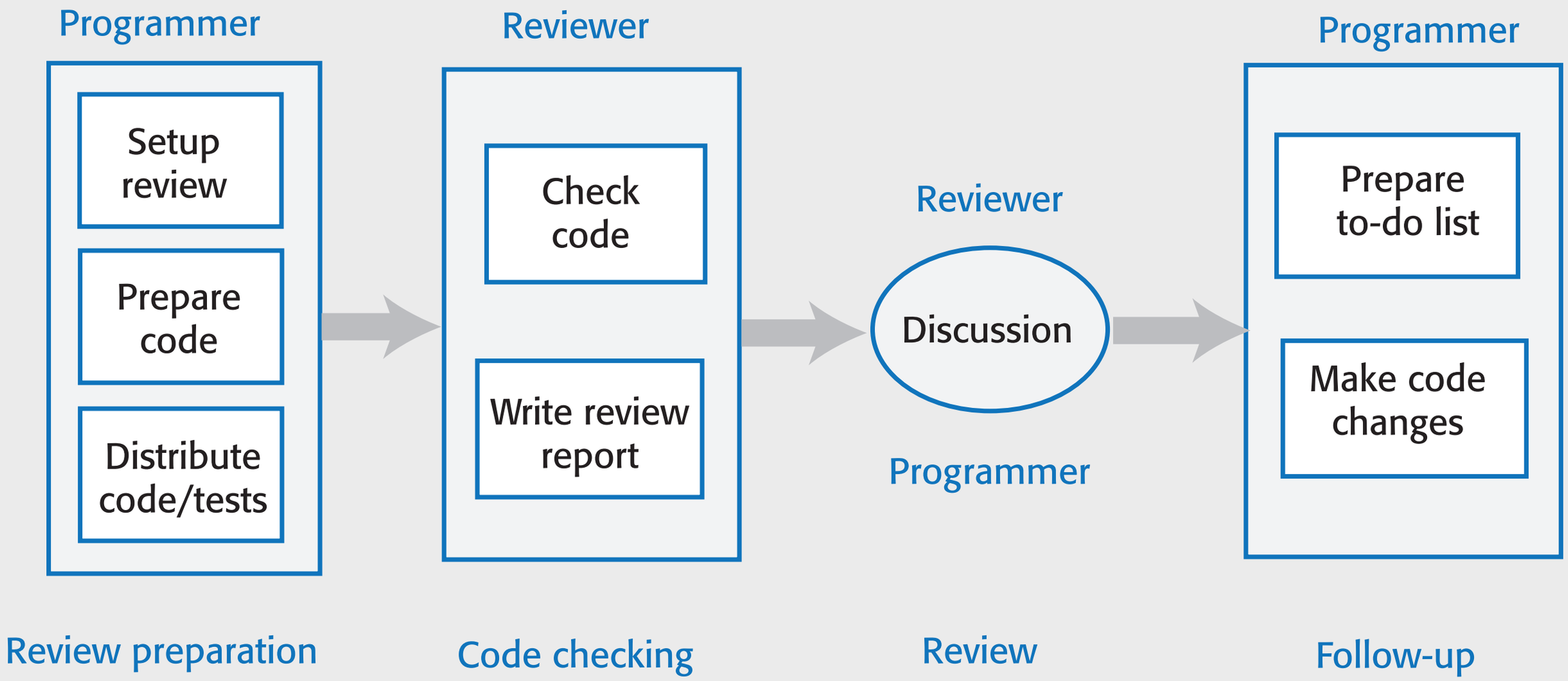


Table 9.12 Code review activities (1)

Setup review
The programmer contacts a reviewer and arranges a review date.

Prepare code
The programmer collects the code and tests for review and annotates them with information for the reviewer about the intended purpose of the code and tests.

Distribute code/tests
The programmer sends code and tests to the reviewer.

Check code
The reviewer systematically checks the code and tests against their understanding of what they are supposed to do.

Write review report
The reviewer annotates the code and tests with a report of the issues to be discussed at the review meeting.

Table 9.12 Code review activities (2)

Discussion
The reviewer and programmer discuss the issues and agree on the actions to resolve these.

Make to-do list
The programmer documents the outcome of the review as a to-do list and shares this with the reviewer.

Make code changes
The programmer modifies their code and tests to address the issues raised in the review.

Table 9.13 Part of a checklist 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.

Key points 1

- The aim of program testing is to find bugs and to show that a program does what its developers expect it to do.
- Four types of testing that are relevant to software products are functional testing, user testing, load and performance testing and security 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.
- 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. You develop a set of executable tests and run these each time you make a change to a system.

Key points 2

- The structure of an automated unit test should be arrange-action-assert. You set up the test parameters, call the function or method being tested, and make an assertion of what should be true after the action has been completed.
- Test-driven development is an approach to development where executable tests are written before the code. Code is then developed to pass the tests.
- A disadvantage of test-driven development is that programmers focus on the detail of passing tests rather than considering the broader structure of their code and algorithms used.
- Security testing may be risk driven where a list of security risks is used to identify tests that may identify system vulnerabilities.
- Code reviews are an effective supplement to testing. They involve people checking the code to comment on the code quality and to look for bugs.