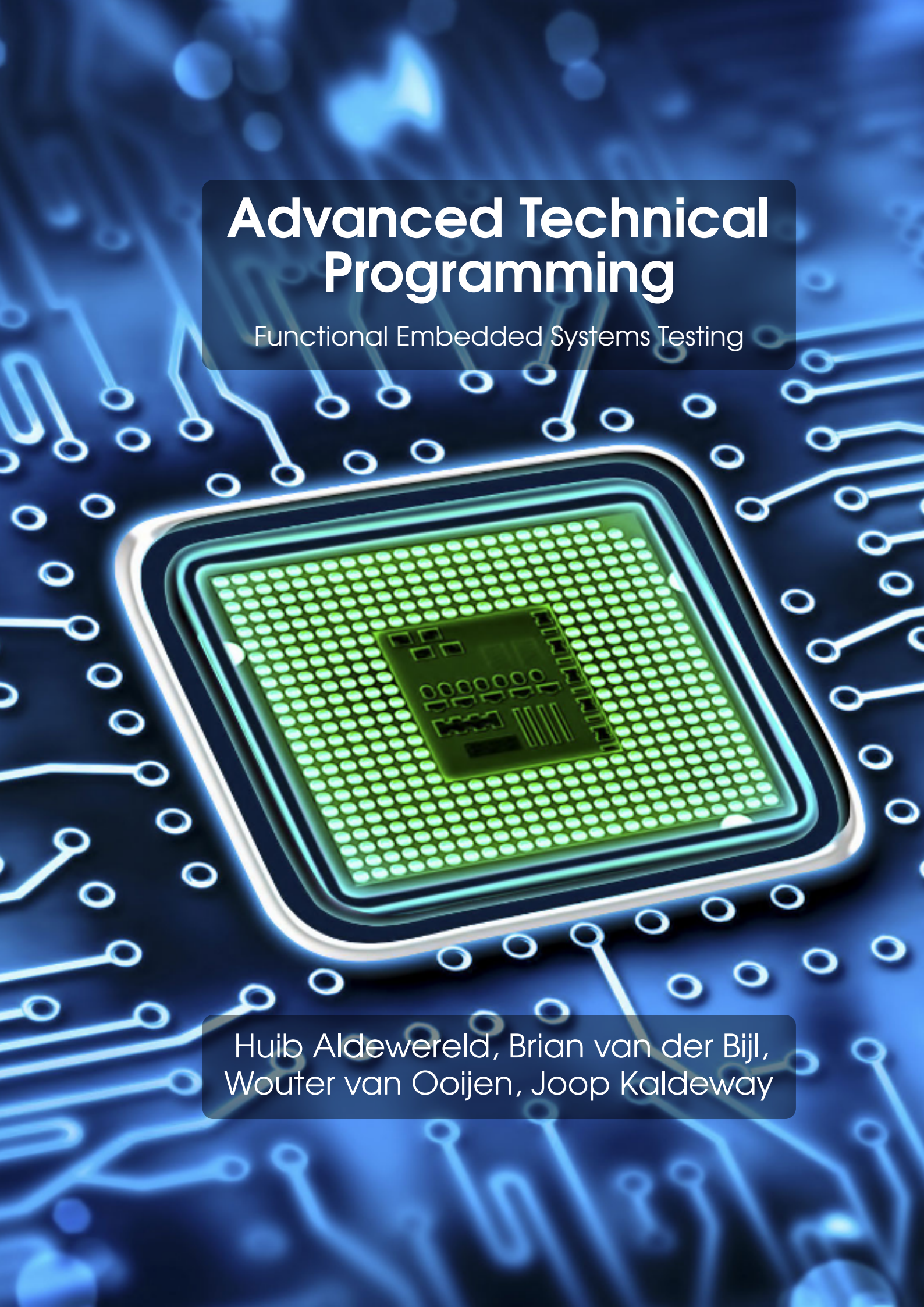
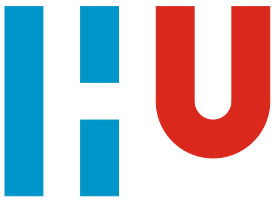
Advanced Technical Programming

Functional Embedded Systems Testing



Huib Aldewereld, Brian van der Bijl,
Wouter van Ooijen, Joop Kaldeway

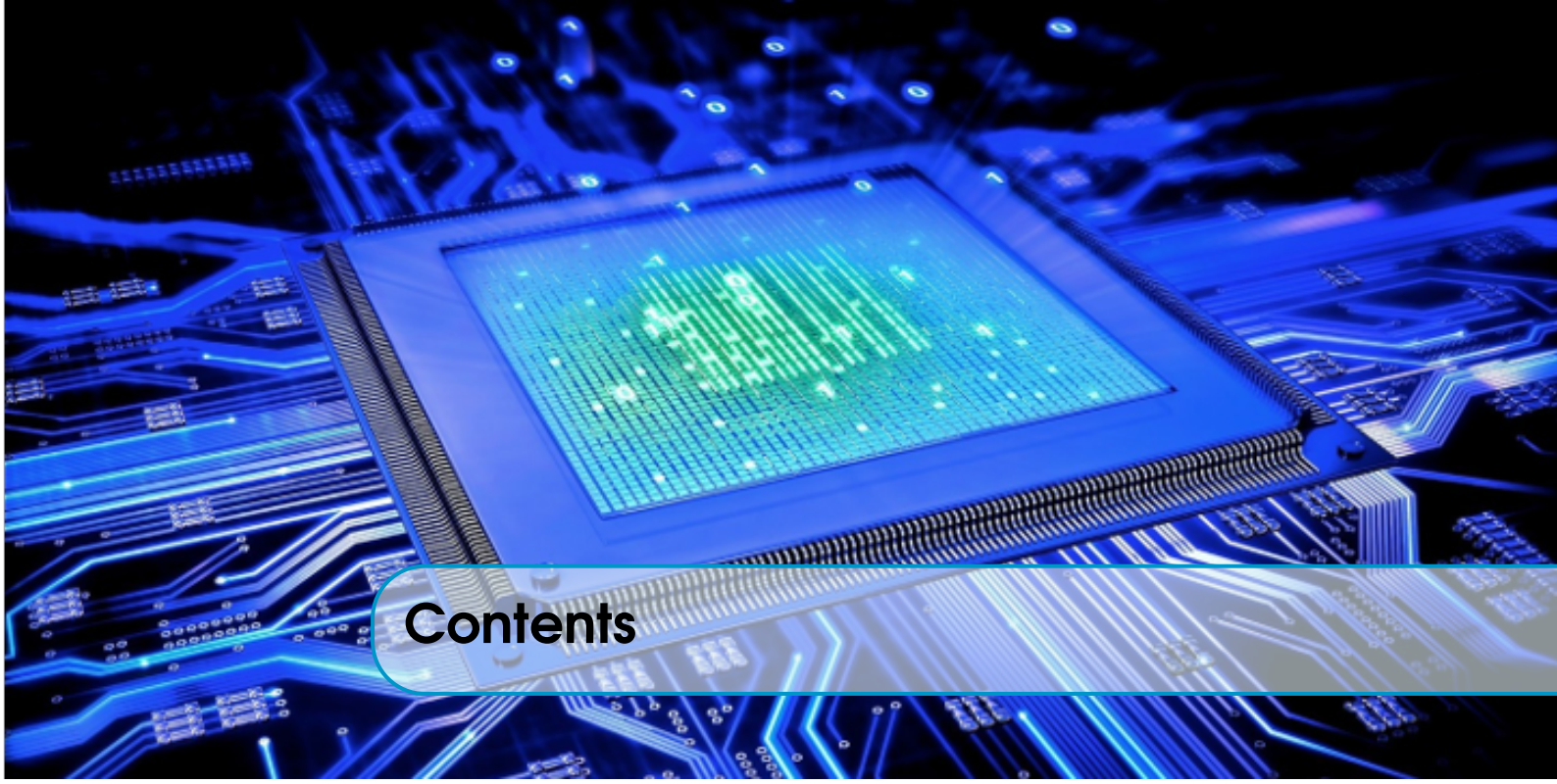


HTTPS://CURSUSSEN.SHAREPOINT.HU.NL/FNT/41/TCTI-VKATP-17/

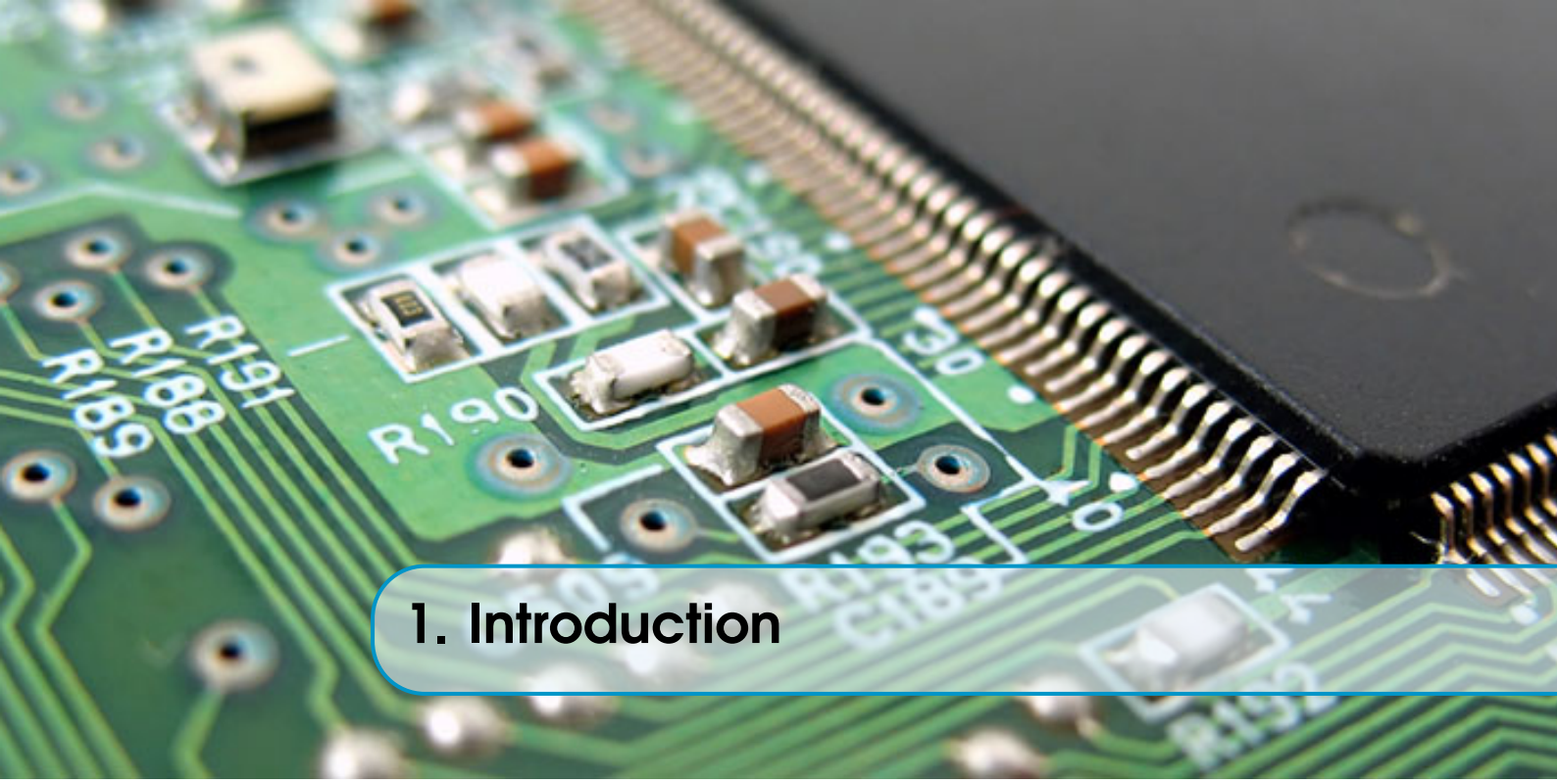
This reader was verified by:

Huib Aldewereld	hogeschooldocent
Brian van der Bijl	trainee
Joop Kaldeway	hogeschooldocent
Leo van Moergestel	hogeschoolhoofddocent
Wouter van Ooijen	hogeschooldocent

First release, August 2017



Contents



1. Introduction

This reader is the accompanying document for the Computer Engineering course 'Advanced Technical Programming' (TCTI-VKATP-17). In advanced technical programming you learn about embedded system testing, functional and aspect-oriented programming and how to substitute parts/components of your embedded system to perform tests. In this introduction we present an overview of this course, and give a motivation why these elements fit together.

1.1 Motivation

The time has gone when one person could develop a complete system alone. Nowadays, systems are developed by large teams, sometimes divided into teams per subsystem, physically separated, or even situated in different continents. The development of systems are large and complex, and that alone puts quality under pressure. The demand for quality is increasing because of of the competitive market or the use of systems in safety critical situations. This explains the need for early and thorough testing. The existence of an independent test team does not mean that testing during the development phase is less important. Both teams have their own important role in the preparation of an end product with the desired quality. An independent test team, in general, executes tests based on the requirements, and their purpose is to provide confidence that the system fulfils those requirements. In contrast, developers start testing at unit level using knowledge of the internal structure of the software. This knowledge is used again to test the integration of the different units with the purpose of delivering a stable system. According to Beizer (1990) requirement-based testing can, in principle, detect all bugs but it would take an infinite time to do so. Unit and integration tests are inherently finite but cannot detect all bugs, even if completely executed. Only a combination of the two approaches together with a risk-based test strategy can detect the important defects. This makes both types of tests essential in

producing systems with the desired quality.

Vocabulary 1.1 — Test types. From *Glossary of terms used in software testing* (BS7925-1).

- **Acceptance testing:** Formal testing conducted to enable a user, customer, or authorized entity to decide whether to accept a system or component.
- **Black-box testing:** Test case selection based on an analysis of the specification of the component without reference to its internal workings.
- **Dynamic testing:** A process of evaluating a system or component based on its behaviour during execution.
- **End-to-end testing:** a technique used to test whether the flow of an application from start to finish is behaving as expected.
- **Integration testing:** Performed to expose faults in the interfaces and in the interaction between integrated components.
- **Regression testing:** Retesting of a previous tested test object following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.
- **Smoke test:** A type of hardware/software testing that comprises of a non-exhaustive set of tests that aim at ensuring that the most important functions work ("if it produces smoke, it is broken").
- **Static testing:** A process of evaluating a system or component without executing the test object.
- **System testing:** A process of testing an integrated system to verify that it meets specified requirements.
- **Unit test**: The testing of individual software components.
- **White-box testing**: Test design techniques that derive test cases from the internal properties of an object, using knowledge of the internal structure of the object.

The main reasons why testing by developers (smoke, unit, integration) is important is listed below:

- Early detected defects are easy to correct. In general, the cost of fixing defects will rise in time (Boehm, 1981).
- High quality basic elements make it easier to establish a high quality system. Low quality basic elements, on the other hand, will lead to an unreliable system and this cannot be solved practically by functional tests.
- Defects detected during post-development stages are difficult to trace back to the source.
- Defects detected during post-development stages have to be corrected and this will lead to time consuming regression tests.
- Good testing during the development stage has a positive influence on the total project time.
- Straight testing of exception handling is only possible at unit level, where exceptions can be triggered individually.

Basically, quality cannot be added to the product at the end by testing – it should be built in right from the start. Checking the quality at every development stage by

testing is a necessity.

Vocabulary 1.2 — Embedded. (see Figure 1.1).

- **Embedded software:** Software in an embedded system, dedicated to controlling the specific hardware of the system.
- **Embedded system:** A system that interacts with the real world using actuators and sensors.
- **Plant**: The environment that interacts with an embedded system.

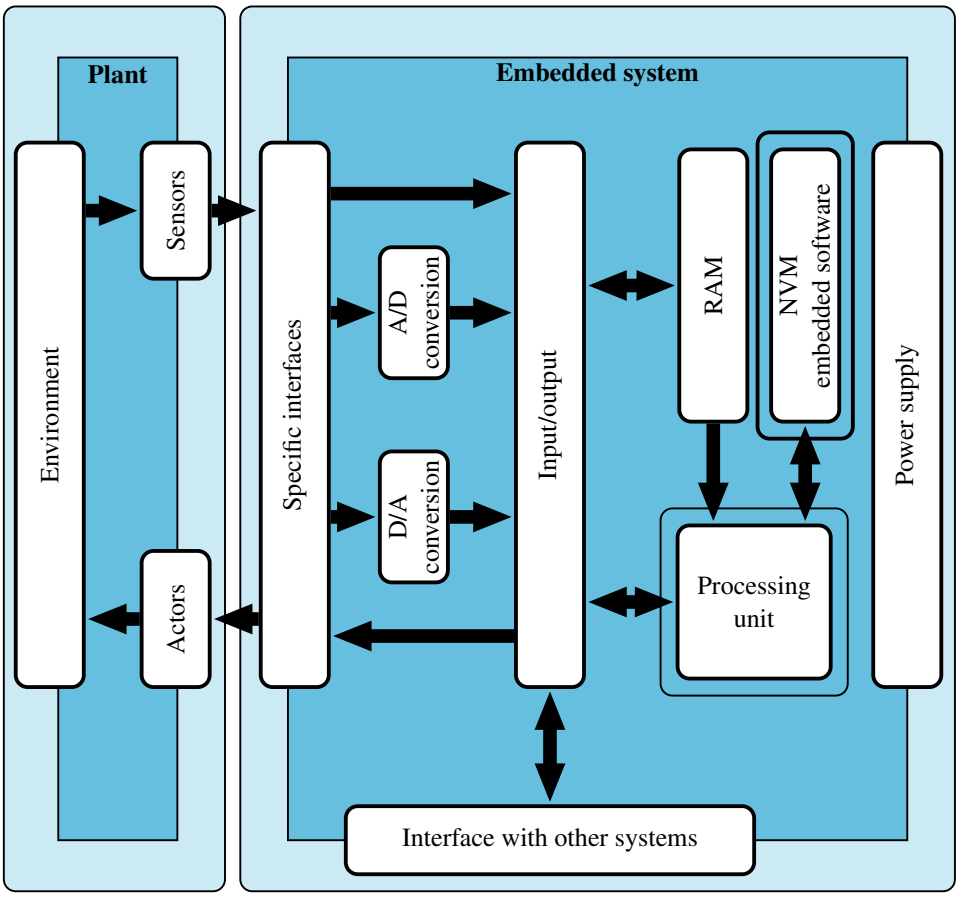


Figure 1.1 Generic scheme of an embedded system.

Testing embedded software and embedded systems is special. Embedded software depends on the hardware used, and vice versa, making it important to test the integration between hardware and software. The hardware, however, typically has limited access for testing, which means that (parts of) the solution could be designed on a different machine. This, however, presents its own unique problems, since the machine or computer that you develop on can have a different architecture (endianness, memory model, hardware accelerators), which might invalidate your test results. Moreover, such machines typically have access to much more memory and CPU cycles than would be available on the target system. This problem can be overcome by the use of emulators or simulators.

In chapter 2 we go deeper into the topic of testing embedded systems. How to

write master test strategies. Which decisions need to be taken about which part(s) to test and how to test them.

1.2 Functional programming

Object-oriented programs tend to grow complex over time. Code-bases grow, and more and more mutable variables are introduced to keep track of the state of objects and the program. The introduction of state and the complexity of keeping track of state makes OOP programs hard to grasp, and therefore hard to test. The way this complexity is managed is by the introduction of unit testing and the art of writing testable code[1]:

- Make it easy to supply dependencies to code under test needs. Avoid things that go out and grab values from global state.
- Avoid side-effects, and side-effects that are needed (database updates, writing files, etc) should be ensured by performing them with a layer of indirection so that during tests one can replace the real object with (for example) a do-nothing mock-up.

Making your code to be free of side-effects and concentrating side-effects to well-defined places, carefully avoiding mixing side-effects and testable/test-worthy code, is making the code more "functional" (or rather, "function-like"). This is exactly the core of what functional programming is about.

Functional programming is a programming paradigm where code is made up from side-effect free functions as a basic building block in the language. Examples of functional programming languages are: Haskell, Erlang, Clean, Miranda, but also procedural languages may allow functional programming, like in Python.

The biggest advantage with functional programming is that the order of the execution of side-effect free functions is not important. This can be used to enable concurrency in a very transparent way. One of the biggest disadvantages, however, was, traditionally, the lack of side-effects. It is difficulty to write useful software without I/O, but I/O is hard to implement without side-effects in functions.

Lots of modern languages have elements from functional programming languages. E.g., Python and C# 3.0 have a lot of functional programming features. One of the reasons why functional programming is increasingly popular is because concurrency is getting a real problem in normal programming. And because we are getting more and more multiprocessor (or multi-core) computers, concurrency is getting increasingly important. Moreover, functional programming languages tend to be more accessible.

In chapter 4 we introduce the core concepts of functional programming, and explain how to functionally program in Python.

1.3 Low-level integration and incomplete systems

As mentioned above, to test embedded systems and embedded software it may be necessary to substitute parts of the system with simulated parts. This process, also

[1] "Testable code" is code that is simple to write unit tests for.

called “scaffolding”, can help you develop and test (software) components in isolation, while still maintaining the integration to other parts of the system.

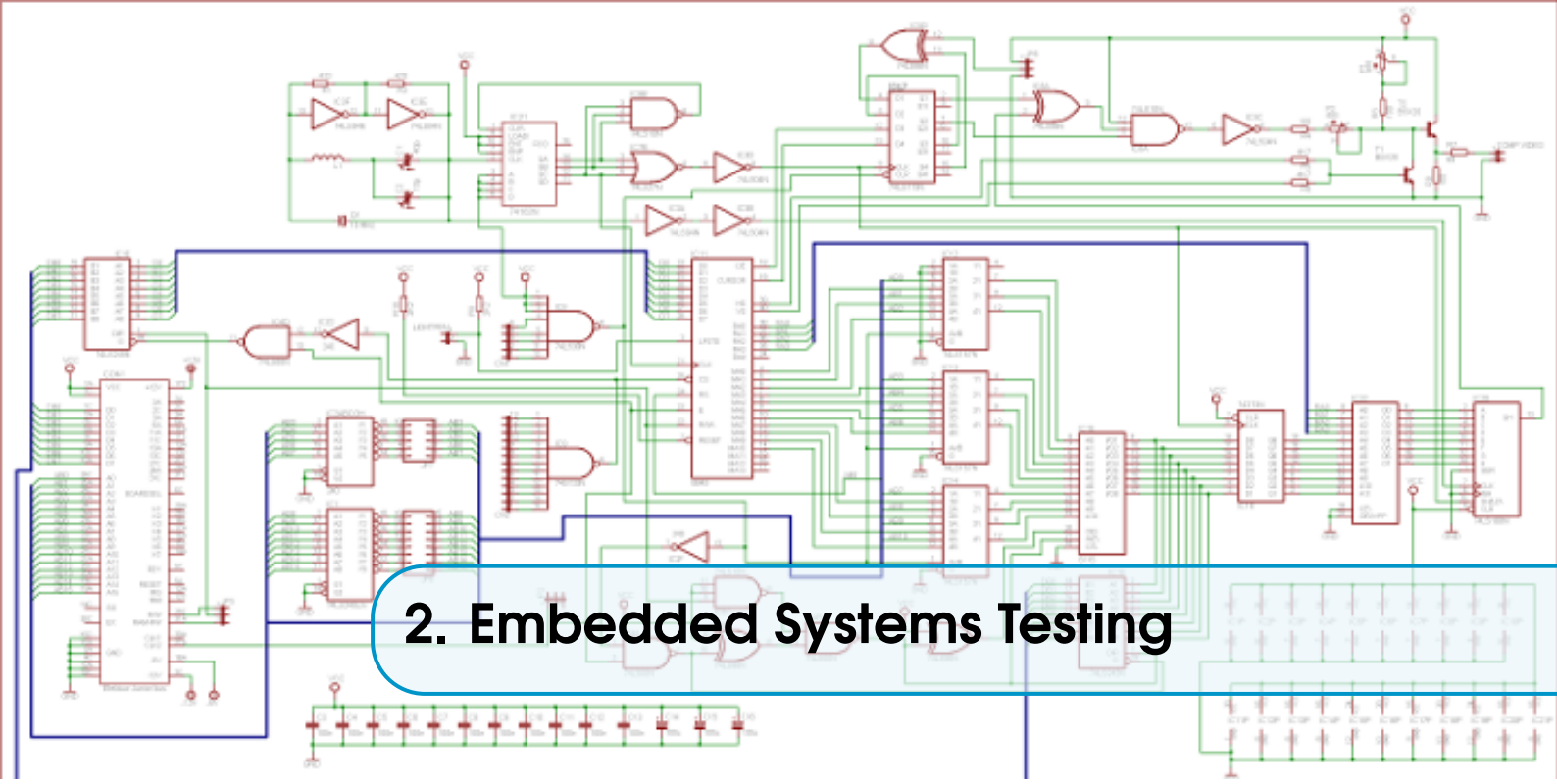
Scaffolding[2] is a technique from model-view-controller frameworks where code generation is used to generate (prototype) implementations of (mainly database) connectors to a specified interface. Instead of having to write all the code for the database connections, scaffolding allows the developer to only specify the interface(s) with database, after which the compiler can complete the needed classes (with the help of templates).

In the context of embedded systems, scaffolding can be better understood as the process of changing the simulator (or simulated parts) of a developed system with the true implementations. Starting with a complete simulation of the system (a system like the one in Figure 1.1), one can slowly replace parts with their target implementations, leaving the other simulated parts in place as scaffolds to develop and test specific pieces of the solution.

In order to apply such techniques, one needs to know more about the development of incomplete systems, the application of simulators, and of drivers and stubs. This is covered in Chapter 5.

Most embedded systems are nowadays still implemented using programming languages such as C or C++, since they present a better control of the hardware (and have a much smaller memory footprint compared to alternatives such as Python or Java). For the development of simulators, stubs and drivers, however, one does not necessarily need to implement those in the target language, but one can opt to use a higher-order language to shorten development time and allow for faster prototyping. In that case, it becomes imperative to be able to link the subsystems (or tests) in the higher-order language to the target implementations in C and C++. In Chapter 6 this link is made.

[2] See `https://en.wikipedia.org/wiki/Scaffold_(programming)`.



2. Embedded Systems Testing

Testing is a process centered around the goal of finding defects in a system. It may be for debugging reasons or acceptance reasons – trying to find defects is an essential part of every test process. Although the whole world agrees that it is much better to prevent defects than to find and correct them, reality is that we are currently unable to produce defect-free systems. Testing is an essential element in system development – it helps to improve the quality of systems

The ultimate goal of testing is to provide the organisation with well-informed advice on how to proceed – advice based on observed defects related to requirements of the system (either explicitly defined or implicitly assumed). The testing itself does not directly improve the quality of the system. But it does indirectly, by providing a clear insight to the observed weaknesses of the system and the associated risks for the organisation. This enables management to make better informed decisions on allocating resources for improving the quality of the system.

To achieve these test goals, every test process contains activities for planning what is needed, specifying what should be tested, and executing those test cases. There is also a universal rule that it is impossible to find all defects and that there are never enough time (or personnel or money) to test everything. Choices have to be made about how to distribute the available resources wisely.

2.1 Challenges for embedded systems

Obviously, the testing of mobile phones will be significantly different from the testing of video set-top boxes or the cruise-control system in cars. They each require specific measures in the test approach to cover specific issues of that system. There is therefore no point in looking for one test approach for embedded systems.

Although there are many reasons why different embedded systems must be tested

This chapter is partly based on (Broekman & Notenboom, 2003).

differently, there are also many similar problems, which have similar solutions, that are involved in any test approach. Some kind of basic test principles must apply to all embedded test projects – but somehow they must be differentiated with several specific measures to tackle the specific problems of testing particular systems.

In the initial stages of the project, choices must be made for specific measures that will be included in the test approach. This "mechanism for assembling the dedicated test approach" is based on the analysis of:

- *Risk*. Measures are chosen to cover the business risks involved in poor quality of the product. This is explained further in section 2.4 below.
- *System characteristics*. Measures are chosen to deal with the problems related to the (technical) characteristics of the product. They are high-level characteristics such as technical-scientific algorithms, mixed signals, safety critical, etc.

It is important to note that we do not aim at achieving a scientifically accurate and complete taxonomy of embedded systems. Rather its purpose is entirely practical and purely from a tester's perspective. It aims at assisting the test manager in answering the question "What makes this system special and what must be included in the test approach to tackle this?".

The following provides a useful initial set of system characteristics:

- safety critical systems;
- technical-scientific algorithms;
- autonomous systems;
- unique system; "one-shot" development;
- analogue input and output (in general, mixed signals);
- hardware restrictions;
- state-based behaviour;
- hard real-time behaviour;
- control systems;
- extreme environmental conditions.

The list is not meant to be exhaustive. Organisations are encouraged to define other system characteristics that are applicable to their products.

Note that no system characteristic is defined for *reactive systems*. A system is described as reactive if it is fast enough to respond to every input event (Erpenbach, Stappert, & Stroop, 1999). It is fully responsible for synchronisation with its environment. This can be sufficiently covered by a combination of the two characteristics state-based behaviour and hard real-time behaviour.

The system characteristics above will be briefly described here.

Safety critical systems

An embedded system is said to be safety critical if a failure can cause serious damage to health (or worse). Administrative systems are seldom safety critical; failure is such systems tends to lead to annoyance and financial damage, but rarely people get hurt. However, many embedded systems have a close physical interaction with people and failure can cause direct physical harm. Examples of such systems are in avionics, medical equipment, and nuclear reactors. With such systems, risk analysis is extremely important and rigorous techniques are applied to analyse and improve reliability.

Technical-scientific algorithms

Often the behaviour of an embedded system seen from the outside is relatively simple and concise. For instance, what cruise-control systems offers to drivers can be described in a few pages. However, realising such a system may be difficult and require a lot of control software to process complex scientific calculations. For such systems, the larger and more complex part of its behaviour is internal and invisible from the outside. This means that the test effort will be focussed on white-box oriented test levels, while relatively less is required for black-box oriented acceptance testing.

Autonomous systems

Some embedded systems are meant to operate autonomously for an indefinite period of time. They are "sent on a mission". After starting up, they require no human intervention. Examples are traffic signalling systems and some weapon systems that, after being activated, perform their tasks automatically. The software in such systems is designed to work continuously and react to certain events without human intervention being needed, or even possible. A direct result of this is that such systems cannot be tested manually. A specific test environment with specific test tools is required to execute the test case and measure and analyse the results.

Unique systems; "one-shot" development

Some systems, such as satellites, are released (or launched) once only and cannot be maintained. They are unique systems (also called "bespoke systems") and meant to be built correctly "in one shot". This is in contrast to mass market products that are released in a competitive market and must be upgraded regularly and have releases to remain attractive. For mass market products, maintenance is an important issue and during development and testing special measures are taken to minimize maintenance costs and time. For instance, reuse of of test-ware and automation of regression testing are probably standard procedures. However, for unique systems the testing has, in principle, no long-term goals because it stops after the first and only release. For this kind of system some rethinking must be done about maintenance, reuse, regression testing, etc.

Analogue input and output (in general, mixed signals)

In the administrative world, the result of a transaction can be predicted exactly and measured. An amount on an invoice, a name, or an address is always defined exactly. This is not the case in embedded systems that deal with analogue signals. The input and output do not have exact values but have a certain tolerance that defines acceptable boundaries. Also, the expected output is not always defined exactly. An output value that lies just outside a defined boundary is not always definitely invalid. A grey area exists where sometimes, intuitively, it is decided if the test output is acceptable or not. In such situations terms such as "hard boundaries" and "soft boundaries" are used.

Hardware restrictions

The limitations of hardware resources can put specific restrictions on embedded software, for instance on memory usage and power consumption. It also happens that specific hardware dependent timing aspects are solved in the software. These kinds

of restrictions on the software have little to do with the required functionality but are essential if a system is to function at all. They require a significant amount of testing effort of a specialised and technical nature.

State-based behaviour

The behaviour of a state machine can be described in terms of transitions from a certain state to another state, triggered by certain events. The response of such a system to a certain input depends on the history of previous events (which resulted in a particular state). A system is said to exhibit state-based behaviour when identical inputs are not always accepted and, if accepted, may produce different outputs (Binder, 2000).

Hard real-time behaviour

The essence of real time is that the exact moment that input or output occurs, influences the system behaviour. A salary system for instance is not real time – it does not make any difference if the system calculates your salary now or after 15 minutes. (Of course if the system waits a whole year, it does matter, hopefully, but that does not make it real time.) In order to test this real-time behaviour, the test cases must contain detailed information about the timing of inputs and outputs. Also, the result of test cases will usually be dependent on the sequence in which they are executed. This has a significant impact on both test design and test execution.

Control systems

A control system interacts with its environment through a continuous feedback mechanism – the system output influences the behaviour of the control system. Therefore the behaviour of the system cannot be described independently, it is dependent on the behaviour of the environment. Testing of such systems usually requires an accurate simulation of the environment behaviour. Examples of such systems are industrial process control systems and aircraft flight control systems.

Extreme environmental conditions

Some systems are required to continue operating under extreme conditions, such as exposure to extreme heat or cold, mechanical shock, chemicals, or radiation. Testing this usually requires specialised equipment to generate the extreme conditions. Similarly, if testing in the real environment is too dangerous, simulation equipment is used instead.o

2.2 Testing lifecycle

A lifecycle structures the process of development and testing embedded systems by dividing it into phases, describing which activities need to be performed, and in what order.

Section 2.2.1 introduces the multiple V-model to describe the development lifecycle of an embedded system and to indicate the place where different types of testing takes place. The tests that take place during the earlier stages in the development lifecycle are low-level tests, such as unit tests and integration tests. They are often performed by developers, or in close collaboration with developers. These test are

usually organised as part of the development plan and do not have a separate test plan or budget. The tests that occur towards the end of the development lifecycle are the high-level tests, such as system tests and acceptance tests. They are often performed by an independent test team which is responsible for a separate test plan and budget.

2.2.1 The multiple V-model

In embedded systems, the test object is not just executable code. The system is usually developed in a sequence of product-appearances that become more real. First a *model* of the system is built on a PC, which simulates the required system behaviour. When the *model* is found to be correct, code is generated from the model and embedded in a *prototype*. The experimental hardware of the prototype is gradually replaced by the real hardware, until the system is built in its final form as it will be used and mass produced. The reason for building those intermediate product appearances is, of course, that it is cheaper and quicker to change a prototype than to change the final product, and even cheaper and quicker to change the model.

Straightforward multiple V-model

The multiple V-model, based on the well-known V-model (Spillner, 2000) is a development model that takes this phenomenon into account. In principle each of the product appearances (model, prototype, and final product) follows a complete V-development cycle, including design, build and test activities. Hence the term "multiple V-model" (see Figure 2.1). The essence of the multiple V-model is that different physical versions

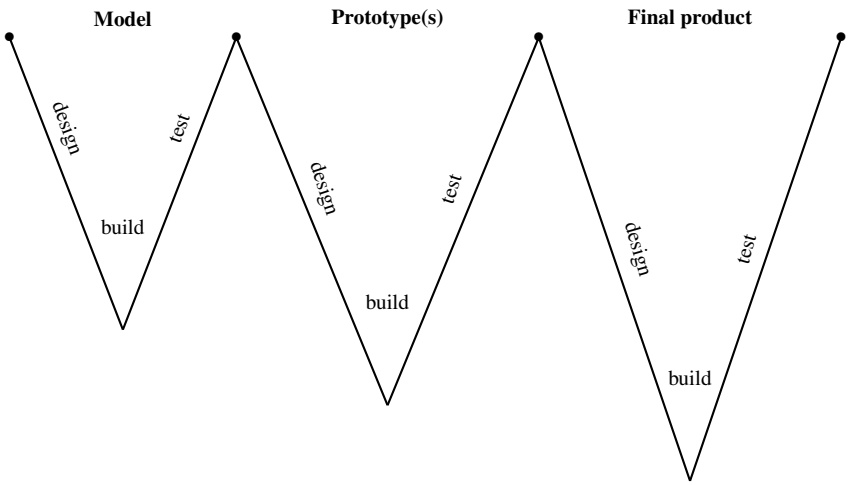


Figure 2.1 Multiple V development lifecycle.

of the same system are developed, each possessing the same required functionality in principle. This means, for instance, that the complete functionality can be tested for the model as well as for the prototype and the final product. On the other hand, certain detailed technical properties cannot be tested very well on the model and must be tested on the prototype – for instance, the impact of environmental conditions can best be tested on the final product. Testing the different physical versions often requires specific techniques and a specific test environment. Therefore a clear relation exists

between the multiple V-model and the various test environments (see section 5.1).

The multiple V-model shows three consecutive V-shaped development cycles (model, prototypes, and final product). It is important to understand that this is a simplified representation of the development process for embedded systems. It should not be regarded as a straightforward sequential process ("waterfall model") where the prototype is first fully designed, then built, tested and made ready as a final product. The middle V especially, where the prototypes are developed is iterative in nature.

The test process involves a huge number of test activities. There are many test design techniques that can and will be applied, test levels and test types that must be executed, and test related issues that require attention. The multiple V-model assists in structuring these activities and issues. By mapping them onto the multiple Vs, it provides insight to the questions: "When can activities best be executed?" and "Which test issues are most relevant at which stage in the development process?".

Nested multiple V-model

The multiple V-model with the three sequential V-cycles does not take into account the practice of (functional) decomposition of a complex system. The development of such a system starts with the specification of the requirements at a high-level, followed by an architectural design phase where it is determined which components (hardware and software) are required to realise this. Those components are then developed separately and finally integrated into a full system. In fact, the simple V-model can be applied to this development process at a high-level. The left-side of the V-model handles the decomposition of the system into its components. The middle part of the V-model handles consists of parallel development cycles for all components. The right side of the V-model handles the integration of the components. This is illustrated in Figure 2.2.

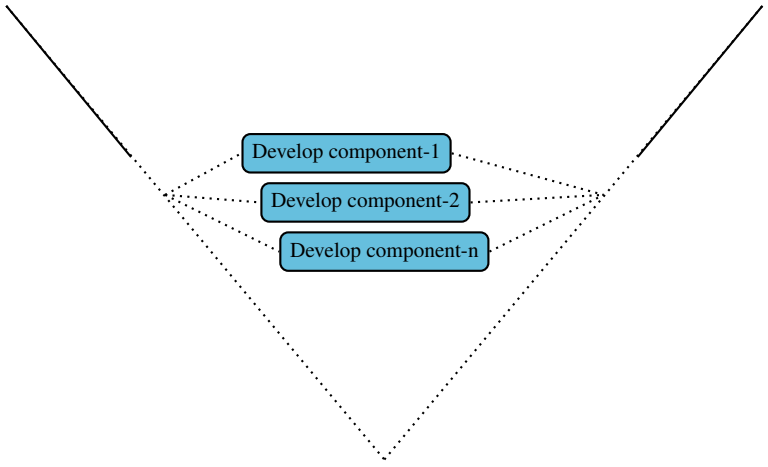


Figure 2.2 Parallel development phases in a V-model.

This principle can be repeated for components that are too big or too complex to develop as one entity. For such a component, an architectural design activity is carried out to determine which subcomponents are required. Because this may result in a V-model within a V-model (within a V-model, ...) the development lifecycle model is

said to be “nested”.

In fact the purely sequential multiple V-model is mainly applicable to the component level. Usually it is not the complete system which is fully modelled first, then completely prototyped, and so on. It is the components that are built in this stepwise way. This explains why some development activities and issues cannot be mapped very well on the 3 Vs in the multiple V-model – for instance high-level and low-level requirements, safety plan, design and build specific tools. That is because they belong to the overall development process.

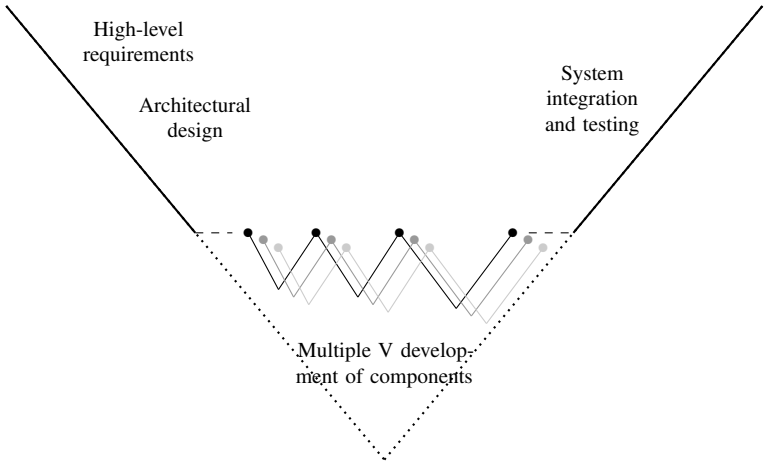


Figure 2.3 The nested multiple V-model.

When the V-model at the system level is combined with the multiple V-model at the component level, it is the so-called “nested multiple V-model” (Figure 2.3).

2.3 Testing techniques

The testing of a large system, with many hardware components and hundreds of thousands lines of code in software, is a complex endeavour involving many specialists performing different testing tasks at various points in the project. Some test the states and transitions of certain components exhaustively, others test transitions on a higher integrated level, and others evaluate user-friendliness features. Some teams are dedicated to testing performance, others are specialised in simulating the real world environments.

In this section we first explain the difference between test types and test levels. It is the difference between which *aspects* are being tested and which *organisational entity* is executing the test.

2.3.1 Test types

A system can be tested from different points of view – functionality, performance, user-friendliness, etc. These are the quality attributes that describe the various aspects of system behaviour. Standards exist, for instance ISO 9126, to define a set of quality attributes that can be used for this purpose. In practice, several quality attributes are often combined in a single test. This gives rise to the concept of *test type*.

> A test type is a group of activities with the aim of evaluating a system on a set of related quality attributes.

Test types state *what* is going to be tested (and what is not). For instance, a tester doing a functional test is not (yet) interested in the performance displayed, and vice versa. Table 2.1 describes some common test types – it is not intended to be complete.

Table 2.1 Test types.

Test type	Description	Quality characteristics included
Functionality	Testing functionality behaviour (includes dealing with input errors)	Functionality
Interfaces	Testing interaction with other systems	Connectivity
Load and stress	Allowing large quantities and numbers to be processed	Continuity, performance
Support (manual)	Providing the expected support in the system's intended environment (such as matching with the user manual procedures)	Suitability
Production	Test production procedures	Operability, continuity
Recovery	Testing recovery and restart facilities	Recoverability
Regression	Testing whether all components function correctly after the system has been changed	All
Security	Testing security	Security
Standards	Testing compliance to standards	Security, user-friendliness
Resources	Measuring the required amount of resources (memory, data communication, power, ...)	Efficiency

2.3.2 Test levels

The test activities are performed by various testers and test teams at various times in the project, in various environments. Those organisational aspects are the reason for introducing *test levels*.

> A test level is a group of activities that is organised and managed as an entity.

Test levels state *who* is going to perform the testing and *when*. The different test levels are related to the development lifecycle of the system. It structures the testing process by applying the principle of *incremental testing* – early in the development process, parts of the system are tested in isolation to check that they conform to their technical specifications. When various components are of satisfactory quality, they

are integrated into larger components or subsystems. These in turn are then tested to check if they conform to higher-level requirements.

Often a distinction is made between low-level and high-level tests. Low-level tests are test on isolated components, executed early in the lifecycle (left side of the multiple V-model) in a development-like environment. High-level tests are tests on the integrated systems or subsystems, executed later in the lifecycle (right side of the multiple V-model) in a (simulated) real-life environment. Section 5.1 describes in more detail the different test environments for the different test levels. Usually low-level tests are more white-box oriented and high-level tests more black-box oriented.

Table 2.2 lists some common test levels for embedded systems – it is not intended to be complete.

Table 2.2 Test levels.

Test level	Level	Environment	Purpose
Hardware unit test	Low	Laboratory	Testing the behaviour of hardware component in isolation
Hardware integration test	Low	Laboratory	Testing hardware connections and protocols
Model in the loop	High/low	Simulation models	Proof of concept; testing control laws; design optimisation
Software unit test, host/target test	Low	Laboratory, host + target processor	Testing the behaviour of software components in isolation
Software integration test	Low	Laboratory, host + target processor	Testing interactions between software components
Hardware/software integration test	High	Laboratory, target processor	Testing interaction between hardware and software components
System test	High	Simulated real life	Testing the system works as specified
Acceptance test	High	Simulated real life	Testing that the system fulfils its purpose for the user/customer
Field test	High	Real life	Testing that the system keeps working under real-life conditions.

2.3.3 Master test plan

If every test level would define for itself the best things to do, there would be a significant risk that some tests would be performed twice and others would be omitted. In practice, it is not unusual for the system test and acceptance test to have the same test design techniques applied to the same system specifications – this is a waste of

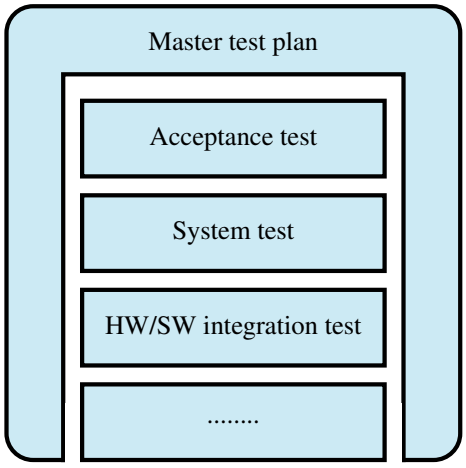


Figure 2.4 Master test plan, providing overall co-ordination of the test levels.

valuable resources. Another risk is that the total test process spends longer on the critical path than is necessary because the planning of the different test levels has not been coordinated.

Of course it is better to co-ordinate the various test levels: it prevents redundancies and omissions; it ensures that a coherent overall test strategy will be implemented. Decisions must be made as to which test level is most suited to testing which system requirements and quality attributes. Scarce resources, such as specialised test equipment and expertise, can be allocated to the various test levels for explicitly agreed periods. To achieve this co-ordination and manage the overall test process, an overall test plan has to be drawn up which defines the tasks, responsibilities, and boundaries for each test level. Such an overall test plan is called a *master test plan* (see Figure 2.4). Often a project manager delegates this task to a designated test manager who is responsible for all test activities.

A master test plan can be viewed as the combination of *what* has to be tested (test types) and *who* is going to perform the test activities (test levels). After completing the master test plan, for each test level the staff involved will know exactly what is expected of them and what level of support and resources they can expect. The master test plan serves as the basis for the detailed test plan of each test level. It does not need to prescribe for each level which activities must be performed in detail – that is for the detailed test plans. The master test plan deals with decisions that concern areas where the various test levels can either help or hinder each other. Three areas of main interest for the master test plan are:

- test strategic choices – what to test and how thorough;
- allocation of scarce resources;
- communication between the disciplines involved.

These areas are dealt with in the next subsection under master test planning activities – respectively, “determine master test strategy”, “specifying infrastructure”, and “define organisation”.

2.3.4 Activities

The test manager starts as early in the development process as possible with the preparations for the overall test process. This is done by developing a master test plan. It starts with the formulation of the overall objectives and responsibilities for testing and defining the scope of the master test plan. Then a global analysis of the system to be tested and the development process is carried out. The next step is to use this information and start discussions in the organisation concerning which measures to take regarding testing and quality assurance to ensure a successful release of the system. This is the determination of the master test strategy and involves choosing what to do, what not to do, and ow to realise it. The test manager's main responsibility is to deliver this master test strategy. To accomplish this, the required resources (infrastructure and personnel) must be defined for the test process. To manage this, communication between all involved disciplines and the required reporting must be defined – in other words, the required organisation.

A good master test plan is not created in a quiet and isolated office. It is a highly political task that requires discussion along with bargaining and persuasion throughout the organisation. The results of this are recorded in a document that is delivered to all stakeholders. To create a master test plan, the following activities must be accomplished:

1. formulating the assignment;
2. global review and study;
3. determine the master test strategy;
4. specify infrastructure;
5. define the organisation;
6. determine a global schedule.

Formulate the assignment

The objective of this activity is to ensure that the rest of the organisation has the correct expectations about what the test organisation will do for them. Often a test organisation is faced with the distressing situation that they are expected to perform miracles but do not get the means to do it. When an assignment is clearly formulated and well communicated, such situations become rare and can be dealt with better.

Some topics need to be discussed concerning formulating an assignment.

Commissioner This is the originator of the assignment to draw up a master test plan – or the person (or organisational entity) who decrees that testing should be performed. This person can be thought of as the *customer* of the test team. Usually the commissioner is the general (project) manager for the system development process who delegates his test responsibilities to the test manager. Often, the user organisation and product management organisation have a customer role. It is important for the test manager to ensure that those concerned understand that being a customer not only means that you may make demands, but also that you have to provide the means (or at least pay for it).

Contractor This is the person responsible for drawing up the master test plan – usually the test manager.

Test levels These are the test levels involved in the master test plan. To be considered

here are inspections, unit and integration tests (of both hardware and software), system test, functional acceptance tests, and the production acceptance tests. When certain tests levels are excluded from the master test plan (e.g., not controlled by the test manager), special attention should be paid to co-operation and communication with those activities. Also, the development of specific test tools or other components of the infrastructure should be considered.

Scope This is defined by the restrictions and limitations of the entire test process. For instance:

- unique identification of the information system to be used;
- interfaces with neighbouring systems;
- conversion or porting activities.

Note that it is just as important to state what does *not* belong to the scope of the test assignment.

Objective The objective of the test process can be described in terms of what will be delivered. For instance:

- services
  - providing management with advice concerning measured quality and the associated risks;
  - maintaining an environment where high priority problems and solutions can be tested immediately;
  - third line helpdesk, for extraordinary problems;
- products to be delivered by each test level.

Preconditions These describe the conditions imposed on the test process externally, for example:

- *Fixed final date* – the date by which the system must be tested is usually already fixed by the time the test assignment is commissioned.
- *Planning* – the planning for delivery of the test basis, the test object and the infrastructure is usually fixed when the test assignment is commissioned.
- *Available resources* – the customer often sets limits on the personnel, means, budget, and time.

Assumptions These describe the conditions imposed by the test process on third parties. They describe what is required to make it all possible, for example:

- *required support* – the test process requires various types of support concerning, for instance, the test basis, the test object, and/or the infrastructure;
- *changes to the test basis* – the test process must be informed about coming changes: in most cases this simply means participating in existing project meetings within the system development process.

During the ensuing planning process, the preconditions and assumptions are elaborated in more detail.

Global review and study

This activity is aimed at gaining insight into the (project) organisation, the objective of the system development process, the information system to be developed, and the requirements the system should meet. It involves two activities:

- study of the available documentation;

- interviews.

Study of the available documentation The documentation, made available by the customer, is studied. Consider here:

- system documentation, such as the results of information analysis or a definition study;
- project documentation, such as the planning for the system development process;
- the organisation scheme and responsibilities, the quality plan, and (global) size estimates;
- a description of the system development method, including standards;
- a description of host and target platforms;
- contracts with suppliers.

If it is the development of a new release of an existing system, the presence and re-usability of existing test-ware is investigated.

Interviews Various people involved in the system development process are interviewed. Consider here:

- representatives of product marketing to gain insight into the company objectives and "selling arguments" of the product;
- user representatives to gain insight into the most appreciated functionality and the most irritating flaws of the system;
- field representatives to gain insight into the production environment of the system at intended customer sites;
- the suppliers of the test basis, test object, and the infrastructure in order to ensure a match between the different disciplines involved.

It is also advisable to consult those who are indirectly involved, such as accountants or lawyers, the manufacturing manager, the future maintenance organisation, etc.

Determine master test strategy

The objective of this activity is to reach a consensus with all stakeholders about what kind of testing is (and is not) going to be carried out, related to the quality the company wishes to be asserted. This has much to do with trading off and making choices. One hundred percent testing is impossible – or at least economically infeasible – so the company must decide how much risk they are willing to take that the system does not reach the desired quality. The test manager has the important task of explaining to all stakeholders what the impact would be of choosing not to do certain test activities, in terms of added risk to the company.

During this activity it is determined which quality characteristics of the system are more important than others. Then it is decided which (test) measures are best suited to cover those quality characteristics. For the more important quality characteristics, more thorough test design techniques can be applied – possibly more sophisticated tools may be desirable. It is also determined at which test level those measures should be carried out. In this way, the master test strategy sets goals for each test level. It prescribes globally for each level what kind of testing should be performed and how much time and resources are allocated. The determination of the test strategy involves three activities:

- review current quality management measures;

- determine strategy;
- global estimation of the test levels.

Review current quality management measures Testing is a part of the total quality management within the system development process. A quality system may be present which, for instance, provides for executing inspections and audits, developing a certain architecture, adhering to certain standards, and following change control procedures. When deciding which test activities are required, the other – not test related – quality management activities, and what they aim at, should also be considered.

Determine strategy The steps to obtain a test strategy are described here briefly (a detailed description is given in section 2.4).

- *Determine quality characteristics.* Based on risks, a set of relevant quality characteristics for the system is chosen. These characteristics are the main justification for the various test activities and they must be considered in the regular reporting during the test process.
- *Determine the relative importance of the quality characteristics.* Based on the results of the previous step, the relative importance of the various quality characteristics is determined. This is not a straightforward exercise because relative importance is subjective. Different people will have different notions of importance of certain characteristics of the particular system. Choosing where to spend scarce resources for testing can cause a clash of interests. For the test process it is important that the stakeholders have experienced this dilemma and that each has the understanding that what is eventually decided may not be perfect but, considering everything, it is the best choice.
- *Allocate quality characteristics to test levels.* In order to optimise the allocation of scarce resources, it is indicated which test level(s) must cover the selected quality characteristics and, roughly, in what way. This results in a concise overview of the kinds of test activities that are going to be performed in which test levels in order to cover which quality characteristics. It can be presented as a matrix – see Table 2.4 for an example.

Global estimation of the test levels For each test level, it can easily be derived from the strategy matrix which test activities must be performed. Next, a global estimation of the test effort is drawn up. During the planning and control phase of the various test levels, a more detailed estimation is performed.

Specify infrastructure

The aim of this activity is to specify at an early stage the infrastructure needed for the test process, especially the parts required for several test levels or those which have a relatively long delivery time. This involves three activities:

- specify required test environments;
- specify required test tools;
- determine the planning of the infrastructure.

Specify required test environments A test environment consists of the facilities needed to execute the test, and is dependent on the system development environment and the future production environment (see Section 5.1). In general, the more it looks like the real production environment, the more expensive it is. Sometimes the testing

of specific properties of the system require specialised and expensive equipment. The master test plan must define, in general terms, the different test environments and explain which test levels will be allotted which environment.

It is important to state the specific demands for the test process. Consider here, for instance, that random external triggers must be simulated or that a solution must be available to generate specific input signals.

In practice the environment is often a fixed situation and you just have to make do with this. In this case the master test plan must clearly explain the risks involved in the possible shortcomings of the test environment that has to be used.

Specify required test tools Test tools may offer support to test activities concerning planning and control, constructing initial data and input sets, test execution, and output analysis. Many tools required by testers are also required by developers, for instance tools that enable manipulation of input signals or analyse specific system behaviour. Sharing of such tools and the knowledge of how to operate them should be discussed and agreed.

When a required tool is not yet available it must be specifically developed, and this should be treated as a separate development project. The master test plan must clearly indicate the dependencies between this development project and the test activities in the master test plan.

Determine the planning of the infrastructure Responsibility for the further elaboration, selection, and acquisition or development of all the necessary parts of the infrastructure will be determined, time lines will be defined, and the agreements will be recorded. In addition, the availability of the various facilities will be outlined.

Define organisation

This activity defines the roles, competencies, tasks, and responsibilities at the level of the entire test process. Setting up the organisation involves three activities:

- determine the required roles;
- establish training;
- assign tasks, competencies, and responsibilities.

Determine the required roles The roles needed to achieve a better match between the various levels must be determined. They deal not with the individual test level, but with the overall test process. It is about co-ordination of different schedules, optimal use of scarce resources, and consistent collection and reporting of information. Consider especially:

- general test management and co-ordination;
- centralised control, for instance of infrastructure or defect administration;
- centralised quality assurance.

It is common that disciplines from the line organisation are employed for these purposes. For instance:

- test policy management;
- test configuration management;
- methodological and technical support;
- planning and monitoring.

The determination of the test roles within the various test levels is made during the

planning and control phase of the test levels concerned.

Establish training If those who will participate in the test levels are not familiar with basic testing principles or proficient with the specific techniques and tools that will be used, they should become trained. The required training can sometimes be obtained from commercial courses, or it can be developed within the organisation. The master test plan must reserve sufficient time for this training.

Assign tasks, competencies and responsibilities Specific tasks, competencies, and responsibilities are assigned to the defined test roles. This applies especially to the tasks related to the tuning between various test levels and the decisions to be taken. For instance:

- drawing up regulations for the products to be delivered by various test levels;
- monitoring that the regulations are applied (internal review);
- coordinating the test activities common to the various test levels, such as the set-up and control of the technical infrastructure;
- drawing up guidelines for communication and reporting between the test levels, and between the various disciplines involved in the overall test process;
- setting up the methodological, technical, and functional support;
- preserving consistency in the various test levels.

Determine global schedule

The aim of this activity is the design of a global schedule for the entire test process. It includes all test levels (within the scope of the master test plan) and the special activities such as development of infrastructure components and training.

For each test level, starting and finishing dates, and the deliverables, are stated. In the planning and control phase of the various test levels, the planning is elaborated in detail.

The global schedule should contain at least:

- description of the high-level activities to be carried out (phases per test level);
- deliverables of those activities;
- allocated time (man-hours) for each test level;
- required and available lead time;
- relations with, and dependencies on, other activities (within or outside the test process, and between the various test levels). The interdependencies between the various test levels are especially important. After all, the execution phases of several test levels are mostly executed sequentially – first unit test, then integration test and system test, and finally acceptance test. Often this is exactly the critical path of the overall test process.

2.4 Risk-based testing

Developing a risk-based test strategy is a means of communicating to the stakeholders what is most important about this system for the company, and what this means for the testing of this system. “Just test everything” is either theoretically impossible, or at least economically infeasible – it would be a waste of resources (time, money, people, and infrastructure). In order to make the best possible use of resources, it is decided on which parts and aspects of the system the test emphasis should fall. The risk-based

test strategy is an important element in a structured test approach and contributes to a more manageable test process.

Priorities must be set and decisions made about what is important and what is not. Now this sounds very easy and straightforward, but it has a few problematic questions:

- what do we mean by "important"?
- what are the things that we evaluate to be important or not?
- who makes these decisions?

What do we mean by important? Is that not a very subjective concept? With a risk-based test strategy, evaluating importance is about answering the question: "How high is the (business) risk if something is NOT OK in this area?" A risk is defined as the chance of a failure occurring related to the damage expected when it does occur. If a system is of insufficient quality, this may imply high damage to the organisation. For instance, it could cause loss of market share, or the company may be obliged to call back millions of sold products to replace the faulty software. Therefore this situation forms a risk for the organisation. Testing helps cover such risks by providing insight into the extend to which a system meets the quality demands. If certain areas or aspects of the system imply high risks for the organisation, then more thorough testing is obviously a solution. Of course, the reverse also holds – no risk, no test.

What are the things we evaluate as important or not? In the test strategy, the relative importance of two kinds of things are evaluated – the subsystems and the quality attributes. Basically, statements such as "the signal decoding subsystem is more important than the scheduling subsystem" and "usability is more important than performance" are made. A subsystem can be any part of a system, which is conveniently treated separately for the purpose of strategy decisions. Usually the architectural decomposition of the system is used for this purpose. Quality attributes describe types of required system behaviour, such as functionality, reliability, usability, etc. Standards for the definition of quality attributes exist, such as ISO 25010.

Who makes these decisions? The task of establishing a risk-based strategy is much like a diplomatic mission. Many different personnel, or departments with different interests, are involved – sales, maintenance, personnel, developers, end users, etc. They all have their own ideas about what is important. If they are not listened to, they can make things difficult for the test project, claiming "I don't agree and I am going to fight it". Getting people involved in the decision-making process, creates higher commitment and decreases the risk of political trouble later in the project. This does not mean that everything has to be tested after all. A better definition of the best test strategy is "that which satisfies no one completely, but on which everyone agrees that, all things considered, is the best compromise".

Now it is true that establishing a test strategy involves much negotiating and making acceptable choices, but some things are just not negotiable. The company may impose compliance with certain *standards* on the development and testing process. Some industries are required to comply to certain industry standards for certification of their product. For instance the DO-178B standard (RTCA/DO-178B, 1992) is required for the safety critical software in the avionics industry. Many standards exist that can help an organisation to fill in parts of their test strategy (Reid, 2001). For instance the software verification and validation standard IEEE 1012 relates integrity levels of

software to required test activities. The process for determining such integrity levels is defined by the ISO 15026 standard. Other useful standards are IEEE 829 which defines a broad range of test documentation, BS7925-1 which is a software testing vocabulary, and BS7925-2 which defines and explains several test design techniques.

2.4.1 Risk assessment

Developing a test strategy requires insight into risks. What will the consequences be when the authentication module does not work correctly? What will the damage be when the system shows insufficient performance? Unfortunately, risk assessment in our industry is not a science. Risk cannot be calculated and expressed in absolute numbers. Nevertheless, this section tries to give the required insights into risks to enable the development of a sound test strategy.

For the purpose of test strategy, risks are assessed on the basis of quality characteristics and subsystems. In order to do that, separate aspects of risk are analysed which are derived from the well-known equations (Reynolds, 1996):

$$\text{Risk} = \text{chance of failure} \times \text{damage}$$

where the chance of failure is related to aspects including frequency of use and the chance of a fault being present in the system. With a component that is used many times a day by many people, the chance of a fault showing itself is high. In assessing the chance of faults occurring, the following list (based partly on (Schaefer, 1996)) may be helpful. It shows the locations where faults tend to cluster:

- complex components;
- completely new components;
- frequently changed components;
- components for which certain tools or techniques where employed for the first time;
- components which were transferred from one developer to another during development;
- components that here constructed under extreme time pressure;
- components which had to be optimised more frequently than usual;
- components in which many defects were found earlier (e.g., in previous releases or during earlier reviews);
- components with many interfaces.

The chance of failure is also greater for:

- inexperienced developers;
- insufficient involvement of user-representatives;
- insufficient quality assurance during development;
- insufficient quality of low-level tests;
- new development tools and development environment;
- large development teams;
- development teams with suboptimal communication (e.g., owing to geographical spread or personal causes);
- components developed under political pressure, with unresolved conflicts in the organisation.

Next to the possible damage must be estimated. When the system fails (does not meet the quality requirements), what will be the damage to the organisation? This damage can take the form of cost of repair, loss of market share caused by negative press, legal claims, foregone income, etc. An attempt should be made to translate each form of damage into money terms. This makes it easier to express the damage in numerical terms and to compare the related risks with other assessed risks.

The required information for risk assessment is usually obtained from different sources. Some people know how the product is used in its intended environment, about frequency of use and possible damage. These include end users, support engineers, and product managers. Project team members such as architects, programmers, testers, and quality assurance staff, know best what the difficulties are in developing the product. They can provide information useful in assessing the chance of a fault.

Because of the complexity of the matter, it is impossible to assess risks with complete objectivity and accuracy. It is a global assessment resulting in a relative ranking of the perceived risks. It is therefore important for the risk assessment to be carried out not only by the test manager, but also by as many of the other involved disciplines and stakeholders in the project. This not only increases the quality of the test strategy, but has the added advantage that everyone involved has a better awareness of risks and what testing can contribute towards handling them. This is extremely important in "setting the right expectations with regard to testing." It must be understood that testing is just one of the ways of managing risks. Figure 2.5 shows a wide range of risk management options Reynolds, 1996. The organisation must decide which risks it wants to cover with which measures (including testing) and how much it is willing to pay for that.

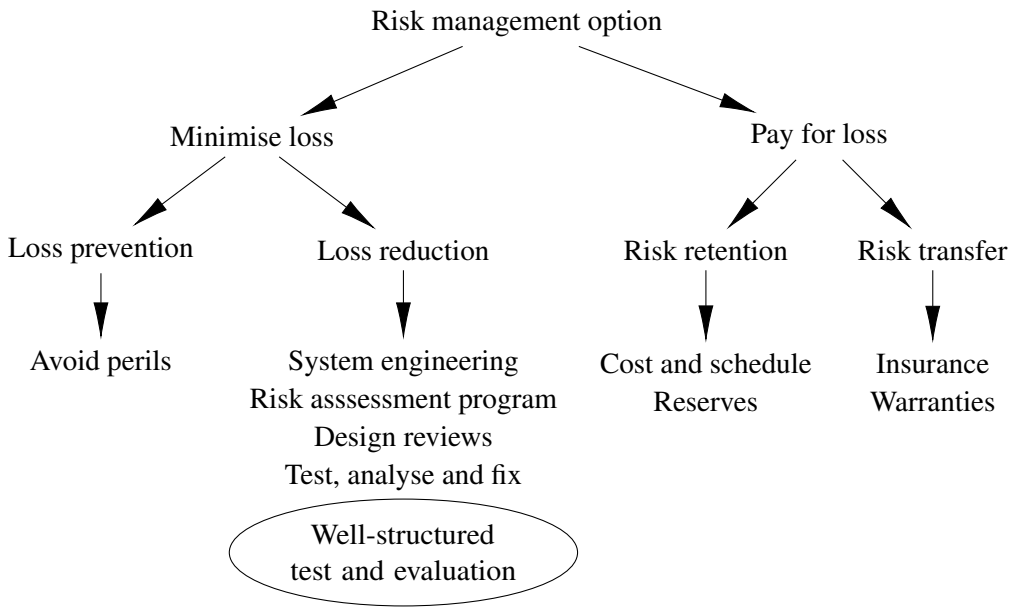


Figure 2.5 Treatment of risks.

2.4.2 Strategy in master test planning

The objective of a test strategy in master test planning is to get organisation-wide *awareness* of the risks that need to be covered and *commitment* on how much testing must be performed where and when in the development process. The following steps must be taken:

- selecting the quality characteristics;
- determining the relative importance of the quality characteristics;
- assigning the quality characteristics to test levels.

Selecting the quality characteristics

In a coordinated effort with all parties involved, a selection of quality characteristics is made on which the tests must focus. Major considerations here are business risks and organisational and international standards and regulations. The selected quality characteristics must be addressed by the tests that are going to be executed. The tests are expected to report on those quality characteristics – to what extend does the system meet the quality requirements, and what are the risks of not conforming to them.

A starting set of quality characteristics that covers the possible quality demands must be available. International standards, such as ISO 25010, provide a useful basis for the organisation to produce its own set of definitions that suits the specific culture and kind of systems it develops. Because many different people are going to discuss the importance of a particular quality characteristic and what to do about it, it is important that everybody is on the same wave-length. This can be promoted by discussing the following topics for each quality characteristics and documenting the results.

- *Clarify the quality characteristic.* For instance, with "performance" do we mean how fast it responds or how much load it can handle? In the case of both aspects being considered relevant, then defining a separate quality characteristic for each will prevent confusion.
- *Provide a typical example.* Formal definitions, especially when copied from international standards, can be too academic for some involved in the development of a test strategy. It helps to give an example of what this quality characteristic could mean for this specific system.
- *Define a way of measuring the quality demand.* During testing it must be analysed to what extent the system meets quality requirements. But how can this be established? Some time should be spent on examining how to establish how well the system behaves with regard to a specific quality characteristic. If this cannot be defined, then it cannot be expected that the test organisation reports anything sensible and reliable about this.

Determining the relative importance of the quality characteristics

After selecting the initial set of quality characteristics, the importance of each in relation to the others must be discussed. Again, the assessed risks are the basis for deciding how important a quality characteristic is for this system. The result of this discussion can be described in a matrix, where the importance of each quality characteristic is indicated as a percentage. An example of this is given in Table 2.3, where a typical organisation specific set of definitions is used. The objective of such a

Table 2.3 Matrix of importance of quality characteristics.

Quality characteristic	Relative importance (%)
Connectivity	10
Efficiency (memory)	–
Functionality	40
Maintainability	–
Performance	15
Recoverability	5
Reliability	10
Security	–
Suitability	20
Usability	–
Total	100

matrix is to provide a general picture and a quick overview. The quality characteristics with the high percentages will get priority attention in the test process, while those with low percentages will get a little attention when time allows. The advantage of filling in the matrix is that the organisation stakeholders are forced to abandon a rather generalised attitude and decide in concrete terms about what is important.

There is a natural tendency to have a cautious opinion when it comes to quality. Most people do not like to say that something is "not important". This may lead to a matrix filled with lots of small numbers, which obscures the general overview. Therefore, a minimum of five percent should be stipulated as a guideline. When a quality characteristic scores zero, this does not necessarily mean that it is totally irrelevant, but that it is not wise to spend scarce resources on it.

Assigning the quality characteristics to test levels

The previous steps have identified which quality characteristics are most important for a system and should form the focus of the test process. Now, the whole test process consists of many test activities performed by various testers and test teams at various times and in various environments (see Section 2.3.2). In order to optimise the allocation of scarce resources, it must be discussed which test level(s) must cover the selected quality characteristics and, roughly, in which way. This results in a concise overview of the kinds of test activities to be performed in which test levels in order to cover which quality characteristics.

This can be presented as a matrix with the test levels as rows and the quality characteristic is covered in this test level using the following symbols:

++	the quality characteristic will be covered thoroughly – it is a major goal in this test level;
+	this test level will cover this quality characteristic;
(empty)	this quality characteristic is not an issue in this test level.

Table 2.4 provides an example of such a strategy matrix. This example shows, for instance, that functionality is the most important quality characteristic (which it often is) and must be tested thoroughly in the unit test and system test. The hardware/software integration test will rely on this and merely perform shallow testing of functionality.

It will focus more on testing how the components interact together technically and with the environment (connectivity), and how the system recovers when a component fails (recoverability). Again, the strength of this matrix is that it provides a concise

Table 2.4 Example of a strategy matrix of a master test plan

	Functionality	Connectivity	Reliability	Recoverability	Performance	Suitability
Relative importance (%)	40	10	10	5	15	20
Unit test	++		+			
Software integration test	+	++				
Hardware/software integration test	+	++		++		
System test	++		+		+	
Acceptance test	+			++		++
Field test			++		++	

overview. If desired, more detailed information can be provided, in plain text. For instance, the required use of specific tools or techniques, by adding footnotes to the matrix.

2.4.3 Strategy for a test level

The objective of a test strategy for a particular level is to make sustained choices as to what to test, how thoroughly, and which test techniques to apply. The test strategy must explain to all the parties involved why testing the system in this way is the best choice, considering the time pressure and scarce resources. The choices are not arbitrary but can be related to what the organisation finds most important (in terms of business risks). Properly executed, the development of a test strategy is not just a technical exercise, but also has a highly political aspect – it assures the stakeholders that testing is not just a goal in itself but is for the benefit of the whole organisation. At the same time, it makes them aware that they will not get the perfect test but the best test considering the circumstances.

Ultimately the test strategy results in defining specific test techniques applied to specific parts of the system. Each can be seen as a concrete test activity and can be planned and monitored separately. This makes the test strategy an important management tool for the test manager.

The following steps must be taken:

1. selecting the quality characteristics;
2. determining the relative importance of the quality characteristics;
3. dividing the system into subsystems;
4. determining the relative importance of the subsystems;
5. determining test importance per subsystem/quality characteristic combination;

6. establishing the test techniques to be used.

If a master test plan, including a test strategy, is available, then the test strategy for a particular test level must be based on this. In that case, steps 1 and 2 will be a relative easy translation of what has already been discussed and decided on a global level. The test strategies for the various test levels can be seen as the more concrete refinements of the global goals that were set in the master test strategy. If a test strategy for the master test plan is not available, the test manager of a particular test level is obliged to initiate and co-ordinate discussions about business risks and quality characteristics, just as would be done for a master test plan.

Selecting the quality characteristics

A list of relevant characteristics is determined. The same instructions apply as for the corresponding step in the strategy development for the master test plan.

Determine the relative importance of the quality characteristics

The importance of each selected quality characteristics in relation to the others is determined. The results are described in a matrix, where the importance of each quality characteristics is indicated as a percentage. Again, the same instructions apply as for the corresponding step in the strategy development for the master test plan. Table 2.5 is an example of a matrix showing the relative importance of the quality characteristics – it is based on the results from the master test plan (see Table 2.3). The numbers in this example differ from those in the master test plan because not all quality characteristics are tested at this particular test level, however, the numbers still have to total 100 percent. Also, if two characteristics have the same relative importance at the level of the master test plan, they do not necessarily have to be tested with the same relative importance at this test level.

Table 2.5 Matrix of importance of quality characteristics for a particular test level.

Quality characteristic	Relative importance (%)
Functionality	40
Performance	25
Reliability	10
Suitability	25
Total	100

Dividing the system into subsystems

The system is divided into subsystems that can be tested separately. Although the term "subsystem" is used here and in the rest of this section, one can think of "component" or "functional unit" or other such terms. The bottom line is that quality demands are not the same for individual parts of the system. Moreover, the various subsystems may differ in terms of risks for the organisation.

In general, this dividing step follows the architectural design in which the systems components and their relationship are already specified. Deviations from this in the test strategy should be clearly motivated. Examples of alternative criteria for division are the extent of risk, or the order of release by the developer. If, for instance, a data

conversion is part of the implementation of a new product, then the conversion module can be treated as a separate subsystem.

The total system is added to the list of subsystems. This serves to indicate that some quality demands can be evaluated only by observing the behaviour of the complete system.

Determining the relative importance of the subsystems

The relative importance of the identified subsystems is determined in much the same way as for the quality characteristics in step 2. The risks for each subsystem are weighted and the results documented in a matrix (see Table 2.6 for an example). It is not about exact percentages in this matrix, but rather of getting an overview of how important the different subsystems are. It should not express the personal view of the test manager, but rather of the people who know how the product is used in its intended environment (such as end users and product management). This step is useful in helping to force these people to form an opinion about what they deem to be important.

Table 2.6 Matrix of relative importance of subsystems

Subsystem	Relative importance (%)
Part A	30
Part B	10
Part C	30
Part D	5
Total system	25
Total	100

Determining the test importance per subsystem/quality characteristic combination

This step refines the strategy by combining the assessments of quality characteristics and subsystems. For example, a refinement may be that performance is an important quality characteristic (rating 25 percent) but that this holds predominantly for subsystem B (which, for example, takes care of routing image data) and not at all for subsystem D (which, for instance, logs statistical data). This kind of information helps to pinpoint areas where testing should focus even more closely.

The result of this refinement step can be presented as a matrix with the quality characteristics as rows and subsystems as columns. Each intersection indicates how important this quality characteristic is for this subsystem using the following symbols:

++	the quality characteristic is predominant for this subsystem;
+	the quality characteristic is relevant for this subsystem;
(empty)	this quality characteristic is insignificant for this subsystem.

Table 2.7 gives an example of such a matrix. It is emphasized once more that test strategy development is not a mathematical exercise. The purpose is to assist in choosing where to put in the major efforts and which areas to ignore or touch briefly. The strength of a matrix such as shown in Table 2.7 is that it provides a concise overview of what the organisation finds important and where testing should focus.

Table 2.7 Matrix of relative importance per subsystem and quality characteristic

Relative importance (%)	Part A	Part B	Part C	Part D	Total system
	100	30	10	30	5	25
Functionality	40	++	+	+	+	+
Performance	25	+	++	+		+
Reliability	10		+			++
Suitability	25	+		+	+	++

It reflects the choices that the organisation has made during the complex process of analysing risks, weighing consequences, and allocating resources.

Establishing the test techniques to be used

The final step involves the selection of the test techniques that will be used to test the subsystems on the relevant quality characteristics. The matrix that resulted from the previous step (see Table 2.7) can be regarded as a "contract" – the test team is expected to organise and execute a testing process that covers what the organisation has defined to be important. There will be rigorous testing in one area and more global testing in others, in accordance with what is specified in the matrix.

Usually, most of the system's functions are tested by means of test cases specifically designed for that purpose (dynamically explicit testing). To this end, many test design techniques are available. Testing can also be done by analysing metrics during the development and test processes (dynamically implicit testing) or by assessing the installed measures on the basis of a check list (static testing). For instance, performance can be tested *during* the testing of functionality by measuring response times with a stopwatch. No explicit test cases were designed to do this – it is tested *implicitly*. Also, security, for instance, can be tested statically by reviewing the security regulations.

It is the responsibility of the test manager to establish the set of techniques that "can do the job". Where the matrix shows a high importance, the use of a rigorous technique achieves a high coverage, or the use of several techniques, is an obvious choice. A low importance implies the use of a technique with lower coverage, or dynamically implicit testing. Just which techniques are the best choice depend on various factors, including:

- *Quality characteristic to be tested.* For instance, a technique may be very good at simulating realistic usage of the system, but very poor at covering variations in functional behaviour.
- *Area of application.* Some techniques are for testing human machine interaction, while others are better suited to test all functional variations within a certain component.
- *Required test basis.* Some techniques require that a specific kind of system documentation is available, for instance state transition diagrams or pseudocode or graphs.
- *Required resources.* The application of a technique requires a certain allocation of resources in terms of human and machine capacity. Often this also depends

on the "required knowledge and skills".

- *Required knowledge and skills.* The knowledge and skills of the testing staff can influence the choice of techniques. The effective use of a technique sometimes requires specific knowledge and/or skills of the tester. In depth domain expertise can be a prerequisite, and some techniques require a trained (mathematical) analytical talent.

The test manager's experience and knowledge of the various test design techniques and their strength and weaknesses is vital in establishing the required test techniques. It is not unusual to find that available test techniques must be tuned or combined into new test techniques that suit a specific test project. The selection and tuning of the techniques must be done early in the test process, allowing for the test team to be trained in required areas.

The result of this step is the definition of the techniques to be used in the test on each subsystem. See table 2.8 for an example. Each + in the table signifies that the corresponding technique will be applied to this particular part of the system. From a management point of view this is a very useful table because each + can be treated as a separate entity that can be planned and monitored. Each can be assigned to a tester who is then responsible for all necessary activities, such as designing test cases and executing tests. For large systems this table is usually further detailed by subdividing the system parts into smaller units that can be properly handled by a single tester. Optionally – particularly in large test projects – the test strategy is not specified in such detail until the preparation phase. In order to ensure that the most important tests

Table 2.8 Establishing which techniques to apply to which part of the system.

Applied test technique	Part A	Part B	Part C	Part D	Total system
W	+	+	+		
X		+	+		
Y	+			+	+
Z					+

are executed as early as possible, this step also determines the order of priority of the established tests (technique-subsystem combination).

Strategy changes during the test process

The test manager must not have the illusion that the test strategy will be developed once and then remain fixed for the duration of the project. The test process is part of a continuously changing world and must react properly to those changes.

In the real world, development and test projects are often put under pressure, especially in the later stages of a project. Suddenly the schedule is adjusted – usually the available time is reduced. The test manager is asked to perform fewer or shorter tests. Which tests can be cancelled or carried out in less depth? Using the test strategy as a basis, the test manager can discuss these topics with the stakeholders in a professional way. When changed circumstances require that testing should be other than previously agreed, then the expectations of just what testing should achieve must also change. Strategic issues must be re-evaluated – has our notion of which aspects

are more important than others changed? Are we willing to accept the increased risk if testing is reduced in a certain area?

A similar situation arises when the release contents change – for example, extra functionality may be added to the system, or the release of parts of the system delayed, or the product is now also targeted for another market segment with different requirements and expectations. In such cases the test strategy must be re-evaluated – is the testing, as it was planned, still adequate for this new situation? If the relative importance of the quality characteristics or the subsystems changes significantly, the planned tests must be changed accordingly.

Another reason to change the test strategy can be because of the results of the testing itself. When testing of a certain part of the system shows an excessive number of defects, it may be wise to increase the test effort in that area – for instance, add extra test cases or apply a more thorough test technique. The inverse is also true – when excessively few or only minor defects are found, then it should be investigated if the test effort may be decreased. In such situations a major discussion issue should be "what is the risk as we perceived now through testing, and does this justify a decision to increase or decrease testing effort?"

Strategy for maintenance testing

The technique described above for developing a test strategy can be used for a newly developed system without reservation. A legitimate question to ask is to what extent the steps described are still useful for a maintenance release of a system that has been developed and tested before.

From the perspective of test strategy, a maintenance release differs mainly in the *change* of a fault happening. During maintenance there is a risk that faults are introduced with changes in the system. Those intended changes of system behaviour must, of course, be tested. But is is also possible that the system, which used to work correctly in the previous release, does not work in the new release as a side effect of the implemented changes. This phenomenon is called *regression*. In maintenance releases, much of the test effort is dedicated to testing that previous functionality works correctly – this is called regression testing. Because the chance of a fault occurring changes when the product enters the maintenance stage, so does the associated risk. This in turn changes the relative importance of the subsystems. For example, a subsystem had a high importance when it was newly developed, because of its business critical nature. The subsystem is unchanged in the maintenance release, so the associated risk is now low and the subsystem may be given a low importance for this maintenance release.

Therefore, in developing a test strategy it is often useful to substitute the concept of "subsystem" with "change". These changes are usually a set of "change requests" that will be implemented, and a set of "known defects" that are corrected. For each change, it is analysed which system parts are modified, which may be indirectly affected, and which quality characteristics are relevant. Various possibilities exist for testing each change – depending on the risks and the desired thoroughness of the test:

- a limited test, focused only on the change itself;
- a complete (re)test of the function or component that has been changed;
- a test of the coherence and interaction of the changed component with adjacent

Table 2.9 Matrix of the relative importance of changes and regression.

Changes/regression	Relative importance (%)
Change request CR-12	15
Change request CR-16	10
Change request CR-17	10
Defect 1226, 1227, 1230	5
Defect 1242	15
Defect 1243	5
...	30
Regression	10
Total	100

components.

With every implementation of change, the risk of regression is introduced. Regression testing is a standard element in a maintenance test project. Usually a specific set of test cases is maintained for this purpose. Depending on the risks and the test budget available, a choice has to be made to either execute the full regression test set or to make a selection of the most relevant test cases. Test tools can be used very effectively to support the execution of the regression test. When the regression test is largely automated, selecting which regression test cases to skip is no longer necessary because the full regression test set can be executed with limited effort.

The decision to formulate the test strategy in terms of change requests instead of subsystems depends largely on the number of changes. When relatively few are implemented, the testing process is better managed by taking those changes as basis for risk evaluations, planning, and progress tracking. When subsystems are heavily modified by many implemented changes, they can be treated in the same way as when they were newly developed. Then it is usually preferred to develop the test strategy based on subsystems. If it is decided to formulate the test strategy in terms of change requests, then the steps for developing a test strategy are as follows:

1. determining changes (implemented change requests and corrected defects);
2. determining the relative importance of the changes and regression;
3. selecting quality characteristics;
4. determining the relative importance of the quality characteristics;
5. determining the relative importance per change (and regression)/quality characteristic combination;
6. establishing the test techniques to be used.

Table 2.9 shows an example of the matrix that results from the first two steps. The other steps are similar to those described earlier in this section. The example shows that regression has been assigned a value of 10 percent. It should be noted that this reflects the importance that is allocated to the issue of regression. It does not mean that 10 percent of the test process should be regression testing. In practice the total effort

of regression testing is much larger than that. The reason that, usually, a relatively small amount of the system functionality is changed while a much larger part remains unchanged. As noted earlier in this section, automation of the regression test can be a tremendous help in freeing scarce time and resources for maintenance testing.

2.5 Test design techniques

Structured testing implies that it has been well thought just which test cases are needed to accomplish the required goals, and that those test cases are prepared before actual test execution. This is in contrast to just "making up smart test cases as you go along". In other words, structured testing implies that test design techniques are applied. A test design technique is a standardised method of deriving test cases from reference information.

Often software testing is considered to be a process to demonstrate that the program functions correctly, but this definition is erroneous. A more appropriate interpretation is that testing is a process to find as many defects as possible. The difference may sound irrelevant, but it is essential in comprehending the philosophy of testing and setting the right kind of goal. If the goal were to be demonstrating that the software works correctly, then the whole process is in danger of steering towards the avoidance of the defects rather than deliberately making the test fail. Testing could thus be described as a destructive sadistic process.

There are a few basic principles in software testing that greatly affect the outcome and efficiency of the testing process. The most important principles are:

- A test case must contain a definition of the expected output and results.
- Each test result should be thoroughly inspected.
- Test cases must be written for both invalid and unexpected input conditions, as well as for input conditions that are valid and expected.
- Test cases should be stored and repeatable.
- Testing effort should be planned in assumption that defects will be found.
- The probability of existence of defects is proportional to the number of defects already found.

These principles are intuitive guidelines for test planning and help getting a grip on the nature of the testing process and its goals.

2.5.1 Characteristics

To assist in the selection of test design techniques and to be able to compare them, some general characteristics of test design techniques are discussed.

Dynamic vs. static testing

Dynamic testing is the most common type of testing and consists of two general types of testing: black-box testing and white-box testing. The difference between the two is the focus and scope of testing. Black-box testing focusses on the behaviour of the system, while white-box testing focusses on the implementation. There is a third variant of dynamic testing, which is a combination of white-box and black-box testing: grey-box testing. Grey-box testing is a black-box testing method that, in addition

to the module specification, has some information about the actual code of the Unit Under Test (UUT) as a basis for the test case design.

Static software testing is the inspection and review of the code or documentations. In static testing, the software is not actually used at all, it is visually inspected. Static testing is proven to be an extremely effective type of testing in both the means of cost-effectiveness and defect spotting effectiveness.

Black-box or white-box

Black-box test design techniques are based on the functional behaviour of systems, without explicit knowledge of the implementation details. In black-box testing, the system is subjected to input and the resulting output is analysed as to whether it conforms to the expected system behaviour. White-box test design techniques are based on knowledge of a system's internal structure. Usually it is based on code, program descriptions, and technical design.

However, the boundary between black-box and white-box characteristics is fuzzy because what is white-box at the system-level translates to black-box at the subsystem level.

Black-box testing methods

An exhaustive testing of all input values leads to practically an infinite number of test cases. The test cases must be scaled down and this can be achieved through *equivalence partitioning* and *boundary-value analysis* techniques. These techniques intuitively scale down the number of test cases.

In *equivalence partitioning* the input values are divided into two or more partitions depending on the function of the UUT. The partition classes are value ranges that are either valid or invalid equivalence classes. In other words, every value on the class range can be safely assumed to be either a valid or invalid input. For example, if the UUT is a function that requires a voltage as a floating point input V in the range between 0.0 and 5.0 volts, the values of V can be divided into three equivalence classes: valid class $0.0 < V \leq 5.0$, and invalid classes $V \leq 0.0$ and $V > 5.0$.

With *boundary-value analysis* test cases can be constructed from the equivalence partitions. In this technique the test cases explore the boundary conditions of the equivalence classes. For example, in the case of the voltage V, the boundary condition test cases would be 0.0, 5.0, -0.001 and 5.001. With these cases probable comparison errors can be detected. In some cases it can be profitable to analyse the output boundaries and try to produce a test case that causes the output to go beyond its specific boundaries. In reality, boundary-value analysis can be a challenging task due to its heuristic nature and its required creativity and specialisation towards the problem.

In addition to these techniques, a *defect guessing* technique can be used to add test cases that are not covered by the equivalence classes. These test cases can be inputs in invalid data types, void inputs, or other defect prone inputs. Effective defect guessing requires experience and natural adeptness.

White-box testing methods

Logic coverage testing is a white-box testing method and it is the most common type of coverage test. Complete logic path testing is impossible due to the practically infinite number of test cases, so the test must be scaled down but should still try to meet the required level of coverage. Finding the optimal test case set is a challenging task.

Statement coverage testing is a test in which every statement of the UUT is executed. Such a test is easily designed but is a very weak type of test since it overlooks possible defects in branch decision statements. It can be said that statement coverage testing as such is so weak that it is practically useless.

A stronger logic coverage test is the *decision coverage* test. In this test type the test cases are written so that every branch decision has a true and false outcome at least once during the test run. Branch decisions include `switch-case`, `while`, and `if-else` structures. Often this also covers statement coverage, but it is advisable that statement coverage is ensured. Decision coverage is stronger than statement coverage but still rather weak since some conditions can be skipped to fulfil complete decision coverage.

Condition coverage is the next stronger criterion. In condition coverage test cases are written so that every condition in a decision takes on all possible outcomes at least once. In addition, to benefit from this criterion, statement coverage must be added on top of the condition coverage. The decision coverage and condition coverage can be combined to achieve acceptable logic coverage.

The choice of criterion depends on the structure of the UUT. For programs that contain only one condition per decision, decision coverage is sufficient. If the program contains decisions that contain more than one condition, it is advisable to apply condition coverage.

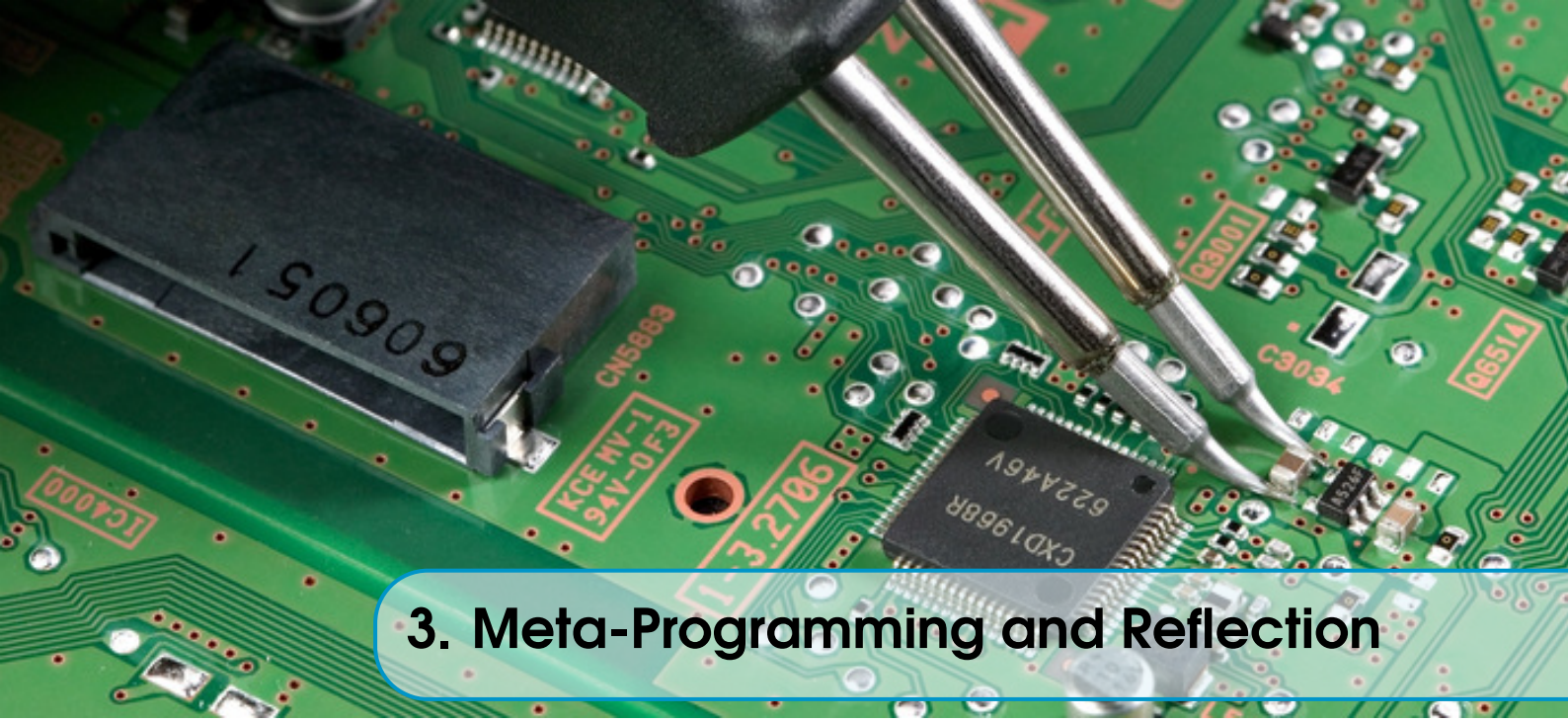
Test-driven design

The general problem with embedded system development is that hardware is developed concurrently with the software. The hardware may go through several iterations causing the need for changes in the software. Moreover, the hardware is typically available only late in the project. This often leads to a situation where software testing is pushed to the first prototype round where it ends up being endless about debugging and regression testing. This process method wastes time and resources.

Test-Driven Development (TDD) addresses this problem. TDD is an agile software development method with the main characteristic that unit tests are written before the actual code. The benefit of this reverse thinking is that it forces developers to plan the unit and understand the specification completely before coding. This way, code is more optimised and defect injections from poor ad hoc planning are prevented. When every unit is tested and planned individually, the problems in the integration phase are also minimised. The unit and integration tests are performed frequently during the coding phase, thus ensuring rigorous and comprehensive testing. Other benefits of TDD are that it offers high test coverage and leads to modular design (Grenning, 2011).

There are also some drawbacks to the TDD method. One problematic feature is

the requirement of complete dedication from the developer. As mentioned earlier, writing comprehensive test cases requires a certain “destructive oriented” mindset that differs greatly from the mindset of a software developer. Adapting to the dualistic feel of the process requires times and patience from the developer and during this adaptation process productivity can be temporarily reduced. In addition, some code can be challenging to modularise, which also means that writing test cases for these parts can be difficult or even impossible.



3. Meta-Programming and Reflection

In the development of (embedded) software solutions, the choice is more and more made to make use of a fast prototyping language that sacrifice performance and efficiency of the solution over the ease of use and efficiency in coding the prototype(s).

Python and C++ are extremely different languages, and most of the differences aren't strictly advantageous in one direction or the other. That said, for most uses, it's easy to pick a side and make a good argument for or against particular language and implementation features. The differences between the two stem from a general difference in philosophy.

C++ tries to give you every language feature under the sun (circa 1990, at least) while at the same time never (forcibly) abstracting anything away that could potentially affect performance.

Python tries to give you only one or a few ways to do things, and those ways are designed to be simple, even at the cost of some language power or running efficiency.

In many cases, Python's philosophy is an advantage because it lets you get most tasks done more easily and more quickly with less mental overhead.

Naturally, when developing for embedded solutions, power and efficiency are an important matter, which is why most solutions are still developed (for production) in C++. However, in earlier phases in development (see Figure 5.1 in chapter 5) it is much more important to build prototypes and proof-of-concepts *fast* rather than 'perfect'; Python tends to work better here.

A number of advanced programming techniques that can help by testing and prototyping include **reflection** and **metaprogramming**.

Definition 3.1 — Reflection.
Reflection is the process by which a program can observe and modify its own structure and behavior at runtime.

Reflection is the process by which a program can perform introspection. This

introspection usually involves the ability to observe and modify its own structure and behavior at runtime. From theoretical perspective reflection relates to the fact that program instructions are stored as data. The distinction between program code and data is a matter of how the information is treated. Hence programs can treat their own code as data and observe or modify them.

Reflection is important since it allows programmers to write programs that do not have to "know" everything at compile time. The code can be written against known interfaces, but the actual classes to be used can be instantiated using reflection from configuration files. Reflection is tightly integrated in scripting languages such as Python (but many other frameworks use it as well), since it feels more natural within the general programming model of those languages.

Reflection is powerful, but should not be used indiscriminately. It introduces a performance overhead (types need to be determined at runtime), can impose security risks (program requires runtime permissions that can be abused), etc. If it is possible to perform an operation without using reflection, then it is peferable to avoid using it.

Many of the features introduced with reflection can be exploited when performing metaprogramming.

Definition 3.2 — Metaprogramming.
Metaprogramming is writing programs that write or manipulate other programs as their data.

Metaprogramming is useful because it can save programmers valuable time. Some languages have support to metaprogram themselves and this allows to create code with great expressive power.

The ability to change (runtime) programs by means of other programs is extremely powerful (and should, therefore, be used cautiously). Metaprogramming allows you to 'fix' problems in code (or code-bases) where you have no access to the actual sourcecode.

In this chapter we introduce advanced programming concepts, like reflection and metaprogramming, in Python. We start with a brief overview of the characteristics of Python to better understand what kind of a language it is. We also present a brief recap of Object-oriented programming, and how it is done within Python.

3.1 Characteristics of Python

Important characteristics of Python:

- Python variables are object references;
- Python is dynamic;
- (Almost) everything in Python is an object;
- Python allows for *reflection*.

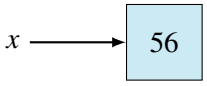
We go into detail on these characteristics later.

Python variables are object references

When the following command is executed in Python:

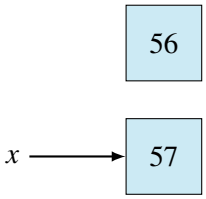
```
x = 56
```

an int-object is created with the value `56` and `x` refers to that object.



int-object

The int-object is 'immutable': the value `56` cannot be changed. The value of `x` can be changed, however. After the statement `x += 1` the situation is as follows:



What will happen with the int-object with value `56` depends on the circumstances. It could be that another variable is pointing towards it, then nothing is done with it. If no other variable is pointing to it, the garbage-collector will remove it.

Python is dynamic

The following statements are allowed in Python:

```
x = 56
x = 'string'
```

First, `x` refers to an integer, and then `x` refers to a string. This example shows that `x` is not bound to a specific type.

This is one of the examples that show that Python is a dynamic language. Other examples are:

- Variables can be added and removed. All variables are stored in a dictionary.
- Attributes and methods can be added and removed to objects.
- All statements from an imported module are executed immediately.
- Each class stores which subclasses have been made.

(Almost) everything in Python is an object

Not everyone realises it, but Python is an Object-Oriented language. Many syntactic constructs in Python appear to be classes and objects, for example:

- **`int`**, **`bool`**, **`float`**, **`str`** are all classes;
- Functions and modules are objects;
- A class is an object.

This can be confusing. When we think of a class, we have a different association as when we are thinking of a function. It appears, however, that functions are also objects. Functions can be assigned attributes. Functions can be passed as a parameter to other functions, and functions can also be handed back as a return-value.

Python allows for reflection

In Python it is possible to investigate a program, ask for the properties of objects and change the structure and behaviour of a program at runtime. This kind of runtime adaptation is know as *reflection* and *meta-programming*, and allow for some interesting programming dynamics.

3.2 Object-oriented programming in Python

In an object all the data, that belong together, and the functions, that are executed on that data, are stored together. The data in an object are called its *attributes* and the functions its *methods*.

The type of an object is called *class*. An *object* is an instance (or instantiation) of a class. An example class is the following:

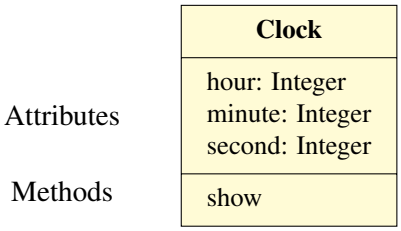


Figure 3.1 UML definition of Clock-class.

In objects the values of the attributes are instantiated.

During the execution of a program, objects are created and (eventually) destroyed. When an object is created, the *constructor* is called. Python only allows for a single constructor, in contrast with other languages (like C++) that allow multiple constructors per class. Removing an object is done by calling the *destructor* of an object. A destructor can close and release the linked resources. The constructor and destructor are both special methods of the class.

Every class has a default-constructor and -destructor. If the programmer does not specify its own constructor/destructor, the default is used instead.

In Python a constructor looks as follows:

```
def __init__(self, ...):
    ...
    ...
```

The constructor always has **self** as first argument. This argument refers to the object to which the constructor belongs.

The destructor looks as follows:

```
def __del__(self):
    ...
    ...
```

An implementation of the class example presented in Figure 3.1 above:

```
class Clock:
```

```
    def __init__(self, hour=0, minute=0, second=0): # constructor
        self.hour = hour     # attribute
        self.minute = minute # attribute
        self.second = second # attribute

    def show(self):
        s = "time: " + \
        "{0:02d}:{1:02d}:{2:02d}".format(self.hour,  self.minute, self.second)
        print(s)

c1 = Clock(10, 25, 30)
c1.show()
```

Output:

```
time: 10:25:30
```

Note that class-methods (member functions) also contain **self** as first argument. Also, attributes of a class/object are preceded by **self**.

When the method is called, however, the **self**-parameter is omitted. In general, the following transformation is taking place:

<object>.<method>(...) ⟶ <class>.<method>(<object>, ...)

For example `c1.show()` is transformed into `Clock.show(c1)`.

Because Python is a dynamic language, attributes can be added and removed 'on the fly'. A demo:

```
print(vars(c1))
del(c1.second)  # delattr(c1, 'second')
print(cars(c1))
```

Output:

```
{'second': 30, 'minute': 25, 'hour': 10}
{'minute': 25, 'hour': 10}
```

3.2.1 Class attributes

A class attribute is an attribute that is not linked to a specific instance of a class[1]. For example, to keep a record of the number of clocks, we can introduce the attribute `nclock`. This is coded as follows:

```
class Clockv2:

    nclock = 0                  # class attribute

    @staticmethod
    def show_nclock():          # class method
```

[1] A class attribute is comparable to a *static* variable in a C++ class

```
        print(Clockv2.nclock)

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
        Clockv2.nclock += 1

    def __del__(self):
        Clockv2.nclock -= 1
        print("destructor called")

    def show(self):
        s = "time: " + \
        "{0:02d}:{1:02d}:{2:02d}".format(self.hour,  self.minute, self.second)
        print(s)

c1 = Clockv2(10, 25, 30)
Clockv2.show_nclock()
c2 = Clockv2(10, 25, 31)
ClockV2.show_nclock()
```

Output:

```
1
2
destructor called
destructor called
```

Exercise 3.1

1. What is shown when the last four statements of the example above are changed to:

   ```
   c1 = Clockv2(10, 25, 30)
   Clockv2.show_nclock()
   c1 = Clockv2(10, 25, 31)  # 'c1' instead of 'c2'
   Clockv2.show_nclock()
   ```

2. Add the following code; what is the new output?

   ```
   c3 = c1
   Clockv2.show_nclock()
   ```

3. Add the following code; what is the new output?

   ```
   import copy
   c3 = copy.copy(c1)
   Clockv2.show_nclock()
   ```

Class attributes are never preceded by `self`. Class methods are preceded by the decorator *@staticmethod*. This changes the method into a function.

Exercise 3.2
Add the following code to the clock example above:

```
c = Clockv2()
print(type(c.show))
print(type(c.show_nclock))
```

What is the output? What is the output when *@staticmethod* is removed?

3.2.2 Printing objects

The method `__str__` defines how an object is supposed to be transformed to a string (for printing)[2]. An example;

```
class Clockv3:
    ...
    ...

    def __str__(self):
        s = \
        "{0:02d}:{1:02d}:{2:02d}".format(self.hour, self.minute, self.second)
        return s

c1 = Clockv3(10, 25, 30)
print('c1:', c1)

c2 = Clockv3(10, 25, 31)
print('c2:', c2)
```

Output:

```
c1: 10:25:30
c2: 10:25:31
```

Exercise 3.3

1. Add the following code to `Clockv3`:

   ```
   a = [c1, c2]
   print('a: ', a)
   ```

   What is the output?
2. Replace in the class `Clockv3` `__str__` by `__repr__`. What is the output now?

[2] This is similar to the standard `toString()` method of Java-objects.

3.2.3 Inheritance

Inheritance is a mechanism in which one object acquires all the properties and behaviors of the parent object. The idea behind inheritance is that you can create new classes that are built upon existing classes or specifying a new implementation to maintain the same behavior (realizing an interface). Such an inherited class is called a subclass of its parent class or super class. It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a hierarchy.

An example of inheritance in Python:

```
class Clockv4(Clockv3):    # Clockv4 is an extension of Clockv3
    def tick(self):
        self.second = (self.second+1)%60
        if self.second == 0:
            self.minute = (self.minute+1)%60
            if self.minute == 0:
               self.hour = (self.hour+1)%24

c = Clockv4(23, 59, 58)

print(c)
c.tick()
print(c)
c.tick()
print(c)
c.tick()
print(c)
c.tick()
print(c)
```

Output:

```
23:59:58
23:59:59
00:00:00
00:00:01
```

It is also possible to extend classes dynamically. This is done as follows:

```
def tick(self):
    self.second = (self.second+1)%60
    if self.second == 0:
        self.minute = (self.minute+1)%60
        if self.minute == 0:
            self.hour = (self.hour+1)%24

Clockv3.tick = tick
```

Furthermore, it is possible to add a method to only one instance:

```
c = Clockv3(23, 59, 57)

def tick2(self):
    print('tick2')

import types
c.tick2 = types.MethodType(tick2, c)
```

Only the instance `c` now has the method `tick2`.

3.2.4 Information hiding

Initially considered to be a part of *Encapsulation*, Information or Data Hiding is a mechanism for restricting access to some of an object's components (e.g., by use of the 'private'-keyword in C++ and Java).

> **Aside 3.1 — Encapsulation vs. Information Hiding.**
> Initially Information/Data Hiding was considered part of Encapsulation, where the access to elements of objects/classes was a core part of the wrapping function that Encapsulation provides.
>
> More recent definitions of Encapsulation lack any reference to Information Hiding and the concepts are considered separate as such. Encapsulation is merely the facilitation of bundling data with methods (or other functions) operating on that data, with no claim about any hiding functionalities.
>
> Likewise, if encapsulation was "the same thing as information hiding", then one might make the argument that "everything that is encapsulated is also hidden", which is obviously not true.

While Python clearly supports the concept of encapsulation, the concept of *information hiding* is only limited available in Python. Attributes that start with `__` (double underscore), but do not end with `__` are considered private. `__<attribute-name>` is replaced by `_<class_name>__<attribute-name>`. An example:

```
class C:
    def __init__(self):
        self.__attr = 123
    def __repr__(self):
        return str(self.__attr)

c = C()
print('c:', c)
```

The following statement returns an error: `print('c.__attr:', c.__attr)`, because internally `c.__attr` has been replaced by `c._C__attr`. The following is allowed, however:

```
print('c.__attr:', c._C__attr)
```

Which indicates that private variables are still available indirectly. Furthermore, Python does not have any notion of *protected* attributes.

Exercise 3.4
Would it be possible to define a *private method*?
Give an example of one. ■

3.2.5 Abstract methods and abstract classes

An abstract method is a method defined in a base class, but may not provide any implementation. The simplest way to achieve this is python is the following:

```
class timeDevice(object):
    def get_time(self):
      raise NotImplementedError
```

Unfortunately, this does not raise an error when attempting to make an object of this class, but only when attempting to invoke the abstract method. This can present problems when implementing a child of `timeDevice`, but forget to implement the `get_time` method.

To circumvent this, you can use the `abc` module.

```
import abc

class timeDevice(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def get_time(self):
      """Method should do something"""
```

Using this approach, you get an error directly when trying to instantiate the base (abstract) class, or any inherited class that does not implement the abstract method.

3.2.6 Multiple inheritance

Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can inherit characteristics and features from more than one parent object or parent class. It is distinct from single inheritance, where an object or class may only inherit from one particular object or class. Multiple inheritance has been a sensitive issue for many years, due to complexity and ambiguity in situations such as the *diamond problem* (discussed below in section 3.2.8).

In Python it is possible to define multiple inheritance. The following example combines the classes `Calendar` and `Clock`.

```
class Calendar:

    @staticmethod
    def leapyear(year):
        return year % 400 == 0 or (year % 4 == 0 and year % 100 != 0)

    def __init__(self, day, month, year):
```

```
        self.day = day
        self.month = month
        self.year = year

    def __repr__(self):
        return \
        "{0:02d}/{1:02d}/{2:4d}".format(self.day, self.month, self.year)

    def next_day(self):
        self.day += 1
        if (self.month in [1,3,5,7,8,10,12] and self.day == 32) or \
            (self.month in [4.6.9.11] and self.day == 31) or \
            (self.month == 2 and self.day == 29 and \
                not Calendar.leapyear(self.year)) or \
            (self.month == 2 and self.day == 30 and \
                Calendar.leapyear(self.year)):
            self.day = 1
            self.month += 1
            if self.month == 13:
                self.month = 1
                self.year += 1

class CalendarClock(Clockv4, Calendar):
    def __init__(self, day, month, year, hour, minute, second):
        Clockv4.__init__(self, hour, minute, second)
        Calendar.__init__(self, day, month, year)

    def tick(self):
        previous_hour = self.hour
        Clockv4.tick(self)
        if self.hour < previous_hour:
            self.next_day()

    def __repr__(self):
        return Calendar.__repr__(self) + ", " + Clockv4.__repr__(self)
```

Every class is derived from the class **object**. That means that, in Python

```
class Calendar:
    ...
```

is the same as

```
class Calendar(object):
    ...
```

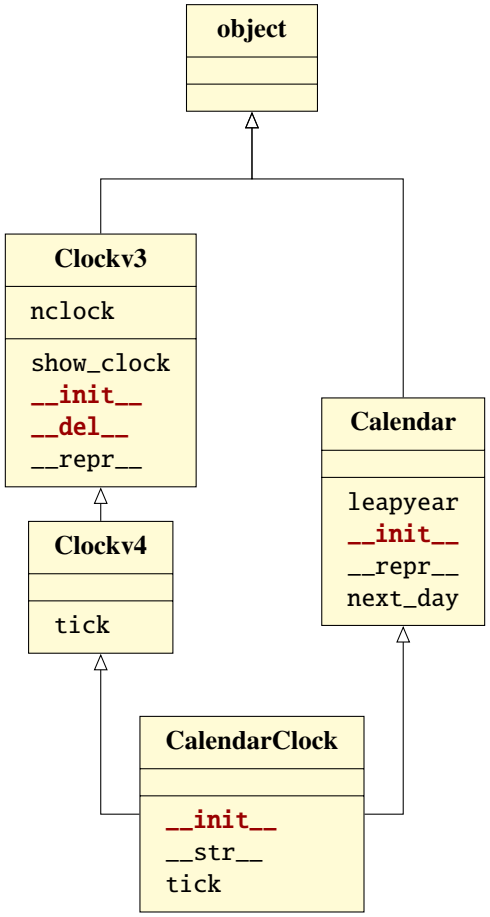


Figure 3.2 Class diagram for multiple inheritance example.

3.2.7 Class-methods

Class methods are methods that are defined that can be used with just the class-name (in contrast with instance- or object-methods, which perform functions on a particular instance or object). In C++ class methods are indicated by means of the 'static'-keyword (cf. the definition of class attributes above in section 3.2.1

In Python, class methods are defined by using `cls` as the first parameter of the method. This refers to the current class (instead of **`self`** which refers to the object). An example:

```
class B:
    @classmethod
    def showClassInfo(cls):
        print('name:', cls.__name__)
        print('bases:', [e.__name__ for e in cls.__bases__])
        print('subclasses:', [e.__name__ for e in cls.__subclasses__()])

class D(B):
    pass
```

```
b = B()
d = D()
b.showClassInfo()   # cls == <class B>
d.showClassInfo()   # cls == <class D>
```

Output:

```
name: B
bases: ['object']
subclasses: ['D']

name: D
bases: ['B']
subclasses: []
```

A method of a base class can be called from an derived class. A class method knows, by means of the parameter `cls` what the current class is.

It is also possible to add a class method dynamically:

```
def showClassInfo(cls):
    print('name:', cls.__name__)
    print('bases:', [e.__name__ for e in cls.__bases__])
        print('subclasses:', [e.__name__ for e in cls.__subclasses__()])
    print()

class B:
    pass

B.showClassInfo = classmethod(showClassInfo)
```

Exercise 3.5

How many subclasses does **object** have? ■

3.2.8 Diamond problem

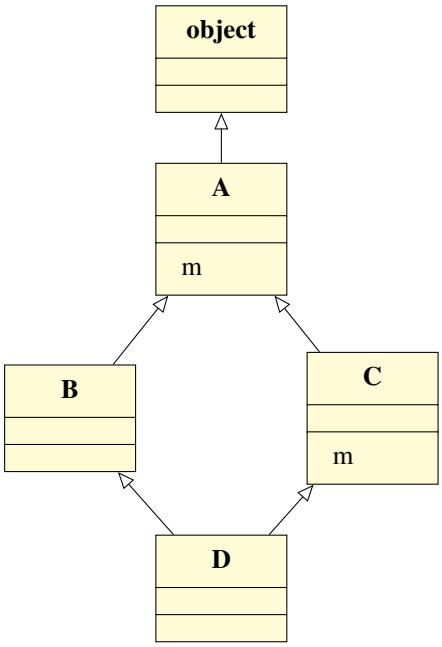
The "diamond problem" (sometimes referred to as the "deadly diamond of death") is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

Multiple inheritance has as disadvantage that it is not always clear which method is being called. Example:

```
class A:
    def m(self):
        print('m of A called')

class B(A):
    pass
```

```
class C(A):
    def m(self):
        print('m of C called')

class D(B,C):
    pass

x = D()
x.m()
```

The implementation has the following structure:



What is being printed? The answer is rather confusing, as it (also) depends on the version of Python you are using. In Python 3, '`m of C called`' is printed, while in Python 2 '`m of A called`' is printed. Python 3 has chosen to use C3 linearisation (or *Method Resolution Order*). Method Resolution Order (or MRO) defines the order in which super classes are used to inherit attributes/functions from. A full discussion of MRO (or C3 Linearisation) is out of scope of this reader, and interested readers are refered to Wikipedia, 2017 as a starting point (beware, interesting but complex stuff!)

The search order satisfies the following two demands:

- Child classes have preference over parent classes.
- When a class multiple inherits from other classes, the classes are searched in the order as specified in the parameter-list. That means that for `class D(A,B,C): ...` the search order is `D,A,B,C`.

In the example mentioned above, the search order is `D,B,C,A` and not `D,B,A,C`, because `C` is a child of `A`. The Method Resolution Order can be obtained by:

```
print([.e__name__ for e in D.mro()])
```

Output:

```
['D', 'B', 'C', 'A', 'object']
```

Exercise 3.6

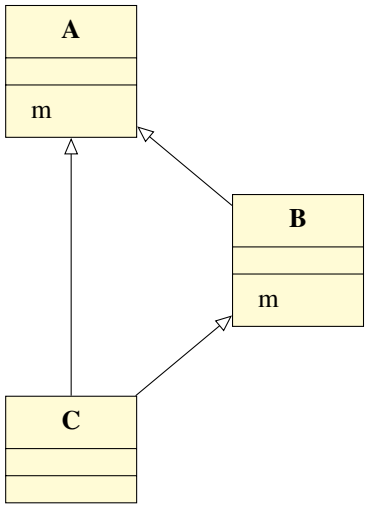
What is printed by the following code:

```
class A:
    def m(self):
                print('m of A called')

class B(A):
    pass

class C:
    def m(self):
                print('m of C called')

class D(B,C):
    pass

x = D()
x.m()
```

What is the class structure of the code above? Draw the UML diagram. ■

The demands mentioned above can lead to *inconsistent* class definitions. For example:

```
class A:
    def m(self):
        print('m of A called')

class B(A):
    def m(self):
        print('m of B called')

class C(A,B):
    pass
```

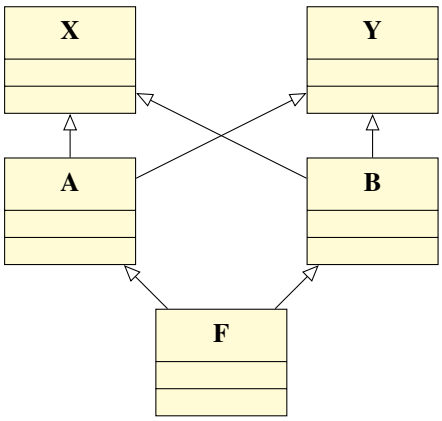


Because of the ordering in parameters in the definition of `C(A,B)`, `A` comes before `B`. But, because of the inheritance relation between `B` and `A`, `B` comes before `A`.

Python does not accept this, and will throw a compile-time error:

```
TypeError: Cannot create a consistent method resolution order (MRO) for
bases B, A
```

Another example:




```
class X():
    def who_am_i(self):
        print("I am a X")

class Y():
    def who_am_i(self):
        print("I am a Y")

class A(X,Y):
    def who_am_i(self):
        print("I am a A")

class B(Y,X):
    def who_am_i(self):
        print("I am a B")

class F(A,B):
    def who_am_i(self):
        print("I am a F")
```

The question now is whether class X takes preference over class Y or vice versa?

Exercise 3.7
Question: what is the search order when `B(Y,X)` is replaced by `B(X,Y)`? ■

3.3 Functions in Python

Functions can be called in different manners:

- with parameters on basis of their position;
- with keyword parameters;
- with a 'variable-length' list.

Combinations of these are also possible. For example:

```
def f(a,b,c):
        return 100*a + 10*b + c

print('f(1,2,3):      ', f(1,2,3))           # parameter position
print('f(b=6,a=9,c=3):', f(b=6, a=9, c=3))   # keyword parameters
print('f(*[1,0,2]):   ', f(*[1,0,2]))        # variable-length parameters
print('f(*(1,0,2)):   ', f(*(1,0,2))
print('f(*[5,6],2):   ', f(*[5,6],2))
print('f(4, c=8, b=6):', f(4, c=8, b=6))
print('f(2,3,c=2):    ', f(2,3,c=2))
print('f(*[5,6],c=9): ', f(*[5,6],c=9))
print('f(c=7,*[5,6]): ', f(c=9,*[5,6]))
```

output:

```
f(1,2,3):          123
f(b=6,a=9,c=3):    963
f(*[1,0,2]):       102
f(*(1,0,2)):       102
f(*[5,6],2):       562
f(4,c=8,b=6):      468
f(2,3,c=2):        232
f(*[5,6],c=9):     569
f(c=7,*[5,6]):     567
```

Specifics:

- Keyword parameters have to be behind position parameters;
- When using keyword parameters, the order of the parameters does not matter
- The length of the list or tuple when using a variable-length parameter has to match the function definition.

When declaring a function, a variable-length parameter can be given in the definition:

```
def arg_count(*args):
    return len(args)

print(arg_count(3,2,5))                  # 3
print(arg_count(*[1,2,3,4,5,6,7,8,9,0])) # 10
```

It is also possible to make the keyword parameters variable:

```
def echo(*args, **kwargs):
    print(args, kwargs)

echo(10, 20, a=30, b=40)                 # output: (10, 20) {'a': 30, 'b': 40}
```

Another example:

```
def f3(*args, **kwargs):
    return [e*e for e in args] + [kwargs[e]*kwargs[e] for e in sorted(kwargs)]

print(f3(3,2,5))                          # [9, 4, 25]
print(f3(*[9,8,3]))                       # [81, 64, 9]
print(f3(3,2,5,c=1,a=8))                  # [9, 4, 25, 64, 1]
print(f3(3,2,5,**{'b':9,'a':8,'c':10}))   # [9, 4, 25, 64, 81, 100]
```

Python, like many modern programming languages also allows for the definition of anonymous (or nameless) functions. As this concepts stems from functional programming, anonymous / *lambda* functions will be explained later in Section 4.3.

3.4 Introspection

Python offers possibilities to extract information about parts of a (run-time) program. We discuss the following types of introspection:

- Introspection of the 'top-level' domain;
- Introspection of imported modules;
- Introspection of Classes;
- Introspection of Functions.

3.4.1 Top-level domain

When Python is started, the predefined variable `__name__` has the value `__main__`. It is the name of the top-level domain, or global scope. This is often used to only start certain functionality of a Python file when Python was started with that file. The file then includes the following code:

```
if __name__ == '__main__':
        ...
        ...
```

The code after the if-statement is only executed when Python was started with this file.

A Python module declares the variables, functions and classes. By processing the module, these objects are placed in a dictionary. That dictionary can be consulted by two functions:

- `dir()` shows a list of names of declared variables and defined objects;
- `vars()` shows a dictionary of objects. It shows the name and value for each of these objects.

An example as illustration, the file `inspection_demo1.py` contains the following code:

```
if __name__ == '__main__':
    print(dir())

i = 100
```

```
if __name__ == '__main__':
    print()
    for e in dir():
        if not (e.startswith('__') and e.endswith('__')):
            print(e)
    print()

    for key, value in sorted(vars().items()):
        print(key, ':', value)
    print()

    print('i:', i)
    print('type(i):', type(i))
    print('id(i):', id(i))
```

Output:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__']

i

__builtins__: <module 'builtins' (built-in)>
__cached__: None
__doc__: None
__file__: E:\aa_ont_omg\python_ont_omg\alds_workspace\ATP\inspection_demo1.py
__loader__: <_frozen_importlib_external.SourceFileLoader object at 0x01F53C30>
__name__: __main__
__package__: None
__spec__: None
('e', 'i')
('i', 100)

i: 100
type(i): <class 'int'>
id(i): 495861056
```

There are a number of predefined variables, each starting and ending with __. Note that after the execution of the statement `for e in dir():` the variable `e` remains. This variable can be removed with the statement `del e`.

The `type`-function can be used to consult the corresponding class of an object. In CPython (the standard implementation of Python), one can request the memory address of an object with the `id`-function.

3.4.2 Imported modules

The following example imports the above created module:

```
import inspection_demo1
```

```
print(dir(inspection_demo1))
print()

for e in sorted(vars(inspection_demo1).items()):
    if e[0] != '__builtins__':
        print(e)
print()
```

Output:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'i']

('__cached__', 'C:\\ATP\\__pycache__\\inspection_demo1.cpython-35.pyc')
('__doc__', None)
('__file__', 'C:\\ATP\\inspection_demo1.py')
('__loader__', <_frozen_importlib_external.SourceFileLoader object at 0x014EF7F0>)
('__name__', 'inspection_demo1')
('__package__', '')
('__spec__', ModuleSpec(name='inspection_demo1',
    loader=<_frozen_importlib_external.SourceFileLoader object at 0x014EF7F0>,
    origin='C:\\ATP\\inspection_demo1.py'))
('i', 100)
```

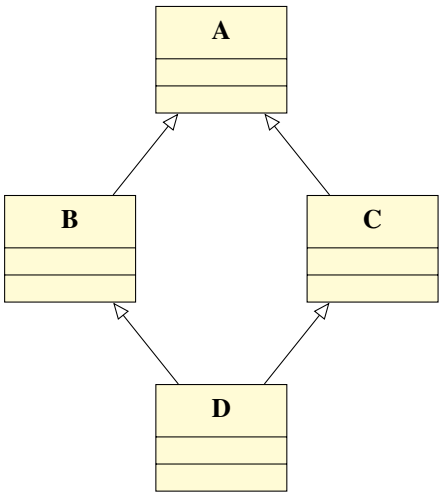
The print of `__builtins__` has been omitted on purpose, because the overview of built-in functions is too big to display here. The variable `__cached__` shows that the file `inspection_demo1.py` has been compiled to a `.pyc`-file. Those types of files contain Python-bytecode. Note that `__name__` does not contain `__main__` any more.

3.4.3 Classes

Lets go a bit deeper on reflection with respect to classes. Lets assume the following class definition:

```
class A:
    def m(self):
        print('m of A called')

class B(A):
    pass

class C(A):
    def m(self):
        print('m of C called')

class D(B,C):
    pass
```

With the following class diagram.



We show how certain aspects of classes can be examined, such as subclasses, bases, members, etc. For that, we extend the code as follows:

```
def getNames(a):
    return [e.__name__ for e in a]

class A:
    def m(self):
        print('m of A called')

print('A.__subclasses__():', getNames(A.__subclasses__()))

class B(A):
    pass

print('A.__subclasses__():', getNames(A.__subclasses__()))

class C(A):
    def m(self):
        print('m of C called')

class D(B,C):
    pass

print('A.__subclasses__():', getNames(A.__subclasses__()))
print('D.__bases__:', getNames(D.__bases__))
print('D.mro():', getNames(D..mro()))
print('issubclass(D,B):', issubclass(D,B))

d = D()
```

```
print('isinstance(d,D):', isinstance(d,D))
print('isinstance(d,B):', isinstance(d,B))
print('isinstance(d,C):', isinstance(d,C))
print('isinstance(d,A):', isinstance(d,A))
print('isinstance(d,object):', isinstance(d,object))

import inspect
print([e for e in inspect.getmembers(D) if \
    not (e[0].startswith('__') and e[0].endswith('__'))])
print([e for e in inspect.getmembers(d) if \
    not (e[0].startswith('__') and e[0].endswith('__'))])

print('inspect.ismethod(D.m):', inspect.ismethod(D.m))
print('inspect.isfunction(D.m):', inspect.isfunction(D.m))

print('inspect.ismethod(d.m):', inspect.ismethod(d.m))
print('inspect.isfunction(d.m):', inspect.isfunction(d.m))

print('inspect.isclass(D):', inspect.isclass(D))
```

The output of this code shows that:

- Python keeps a dynamic record of what the subclasses of a class are;
- Classes have a record of their bases and subclasses;
- Object are an instance of multiple classes;
- Given a class or object, one can see which elements are functions and which are methods.

```
A.__subclasses__(): []
A.__subclasses__(): ['B']
A.__subclasses__(): ['B', 'C']
D.__bases__: ['B', 'C']
D.mro(): ['D', 'B', 'C', 'A', 'object']
issubclass(D,B): True
isinstance(d,D): True
isinstance(d,B): True
isinstance(d,C): True
isinstance(d,A): True
isinstance(d,object): True
[('m', <function C.m at 0x01812270>)]
[('m', <bound method C.m of <__main__.D object at 0x01663C30>>)]
inspect.ismethod(D.m): False
inspect.isfunction(D.m): True
inspect.ismethod(d.m): True
inspect.isfunction(d.m): False
inspect.isclass(D): True
```

The next example shows how the **dir** and **vars** functions work on classes and

instances. Every instance keeps its own list and dictionary. The list contains all attributes and methods of the class and of the base classes. The dictionary contains the instance-attributes and -methods. These additions can change per instance.

```
def mylist(a):
    return [e for e in a if not (e.startswith('__') and e.endswith('__'))]

def mydir(d):
    return [e for e in d.items{} \
            if not (e[0].startswith('__') and e[0].endswith('__'))]

class B:
    B_attr = 1000     # class attribute

    def B_m(self):
        print('method m of B')

def D_m2(self):
    print('method m of D')

class D(B):
    D_attr = 2000     # class attribute

    def D_m(self):
        print('method m of D')

D.D_attr2 = 3000     # class attribute
D.D_m2 = D_m2        # method

d1 = D()
d2 = D()

print('mylist(dir(B)):', mylist(dir(B)))
print('mydir(vars(B)):', mydir(vars(B)))

print('mylist(dir(D)):', mylist(dir(D)))
print('mydir(vars(D)):', mydir(vars(D)))
print()

def m1(self):
    print('m1 call')

def m2(self):
    print('m2 call')

d1.c_attr1 = 100
```

```
d1.m1 = m1
d2.c_attr1 = 200
d2.m2 = m2

print('mylist(dir(d1)):', mylist(dir(d1)))
print('vars(d1):       ', vars(d1))
print('mylist(dir(d2)):', mylist(dir(d2)))
print('vars(d2):       ', vars(d2))
```

Output:

```
mylist(dir(B)): ['B_attr', 'B_m']
mydir(vars(B)): [('B_m', <function B.B_m at 0x007F22B8>), ('B_attr', 1000)]
mylist(dir(D)): ['B_attr', 'B_m', 'D_attr', 'D_attr2', 'D_m', 'D_m2']
mydir(vars(D)): [('D_attr', 2000), ('D_m2', <function D_m2 at 0x007F21E0>),
                 ('D_attr2', 3000), ('D_m', <function D.D_m at 0x007F2270>)]

mylist(dir(d1)): ['B_attr', 'B_m', 'D_attr', 'D_attr2', 'D_m', 'D_m2',
                  'c_attr1', 'm1']
vars(d1):        {'m1': <function m1 at 0x007F2228>, 'c_attr1': 100}
mylist(dir(d2)): ['B_attr', 'B_m', 'D_attr', 'D_attr2', 'D_m', 'D_m2',
                  'c_attr2', 'm2']
vars(d2):        {'c_attr': 200, 'm2': <function m2 at 0x007F2348>}
```

3.4.4 Functions

Now that we have looked at modules and classes, lets focus on the introspection of functions.

A function has attributes. For example:

```
def f():
    print('f-call')

print(dir(f))
print()
print(f.__class__)    # this is similar to print(type(f))
```

Output:

```
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__',
'__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']

<class 'function'>
```

It is also possible to add attributes to functions yourself. For example:

```
def g():
    g.ncalls += 1
    print('g-call')

g.ncalls = 0      # has to be done after g is defined, but before g is called

g()
g()
g()
print(g.ncalls)
print(vars(g))    # show the attributes of g
```

Output:

```
g-call
g-call
g-call
3
{'ncalls': 3}
```

More information about the attributes of a function can be retrieved by use of the `getmembers` function from the `inspect` module.

```
import inspect
a = inspect.getmembers(g)
for e in a:
    print(e)
```

It is also possible to request the *signature* of a function. For example:

```
def func(a,b=3,c=5):
    pass

import inspect
print(inspect.signature(func))    # output: (a, b=3, c=5)
```

It is even possible to inspect and analyse the function code by using a disassembler.

```
def sqr(x):
   return x*x

code = sqr.__code__
print(code)
print(code.co_varnames)
print(code.co_argcount)
print(code.co_code)

import dis    # disassembler
```

```
dis.dis(code)
```

Output:

```
<code object sqr at 0x00C58930, file
 "E:\python_ont_omg\reflection_demo4.py", line 11>
('x', )
1
b'|\x00\x00|\x00\x00\x14S'
 12           0 LOAD_FAST                0 (x)
              3 LOAD_FAST                0 (x)
              6 BINARY_MULTIPLY
              7 RETURN_VALUE
```

The following statements check whether a given variable `sqr` is in fact a function:

```
import types
print('isinstance(sqr,types.FunctionType):',
    isinstance(sqr, types.FunctionType))

import inspect
print('inspect.isfunction(sqr):', inspect.isfunction(sqr))
```

3.5 Decorators

Decorators in Python function a bit different than in other OO programming languages.

A decorator is any callable Python object that is used to modify a function, method or class definition. A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition. Python decorators were inspired in part by Java annotations, and have a similar syntax; the decorator syntax is pure syntactic sugar, using @ as the keyword.

Decorators are a form of metaprogramming; they enhance the action of the function or method they decorate.

Despite the name, Python decorators are not an implementation of the decorator pattern. The decorator pattern is a design pattern used in statically typed object-oriented programming languages to allow functionality to be added to objects at run time; Python decorators add functionality to functions and methods at definition time, and thus are a higher-level construct than decorator-pattern classes. The decorator pattern itself is trivially implementable in Python, and so is not usually considered as such.

Decorators in Python can be used as a form of *Aspect-Oriented Programming*. Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding additional behavior to existing code (an advice) without modifying the code itself, instead separately specifying which code is modified via a "pointcut" specification, such as "log all function calls when the function's name begins with 'set'". This allows behaviors that are not central to the business logic (such as logging) to be added to a program without cluttering the code, core to the functionality.

Next to pure functionalities, there are often additional demands, like:

- security
- logging
- serialisation
- dispatching
- delegating

Decorators allow the addition of these demands separately. A decorator is a function with a function as parameter and a function as result. By using decorators, code can be tidied up into separate uses or concerns (separation of concerns). For example, the following function returns an error when `b == 0`:

```
def divide(a, b):
    return a/b
```

Which can be solved as follows[3]:

```
def smart_divide(f):
    def inner(a, b):
        print('I am going to divide',a,'and',b)
        if b == 0:
            print('Whoops! cannot divide')
            return
        return f(a,b)
    return inner

divide = smart_divide(divide)

print(divide(3,4))
print(divide(3,0))
```

Output:

```
I am going to divide 3 and 4
0.75
I am going to divide 3 and 0
Whoops! cannot divide
None
```

The function `smart_divide` has function `f` as parameter and returns function `inner`. The code:

```
def divide(a,b):
    return a/b

divide = smart_divide(divide)
```

can be replaced by the following shorter variant:

[3]Note that the solution uses a sort of *nested function*, which is in fact a *closure*. Closure is another concept from functional programming, and is explained later in Section 4.3.1.

```
@smart_divide
def divide(a,b)
    return a/b
```

Lets consider another example. We might want to know how often a function is called, but we do not want to change the function code itself.

```
def count(f):
    def inner(a, b):
        inner.counter += 1
        return f(a, b)
    inner.counter = 0
    return inner

@count
def myadd(a,b):
    return a+b

print(myadd(1,2))                          # 3
print(myadd(3,4))                          # 7
print(myadd(5,6))                          # 11

print('myadd.counter = ', myadd.counter)  # myadd.counter = 3
```

The myadd function now has an attribute that we can use to show how often the function has been called. However, our count function only works for functions with two parameters. This can be generalised by using argument lists as discussed earlier. An example:

```
from functools import wraps

def count_V2(f):
    @wraps(f)
    def inner(*args, **kwargs):
        inner.counter += 1
        return f(*args, **kwargs)
    inner.counter = 0
    return inner

@count_v2
def mysum(*args):
    return sum(args)

print(mysum(1,2,3))                       # 6
print(mysum(4,5,6,7))                     # 22
print(mysum(8,9,10,11,12))                # 50

print('mysum.counter =', mysum.counter)  # mysum.counter = 3
```

```
print('mysum.__name__:', mysum.__name__) # mysum.__name__: mysum
```

The *@wraps* decorator ensures that function attributes remain with the original function.

Decorators can be combined:

```
def argc(f):
    @wraps(f)
    def inner(*args, **kwargs):
        print('argc:', len(args) + len(kwargs))
        return f(*args, **kwargs)
    return inner

@count_v2
@argc
def mysum(*args):
    return sum(args)

print(mysum(1,2,3,*[1000,2000])
print(mysum(1,2,3,4))

print('mysum.counter=', mysum.counter)
```

3.5.1 Memoization

In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. Memoization can be easily achieved in Python through the use of decorators.

The Fibonacci numbers can be generated recursively as follows:

```
def fib(n):
    return n if n<2 else fib(n-1) + fib(n-2)
```

The problem with this implementation is that every call to `fib` is calculated even when in fact it was already calculated in an earlier call. When trying to calculate 'large' Fibonacci numbers (> 40), it can be quite slow.

A solution is to use *memoization*, which means that intermediate results are stored for reuse.

```
cache = {}

def fib2(n):
    if n in cache:
        return cache[n]
    else:
        cache[n] = n if n < 2 else fib2(n-1) + fib2(n-2)
        return cache[n]
```

This can also be solved by realisation of an appropriate decorator:

```
def memoize(f):
    cache = {}
    def helper(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return helper

@memoize
def fib(n):
    return n if n<2 else fib(n-1) + fib(n-2)
```

3.5.2 Adding decorators at runtime

Combining the introspective and metaprogramming properties of Python introduced above, we can, for instance, change our existing code (at runtime) by adding decorators.

Let us consider, again, the `Clock` class as defined earlier (any version will do). For instance, `Clock` saved in `Clock.py`:

```
class Clock:
  def __init__(self, hour=0, minute=0, second=0):
    self.hour = hour
    self.minute = minute
    self.second = second

  def show(self):
    s = "time " + \
      "{:02d}:{:02d}:{:02d}".format(self.hour, self.minute, self.second)
    print(s)
```

We can now define a logging functionality in a separate file, e.g.:

```
import inspect

def log(func):
  @wraps(func)
  def wrapped(*args, **kwargs):
    try:
      print("Entering: [%s] with parameters %s" % (func.__name__, args))
      try:
        return func(*args, **kwargs)
      except Exception as e:
        print('Exception in %s : %s' % (func.__name__, e))
    finally:
      print("Exiting: [%s]" % func.__name__)
```

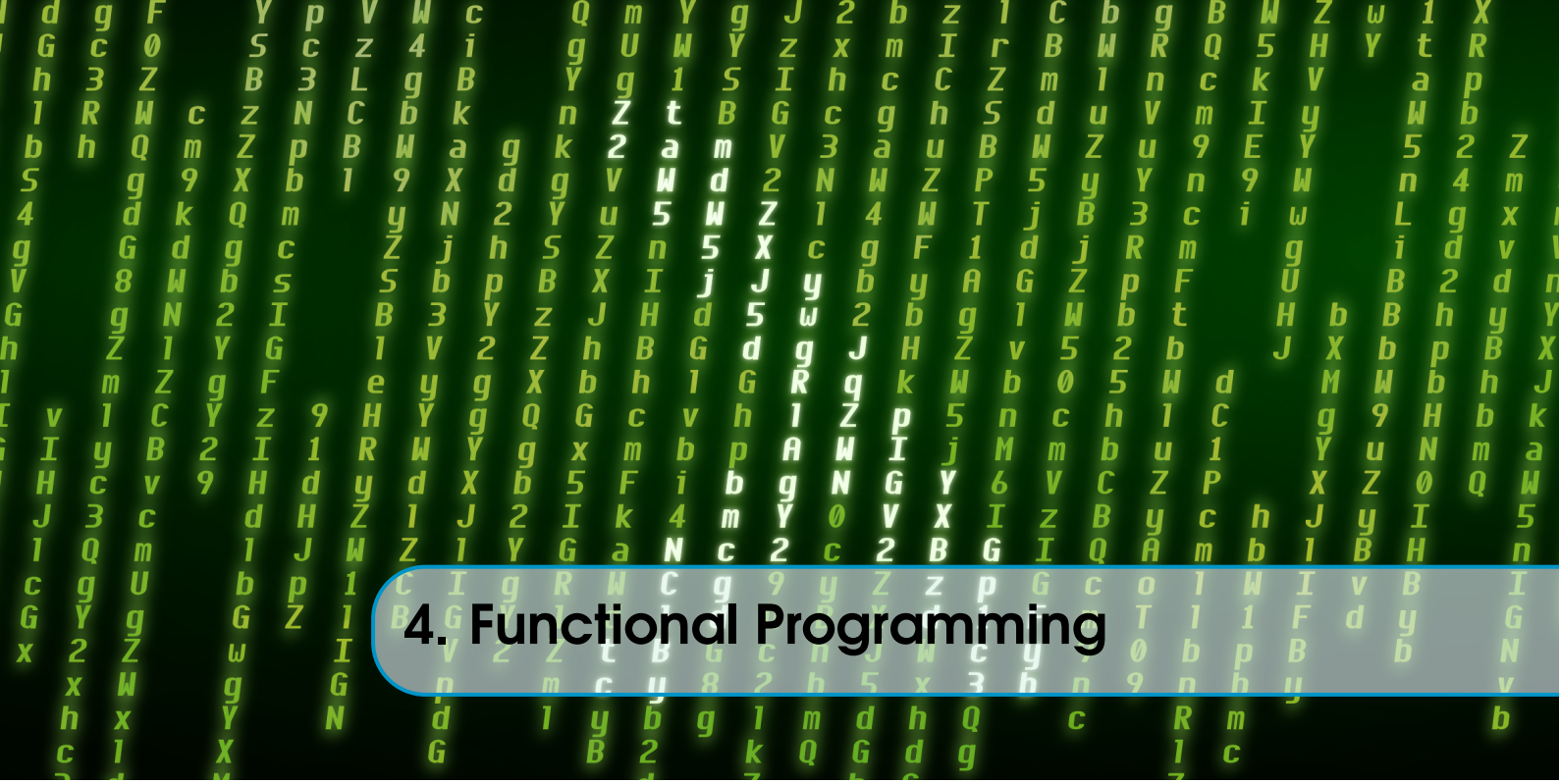
```
  return wrapped

def trace(cls):
  for name, m in inspect.getmembers(cls, inspect.isfunction):
    setattr(cls,name,log(m))
  return cls
```

To attach the trace functionality to our earlier defined `Clock`-class, we use the following:

```
from trace import * # get the definition of trace
                    # (assuming we stored the previous excerpt as trace.py)
import Clock        # get the Clock class definition

Clock = trace(Clock.Clock) # attach the 'trace' decorator to the Clock-class

c1 = Clock(10, 25, 30)
c1.show()
```

Output:

```
Entering: [__init__] with parameters
(<Clock.Clock object at 0x00000025D5123278>, 10, 25, 30)
Exiting: [__init__]
Entering: [show] with parameters
(<Clock.Clock object at 0x00000025D5123278>,)
time 10:25:30
Exiting: [show]
```



4. Functional Programming

This chapter will give an introduction to the concept of functional programming. It serves as a companion to the four lectures given on this subject — Table 4.1 provides an overview of the relation between the various sections and the lectures. You will notice this chapter does not contain any exercises; these are collected in a Jupyter Notebook available at this[1] location.

Table 4.1 Sections per lecture

Lecture 1	4.1, 4.2, 4.3
Lecture 2	4.4, 4.6
Lecture 3	4.7
Lecture 4	4.8

4.1 Preliminaries

Functional Programming (FP) is a programming paradigm: a specific way to consider fundamental concepts regarding control flow and computation whilst programming. Programming paradigms are generally defined by a set of concepts which are either discouraged or, in more extreme cases, even outright missing, as well as a set of concepts introduced or elevated to first-class status to replace them. The goal of programming paradigms is providing a coherent style of programming, making it easier to write correct code, and making it harder to inadvertently introduce bugs. One of the first paradigms, Procedural Programming (PP), introduced the procedure[2] to replace the `goto` statement, which at that time was a major source of "unintended features". In a similar vein, OOP introduces objects to encapsulate program structure and data, attempting to reduce the dependency on global variables present in PP.

[1] `https://github.com/aldewereld/practicum-atp`

[2] Also known as a subroutine or a function, not to be confused with the functions we are defining in this section. We stick to the term *procedure*, to clearly show the link to the PP paradigm.

Functional Programming uses the pure, first-class function to replace mutable state and side-effects.

4.1.1 Functions, procedures, expressions, oh my!

In FP, the concept of a function differs in meaning from the "function" as we know it in PP and OOP (we will refer to those as *procedure* and *method* from now on, respectively). A *pure function*, from this point on referred to as just a *function*, is defined as a block of code which uses its input to calculate the its output. That's all: no side-effects (IO other than parameters and return values), no mutated state (no variables are changed — `answer = 42` means that the answer is defined to be equal to 42, not that it is 42 right now but might be something else tomorrow), and no hidden voodoo. Compare this to languages such as C, where side-effects and mutable state are prevalent — even necessary to get anything done. A consequence of this is that every function must have both input and output: without input, a function is just a constant[3]. Consider on the other hand a function without output, and with no side effects: it wouldn't do anything. This also means that there is no distinction between functions as statements and expressions: everything has a return value, so everything is an expression.

4.1.2 First-class functions

In order for a language to allow for functional programming, support for functions as first-class citizens is a must. This means a function is considered a value, just as an `int` or a `boolean`. Functions can be assigned to variables[4], can be used as parameters for other functions, can be the return value of another function, and can be stored in data-structures such as lists or trees. Furthermore, a function does not need to have a label; just as the number 3 will always be the same number, regardless of whether it is assigned to a variable, the same should hold for functions. This concept is known as "anonymous functions" or "λ-functions" (pronounced: lambda function), and will be explained in Section 4.3.

4.1.3 Functional languages

We have now seen a number of features that define Functional Programming. As you may have noticed, some of the requirements are rather strict. Does this mean FP is only possible in very specific languages? And are all languages either purely functional, or not functional at all? The answer to both of these questions is no; FP should be considered first and foremost a style of programming, applicable to a diverse set of languages. That said, it is true that some languages are more fit to be used for FP than others; some languages, such as Haskell, promote FP to such an extent that it is hard to avoid FP while programming in them. Others support a variety of paradigms, including FP or some features thereof — these are known as *multi-paradigm* languages. Python, the language used in this course, is an example of the

[3] Certain purely functional programming languages often consider constants as just nullary function, reinforcing the "everything is a function" mantra.

[4] Even though variables are generally immutable in functional programming, they are still referred to as variables.

latter category, supporting imperative, functional, object-oriented and aspect-oriented programming, among others. In general, any language supporting first-class functions can be used for FP[5]. Fortunately, most languages have some support for first-class functions: modern languages often support closures or blocks, OO languages allow you to embed a function within an object, and even in low-level languages such as C this behaviour can be simulated using function pointers[6].

As mentioned earlier, some languages exists at the other end of this spectrum, actively promoting the use of FP. These are called *Functional Languages*, and are distinguished by the fact that it becomes easier to write functional code than it is to write imperative[7] code. In this chapter one such language, the aforementioned Haskell, is sometimes used for code examples. You are not required to be able to understand or to write such code; these examples exist solely to clarify concepts in a language more suited syntactically to convey the underlying ideas. Python, being a multi-paradigm language, does not always provide the same transparent syntax for these concepts. In these cases, code in both languages is presented side-by-side, where you are invited to regard the Haskell code as a kind of functional "pseudo-code". Do not worry when these examples raise more questions than they answer: feel free to ask for clarification or ignore these examples. If, on the other hand, your interest has been piqued, you can use these examples as a starting point for exploring a purely functional language.

4.2 Recursion

With the use of mutable state forbidden, a lot of familiar concepts become unusable. Loops are a prime example of this: without a mutable iterator, the **for**-loop is not possible. The same holds for the **while**-loop, as the conditions will never change and the loop will run forever. For this reason, FP makes extensive use of recursion: a function calling itself with slightly different parameters, repeating the process until the parameters match what is called the *base case*. Examples include an integer reaching 0 or a list becoming shorter each recursive step until only an empty list remains. Below, a number of examples are provided in imperative style, relying on loops, as well as their functional counterparts utilising recursion.

```
# Calculate the $n$—th Fibonacci number (The sequence 0, 1, 1, 2, 3, ...;
# every new number equals the sum of the two previous numbers).

def fib_loop(n):
    fibs = [0,1]
    for i in range(2,n):
        fibs.append(fibs[i—1] + fibs[i—2]
    return fibs[—1]

def fib_rec(n):
```

[5] Even without this, it is possible to apply some FP methods, but it will likely be very hard to do even simple things in a purely functional way.

[6] *Do not* try this at home

[7] Imperative means either procedural or object-oriented.

```
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fib_rec(n-1) + fib_rec(n-2)

# Calculate the greatest common divisor of two numbers, the largest
# number by which both operands can be divided.

def gcd_loop(a,b):
    while b != 0:
        a,b = b, a%b
    return a

def gcd_rec(a,b):
    if b == 0:
        return a
    else:
        gcd_rec(b, a%b)
```

4.2.1 Tail recursion

In most languages, recursion comes at a cost: as each call adds a new stack frame to the call stack, deeply nested recursive functions typically require more resources than a loop performing the same actions would. Functional languages typically have special optimisations in place to counteract this, as their compilers were designed with recursion explicitly in mind. Multi-paradigm languages typically have fewer of these optimisations, penalising the use of recursion over loops. A form of optimisation most languages do have, however, is tail call optimisation. A tail call is a function call at the tail of a function, i.e. the last expression is a function call. In this case, the return value of the function call is the return value of its parent function. Compilers are generally able to figure out when a function call, specifically a recursive call, occurs in this position, and discard the previous frame as it will be garbage-collected once the function call finishes, anyway. Python does not use tail call optimisation by default, but implementations do exist[8].

When working with tail call optimisation, the goal is to make sure that there is nothing left to do in your functions after the recursive call. Take, for example, this non-tail recursive factorial functions:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

[8] https://github.com/baruchel/tco

Figure 4.1 Tail recursion in Python, the wrong way

Even though the recursive call is on the end, it is not in tail position: after the value of the recursive call has been calculated, the value gets multiplied by `n` before the function returns. This way, the compiler cannot discard the frame yet. Even though the multiplication is simple, it is not nothing, so there's something to keep track of. By introducing an accumulator, we can make a better version of factorial that does have tail recursion:

```
def factorial(n):
    def factorial1(n, accumulator)
        if n == 0:
            return accumulator
        else:
            return factorial1(n − 1, n ∗ accumulator)
    return factorial1(n,1)
```

Now, instead of building up a stack of multiplications, we pass along the intermediate result to each recursive call. When the recursion reaches its base case, the accumulator is simply returned. This way, the multiplications have all been performed before the recursive call instead of after it. This means that the recursive call is now in tail position, as the only thing left to do after obtaining the result of the recursive call is returning it. The difference is shown in Figure 4.2.

4.2.2 Mutual recursion

Mutual recursion is a special case of recursion, where instead of calling itself, a function calls another function, which in turn calls the original function. This type of recursion has many applications, but the most common example is introducing some pseudo-state. Consider, for example, a function parsing code and marked-up block

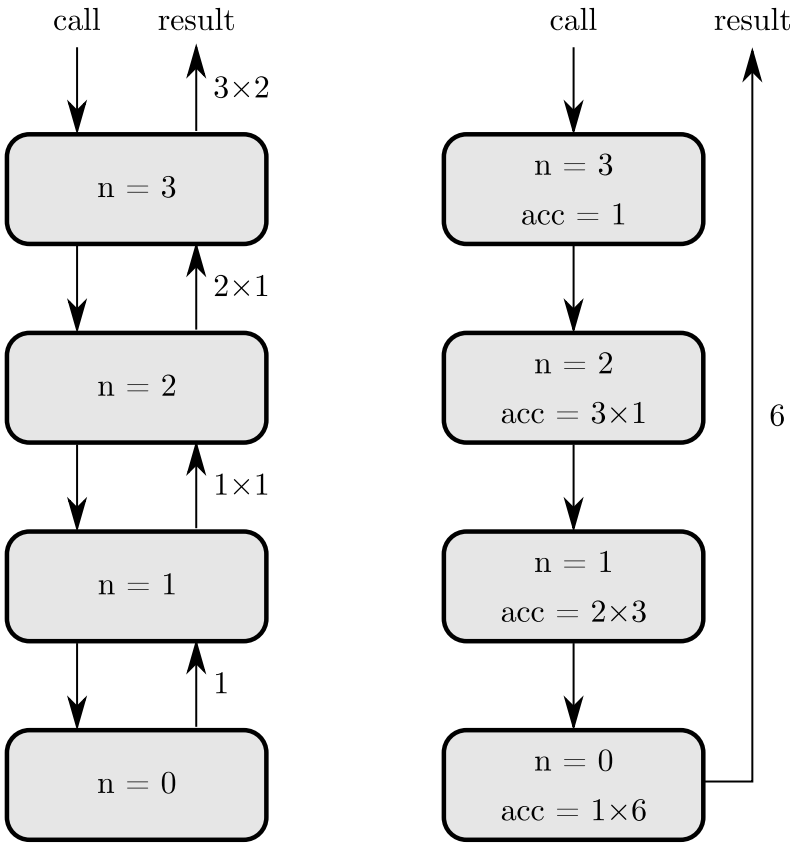


Figure 4.2 Normal and tail recursive factorial.

comments, as shown in Figure 4.3. This could be implemented using mutual recursion by writing a function to recursively parse the code, and calling another function to parse the mark-up once the start of a comment is encountered. The mark-up function will continue to recursively call itself until the end the comment is encountered, at which point it will call the original function.

For more information on recursion, please refer to Section 4.2.

4.3 Anonymous functions

In most imperative programming languages, the default method of creating functions is by specifying a name, parameters and function body. In FP, this need not be the case. As functions are first-class citizens, and can freely be assigned to variables, they are not bound to this name in the same way as subroutines in, for example, C or Java. Functions can be created anonymously using the `lambda` syntax. The keyword `lambda()` acts as the binder, associating supplied parameters to their variables `x`:

```
def double_square(x): # Not anonymous
    return (x ** 2) * 2

anonymous_double_square = lambda x: (x ** 2) * 2 # Anonymous / Lambda
```

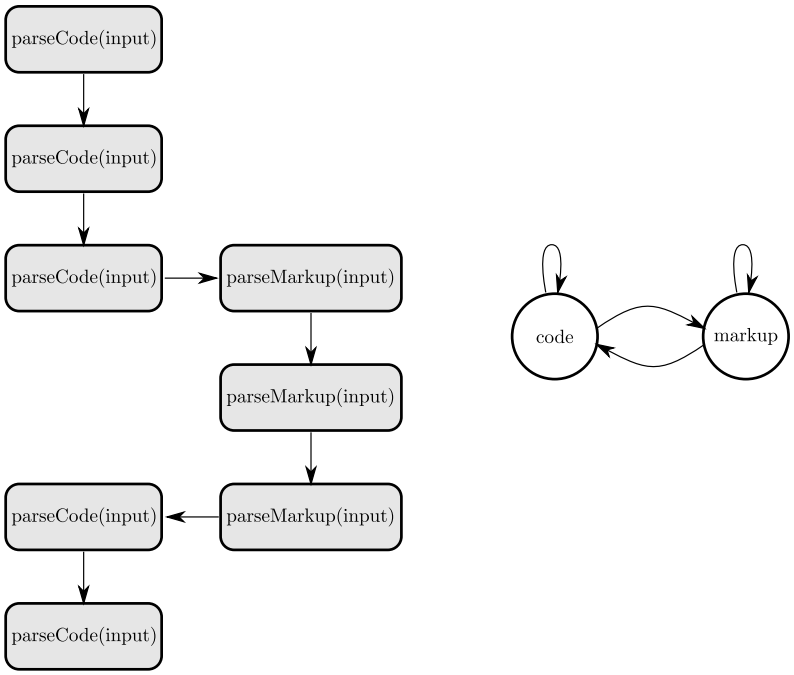


Figure 4.3 Mutual Recursion and State

```
double_square(2) # 8
anonymous_double_square(2) # Also 8
```

Here, the definition of the function is separated from the assignment of the function to a variable. In Python, these two methods behave exactly the same. The real use of anonymous functions, of course, is not directly assigning functions to variables: anonymous functions can be passed to other functions, or stored into data-types without polluting the namespace with function-names that will only be used once. Consider the example below, where the anonymous function allows for more concise code:

```
def pass_me_a_function(f):
    f(1) + f(2) - f(3)

# Execution using anonymous functions:
pass_me_a_function(lambda x: x + x*x + x**x)

# Execution without using anonymous functions:
def my_function(x):
    x + x*x + x**x

pass_me_a_function(my_function)
```

In general, it is considered good practice to use anonymous functions whenever you would otherwise define a small function for one-time use.

Aside 4.1 — On the origin of Lambda.
The concept of anonymous functions has been borrowed from the λ-calculus[a], a computational model developed in the 1930s by Alonzo Church. The λ-calculus forms the basis of FP, in the same way as the Turing Machine and the Von Neumann Architecture were of critical importance for the imperative programming paradigm. Both the λ-calculus and the Turing Machine arose as formal models for expressing computation in mathematical logic to find an answer to Hilbert's Entscheidungsproblem. Both models have been proven to be equivalent, in that they share the same computational power — both are universal models, believed to be able to compute anything that is computable; this property is known as Turing Completeness. Whereas the Turing Machine relies on an infinite amount of tape containing state and a machine to manipulate that state according to certain rules, the λ-calculus defines just the anonymous function and two reduction rules on how to evaluate them. Together, Church and Turing created the foundation of modern computer science. Coincidentally, Turing was as PhD student of Church, and the two have published joint papers.

[a] Hence the name "lambda-function".

4.3.1 Closure

In functional programming, functions can be defined within other functions. Python also has support for such nested functions, as seen in Chapter 3. Furthermore, nested functions can use variables bound in the scope containing the nested functions:

```
def outer_function(argument):
    def inner_function():
        print(argument)
    return inner_function()

outer_function(42) # Prints 42
```

Here, the inner function references `argument`, even though it is a free (not bound) variable. In this case, Python is smart enough to look for a bound variable matching the name in the containing function. In this manner, a variable can be bound an arbitrary number of nested functions away from where it is referenced: Python will first look

within the function itself, then in the containing function, etc. until a scope is found where the variable is bound. The first scope found will be used, so the innermost bound variable will be used:

```
def this():
    var = 0
    def is():
        var = 1
        def a():
            def contrived():
                var = 42
                def example():
                    print(var) # Will print 42
            print(var) # Will print 1
```

In Python, a free variable bound in a shallower scope is read-only[9] by default, unless the `nonlocal` keyword is used:

```
def without_nonlocal():
    var = 1
    def inner():
        var = 2
        print(var) # 2
    inner()
    print(var) # 1

def with_nonlocal():
    var = 1
    def inner():
        nonlocal var # Borrow writeable
        var = 2
        print(var) # 2
    inner()
    print(var) # 2
```

Now, let's look at an example only slightly different from the first:

```
def outer_function(argument):
    def inner_function():
        print(argument)
    return inner_function

give_me_the_answer = outer_function(42)
give_me_the_answer() # 42
```

Here, the outer function call has been completed, but `argument` has been preserved attached to the inner function, preventing a nasty error. This concept is called a *closure*,

[9] Of course, as we're programming functionally here, we don't want to change a variable either way; nevertheless, it's good to know how Python behaves in this regard.

as it closes around a portion of the environment, embedding it within the function for later use (when the original environment may or may not be in scope any more, or even exist). Closures are often used in functional programming to provide a form of data hiding and to avoid global variables in non-purely functional languages such as Python. Furthermore, closures are used in Python to implement decorators as seen in Section 3.5.

4.4 Typing in Python

Later in this chapter, we will take a look at type theory, which is the basis of the type systems used in most functional languages. Types are not a fundamental part of functional programming, but most functional languages (with LISP dialects being the most prominent exception) support a powerful type system. To get used to thinking in types, we will start annotating code examples with types from this point forward. Though Python is primarily known for being a "duck-typed"[10] language (data is typed not by definition but by how it is acted upon), it nevertheless supports a type annotation system. Furthermore, Python's type system supports some of the advanced features present in the type systems of most functional languages. For now, we will focus only on the notation of type annotations in Python. In Section 4.7, a more expansive introduction to basic type theory will be provided. Consider the type-annotated functions given below[11]:

```
# foo :: Int → String
def foo(i : int) → str:
    return str(i)

# bar :: String → Char
def bar(s : str) → str:
    return s[0]

# foobar :: Int → Char
def foobar(i : int) → str:
    return bar(foo(i))
```

In the above examples, the function `foo` converts an integer into a string and `bar` converts a string into a character. These two functions can be combined, but only in one order: applying `foo` and then `bar` composes into a single function taking an integer to a character. Applying these functions the other way around is impossible because of the types: `bar` will return a character, whereas `foo` requires an integer value. Note that the Haskell type `Char` corresponds to the Python type **`str`**, as in Python characters are just one-letter strings, whereas in Haskell strings are lists of characters.

[10] Duck-typing is derived from typing by means of a "duck test", which is a form of abductive reasoning: e.g. "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

[11] The comments above each functions are the type annotations as they would be used in Haskell, for comparison and added clarity.

The next example, `baz`, shows a parametrised type: `List(str)` types a list of strings.

```
from typing import List

# baz :: Int → String → [String]
def baz(i : int, s : str) → List(str):
    ...
```

Finally, `quux` introduces a type-variable: a place holder referring to any type. This notation is used when the type does not matter. Imagine `quux` as returning the length of the list; in this case, it does not matter whether the list contains numbers, characters or anything else. Note that in order for a type variable to be used, it must first be declared.

```
A = TypeVar('A')

# quux :: [a] → Int
def quux(list_of_something : List[A]) → int:
    ...
```

When using type variables, keep in mind that a single variable can stand for only a single type per function. Consider for example the following functions:

```
# function1 :: a → a → a
def function1(par1 : A, par2 : A) → A:
    ...

# function1 :: a → b → a
def function2(par1 : A, par2 : B) → A:
    ...
```

In `function1`, both parameters must be of the same type, and the return type will be the same as well. The opposite does not hold: in `function2`, `A` and `B` may be different, but may also be the same.

4.5 Function composition

In our example above, the function `foobar` was made by calling first `foo` on a value, followed by `bar`. As can be expected in a paradigm built on nothing but functions, this is a fairly common occurrence. This chaining of functions is called function-composition, and this term nicely captures the intended way to think about this concept. Instead of reasoning about a value `i` getting pushed through two functions in the body of another function, we say that `foobar` is the composition of `bar` and `foo` (note the order: foo is applied first but mentioned last; this is because of the difference between the direction in which function application and (English) text are read — take a look back at the definition of `foobar` to see for yourself). Mathematically, as well as in

Haskell, instead of writing `h i = g(f i)`[12], we prefer to write `h i = (g ∘ f) i`, or even better, `h = g ∘ f`. The little circle is called the composition operator, and combines two functions into one. In the first example, we still show the parameter to make this intermediate step clear, but the second example really shows how it's supposed to be done. In FP, we often reason about functions as being the topic of interest instead of the parameters to the functions.

4.6 Higher-order functions

Higher-order functions is one of the most important aspects of functional programming. One of the main benefits of first-class functions is that it allows for extra abstraction. As it becomes easy to pass functions around, function structure and specific behaviour can be separated into isolated functions.

4.6.1 Motivating example

Consider, for example, a pair of functions for calculating the sum and product of a list of numbers:

```
# sum :: [Int] → Int
def sum(list : List[int]) → int:
    if len(list) == 1:
        return list[0]
    else:
        head, *tail = list
        return head + sum(tail)

# product :: [Int] → Int
def product(list : List[int]) → int:
    if len(list) == 1:
        return list[0]
    else:
        head, *tail = list
        return head * product(tail)
```

In this code, the $*$ operator is used to split a list into its first element (bound to head) and the rest (bound to tail). The $*$ operator is called "splat", and is generally used to unpack lists — useful, for example, for providing a list of arguments to a function without manually extracting each element. Here, the splat operator is used to bind what remains of the list after the first element is removed.

The only difference between **sum** and `product` is a single line of code: The **sum**-function uses the addition operator and recursively calls **sum**, whereas the `product` uses multiplication and `product`. Using higher-order functions, we can abstract a more general **fold** or **reduce** function, and supply the specific behaviour (in this case the operator used) as a parameter:

[12] Haskell's `h i` translates to Python's `h(i)`; `f x y` would be `f(x,y)`, and so on. For more details, see notation cheat sheet on Sharepoint.

```
# fold :: (a → a → a) → [a] → a
def fold(f : Callable[[A, A], A], list : List[A]) → A:
    if len(list) == 1:
        return list[0]
    else:
        head, *tail = list
        return f(head, sum(tail))

# Below, the first argument has been fulfilled with (Int → Int → Int),
#   by + and * respectively so the rest of the type becomes [Int] → Int

# sum :: [Int] → Int
def sum(list : List[A]) → A:
    return fold(operator.add)

# product :: [Int] → Int
def product(list : List[A]) → A:
    return fold(operator.mul)
```

Aside 4.2 — Operators as parameters.
While Python supports passing functions as parameters, operators unfortunately do not receive the same privilege. To pass + to a fold, for example, you have a few options. The first is the module `operator`, which defines functions such as `operator.add(a,b)`.
We could of course do the same thing ourselves, defining a simple wrapper function. In cases such as this, it is considered good style to define small auxiliary functions such as these as anonymous functions (provided no predefined function such as `operator.add` is available).

Notice that we introduced a new type in defining `fold`: the function type. Function types are written using `Callable`, with two arguments: a list of input types, and a single output type. So in this example, the type `Callable[[A,A],A]` reads as "a function, taking two parameters of type `A`, returning an `A`". Note the use of square brackets when working with type annotations in Python.

Aside 4.3 — Arrays vs. List.
As you may have noticed, a lot of examples in this chapter are concerned with lists, where in other contexts you might have expected arrays. This is not coincidental, but rather a general feature of functional programming. The reason for this is that lists are recursively defined, which nicely matches the fact that functional programming replaces iteration by recursion. When working with lists, the common pattern is to split the list into the head and tail. The head is the current element being considered, the first element of the list. The tail is simply the rest of the list (which can be empty when the original list has a length of 1). An operation is then performed on the head, after which the function is called recursively on the tail.

In contrast, arrays nicely match the imperative way of looking at computation. We can define a mutable variable to iterate over the array, which is made of subsequent memory positions.

```
data List a = a : List a | []
-- List a  is normally written  [a]

int[3] array = {1,2,3};
```

4.6.2 Map

The most common example of a higher order function is **map**, which is used to apply a function to each element of a list. Consider our function `foobar` — we can apply this recursively to a list of ints:

```
numbers = [1,2,3,4,5,6,7,8,9,10]

# foobar_loop :: [Int] → [String]
def foobar_loop(list : List[int]) → List[str]:
  if len(list) == 0:
    return []
  else:
    head, *tail = list
    return [foobar(head)] + foobar_loop(tail)
```

A lot of this code would be exactly the same if we were to apply a second function to a list of values. Let's define, for example, a `tally_loop` function:

```
# tally :: Int → String
def tally(number : int) → str:
    fives, ones = divmod(number, 5)
    return fives * "#" + ones * "|"

# tally_loop :: [Int] → [String]
def tally_loop(list : List[int]) → List[str]:
  if len(list) == 0:
    return []
  else:
    head, *tail = list
    return [tally(head)] + tally_loop(tail)
```

Now that we've seen two examples, we can compare both and define a new function **map** containing only the shared parts, parametrising the differences:

```
# map :: (Int → String) → [Int] → [String]
def map(f : Callable[[int], str], list : List[int]) → List[str]:
    if len(list) == 0:
        return []
    else:
```

```
        head, *tail = list
        return [f(head)] + map(f,tail)
```

We can further generalise this by relaxing the types: it does not matter for the looping behaviour what the types are, only that the input of the supplied function matches the contents of the list. Furthermore, we known that the output of the supplied function will match the contents of the output list. We can annotate this using two type variables:

```
A = TypeVar('A')
B = TypeVar('B')

# map :: (a → b) → [a] → [b]
def map(f : Callable[[A], B], list : List[A]) → List[B]:
    if len(list) == 0:
        return []
    else:
        head, *tail = list
        return [f(head)] + map(f,tail)
```

The same function in Haskell is shown below for comparison. Note the _, which is an anonymous parameter: it tells us that even though there is a parameter, it will not be used. Therefore, it is not bound to any variable.

```
map :: (a → b) → [a] → [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

The **map** function we have just defined is also present in the main Python libraries. It is one of functional programming's "breakout functions", which has since been adopted in a wide variety of imperative programming languages due to its convenience. The **for** ... **in** loop is one of the descendants from this very same idea.

4.6.3 Fold, revisited

Now that we've got a basic level of higher-order-function-fu, it's time to look at our motivating example again: **fold**. Though our definition of fold looks nice and tidy, we can yet make this function more general and more abstract. In doing so, we discover that **fold** is not even a single function, but comes in a variety of flavours.

Consider what would happen if we were to call **fold** with the $-$ operator. In contrast to $+$ and $\times$, $-$ is not associative: in general, $(a-b)-c \neq a-(b-c)$, whereas $(a+b)+c = a+(b+c)$. In other words, the order of applying the operations matters. We can split up our **fold** into a left fold **foldl** and a right fold **foldr**:

```
A = TypeVar('A')

# foldl :: (a → a → a) → a → [a] → a
def foldl(f : Callable[[A, A], A], list : List[A]) → A
    if len(list) == 1:
        return list[0]
```

```
    else:
        head, *tail = list
        return f(fold(f, tail), head)

# foldr :: (a → a → a) → a → [a] → a
def foldr(f : Callable[[A, A], A], list : List[A]) → A
    if len(list) == 1:
        return list[0]
    else:
        head, *tail = list
        return f(head, fold(f, tail))
```

There is one more thing left to do: we now expect all the types to be the same (`A`), but with a small tweak, we can accommodate an even wider range of uses. First, we add an extra parameter `base` of type `b` to both our variants, acting as a kind of accumulator as we've seen when discussing tail recursion. We then change the type of the function argument from $a \to a \to a$ to $b \to a \to b$ and $a \to b \to b$ for the left and right fold respectively, and change the result type to b as well. Our new functions are now typed **foldl** : $(b \to a \to b) \to b \to [a] \to b$ and **foldr** : $(a \to b \to b) \to b \to [a] \to b$. Finally, we update our function bodies to make use of the new `base` parameter. We can now see why we needed to add the `base` parameter: it will serve as our starting point when applying the operator on the list. Without it, we would begin applying the function `f` to our first two elements of the list (for **foldl**). But as our function now has type $b \to a \to b$, and the list contains elements of only type a, this would not work. We must provide a starting point of type b, which will be repeatedly combined with items from the list, resulting in another value of type b that we can use for the next iteration. The same holds for **foldr**, only this time time we work from the right: our function of type $a \to b \to b$ expects its right operand to be of type b, which is provided by the `base` parameter. After the first iteration, the result (also of type b) can be used in its place.

```
A = TypeVar('A')
B = TypeVar('B')

# foldl :: (b → a → b) → b → [a] → b
def foldl(f : Callable[[B, A], B], base : B, list : List[A]) → B
    if len(list) == 0: # Was: 1. Well, not anymore!
        return base
    else:
        head, *tail = list
        return f(fold(f, tail), head)

# foldr :: (a → b → b) → b → [a] → b
def foldr(f : Callable[[A, B], B], base : B, list : List[A]) → B
    if len(list) == 0:
        return base
    else:
```

```
    head, *tail = list
    return f(head, fold(f, tail))
```

But how exactly does this help us? Consider as a motivating example, a list of 3×3 matrices. As you know, a matrix can be seen as a linear transformation on a $\mathbb{R}^3$ vector. We can multiply our first matrix and a vector, multiply the second matrix and the result, and so on until we have applied each transformation in order. As a matrix and a vector would have different types, we could not have performed this operation with the less general **fold**s we defined before[13].

Concluding, we now have four different variations of our once simple **fold**. The first two versions, without the base parameter, we will call **foldl1** and **foldr1** for the left and right versions respectively. Our total arsenal of **fold**s now looks like this:

$$\begin{aligned}
\textbf{foldl} &: (b \to a \to b) \to b \to [a] \to b \\
\textbf{foldl1} &: (a \to a \to a) \to [a] \to a \\
\textbf{foldr} &: (a \to b \to b) \to b \to [a] \to b \\
\textbf{foldr1} &: (a \to a \to a) \to [a] \to a
\end{aligned}$$

All these folds correspond to different evaluation trees as well, as shown in Figure 4.4.

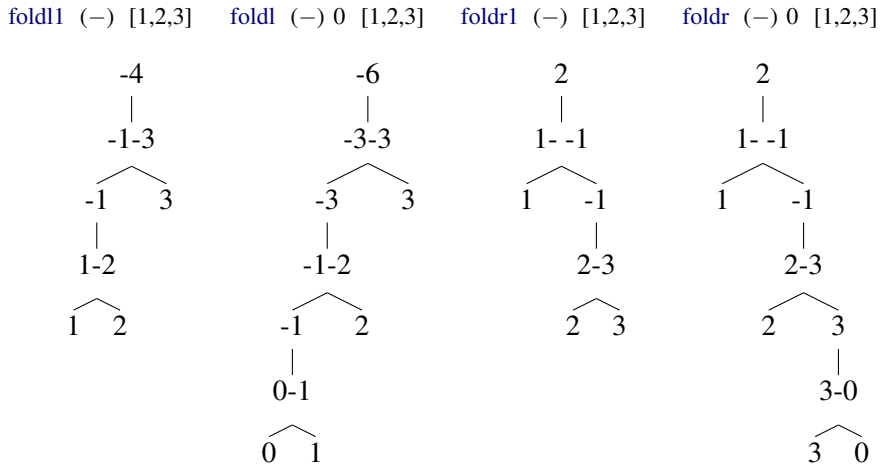


Figure 4.4 Evaluation trees of various folds.

In Python, **foldl1** is available as **reduce** in the `functools` module. By passing an optional `initialiser` parameter to **reduce**, it will behave as **foldl** (without the `1`).

foldr and **foldr1** can be expressed in terms of **foldl**, which you are asked to do in the exercises.

[13] Of course, we could have used a fold to combine all matrices into a single transformation and multiplied that with our vector, but what would be the fun in that?

Aside 4.4 — MapReduce.
The two higher order functions **map** and **fold** can be combined into a single `mapReduce` function, first mapping a set of functions on large amounts of data, and then reducing the data to generate a summary. This model allows for parallellised data-processing, as each chunk of data is self-contained and can be processed without knowledge of the entire dataset. Only in the final reduction-phase is data combined again. This concept forms the basis for the *MapReduce* programming model prevalent in data-processing tasks today (e.g. Apache Hadoop), as we will see in the Applied Artificial Intelligence course.

4.6.4 ZipWith

We have seen two families of higher-order functions working on lists. There is, however, no reason for such a function to be limited to a single list: enter **zipWith**[14]. This function is used when two lists need to be combined in a point-wise manner, like for example in vector-addition.

```
zipWith (+) [1,2,3] [4,5,6] ⇒ [5,7,9]
```

```
A = TypeVar('A')
B = TypeVar('B')
C = TypeVar('C')

# zipWith :: (a → b → c) → [a] → [b] → [c]
def zipWith(f : Callable[[A, B], C],
            xs : List[A],
            ys : List[B])
            → List[C]:
    if len(xs) == 0 or len(ys) == 0:
        return [] # Stop when either list is empty
    else:
        x, *xrest = xs # Split head x of list xs
        y, *yrest = ys # Split head y of list ys
        return [f(x,y)] + zipWith(f, xrest, yrest) # Recursion:
          # apply f to x and y, then concatenate to f applied to rest
```

The function **zip** is a special case of **zipWith**, where the function used to combine the two lists is the tuple constructor. In other words, **zip** combines two lists into a lists of tuples, zipping them together like a zipper would.

We can define **zipWith** using **zip** and vice-versa. Zipping two lists together and then applying a map to the resulting list emulates **zipWith**'s behaviour. In order to do this, however, there is one small obstacle to overcome: **zipWith** expects a function to have two argument, whilst mapping over a zipped list will provide tuples. We therefore need to translate our function (of type $A \to B \to C$) to work on tuples $((A,B) \to C)$, which luckily for us is not only possible, but even rather common. This concept is

[14] **zipWith** is sometimes confusingly referred to as just **zip**; we will consistently use **zipWith**, as **zip** refers to another function.

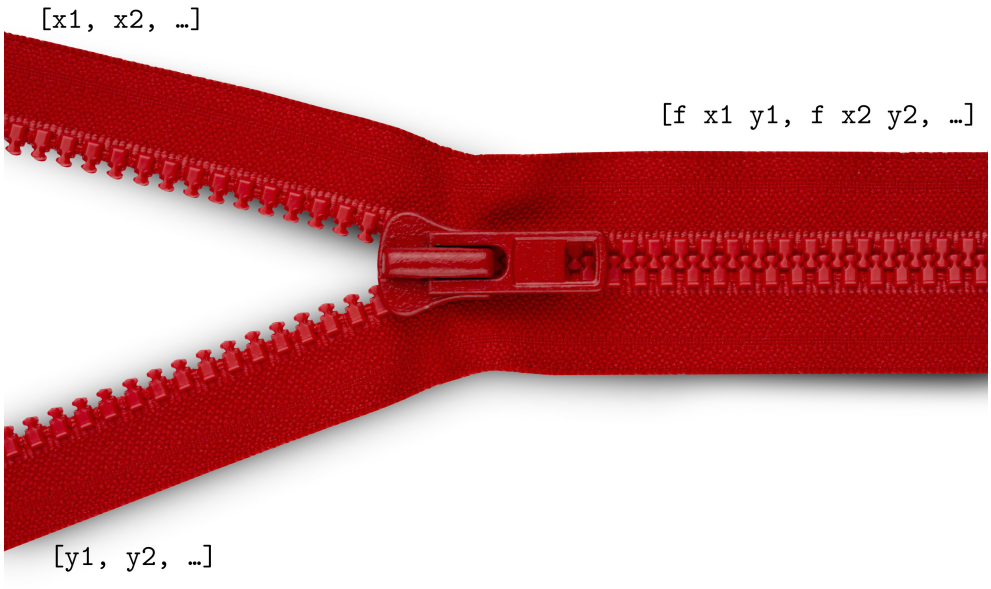


Figure 4.5 zipWith and zip act as a zipper.

called *currying*, and will be expanded upon in Section 4.7.1. For now, we only need to convert one way, using the uncurry function given below:

```
# uncurry :: (a → b → c) → ((a, b) → c)
def uncurry(f: Callable[[A,B], C]) → Callable[[Tuple[A,B]], C]:
    return lambda xy: f(xy[0],xy[1])

# uncurry transforms a function to accept a tuple
# instead of two arguments, for example:
uncurry(operator.mul)((6,7)) # 42

# alternativeZipWith :: (a → b → c) → [a] → [b] → [c]
def alternative_zipWith(f : Callable[[A,B], C],
                        xs : List[A],
                        ys : List[B])
                        → List[C]:
    return map(uncurry(f), list(zip(xs,ys)))
```

4.7 Type theory

Though not an integral part of the functional programming paradigm, most modern FP languages sport a simple yet exceptionally advanced type system. Whereas types in low-level languages such as C help in preventing some common types of errors, they often tend to get in the way; rather than fixing the issue, the most common response is to just add casts everywhere, defeating the purpose. Because of this, languages without a strict type system enjoy great popularity and type correctness is seen as a

chore.

Most functional languages, including Haskell, adopted a type system based on the mathematical field of *type theory* (Thompson, 1991). To a lesser extent, this model is also available in Python's type annotations.

4.7.1 Function types and currying

When playing around with **`zip`** and **`zipWith`** in Section 4.6.4, we scratched the issue on the concept of Currying. The general idea is that functions always have a single input and a single output, or in terms of types are of the form $A \to B$. This is great, as it makes it easy to compose functions: you only need to make sure the output of the first matches the input of the second. Given $f : A \to B$ (read: the function f takes an argument of type A and produces a value of type B) and $g : B \to C$, we can define a new function $h(x) = g(f(x)) : A \to C$. This composition is more commonly noted as $h(x) = (g \circ f)(x)$ or simply $h = g \circ f$.

That is all dandy, but what about a function with two arguments? Surely you need two operands for the + operator? Well, yeah, but even this can be interpreted as a single-argument (or "unary", if you want to sound formal) function. And there's two ways to go about this:

Firstly, we can interpret a function having two arguments (a binary function) as having a single argument in the form of a tuple. So the function `add` applied to `x` and `y` becomes `add((x,y)) : (Int, Int) → Int`. In mathematical notation, which is often used in functional languages, the outer parentheses are often omitted (Python `fac(n)` would be `fac n` in Haskell). This leaves only `add(x,y)` for our original example, which looks a lot like you would use the function in an imperative function. Neat.

There is, however, another way: `add x y` (notice we omit the parentheses, following the "functional convention") can be interpreted as having the type `Int → (Int → Int)`: a function that, when given an integer argument, will produce a new function — one that will transform an integer to another integer. Using this convention, `add x y` really stands for `(add x) y`, which is how a language such as Haskell would interpret it. The parentheses can safely be omitted, as precedence is defined in such a way that `f a b` is `(f a) b`, just with less typing. The same goes for the types: `Int → (Int → Int)` can unambiguously be written `Int → Int → Int`, which explains the funky Haskell notation you might have been wondering about. Also note that the result of only providing one argument to a binary function is perfectly valid and stable. The result can be assigned to a new function name (`succ = add 1`) or given as an argument to a higher-order function (`map (add 1) [1, 2, 3, 4, 5]`)[15].

It is easy to see that one can translate one interpretation to the other, and vice versa. Going from a tuple-representation to a function-result-interpretation is called Currying, whereas the reverse is called Uncurrying:

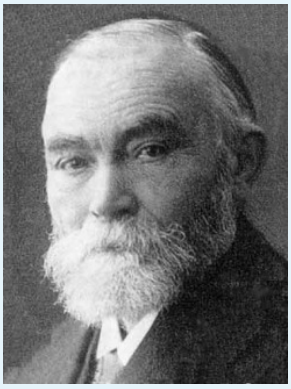
```
curry :: ((a, b) → c) → (a → (b → c))
uncurry :: (a → (b → c)) → ((a, b) → c)
```

[15] As Haskell is a bit less squeamish about the whole operators vs. functions thing, we can also use the + operator between parentheses, e.g. `map (+1) [1, 2, 3, 4, 5]`. Conversely, a function can be used as an infix operator by putting it between back ticks, e.g. `3 `mod` 2` for the modulo function.

Theory

The equivalence of both representations is also found in logic, which is closely linked to functional Type Theory. For the tuple, we often use the $a \times b$ notation for the type and (a, b) for the values. The curry function then corresponds to the logical formula $(A \times B) \to C = A \to (B \to C)$. The $\times$ symbol is read as the logical *and*. We can therefore interpret the equality as, for example, "if I get a good nights sleep (A) and ($\times$) I study hard (B), then ($\to$) I will pass the test (C)" meaning the same as "If I get a good nights sleep (A), then ($\to$) I only need to study (B) to ($\to$) pass the test (C)". The same, of course, applies the other way around.

Aside 4.5 — Schönfinkeling.
Despite what the name suggests, the concept of Currying was *not* developed by Haskell Curry, but rather by Moses Schönfinkel and based on the work of Gottlob Frege. Therefore, we should really be calling the process *Schönfinkeling* or even *Freging*.


Gottlob Frege


Moses Schönfinkel


Haskell Curry

4.7.2 Algebraic data types

Typed functional languages generally use a mathematics-inspired model for data types known as Algebraic Data Types (ADTs). ADTs are defined recursively as either being:

- a simple data type such as `Int` or `Bool`,
- an enumeration (e.g. `data Colour = R | G | B`),
- the product of two data types or
- the sum of two data types.

This mirrors the mathematical concept of an Algebra (hence the name) which is a set of some sort, combined with two operators (generally mapped to addition and multiplication). Members of the set can be freely multiplied and/or added, with the result always being an element of the set. ADTs form a sort of pseudo-algebra, as the definition above applies to them. There are additional rules, however, for truly being considered an algebra, most of which are not satisfied or simply difficult to interpret.

Product types

In Section 4.7.1, we have already spotted our first product type: the tuple. Taking the product of two types (a and b) results in a new type, containing both (a and b).

The reason this is called a product, is that the number of possible values inhabiting the product type ($\#(A \times B)$) is equal to the product of the possible values of the two separate types ($\#(A) \times \#(B)$). We can easily see an example by enumerating the possible combinations of two relatively small types, as shown in the first part of Figure 4.6. Another example would be a Person type combining an age, a name and a gender (represented as an enumerative type): this would be representable as Person = Int × String × Gender. Product types correspond to the logical *and* and the *conjunction* of two sets, in that both operands are required to be present.

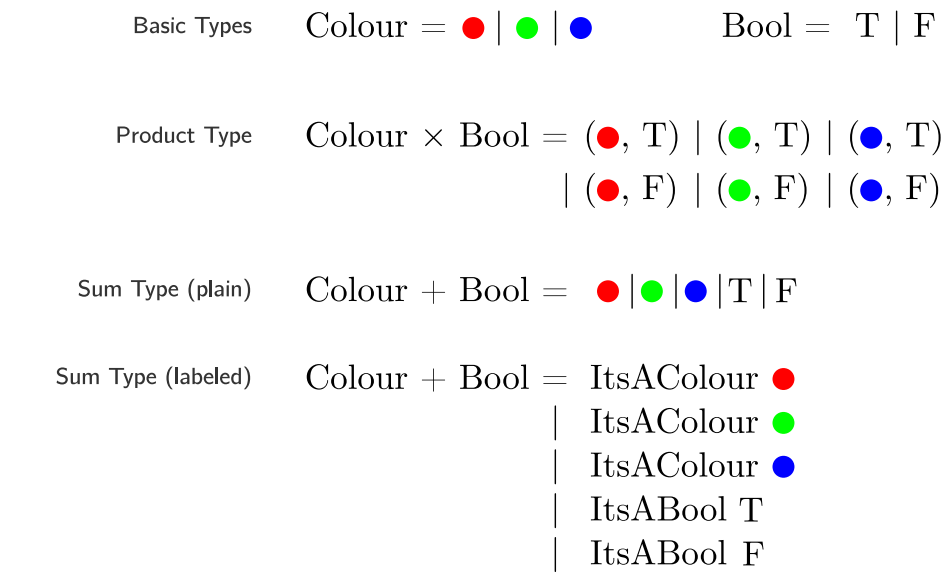


Figure 4.6 Visual example of a product type

Product types are generally realised using tuples, though any type with a constructor requiring two (or more) type arguments could qualify.

Sum types

Sum types differ from product types in that a sum combination does not require (or even allow) both types to be present. Whereas products combine two types into a single type, sum allow a choice of either type. If the product type is related to the logical *and* and the set-theoretic *conjunction*, the sum type is its dual corresponding to the logical *or* and the set-theoretic *disjunction*, offering choice. If we were to combine colours and booleans as per our previous example, the resulting set would have a size of 5 (or, in more general terms, the size of a sum type $\#(A+B)$ must be equal to $(\#(A)+\#(B))$, the sum of the constituent types).

An example using the same Colour and Bool is shown visually in the second part of Figure 4.6. Notice the two different versions of the sum type: the plain version corresponds to the way sum types are handled in Python, where its naturally lax typing allows for the omission of constructors. Haskell, on the other hand, uses the labelled version, utilising the syntax shown in the code fragment above. The labels are used to ensure type-safety when extracting the values from the sum type: by pattern-matching, different blocks of code can be distinguished to handle the different types contained within the complex type. A simple example of using labelled sum

types and pattern-matching in this way is shown below:

```
data ColourOrBool = ItsAColour Colour
                  | ItsABool Bool

doSomethingWithColourOrBool :: ColourOrBool → String
doSomethingWithColourOrBool (ItsAColour c) = "My cat has " ++ show c ++ "fur."
doSomethingWithColourOrBool (ItsABool b)   = if b
                                               then "My cat is potty trained."
                                               else "My cat must remain outside."
```

Sum types are often used in functional error handling, as they represent possibilities. Two sum types often encountered are called *Maybe*, which signifies an item is either present, or not, and *Either*, which signifies the type can be inhabited by values from one type or the other.

The maybe type is represented in Python with `Optional[X]`, which is shorthand for `Union[X, None]`. As the *None* type has only one possible inhabitant, the resulting type is one larger than `X`. This type can be used when a computation may fail, such as getting an item from a database or the internet. This is the most lightweight option, as no information about the failure can be attached. Consider a trivial example defining `multiplicativeInverse`, that will turn a number x into $\frac{1}{x}$:

```
def multiplicative_inverse(denominator: float) → float
    return (1 / denominator)
```

This function promises to turn any float into another float, but what happens when we ask it to give the multiplicative inverse of `0`? The result would be a `ZeroDivisionError`, despite the type signature saying it will return a float. Instead of throwing around and catching exceptions, in functional programming we try to code possible exceptions into the type, for example by returning and `Optional[float]`:

```
def better_multiplicative_inverse(denominator: float) → Optional[float]:
    if denominator == 0:
        return None
    else:
        return (1/denominator)
```

For more complex examples, where clarification about what went wrong might need to be attached, we can use the `Union[String,X]` type. If the computation succeeds, the result of type `X` will be returned. If something goes wrong, the `String` can be used to contain information about the nature of the exception.

Recursive types

Using only two simple connectives, we can now define any type we want. There is, however, one further part to the puzzle: recursion at the type level. We have seen instances of this before, but so far we haven't really dwelled on it. The prime example of a recursive data type is the list we have been using in most of our examples. Remember its definition:

```
data List a = [] | a : List a
```

Basically, a list is either the empty list, or some element of the correct type added to the front of another list. Recursive data types are not limited to lists, however: we can define trees, by using multiple recursive references — a tree not only points to the next element, but to two... or even more. Another example is the Peano representation of the natural numbers:

```
data Int = Zero | Next Int

0 :: Int
0 = Zero
1 :: Int
1 = Next Zero
2 :: Int
2 = Next (Next Zero)
```

Natural numbers are recursively defined as either being zero, or one larger than another natural number. Some experimental functional programming languages even use this representation internally (though of course the compiler converts everything to regular integers).

4.8 Functional Reactive Programming

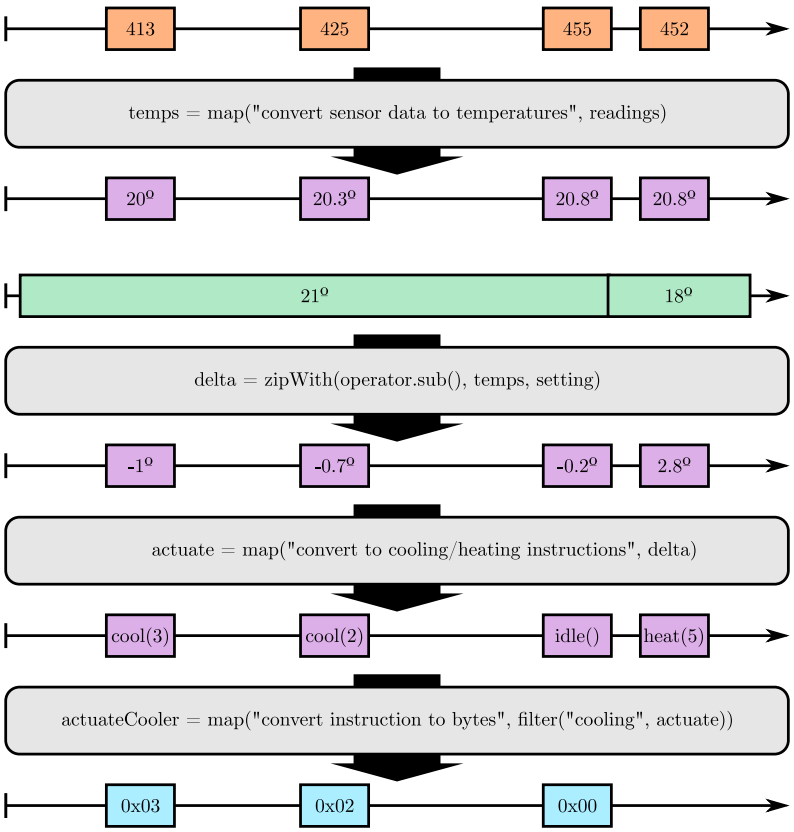


Figure 4.7 Chain of FRP functions to simulate behaviour in an air conditioner

Due to the limitations placed upon mutable state in FP, you might think the paradigm to be badly equipped to deal with interactive and time-based applications. Though

this is generally the case, the Functional Reactive Programming (FRP) paradigm has arisen to facilitate this domain. The idea is that of explicitly modelling time as function parameters. Functions are interpreted as continuous mappings between input (including time) and output. Both input and output are generally interpreted as being streams of data; streams being represented as (or synonymous with) infinite lists.

FRP is particularly useful for working with asynchronous data streams, as shown for example in Figure 4.7. Here, we model a function in an air conditioner. We have two input streams: the raw readings of a temperature sensor (orange), and the desired temperature in degrees Celsius as specified by the user (green). Purple streams represent intermediate results, and the light blue stream is the final result which is sent to an actuator. All streams are considered to be infinite, although of course they are not available before the system is turned on for the first time, nor after the systems life has ended. The grey blocks represent stream functions, and the entire process works from top to bottom. The first step in this example is converting from raw sensor data into temperatures in degrees Celsius by using a **`map`** function. Note that we have a few discrete readings (assume the sensor to take a new reading as soon as the previous reading has been read from the bus) which are not necessarily nicely spaced. As far as the system is concerned, the data is continuous; until the 425 reading becomes available, the stream is filled with values of 413. In the second step, we subtract the desired temperatures from the converted temperatures using a **`zipWith`**. The temperature is set by the user via a potentiometer, so there is a continuous stream of data here. The resulting stream is then passed through another **`map`** which converts the temperature differences into a stream of actions to be performed by the system. In the final step, the actions specific to the cooler are filtered out and converted to raw byte code to be sent to the actuator.

Aside 4.6 — Infinite lists.
In contrast to most imperative languages, functional languages generally use lazy evaluation (as opposed to strict evaluation). The idea here is that instead of computing values when the function calls are encountered, they are deferred until the value would actually be used. One of the advantages of this is support for infinite lists. As values are only computed when they are actually needed, and each next item in a list is a separate computation, infinite lists can be created provided their last item is never needed (which would require infinite time). You can safely use items from the head of the list, as the rest of the list is only kept as a potential computation.

Though Python is not lazy by default, generators can be used to achieve similar effects, most notably the definition of infinite lists without immediately evaluating the entire thing. For example, to get an infinite list of fibonnaci numbers as a generator, one could do the following (using a while loop — *yuck!* — because there is no parameter to recurse on):

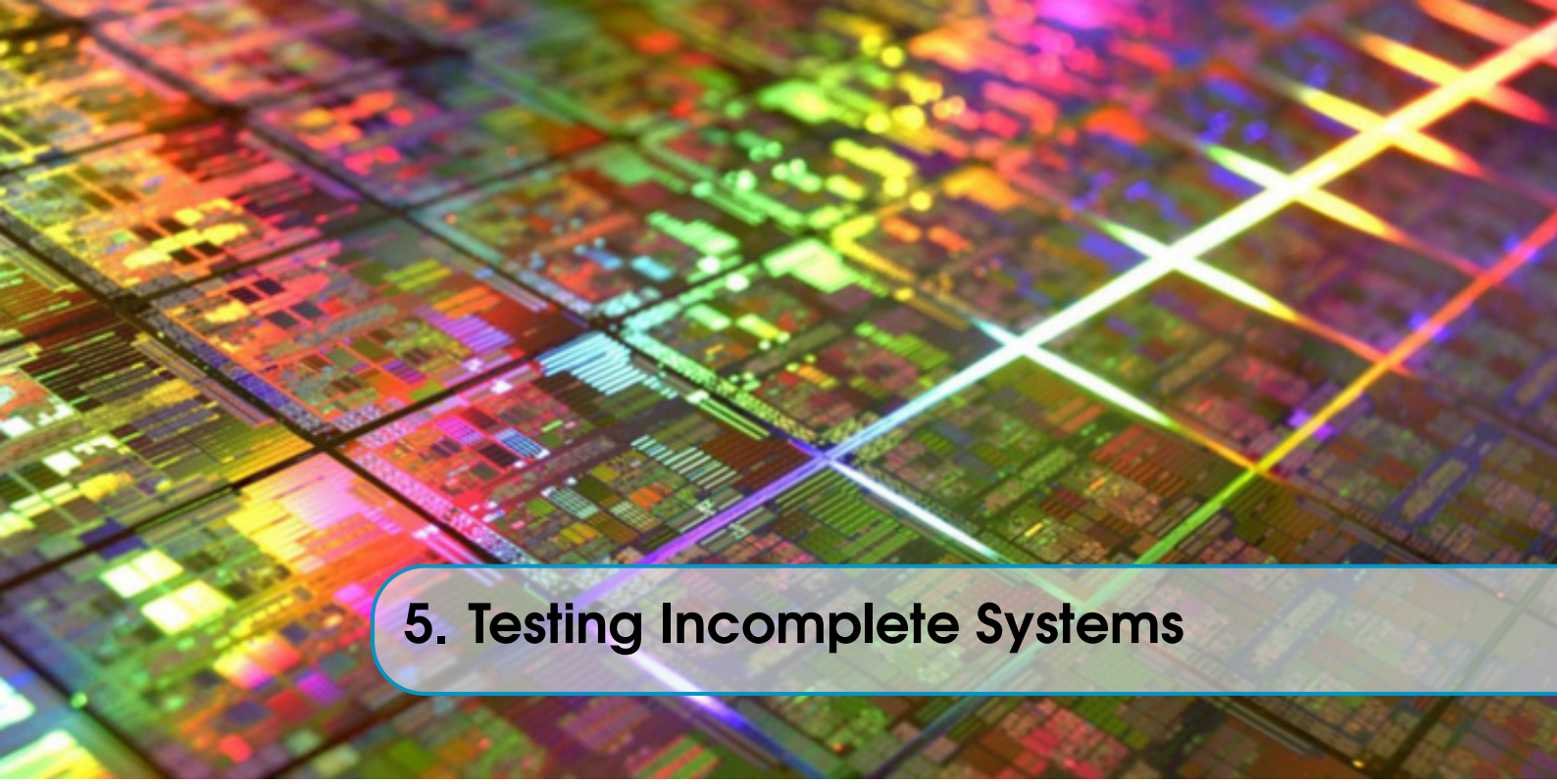
```
def fib_gen():
    yield 0
    a, b = 0, 1
```

```
        while True:
            yield b
            a,b = b,a+b
```

This will return 0 on its first run, executing the function until the `yield` statement. When called again, it will continue where it left of and enter the **`while`** loop. This will run until it encounters another `yield`, suspending until the next call.

4.8.1 FRP and IO

In order to support interaction, we can turn to Interactive FRP. An interactive FRP program generally consists of an input-process-output loop acting as a buffer between the purely functional world and the impure external world. The state of the world (or at least, the relevant, modelled parts of it) is determined (sense) passed to a pure function, which determines how the world should be influenced (actuate), queues to relevant actions for the outside world, and repeats. This form of FRP is often referred to as just being FRP, but is in reality an addition to the concept. The reason for this is that FRP happens to be a particularly good way to implement interaction in purely functional languages, so a lot of people interested in FRP are really just interested in Interactive FRP.



5. Testing Incomplete Systems

Often, in the early stages of the development (and test) process, much of the hardware needed is not yet available to testers – they need simulators or prototypes of the hardware components. In the later stages of the test process these must be replaced by upgraded prototypes, or even the real components. Section 5.1 describes the different *test environments* required at different stages in the test process.

Tools can make life a lot easier for testers. The range of *test tools* is extensive, offering support for a broad range of test activities. Section 5.2 provides an overview of the various tools that can be applied in the different phases of the test process.

Successful automation of test execution requires more than just a powerful tool. The dream of "the tool that will automate your tests for you" was shattered long ago. Section 5.3 explains how to achieve successful *test automation*, describing the technical issues.

Vocabulary 5.1 — Test levels and infrastructure. From *Glossary of terms used in software testing* (BS7925-1)

- **Driver**: A skeletal or purpose-specific implementation of a software module used to develop or test a component that is called by the driver.
- **Emulator**: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.
- **HiL (hardware-in-the-loop)**: A test level where real hardware is used and tested in a simulated environment.
- **MiL (model-in-the-loop)**: A test level where the simulation model of the system is tested dynamically in a simulated environment.
- **Plant**: The environment that interacts with an embedded system.

This chapter is partly based on (Broekman & Notenboom, 2003).

- **Rapid prototyping**: A test level where a simulated embedded system is tested while connected to the real environment.
- **Simulation**: A representation of selected behavioural characteristics of one physically or abstract system by another system.
- **Simulator**: A device, computer program, or system used during software verification that behaves or operates like a given system when provided with a set of controlled inputs.
- **SiL (software-in-the-loop)**: A test level where the real software is used and tested in a simulated environment with experimental hardware.
- **Stub**: A skeletal or purpose-specific implementation of a software module used to develop or test a component that calls or is otherwise dependent on it.
- **Test automation**: The use of software to control the execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and test reporting functions.

5.1 Embedded testing infrastructure

The development of a system containing embedded software involves many test activities at different stages in the development process, each requiring specific test facilities. This section discusses what kind of test environments related to those test activities and goals are required.

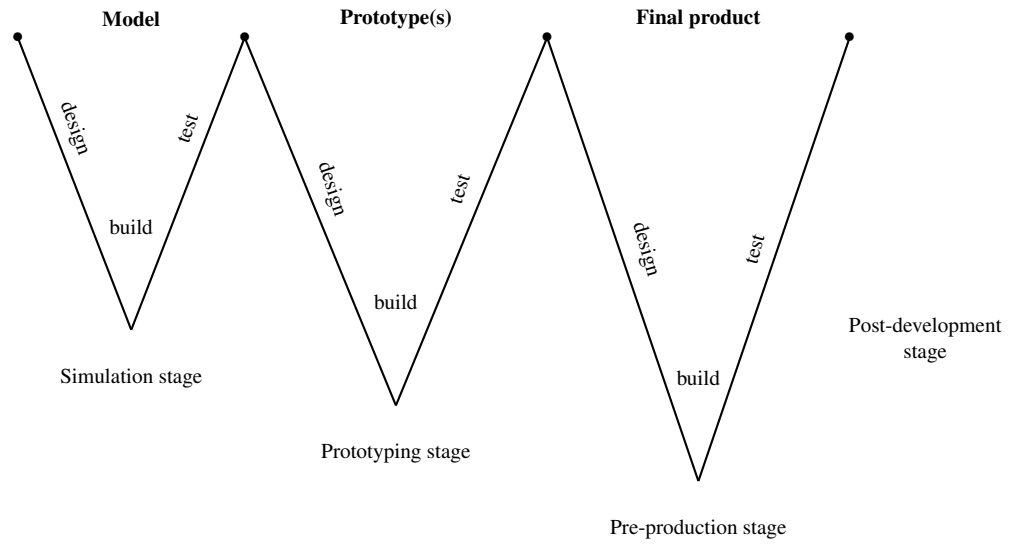


Figure 5.1 Relationship between the development stages and the multiple V-model.

This section applies to all projects where embedded systems are developed in stages: starting with simulated parts (*simulation stage*) and then replacing them one by one by the real thing (*prototyping stage*) until finally the real system works in its real environment (*pre-production stage*). After development, a final stage can be distinguished in which the manufacturing process is tested and monitored (*post-development stage*). These stages can be related to the multiple-V-lifecycle (see section

2.2.1) as shown in Figure 5.1. The detailed descriptions of test environments in this section are structured according to the multiple V-model but they are not restricted to projects that apply this model.

An example of a testing process divided into tests as shown in Figure 5.2 can be mapped onto the development stages as mentioned above as follows. The early models developed to simulate the system are tested in "model tests" (MT) and "model-in-the-loop" tests (MiL) – this corresponds to the simulation stage. In "rapid prototyping" (RP) an experimental model with high performance capacity and plenty of resources (for example 32-bit floating point processing) is tried out in a real-life environment to check if the system can fulfil its purpose – this is also part of the simulation stage. In "software-in-the-loop" testing (SiL) the real software including all resource restrictions (for example, 16-bit or 8-bit integer processing) is tested in a simulated environment or with experimental hardware. In "hardware-in-the-loop" testing (HiL) the real hardware is used and tested in a simulated environment – both "SiL" and " HiL" are part of the prototyping stage. In "system test" (ST) the real system is tested in its real environment – this corresponds to the pre-production stage.

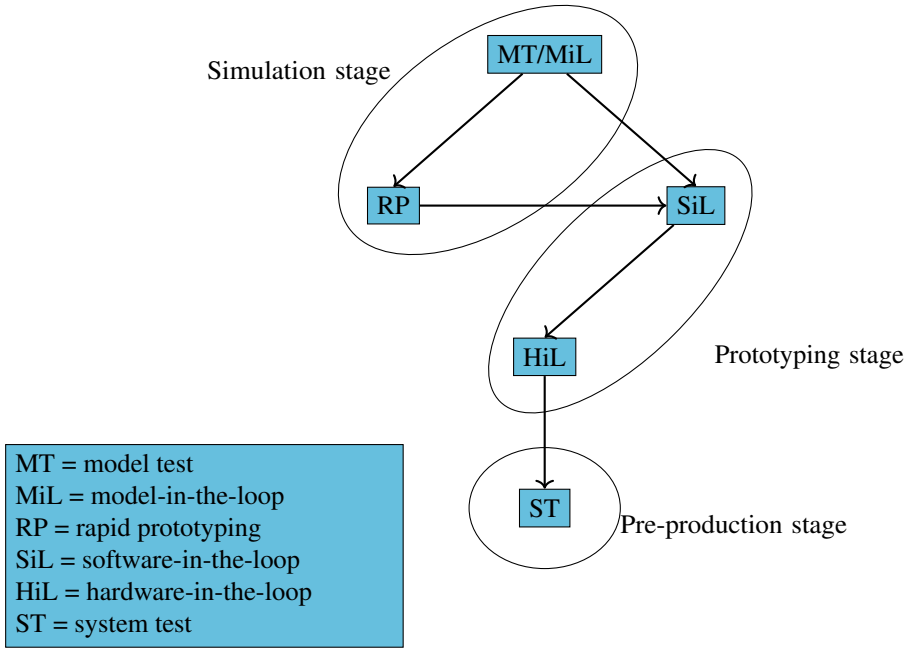


Figure 5.2 A test process showing the gradual transitions from simulated to real.

This section focusses on the test facilities required to enable the *execution* of tests on the embedded system. In addition to this, test facilities may be used to support the *testing* process. Here it will be necessary to review and analyse earlier test results and compare these with current results. Sometimes the customer may demand that detailed information is submitted on how, when, where, and with which configuration the test results where obtained. Thus, it is essential to keep accurate logs and to maintain strict hardware and software configuration control over the prototype units. If the development (and test) process is long and complicated, it is advisable to maintain a database to store all the test data. This should contain all relevant data such as date/time, location, serial number of the prototype under test, identification of

hardware and software configuration, calibration data for data recording equipment, test run number, data recording numbers, identification of test plan, identification of test report, etc.

5.1.1 First stage: simulation

In model-based development, after verification of the requirements and conceptual design, a simulation model is built. The developer of an embedded system uses the executable simulation models to support the initial design, to prove the concept, and to develop and verify detailed requirements for the next development steps. This stage is also described as "model testing" and "model-in-the-loop". The goals of testing at this stage are:

- proof of concept;
- design optimisation.

Executing a simulated model and testing its behaviour requires a specific test environment or a test bed. Dedicated software tools are needed to build and run the simulated model, to generate signals, and analyse responses. This section explains different ways of testing the simulation models and the required test environments.

Testing in the simulation stage consists, in general, of the following steps.

1. *One-way simulation.* The model of the embedded system is tested in isolation. One at a time, input is fed into the simulated system and the resulting output is analysed. The dynamic interaction with the environment is ignored.
2. *Feedback simulation.* The interaction between the simulated embedded system and the plant is tested. Another term for this (often used with process control systems) is that the "control laws" are verified. The model of the plant generates input for the embedded system. The resulting output is fed back into the plant, resulting in new input for the embedded system, and so on.
3. *Rapid prototyping.* The simulated embedded system is tested while connected to the real environment. This is the ultimate way of assessing the validity of the simulation model

Feedback simulation requires that a valid model of the plant, which is as close to the dynamic behaviour of the plant as desired or as possible, is developed first. This model is verified by comparing it with the actual behaviour of the environment. It often starts with the development of a detailed model followed by a simplification, both of which must be verified. The model of the plant can be verified by one-way simulation.

It is sound engineering practice, in model-based development, to develop the model of the plant *before* the model of the embedded system itself. The (simplified) model of the plant is then used to derive the control laws that must be implemented in the embedded system. Those control laws are then validated using one-way simulation, feedback simulation, and rapid prototyping.

One-way simulation

The embedded system is simulated in an executable model and the behaviour of the environment is ignored. Input signals are generated and fed into the model of the embedded system and output signals monitored, recorded, and analysed. The simulation is "one-way" because the dynamic behaviour of the plant is not included in

the model – Figure 5.3 depicts this situation schematically.

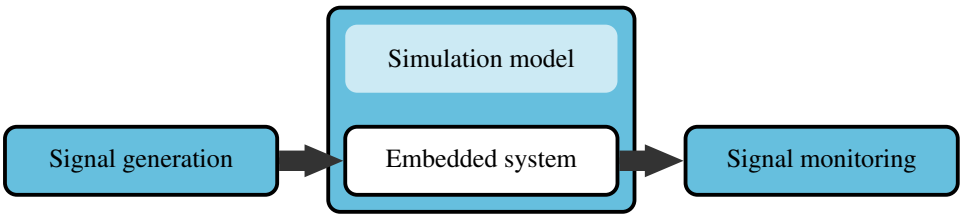


Figure 5.3 One-way simulation.

Tools are required to generate the input signals for the model of the embedded system and to record the output signals. Comparing the recorded signals against expected results can be performed manually, but alternatively by a tool. The signal-generating tool may even generate new input signals based on the actual results in order to narrow down the possible shortcomings of design. Depending on the simulation environment, the generation of input signals and the recording of output signals can be done in various ways.

- Simulation of the embedded system on a computer platform. Delivery of input signals and recording of output signals by means of hardware on the peripheral bus of the simulation platform, and by manual control and readout of variables within the model via the operation terminal of the computer platform.
- Simulation of the embedded system in a CASE environment as well as generating the input stimuli and capturing the output responses in the same environment. The executable model is created in a modelling language such as UML. From the corresponding use cases and sequence diagrams, test cases can be generated automatically.

One-way simulation is also used for the simulation of a model of the plant. In this case, the dynamic behaviour of the embedded system (incorporating the control laws) is not included in the model.

Vocabulary 5.2 — Computer-aided software engineering (CASE). CASE is the domain of software tools used to design and implement applications. CASE tools are similar to and were partly inspired by computer-aided design (CAD) tools used for designing hardware products. CASE tools are used for developing high-quality, defect-free, and maintainable software (Kuhn, 1989).

Feedback simulation

The embedded system and its surrounding environment (the plant) are both simulated in one executable dynamic model. This option is feasible if the complexity of the simulation model of the embedded system and its environment can be limited without compromising the accuracy of the model.

For instance, suppose that the embedded system to be designed is the cruise control of a car. The environment of the system is the car itself – and also the road, the wind, the outside temperature, the driver, etc. From the perspective of feasibility and usefulness it can be decided that the model is (initially) limited to the cruise control mechanism, the position of the throttle, and the speed of the car. Figure 5.4 shows a schematic representation of this type of simulation. Feedback simulation may be

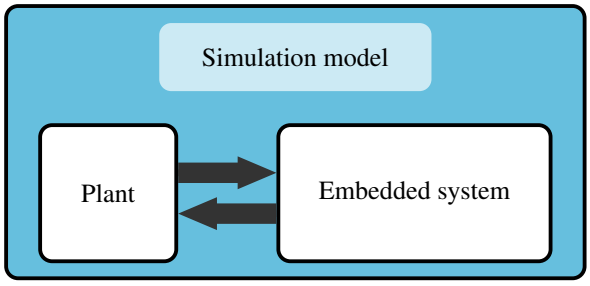


Figure 5.4 Feedback simulation.

the next step in a complex design process, after one-way simulation of the embedded system itself or the plant.

Verification of the design is obtained through the execution of a number of test cases that may include deviations in characteristics of the model and the changes in the state in which the model is running. The response of the model is monitored, recorded, and subsequently analysed.

The test bed for the model must afford the ability to change characteristics within the model of the embedded system or within the model of its environment. It must also offer readout and capture of other characteristics from which the behaviour of the model can be derived. Again, as with one-way simulation, the test bed can be arranged on a dedicated computer platform as well as in a CASE environment.

The tools used in this stage are:

- signal generating and monitoring devices;
- simulation computers;
- CASE tools.

Rapid prototyping

An optional step is the additional verification of control laws by replacing the detailed simulation of the plant with the actual plant or a close equivalent (see Figure 5.5). For instance, the simulation of the control laws for a cruise control mechanism, running on a simulation computer, can be verified by placing the computer on the passenger seat of a car and connecting it to sensors and actuators near the mechanics and electronics of the car. A high performance computer is typically used here and resource restrictions of the system are disregarded for a time. For instance, the rapid prototype software may be 32-bit floating point, while the end product is restricted to 8-bit integer processing.

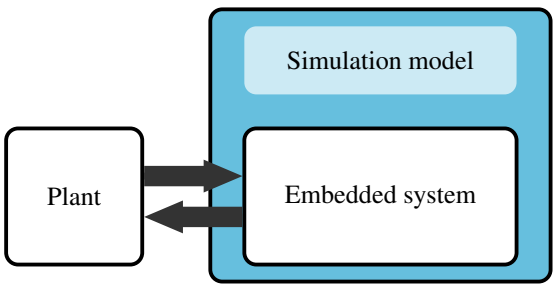


Figure 5.5 Rapid prototyping.

5.1.2 Second stage: prototyping

The development of an embedded system without the use of a model starts at this point. As in the case of model-based development as described above, the prototyping stage starts after the goals for the simulation stage have been achieved.

The goals of testing in this stage are:

- proving the validity of the simulation model (from the previous stage);
- verifying that the system meets the requirements
- releasing the pre-production unit.

In the prototyping stage, actual system hardware, computer hardware, and experimental and actual versions of the software gradually replace simulated components. The need for interfaces between the simulation model and the hardware arises. A processor emulator may be used. Signals that are readily available in the simulation model may not be as accessible with the real hardware. Therefore, specific signal pickups and transducers, may have to be installed as well as signal recording and analysis equipment. It may be necessary to calibrate pickups, transducers, and recording equipment and to save the calibration logs for correction of the recorded data. One or more prototypes are developed. Often this is a highly iterative process, where the software and hardware are further developed in parallel and frequently integrated to check that they still work together. With each successive prototype, the gap with the final product narrows. With each step, the uncertainty about the validity of the conclusions reached earlier is further reduced.

To clarify and illustrate this gradual replacement of simulated components, four major areas are identified in the generic scheme of an embedded system (see Figure 5.6). They are the parts of the the whole that can be individually subjected to simulation, emulation, and experimentation or preliminary versions in the prototyping stage:

1. The system behaviour realised by the *embedded software*. It can be simulated by:
   - running the embedded software on a host computer compiled for this host;
   - running the embedded software on an emulator of the target processor running on the host computer. The executable of the embedded software can be considered “real” because it is compiled for the target processor.
2. The *processor*. In the early development stages, a high performance processor can be used in a development environment. An emulator of the final processor can be used to test the real software – compiled for the target processor – in the development environment.
3. The *rest of the embedded system*. It can be simulated by:
   - simulating the system in the test bed on the host computer;
   - constructing an experimental hardware configuration;
   - constructing a preliminary (prototype) printed circuit board with all of its components.
4. The *plant* or the environment of the embedded system. It can be simulated statically by means of signal generation, or dynamically by means of a simulation computer.

In the prototyping stage, the following test levels (see section 2.3.2) are applicable:

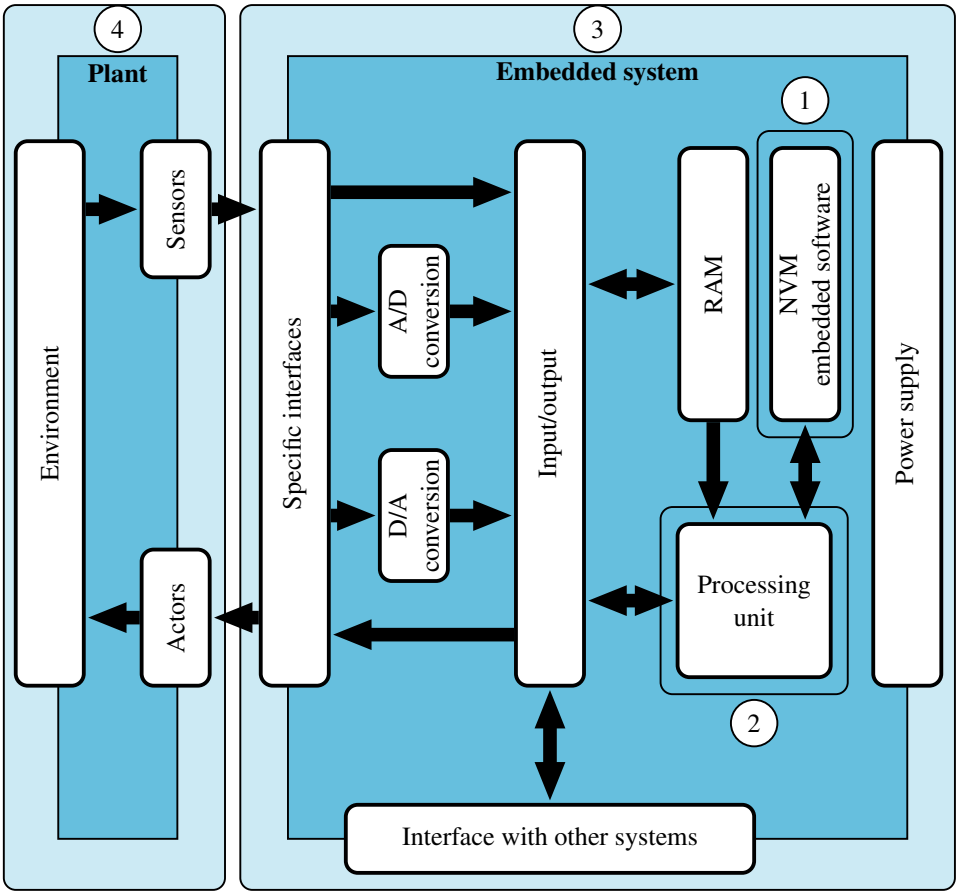


Figure 5.6 Simulation areas in an embedded system.

- software unit test;
- software integration test;
- hardware/software integration test;
- system integration test;
- environmental test.

Software unit and software integration tests

For the software unit (SW/U) tests and the software integration (SW/I) tests, a test bed is created that is comparable to the test environment for the simulation model. The difference lies in the object that is executed. In the prototype stage the test object is an executable version of a software unit, or a set of integrated software units, that is developed on basis of the design or generated from the simulation model.

The first step is to compile the software for execution on the host computer. This environment (host) has no restrictions on resources or performance, and powerful tools are commercially available. This makes development and testing a lot easier than in the target environment. This kind of testing is also known as *host/target testing*. The goal of the tests on these "host-compiled" software units and integrated software units is to verify their behaviour according to the technical design and the validation of the simulation model used in the previous stage.

The second step in SW/U and SW/I tests is to compile the software for execution on the target processor of the embedded system. Before actual execution of this software on the target hardware, the compiled version can be executed within an emulator of the target processor. This emulator may run on the development system or on another host computer. The goal of these tests is to verify that the software will execute correctly on the target processor.

In both of the above mentioned situations the test bed must offer stimuli on the input side of the test object, and provide additional features to monitor the behaviour of the test object and to record signals. This is often provided in the form of entering break points, and storing, reading, and manipulating variables. Tools that provide such features can be part of the CASE environment or can be specifically developed by the organisation developing the embedded system.

Table 5.1 provides an overview of the level of simulation in both steps in SW/U and SW/I tests. The columns refer to the simulation areas in the generic scheme of an embedded system (see Figure 5.6).

Table 5.1 Simulation level for SW/U and SW/I tests.

	Embedded Software	**Processor**	**Rest of embedded system**	**Plant**
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated

Hardware/software integration tests

In the hardware/software integration (HW/SW/I) tests the test object is a hardware part on which the integrated software is loaded. The software is incorporated in the hardware in memory, usually (E)EPROM. The piece of hardware can be an experimental configuration, for instance a hard-wired circuit board containing several components including the memory. The term "experimental" indicates that the hardware used will not be developed further (in contrast with a prototype which is usually subject to development). The goal of the HW/SW/I test is to verify the correct execution of the embedded software on the target processor in co-operation with surrounding hardware. Because the behaviour of the hardware is an essential part of this test, it is often referred to as "hardware-in-the-loop".

The test environment for the HW/SW/I test will have to interface with the hardware. Depending on the test object and its degree of completeness, the following possibilities exist:

- offering input stimuli with signal generators;
- output monitoring with oscilloscopes or a logic analyser, combined with data storage devices;
- in-circuit test equipment to monitor system behaviour on points other than the outputs;
- simulation of the environment of the test object (the "plant") in a real-time simulator.

Table 5.2 provides an overview of the level of simulation in the HW/SW/I test. The columns refer to the simulation areas in the generic scheme of an embedded system (see Figure 5.6).

Table 5.2 Simulation level for HW/SW/I tests.

	Embedded Software	Processor	Rest of embedded system	Plant
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	Real (target)	Real (target)	Experimental	Simulated

System integration tests

All the hardware parts that the embedded system contains are brought together in the system integration test, generally on a prototype printed circuit board. Obviously, all the preceding tests SW/U, SW/I and HW/SW integration level will have to be executed successfully. All the system software has to be loaded. The goal of the system integration test is to verify the correct operation of the embedded system.

The test environment for the system integration test is very similar to that for the HW/SW/I test – after all, the complete embedded system is also a piece of hardware containing software. One difference may be found in the fact that the prototype printed circuit board of the complete system is provided with its final I/O and power supply connectors. The offering of stimuli and monitoring of output signals, possibly combined with dynamic simulation, will take place via these connectors. After signal monitoring on connectors, in-circuit testing may take place at predefined locations on the printed circuit board.

Table 5.3 provides an overview of the level of simulation in the system integration tests. The columns refer to the simulation areas in the generic scheme of an embedded system (see Figure 5.6).

Table 5.3 Simulation level for system integration tests.

	Embedded Software	Processor	Rest of embedded system	Plant
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	Real (target)	Real (target)	Experimental	Simulated
System integration	Real (target)	Real (target)	Prototype	Simulated

Environmental tests

The goal of environmental tests in this stage is to detect and correct problems in this area in the earliest possible stage. This is in contrast with environmental tests during the pre-production stage, which serve to demonstrate conformity. Environmental tests preferably take place on prototypes that have reached a sufficient level of maturity. These prototypes are built on printed circuit boards and with hardware of the correct specifications. If EMI countermeasures are required in the embedded system, they will also have to be taken on the prototype that is subject to the tests.

The environmental tests require very specific test environments. In most cases the test environments can be purchased or assembled from purchased components to meet requirements. If specific types of environmental tests are performed only occasionally, it is recommended that the necessary equipment is rented, or that the tests are contracted out to a specialist company.

During the environmental tests, the embedded system may be tested under operation, in which case simulation of the plant will have to be created as in the two

previous test levels.

Environmental tests consist of two types:

- *Emphasis on measuring.* This type of test is intended to determine the degree to which the embedded system influences its environment – for instance, the electromagnetic compatibility of the system. The tools required for this type of tests are therefore measurement equipment.
- *Emphasis on generating.* These tests serve to determine the susceptibility of the embedded system to surrounding conditions. Examples are temperature and humidity tests, shock and vibration tests, and tests on electromagnetic interference. The required test facilities are devices that inflict these conditions on the system, such as climate chambers, shock tables and for EMC/EMI tests equipped areas. Measurement equipment is required as well.

Table 5.4 provides an overview of the level of simulation in the environmental test.

Table 5.4 Simulation level for environmental tests.

	Embedded Software	**Processor**	**Rest of embedded system**	**Plant**
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	Real (target)	Real (target)	Experimental	Simulated
System integration	Real (target)	Real (target)	Prototype	Simulated
Environmental	Real (target)	Real (target)	Real	Simulated

5.1.3 Third stage: pre-production

The pre-production stage involves the construction of a pre-production unit, which is used to provide final proof that all requirements, including environmental and governmental, have been met and to release the final design for production.

There are several goals of testing in this stage:

- To finally prove that all requirements are met.
- To demonstrate conformity with environmental, industry, ISO, military, and governmental standards. The pre-production nit is the first unit that is representative of the final product. Preliminary tests may have been performed on earlier prototypes to detect and correct shortcomings but the results of these tests may not be accepted as final proof.
- To demonstrate the system can be built in a production environment within the scheduled time with the scheduled effort.
- To demonstrate the system can be maintained in a real-life environment and that the mean-time-to-repair requirements are met.
- Demonstration of the product to (potential) customers.

The pre-production unit is a real system, tested in the real-life environment. It is the culmination of all the efforts of previous stages, as shown in table 5.5. The pre-production unit is equal to, or at least representative of the final production units. The difference with the production units is that the pre-production units may still have test provisions such as signal pickoffs, etc. However, these provisions are of production quality instead of laboratory quality. It may be necessary to have more than one specimen because tests may run in parallel or units may be destroyed in tests.

Sometimes the pre-production units are referred to as "red-label units" whereas real production units are called "black-label units".

Table 5.5 Pre-production as the conclusion of the gradual replacement of simulated components by real ones.

	Embedded Software	Processor	Rest of embedded system	Plant
SW/U, SW/I (1)	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	Real (target)	Real (target)	Experimental	Simulated
System integration	Real (target)	Real (target)	Prototype	Simulated
Environmental	Real (target)	Real (target)	Real	Simulated
Pre-production	Real (target)	Real (target)	Real	Real

The pre-production unit is subjected to real-life testing according to one or more predetermined test scenarios. In the example of a cruise control mechanism, these tests might consist of a number of test drives with a pre-production unit of the mechanism installed in a prototype car. Other pre-production units may be subjected to environmental qualification tests. In the previous stages, the test input signals may have been generated from the simulation model, and the output data may have been monitored online or stored on hard disk for analysis later. In the pre-production stage, instrumentation and data display and recording facilities are required. Maybe telemetry is needed – this might be the case if the tests involve a missile, an aircraft, a racing car or other vehicles with limited space for instrumentation or human observers. Due attention should be given to the quality and calibration of the test instrumentation, and the telemetry, the data display and recording equipment – if the data gathered cannot be trusted, the tests are worthless.

The types of tests that are applicable to this stage are:

- system acceptance test;
- qualification tests;
- safety execution tests;
- tests of production and maintenance test facilities;
- inspection and/or test by government officials.

Applicable test techniques are:

- real-life testing;
- random testing;
- fault injection.

Typical tools used in this stage are:

- environmental test facilities, such as climate chambers, vibration tables, etc.;
- data gathering and recording equipment;
- telemetry;
- data analysis tools;
- fault injection tools.

Table 5.6 provides a final and complete overview of all the discussed test levels (using the terminology from both sections 2.3.2 and 5.1) and the gradual replacement of simulated components by real ones.

Table 5.6 Pre-production as the conclusion of the gradual replacement of simulated components by real ones.

Test levels		Embedded Software	Processor	Rest of embedded system	Plant
One-way simulation	MT	Simulated	-	-	-
Feed-back simulation	MiL	Simulated	-	-	Simulated
Rapid prototyping	RP	Experimental	Experimental	Experimental	Real
SW/U, SW/I (1)	SiL	Experimental (host)	Host	Simulated	Simulated
SW/U, SW/I (2)	SiL	Real (target)	Emulator	Simulated	Simulated
HW/SW/I	HiL	Real (target)	Real (target)	Experimental	Simulated
System integration	HiL	Real (target)	Real (target)	Prototype	Simulated
Environmental	HiL/ST	Real (target)	Real (target)	Real	Simulated
Pre-production	ST	Real (target)	Real (target)	Real	Real

5.1.4 Fourth stage: post-development

The prototyping and pre-production stages finally result in a "release for production" of the system. This means that the tested system is of sufficient quality to be sold to the customer. But how can the organisation be sure that all the products that are going to be produced will have the same level of quality? In other words, if the production process is of insufficient quality, then the products will be of insufficient quality, despite a successful release for production. Therefore, before the start of large-scale production, the organisation may require additional measures in order to make this production process controllable and assure the quality of manufactured products.

The following measures should be considered:

- *Development and test of production facilities.* In the development of embedded systems, the development and test of the facilities for the production of the released system may be equally important. The quality of the production line has a major influence on the quality of an embedded system, therefore it is important to acknowledge the necessity of this measure. Development and subsequent testing of the production line is defined as a post-development activity, but can be done at any time during the development of the embedded system. However, it cannot take place during the modelling stage because of the absence of the final characteristics of the system. On the other hand, the production line has to be available before the actual start of production.
- *First article inspection.* Sometimes it is required to perform a first article inspection on the first production (black-label) units. The units are checked against the final specifications, change requests, production drawings, and quality standards to ensure that the production units conform with all specifications and standards. In the multiple V-model (see Section 2.2.1) this might be considered to be the very last V. The purpose is to develop and test the production process. After all, if the quality of the end product falls short, then all the development and test efforts will have been in vain.
- *Production and maintenance tests.* For quality control purposes it may be required to perform tests on production units. This may involve tests on each individual unit or more elaborate tests on production samples. Built-in test facilities may be required for troubleshooting and maintenance in the field. Facilities for production and maintenance tests must be designed in the unit (design for test) and tested in the development process. Moreover, development tests and their results may be used as basis for production and maintenance

tests.

5.2 Tools

Software now makes up the largest part of embedded systems and the importance of software is still increasing. This software becomes more complex and is implemented in critical systems more often. This has a major impact on testing – there is more to be tested and it is more complicated. Also changes in the embedded systems market have an impact on testing. Time-to-market and quality become major competing factors. This means that more and more complex software has to be developed at a higher quality level in less time. As a consequence, time pressure for testing and quality demands increase. A well-structured test process alone is now not sufficient to fulfil the quality demands within time. This, and the complexity of systems, make today's testing job almost impossible without the use of test tools. With the proper use of test tools, faster testing with consequently good quality is possible.

A test tool is an automated resource that offers support to one or more test activities, such as planning and control, specification, constructing initial test files, execution of tests, and assessment (Pol, Teunissen, & Veenendaal, 2002).

The using of test tools is not an objective on its own, but it has to support the test process. Using test tools makes only sense if it is profitable in terms of time, money, or quality. Introduction of test tools in a non-structured test process will seldom be successful. For high quality testing, structuring and automation are necessary. For successful introduction of test automation there must be a lifecycle and knowledge of all the test activities. The test process must be repeatable.

Tools are available for every phase in the test lifecycle. The list below provides an overview of how tools can be categorized according to where they are applied in the testing lifecycle:

Preparation : Case tool analyser, complexity analyser;

Specification : Test case generator;

Execution : Test data generator, record and playback tool, load and stress tool, simulator, stubs and drivers, debugger, static source code analyser, error detection tool, performance analyser, code coverage analyser, thread and event analyser, threat detection tool;

Planning and control : Defect management tool, test management tool, configuration management tool, scheduling and progress monitoring tool.

In this section we focus on the execution phase tools, which will be explained in more detail in the following subsection – the list above is not meant to be exhaustive. Most tools are available commercially but it is also possible that project-specific tools are developed in-house.

5.2.1 Execution phase

Many types of tools are available for the execution phase. Many tool vendors have generic and highly specialised tools in their portfolio. In this section, we describe the following tools:

- test data generator

- record and playback tool
- load and stress test tool
- simulator
- stubs and drivers
- debugger
- static source code analyser
- error detection tool
- performance analyser
- code coverage analyser
- thread and event analyser
- threat detection tool.

Test data generator

This generates a large amount of different input within a pre-specified input domain. The input is used to test if the system is able to handle the data within the input domain. This type of test is used when it is not certain beforehand if the system can handle all data within a certain input domain. For example, to perform evolutionary algorithm testing an automated test generator is essential.

Record and playback tool

This tool is very helpful in building up a regression test. Much of the test execution can then be done with the tool and the tester can mainly focus on new or changed functionality. Building up these tests is very labour intensive and there is always a maintenance problem. To make this type of tool profitable in terms of time, money, or quality, delicate organisational and technical decisions have to be made. Section 5.3 shows how to organise a test automation process.

Load and stress test tool

Systems that have to function under different load conditions have to be tested for their performance. Load and stress tools have the ability to simulate different load conditions. Sometimes, one may be interested only in the performance under load or, sometimes more interesting, the functional behaviour of the system under load or stress. The latter has to be tested in combination with a functional test execution tool.

Another reason to use this type of tool is to speed up the reliability test of a system. A straightforward reliability test takes at least as long as the mean time between failures, which often is in the order of months. The test duration can be reduced to a matter of days, by using load and stress tools that can simulate the same amount of use in a much shorter period.

Simulator

A simulator is used to test a system under controlled conditions. The simulator is used to simulate the environment of the system, or to simulate other systems which may be connected to the system under test. A simulator is also used to test systems under conditions that are too hazardous to test in real life, for example the control system of a nuclear power plant.

Simulators can sometimes be purchased on the open market, but usually a dedicated simulation environment has to be developed and built for the system under test.

See section 5.1.1 for detailed information about simulation.

Stubs and drivers

Interfaces between two system parts can only be tested if both system parts are available. This can have serious consequences for the testing time. To avoid this and to start testing a system part as early as possible, stubs and drivers are used. A stub is called by the system under test and provides the information the missing system part should have given – a driver calls the system part.

Standardisation and the use of a testbed architecture (see Figure 5.7) can greatly improve the effective use of stubs and drivers. The testbed provides a standard (programmable) interface for the tester to construct and execute test cases. Each separate unit to be tested must have a stub and driver build for it with a specific interface to that unit but with a standardised interface to the testbed. Techniques for test automation, such as data-driven testing, can be applied effectively. Such a testbed architecture facilitates the testing of any unit, the reuse of such tests during integration testing, and large-scale automation of low-level testing.

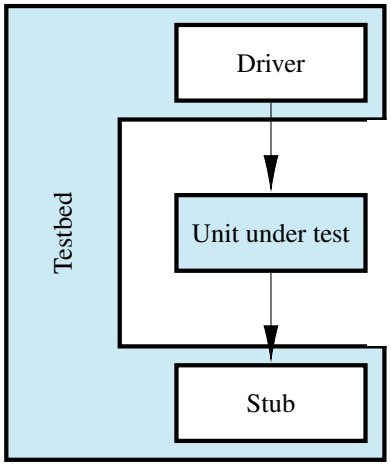


Figure 5.7 Testbed for testing a unit using stubs and drivers.

Debugger

A debugger gives a developer the opportunity to run a program under controlled conditions prior to its compilation. A debugger can detect syntactic failures. Debuggers have the ability to run the application stepwise and to set, monitor, and manipulate variables. Debuggers are standard elements of developer tools.

Static source code analyser

Coding standards are often used during projects. These standards define, for instance, the layout of the code, that pre- and post-conditions should be described, and that comments should be used to explain complicated structures. Additional standards which forbid certain error-prone, confusing or exotic structures, or forbid high complexity, can be used to enforce a certain stability on the code beforehand. These standards can be forced and checked by standard analysers. These already contain a set of coding standards which can be customized. They analyse the software and show the code which violates the standards. Be aware that coding standards are language dependent.

Error detection tool

This type of tool is used to detect run-time errors. The tool detects a failure and diagnoses it. Instead of vague and cryptic error messages, it produces an exact problem description based on the diagnosis. This helps to locate the fault in the source code and correct it.

Performance analyser

This type of tool can show resource usage at algorithm level while load and stress tools are used to do performance checks at system level. The tool can show the execution time of algorithms, memory use, and CPU-capacity. All these figures give the opportunity to optimize algorithms in terms of these parameters.

Code coverage analyser

This type of tool shows the coverage of the test cases. It monitors the lines of code triggered by the execution of test cases. The lines of code are a metric for the coverage of the test cases. Be aware that code coverage is not a good measure for the quality of the test set. At unit test level, for instance, the minimum requirement should be 100 percent statement coverage (otherwise this would mean that parts of the code have never executed at all).

Thread and event analyser

This type of tool supports the detection of run-time multi-threading problems that can degrade Java performance and reliability. It helps to detect the causes of thread starvation or thrashing, deadlocks, unsynchronised access to data, and thread leaks.

Threat detection tool

This type of tool can detect possible threats that can lead to malfunctioning. The type of threats are memory leaks, null pointer dereference, bad deallocation, out of bounds array access, uninitialised variables, or possible division by zero. The tool detect possible threats which mean that further analysis is necessary to decide whether the threat is real.

5.3 Test automation

Test execution is situated on the critical path to product introduction. Test automation is used, for instance, to minimise the time needed for test execution. Automation can be profitable in terms of time, money, and/or quality. It can also be used to repeat boring test procedures. Some tests, such as statistical usage testing and evolutionary algorithms, are impossible without test automation.

In principle, the execution of almost every test can be automated. In practice, only a small part of the tests are automated. Test automation is often used in the following situations:

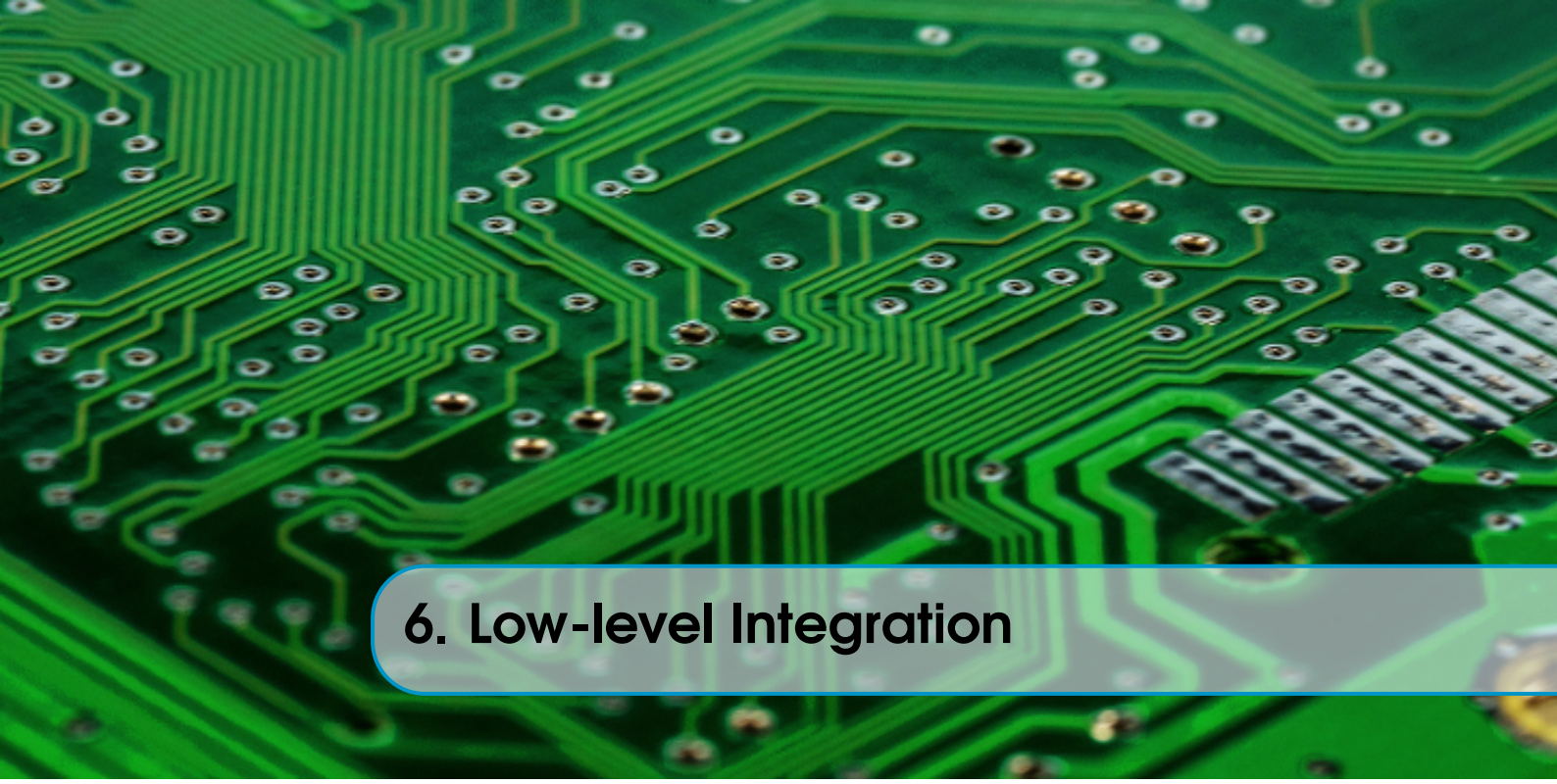
- tests that can be repeated many times;
- a basic test with a huge range of input variations – the same steps have to be executed every time but with different data (for instance, evolutionary algorithms);
- very complicated or error-prone tests;

- testing requires specialised equipment that generates the appropriate input signals, activates the system, and/or captures and analyses the output.

Changes in the system, design, and requirements are common and are also a threat to the usability of the automated tests. The consequences of these changes can be:

- additional test cases;
- changed test cases;
- different result checks;
- changed system interface;
- changed functionality;
- different signals;
- different target platform;
- different internal technical implementation.

Automated testing saves money and time if a test set can be used several times (payback is considered to be achieved if the complete test set is used between three and six times). This means that the automated tests should be used for consecutive releases, or in different stages of development. Which also means that automated tests have to be designed to deal with these uncertainties due to changes in the system, design and requirements.



6. Low-level Integration

When choosing a fast development language for prototyping (e.g., Python), but also a fast (efficient) language with respect to performance for the target platform (e.g., C++), at some point a switch-over needs to be made from one to the other. A drastic switch-over, by replacing the entire Python-code base (hopefully tested on bugs!) to a new code base in C++ can introduce lots of bugs that can be difficult to find. It is better to switch code piece by piece, while trying to build as much as possible on existing testing infrastructure to ensure that the new code (in C++) still adheres to the system and acceptance test that where specified for the Python code.

In order to be able to achieve this, it is beneficial to be able to link Python and C++ code. In that way, for example, you can start implementing a core piece of functionality in the target language, but connect it to the testing infrastructure (monitoring or simulation frameworks) that you have developed for the Python prototype.

In the following we explain the desired system solution that we want to design, using the phases as described in the previous chapter.

6.1 Casus: liquid mixer

In this section we describe the casus that is used to describe the various aspects of the low-level interfacing. We follow the steps of development as described in 5 for the development of the control system (embedded system and software) of a liquid mixer.

6.1.1 Summary

A control system for a liquid mixing system must be developed. This control system must mix two liquids and maintain the mixture at a set temperature. Our task is to develop the control system. The mixing system itself and the control system hardware are developed in parallel. Hence we start the development of the control software in a simulated environment, realised in Python. We gradually replace parts of the

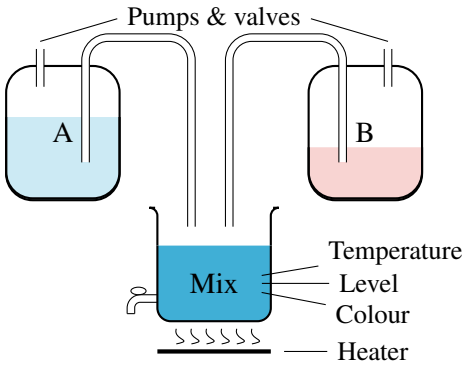


Figure 6.1 A mixture gone wrong (left); liquid mixer system layout (right).

simulated system with the physical mixing system and the embedded hardware.

6.1.2 System description

Two storage vessels contain cool liquids. Each storage vessel has an air pump that can push liquid from the storage vessel into the mixing vessel, and an air valve to vent the air pressure. The mixing vessel has analog sensors for level, temperature and colour, and a heating element. The ratio of the liquids determines the colour of the mixture. The control system must maintain the level in the mixing vessel, the colour of the mixture, and its temperature. These set points for these values can be changed.

Electronic interface

The mixing system has the following inputs and outputs.

Table 6.1 Description of the electronic interface.

Name	Type and direction	Description
Air pump A	Digital input	3..5V activates the air pump that pushes the liquid from storage vessel A into the mixing vessel.
Air valve A	Digital input	3..5V activates the air valve that releases the air pressure in storage vessel A.
Air pump B	Digital input	3..5V activates the air pump that pushes the liquid from storage vessel B into the mixing vessel.
Air valve B	Digital input	3..5V activates the air valve that releases the air pressure in storage vessel B.
Heater	Digital input	3..5V activates the heater of the mixing vessel.
Colour	Analog output	0..3V, depending on the colour of the mixture.
Temperature	Analog output	0..3V, depending on the temperature of the mixture.
Level	Analog output	0..3V, depending on the level of the liquid in the mixture vessel.

The system being built is a type control system commonly known as a *negative*

feedback control system. A control system is a system that manages or regulates the behaviour of other devices or systems. Control systems range from home heating controllers using a thermostat controlling a boiler to large industrial control systems which are used for controlling processes and machines.

In the most common form, a *feedback control system* is desired to control a process, called the plant, so its output follows a control signal, which may be a fixed or changing value. The control system compares the output of the plant to the control signal, and applies the difference as an error signal to bring the output of the plant closer to the control signal. A well-known type of feedback control is the PID-controller.

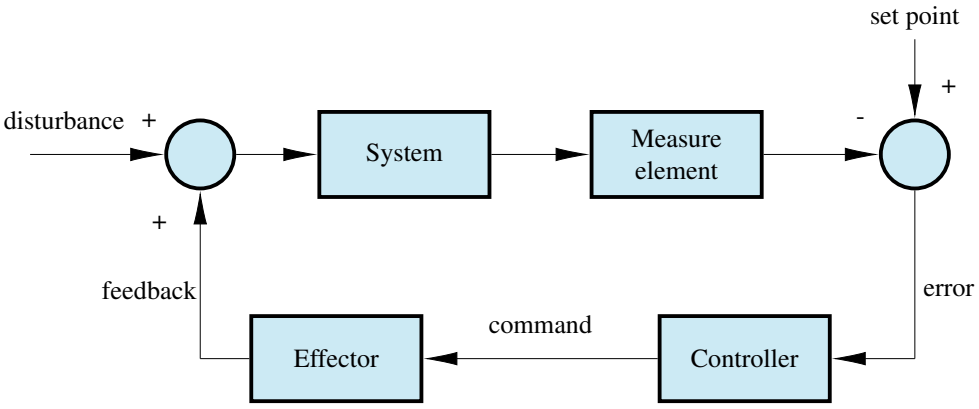


Figure 6.2 A negative feedback control system.

Figure 6.2 shows an example of an *negative feedback control system*. Such a control system tries to establish an equilibrium on predefined values (set points). The negative feedback then occurs when some function of the output of the plant is fed back in a manner to attempt to reduce the fluctuations in the output. Negative feedback, as opposed to positive feedback, tends to promote a settling to an equilibrium, and reduces the effects of perturbations. Positive feedback tends to lead to instability via exponential growth, oscillation or chaotic behaviour. Negative feedback loops, with the right amount of correction applied with optimum timing, can be very stable, accurate, and responsive.

Set points

The control module manipulates the inputs of the mixing system in order to keep its key values (level, temperature, colour) at the values indicated by the set points. The control module has 3 set points: The characteristic of the colour sensor is not easily

Table 6.2 Liquid mixer set points.

Set point	Unit	Effect
Colour	Volt	Sets the desired colour of the mixture.
Temperature	°C	Sets the desired temperature of the mixture.
Level	ml	Sets the desired level of the mixture (liquid in the mixing vessel).

described in physical terms, hence the colour set point is specified in Volt[1].

Characteristics

The physical system has a number of known characteristics. In order to be realistic, the simulator must reflect these characteristics.

Note that the characteristics mentioned here are an estimation of what the physical system will actually exhibit. In order to make a correct simulation, these values need to be validated with respect to the actual implementation of the physical system (called a Model-in-the-Loop (MiL) test, see vocabulary 5.1 and section 5.1.1).

Table 6.3 Liquid mixer characteristics.

Name	Value	Description
Air pressure ramp-up	3s	Time from start of the air pump to start of liquid flow (from storage vessel to mixing vessel).
Air pressure ramp-down	20s	Time from stop of air pump to stop of liquid flow.
Liquid flow	1ml/s	Amount of liquid that flows.
Mixing vessel geometry	10cm diameter	Surface of bottom of mixing vessel.
Environment temperature	20°C	This is also the temperature of the liquid in the storage vessels.
Heat conductivity	10°C/W	The heat conductivity from the mixing vessel liquid to the environment.
Temperature sensor	0.05 V/°C	0.00V = 0.0°C.
Level sensor	0.5V / cm	Senses the liquid level (0V is empty).

Disturbances

Beside the set points the system can be disturbed in a number of ways:

- First and foremost: draining liquid from the mixing vessel. After all, the purpose of the system is to provide the appropriate mixture at the appropriate temperature.
- Changing the environment temperature, or the temperature in one of the storage vessels.
- Changing the temperature or colour of the liquid in the mixing vessel. This is less likely to occur in practice, but the control system should still respond appropriately to it.

6.1.3 Initial situation

Initially the following modules must be developed in Python[2]:

[1] Colour can be expressed in brightness (*value* from the HSV Hue/Saturation/Value scale; which ranges from 0% to 100% representing pitch black and bright white, respectively. The value of a colour is the brightness of its related (closest) grey value.

[2] A partial implemented simulator is provided to help you get started; the UML of this implementation can be found in Figure 6.3.

- A simulator module that simulates the behaviour of the physical system. It implements the inputs and outputs of the system. The relevant characteristics are also inputs to the simulator.
- A control module that accepts the set points, and the measurements from the simulator, and controls the simulators inputs to maintain its process parameters at the levels set by the set points.
- A GUI that can manipulate selected variables in the simulator and control modules.
- A test driver that exercises the simulator and control modules according to a script, and records the key values of the simulated system, asserting that they are within acceptable ranges. Either the GUI or the test driver are used.
- A monitoring module that records and displays selected variables from the other modules. It can display both actual (current) and historic values.

6.1.4 Test scripts

The system (physical or simulated) must be tested in a repeatable way. To do this, test scripts must be written that specify what is done to the system (like 'drain 100 ml from the mixing vessel'), and what the acceptable response of the system is (like 'within 20 seconds the mixing vessel level is restored, within 60 seconds the temperature is restored.'). These test scripts can be run automatically on the simulated system, or by hand on the physical system, etc.

Verification of the model by using the physical system

The development process assumes that the model is a reasonable approximation of the real physical system. This assumption must be verified using the real physical system. To do this, the model of the system (in Python) must be replaced by a (Python) module that exposes the same interface but interacts directly with the real physical system, via the GPIO pins of the hardware. The analog outputs of the physical system can be read either by AD inputs (if the hardware has any) or by using a suitable AD converter module, interfaced via I2C or SPI.

Rewriting the control module in C++

As a step towards running the control software on an embedded system the Python control module is rewritten in C++, but still run with the (other) Python modules. This requires that the C++ control module itself is supplemented with C++ code that interfaces with the other (Python) modules.

Embedded control of the physical system

The end situation is the that the control module, implemented in C++, running on embedded hardware (Arduino Due or similar) controls the physical system. This combination is tested by a combination of manual steps ('drain 100 ml from the storage vessel') and either human or automated checks.

6.1.5 Hardware

The following hardware modules are available:

- physical system (2)

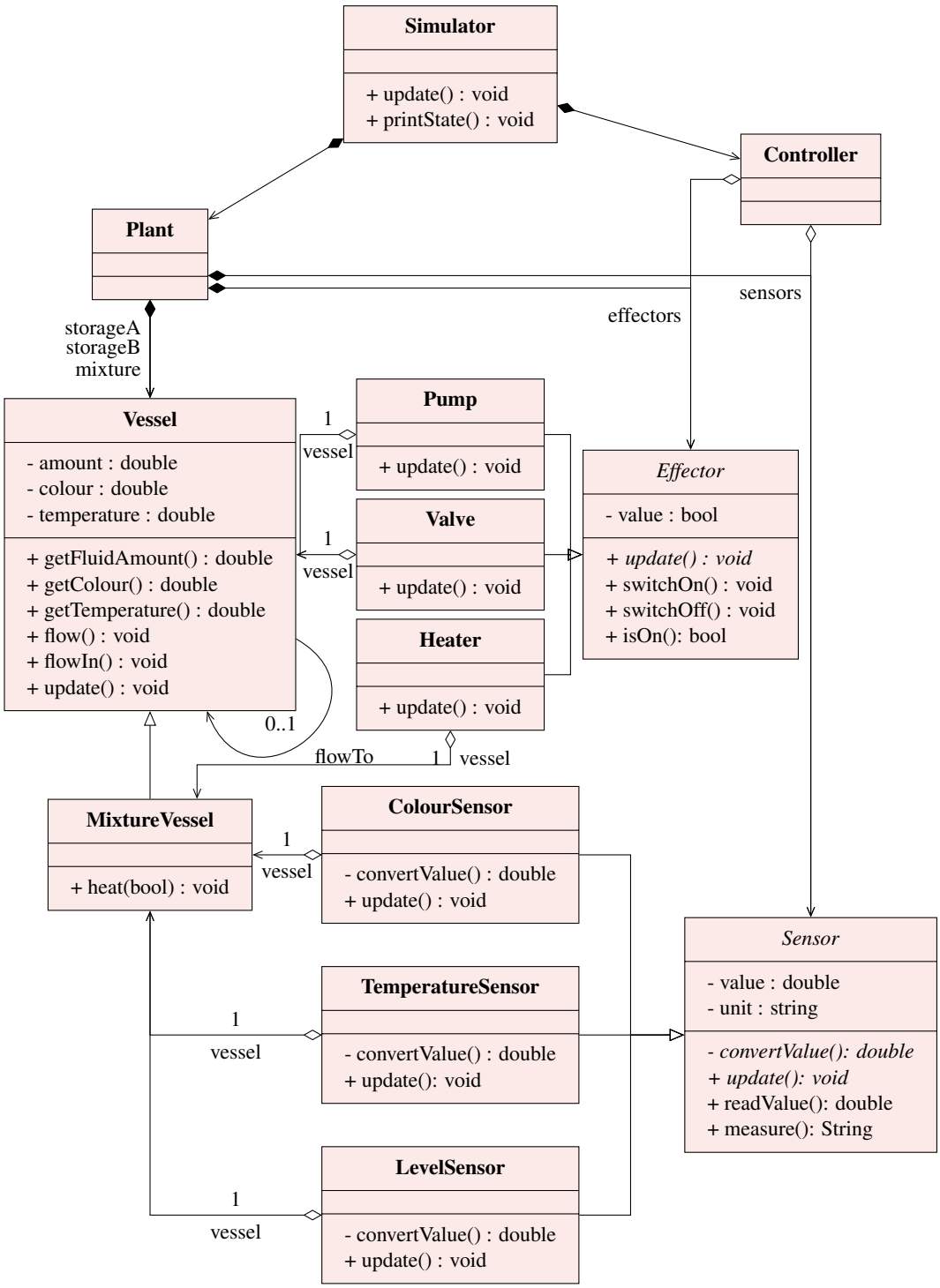


Figure 6.3 Class diagram of Liquid Mixer Simulator.

  - This is the physical system that is being controlled. It has 3 vessels, 2 pumps, 2 valves, 1 heater, 3 analog sensors.
- monitoring panel (2)

  - This can be connected before the physical system to displays the 4 digital values to the physical system and the 3 analog values from the physical system.
- PC interface (2)
  - This interface has 4 digital outputs and 3 analog inputs. Two options: an Arduino Due that has a serial interface to the computer (PC or Pi), or a header + 3-input analog interface (Pi only)
- Control panel (2)
  - This panel can be used to control the physical hardware directly. It has 4 switches that drive the 4 digital outputs.

6.2 Realisation: the Lemonator

This section describes the lemonator hardware, the C++ source code available for it, and the assignments to be done with this hardware and software.

The aim of this part is to get experience with mixed-language (Python and C++) and mixed-platform (PC and micro-controller) development. The application is first developed in Python, verified against a Python simulator of the target system. In steps, this application is translated to an embedded C++ application running on the real hardware.

6.2.1 Hardware

The lemonator is a device (see Figure 6.4) that can pour two liquids (e.g., syrup and water) into a cup, and keep the cup warm.

Figure 6.4 The Lemonator hardware.

Control

An Arduino Due controls the hardware of the lemonator. Note that an Arduino Due will **not** automatically start when power is applied: you must press the reset button

(lower right side of the device) to active the application stored in its flash.

Liquid handling

The two liquids are handled the same way. An air pump builds up air pressure in the container, which pushes the liquid out through a hose, into the cup. An air valve releases the air pressure (quickly), which halts the liquid flow. This setup avoids liquid flowing through a pump, which could cause contamination of the liquid or obstruction of the pump.

LEDs show the activity of each pump (red) and valve (green). The two liquids are handled the same way: the two liquid containers each have a pump, a valve and two LEDs. The pumps and valves are controlled by the Due, the LEDs are connected in parallel (they are not separately controlled by the Arduino Due).

Cup sensors

The cup location has three sensors:

- A presence sensor (CNY70 IR reflex sensor) that detects whether something (presumably a cup) is present or not.
- A level sensor (SR04 US distance sensor) measures the distance from this sensor (placed above the cup) to the liquid surface.
- A colour sensor (TCS3200) measures the colour of the liquid.

The level sensor is not very reliable, especially when no liquid is present, or when the liquid surface has ripples or bubbles.

A green LED left of the cup position is controlled by the Arduino Due and is meant to show the presence of a cup.

User interface

The lemonator has a character LCD (4x20 HD44780) and a keypad (4x4 matrix) keypad as user interface. The LCD can be used to show user messages (like a selection menu), and the keypad can be programmed to allow the user to perform various functions of the lemonator.

Heater

The metal plate below the cup is heated by the (orange) heater (12V, 1A). The heater is switched by the controller. The temperature of the plate is measured by a (DS1820) temperature sensor connected to the controller. A yellow LED left of the cup position is controlled by the Due and meant to show the activity of the heater.

The DS1820 interface is very slow: the chip requires 800 ms for a temperature conversion.

The temperature is also measured by another sensor and shown by a 3-digit 7-segment display. This display is for diagnostics only, it is not connected to the Arduino Due. When left activated, the heater plate will reach a temperature of $\sim 70°$C.

The heater is not very functional for heating the liquid in the cup, because of its low capacity (12W) and the lack of coupling to the cup.

Power

The lemonator is powered by a 12V (at least) 2A supply. The Arduino Due controller is powered from this 12V supply, or by its USB connection to a laptop. The power

connector is a male XLR chassis part (1 = GND, 2 = 12V).

Panic plug

The lemonator has an XLR plug (to the left of the keypad) that can be pulled in case of emergency. This stops both air pumps, and activates both air valves, thus stopping the flow from both liquid containers (this, of course, assumes that 12V power is still present). This feature was added as an extra precaution. Please do not rely on it to keeps things dry.

6.2.2 Lemonator software

Abstract hardware interface

An abstract interface has been defined for the lemonator hardware. This is a class with a number of (references to) objects[3].

```
class lemonator_interface {
public:

   hwlib::ostream               & lcd;
   hwlib::istream               & keypad;

   hwlib::sensor_distance       & distance;
   hwlib::sensor_rgb            & colour;
   hwlib::sensor_temperature    & temperature;
   hwlib::pin_in                & presence;

   hwlib::pin_out               & heater;
   hwlib::pin_out               & syrup_pump;
   hwlib::pin_out               & syrup_valve;
   hwlib::pin_out               & water_pump;
   hwlib::pin_out               & water_valve;
   hwlib::pin_out               & led_green;
   hwlib::pin_out               & led_yellow;

   filler                       syrup;
   filler                       water;

   . . .
};
```

The digital inputs and outputs are represented by `hwlib::pin_in` and `hwlib_pin_out` objects. All digital inputs and outputs in the abstract interface are active high (even when the actual hardware interface is active low).

The keypad is represented by an `hwlib::istream`, which has a function `getc_nowait()`, which will return either a character, or `'\0'` when no new character is available. The

[3] All references to **hwlib** refer to the library available at `https://github.com/wovo/hwlib`.

stream returns key presses as characters, it has no way to detect whether a key is hold down.

The LCD is represented by an `ostream`, which supports formatting using the various `operator<<` functions, and the `hwlib::console` cursor positioning:

- `'\n'` puts the cursor at the first position of the next line;
- `'\r'` puts the cursor at the start of the current line;
- `'\f'` puts the cursor at the top-left position and clears the console;
- `'\txxyy'` puts the cursor at the position (*xx*,*yy*).

The distance, colour and temperature sensors are represented by objects that provide read operations for these values.

```
class sensor_distance {
public:
   /// distance in mm
   virtual int read_mm() = 0;
};
```

The colour sensor returns the 3 colours as percentages (they add up to 100). It not very sensitive, the variations from clear water to reasonable lemonade are small.

```
class sensor_rgb {
public:
   struct rgb {
      int r, g, b;
      rgb( int r, int g, int b ): r( r ), g( g ), b( b ){}
   };

   template< typename T >
   friend T & operator<<( T & cout, const rgb & c ){
      return cout << "(" << c.r << "," << c.g << "," << c.b << ")";
   }

   /// return the colour as 3 percentages
   virtual rgb read_rgb() = 0;
};

class sensor_temperature {
public:
   /// temperature in millidegrees Celsius
   virtual int read_mc() = 0;
};
```

The fillers each represent an air pump and an air valve.

```
class filler {
public:
   hwlib::pin_out & pump;
   hwlib::pin_out & valve;
```

```
   . . .
};
```

Direct hardware interface

For use on the Arduino Due an implementation of the abstract interface is available that directly controls the hardware. This is how the final application has to work: running directly on the Arduino Due[4].

```
#include "lemonator_hardware.hpp"

int main( void ){
   . . .
   auto hw = lemonator_hardware();
   hw.lcd << "\\fHello world!"
   hw.syrup_pump.set( 1 );
}
```

Remote server

To use the lemonator remotely (over the USB/serial interface) a server application is available to run on the Arduino Due. It responds to simple commands issued over the serial interface. You could use it to experiment with the hardware without having to write embedded code, but its main purpose is to act as a remote for the proxy interface described next. For example, to enable the water pump, type `1wp`.

Proxy interface

A proxy implementation of the interface is available for use on a PC. It communicates over the USB/serial port with the server application running on the Arduino Due. It implements the same interface as the direct hardware interface. This makes it possible to run the same 'application' code either on the PC or on the Arduino Due.

```
#include "lemonator_proxy.hpp"

int main( void ){

   // COM8 => 7, COM3 => 2
   auto hw = lemonator_proxy( 2, 0, 0 );

    // wait for remote to re-start after opening the com port
   hwlib::wait_ms( 4'000 );

   test: blink a LED
   hwlib::blink( hw.led_yellow );
}
```

[4]All Lemonator code fragments are available on `https://github.com/wovo/vkatp-lemonator/`.

Table 6.4 Lemonator command table.

Command	Effect	Response (example)
`t`	Read and return temperature	`temperature=21000\n`
`d`	Read and return distance	`distance=88\n`
`c`	Read and return colour	`color=(12,50,38)\n`
`r`	Read and return presence (reflex sensor)	`reflex=1\n`
`0`	Set current value to 0 (in preparation for a real command	
`1`	Set current value to 1 (idem)	
`w`	Select water filler (in preparation for real command)	
`s`	Select syrup filler (idem)	
`p`	Set pump of selected filler to current value	`syrup pump on\n`
`v`	Set valve of selected filler to current value	`water valve off\n`
`g`	Set the green LED to the current value	`green led on\n`
`y`	Set the yellow LED to current value	`yellow led off\n`
`x...?`	Send the characters ... to the LCD. `?` ends the sequence of characters.	
`z`	Read keypad and return the (one) character that was read, or a space when no (new) character is available.	`kbd=xn`
`<space>`	Stop all: disable pumps and heater; enable valves; wait half a second; disable valves.	

Note that opening the serial port will reset the Due, hence a remote application must, after the port has been opened by constructing the proxy object, wait a few seconds for the Arduino to come to live.

The proxy uses a serial port library (RS-232) that is supposed to be compatible with Linux, but this has not been tested.

Python interface

A first shot at a Python interface is available, using pybind11. It creates a `.pyd` file that can be loaded into Python. To build this file, you need the vkatp-lemonator files, the latest bmptk, and put pybind11 next to the vkatp-lemonator files. (All can be found on Github[5].) The bmptk build assumes Python 3.5 in `C:\Program Files (x86)\Python`. For other pythons you have to make some adjustments. Alternatively, there is CMake support in pybind11.

In Python you can load the interface module, create a lemonator object, and use it. The lemonator constructor argument is the serial port number (COM1 == 0). The following Python code does this and blinks the yellow LED.

```
import lemonator
import time

hw = lemonator.lemonator( 2 )
```

[5] `https://github.com/wovo/`

```
led = hw.led_yellow
while 1:
   led.set( 1 )
   time.sleep( 0.5 )
   led.set( 0 )
   time.sleep( 0.5 )
```

The interface to Python is created by the pybind11 library. This (header only) library uses some deprecated features, hence that specific warnings are disabled for the include.

The `PYBIND11_MODULE` macro provides Python a module with the name "lemonator". Note that the first argument is effectively a string, not a variable name. The second argument is an identifier that is used in the ... part (it could be any other identifier, if it was used consistently).

```
#include "lemonator_proxy.hpp"

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wdeprecated-declarations"
#include "pybind11/pybind11.h"
#pragma GCC diagnostic pop

namespace py = pybind11;
PYBIND11_MODULE( lemonator, m ){
   . . .
}
```

In the ... part we must describe the lemonator C++ types that are provided to Python within the lemonator module. The main type is the `lemonator_interface` class, which is provided under the name "lemonator". We must mention which elements of this class are provided to Python, how they are provided, and under what name.

Unfortunately, the `lemonator_interface` uses references to the abstract interface objects, and pybind11 seems to be unable to provide Python access to references (please correct me if I am wrong). Hence the Python interface provides access to the concrete `lemonator_proxy`, which contains the concrete (proxy) objects. For now, only the constructor and the `p_led_yellow` object are provided.

```
class lemonator_proxy : public lemonator_interface {
public:
   lemonator_proxy(
      int p,
      bool log_transactions = 0,
      bool log_characters = 0
   );
   output_proxy         p_led_yellow;
   . . .
};
```

```
   py::class_< lemonator_proxy >( m, "lemonator" )
      .def( py::init< int >() )
      .def_readonly( "led_yellow", &lemonator_proxy::p_led_yellow );
```

The p_led_yellow is of the type output_proxy, hence we must describe that type in order for the python code to do anything with it.

```
class output_proxy : public hwlib::pin_out {
public:
   void set(
      bool b,
      hwlib::buffering buf = hwlib::buffering::unbuffered
   ) override { ... }
};
```

```
   py::class_< output_proxy >( m, "output_proxy" )
      .def( "set", &output_proxy::set, "",
         py::arg("v"), py::arg("buffering") =
                                        hwlib::buffering::unbuffered );
```

The one function that we provide has a (default) argument of an enumerate type, hence we must also describe that type.

```
namespace hwlib {
   enum class buffering { unbuffered, buffered };
};
```

```
   py::enum_< hwlib::buffering >( m, "buffering")
      .value( "unbuffered", hwlib::buffering::unbuffered )
      .value( "buffered", hwlib::buffering::buffered )
      .export_values();
```

This completes the necessary description needed to access the yellow LED. This is the full code of the (very limited) interface.

```
#include "lemonator_proxy.hpp"

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wdeprecated-declarations"
#include "pybind11/pybind11.h"
#pragma GCC diagnostic pop

namespace py = pybind11;

PYBIND11_MODULE( lemonator, m ) {

   py::enum_< hwlib::buffering >( m, "buffering")
      .value( "unbuffered", hwlib::buffering::unbuffered )
      .value( "buffered", hwlib::buffering::buffered )
      .export_values();

   py::class_< output_proxy >( m, "output_proxy" )
      .def( "set", &output_proxy::set, "",
         py::arg("v"), py::arg("buffering") =
                                    hwlib::buffering::unbuffered );

   py::class_< lemonator_proxy >( m, "lemonator" )
      .def( py::init< int >() )
      .def_readonly( "led_yellow", &lemonator_proxy::p_led_yellow );
}
```

Overview

Figure 6.5 shows the relations between the available lemonator software elements.

On the Arduino Due, the `lemonator_hardware` implements the `lemonator_interface` and drives to the physical hardware. The lemonator server uses a `lemonator_interface`

(presumably the `lemonator_hardware`) to provide a serial port interface.

On the PC, the `lemonator_proxy` uses the serial port connection to the `lemonator_server` to provide the `lemonator_interface`. The pybind11 lemonator-to-python bridge provides this interface in Python. Finally, Python code uses this interface to talk to the physical lemonator hardware.

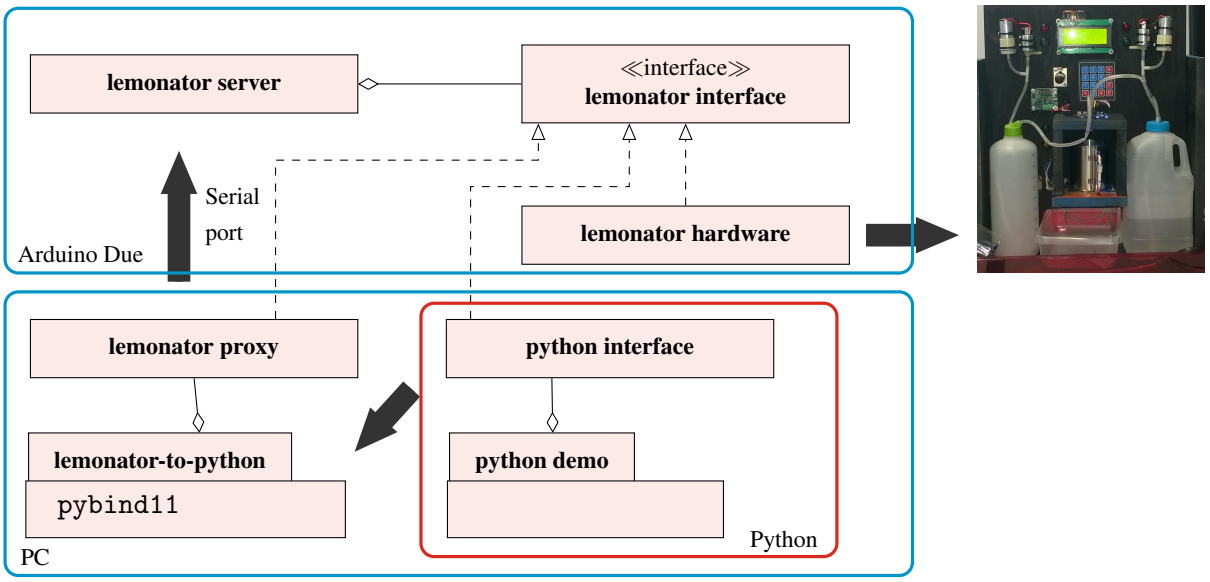
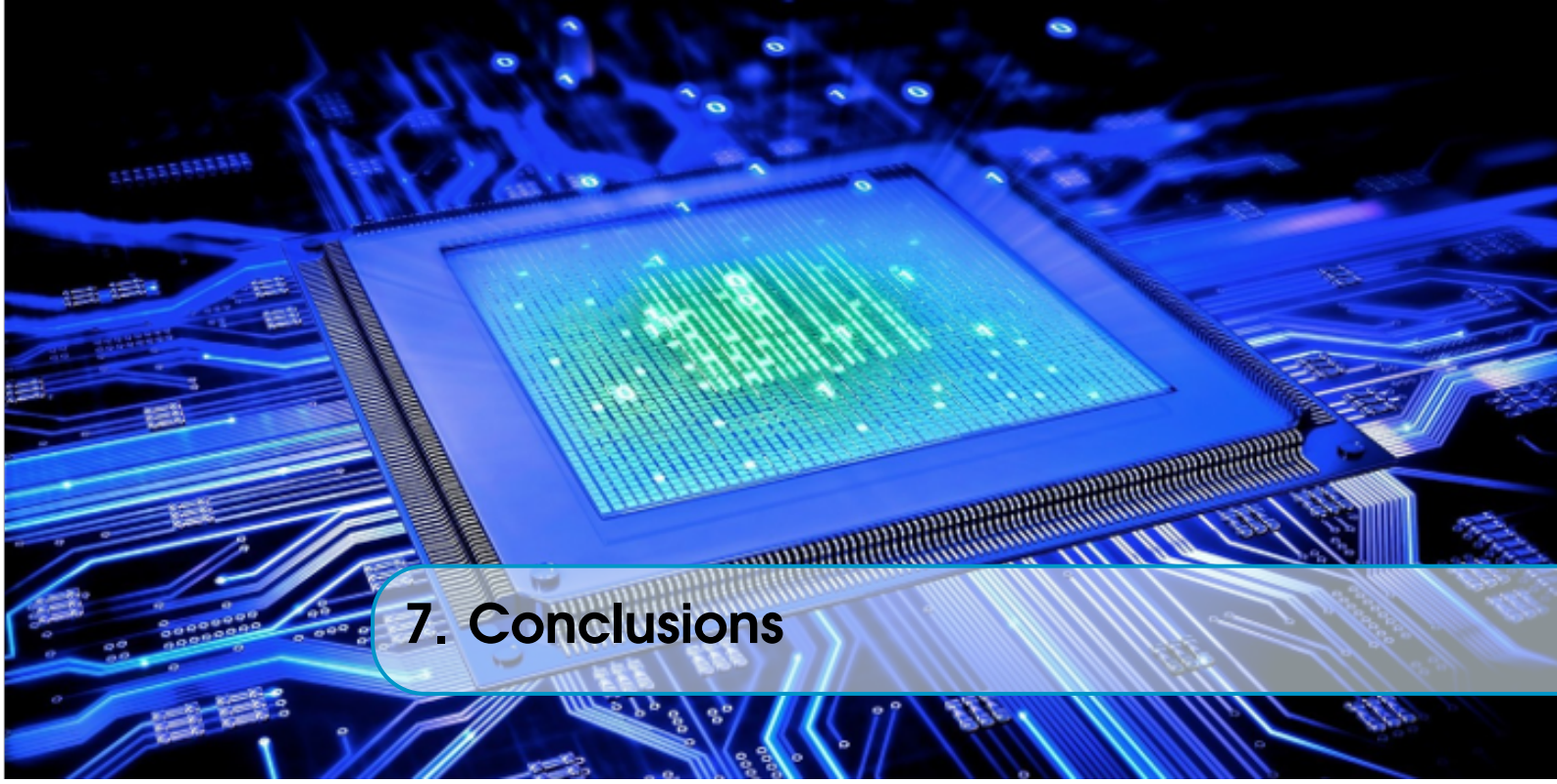


Figure 6.5 Lemonator overview.



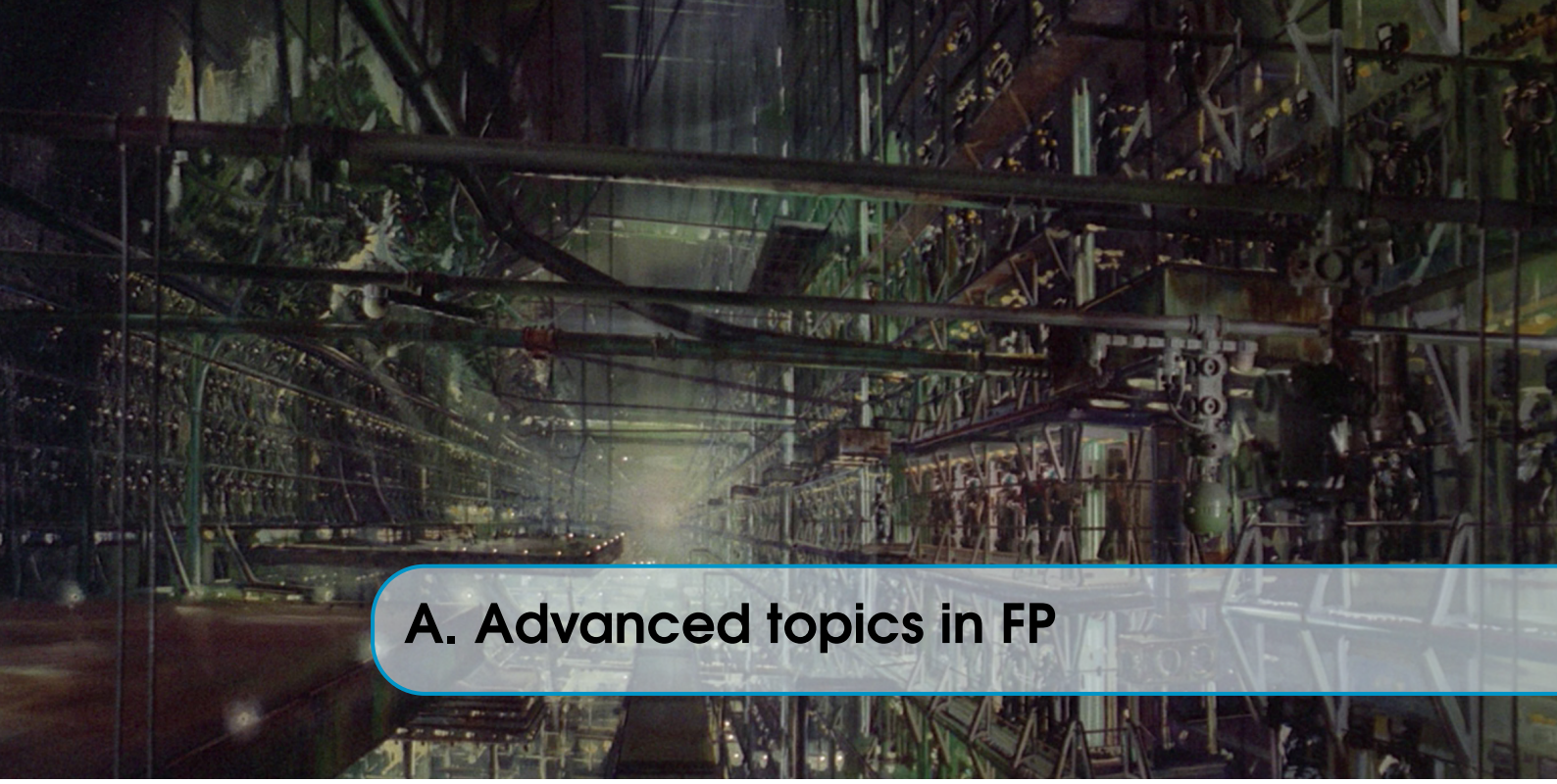
7. Conclusions

Developing for embedded systems is a special kind of software development, due to the limitations presented by the hardware that is being developed for. This means that the development of embedded systems requires a different kind of testing than typically exercised in software development.

Though the principles and steps are the same, special attention should be presented to the phases of development, and the phases of development at which tests are run.

While embedded systems often require the most performance and efficiency one can get from a target platform, that does not mean that the design of an embedded system has to be done in a performance oriented programming language as a necessity. Often it is wiser to design and develop (rapid prototyping) in a simpler language, to increase the speed of the design and developmend, before the software is ported later to a more performance oriented programming language in a later development phase.

In this reader we have discussed topics ranging from the testing of embedded systems, advanced programming concepts (related to Python), functional programming, testing incomplete systems, and relating prototypes written in Python with (partial) target implementations in C/C++ to assist in the transition between the prototyping and the deployment stages of development.



A. Advanced topics in FP

This section provides an overview of more advanced topics and recent innovations in Functional Programming, which aim to adapt the paradigm to new applications. The goal of this section to act as a reference when encountering these terms, or to serve as a starting point for further studies. The material in this section is not part of the exam. Consider this entire section the largest aside in the reader.

It is, however, recommended to work through the topics presented here in order (if at all), as subsequent ideas often build upon each other. If you choose to read a specific section only and encounter undefined terms, please refer to sections earlier in this chapter.

A.1 Type classes

For now, try to forget the OO concept of classes and instances. Despite having similar names, the type classes described here are more akin to *interfaces* or *traits* you may have come across in OOP languages, only more powerful.

Type classes are not really an advanced topic in and on itself, but has been moved to this section because there is no real equivalent in Python. Furthermore, some of the topics later in this section build upon type classes as their basis. Type classes are a concept in type theory used to facilitate controlled polymorphism. While no real Python equivalent exists (or is needed in Python's dynamic type system) it can be considered a more formal version of the aforementioned duck typing.

A type class describes a common behaviour (or set of related behaviours) present in multiple types. Types with similar functionality can be declared to be instances of a shared type class. To do so, they must implement the functions described by the class, after which these functions provide a more abstract interface to this behaviour. The type system no longer needs to know an exact type, just that it supports the functions it needs to call on the types.

As an example, let's look at the type class Monoid. This class derives from a mathematical concept with the same name which refers to an set of elements, a zero or *identity* element, and a $\oplus$ operation, obeying certain laws. The notion of $\oplus$ is more general than $+$, as certain properties such as commutativity[1] are not required. The only requirement is associativity, meaning $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ — the order of application doesn't matter. We can omit the parentheses, and both interpretations would be equal. To make it all a bit less abstract, consider a few examples:

Firstly, the natural numbers are a monoid with 0 as the zero element and $+$ as the $\oplus$ operator. We know that $(a+b)+c = a+(b+c)$, so everything checks out. At the same time, we can also view the natural numbers as a monoid with 1 as the zero element and $\times$ as the operator. Again, associativity checks out. But monoids are not limited to numbers: Strings, for example, also form a monoid. Here, the operator is concatenation (written $++$) and the zero element is the empty string `""`. There's a lot more where that came from, but the general gist with monoids is that they can be added together, there is a notion of an empty element, and multiple additions can be evaluated in any order. As a small exercise, consider lists (regardless of the type of element) — these are also monoids, but what are the operator and the zero element?

Now suppose we are writing a library, and want to perform logging. We can associate a string for the logging information, or we could choose a list of strings. But maybe it makes more sense for a user to log numbers, or arbitrarily complex data-structures. By utilising the Monoid type class, we leave this detail out. The user of your library can pick any type to use for logging, as long as it is a monoid. For lists and strings, the monoid type class is usually predefined (for numbers this often is not the case, as there is a choice between addition and multiplication), but any other type can be used, as long as it can be shown to be a monoid (by defining what the zero element and the operator are). In our library, we can then use the monoid function mPlus and the constant mZero instead of having to define functions for every possible log type that we want to support:

```
analyseDataStringLog :: ([[Float]], String) → ([Float], String)
analyseDataStringLog (data, log) = let result = applySomeFunction data
                                   in (result, log ++ createStringLogItem result)

analyseDataIntLog :: ([[Float]], Int) → ([Float], Int)
analyseDataIntLog (data, log) = let result = applySomeFunction data
                                in (result, log + createIntLogItem result)

analyseDataGeneral :: Monoid m ⇒ ([[Float]], m) → ([Float], m)
analyseDataGeneral (data, log) = let result = applySomeFunction data
                                 in (result, log `mPlus` createLogItem result)
```

This example analyseData function transforms some input data (represented as a list of lists of floats — a matrix or table) into some summary (represented as a list of floats) by calling applySomeFunction. In addition, the function receives a log and returns an expanded log. The type of the log can either be a concrete type (e.g. a String) or constrained to be any type of monoid (here the $\Rightarrow$ arrow means: provided that m refers to a type proven to be a monoid, it can be substituted into the right hand side to get a

[1] Commutativity means $a \oplus b = b \oplus a$, which is not necessarily the case for monoids.

valid type; m is a *type variable*).

As we do not need to know whether the log is a string (in which case we need to concatenate), or another structure (with another operation), we should define a general version using mPlus instead of several specific versions.

Keep in mind that if we go down this path and type our function using Monoid m, we *need to* use mPlus as well. Using, for example, the list append ++ here would not type check, as the operator is only defined for lists and the argument might be something else — the compiler only knows it to be some sort of monoid. On the other hand, we could for example use mPlus in our anlayseDataStringLog example (with the more concrete type), since a string is guaranteed to be monoid as well. In general, we can always use more general functions with concrete types, but we cannot use specific functions when our type is not sufficiently specific as well:

```
(++) :: [a] → [a] → [a]
mPlus :: Monoid m ⇒ m → m → m

listFunction :: ([a], [a], [a]) → [a]
listFunction (a, b, c) = a ++ b ++ c

monoidFunction :: Monoid m ⇒ (m, m, m) → m
monoidFunction (a, b, c) = a `mPlus` b `mPlus` c

alsoCorrect :: ([a], [a], [a]) → [a]          -- we know [a] is a monoid
alsoCorrect (a, b, c) = a `mPlus` b `mPlus` c -- so we can use mPlus

wrong :: Monoid m ⇒ (m, m, m) → m -- m might be [a], or something else
wrong (a, b, c) = a ++ b ++ c     -- so we cannot use (++)
```

A.2 Monads

Monads are a concept from the mathematical area of *category theory* (Pierce, 1991; Milewski, 2014) that forms an integral part in some functional languages and is unnecessary and awkward in most others. Category theory concerns itself with the structure of mathematics, ignoring *what* a function or a value is, instead focusing on *composition*: how the functions can be combined. As we have seen, the concept of functional programming centres on small pieces of declarative code which are combined to build more complex behaviour. This match is not incidental: category theory and functional programming share a lot of common history.

Monads can best be seen as a functional design pattern, akin to *factories* and *facades* in OOP. To understand monads, we take a look at a specific example: Maybe, which we have seen before as the most simple sum-type. We then look at how this behaviour generalises to a wide variety of situations. Due to the nature of this topic, our examples here are primarily in Haskell. This mirrors the use of monads in both languages: in Haskell, monads are an integral part of the language required for emulating imperative behaviour in a purely functional way, most notably for IO; in Python monads are mostly a curiosity, a nice way to structure a program but with limited practical use. Monads are implemented as type classes, with the following functions:

```
return :: Monad m ⇒ a → m a
»=     :: Monad m ⇒ m a → (a → m b) → m b
```

The type Monad m ⇒m a means that we have a container, which is a monad, containing something of type a. In general, what a is does not matter, as monads are only concerned with the structure. The return function simply puts something inside the monad. The »= operator is where the magic happens: given something which is already inside a monad, and a function whose result is in the same monad, we can extract the value from the monad, apply the function, and ensure the result is still in the monad. But why is this useful?

Consider Maybe, and how it can be a monad. In Haskell, Maybe has two constructors:

```
data Maybe a = Nothing | Just a
```

This is different from the `Optional` in Python, where Just a is just `a`, without any constructor. The constructor can be used in Haskell for pattern matching, so a function definition taking a Maybe could match the constructor to bind arguments and distinguish case:

```
maybeListLength :: Maybe [a] → Int
maybeListLength Nothing = 0
maybeListLength (Just list) = length list
```

Because Maybe is a monad, return provides a common interface to construct a Maybe value. As return requires a parameter, the most sensible thing for the function to do is turning a value into a Just value, e.g. return 42 :: Maybe Int[2] ⇝ Just 42. »= (henceforth also referred to as the bind operator) can be used to chain Maybe computations together, without having to do the manual pattern matching on each intermediate result. Consider a father :: Person → Maybe Person function, which returns the father of the person provided in its argument. Using this function, we can also determine a persons paternal grandfather, if such a person exists. Doing this without the bind operator requires manual unpacking of values:

```
grandfather :: Person → Maybe Person
grandfather p = let f = father p
                in if isJust f
                     then father (fromJust f)
                     else Nothing

grandfather' :: Person → Maybe Person
grandfather' p = father p »= father

grandfather'' :: Person → Maybe Person
grandfather'' p = return p »= father »= father
```

As you can see, the bind notation is a lot clearer, especially if you consider what would happen if more and more a → Maybe a functions are chained together. The bottom two examples are equivalent, but highlight different ways to structure even

[2]Type annotations can be used inline and will restrict the type as expected, but without the need to bind a variable. Here we ask for the result of return 42, but also specified that we want the result to be of type Maybe Int.

a simple example. In `grandfather''`, the flow moves nicely left to right, whereas the shorter `grandfather'` is arguably less readable.

The monad pattern can be applied in most instances where a function or data type gets data associated with it, which then needs to be moved around. Our logging example above could also benefit from being a monad:

```
functionA :: a → Logger b
functionA = undefined

functionB :: b → Logger c
functionB = undefined

doAllTheFunctions :: a → Logger c
doAllTheFunctions input = return input >>= functionA >>= functionB
```

Here, we no longer need to pass the log as input to our functions. Rather, this is handled by the bind operator. In reality, the `Logger` monad here is called `Writer`, and is parametrised on both the log type (which can be any monoid) and the output type of the function. It even provides a nice `tell` function to write to the log. Notice the `>>` operator, which behave kind of like `>>=`, but doesn't pass the result on to the next function. As `tell` has no result[3] beside the logging, and `return` uses the `input` variable as its input, this works out (and wouldn't work with `>>=`, as then `return` would have both `input` and `()` to deal with.

```
functionA :: a → Writer String b
functionA = undefined

functionB :: b → Writer String c
functionB = undefined

doAllTheFunctions :: a → Writer String c
doAllTheFunctions input = tell "Doing all the functions!"
                       >> return input >>= functionA >>= functionB
```

Aside A.1 — Monad Laws.
The definition of monads as implementing two functions (bind and return) is not entirely complete: In addition to implementing the above two functions, there is also a set of equalities which must be obeyed. This responsibility belongs to the programmer defining the monad, but is nevertheless good to know. The existence of these rules ensures monads behave as expected, without nasty surprises. The three monad laws are:

```
return a >>= f   = f a
m >>= return     = m
(m >>= f) >>= g  = m >>= λa → f a >>= g
```

The first law allowed us to write two different versions of the grandfather functions using monads. The second law is equally simple to understand: using bind to extract

[3] In fact, it has, because every function has a result. The result is `()`, which does not contain any information whatsoever. Its type is also `()`, and the only value that type can take is `()`. So while there is a result, it's just not particularly useful.

the value from the monad, and then using return as the `a → m b` function (essentially repackaging the value inside the monad) accomplishes nothing whatsoever. The third law is a little bit more of a mouthful, but simply defines that when multiple binds are chained together, the result is what we'd intuitively expect: the value is passed through both functions, one after the other. Together, these three rules ensure monads work as expected, which in turn allows for syntactic sugar[a] such as the do-notation.

[a] Stuff added to programming languages not to add functionality, but to increase programmer quality of life.

Aside A.2 — do-Notation.
Because a function like doAllTheFunctions can get a bit hairy, Haskell supports a special type of notation to make this more readable, and facilitate the passing around of variables:

```
doAllTheFunctions = do
  tell "Doing all the functions!"
  a ← functionA input
  b ← functionB a
  return b
```

This way of writing code looks more familiar to those who are used to imperative programming. The fact that the function to put a value in a monad is called return is because of this notation, where do blocks often end in a return statement to package the result. Note that return does not actually exit the function: you could enter a return anywhere in the block before the final line without changing anything. The compiler converts the above notation to a mess of binds and lambda functions, which is not pretty.

A second use of the monad is to hide implementation details. As you can see from the Logger / Writer example, it is not necessary to know how exactly your log gets built, only that it does so in a predictable manner. For this reason, Haskell also uses monads to contain IO: any value obtained by IO is always packaged in the IO monad, and no function is provided to extract the value. The only way to interact with a value in the IO monad is the bind operator and the ← operator in a do-block, both of which require the result to packed inside the IO monad again.

A.2.1 Monads in Python

As we have seen, the monad has two general applications: hiding implementation details and boilerplate code, and chaining computations. As Python is primarily an imperative and object-oriented language, there are often more idiomatic ways accomplish the same effect in it. Monads support adding some features from imperative languages in a purely functional way, which only matters when you are (or your compiler is) concerned with keeping everything purely functional. When programming a multi-paradigm language such as Python, there are often ways better suited for the language. Nevertheless, should you ever continue your journey towards the black belt of functional programming, you will likely encounter monads quite early, but at least

now you are prepared. Monads are only a single part of a large family of type classes containing scary words such as applicatives, functors, transformers and arrows, yet monads are probably the most widely used abstraction.

Should you really want to play around with monads, you can use the *PyMonad*[4] library, which brings a lot of data abstractions from FP to Python.

A.3 Continuations

In normal programming practice, functions return a result when they are done with the task at hand. In Continuation-passing style (CPS) functions do not return, they continue. Each function gets an extra argument glued to the back, which will be passed a function that will continue after the function has finished:

```
def double(num : int) → int:
    return num*2

def double_CPS(num : int, continuation : Callable[[int], A]) → A:
    continuation(num*2)
```

Continuations can be used in any place where normal ("direct") functions are applicable. Some situations, however, call for continuations as a clearer way of expressing ideas. Most do not. One area where CPS is of interest is search algorithms, where multiple continuations are passed to a function, e.g. `success(intermediate_res, fail)` and `fail` for partial success and failure respectively. If the conditions for the search remain achievable, control passes to the success continuation, which takes both an intermediate result `intermediate_res` and the failure continuation. If at any point the conditions are determined to be unsatisfiable, the failure continuation is called with no further arguments. This process repeatedly calls the success continuation until either a failure is reached, or until the search is finished — an identity continuation (`lambda x: x`) or `return` statement is then called as the last continuation, which finally breaks the cycle. This way, continuations provide more explicit control over program execution, which can be considered a form of syntactic semtex[5]. Continuations allow for jumps and early returns, and are sometimes compared to "cleaner" `gotos`.

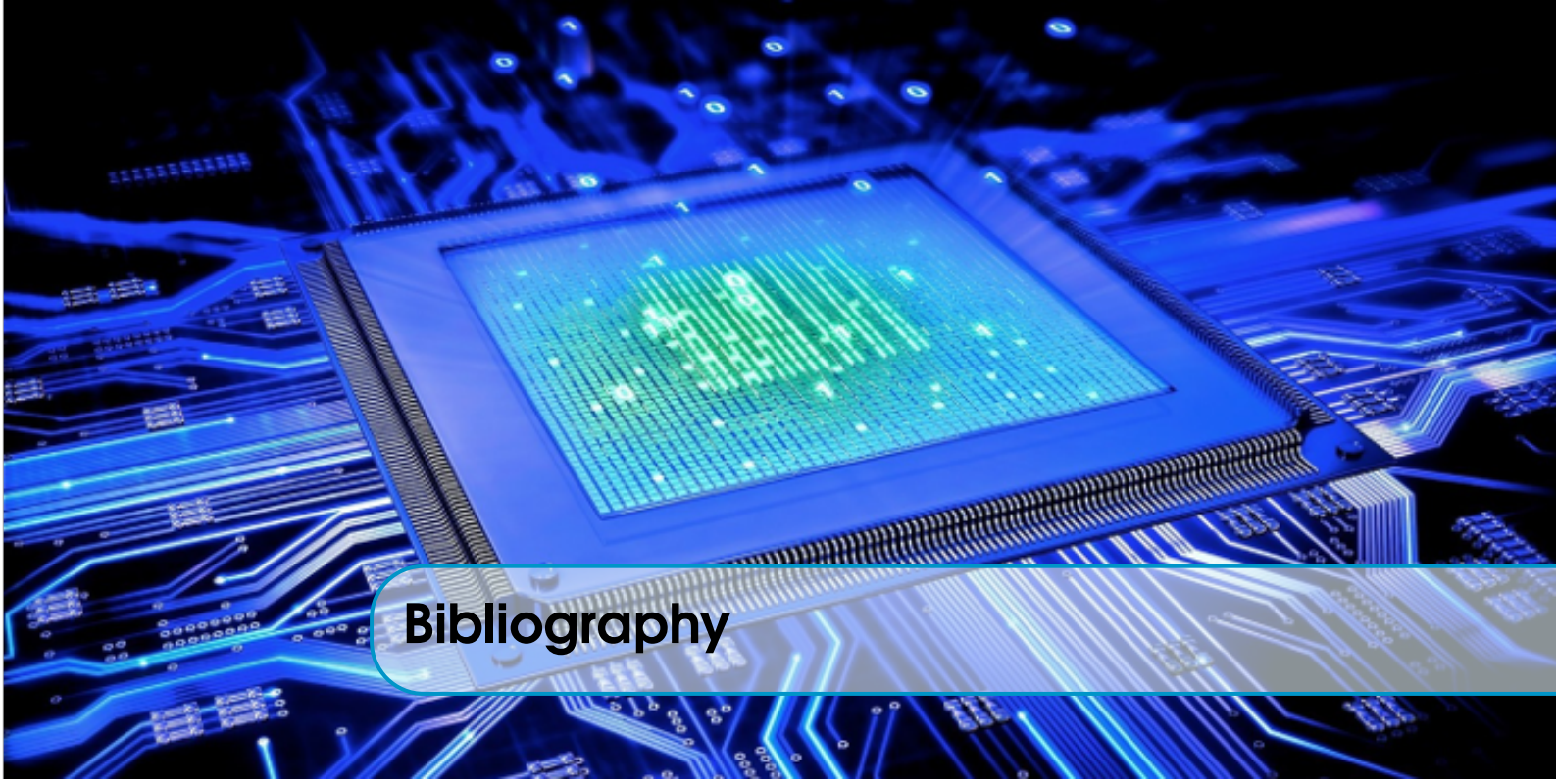
Aside A.3 — Continuations in Compiler Technology.
Most functional compilers actually use CPS transformation as its basis. This generally introduces a large mess of nested lambdas (closures), but as functional compilers often reduce everything to lambdas anyway, this actually optimises nicely. CPS naturally leads to all calls being tail calls, which also benefits from tail call optimisation.

A.4 Dependently typed Functional Programming

Even though functional languages like Haskell have a pretty solid type system, there

[4] https://pypi.python.org/pypi/PyMonad
[5] Exceptionally powerful in skilled hands, likely to blow your legs off for everybody else.

is still more we can do to make compilers indistinguishable from magic. Recently, languages have begun to develop which are dependently typed, which means the types can be manipulated by the language itself. Type systems such as Haskell's feature some pretty complex machinery, allowing simple problems to even be solved on the type level without requiring a line of actual code. Still, what if we had the full power of the entire language at our disposal in the type system? This is what dependently typed FP gives us, and this allows the type-checker to infer types dependent on their contents, such as whether a list is empty or not. Even in Haskell, taking the first element of an empty list will raise a runtime error. In a dependently typed language such as Agda or Idris, code performing this action simply would not compile, moving even more potential bugs from runtime to compile time where they can be solved before the user discovers these unintended features.



Bibliography

Beizer, B. (1990). *Software Testing Techniques*. International Thomson Computer Press.

Binder, R.V. (2000). *Testing Object-Oriented Systems: Models, Patterns and Tools*. Addison Wesley.

Boehm, B.W. (1981). *Software Engineering Economics*. Prentice Hall.

Broekman, Bart and Edwin Notenboom (2003). *Testing Embedded Software*. Essex, Great Britain: Addison Wesley.

Erpenbach, E., F. Stappert, and J. Stroop (1999). "Compilation and timing of statechart models for embedded systems". In: *Cases99 Conference*.

Grenning, James W. (2011). *Test-Driven Development for Embedded C*. Pragmatic Programmers.

Kuhn, D.L. (1989). "Selecting and effectively using a computer aided software engineering tool". In: *Annual Westinghouse computer symposium*. DOE Project.

Milewski, Bartosz (2014). *Category Theory for Programmers*. URL: `https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/` (visited on 07/16/2017).

Pierce, Benjamin C (1991). *Basic category theory for computer scientists*. MIT press.

Pol, M., R. Teunissen, and E. van Veenendaal (2002). *Software testing: a guide to the TMap® Approach*. Addison-Wesley.

Reid, S. (2001). "Standards for software testing". In: *The Tester (BCS SIGIST Journal)* 2-3.

Reynolds, M.T. (1996). *Test and Evaluation of Complex Systems*. John Willey and Sons.

RTCA/DO-178B (1992). *Software Considerations in Airborn Systems ad Equipment Certification*. RTCA.

Schaefer, H. (1996). "Surviving under time and budget pressure". In:

Spillner, A. (2000). “From V-Model to W-Model – Establishing the Whole Test Process”. In: *Conquest 2000 Conference Proceedings*. ASQF.

Thompson, Simon (1991). *Type theory and functional programming*. International computer science series. Addison-Wesley, pp. I–XV, 1–372. ISBN: 978-0-201-41667-1.

Wikipedia (2017). *C3 linearization*. URL: `https://en.wikipedia.org/wiki/C3_linearization` (visited on 08/24/2017).