Specification based OO development: Procedural Guidance

OUNL-CS Technical Report, No 6, 2022

Harrie Passier & Lex Bijlsma & Ruurd Kuiper
With the cooperation of:
Greg Alpar
Niels Doorn
Stijn de Gouw
Cornelis Huizing
Harold Pootjes
Stefano Schivo

Co-funded by the Erasmus+ Programme of the European Union

Contents

1 Introduction **5**

2 Overview of the approach **7**
2.1 The artifacts . . . 7
2.1.1 Modularity . . . 7
2.1.2 Views . . . 8
2.2 The activities . . . 11

3 Aspects of the approach **14**
3.1 Principles, methods and methodology . . . 14
3.2 Flexibility in level of formality . . . 14
3.3 Flexibility in development order . . . 15
3.4 Flexibility of application of the approach . . . 16

4 Concepts **18**
4.1 Requirements . . . 18
4.2 Analysis . . . 19
4.3 External versus internal . . . 19
4.4 Design versus specification . . . 20
4.5 Happy path and non-happy path . . . 21

5 External View **22**
5.1 External Analysis . . . 22
5.2 External Design . . . 23
5.3 External Specification . . . 24

6 Internal View **28**
6.1 Internal Analysis . . . 28
6.1.1 Representation . . . 28
6.1.2 Attributes and private methods . . . 29
6.1.3 Functional decomposition and helper objects . . . 29
6.2 Internal Design . . . 30
6.3 Internal Specification . . . 31
6.4 Correspondence between External and Internal Specifications . . . 33
6.4.1 Proof rules . . . 33
6.4.2 Class invariants . . . 34
6.4.3 Representation relations are not necessarily functional . . . 34
6.5 Subspecifications . . . 34

7 Annotated Code View **37**
7.1 Code Analysis . . . 37
7.2 Code Design . . . 37
7.3 Coding . . . 38

8 Testing **40**
- 8.1 Introduction . . . 40
- 8.2 External Tests . . . 40
  - 8.2.1 Step 1: Make a model . . . 41
  - 8.2.2 Step 2: Choose a coverage criterion . . . 41
  - 8.2.3 Step 3: Design test cases . . . 42
- 8.3 Internal Tests . . . 43
  - 8.3.1 Step 1: Indicate extra test cases . . . 43
  - 8.3.2 Step 2: Determine how to test the extra conditions and private helper methods . . . 45
- 8.4 Code Tests . . . 46

9 Robustness **49**
- 9.1 Introduction . . . 49
- 9.2 Robust versus non-robust software . . . 50
- 9.3 Making software robust . . . 50
- 9.4 Overlapping subspecifications . . . 52

10 Systems consisting of more than one class **54**
- 10.1 Auxiliary classes originating in external design . . . 54
- 10.2 Example: Shortest path . . . 55
  - 10.2.1 Assignment . . . 55
  - 10.2.2 External Analysis . . . 55
  - 10.2.3 External Design . . . 55
  - 10.2.4 External Specification . . . 56
  - 10.2.5 Internal Analysis . . . 57
  - 10.2.6 Internal Design . . . 58
  - 10.2.7 Internal Specification . . . 58
  - 10.2.8 Code Analysis . . . 59
  - 10.2.9 Code Design . . . 59
  - 10.2.10 Coding . . . 60
  - 10.2.11 Robustness . . . 61
  - 10.2.12 Test construction . . . 62
- 10.3 Auxiliary classes originating in internal design . . . 63
- 10.4 Example: Rental agency . . . 63
  - 10.4.1 Assignment . . . 63
  - 10.4.2 External Analysis . . . 63
  - 10.4.3 External Design . . . 63
  - 10.4.4 External Specification . . . 64
  - 10.4.5 Internal Analysis . . . 65
  - 10.4.6 Internal Design . . . 65
  - 10.4.7 Internal Specification . . . 66
  - 10.4.8 Robustness . . . 67
  - 10.4.9 Coding . . . 68
  - 10.4.10 Test construction . . . 68
- 10.5 Developing larger OO systems . . . 69

10.6 Test execution for multi-class systems 70

A For the teacher **71**
A.1 How to apply the procedure in a curriculum? 71
A.2 Didactic aspects 71

1 Introduction

This book presents an approach for specification based object oriented (OO) development, including all corresponding tests, exemplified on, but relatively independent of, Java. Besides the conceptuel knowledge needed, the procedural knowledge [11] of how to develop the specifications, implementation and tests is treated. The description of 'how to' is what we call Procedural Guidance. The guidance scaffolds students to achieve proficiency in a flexible process that installs quality awareness and improves software quality. In the development approach specifications, code and tests are developed in an integrated fashion: the focus is on specifications as crucial to achieve quality.

Important ingredients in software quality assurance are artifacts like annotated code, where annotations include specifications, and tests. To obtain these artifacts, a structured development method is a natural way to proceed. Tools may support such an approach. All of these exist in various concrete forms; the challenge is to foster their use in a process that enhances software quality. This is where Procedural Guidance comes in.

The right piece of guidance for the development of the artifacts must be provided to the student at the appropriate point in the development. In classroom education and even more so in current online education, where instructors and students are less directly interacting, this is challenging. A distinguishing feature of our approach is that the artifacts are made explicit and that for each of the artifacts corresponding activities that produce these are identified. Procedural Guidance is then given as to when and how to perform the activities to assure the quality of that artifact. This in contrast with classical approaches that merely list obligations for code and tests, and leave it to the student to fulfill these.

The Procedural Guidance is quite detailed, but not tied to a specific, directive development method. E.g., there is guidance as to what the relationships between the various artifacts are and at which point in the development activities are performed, and how, but the order in which activities are performed is quite flexible.

The Procedural Guidance is an educational device: it guides the student towards proficiency in the development process, and then, when internalized, can be regarded as scaffolding and left behind.

The presentation of the approach aims to enable a teacher to use the Procedural Guidance with existing courses. The book can also be used by students to assist in the understanding and application of the approach.

The book is organized as follows. In the Overview section 2 the role and consequences of Procedural Guidance for the program development process are introduced, without going into the specific, concrete guiding procedures that are the subject proper of the book. In particular, in this section the artifacts, the activities and the relationships between them are briefly introduced. Section 3 addresses some aspects of the approach and section 4 discusses some concepts used. Sections 5 up to and including 8 present the Procedural Guidance, exemplified on, essentially, the development of one class including testing.

The approach also addresses robustness, the ability of programs to cope with unintended situations during execution. This is done in Section 9. Section 10 extends the approach to structures of more classes. The book concludes with an Appendix A which contains suggestions for teachers as to how to apply the approach in an educational setting.

2 Overview of the approach

Procedural Guidance for specification based OO development is the subject of this book.

A program asked for in a programming assignment is obtained as the result of program development. This is a complex, difficult process that requires knowledge, skill and creativity.

Object-oriented programming (OOP) is a programming paradigm that enables object-oriented development that deals with complexity in a specific, effective manner, using the structuring provided by OO languages.

Procedural guidance is guidance to achieve proficiency in some complex process towards some complex goal. The guidance is procedural in that it does not address the process at a conceptual level but makes explicit what the relevant activities in the process are and provides stepwise advice for performing these. The aim is to achieve proficiency in the process, as internalized skills; the guidance can then be left behind as temporary scaffolding. Procedural guidance is different from a method: the guidance engenders general, flexible proficiency in a process, whereas a method is more prescriptive, providing precisely defined operations, to be carried out in a fixed order to achieve specific goals. Procedural guidance is largely independent of specific methods: a method can benefit from and further structure the application of skills acquired through the guidance.

Subsection 2.1 introduces the artifacts into which program code, descriptions of behavior and tests that figure in the development are organized, and the relationships between them. Subsection 2.2 introduces the activities that produce the artifacts, and the relationships between them. This section also provides some idea about how procedural guidance applies to navigate between activities (macro guidance) and, inside activities, how to perform steps in developing the various artifacts (micro guidance).

2.1 The artifacts

A program for a programming assignment, code, must, when run, provide behavior as required in the assignment. The development of a program involves at various stages in the development intermediate code, specifications of required behavior of code, and establishing that behavior of code satisfies specifications. Satisfaction can be established in various ways: the approach focuses on tests.

In the approach artifacts are developed that lead to a program for the assignment. In object-oriented development two ways to deal with complexity are modularity and views: these underly the choice and form of the artifacts.

2.1.1 Modularity

Modularity deals with complexity by partitioning a program into parts have high cohesion within parts and low coupling between parts, and hence can each be developed relatively independently. The limitation of the interaction is enforced through encapsulation.

OOP employs the modularity provided by OO languages to design a program as composed of parts that are high in cohesion and low in coupling: classes. In OOP a program is a set of classes. When a program is run, from the classes objects are instantiated that are linked by references and collaborate through method calls. Classes are further structured by decomposition of storage, in attributes, and decomposition and abstraction of manipulation, in methods. Methods also enable abstraction of storage. Encapsulation is provided through access control: scoping and access modifiers.

The approach exploits the OO modularity by taking the class as the unit of development and using for its development the structuring OO provides.

For each class there are in the development specifications that describe required behavior, code that, when run, produces behavior and tests that test whether code produces specified behavior. The specifications are attached to the corresponding pieces of code, i.e., to the class as a whole and to methods. Hence the specifications are called annotation of the code.

The first artifact, for each class `X`, is therefore the class itself.

- `X` consists of annotated code (AC).
  - Code (C), structured using OO syntax.
  - Annotation (A), specifications, integrated in C, attached to the corresponding pieces of C, following the structure of C.

To establish that C satisfies A, tests are provided:
The second artifact, for each class `X`, is therefore the test class `XTest`.

- `XTest` consists of tests (T).
  - For each specification in the AC test methods that test the corresponding code.
  - The test method are JUnit test methods, with test values and the required result value, that return a verdict.

So for each class there are two artifacts, referred to as AC+T. T and AC are separate artifacts, T are run on C.

2.1.2 Views

Orthogonal to the modularization, views deal with complexity by presenting what, and in which terms, at increasing levels of detail is conceived about the program.

OOP employs three views on the AC+T: the External View, corresponding to the perspective of an external user (the functionality provided by the program to a user or re-user of the class), the Internal View, corresponding to the perspective of a developer (the internal OO structure of the program), and the Annotated Code View, corresponding to the perspective of a writer of code (the statement, including method calls, level).

The approach exploits the views to develop the AC+T for a class incrementally.

For the AC, the three views are three artifacts, called by the name of the view: External View, Internal View and Annotated Code View. During the development the three artifacts build upon each other: incrementally and iteratively the AC is developed. The three artifacts are therefore developed as just one, growing, artifact. When the AC is complete, the views can be regarded as projections of the AC. It is quite possible to jump forward and backward between views during the development. The only restriction is, that development of a view cannot be considered finished before the view it builds on is finished, i.e., the External View must be finished before the Internal View can be finished, and the Internal View must be finished before the Annotated Code View can be finished.

The three artifacts for the AC are as follows.

- **External View.** The start of the development of the AC
  - Signatures of the public methods.
  - Specification of the behavior of the methods in terms of the domain (and the parameters and returns).
  - Invariants for the domain variables.
- **Internal View.** Added to the AC so far are the following:
  - Attributes.
  - Signatures of all methods.
  - Specification of the behavior of the methods in terms of the attributes and parameters.
  - Invariants for the attributes.
  - Representation relations between attributes and domain variables.
- **Annotated Code View.** Added to the AC so far are the following:
  - Code and annotation for the methods in terms of attributes and local variables (including method calls).

For the T, the three views also lead to three artifacts. Tests apply to the code and the specified behavior as provided in the corresponding view on the AC. The three artifacts are called External Tests, Internal Tests and Code Tests. Again, during the development the three artifacts build upon each other: incrementally and iteratively the T is developed. These three artifacts build upon each other but are developed with the information in the corresponding view on the AC. Hence the External View must be finished before the External Tests can be finished, the Internal View must be finished before the Internal Tests can be finished, and the AC must be finished before the Code Tests can

be finished. Again it is quite possible to jump forward and backward between artifacts during the development.

The three artifacts for the AC are as follows.

The additional tests are tests for additional methods, e.g., private methods in the Internal View that were not present in the External view. Additional test can also be extra or more stringent tests for methods that were already tested: a specification for a method may in the Internal View well be more demanding than in the External View. Furthermore, in the Annotated Code View code information about crucial features of the code of a method may be reason for adding specific test cases to the code test for that method.

- **External Tests.** Start of the T:
  - For each specification in the External View test method for that specification.
- **Internal Tests.** Added to the T so far are the following:
  - For each specification in the Internal View a test method for that specification. In case such a test method already exists in the External Tests it may be extended with additional test cases to reflect a sharpening of the specification.
- **Code Tests.** Added to the T so far are the following:
  - For each specification in the Internal View a test method with tests for that specification. The test method in the Internal Tests may be extended with additional test cases to provide code coverage and to check preconditions of called methods.

Figure 1 shows the six artifacts and the needs relation between them: $x \to y$. y needs x, if y can only be finished after x is finished.

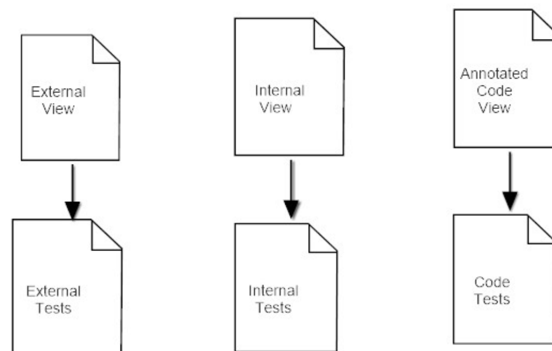

Figure 1: The Artifacts

2.2 The activities

The activities are where the detailed Procedural Guidance for their construction is provided. The activities are related in several ways to artifacts, and there are also relationships between activities themselves.

For the construction of the External View and the Internal View we distinguish the activities Design (structure, signatures) and Specification (behavior, specifications, invariants) for each of them, for the construction of the Annotated Code the activities are Design (choice of algorithm, local variables) and Coding. The Tests are build in a standard manner: a test methods for each specification, therefore the activity for each of the test artifacts is just Construction.

Before embarking on the activities that produce artifacts, Analysis is required, e.g., as to what exactly is intended by the Assignment, or what information and ideas are needed for performing an Activity. This is indicated by Analysis, which yields no artifact but prepares for each activity that does yield the Artifact. Analysis is quite important: think before acting!

There is one more activity, Run Tests, which is the only activity that does not result in a document, an artifact. It does however change the status of the Tests artifacts, namely that they are executed. The AC+T is the solution to a programming assignment only if all Tests are satisfied.

Figure 2 shows an overview of the approach in terms of the artifacts, activities and, represented by arrows, the relations between them.

The explanation of the symbols in this diagram is shown in Figure 3.

Figure 2 shows all of the activities that are in principle to be performed to develop software. The three construction artifacts evolve, through the activities, from each other, they are grouped horizontally in the figure. The three Test artifacts primarily apply to and hence are based on (need), the corresponding program artifacts; each of them, with the corresponding construction activity , is placed below the artifact it applies to. However, the Tests also evolve from each other in the sense that they access progressively more details of the Code. Furthermore, all tests are to be executed on the Code.

It is a crucial feature of the approach, that the activities provide information as to how to develop an artifact but do not prescribe an ordering. It is quite feasible when performing one activity, it is opportune to leave that one and proceed with another, and later come back to the original one. An example is, that when Coding, the need for an extra method becomes apparent, which leads to a temporary return to the Internal View. Another example is, that when developing a test in one of the Test Construction activities, this might lead to an insight that a specification in a Specification activity should be changed. These are examples of macro guidance: how to navigate between activities and artifacts.

Micro guidance concerns what to do inside an activity. An example is, that to start a development in the External Analysis it has to be decided which the public classes are.

The three views are represented in the figure as columns. In case of a simple development task, like a function to develop, some of the decisions and actions to

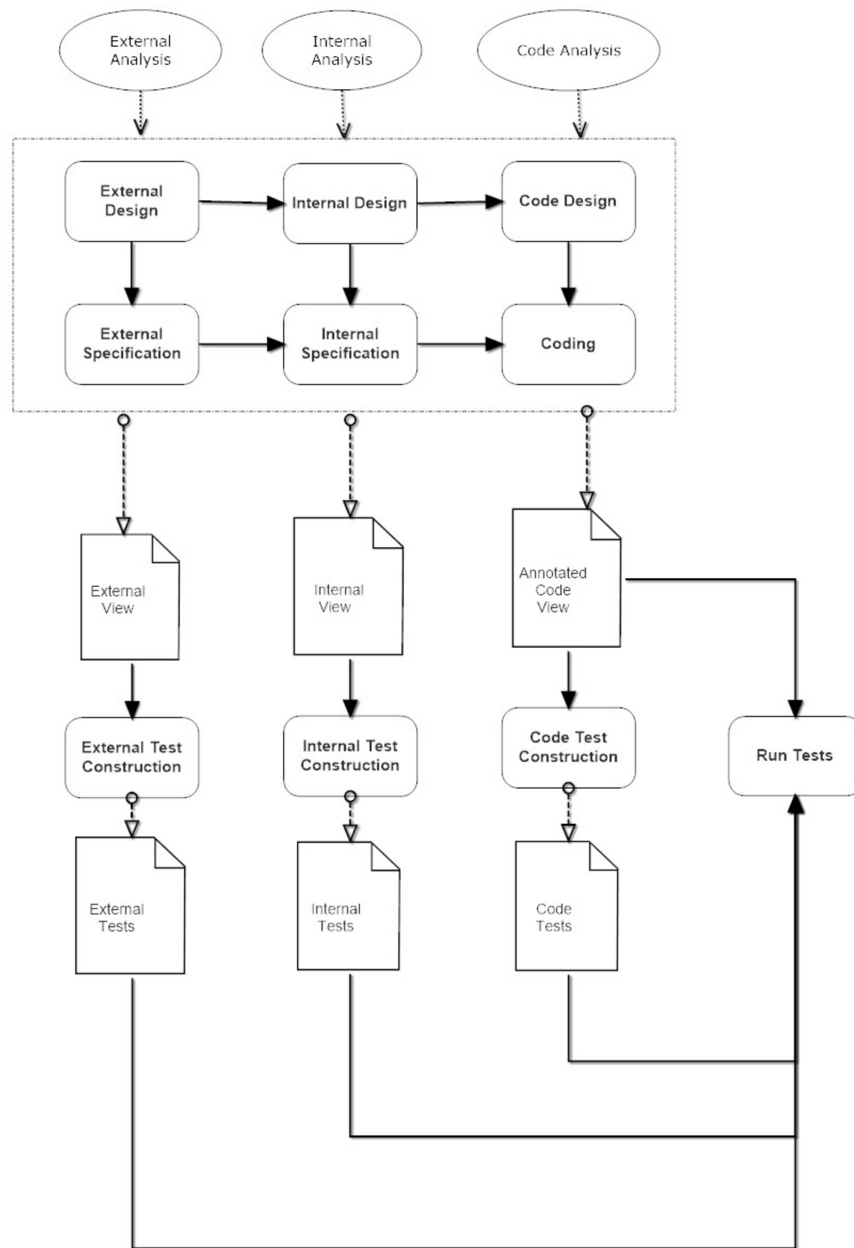

Figure 2: Overview of the approach

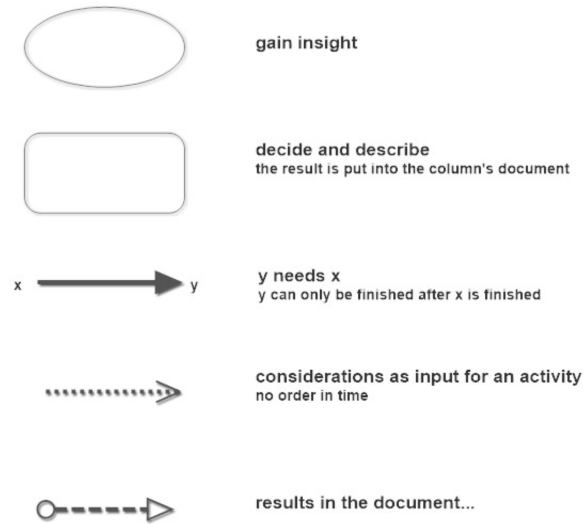

Figure 3: Meaning of symbols

take are very simple and hardly require explicit consideration: the first and last column suffice. But in the case of complex software, all of them, and especially the middle column, are important: the Procedural Guidance guides a developer along all the decisions to make and the order in which they can best be made.

3 Aspects of the approach

In this section, we discuss some typical aspects of the Procedural Guidance. The Procedural Guidance describes the steps through which a piece of functionality is developed, what has to be done per step, how the results are laid down and what the relationship is between the steps. However, this does not mean that the Procedural Guidance enforces a particular order of the steps, or that a particular (formal) notation must be used. The procedural guidance allows flexibility: both the notation (informal/formal), the order of the steps, and even which parts of the guidance are applied can be determined freely.

3.1 Principles, methods and methodology

The procedure aims for general known principles as rigor, appropriate formality, separation of concerns, decomposition, abstraction, and modularity. But satisfying these principles is no sinecure. Software engineers should be equipped with appropriate methods and techniques [3]. Methods are general guidelines that govern the execution of some activity. Techniques are naturally more mechanical and have more restricted applicability. Methods and techniques can be packaged together to form a methodology: it promotes a certain approach to solve a certain category of problems by preselected methods and techniques to use.

From a software engineering perspective, the procedure in this monograph forms a methodology: a series of activities (steps) is described that can be executed in some orders. For each activity rules of thumb, hints, standard questions, decisions to take, and notations are described. The goal of this methodology is: 'getting quality software', i.e. software that satisfies the fundamental principles mentioned. Actually, all the methods and techniques used are known. What is new, is that they are put together in one coherent and teachable software development methodology.

The steps and the order of these steps exhibit some rigor: For each step it is clear what to do and when the step can be finished. The techniques used within each step lean on formal notations, but we use them in an informal way. The steps give the possibility to test during the development at multiple points in the process.

3.2 Flexibility in level of formality

Nothing in the above mandates that either the External View or the Internal View be written in a formal notation derived from logic or mathematics. One may do so if it is advantageous and easily readable: writing $a + b$ is less work and clearer than 'the sum of a and b'. But there should be no obligation if one does, for example, not aim for automated verification.

A specification records the precise requirements about a program, and, as a consequence, also what should be tested. It is important to observe that precision and formality are not the same. The first can be obtained without the

second, e.g., by describing all requirements in unambiguous natural language. The second does not automatically provide the first, e.g., when an formal specification is not complete. For example, the specifications in Meyer's book [12] are formal Eiffel expressions, but this limits the expressiveness, as no quantifiers are available.

Without going into the details of our approach, we elaborate a little on these notions.

The main difference between formal and informal specifications is that the former are amenable to automated checks: on syntactic or semantic aspects of the specifications themselves, on the relationships between them and on the relationship to the implementation. In case of informal language, human interpretation is necessary to interpret the specifications.

The external specification, as part of the External View, is in terms of domain entities, therefore no single formal language can be given. Nevertheless it is possible to provide formal languages with associated tooling for specific domains - this means that several formalisms and tools may be used.

The internal specification, as part of the Internal View, will be partially in terms of implementation entities. In some cases, it may be feasible to eliminate domain concepts entirely and express the resulting contract in a formalism. This means that one domain independent formalism and toolset can be selected. However, this is not mandatory and the level of formality may be freely decided upon by users of our guidance.

It should be noted that the formal/informal choice is only relevant with respect to specifications. The design is formal in the sense because method headings satisfy the rules of the programming language, and of course so does the code. In this context it may be observed that UML, although formal, is not a suitable language to record the kind of specifications we mean. In our process, UML plays a role in recording and visualizing the software's design, but we will use a flexible JML-influenced style for specifications.

It is important to realize that also without having automation as an aim formal specifications can be very useful: they provide help to attain precision in specifying because the meaning of the language is unambiguous. An example of this is the use of formal invariants as an ingredient of informal specifications.

Similar observations apply to tests: by nature a test, written in a computer language, is formal. The tests we aim at will only consist of calls to JUnit-provided methods such as `assertEquals`, so there is no need for a separate design or specification of the tests. In fact, the case analysis involved in specifying helps enormously in determining the test cases needed. We consider this a major asset of such a process.

3.3 Flexibility in development order

The fact that our guidance consists of a number of partially ordered steps might, at first sight, create the illusion that we are proposing something resembling the waterfall development method. However, this is not the case. The steps are applicable to every programming activity that results in a compilable, testable

code fragment. Typically, this will be the result of a single iteration in an incremental and iterative development such as the Unified Process [5]. In fact, even for a single use case the steps will be taken repeatedly, as we first build the basic functionality and subsequently address concerns such as robustness, security and concurrency.

The three artifacts need not necessarily be completed in sequence! For instance, it is quite possible to make a choice for the private methods and attributes as soon as the public ones have been defined, without first writing the external specification. Furthermore, the drawn arrows in Figure 2 should not be interpreted as a sequential order: for instance, useful work can be done on specification before the design is complete. What they do express is that the downstream activity cannot be considered completed before its predecessors are: only during the last pass the arrows must be followed. This means that changes in the design may force changes in or additions to the specification, so the specification cannot be considered finished as long as the design is still under construction.

In essence, our guidance is intended to provide a stepwise transformation from requirements to the finished software unit, where every phase in this transformation process contributes to the test suite. In this way, test development proceeds hand in hand with software development [2]. This process may be regarded as an instance of transformational programming [13, 15], but it is less formal and has more emphasis on testing, less on code derivation than the existing literature in that field.

One set of steps in our process concerns itself with the External View and produces a specification exclusively in terms of the application domain. This domain-based External View already serves as the basis for tests, which can be defined before any implementation decisions have been taken [1]. It is only in a later phase that attributes to represent the domain concepts are selected: this activity is traditionally known as data refinement [14].

The stepwise development procedure we are advocating should not be confused with the notion of stepwise refinement [16], as that is mainly concerned with program decomposition. In our procedure this occurs during a single step we shall call Internal Design; but the complete procedure entails many more activities as analysis, specification, data refinement, and test development.

3.4 Flexibility of application of the approach

We have already stated that one set of activities in our process is concerned with the software as it appears to those who will use and possibly extend it. The end product of this is the External View, which consists of the signatures and specifications of public methods and the classes containing these. The level of formality can be freely chosen as appropriate to the project at hand. However, when we wish to express something for which JML [8] has a convenient notation, we adopt it in order to facilitate migration to a more formal approach at a later stage. For instance, preconditions and postconditions of methods will be denoted JML-like by `@requires` and `@ensures` respectively.

It depends on the nature of the project whether the External View, Internal View or Code will present the greatest difficulty and leads to the most momentous decisions. There are problems where the essential difficulty is to find out the precise requirements (External View), others where selecting appropriate data types is crucial (Internal View), and still others where algorithm design is the core (Code). It is also quite possible that some steps are essentially empty: for instance, in cases where the external specification is already expressed in types provided by the programming language, it may not be necessary to specify their internal representation. Also for didactic reasons, it can be decided to only introduce the External view in a first course about programming and not to pay any attention to the Internal View yet, which can be treated as part of a course about data structures and algorithms.

4 Concepts

In this section, we describe the concepts that are crucial in our method. The following concepts are discussed:

- Requirements
- Analysis
- External versus internal
- Design versus specification
- Happy path and non-happy path

4.1 Requirements

Definition 4.1 A *requirement* is a prescriptive statement to be enforced by the software-to-be, possibly in cooperation with other system components, and formulated in terms of environmental phenomena [6].

The restriction to environmental phenomena means that a requirement should be formulated in terms of the domain wherein the software should function. Software to be used for elevator control should be described in terms of movement between floors and of doors opening and closing, not in terms of assignment to program variables. How these domain concepts should be represented in terms of program entities is a matter to be decided later, in the internal design phase. Requirements should be comprehensible to future users of the program, in order to prevent systems being built that do not satisfy expectations. In terms of the activities described in Figure 2, requirements are looked at during 'external analysis'.

In practice, establishing requirements is not a trivial task.

Definition 4.2 *Requirements elicitation* is the activity of discovering requirements and assumptions that will shape the system-to-be, based on domain understanding. The requirements are discovered incrementally, in relation to higher-level concerns, through exploration of the problem world. In real life this is a cooperative learning process in close collaboration with the system stakeholders [6].

In a first-year programming course, usually the programming assignments are given in written form, without stakeholders being available for clarification. However, the written assignment is often insufficient to serve as requirements: it may be under specific and necessitate making additional assumptions that should be recorded.

A typical exercise is as follows:

Example 4.1 Write a class called Person that has information about the person's name and age. The class should have methods to ask for the person's name and age.

We will use this as a running example.

4.2 Analysis

Definition 4.3 *Analysis* is the inventory of all information needed to support all decisions made during design and specification.

As we can see in Figure 2, design, specification and coding are all based on analysis. By analysis we mean the ongoing deliberations that lead to informed choices and design decisions. This analysis is not a phase that can be considered closed at any point, nor does it produce a document of its own: the results of analysis find their way into design, specification and code.

From a didactic point of view, we do not consider it a good idea to force students to write out the reasons for their decisions in full, as in literate programming [4]. This will probably overtax their prose composition ability, and they will not appreciate this as a useful contribution to code quality. However, students should be prepared to answer questions about the reasoning behind their choices.

Note that the analysis step is very important. If for example insufficient or even no thought is given to the extent to which a method should be robust, i.e. what can go wrong, then this aspect will be insufficient or even not reflected in the design, specifications, implementation and test cases.

Example 4.2 Here we can think of which names are allowed. For example, a name may only consist of letters. Age is expressed in years, i.e. integers greater than or equal to zero. Age is determined relative to the current date.

4.3 External versus internal

In our approach, the distinction between internal and external a class or method is important. The boundary between internal and external is formed by an interface.

Definition 4.4 *An interface* is the set of services that a module provides to its clients[1] [3].

In an OO-language a module is a class delivering its services by means of methods. A class interface describes exactly what a client need to know in order to avail themselves of the services provided by the class. This is the External View of an interface. The way the services are accomplished by the class is called the implementation of the class. This is the Internal View of an interface.

Example 4.3 An external design of the Person class can be as follows

```
public class Person {
  public Person(String name, Date birth_date)
  public String getName()
  public int getAge()
}
```

[1]This concept should be distinguished from the Java syntax element of the same name.

Note that only public members are included. An internal description can be:

```
public class Person {

  private String name
  private Date birthDate

  public Person(String name, Date birth_date)
  public String getName()
  public int getAge()
}
```

Note that besides the public members also the private members are incorporated. Note also the attribute `birthDate` of type `Date`, used to calculate a person's age. Instead of date of birth, age of type positive integer could have been chosen. During analysis we decide for date of birth to avoid having to update the age every year.

4.4 Design versus specification

In our procedure, we distinguish design from specification. Generally, a software design is about a system decomposition into modules, a description of what each module is intended to do and of the relationship among the modules. Such a description is often called a software structure [3]. In our approach, the focus is on the development of one class with its methods. A class design is described in terms of the class definition with its methods ans attributes.

Definition 4.5 *A class design* is the class name with its attributes and methods. A method is defined by its signature, i.e. method name, names and type of parameters, and return type. An attribute is defined by its name and type. Attributes and methods can be public of private accessible.

Example 4.4 The external design of the class Person is as follows:

```
public class Person {
  public Person(String name, Date birthDate)
  public String getName()
  public int getAge()
}
```

Design as a process means adding syntactical entities as class, attribute and method. A design can be external (class header and public methods) as well as internal (class header, public and private methods, and attributes).

A specification is a statement of an agreement between the implementer of a service and a user of that service [3]. A specifications add rules to the design entities, i.e. the requirements that must be met when a service is used (preconditions) and how the entity then behaves (post conditions and invariants).

Definition 4.6 *A specification* adds rules to design entities in the form of preconditions, postconditions and invariants.

Example 4.5 We can specify the following two rules for the constructor's parameters date of birth and name as part of the external design of the Person class: 'The date of birth is today or in the past' and 'name is a valid name'.

Specifying as a process means adding rules to attributes (private) and methods (public and private). A specification can be external (belonging to a public method) or internal (belonging to a private attribute or private method).

A complete class design and specification lead to a complete internal and external description of a class:

- An external design and external specification results in an External View (also known as API)
- An internal design and internal specification results in an Internal View.

4.5 Happy path and non-happy path

In our approach, we make a distinction between success or happy path behavior and non-success or non-happy path behavior. From now, we will use the terms happy path and non-happy path behavior.

Definition 4.7 *Happy path* behavior describes a successful method call that satisfies the interest of a client. *Non-happy path* behavior describes a not successful method call [7].

Happy path behavior is as everthing goes well, i.e. it is possible to realize the intent of the module. A non-happy path describes what happens if the intent of the software can not be realized. It often results is an error message like an error-code, an exception, or the boolean value false.

The distinction between happy- and non-happy path is an example of the principle *Separation of concerns*: it allows us to deal with different individual aspects of a problem so that we can concentrate on each separately [3].

In our approach, we first concentrate on happy path behavior. After that, we concentrate on non-happy path behavior for which we extend the design, specification, implementation and test cases. Adding the non-happy path behavior is known as providing *robustness* (see Section 9).

Example 4.6 Looking to a log-in process, happy path behavior is as the string value is correct. We can make a design and specification for this behavior. Non-happy path behavior happens when the string value is not correct. We can extend the design and specification, for example by extending the method's signature by adding an exception and by adding additional rules that describe when the string value is incorrect and that in such a case the exception is thrown.

5 External View

5.1 External Analysis

The External Analysis focuses on the question: 'Precisely what functionality must be realized?' We distinguish two situations as examples:

1. A class must be realized as part of an assignment. The class signature and methods' signatures are given. The exercise is to complete this design with conditions specifying the effects of the methods and to implement the class.
2. A class must be realized as part of a project. There is a rough idea of what the class should do, but all the details, such as signatures, have yet to be determined.

In the first situation, the analysis mainly consists of understanding the given design and to gather all the information needed to be able to draw up the conditions as part of the specification step. In the second situation, the room for choice is much larger. The External View as a whole must be set up entirely. Now, for example, one has to think about suitable names for the class and methods, what parameters are needed, the type of these parameters, what conditions apply, et cetera.

External Analysis means studying the domain in which the class or function will function. This type om of knowledge is called *domain knowledge*. The domain can be about mathematics such as a function that adds a series of numbers or a function that solves a system of n equations, or can be non-mathematical as for example a class for the registration of a person or a bank account.

Guidance From the context given or assumed, the following questions can give relevant information:

- Which data is minimally required to do the processing?
- Is there a special notation for the data? Think of km/s instead of m/s or cm instead of inches!
- Give precise definitions of all the domain concepts that play a role in the exercise.
- Do implicit arrangements exist in the domain? Yes, make them explicit! For example, strings representing a calendar date have a different order in the US and in Europe (month/day versus day-month). Leaving the meaning implicit will incur serious misunderstanding in international use.
- Which return value(s) is/are expected? Is it a simple value? For example, the result of an addition. Is it a complex value? For example, a path through a network or two roots of a quadratic equation. Is it a side effect, for example text saved in a file.

- Are there constraints on the data? For example a temperature in Celsius can not go below -273.15 (absolute zero), a list with items contains zero or more elements.
- Give examples of use and input/output.

Example 5.1 The results of the analysis for the `Person` class, Example 4.1, can be:

- We consider age in relation to the current date.
- Age has unit year.
- Age is a zero or positive integer.
- We choose not to have an upper boundary.

5.2 External Design

Design is about adding syntax elements based on the analysis results. At this stage it is about the public elements of a class and/or method. The class's and methods' signatures are given, provided with a description of the responsibility of the entity using the description tag `@desc`.

Guidance

- Give each class and method one responsibility, i.e. high cohesion. If the word 'and' is used, the class or method has probably to much responsibilities.
- Think of clear names for the class (noun) and its methods (verb) reflecting their responsibilities.
- Write the name of the class and the signatures of the public methods.
- Describe the class's and methods' responsibilities (with the tag `@desc`) in terms of domain-specific concepts gathered during the External Analysis.
- Often, getters and setters are default and not part of the design.

Example 5.2 The External Design for the `Person` class, Example 4.1, can be:

```
/**
* @desc This class contains the information about a
* person consisting of his/her name and age in years in
* relation to the current date.
*/
public class Person {

  /**
  * @desc Constructs a person object
  */
```

```
  public Person(String name, Date birthDate)

  /**
  * @desc Returns the person's name
  */
  public String getName()

  /**
  * @desc Returns the person's age
  */
  public int getAge()
}
```

5.3 External Specification

An External Specification adds conditions to the External Design entities and thereby completes the External View (often called the API, for Application Programming Interface). These External Design entities correspond to the concepts in the problem domain (represented by the classes) and the tasks working on the concepts (represented by the methods). These concepts and tasks must satisfy some conditions on their behavior.

Example 5.3 A person's name should be a valid name. It must be, for example, only made up of characters; numbers are not allowed. A person's age, must be a positive whole number, also on the person's birthday. A person is not 20,5 years old for a little while. Instead, one can have age 20 or 21 years and nothing in between.

General principles There are two general principles for External Specifications:

1. They are *public*, which means that they are intended for the clients of the software entity, not just for the implementer.
2. They should not refer to internal implementation details but only to 'what is visible to clients', i.e. the concepts from the application domain and the signatures of the public methods.

In our approach, the External Specifications can be written in natural language.

Example 5.4 As part of an External Specification of a stack, the method 'pop' can be described as 'removes the top element of the stack'. A rule can be 'the number of elements in the stack can not be negative'.

Elements of an External Specification An External Specification comprises the following elements:
For each class:

- `@desc`: the responsibility of the class, adopted from the External Design.

- `@inv`: the states an object of the class may be in expressed in terms of domain concepts.

For each method:

- `@desc`: the responsibility of the method.
- `@requires`: the precondition that must be met when the method is called, in terms of the method's parameters and/or other domain concepts.
- `@ensures`: the postcondition that will be met when the method was called and the precondition was true. It comprises:
  - A boolean function on the domain concepts and methods arguments.
  - In case of a result, this appears in the post condition as `\result =` description of result value.
- `@assignable`: Properties of the state of this or other objects that the method may change.
- `@pure`: Equals `@assignable \nothing`, i..e. does not change any object's state.

The reason for having an `@assignable` clause is that this avoids cluttering the specification by enumeration of all the state properties that will *not* change upon execution of the method. In particular, when the method merely inspects the state and does not change it at all, we use the abbreviation `@pure`. Please note that this is a less rigorous restriction than the 'referential transparency' concept used in functional programming, which requires that the value of a function call depends only on the value of the parameters, not on any attributes or global variables.

Guidelines for class specifications

- Are there one or more class invariants? An example of a class invariant, as part of a sorted collection', written in natural-language in terms of domain concepts is: 'This collection is sorted'.

Guidelines for method specifications

- Which combinations of values for the parameters should be allowed? In what state can the method be called? (`@requires`)
- Which return values are possible?
- How does the method affect the (state of) the object? (`@ensures`)
- What does the method change within this or other objects? (`@assignable` clause or `@pure`)

Example 5.5 After completing the External Specification, the `Person` example (Example 4.1) looks like this:

```
/**
* @desc This class contains the information about a
* person consisting of his/her name and age in years in
* relation to the current date.
* @inv age >= 0 AND name is a valid name
*/
public class Person {

 /**
  * @requires birthDate is today or in the past
  * AND name is a valid name
  */
  public Person(String name, Date birthDate)

 /**
  * @desc Returns the person's name
  * @ensures \result = the person's name
  * @pure
  */
  public String getName()

 /**
  * @desc Returns the person's age in years
  * @ensures \result = the person's age in years
  * @pure
  */
  public int getAge()
}
```

Guidance

- As the External View contains no implementation code, no attributes and only public method signatures, it is ideally suited to the Java interface concept. It is, in fact, often preferable to store this as an interface to be implemented by the class we are about to design, as this opens the way to multiple implementations and allows client methods to be equally applicable to all of these.
- If the class under construction implements an interface, the External Specification of the interface is to be regarded as the External Specification of the class as well.

In the examples discussed so far, we have defined one specification per method. We are free to have more than one specification per method. The following gives an example.

Example 5.6 Suppose we have to specify a method `power(x, y)` calculating the expression x^y. Depending on the requirements, we may have the following external design and specification:

```
/**
  * @desc Calculates the power of x to y
  * @requires x >=0 and y >= 0
```

```
 * @ensures \result is x^y
 * @pure
 */
 public static int power(int x, int y)
```

However, 0^0 is not defined mathematically. We can choose for example 0^0 equals 0 or 1. Using subspecifications, we are able to make our choice explicit.

```
/**
 * @desc Calculates the power of x to y
 * @sub x != 0 {
 *   @requires x > 0 and y >= 0
 *   @ensures \result is x^y
 * }
 * @sub x == 0 {
 *   @requires x ==0 and y >= 0
 *   @ensures \result is 1
 * }
 * @pure
 */
 public static int power(int x, int y)
```

The extra subspecification `x == 0` is valuable for the client of the method `power`. It alerts the client to this special case. As we will show later, it helps the programmer too. It helps to implement the body of the method using the two distinguished cases and it helps to define test cases.

To use a subspecification, we add the tag `@sub` followed by a description of the case. This description is essentially a comment for the human reader, so it does not have to adhere to Java naming conventions: natural language and various mathematical formulas are allowed. However, as we shall see in the chapter on testing, test methods will correspond to these subspecifications and need to have names showing to which they belong. The names of test methods must abide by Java rules for method names. In the example, this would probably lead to something like `testPowerNonzero` and `testPowerZero`.

We will discuss subspecifications in more detail in Section 6.5, because subspecifications mostly occurs in the Internal Specification activity. However, it can be valuable to apply subspecifications in the external view too.

6 Internal View

6.1 Internal Analysis

6.1.1 Representation

An important decision to be made as the foundation for Internal Design is how to represent domain concepts in term of programming language entities. Of course, in many cases this is a trivial decision: most programs having to manipulate integer values will simply use the standard type `int` (but some might need `java.math.BigInteger` instead). However, there are many cases where there is no standard implementation available (think of graph algorithms) or, on the other hand, there are many possibilities to choose from (for instance collections).

Example 6.1 Consider a class `Producer` whose External View consists of a set of strings, with an operation `String produce()` that removes a single element from the set and produces that element as the result. The Java library interface `java.util.Set<String>` does not offer this functionality, so the programmers must build their own. There are many options to choose from here; however, the point we want to make is that the chosen option is not relevant for the users of this class and must not appear in the External View. This has the advantage that a change in the representation will not influence any programs that use class `Producer`. However, the Internal View, which sets the task for coding, must provide the decision taken.

Let's consider one option: give class `Producer` an attribute `stack` of type `java.util.Stack<String>`. Take care that every element of the set we wish to represent is present in `stack` exactly once. Then method `produce` can be implemented as `stack.pop()`. Note that the order in which the elements appear in the stack is not relevant; any order suffices.

Guidance

- Decisions at this point will depend heavily on arguments of practicality and efficiency. For instance, in the `Person` example (Example 4.3), the insight that it is more practical to store the birth date rather than the age (because the latter must be updated at every birthday), and to recalculate the age everytime it is needed, is typical for the discussions in Internal Analysis.
- The correspondence between the current state of the domain concepts and that of the language entities will be documented in the Internal Specification by a `@represents` clause. This is not necessarily bijective, as is obvious from Example 6.1: one and the same set corresponds to many stacks that differ only in the order of the elements.
- To emphasize that the External View is entirely independent of the chosen data representation, it may be advisable to make the External View into an interface rather than a class. This has the advantage that multiple implementations may be present in the same software.

6.1.2 Attributes and private methods

Once we have decided on the data types to represent domain concepts, we can complete the choice of attributes to be stored in the objects and of private methods useful for working with these attributes.

Guidance

- One good reason to introduce private auxiliary methods is the avoidance of code duplication, which will occur if the same computation must take place in several different circumstances.
- It is seldom a good idea to have public attributes. Having all attributes private and regulating access through public getters and setters has two advantages: in the first place this makes it possible to refuse values that do not correspond to possible states of the domain (think of negative ages), in the second place the representation may be changed later without influencing clients.
- Often there is a choice to be made between storing computed values in an attribute and recomputing them when needed. For example, for a class `Person` we can add an attribute `age` storing an age value. Alternatively, we can recompute the age value when the age value is needed. Notice that in the first option the age value must be recomputed yearly too.
- Both attributes and methods should receive a clearly understandable name that reflects their function.

6.1.3 Functional decomposition and helper objects

Whenever a programming task results in voluminous code, it is not a good idea to put it all in a single method. For understandability and reusability, it is better to have simple methods dedicated to a single task, a principle known as *cohesion*. The principle of cohesion also functions at the class level: if a class seems to serve several dinstinct purposes, it is better to split it into separate classes, each with a single responsibility. This leads to so-called helper objects, that relieve the main class of some of its burdens. An example is as follows:

Example 6.2 For calculating the price of a product in a certain country, several tax rules are needed. With a number of countries, defining all these rules in class `Price` results in low cohesion of class `Price`. It is better to define an extra interface Tax with subclasses for each country. Class `Price` uses one of these subclasses by delegation.

We will return to this issue in more detail in a later section.

Guidance

- The functional decomposition process may go through several stages, each time introducing smaller and simpler steps until we reach the level where these have an obvious implementation consisting of only a few statements.
- Auxiliary calculation steps that are understandable on their own should be delegated to separate methods. Consider carefully whether these auxiliary methods are useful only for the present purpose, in which case they should be private methods of the class under construction, or will be more generally useful, in which case they should be public methods, possibly of a different, more specialized class.
- In many cases the helper object classes need not be constructed from scratch, but can be found in a library or be defined as a subclass of an existing library class. For instance, class `Date` occurring in the standard `Person` example (Example 4.3) can be implemented using `java.util.GregorianCalendar`.

6.2 Internal Design

In the Internal Design activity, we record the decisions made during Internal Analysis.

Example 6.3 Following the running example, the person's name and age should be retrievable, so we decide on private attributes `name` of type `String` (there being no use for a `StringBuffer` because names do not change all the time), and `birthDate` of a type `Date` that can be used for computations with calendar dates (rather than an attribute `age`, for the reasons explained above).

Attribute `birthDate` is an example of a helper object. For the implementation of method `getAge()` it is necessary that class `Date` be equipped with a suitable method to calculate the number of years between two given dates. Auxiliary private methods in class `Person` are not needed in this example.

Guidance

- Generally, we determine at this stage what attributes and methods will certainly be needed. When it turns out later, for instance during implementation, that other attributes and methods are needed, these can be added later. The development process is iterative and incremental.
- At the point of design, we also add descriptions of the classes and methods to explain their use. These will be elaborated upon, to make them precise and complete, during specification activities.
- As in External Design, we use the convention that all descriptions and specifications are placed in JML-like comments above the entity described.

6.3 Internal Specification

The Internal Specification of a method differs from the External Specification in that there is an extra clause tagged by `@represents` that determines how the problem domain concepts (used in the external specification) correspond to class attributes. This clause is called the *representation relation*. Just like the External Specification, there are `@requires` and `@ensures` tags listing respectively the precondition and postcondition of the method. However, in the Internal Specification we prefer to use only attributes and parameters here, in order to make the implementation independent of domain knowledge.

Similar differences exist between the External and Internal Specification of a class; however, at the class level we do not have preconditions and postconditions, but we do have an invariant, tagged with `inv`, that limits the states the object may legally take.

Example 6.4 After completing the Internal Specification, the `Person` example (Example 4.3) looks like this:

```
/**
 * @desc This class contains the information about a
 * person consisting of his/her name and age in years in
 * relation to the current date.
 */
 public class Person {
 /**
  * @represents age = Date.yearsBetween(Date.today, birthDate)
  *             AND name = lastName
  * @inv birthDate <= Date.today AND lastName is a valid name
  */
  private String lastName
  private Date birthDate

  public Person(String aName, Date birthDate) {
    /** @requires birthDate <= Date.today AND aName is a valid name
    * @ensures this.birthDate = birthDate AND lastName = aName
    */
  }

  /** @desc Returns the person's name */
  public String getName() {
    /** @ensures \result = lastNam
     * @pure
     */
  }

  /** @desc Returns the person's age in years */
  public int getAge() {
    /** @ensures \result = Date.yearsBetween(Date.today, birthDate)
     * @pure
     */
  }
}
```

Guidance

- Internal Specifications are only visible to the developer(s) of the class itself (or perhaps the developers of the enclosing package). Clients of the class should not depend on them in any way.

- An important difference between external and internal specifications is that external specifications also bind all subclasses, whereas internal specifications only apply to the class in which they appear. The reason is that, due to dynamic binding in OO languages, a variable declared to be of a certain class type may actually point to an object of some subclass, so that method calls will perform an overridden version. Hence the client must be able to trust that such overridden methods will also conform to the external specification of the superclass [10].

- Internal Specifications focus on the internal state (attribute values). Formally, they consists of assertions on the state. State consists of the momentary values of the object's attributes.

- When designing a class, determine whether all combinations of values for the attributes are allowed. Do some combinations result in an inconsistent object? The class invariant `@inv` is the place to record this. Every method should respect the class invariant, in the sense that this condition is added silently to the precondition and postcondition.

- In practice, avoiding domain concepts in Internal Specifications is not always quite feasible, as it may result in conditions that are hard to read or understand. To see an example, look at the conjunct `aName is a valid name` in Example 6.4.

- The Internal Specification is not independent of the External Specification, but must be a refinement of it. The exact rules governing the correspondence will be listed in Section 6.4

- In writing specifications, we follow the general assumption that parameters (and array elements occurring in them) are non-null.

- In a postcondition, we can refer to the value a parameter or attribute had in the situation of the precondition by preceding its name with `\old`.

- It is practical if all views are generated from a single source document. This requires some extra editor software, but avoids the complications involved in keeping all views in synchrony. A helpful convention is then to put external specifications of classes and methods above the heading, internal specifications below (within the name space introduced by the definition).

6.4 Correspondence between External and Internal Specifications

6.4.1 Proof rules

The internal specification does not need to be a precise translation of the external specification. What we need is that all methods built by using the internal specification will satisfy the external specification promised to the client. That means that the internal specification is allowed to be more strict than the external one: we may deliver a better implementation than is contractually required. What we mean by 'better' is that the implementation may be usable in more situations than the external specification required, so the internal precondition may be weaker; and the results delivered may be more precisely described, so the internal postcondition and the internal class invariant may be stronger than the external ones. To remember this, keep in mind that in real life no client will object if the product delivered turns out to have a lower price and better performance than specified in the sales contract. See Example 6.1 for a case where the internal specification really demands much more than the external one.

The formal rules governing the correspondence between the two levels of specification may strike the reader as somewhat abstract and difficult. Fortunately the concrete proof obligations arising from this correspondence rarely amount to much.

- The representation relation (`represents`) and the internal class invariant should together imply the external class invariant. In our running example 6.4, the internal invariant of `Person` is `birthDate <= Date.today`. The representation relation is `age = Date.yearsBetween(Date.today, birthDate)`. We have to check that these two together imply the external invariant `age >= 0`. In this case, the check is a consequence of the specification of `Date.yearsBetween`.

- For every method, the representation relation and the external precondition should together imply the internal precondition. In the running example the external precondition of the constructor of `Person`, as given in Example 5.5 is `birthDate is today or in the past`. The representation relation is `age = Date.yearsBetween(Date.today, birthDate)`, but we do not actually need that to check the internal precondition `birthDate <= Date.today`. In this case, the check is trivial.

- For every method, the representation relation and the internal postcondition should imply the external postcondition. In the example, the internal postcondition of `getAge()` is `\result = Date.yearsBetween(Date.today, birthdate)`. The representation relation is `age = Date.yearsBetween(Date.today, birthDate)`. We have to check that these two together imply the external postcondition. `\result = age`. Once again, this check is trivial.

6.4.2 Class invariants

The representation relation may also entail extra class invariants (sometimes [9] called *representation invariants*), used to limit the range of values allowable for attributes. Some examples are:

- If, in a system dealing with calendar dates, the month is represented by an attribute `int m`, an extra condition $1 \leq \mathsf{m} \leq 12$ is to be added to the class invariant (and hence to all preconditions and postconditions).
- If a string is used to represent a person's name, an extra class invariant should state that all the characters in the string are letters (and not, for instance, &).
- If an array is used to store a number of input values, an internal invariant should limit the number of values to prevent overflow.
- When, as in Example 6.1, a stack is used as representation of a set, an extra class invariant is needed stating that all elements of the stack should be different.

6.4.3 Representation relations are not necessarily functional

A question one might legitimately ask is whether the connection between domain entities and class attributes is a function in any direction. Many textbooks seem to think so, using terms like 'abstraction function'. However, in general the answer is no. We give two examples to show this.

To show that a single value in the problem domain may correspond to several different attribute states, we can return to Example 6.1. Different states of attribute `stack` that differ only in the order of the elements in the stack will represent the same set of strings in the domain. So there is no function from the domain entities to the attribute values.

In the other direction, consider a class used to print boarding cards for an airplane. Next to flight number, boarding time, and assigned seat, these contain only the last name and first initial of the passenger. So this requires a class `Passenger` that only stores those two facts. However, it is quite possible that two different passengers are on the same flight and have the same last name and first initial. These will correspond to separate objects in the seat allocation process, but separate objects with the same attribute values. So there is no function from the attribute values to the domain entities.

6.5 Subspecifications

It is often useful to include specific subspecifications per method instead of one comprehensive specification. We have already seen an example of this at the end of Section 5.3. The following gives another example.

Example 6.5 Consider a class modeling phone numbers, with a method that will call another phone. The corresponding contract would be something like

```
/**
 * @desc This class contains a phone number.
 */
class Phone {
 /**
  * @represents theNumber contains the digits of the phone number
  * @inv all theNumber[i] are in range [0..9] AND theNumber.length >= 10
  */
  private int[] theNumber;

  public Phone(int[] number) {
    /**
    * @requires all number[i] are in range [0..9] AND number.length >= 10
    * @ensures theNumber = number
    */
  }

  /**
   * @desc establishes a phone call between this and that
   */
  public call(Phone that)
}
```

When we think about the implementation of method `call`, we realize that, due to properties of the telephone exchange, the process of realizing the connection is different for national and foreign telephone numbers. Therefore class `Phone` needs a way to enable clients to check whether the phone number is national or foreign. These can be distinguished because the latter begin with 00, whereas inland phone numbers begin with 0 followed by a nonzero digit. Examining the result `isForeign` naturally produces a case analysis. The idea of subspecifications caters for such a situation. In Example 6.5, the specification of `call` would then look as follows.

Example 6.6

```
/**
 * @desc establishes a phone call between this and that
 * @sub Phone number is not foreign {
 *    @requires !that.isForeign()
 *    @ensures phone call on the national exchange is established
 * }
 * @sub Phone number is foreign {
 *    @requires that.isForeign()
 *    @ensures phone call on the international exchange is established
 * }
 */
public call(Phone that)

/**
 * @desc determines whether the phone number is a foreign one
 * @pure
 */
public boolean isForeign() {
  /**
   * @requires theNumber[0] = 0
```

```
   * @ensures \result = theNumber[1] = 0
   */
}
```

This leaves the question what should happen in case `theNumber[0] != 0`. In that case the telephone number is ambiguous and we must decide what course to follow. One option would be to raise an exception, another to assume that a local number is meant and to add the standard local prefix. How to specify this will be discussed in the Section 9 on robustness.

Subspecifications satify the following general pattern:

```
@sub <description of the situation> {
  @requires <precondition>
  @ensures <postcondition>
  @assignable <footprint
}
```

Guidance

- The splitting of a contract into subspecifications mostly occurs in the Internal Specification activity, as this is usually the phase where the need for case analysis first appears. However, it is possible that this is already clear during External Specification, and in that case the External View will also contain subspecifications with the same syntax.
- The idea of subspecifications is taken directly from JML. However, a major difference is that JML does not require to adorn the subcontracts with a descriptive name, and merely separates them with the keyword `also`.
- In writing specifications, we follow the general assumption that parameters (and array elements occurring in them) are non-null.
- In a postcondition, we can refer to the value a parameter or attribute had in the situation of the precondition by preceding its name with `\old`.
- During implementation it may turn out that the specification can not be fulfilled in an efficient way. The solution can be to negotiate a change in the specification.
- Experience with the finished program, may indicate that the requirements did not capture the intention of the problem owner very well. Again, this may lead to negotiating changes in the requirements, design and specifications.

7 Annotated Code View

The treatment of the Coding acvitivy in this section is relatively terse. This is because this subject is treated in detail in existing textbooks and courses; we have little to add

7.1 Code Analysis

During Coding, the methods are provided with a body. The signature (a result from Internal Design) and conditions (a result from Internal Specification) both indicate what must be done and which conditions apply. But what the method body will look like, is still open. At the method level, it makes sense to determine whether each method so far has only one responsibility or whether it should be split up (cohesion). Additional private help methods can be added if necessary. For more algorithmic problems, coming up with an algorithm can still be a complex affair. For this, we refer to standard courses about data structures and algorithms.

During this analysis step, we get *insight* in how to implement the method bodies. In the next step, these method bodies are designed and coded.

Guidance

- Think of further decomposition of the problem.
- If needed, think about additional (private) helper methods.
- Try to abstract, i.e. make the method domain-independent. For example, sorting ages can be considered as sorting integers.
- If the problem is too complex, try to simplify it. For example, first try to sort two numbers, after that sort more numbers.
- Even if the process is known, write out the solution completely. Very often, you will get insights in all the details valuable for the specification and the implementation.
- Apply pen and paper elaborations: how to come from input to output?

7.2 Code Design

Based on the analysis, a method's body structure is chosen or invented. In case of an algorithmic problem, a concrete algorithm should be designed. Sometimes, only a sketch of this algorithm is available based on the analysis step. Sometimes, an algorithm is known as a standard for solving the problem at hand. In both cases, we have to think about meaningful local variables and if needed, we have to design extra (private) helper methods.

Guidance

- Describe the solution's process and its data in abstract terms, say as comments in natural language or math notation; do not limit yourself to legal Java expressions.
- The algorithmic decisions reached in Code Design can be recorded as ordinary comments. Once the executable code has been produced in the next step, these may remain as explanation to make the code more readable and maintainable.
- The local variables in the code serve to record temporary versions of results that are built up step by step. For instance, the greatest common divisor of two integers is usually calculated by Euclid's algorithm, which involves a repeated application of division remainders.
- A different use for local variables is to record the result of a calculation that is used repeatedly. For example: solving a quadratic equation we can use the abc-formula. The abc-formula makes use of a discriminant which can be defined as a local variable.
- Introduce additional private helper methods as a result of the decomposition. For instance, in writing a method for sorting an array, the decision to use the Quicksort algorithm calls for an auxiliary method for sorting part of the array (that will be called recursively).
- If necessary rethink Internal Design and Specification issues. It may occur that the algorithm requires operations on the attributes that cannot be efficiently implemented with the chosen type; in such a case, we must consider the possibility of choosing a different representation.

7.3 Coding

During coding, the method body's structure in terms of local variables, pseudo code or math language is translated into an machine-interpretable language such as Java. The result is called the *Code*; the *Annotated Code View* consists of the Code together with the specifications and comments developed earlier. These are not to be discarded when the Code has been completed, as they have an important function in ensuring maintainability of the software, and also form the basis for Test Construction (as we will see in the next section). Hence the Annotated Code View contains both the External View and the Internal View. We can consider the subsequent views as a continually growing document. The complete solution of a programming task consists of this document together with all tests.

Guidance

- Search for duplication in code. In case of duplicated code, think about extra helper methods (design).

- Ensure that the code satisfies the design and specifications, i.e. ensure the correctness of the functionality. Manually check the correctness of the code against the specification. Later, we will define External and Internal Tests to check this automatically.
- Avoid all kinds of bugs, think about statements that are not guaranteed to succeed. Later, we will define Code Tests to check this automatically. Examples of bugs are:
  - Array-access out of bounds.
  - Creation of an array with negative size.
  - Operations that cause overflow and malfunction.
  - Method calls on variables and fields that have the null-value.... etc.

8 Testing

8.1 Introduction

Having the three views External View, Internal View and Annotated Code View, *finding* suitable tests has become easier: the three views give a lot of information for this. For each of these views, we define dedicated tests:

- External Tests belonging to the External View. These are JUnit test cases testing whether the external specifications are satisfied.
- Internal Tests belonging to the Internal View. These are assertions based on the internal specifications. These assertions can be tested by extra JUnit test cases.
- Code Tests belonging to the Annotated Code View. These are extra test cases covering critical code fragments, for instance situations of overflow, exceedance of array boundaries, or the existence of a file that has to be read. Code Tests can be implemented in the form of JUnit test cases or assertions.

Our focus is on classes and methods. For testing these units of code, *unit testing* is a suitable approach, which is technical feasible with Java's JUnit test framework.

Use of JUnit We assume the following concepts: Each Java class `X` has its own test class `XTest` implemented in a separated file. For each subspecification of each method of class `X`, class `Xtest` contains a test method annotated with `@Test`. Each test method contains one or more test cases, represented by assertions, i.e. predicate expressions that are either true or false, for example `assertTrue`. A test case defines a pair of test input and expected result.

Assertions can be applied in the class under test as well, with Java's `assert` keyword. These predicates are located at certain places in the code whose value is interesting during the execution of the code at that place.

From specifications to test code In our framework, the step Test Construction consists of the translation of all the (sub)specifications as part of the External View and Internal View into test cases, i.e. pairs of test input and expected result. The translation of these pairs into JUnit test code is a relatively easy task: for each combination 'method `M` – (sub)specification `C`' a test method with name `testMC` is defined containing the assertions needed and provided with annotation `@Test`.

8.2 External Tests

External tests are exclusively based on the External View. The reason to produce External Tests is that these may be applied time and time again during

the development process as the Code is being constructed and refined. External Tests may be defined as soon as the External View is available; it turns out that the definition of tests itself contributes to getting the specifications clear and complete.

Defining an External Test, we apply the following approach consisting of three steps:

8.2.1 Step 1: Make a model

Being able to test sufficiently whether the functionality implemented satisfies the contracts as part of the External View, we have to 'translate' these contracts to test cases. To do this in a way giving us sufficient certainty that we have all test cases needed, we make use of test models. There exist several test models, for example Equivalence Classes of input domain variables, Boundaries of input domain variables, Decision Tables, Combinatorial and Mutation Testing. For practical reasons, we choose for the first two mentioned: Equivalence Classes and Boundaries of input domain variables. With both models, we are able to find sufficient test cases corresponding to a contract.

Example 8.1 Suppose we have the following External View:

```
/**
* @desc Calculates the special sum of two restricted variables
* @sub happy path {
*   @requires 0 <= x <= 9 and 1 <= y <= 10 and x+y <= 17
*   @ensures \result = x + y
* }
*/
int specialAdd(int x, int y)
```

Based on this contract, we can define the following equivalent classes and boundaries of the input domain, where vP means valid partition (in Section 9 about robustness we will add invalid partitions too):

Variable	Valid or invalid class	Range of class	Boundaries
x	vP1	[0..9]	0 and 9
y	vP2	[1..10]	1 and 10
x+y	vP3	[1..17]	1 and 17

8.2.2 Step 2: Choose a coverage criterion

Also with two test models Equivalent classes and Boundary testing, it is often impossible to apply all possible inputs to test whether the software produces the correct output. Therefore, we choose a coverage criterion. There exist several coverage criteria. Again, for practical reasons, we choose one criterion, namely the all combinations coverage. In combination with the Equivalence Classes model, this means that we have to take for each input variable all valid classes of its input domain and then combine the classes of these input variables in every possible way. To reduce the number of test cases, we choose from each

input class minimal one value. As boundaries, we choose the minimum and/or maximum value at the boundary of each equivalent class.

Example 8.2 Following Example 8.1, we choose to use the boundary values and one extra value from each equivalent class. How to choose the values becomes more obvious by showing a graphical representation of the values of variables x and y, which is in this example possible. Figure 4 shows a sketch of this graph. This results in the next table:

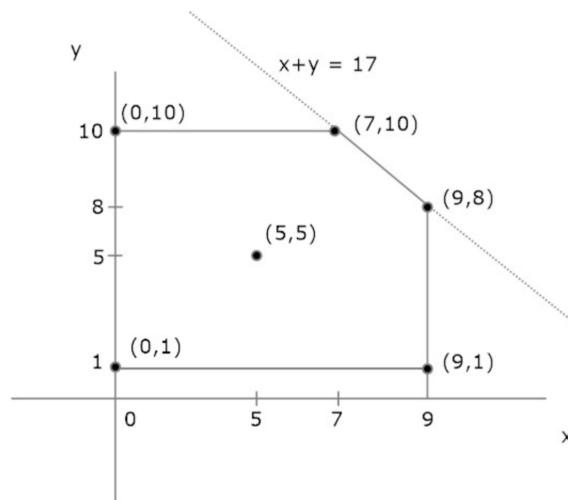

Figure 4: A sketch of the graph of the valid values of x and y

Variable(s)	Valid or invalid class	Input values
x	vP1	0, 5, 9
y	vP2	1, 5, 10
(x,y)	vP3	(0,1),(0,5),(0,10),(5,1),(5,5),(5,10),(9,1),(9,8),(7,10)

8.2.3 Step 3: Design test cases

Having the allowed values for the input variables, we now have to establish the test cases by combining these values and determining the expected results.

Example 8.3 Following our example, the happy path test cases becomes then:

x	y	Expected result
0	1	1
0	5	5
0	10	10
5	1	6
5	5	10
5	10	15
9	1	10
9	8	17
7	10	17

Based on this table the JUnit testcases can be easily implemented. In pseudocode, the final test code becomes:

```
@Test
testSpecialAddHappyPath() {
  assertEquals(specialAdd(0, 1), 1)
  assertEquals(specialAdd(0, 5), 5)
  assertEquals(specialAdd(0,10), 10)
  assertEquals(specialAdd(5, 1), 6)
  assertEquals(specialAdd(5, 5), 10)
  assertEquals(specialAdd(5,10), 15)
  assertEquals(specialAdd(9, 1), 10)
  assertEquals(specialAdd(9, 8), 17)
  assertEquals(specialAdd(7,10), 17)
}
```

8.3 Internal Tests

Defining an Internal Test, we apply the following approach consisting of two steps:

8.3.1 Step 1: Indicate extra test cases

After the internal design and specification are completed, we know much more about the workings of the program we are developing. The most important new knowledge is that, by means the representation rule, the more or less vague External View specifications are often replaced by much more precise conditions in terms of attribute values. These more precise conditions can result in extra test cases. In the following paragraphs, we discuss the different causes and provide these with examples.

Extra test cases due to a representation relation A representation relation limits the possible values of an attribute. Furthermore, a representation variant can lead to a strengthening or refinement of the specifications of methods, i.e. it can result in a different structure of the input domain and can add extra boundary values. Hence, more test cases are often needed.

Example 8.4 Suppose a class for calculating and holding the solutions of a second order equation. In the External View, we can say that such an equation can have zero, one or two solutions. In the Internal View, one can decide to represent these solutions by an array, which is empty in case of zero solutions and has one or two elements in case there are one or two solutions. In this case, test cases should be added to check whether the array contains zero, one or two solutions corresponding to the input values of the coefficients a, b and c.

Extra test cases due to extra class invariants A representation rule may also entail extra class invariants, used to limit the range of values allowable for attributes. Again, these leads to more boundary cases to be tested. Next some examples which are discussed in the section about the Internal View too.

Example 8.5 If, in a system dealing with calendar dates, the month is represented by an attribute `int m`, an extra condition $1 \leq \mathsf{m} \leq 12$ is to be added as a class invariant (and hence to all preconditions and postconditions).

Example 8.6 If a string is used to represent a person's name, an extra class invariant should state that all the characters in the string are letters (and not, for instance, &).

Example 8.7 If an array is used to store a number of input values, an internal invariant should limit the number of values to prevent memory overflow.

Example 8.8 Suppose a method that generates a next house number based on a complex calculation. Where the External View talks about 'a number', in the Internal View the number is restricted to positive integers only.

Extra test cases due to private methods Auxiliary private methods may have been added. Such methods can be provided with a specification, and can be tested separately.

Example 8.9 Following the second order equations example, a helper method `discriminant` can be added. This method can have it's own specification and as such be tested separately.

Example 8.10 Suppose an External Specification about a person's name states a condition

```
n is a valid name
```

Because this condition occurs at more than one place, avoiding repetition and achieving coherence may point to the need for an auxiliary private method

```
private boolean isValidName(String n)
```

Extra test cases due to changing pre- and postconditions Besides that pre- and postconditions of methods can be changed due to representation relations and/or class invariants, the developer can choose to weaken the precondition and/or to strengthen the post condition extra. Weakening the precondition, ultimately to a precondition true, can be part of making the method robust, a topic that will be discussed in a next section.

Example 8.11 Given the abstract data type `Stack`, with the constraint that the values in the stack, from top to bottom, form an ascending sequence. In order to combine two such stacks into a single one, we produce the following Java code:

```
/**
* @merges ascending stacks a and b into a single ascending stack
* @requires a and b are ascending from top to bottom
* @ensures \result is ascending from top to bottom
* and contains all items from a and b
* @pure
*/
public static Stack<Integer> merge(Stack<Integer> a, Stack<Integer> b) {
  Stack<Integer> c = new Stack<Integer>();
  while (!a.empty() && !b.empty()) {
    if (a.peek() < b.peek()) {
      c.push(a.pop());
    }
    else {
      c.push(b.pop());
    }
  }
  while (!a.empty()) {
    c.push(a.pop());
  }
  while (!b.empty()) {
    c.push(b.pop());
  }
  Stack<Integer> d = new Stack<Integer>();
  while (!c.empty()) {
    d.push(c.pop());
  }
  return d;
}
```

Since the last three loops have a similar shape and differ only in the identity of the stacks involved, they form a good candidate for extracting into a private method. It is important to notice that, while in the context of `merge`, all of the stacks are ordered, this condition is not necessary for the proper working of the extracted method `move`. In order to make `move` reusable in other programs, it is undesirable to import this precondition blindly from `merge`.

8.3.2 Step 2: Determine how to test the extra conditions and private helper methods

Private attributes Private attributes can't be reached directly by test methods part of a separate test file. To be able to read the value of a private at-

tribute, one can add an extra getter. To test whether a class invariant holds after a method has been executed, the test case can use the extra getter to read the attribute's value.

Another method is to add a method that checks the invariant directly.

Example 8.12 Suppose a class has an attribute `housenumber`, which value must be `>= 0`. To test this class invariant, we can add a boolean function `checkHouseNumber` of package scope that returns the value of `housenumber >= 0`.

Adding these boolean methods is not dangerous since it will not enable other classes to read or change the house number. The test class will then be able to check an internal invariant by invoking the corresponing boolean method.

Alternatively, we may postpone this check to Code testing; once the code is available, we may put `assert( housenumber >= 0 );` into the code.

Private methods The best way to test a private method is to add, again, an extra boolean method that test the postcondition after finishing the private method. In a test file, directly after testing the client method calling the private method, the postcondition of the private method can be tested by invoking the extra boolean method.

Another way to test private methods is by declaring them with access modifier package.

Lastly, in case a private method is not provided with a specification, one can consider the functioning of this method as part of the functioning of the calling method which is tested.

Guidance In summary, the guiding steps are:

1. Indicate extra test cases.

2. Determine how to test the extra conditions and private helper methods.

8.4 Code Tests

Code testing is also called structural testing, because the internal structure of the program's code is used as information to define test cases.

Coverage criteria The standard approach of Code Tests is to apply one or more coverage criteria (see for example [3]):

- Statement coverage: each statement in the program is executed at least once.

- Edge coverage: each edge in control flow graph of the program is traversed at least once.

- Condition coverage: all possible values of the constituents of compound conditions are exercised at least once.

- Path coverage: all paths leading from the initial to the final node of the program's control flow graph are traversed.

In these notes, we consider code testing as foreknowledge and, if needed, refer to the standard literature for more information.

Special code constructions Another aspect is to search for special code constructions that can fail under certain circumstances.

Example 8.13 Suppose a method call to open a file to be able to read the content. What happens when the file does not exist? It is thinkable that a test satisfies all the coverage criteria, but fails in testing this aspect of code sufficiently.

Example 8.14 A similar situation happens when the value of a variable, say `a`, is read from a file: `a: = 'read␣from␣file'; return 1/a;`. This situation can not be tested by means of JUnit with distinguishing input parameters. Instead, an assertion should be used: `a: = 'read␣from␣file'; assert(a != 0); return 1/a;`. Remark: this is exactly what went wrong in the software of Ap ollo 11, where a sensor gave value zero, used as denominator in a fraction.

The use of Java assertions to check whether the preconditions of called methods were satisfied is quite different from the JUnit testing we have discussed earlier. These assertions are not part of test code that is executed separately during the Run Tests activity, but are placed in the original program code where it is checked every time the program is run. Moreover, it is not possible to predict under what circumstances running the code will produce an exception; so the only option available is to leave the assertion in place (at least provisionally) and take precautions to catch the exception should it be produced.

Example 8.15 Suppose we implement a method to calculate the roots of a quadratic formula, i.e. $ax^2 + bx + c = 0$. If we implement the abc-formula in the body of this method to determine the roots, i.e.

$$r_1, r_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

then, an extra test case should be added to check the method's behavior when coefficient a equals zero. Again, it is conceivable that a test satisfies all the coverage criteria, but fails in testing this aspect of the code.

Supplementing or not? In some situations Code test cases do not supplement the External View and Internal View test cases.

Example 8.16 Suppose we have the following method `compare` with `@requires` equals true:

```
/**
 * @desc Compares two integers
 * @ensures \result = -1 if x < y
 *                     0 if x = y
 *                     1 if x > y
 * @pure
 */
public int compare(int x, int y) {
    int result = 1;
    if (x < y) result = -1;
    else  if (x == y) result = 0;
    return result;
}
```

A complete set of External View test cases, based on the test models equivalent classes and boundary values, satisfies all the coverage criteria. Applying for example the test cases `compare(0,2)`, `compare(1,1)`, and `compare(3,0)`, covers all code's statements and conditions, and all edges and and paths in the control flow graph. The Code does not contain special code contructs that can fail under certain circumstances. As a result, no extra test cases are needed.

Sometimes, however, implementation specific situations occur that need additional test cases.

Example 8.17 When we implement the abc-formula in the body of a method to calculate the roots of a quadratic formlua (see Example 8.15), then an extra test case should be added to check the behavior of the method when coefficient a equals zero. Depending on the External View, this can result in, for example, an exception. However, had we applied the Newton-Raphson method instead, then this peculiarity did not exist.

Guidance In summary, the guiding steps are:

1. Choose and apply one or more coverage criteria.
2. Search for special code constructions that can fail.
3. Define corresponding test cases.

Finally, we mention some common points of attention for testing in the next table.

Type	Pay attention to
String	null value, empty string, compare strings
Collection	null value, null value as element
Array	null value, null value as element, index $>$ capacity or index $<$ 0
int, byte, short	maximum and minimum values
double, float, ...	overflow, maximum and minimum values, comparison of two values
Integer, Double, ...	null value, overflow, maximum and minimum values, comparison of two values
Class types	null values

9 Robustness

9.1 Introduction

In the preceding sections we assumed that the methods we develop will always be called with arguments satisfying the precondition (`@requires`) of a (sub)specification; then we merely have to guarantee that the end result satisfies the corresponding postcondition (`@ensures`). Providing suitable arguments is the responsibility of the calling environment. However, we can only be assured about its behavior if we have control of the caller ourselves. This will always be the case for the parameters of a private method. If our method is to be publicly released and may be called by software from unknown provenance, it may well be called with arguments that do not satisfy any precondition. The same is true if the method deals with unfiltered user input. Failure to specify what will happen with such unintended arguments then constitutes a security risk. In fact, many malware patterns actively exploit this possibility.

Therefore a program whose use is not totally under our control should provide for its use with argument values that we did not intend; in most cases, such values will make it even impossible to reach the postcondition.

Example 9.1 Suppose we want to make a function that produces a certificate (a fancy document) for a university course (referred to as ‘this course’) that produces two grades. Here grades are integers in the range [1..10]. The course is passed and will merit a certificate if the average of the two grades is at least 6. The specification that expresses this is as follows:

```
/** @desc Produces a certificate based on two grades a and b
*    @requires 1 <= a <= 10
*    @requires 1 <= b <= 10
*    @requires a + b >= 12
*    @ensures \result = a certificate for this course
*    @pure
*/
public Certificate getCertificate(int a, int b))
```

The method’s behavior in this example is only specified for the situation that the values of the parameters `a` and `b` are in the range $[1 \ldots 10]$ and that `a + b` is in the range $[12 \ldots]$. It is at present not specified what happens when these requirements are not met, i.e. the effect of a call to method `getGrading` with for example parameter values `a`$= 11$ and `b`$= -1$. The method might even crash. A specification that does not cater for such inputs is called a *happy path* specification.

A second class of problems can arise outside a programmer’s control but must be taken into account. Here we can think of a database that cannot be reached due to communication problems or a file that needs to be read but does not exist. In these cases, a language like Java itself throws an exception and these must be caught. This needs to be taken into account in both the External View, where the design should be extended with an exception, and the

implementation, where the body of the method in question should be extended with code to handle or throw the exception further.

9.2 Robust versus non-robust software

A program that can withstand incorrect input is called *robust*. It is the designer's choice whether or not to provide a robust version of the software, depending on the amount of control over the calling environment and the seriousness of possible consequences of incorrect use.

Definition 9.1 *Robust* means that for each combination of inputs we know how the program responds – input that satisfies the happy path's precondition as well as input that does not satisfies the happy path's precondition.

Robustness means that the union of all preconditions equals true. To achieve this, we have to add one or more extra subspecifications consisting of precondition and postcondition pairs. These extra subspecifications describe what we mean with wrong input and how the program is to react if it encounters wrong input. The tag `@robust` signals the method is robust.

Not robust means that it is not defined what happens if the precondition is not satisfied. In this case it is enough to define the happy path.

We can choose for a non-robust version of software in cases where we have complete control over the caller, and in cases where we genuinely do not care what happens in case of wrong input. In such cases, we can safely make it the responsibility of the caller.

Example 9.2 In Example 9.1, it is the caller who has to check the values of `a` and `b`. If the values of `a` and `b` satisfy the happy path precondition, the post condition will hold. If the precondition is not true, the result is undefined.

9.3 Making software robust

Note that the happy path specification in Example 9.1 does not impose any obligation on the method to check the values of `a` or `b`. Possibly, it will simply go ahead and produce a certificate with *any* input value. In that case, it is not really surprising that a group of criminals trading in fake credentials has exploited this flaw, succeeding in producing certificates for students whose grades should have led to their failing the course.

In order to limit the use of the method to legal situations, we add subspecifications dealing with all possible inputs that do not satisfy the happy path precondition. As that consists of three separate clauses, we distinguish three possible problem states, each producing an appropriate error message. Note, however, that these overlap: it is entirely possible that more than one clause in the happy path precondition is not satisfied, for example in the call `getCertificate(-1, -1)`. In Subsection 9.4 we shall discuss the problems this presents.

Example 9.3 Adding a separate subspecification for each of the three clauses in the happy path precondition, we get

```
/** @desc Produces a certificate based on two grades a and b
 *    @sub happy path {
 *      @requires 1 <= a <= 10
 *      @requires 1 <= b <= 10
 *      @requires a + b >= 12
 *      @ensures \result = a certificate for this course
 *    }
 *    @sub a not in interval {
 *      @requires !(1 <= a <= 10)
 *      @signals ArgumentOutOfRangeException("first grade not in [1..10]")
 *    }
 *    @sub b not in interval {
 *      @requires!(1 <= b <= 10)
 *      @signals ArgumentOutOfRangeException("second grade not in [1..10]")
 *    }
 *    @sub a+b not in interval {
 *      @requires !(a + b <= 12)
 *      @signals  ArgumentOutOfRangeException("average of grades below 6")
 *    }
 *   @robust
 *   @pure
 */
public Certificate getCertificate(int a, int b)
```

Guidance

- In case it is impossible to satisfy a precondition that will enable the method to achieve its purpose, we may want to signal this by raising an exception. We use the tag `@signals` for this followed by the type of the exception produced. But there are more solutions, for example returning a special value. In the shortest-path algorithm of subsection 10.2 the absence of any path will produce a path of length 0.
- When undesired user input is detected, the client may be offered the opportunity to supply a different input value.
- Adding robustness is not just a matter of adding a few subspecifications to the External View only. These subspecifications call for analysis, design and implementation activities. As part of the analysis, we have to analyze where the information exist to handle the undesired situation effectively. That can be locally or another location in the call chain. Furthermore, we have to decide which form of communication can be applied best, i.e. throwing an exception or returning a special value. As part of the design, we have to extend the signature with extra information about the exception that can be thrown or special values that can be returned. In the implementation, the subspecifications with a `@signals` tag make it necessary to raise an exception in the code. Sometimes, a dedicated exception class has to be implemented.

- It is possible to aim for robustness from the beginning, and take undesired inputs into account in all development activities. However, often it is more practical to ignore robustness until the happy path scenario has been implemented. Then a second pass through all development activities is needed, adding robustness to all artifacts.
- It is a good habit to mark the code needed to make a method robust with inline comments referencing to the subspecifications concerning these robustness aspects. Example 9.4 shows the code's structure that can be used.
- It is good practice to explicitly state in the javadoc if a method is not robust.
- Finally, each extra subspecification should be translated to extra test cases, again applying the equivalent class and boundary value techniques.

Example 9.4 The use of comments referencing to the subspecifications to structure the robustness aspects in code.

```
public Certificate getCertificate(int a, int b)
       throws ArgumentOutOfRangeException {
  // a not in interval
  if (a < 1 || a > 10) {
    throw new ArgumentOutOfRangeException("First␣grade␣not␣in␣[1..10]");
  }
  // b not in interval
  if (b < 1 || b > 10) {
    throw new ArgumentOutOfRangeException("Second␣grade␣not␣in␣[1..10]");
  // a+b not in interval
  if (a+b < 12) {
    throw new ArgumentOutOfRangeException("Average␣of␣grades␣below␣6");
  }
  // happy path
  return new Certificate(a, b);
}
```

9.4 Overlapping subspecifications

In Example 9.3, there are three subspecifications that signal an exception. It is, however, quite possible that more than one of these applies: in a call `getCertificate(-1, 2)` both the case `a not in interval` and the case `a+b not in interval` have their precondition satisfied. This presents a problem as it is not possible to end program execution by throwing two exceptions simultaneously. In theory this could be solved by making these subspecifications non-overlapping: that would require introducing a separate subspecification for the case where the first and third problem appear but not the second, and so on for all possible combinations, thus creating eight subspecifications in all. We consider such proliferation not feasible. Another solution would be to replace the separate

preconditions by a single one consisting of their disjunction, and let that produce an `ArgumentOutOfRangeException` with the message 'one or more arguments do not satisfy one or more value restrictions'. That would result in intolerably vague error messages.

Consequently we propose to let the specification stand as it is in Example 9.3. The semantics of this would be that calling the method with arguments that do not satisfy the happy path precondition produces an exception pinpointing just one of the problems. Which one it is depends on the implementation and is not determined by the specification in this form; moreover, repairing the input value may lead to a new exception pointing to yet another problem. In particular, we do not demand that the program code will check the subspecifications in the order in which they appear in the specification, as doing so – and depending on it – would make the specification far less readable and less understandable.

The above discussion applies only to non-happy path subspecifications that are supposed to signal an exception. Subspecifications that actually contain a desired postcondition should always be made nonoverlapping, in order to prevent contradictory demands on the resulting state.

10 Systems consisting of more than one class

In the preceding sections, we concentrated on programming assignments that could be solved by a single class or even a single method. In practice, object-oriented solutions will most of the time consist of a number of cooperating classes. Yet the documents we have created – the external view, the internal view and the annotated code – all refer to a single class. For a system containing more classes, these documents will have to be provided separately for each of the participating classes. We now discuss at what stage of development the need for additional classes arises.

10.1 Auxiliary classes originating in external design

It may be the case that the requirements for the system to be developed need a second class for expression of the signature in the external design. We have already seen this phenomenon in Example 4.3, where the constructor of class `Person` required a birthdate to be provided in order to make age calculation possible. This led to the constructor signature

```
public Person(String name, Date birthDate)
```

which mentions a class `Date`, of which as yet we know nothing than that it represents a calendar date.

When next we proceed to the internal design of class `Person`, as shown in Subsection 6.2, we see the usefulness of maintaining a private attribute `birthDate` of type `Date`, chosen because it does not need to be updated every year, in contrast to the alternative attribute `age`. The point of having either attribute is that some form of age information is needed to satisfy the external specification of method `getAge`, namely that it returns the person's age.

Having opted for an attribute of type `Date`, we used this to formulate, in Example 6.4, the internal specification of method `getAge` as follows:

```
/**
 * @desc Returns the person's age in years
 * @ensures \result = Date.yearsBetween(Date.today, birthDate)
 * @pure
 */
public int getAge()
```

We now know a lot more of class `Date` – in fact, it is now obvious that we are using it to palm off the hard work involving counting days and years. More precisely stated: we are, in fact, able to give the external design and specification of class `Date`. So now the system under development consists of two classes, of which class `Person` already has a complete internal view, while class `Date` is only available in its external view.

Once the external view of `Date` is available, we may well ask ourselves whether it is worthwhile to build this class ourselves: it may be the case that some library class like `GregorianCalendar` already provides sufficient functionality.

If we do decide to build `Date`, there are at this point two options available to us:

- We may start working on the code of `Person`, leaving work on `Date` until later;
- We may start working on the internal design of `Date`, leaving the implementation of `Person` until later.

In case this assignment is solved by a team rather than a single programmer, both options may also be exercised concurrently. The role of the specifications is precisely to provide a common point of reference for subteams, preventing misunderstandings about what is to be implemented.

10.2 Example: Shortest path

10.2.1 Assignment

The following exercise is taken verbatim from the year 1 course on Imperative Programming at Utrecht University, circa 1999.

> The program should enable the user to find the shortest, or cheapest, route between two locations, given a map of a railway or road or telecom network.

10.2.2 External Analysis

It is not clear from the problem description what ‘a map’ means. The original Utrecht exercise was meant in part to practice working with GUI elements, so there the map was literally implemented as an image; users could selected locations by pointing and clicking. With a view to separation of concerns, we shall omit GUI issues. Moreover, in order to make the program usable in a variety of contexts, we use neutral graph theory terms for the objects that do not imply a choice for railways, roads, or telecom networks. We shall also talk about the length of an edge or a path, despite the fact that this length may actually model something else like the cost of traversing it. Hence the exercise can be stated as the design of a method

```
public Path shortestPath(Node start, Node finish)
```

where classes `Node` and `Path` still have to be designed. However, there is a decision still to be made: are we thinking of a directed or an undirected graph? Railway lines tend to be bidirectional, but roads may be one-way. In order to be as general as possible, we choose a directed graph: this may be used to simulate an undirected graph by defining directed edges in both directions, while the opposite is not possible. Finally, as this is supposed to be an introductory course exercise, we shall assume no knowledge of Dijkstra’s shortest path algorithm and construct a straightforward solution without attention to efficiency.

10.2.3 External Design

A natural place to put the `shortestPath` method is in a class `Network`, where the information about the connections and their length is stored. However, when

exploring a path the information we constantly need on how to proceed is what nodes are reachable with what traversal length from a given node. Therefore we decide to use the class `Network` only for storing the nodes, whereas the reachability information will reside in the individual nodes. This leads to

```
public class Network {
  public Path shortestPath(Node start, Node finish)
  public void addNode(Node newNode)
}
```

For a `Node` it is then important to determine the outgoing edges, say as a set. Hence we need another class `Edge` that has information about its weight and endpoint, so we design

```
public class Node {
  public Set<Edge> getEdges()
  public void addEdge(Edge newEdge)
}

public class Edge {
  public Node getDestination()
  public int getDistance()
}
```

Finally, we need a class `Path`. Conceptually, we can think of a path as a sequence of adjacent nodes or a sequence of linked edges. Which representation is chosen will be decided in the internal design phase. Because we are interested in a shortest path, we also need to keep track of path lengths. It will clearly also be necessary to determine whether the given destination is on a candidate path. We add a boolean function to check this.

```
public class Path {
  public int getLength()
  public boolean contains(Node node)
  public void addNode(Node newNode)
}
```

10.2.4 External Specification

The various classes and methods we have introduced need to be specified in terms of the problem domain, i.e. the directed graph. This leads to

```
/**
 * @desc Contains a set of nodes and enables the calculation
 * of shortest paths in a network
 */
public class Network {
  /** @ensures \result = a shortest path between start and finish */
  public static Path shortestPath(Node start, Node finish)
  /** @ensures newNode has been added to the network */
  public void addNode(Node newNode)
}

/**
 * @desc Models a node together with its outgoing edges
```

```
 */
public class Node {
  /** @ensures \result = the set of edges leading from this node
      @pure
   */
  public Set<Edge> getEdges()
  /** @ensures newEdge has been added as an outgoing edge from this node */
  public addEdge(Edge newEdge)
}

/**
 * @desc Models a directed edge in the graph
 */
public class Edge {
  /** @ensures \result = the edge's endpoint
      @pure
   */
  public Node getDestination()
  /** @ensures \result = the edge's length
      @pure
   */
  public int getDistance()
}

/**
 * @desc Models a sequence of nodes linked by edges
 */
public class Path {
  /** @ensures \result = the total length of the path's edges
      @pure
   */
  public int getLength()
  /** @ensures \result = node occurs on the path
      @pure
   */
  public boolean contains(Node node)
  /** @ensures newNode has been added to the end of the path
   *  and its length has been increased by addLength
   */
  public void addNode(Node newNode, int addLength)
}
```

10.2.5 Internal Analysis

We have already decided that `Path` models a sequence of nodes linked by edges; and that the edges will be administered by the nodes from which they depart. The remaining decision is to choose a suitable collection type for the nodes in the path. Here we may observe that, on account of the decisions already taken, paths may be extended only from their endpoint onward. Therefore it is simplest to store the nodes of the path in a `Stack`, with the endpoint as the top element. Extending the path is then just a `push` operation.

As to the nodes, we stated above that each node would feature a set of edges. However, `java.util.Set` is an interface, so we need to choose an implementation, for instance `java.util.HashSet`.

10.2.6 Internal Design

Adding the proper attributes to the external design, we get

```
public class Network {
  private HashSet<Node> nodes;
  public Path shortestPath(Node start, Node finish)
  public void addNode(Node newNode)
}

public class Node {
  private String name;
  private HashSet<Edge> edges;
  public Set<Edge> getEdges()
  public void addEdge(Edge newEdge)
}

public class Edge {
  private Node destination;
  private int distance;
  public Node getDestination()
  public int getDistance()
}

public class Path {
  private Stack<Edge> edges;
  private int length;
  private Node start;
  public int getLength()
  public boolean contains(Node node)
  public void addNode(Node newNode)
}
```

10.2.7 Internal Specification

When moving from the external view to the internal view, the information we need to add is to record how the attributes we have just added correspond to the graph theoretical concepts from the problem domain. To this end, we add the following comments:

```
public class Node {
 /** @represents edges is the set of all edges leading from this node */
  private HashSet<Edge> edges;
}

public class Edge {
 /**
  * @represents The end point of the edge is destination,
  * its length is distance
  */
  private Node destination;
  private int distance;
}

public class Path {
 /**
```

```
   * @represents The path consists of the contents of attribute
   * nodes in reverse order
   * @inv length is the total length of the path's edges
   */
  private Stack<Edge> edges;
  private int length;
  private Node start;
}
```

Here we have left out the method specifications: either these were already adequately specified in the external specification or they are just getters and setters whose semantics are obvious. In practice, all headings and specifications will reside in the same Java file, so there is never a need to copy previously written lines.

10.2.8 Code Analysis

As a first approach, we just examine all the paths in the graph (in any order). Whenever one of these paths reaches the required destination `finish`, its length is compared to any previously found completed path with a view to determining a shortest one. The only thing we must guard against is the possibility of cyclic paths: whenever we are tempted to add a node already on the path, this possibility is rejected.

This solution is not efficient. However, for application to a graph of limited size such as the national railway network its speed and storage requirements are adequate. Storage may be minimized by observing that many of the paths being built share their beginning nodes. As these never change, they may actually share that part rather than retain their own copy. Speed can be optimized that a shortes path to the final destination also contains a shortest path to each of its nodes. Therefore whenever we add a node to a path, we may immediately delete all longer paths to that node because these will never be part of a solution.

10.2.9 Code Design

During the computation of a shortest path we need to keep track of all the partial paths built so far. The simplest idea would be to use a set as well, but during the computation this set is modified all the time by removing one path and adding all its single-edge extensions. Due to the dynamic nature of the set, it is not feasible to do this in a simple `for`-statement iterating over its elements. What we in fact need is the operation 'pick an arbitrary element from the set', which is not provided in the interface. An easy way to solve this is not to take a set but once again a stack; then the `pop` operation does exactly what we need. However, do observe that the order in which the paths occur in this stack is entirely immaterial.

With this choice, our plan for the code becomes

```
public Path shortestPath(Node start, Node finish) {
  // keep track of set of paths under investigation
  Stack<Path> paths = new Stack<Path>();
  // remember the shortest path to finish seen so far
```

```
  Path shortest = new Path(start);
  do {// take one of the paths under investigation
    // if it reaches finish
    // compare it to the shortest nonzero path so far
    // if it does not reach finish and is not cyclic
    // add all one-edge extensions to the set
  } while (!paths.isEmpty());
  return shortest;
}
```

10.2.10 Coding

After these considerations, the algorithm can be coded in straightforward manner. It would be redundant to insert an implementation design where the algorithm is split up into separate steps that could be implemented as private methods.

```
public Path shortestPath(Node start, Node finish) {
  // keep track of set of paths under investigation
  Stack<Path> paths = new Stack<Path>();
  // remember the shortest path to finish seen so far
  Path shortest = new Path(start);
  paths.add(shortest);
  do {// take one of the paths under investigation
    Path path = paths.pop();
    // if it reaches finish
    // compare it to the shortest nonzero path so far
    if (path.contains(finish)) {
      int min = shortest.getLength();
      if (min == 0 || path.getLength() < min) shortest = path;
    }
    // if it does not reach finish and is not cyclic
    // add all one-edge extensions to the set
    else {
      Set<Edge> edges = path.getDestination().getEdges();
      for (Edge edge: edges) {
        Node destination = edge.getDestination();
        int addLength = edge.getDistance();
        if (!path.contains(destination)) {
          Path newPath =
          path.addNode(destination, addLength);
          paths.push(newPath);
        }
      }
    }
  } while (!paths.isEmpty());
  return shortest;
}
```

From this code it becomes clear what additional tasks we need class `Path` to perform. Apart from `getLength()` and `contains(Node node)`, whose usefulness we had already foreseen in the external design phase, it turns out that with the present algorithm it becomes necessary to have a method `addNode(Node newNode, int addLength)` that extends the given path with an edge of length `addLength` to node `newNode`.

Note that, while we have finished class `Network`, we are still adding functionality to the external design of class `Path`. As argued in Subsection 10.1, not all classes in a system need to be at the same level of development simultaneously.

Adding a node to the path must be done in a nondestructive manner, because later on we will want to extend the same original path with a different node. The implementation of this new method `addNode` then becomes

```
public Path addNode(Node newNode, int addLength) {
  Path result = clone();
  result.nodes.push(newNode);
  result.length += addLength;
  return result;
}
```

Here `clone` produces a copy of the current path. It overrides `Object.clone` and can be defined by

```
@Override protected Path clone() {
  Path result = new Path();
  result.nodes = (Stack<Node>) nodes.clone();
  result.length = length;
  return result;
}
```

10.2.11 Robustness

A major defect of the external specification as given is that it may not be possible to satisfy the requirement. When there exists no path at all between `start` and `finish`, there is no shortest path either. The specification should definitely take this possibility into account. Moreover, its occurrence is not an indication of an error or defect, so throwing an exception is not a proper response. Given that the return type of `shortestPath` is `Path`, the question arises if it is possible to signal this situation through the `Path` returned.

Looking at the code of `shortestPath`, we see that in the absence of paths reaching `finish` the method will return a path of length 0 consisting of the single node `start`. Since this cannot be the right answer (except in the useless case where the user explicitly asks for a path that ends where it starts), we may declare ourselves satisfied with this outcome and we do not need to change the code. (We admit to a degree of opportunism here.) However, the case needs to be recorded separately in the specification, as follows:

```
/** @sub connected {
 * @requires there is a path between start and finish
 * @ensures \result = a shortest path between start and finish
 *}
 * @sub unconnected {
 *  @requires there is no path between start and finish
 *  @ensures \result = the path from start of length 0
 *}
*/
public static Path shortestPath(Node start, Node finish)
```

10.2.12 Test construction

As the methods in the auxiliary classes are very simple, we only discuss testing of method `shortestPath`. It is clear that, even for a test of the happy path scenario, we will need to construct a graph, using the methods of `Network`, `Node` and `Edge`, before we can check whether `shortestPath` really returns the shortest path in the graph. So any call to `testShortestPath` will have to be preceded by a `setUp` phase which builds the desired graph.

However, after the discussion of robustness, we see that we also have to check the case where there is no path at all between certain nodes (subspecification `unconnected`) and the case where there is more than one shortest path (so that subspecification `connected` is nondeterministic).

Following our test approach, testing the happy path scenario, i.e. there is a connection between two nodes, we have two equivalent classes: deterministic and nondeterministic. There are no boundaries. For the non happy path scenario, we have one equivalent class: unconnected.

A very simple test class satisfying these requirements would look like this:

```
public class NetworkTest {

  private Node roermond;
  private Node venlo;
  private Node nijmegen;
  private Node eindhoven;
  private Network network;

  private void setUp() {
    roermond = new Node("Roermond");
    weert = new Node("Weert");
    venlo = new Node("Venlo");
    nijmegen = new Node("Nijmegen");
    eindhoven = new Node("Eindhoven");
    roermond.addEdge(new Edge(venlo, 26));
    roermond.addEdge(new Edge(weert, 22);
    roermond.addEdge(new Edge(eindhoven, 33));
    weert.addEdge(new Edge(eindhoven, 11));
    venlo.addEdge(new Edge(nijmegen, 55));
    eindhoven.addEdge(new Edge(nijmegen, 57));
    network = new Network();
    network.addNode(roermond);
    network.addNode(weert);
    network.addNode(venlo);
    network.addNode(nijmegen);
    network.addNode(eindhoven);
  }

  @Test
  public void testShortestPathConnected() {
    setUp();
    //deterministic
    Path path = network.shortestPath(roermond, nijmegen);
    assertEquals("␣Roermond␣Venlo␣Nijmegen", path.toString());
    //nondeterministic
    path = network.shortestPath(roermond, eindhoven);
```

```
    assertTrue(path.toString.equals("␣Roermond␣Weert␣Eindhoven") ||
    path.toString.equals("␣Roermond␣Eindhoven"))
  }

  @Test
  public void testShortestPathUnconnected() {
    setUp();
    path = network.shortestPath(nijmegen, roermond);
    assertEquals("␣Nijmegen", path.toString());
  }
}
```

10.3 Auxiliary classes originating in internal design

In the preceding example the auxiliary classes followed naturally from the method signatures necessary to realize the system requirements. But there are also cases where the division of responsibilities between classes is geared towards nonfunctional software quality characteristics, such as maintainability and changeability. Designing such a system requires more creativity. The classes needed are then not immediately obvious from the exercise text, and will only emerge during the internal design. Knowledge of UML class diagrams and design patterns definitely helps. The next subsection contains an example.

10.4 Example: Rental agency

10.4.1 Assignment

The following exercise is taken verbatim from the 2nd year course on Modeling and System Development at Utrecht University, circa 1999.

> A company rents out houses and boats. Some of these are the property of the company, others belong to private owners. The company pays taxes only for the objects it owns. Taxes are calculated from value, differently for houses and boats.

10.4.2 External Analysis

It is implicit in the exercise text that the point is to calculate the tax due on any rental object. Hence the solution should contain a method

```
public int tax()
```

that calculates the amount of tax to be paid. We use the return type `int` because tax amounts are always calculated in whole euros, and never so high that overflow becomes a concern.

10.4.3 External Design

We cannot just put this method into an ordinary class, because the constructor has to be different for a company property (whose value needs to be recorded)

and a private property (for which this information is not known). Therefore we need an abstract class with two subclasses that both contain this method but have different-looking constructors.

According to tis analysis, as a first approximation we get:

```
public abstract class Rental {
  public abstract int tax();
}

public class CompanyProperty extends Rental {
  private int value;
  public CompanyProperty(int value)
  public int tax()
}

public class PrivateProperty extends Rental {
  public PrivateProperty()
  public int tax()
}
```

10.4.4 External Specification

In terms of the application domain, a company property is one that belongs to the rental agency, a private property belongs to other owners that use the services of the company in securing rental contracts. For company properties, the value in the sense of tax legislation is known. Adding these as comments, we get:

```
/**
 * @desc This class contains the information about
 * a property (boat or house) being rented out by the
 * agency and calculate the amount of tax to be paid
 */
public abstract class Rental {
  /**
   * @ensures \result = the amount of tax to be paid
     @pure
   */
   public abstract int tax();
}

/**
 * @desc A CompanyProperty is a Rental owned by the company
 */
public class CompanyProperty extends Rental {

 /**
  * @requires value is the legal value in euros
  */
  public CompanyProperty(int value)
  public int tax()
}

/**
 * @desc A PrivateProperty is a Rental not owned by the company
```

```
 */
public class PrivateProperty extends Rental {
  public PrivateProperty()
  public int tax()
}
```

10.4.5 Internal Analysis

Calculating tax is something that only has to be done for company properties: for private properties the tax is always 0. The remaining problem for company properties is that the algorithms for calculating the tax from the value uses a different algorithm in the case of houses and of boats. We could solve this by using a `switch` statement in the method's body, but that would necessitate changing the code when the company decides to rent out other properties such as recreational vehicles. Therefore, according to the Open-Closed Principle, we had better put the algorithms into separate subclasses, say `CompanyHouse` and `CompanyBoat`.

However, in that solution it is hardwired that the distinction between houses and boats will play no role whatsoever for private properties. This may not be true for future extensions: for instance, the company would like to keep track of the amount of fuel provided when renting out a powered boat. Thus, merely to cater for plausible future developments, it is preferable to put the information about the nature of the physical object rented out into a separate entity, which we will call its rental type.

10.4.6 Internal Design

According to our analysis, we decide that the actual calculation of the tax will be delegated to separate classes `House` and `Boat`. When creating a property, its rental type will have to be communicated to the constructor. To avoid explicit case analysis, the two rental types will share a common interface. This leads to the following structure, in which readers familiar with design patterns will recognize the *Bridge* pattern.

The code corresponding to this is

```
public abstract class Rental {
  protected RentalType type;
  public abstract int tax();
}

public class CompanyProperty extends Rental {
  private int value;
  public CompanyProperty(RentalType type, int value)
  public int tax() {return type.calculateTax(value);}
}

public class PrivateProperty extends Rental {
  public PrivateProperty(RentalType type)
  public int tax() {return 0;}
}
```

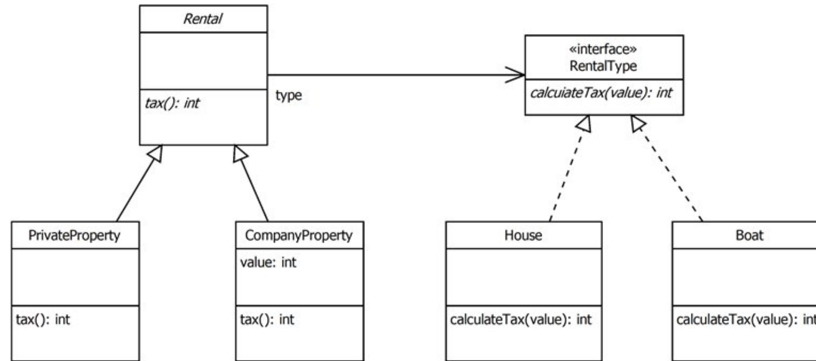

Figure 5: UML class diagram of the design

```
public interface RentalType {
  public int calculateTax(int value);
}

public class House implements RentalType {
  public int calculateTax(int value)
}

public class Boat implements RentalType {
  public int calculateTax(int value)
}
```

10.4.7 Internal Specification

The task of the internal specification is, first, to show how the entities from the problem domain are represented by the class attributes, and next, how the external specifications can be translated into specifications in terms of these attributes. The first task presents no problems: to class `Rental` we add

```
/**
 * @represents type is a House or a Boat, according
 * to the sort of physical object
 */
protected RentalType type;
```

and to class `CompanyProperty` we add

```
/**
 * @represents value is the legal value in euros
 */
private int value;
```

In order to produce specifications for the various tax-related methods, however, we cannot proceed further with eliminating domain concepts without going into the precise rules for computing tax on a house or boat. This part was not

included in the original exercise, since that only had the purpose to practise with a realistic instance of the *Bridge* pattern.

For the sake of the argument, let us assume that the tax on a house is 0.85% of its legal value, the tax on a boat is 1.52% of its value. (In reality, things are very much more complicated and subject to local variations.) However, the 'legal value' of a house can also be negative, e.g. in the case where the mortgage burden exceeds the market price. For houses with a negative legal value, no tax is required. This directly leads to two separate cases in the specification of method `calculateTax` in class `House`:

```
/** @sub nonnegative {
 *    @requires value >= 0
 *    @ensures \result = Math.round(0.0085 * value)
 *    @pure
 * }
 * @sub negative {
 *    @requires value < 0
 *    @ensures \result = 0
 *    @pure
 * }
 */
public int calculateTax(int value)
```

10.4.8 Robustness

There is no mortgage exception in the tax laws for boats; hence, a boat can never have a negative value. (We repeat that nothing in this text accurately reflects the tax laws of any country.) This means that attempting to create a `Boat` object with a negative value is an error that must be signaled to the enclosing program in order to take corrective measures. So for the constructor of class `CompanyProperty` we get

```
/**
 * @sub possible {
 *    @requires value >= 0 || type instanceof House
 * }
 * @sub impossible {
 *    @requires value < 0 && type instanceof Boat
 *    @signals ValueException("Boat with negative value")
 * }
 */
public CompanyProperty(RentalType type, int value)
```

Similarly, method `calculateTax` of class `Boat` should signal a `ValueException` in case its argument `value` takes a negative value.

```
/**
 * @sub nonnegative {
 *    @requires value >= 0
 *    @ensures \result = Math.round(0.0152 * value)
 *    @pure
 * }
 * @sub negative {
 *    @requires value < 0
```

```
 *    @signals ValueException("Boat with negative value")
 *    @pure
 * }
 */
public int calculateTax(int value)
```

Observe that case `negative` will never arise as long as

- this method is only ever called with the attribute `value` from class `CompanyProperty` as argument,
- there is no way to change the attribute `value` from class `CompanyProperty` once the object has been constructed.

In the present state of the software, these conditions are met. Hence there is no way method `calculateTax` of class `Boat` can raise an exception. Still, the subspecification for this case needs to be present to cater for future extensions. One reason for this is that Java does not allow to specify that a method can only be called from one specific code location.

10.4.9 Coding

After the work we have now done, producing the code is trivial. For an example, the code of method `calculateTax` of class `House` is merely

```
public int calculateTax(int value) {
  int tax = 0;
  if (value >= 0)        tax = Math.round(0.0085 * value);
  return tax;
}
```

10.4.10 Test construction

As the rental objects naturally fall into four different categories, method `tax` should be called for a house and for a boat, each with private as well as company ownership. In the case of company-owned houses, we need to check separately for positive and negative value. Moreover, to ensure robustness, we should also check that it is not possible to create a company-owned boat with negative value. Moreover, to guarantee robustness in case class `Boat` will in future be called by other classes than `CompanyProperty` using a negative `value` argument, we will also check whether this will produce the required exception. This leads to a test class of the form, where the names of the test methods refer to the corresponding subspecifications:

```
public class RentalTest {
  @Test
  public void testTax() {
    // house taxes
    House house = new House();
    Rental privateHouse = new PrivateProperty(house);
    assertEquals(0, privateHouse.tax());
```

```
    Rental positiveHouse = new CompanyProperty(house, 10000);
    assertEquals(85, positiveHouse.tax());
    Rental negativeHouse = new CompanyProperty(house, -10000);
    assertEquals(0, negativeHouse.tax());

    // boat taxes
    Boat boat = new Boat();
    Rental privateBoat = new PrivateProperty(boat);
    assertEquals(0, privateBoat.tax());
    Rental companyBoat = new CompanyProperty(boat, 10000);
    assertEquals(125, companyBoat.tax());
  }
}

public class CompanyPropertyTest() {
  @Test
  public void testConstructorImpossible() {
    Boat boat = new Boat();
    assertThrows(ValueException.class, () -> {
      Rental negativeBoat = new CompanyProperty(boat, -10000);
    }
  }
}

public class BoatTest() {
  @Test
  public void testCalculateTaxNonnegative() {
    Boat boat = new Boat();
    assertEquals(125, boat.calculateTax(10000));
  }

  @Test
  public void testCalculateTaxNegative() {
    Boat boat = new Boat();
    assertThrows(ValueException.class, () -> {
      boat.calculateTax(-10000);
    }
  }
}
```

Note that the `assertThrown` syntax assumes JUnit version 5. The classes testing exceptions should therefore be preceded with

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
```

10.5 Developing larger OO systems

If a larger system has to be developed consisting of, say, five or more classes, the discussed method alone is not sufficient. In such a situation, an overview must be firstly obtained of the classes that are needed and the relationships between them. For example, the Unified Process (UP) [7] can be used for this.

In UP, one or more use cases are first drawn up and a domain model is created. Then a design class diagram is made on the basis of system sequence diagrams and communication diagrams. This design class diagram consists of

classes with associations provided with communication direction, attributes and methods. A design class diagram largely corresponds to the External Design (classes and public method signatures) and the Internal Design (attributes representing the associations to other classes).

This is the starting point for applying our method. Each class can then be further developed: descriptions can be added to the classes and methods expressing their responsibilities, the specifications can be added to the External and Internal Design completing the External View and Internal View, and the Code View can be realized. Each class can be provided with unit tests based on the three views.

10.6 Test execution for multi-class systems

Although test construction is an activity that can take place in all phases of development, whenever sufficient information about the program structure becomes available, executing the tests requires the availability of code. Not only must the method being tested be implemented before its test can be run, any other methods being called from its code must also be executable. Such a limitation engenders a strict bottom-up order of testing, beginning with the methods at the leaves of the call tree and step-by-step working up to the root, which corresponds to the original requirement. This is inefficient because it forces programmers working on higher node to postpone testing until subsidiary functionality is implemented. And it may not even be possible in cases where there is mutual dependence between methods.

Therefore testing a multi-class system often necessitates the use of temporary simulations that may take the place of methods not yet implemented.

Definition 10.1 A *stub* is a very simple method that can take the place, during testing, of a method to be developed. A stub produces plausible result of the correct type, but typically does not execute complex algorithms or interface with relational databases.

An added advantage of the use of stubs rather than finished implementations is that it enables testing the calling method with different arbitrary values satisfying the specification of the called method, rather than just with the specific values produced by its implementation, which may not cover the entire range allowed by the specification. In the latter case it may be that the calling method functions well within the system as it has been written, but develops unforeseen problems when the called method is replaced with a different implementation.

A For the teacher

A.1 How to apply the procedure in a curriculum?

We think the procedure can be taught in the following ways:

1. The procedure can be taught as a whole as part of a Software Engineering course. Such a course is typically placed in a second or third study year and requires prior knowledge and experience with programming and testing.

2. The procedure is scattered over several courses. Examples are:

   - Functions are discussed in an introductory programming course. In addition to teaching the syntax and the use of functions, the first column (External View) and the third column (Code View) can be applied from the procedure. These views indicate which aspects of a function can be distinguished, in which order they can be developed, how this leads to a design, specifications, implementation and associated tests, and documentation.
   - In a first year OO programming course, classes and methods are introduced. Here the approach may be limited to developing an External View and Code View with associated tests. Attributes are added to classes, but describing them precisely in the form of an Internal View is postponed until a follow-up course (see the next bullet).
   - In a course about data structures and algorithms, the Internal View is of special interest besides the External View and Code View. Here, the translation from domain variables to attributes and corresponding specifications is a critical activity.

A.2 Didactic aspects

The procedure itself is for beginning students a complex thing and learning the concepts underlying the procedure and how to use the procedure cost time. It is important to introduce the procedure using simple examples, so the student has to deal with one complexity. Once the procedure is understood, it can be used to solve more complex exercises. It is important to understand that the use of the procedure is not a goal in itself; instead, it provides guidance where the complexity is great.

The procedure can be explained through traditional classroom teaching and practiced by homework assignments that students work on alone or in groups.

The procedure can also be taught by having the entire class, or groups of students, work together on assignments and frequently, after each procedural step, share their results and discuss these. We think this approach is most effective. Together, many more aspects are found that need to be taken into account in a design and specification. It is precisely this experience that is important!

References

[1] K. Beck. *Test-driven Development: by Example*. Addison-Wesley Professional, 2003.

[2] A. Bijlsma, H.J.M. Passier, H.J. Pootjes, and S. Stuurman. Integrated test development: An integrated and incremental approach to write software of high quality. In *7th Computer Science Education Research Conference (CSERC 2018*, pages 9–20, 2018.

[3] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, 2002.

[4] D.E. Knuth. Literate Programming. *The Computer Journal*, 27:97–111, 1984.

[5] P. Kruchten. *The Rational Unified Process: an Introduction*. Addison-Wesley Professional, 2004.

[6] A. van Lamsweerde. *Requiements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[7] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, 2004.

[8] G.T. Leavens and C. Baker, A.L. Ruby. JML: A notation for detailed design. In *Behavioral specifications of Businesses and Systems*, pages 175–188. Springer, 1999.

[9] B. Liskov and J. Guttag. *Program development in JAVA: abstraction, specification, and object-oriented design*. Pearson Education, 2000.

[10] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16:1811–1841, 1994.

[11] J.G. Merriënboer and P.A. Kirschner. *Ten Steps to Complex Learning*. Routledge, 2007.

[12] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[13] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[14] J.M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[15] M. Ward and H. Zedan. Transformational programming and the derivation of algorithms. In *Computer and Information Sciences*, pages 17–22. Springer, 2011.

[16] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.