CHAPMAN & HALL/CRC
TEXTBOOKS IN COMPUTING

FLEXIBLE, RELIABLE SOFTWARE

Using Patterns and Agile Development

Henrik Bærbak Christensen

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

The TDD Rhythm

The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

Essential TDD Principles

TDD Principle: **Test First**

When should you write your tests? Before you write the code that is to be tested.

TDD Principle: **Test List**

What should you test? Before you begin, write a list of all the tests you know you will have to write. Add to it as you find new potential tests.

TDD Principle: **One Step Test**

Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

TDD Principle: **Isolated Test**

How should the running of tests affect one another? Not at all.

TDD Principle: **Evident Tests**

How do we avoid writing defective tests? By keeping the testing code evident, readable, and as simple as possible.

TDD Principle: **Fake It ('Til You Make It)**

What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

TDD Principle: **Triangulation**

How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.

TDD Principle: **Assert First**

When should you write the asserts? Try writing them first.

TDD Principle: **Break**

What do you do when you feel tired or stuck? Take a break.

TDD Principle: **Evident Data**

How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

TDD Principle: **Obvious Implementation**

How do you implement simple operations? Just implement them.

TDD Principle: **Representative Data**

What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or a special computational processing.

TDD Principle: **Automated Test**

How do you test your software? Write an automated test.

TDD Principle: **Test Data**

What data do you use for test-first tests? Use data that makes the tests easy to read and follow. If there is a difference in the data, then it should be meaningful. If there isn't a conceptual difference between 1 and 2, use 1.

TDD Principle: **Child Test**

How do you get a test case running that turns out to be too big? Write a smaller test case that represents the broken part of the bigger test case. Get the smaller test case running. Reintroduce the larger test case.

TDD Principle: **Do Over**

What do you do when you are feeling lost? Throw away the code and start over.

TDD Principle: **Regression Test**

What's the first thing you do when a defect is reported? Write the smallest possible test that fails and that, once run, will be repaired.

FLEXIBLE, RELIABLE SOFTWARE

Using Patterns and Agile Development

CHAPMAN & HALL/CRC TEXTBOOKS IN COMPUTING

Series Editors

John Impagliazzo
ICT Endowed Chair
Computer Science and Engineering
Qatar University

Professor Emeritus, Hofstra University

Andrew McGettrick
Department of Computer
and Information Sciences
University of Strathclyde

Aims and Scope

This series covers traditional areas of computing, as well as related technical areas, such as software engineering, artificial intelligence, computer engineering, information systems, and information technology. The series will accommodate textbooks for undergraduate and graduate students, generally adhering to worldwide curriculum standards from professional societies. The editors wish to encourage new and imaginative ideas and proposals, and are keen to help and encourage new authors. The editors welcome proposals that: provide groundbreaking and imaginative perspectives on aspects of computing; present topics in a new and exciting context; open up opportunities for emerging areas, such as multi-media, security, and mobile systems; capture new developments and applications in emerging fields of computing; and address topics that provide support for computing, such as mathematics, statistics, life and physical sciences, and business.

Published Titles

Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph, Foundations of Semantic Web Technologies

Uvais Qidwai and C.H. Chen, Digital Image Processing: An Algorithmic Approach with MATLAB®

Henrik Bærbak Christensen, Flexible, Reliable Software: Using Patterns and Agile Development

CHAPMAN & HALL/CRC
TEXTBOOKS IN COMPUTING

FLEXIBLE, RELIABLE SOFTWARE

Using Patterns and Agile Development

Henrik Bærbak Christensen

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business
A CHAPMAN & HALL BOOK

The cover picture shows the first pyramid ever built, Pharaoh Djoser's step pyramid. It was built around 2600 BC by Djoser's chancellor, Imhotep. Imhotep was the first engineer and architect in history known by name, and he was deified almost 2000 years after his death. Imhotep's ingenious idea was to reuse the existing tomb design of a flat-roofed, rectangular structure, the *mastaba*, and create the royal tomb by building six such mastabas of decreasing size atop one another. You can still admire the pyramid at Saqqara, Egypt, today, more than 4600 years after it was completed. It is a design that has stood the test of time from an architect who was a deified-worthy role model for all who create designs and realize them.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2010 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20110715

International Standard Book Number-13: 978-1-4398-8272-6 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To my children,
Mikkel, Magnus, and Mathilde

for defining the ultimate purpose
a man can assume…

To my wife,
Susanne

for the greatest in life:
to love and be loved in return…

Contents

Foreword

by Michael Kölling

Teaching to program well is a hard challenge. Writing a book about it is a difficult undertaking.

The bulk of my own experience in programming teaching is at the introductory level. I see my students leave the first programming course, many of them thinking they are good programmers now. Most of them are not. Only the good ones realise how much they have yet to learn. Learning how to build good quality software is much more than mastering the syntax and semantics of a language. This book is about the next phase of learning they are about to face.

Most academic discussion about the teaching of programming revolves around introductory teaching in the first semester of study, and by far the largest number of books on programming cover the beginners' aspects. Many fewer books are available that cover more advanced aspects—as this one does—and even fewer do it well.

The reason is just that introductory programming is easier to handle, and still so difficult that for many years we—as a teaching community—could not agree how to approach the teaching of modern, object-oriented programming in a technically and pedagogically sound manner. It has taken well over 10 years and more than one hundred published introductory textbooks on learning object orientation with Java alone to get to where we are now: a state where introductory texts are available that are not only a variation of a commented language specification, but that follow sound pedagogical approaches, that are written with learners in mind, that emphasise process over product, and that deal with real problems from real contexts.

For more advanced programming books—usable in advanced programming courses—the situation is less rosy. There is much less agreement about the topics that such a course should cover, and fewer authors have taken the difficult step to write such a book. Many programming books at this level are in character where introductory books were ten years ago: Descriptions of techniques and technologies, written with great emphasis on technical aspects, but with little pedagogical consideration.

This book is a refreshing change in this pattern. This book brings together a careful selection of topics that are relevant, indeed crucial, for developing good quality software with a carefully designed pedagogy that leads the reader through an experience of active learning. The emphasis in the content is on practical goals—how to construct reliable and flexible software systems—covering many topics that every software engineer should have studied. The emphasis in the method is on providing a practical context, hands on projects, and guidance on *process*.

This last point—process—is crucial. The text discusses not only what the end product should be like, but also how to get there.

I know that this book will be a great help for many of my students on the path from a novice programmer to a mature, professional software developer.

—Michael Kölling
Originator of the BlueJ and Greenfoot Environments.
Author and coauthor of the best-selling books
Objects First with Java and
Introduction to Programming with Greenfoot.

Preface

Mostly for the Students...

This is a book about designing and programming flexible and reliable software. The big problem with a book about making software is that you do not learn to make software—by reading a book. You learn it by reading about the techniques, concepts and mind-sets that I present; apply them in practice, perhaps trying alternatives; and reflect upon your experiences. This means you face a lot of challenging and fun programming work at the computer! This is the best way to investigate a problem and its potential solutions: programming is a software engineer's laboratory where great experiments are performed and new insights are gained.

I have tried to give the book both a practical as well as an theoretical and academic flavor. Practical because all the techniques are presented based on concrete and plausible (well, most of the time) requirements that you are likely to face if you are employed in the software industry. Practical because the solution that I choose works well in practice even in large software projects and not just toy projects like the ones I can squeeze down into this book. Practical because the techniques I present are all ones that have been and are used in practical software development. Theoretical and academic because I am not satisfied with the first solution that I can think of and because I try hard to evaluate benefits and liabilities of all the possible solutions that I can find so I can pick the best. In your design and programming try to do the same: Practical because you will not make a living from making software that does not work in practice; academic because you get a better pay-check if your software is smarter than the competitors'.

Mostly for the Teachers...

This book has an ambitious goal: to provide the best learning context for a student to become a good software engineer. Building flexible and reliable software is a major challenge in itself even for seasoned developers. To a young student it is even more challenging! First, many software engineering techniques are basically solutions to problems that an inexperienced programmer has never had. For instance, why introduce a design pattern to increase flexibility if the program will never be maintained as is the case with most programming assignments in teaching? Second, real software design and development require numerous techniques to be combined—picking the right technique at the right time for the problem at hand. For instance, to do automated testing and test-driven development you need to decouple abstractions and thus pick the right patterns—thus in practice, automated testing and design patterns benefit from being combined.

This book sets out to lessen these problems facing our students. It does so by *story telling* (Christensen 2009), by explaining the design and programming *process*, and by using *projects* as a learning context. Many chapters in the book are telling the story of a company developing software for parking lot pay stations and the students are invited to join the development team. The software is continuously exposed to new requirements as new customers buy variations of the system. This story thus sets a natural context for students to understand why a given technique is required and why techniques must be combined to overcome the challenges facing the developers. An agile and test-driven approach is applied and space is devoted to explaining the programming and design process in detail—because often *the devil is in the detail.* Finally, the projects in the last part of the book define larger contexts, similar to real, industrial, development, in which the students via a set of assignments apply and learn the techniques of the book.

A Tour of the Book

The book is structured into nine parts—eight *learning iterations*, parts 1–8, and one *project* part, part 9. The eight learning iterations each defines a "release" of knowledge and skills that you can use right away in software development as well as use as a stepping stone for the next learning iteration. An overview of the learning iterations and their chapters is outlined in Figure 1. The diagram is organized having introductory topics/chapters at the bottom and advanced topics/chapters at the top. Chapters marked with thick borders cover core topics of the book and are generally required to proceed. Chapters marked with a gray background cover material that adds perspective, background, or reflections to the core topics. The black chapters are the project chapters that define exercises.

For easy reference, an overview of the rhythm and principles of test-driven development is printed as the first two pages, and an index of all design patterns at the last page. In addition to a normal index, you will also find an index of sidebars and key points at the end of the book.

Learning iteration 1 is primarily an overview and introduction of basic terminology that are used in the book. Learning iterations 2 to 5 present the core practices, concepts, tools and analytic skills for designing flexible and reliable software. These iterations use a *story telling* approach as they unfold a story about a company that is producing software for pay stations, a system that is facing new requirements as time passes. Thus software development techniques are introduced as a response to realistic challenges. Learning iteration 6 is a collection of design patterns—that can now be presented in a terse form due to the skills acquired in the previous iterations. The learning focus of iteration 7 is frameworks which both introduce new terminology as well as demonstrate all acquired skills on a much bigger example, MiniDraw. Learning iteration 8 covers two topics that are important for flexible and reliable software development but nevertheless are relatively independent of the previous iterations.

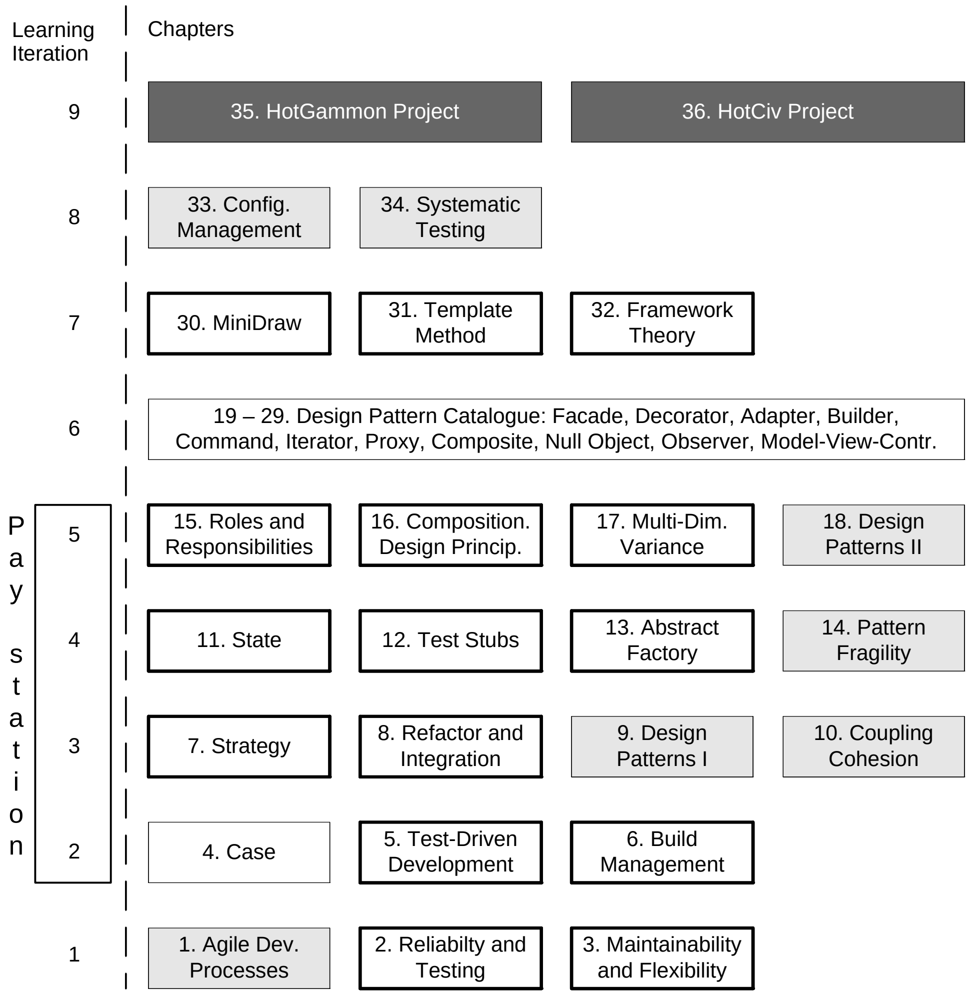

Figure 1: Overview of learning iterations and chapters.

Part 9, *Projects*, defines two large project assignments. These projects are large systems that are developed through a set of assignments covering the learning objectives of the book. Each project is structured into seven releases or iterations that roughly match learning iteration 2 to 8 of the book. Thus by completing the exercises in, say, project HotCiv's iteration on frameworks you will practice the skills and learning objectives defined in the framework learning iteration of the book. If you complete most or all iterations in a project you will end up with a reliable and usable implementation of a large and complex software system, complete with a graphical user interface. The HotGammon project will even include an opponent artificial intelligence player.

Each learning iteration starts with an overview of its chapters, and in turn each chapter follows a common layout:

- *Learning Objectives* state the learning contents of the chapter.
- Next comes a presentation and discussion of the new material usually ending in a section that discusses benefits and liabilities of the approach.

- *Summary of Key Concepts* tries to sum up the main concepts, definitions, and results of the chapter in a few words.
- *Selected Solutions* discusses exercises in the chapter's main text if any.
- *Review Questions* presents a number of questions about the main learning contents of the chapter. You can use these to test your knowledge of the topics. Remember though that many of the learning objectives require you to program and experiment at the computer to ensure that you experience a deep learning process.
- *Further Exercises* presents additional, small, exercises to sharpen your skills. Note, however, that the main body of exercises is defined in the projects in part 9.

Some of the chapters, notably the short design pattern presentations in learning iteration 6, *A Design Pattern Catalogue*, will leave out some of these subsections.

How to Use the Book

This book can be used in a number of ways. The book is written for courses with a strong emphasis on practical software development with a substantial project work element leading all the way to students designing and implementing their own frameworks with several concrete instantiations. The book has been used in semester length, quarter length, and short courses. Below I will outline variations of this theme as well as alternative uses of the book.

Semester lengths project courses. Learning iterations 1–8 of the book are organized to follow a logical path that demonstrates how all the many different development techniques fit nicely together to allow students to build flexible frameworks and discuss them from both the theoretical as well as practical level. Each iteration roughly correlates to two weeks of the course. Topics from iteration 8 need not be introduced last but can more or less be introduced at any time. For instance, it may make sense to introduce a software configuration management tool early to support team collaboration on source code development. The projects in part 9 follow the rhythm of the book and students can start working on these as soon as the test-driven development chapter has been introduced. Alternative projects can be defined, however, consult Christensen (2009) for some pitfalls to avoid.

Quarter lengths project courses. Here the basic organization of the book is still followed but aspects must be left out or treated cursory. Chapters marked by a gray background in Figure 1 are candidates. Depending on the entry level of the students, parts of the *Basic Terminology* part can be cursory reading or introduced as part of a topic in the later iterations—for instance introducing the notion of test cases as part of demonstrating test-driven development, or just introduce maintainability without going into its sub qualities. The build management topic can be skipped and replaced by an introduction to integrated development environments or Ant scripts can be supplied by the teacher, as it is possible to do the projects without doing the build script exercises. The projects in the last part of the book work even in quarter length courses, note however that this may require the teacher to supply additional code to lower the implementation burden. This is especially true for the MiniDraw integration aspect of the framework iteration.

Short courses. Two-three day courses for professional software developers can be organized as full day seminars alternating between presentations of test-driven development, design patterns, variability management, compositional designs, and frameworks, and hands-on sessions working on the pay station case. Depending on the orientation of your course, topics are cursory or optional.

Design pattern courses. Here you may skip the test-driven development aspects all together. Of course this means skipping the specific chapter in part 2, the test stub chapter in part 4, as well as skipping the construction focused sections in the chapters in parts 3 and 4. I advise to spend time on the theory of roles and compositional design in part 5. Optionally part 7 may be skipped altogether and time spent on covering all the patterns present in the catalogue in part 6. The patterns may be supplemented by chapters from other pattern books.

Software engineering courses. Here less emphasis can be put on the pattern catalogue in part 6 and frameworks in part 7 and instead go into more details with tools and techniques for systematic testing, build-management, and configuration management.

Framework oriented courses. Here emphasis is put on the initial patterns from parts 3 and 4 and on the theory in part 5. Only a few of the patterns from part 6 are presented, primarily as examples of compositional design and for understanding the framework case, MiniDraw, in part 7.

Prerequisites

I expect you to be a programmer that has a working experience with Java, C#, or similar modern object-oriented programming languages. I expect that you understand basic object oriented concepts and can design small object-oriented systems and make them "work". I also expect you to be able to read and draw UML class and sequence diagrams.

Conventions

I have used a number of typographic conventions in this book to highlight various aspects. Generally *definitions* and *principles* are typeset in their own gray box for easy visual reference. I use side bars to present additional material such as war stories, installation notes, etc. The design patterns I present are all summarized in a single page side bar (a **pattern box**)—remember that a more thorough analysis of the pattern can be found in the text.

I use type faces to distinguish class names, role names, and other special meaning words from the main text.

- ClassName and methodName are used for programming language class and method names.

- PatternName is used for names of design patterns.
- *packagename* is used for packages and paths.
- `task` is used for Ant task names.
- **roleName** is used for the names of roles in designs and design patterns. Bold is also used when new terms are introduced in the text.

Web Resources

The book's Web site, http://www.baerbak.com, contains source code for all examples and projects in the book, installation guides for tools, as well as additional resources. Source code for all chapters, examples, exercises, and projects in the book are available in a single zip file for download. To locate the proper file within this zip file, most listings in chapters are headed by path and filename, like

Fragment: chapter/tdd/iteration-0/PayStation.java

```
public interface PayStation {
```

That is, PayStation.java is located in folder *chapter/tdd/iteration-0* in the zipfile. Several exercises are also marked by a folder location, like

Exercise 0.1. Source code directory:
`exercise/iterator/chess`

Permissions and Copyrights

The short formulation of the TDD principles in the book and on the inner cover are reproduced by permission of Pearson Education, Inc., from *Beck, TEST DRIVEN DEVELOPMENT:BY EXAMPLE, © 2003 Pearson Education, Inc and Kent Beck.* The historical account of design patterns in Chapter 9 was first written by Morten Lindholm and published in *Computer Music Journal 29:3* and is reprinted by permission of MIT Press. IEEE term definitions reprinted by permission of Dansk Standard. The *intent* section of the short design pattern overviews in the pattern side bars as well as the formulation of the *program to an interface* and *favor object composition over class inheritance* are reprinted by permission of Pearson Education, Inc., from *Gamma/Helm/Johnson/Vlissides, DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED DESIGN*. Other copyrighted material is reproduced as *fair use* by citing the authors.

Java technology and *Java* are registered trademarks of Sun Microsystems, Inc. *Windows* is a registered trademark of Microsoft Corporation in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. All other product names mentioned throughout the book are trademarks of their respective owners.

The lnkscape image on page 337 was made by Konstantin Rotkevich and is copyleft under the Free Art Licence. Michael Margold at SoftCollection kindly gave permission to use their Java source code for the LCD display code used in the pay station graphical user interface first introduced in the Facade chapter. Karl Hörnell gave permission to copy the IceBlox game from his web site `www.javaonthebrain.com`. The graphical tile set used for drawing the map in HotCiv is a copy of the neotrident tile set for FreeCiv 2.1.0, released under the GNU public license.

Acknowledgments

The following students have made valuable contributions by pointing out problems in the text or in the exercises: Anders Breindahl, Carsten Moberg Hammer, Emil Nauerby, Jens Peter S. Aggerholm, Jens Bennedsen, Hans Kolind Pedersen, Kristian Ellebæk Kjær, Karsten Noe, Kenneth Sejdenfaden Bøgh, Mads Schaarup Andersen, Mark Sjøner Rasmussen, Marianne Dammand Iversen, Mark Surrow, Martin Norre Christensen, Michael Dahl, Michael Lind Mortensen, Mikael Kragbæk Damborg Jensen, Mikkel Kjeldsen, Morten Wegelbye Nissen, Ole Rasmussen, Peter Urbak, Rasmus Osterlund Feldthaus Hansen, and Søren Kaa. Henrik Agerskov drew the initial graphics for the Backgammon graphical user interface.

A special thanks to Finn Rosenbech Jensen for some good discussions, much enthusiasm, and valuable comments. I would like to thank Morten Lindholm Nielsen that contributed to Chapter 9. Jens Bennedsen, Jürgen Börstler, Erik Ernst, Edward F. Gehringer, Klaus Marius Hansen, John Impagliazzo, Michael Kölling, Andrew McGettrick, and Cyndi Rader provided valuable reviews and comments throughout the process. A special thanks to Michael E. Caspersen for getting CRC Press interested in my book. I would also like to thank Alan Apt at CRC Press for being an enthusiastic editor, and to Amy Blalock and Michele Dimont for helping me through the maze of tasks associated with writing a book. My collegues at Department of Computer Science, Aarhus University, I thank for an inspiring work environment, and the opportunity to spend part of my time writing this book.

Finally, I dedicate this book to my wife, Susanne, and my children, Mikkel, Magnus, and Mathilde. *Home is not a place but the love of your family…*

Iteration 1

Basic Terminology

Developing reliable and flexible software is a major challenge that requires a lot of techniques, practices, tools, and analytical skills. In order to evaluate and understand techniques, however, you have to know what the terms *reliable* and *flexible* really means. In the *Basic Terminology* part of the book I will present the terms that are essential for the analyses and discussions in the rest of the book. I will provide small examples in this introduction, but larger and more complex examples will be presented later.

Chapter	Learning Objective
Chapter 1	*Agile Development Processes.* I will focus quite a lot on the *process of programming* in the beginning of the book—that is, the process you go through as you move from the initial requirements and design ideas for a software system towards a high quality program. As time has shown, requirements to software systems change frequently, and I need processes that can cope with that. The objective of this chapter is to present the ideas and values that are fundamental for agile development processes like Extreme Programming, Scrum, and Test-Driven Development.
Chapter 2	*Reliability and Testing.* A major learning objective of this book is to provide you with skills and techniques for writing high quality programs—but what is quality after all? One main aspect is *reliability*: that I can trust my programs not to fail. In this chapter, the objective is learning the terms and definitions that allow us to discuss reliable programs. A central technique to increase reliability is *testing* which has its own set of terms that are also introduced in this chapter.
Chapter 3	*Flexibility and Maintainability.* Another important quality of software today is its ability to adapt to changing requirements at low cost. The objective of this chapter is to introduce the terms and definitions concerning flexibility and maintainability that allow us to discuss these aspects precisely.

Chapter 1

Agile Development Processes

Learning Objectives

In this chapter, the learning objective is an understanding of the main ideas of agile development methods. One particular and influential agile method, namely Extreme Programming, is treated is greater detail. Several of the techniques and practices of Extreme Programming are discussed in great detail later in the book, and this chapter thus primarily serves to create the context in which to understand them.

1.1 Software Development Methods

No matter how you develop software, you apply a certain *software development process*, that is, a structure imposed on the tasks and activities you carry out to build software that meets the expectations of your customer. A software development process must define techniques to deal with activities or steps in development. Such steps will usually include:

- *Requirements.* How do you collect and document the users' and customers' expectations to the software, i.e. what is it supposed to do?
- *Design.* How do you partition and structure the software and how do you communicate this structure?
- *Implementation.* How do developers program the software so that it fulfills the requirements and adheres to the design?
- *Testing.* How do you verify that the executing system indeed conforms to the requirements and users' expectations?
- *Deployment.* How do you ensure that the produced software system executes in the right environment at the user's location?

- *Maintenance.* How do you ensure that the software is corrected and enhanced as defects are discovered by users or new requests for functionality are made?

The amount of rigor in defining processes and tools for these steps vary according to the size of a project: Building aircraft control software in a project with hundreds of developers requires stricter control than a spare time game developed by two friends.

Exercise 1.1: Consider your last project or programming exercise. How was the activities/steps defined, executed, and controlled?

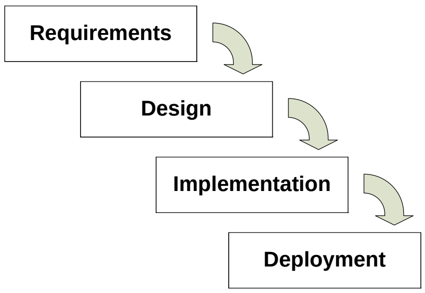

Figure 1.1: Waterfall model.

Over the years researchers and practitioners of software engineering have described and tested a large number of software processes. For many years there was a tendency for them to be heavy-weight, that is, they put much emphasis on strict communication and process rules, on producing large amounts of detailed documentation, and on not beginning one activity before the previous activity had been analyzed and understood in detail. One such model was the **waterfall model**, sketched in Figure 1.1, in which one proceeds from one activity to another in a purely sequential manner: you get all the requirements documented in full detail before you start designing; you do not start implementing before the design is complete, etc. While this process may seem appealing, as it is much cheaper to correct a mistake early in the process, it was quickly realized that perfecting one stage before starting the next was impossible even for small projects. More often than not, requirements and designs are not well understood until a partially working software system has been tested by users. Thus in the waterfall model, such insights discovered late in the phase invalidates a large investment in the early phases. The agile development processes are characterized as lightweight and can be seen as ways to ensure that these insights invalidates as little invested effort as possible.

☞ Find literature or internet resources on some common development models such as *waterfall, cleanroom, spiral model, V-model, XP, Scrum, RUP, etc.*

1.2 Agile Methods

The core of agile methods is expressed in the *agile manifesto*, reproduced here from the agile manifesto web page `agilemanifesto.org`:

Manifesto for Agile Software Development
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

☞ Find the manifesto on the internet and note that it is "signed" by quite a few people. Several of these people have invented or significantly contributed to the techniques described in this book.

Several points are notable in this manifesto. First of all, it is written by practitioning software developers: "by doing it and helping others do it." Next, the agile methods are just as much about *values* as it is about concrete techniques. At present there are quite a few agile methods around: Extreme Programming (XP), Scrum, Crystal Clear, and others; but they share the same core values as expressed in the manifesto. This makes it relatively easy to understand e.g. Scrum once you have understood XP. I will generally use the terminology of XP as the programming process used in this book, *Test-Driven Development*, was invented as part of XP. A final, but central, point is the word "agile." Agile methods value to move towards the defined goal with speed while maintaining the ability to change to a better route without great costs.

Individuals and interactions are emphasized. Agile methods put a lot of emphasis on software development as a team effort where individual's creativity and contribution is central to overall success. Thus forming a good context for individuals and their collaboration is central. Earlier development methods had a tendency to view individuals as "production units" that mechanically produce software code, designs, test plans, etc., and thus little attention was paid to making individuals feel comfortable and more attention paid to documents and processes to control collaboration. Agile methods have suggested a number of practices to ensure that the right decisions are made because people are responsible, want to do the right thing, and have the proper information at hand to make qualified decisions. For instance, attention is paid to how teams are located in buildings: if two teams that are supposed to collaborate are on different floors then they will communicate much less than if located in offices next to each other. And with less communication the risk of misunderstandings and thus defects in the product rise significantly.

Working software. Bertrand Meyer, the inventor of the Eiffel programming language, once said that *"once everything is said, software is defined by code..."*(Meyer 1988). The design may be just right, the UML diagrams beautifully drawn, but if there is no code there is no product and thus no revenue to pay the bills. Agile methods focus on making code of high quality and less on writing documents about the code. The reasoning is that it takes time to write documentation which is then not used to make code. As *individuals and interactions* are emphasized it is much faster and accurate to ask the relevant people than trying to find the information in the documentation. However, it is *working* software that is valued: the code should be of high quality. To this end, *testing* is central: exercising the software to find defects in it. Another central technique to keep the code of high quality is *refactoring*: improving code structure without affecting its functionality. Both techniques are central for this book and discussed in detail later.

Customer collaboration is faster than negotiating contracts. Bjarne Stoustrup, the inventor of the C++ programming language, has said that *"to program is to understand..."* As you implement customer requirements, you get a much deeper understanding of them and spot both ambiguities to be resolved as well as opportunities for improving the final product. If these matter have to go through a long chain of command, *developers ask managers that ask sales people that ask buyers that ask everyday users...*, the feedback loop is too slow. The result is defective or cumbersome software and missed opportunities for a better product. In the vein of focusing on interaction between people, agile methods require customers and users to be readily available for questioning and discussions. A central technique is **small releases** where working but functionally incomplete systems are presented to customers for them to use. These releases are a better starting point for discussing ambiguities and improvement opportunities than long requirement specifications.

Responding to change means that the best route to a goal may be another than that planned at the beginning of the journey. Working in the project means learning and improving the understanding of what best fits the customer—especially as they are integrated in the project as outlined above. Thus an initial plan is important but the plan should be revised as experience accumulate. Maybe the customer thought they wanted feature X but if working with a small release part way into the project shows feature X to be less important but feature Y much more relevant, then why not work on feature Y for the next small release?

1.3 Extreme Programming

One of the first agile methods was **Extreme Programming**, or short **XP**, that received a lot of attention in the beginning of the millennium. XP pioneered many central techniques that are presented in this book and it serves well as an example of an agile method.

1.3.1 Quality and Scope

In the book *Extreme Programming Explained*, Kent Beck (2000) presents a model for software development to explain some of the decisions made in XP. In this model, a software product is controlled by four parameters: *cost*, *time*, *scope*, and *quality*. Cost is basically the price of the product which again correlates strongly to the number of people assigned to work on the project. Time is the amount of time to the delivery deadline. Scope is the size of the project in terms of required functionality. Quality is aspects like usability, fitness for purpose, and reliability. These four parameters have a complex relationship, and changing one in a project affects the others. The relationship is, however, not simple. For instance, you may deliver faster (decrease time) by putting additional programmers on a project (increase cost) but doubling the number of programmers will certainly not cut the delivery time in half.

Exercise 1.2: Explain why doubling the number of programmers will likely not cut the time spent on the project in half.

The point made is that often organizations decide the first three parameters (cost, time, scope) and leave the forth parameter (quality) as the only free parameter for developers to control. Typically, the onset of a project is: *We have three months (fix time) to complete the requirements in this specification (fix scope) and this group of eight developers are assigned (fix cost).* As estimates are often optimistic, the developers have no other option than sacrifice quality to meet the other criteria: delivery on time, all features implemented, team size constant. Even still, our software development profession is full of examples of projects, that did not deliver in time, did not deliver all features, and went over budget.

Exercise 1.3: Consider a standard design or programming project in a university or computer science school course. Which parameters are typically fixed by the teacher in the project specification?

XP focuses on making *scope* the parameter for teams to control, not quality. That is, the three parameters cost, time, and quality are fixed, but the scope is left open to vary by the team. For example, consider that three features are wanted in the next small release, but after work has started it is realized that there is not enough time to implement them all in high quality. In XP, the customers are then involved to select the one or two most important features to make it into the release. *Two working features are valued higher than three defective or incomplete ones.*

Needless to say, this swapping of the roles of *scope* and *quality* in XP is underlying many of its practices:

- to control and measure quality, *automated testing* is introduced, and made into a paradigm for programming called *test-driven development*. Test-driven development is the learning focus of Chapter 5. To ensure code quality, *refactoring* is integrated in the development process. Refactoring is a learning focus of Chapter 8.
- to control scope, an *on-site customer* is required so the interaction and decisions can be made quickly and without distortion of information. *Small releases* are produced frequently to ensure time is always invested in those features and aspects that serves the user's need best.

1.3.2 Values and Practices

XP rests upon four central values

- *Communication.* A primary cure for mistakes is to make people communicate with each other. XP value interaction between people: between developers, with customers, management, etc.
- *Simplicity.* "What is the simplest thing that could possibly work?" In XP you focus on the features to put into the next small release, not on what may be needed in six months.

- *Feedback.* You need feedback to know you are on the right path towards the goal, and you need it in a timely manner. If the feature you are developing today will not suit the need of the customer you need to know it today, not in six months, to stay productive. XP focus on feedback in the minutes and hours time scale from automated tests and from the on-site customer, and on the week and month scale from small releases.
- *Courage.* It takes courage to throw away code or make major changes to a design. However, keep developing based on a bad design or keep fixing defects in low quality code is a waste of resources in the long run.

Based on these values, a lot of practices have evolved. Below, I will describe a few that are central to the practices of programming and the context of this book. It should be noted that XP contains many more practices but these are focused on other aspects of development such as planning, management, and people issues, which are not core topics of this book.

A central technique in XP is **pair programming**. Code is never produced by an individual but by a pair of persons sitting together at a single computer. Each person has a specific role. The person having the keyboard focuses on the best possible implementation at the detailed level. The other person is thinking more strategically and evaluates the design, reviews the produced code, looks for opportunities for simplifications, defines new tests, and keeps track of progress. The pairs are dynamic in that people pair with different people over the day and take turns having the two different roles. To make this work, **collective code ownership** is important: any programmer may change any code if it adds value for the team. Note that this is different from *no ownership* where people may change any code to fit their own purpose irrespective if this is a benefit or not for the team. Collective ownership also force programmers to adhere to the same **coding standards** i.e. the same style of indentation, same rules for naming classes and variables, etc. Pair programming is both a quality and communication technique. It focuses on quality because no code is ever written without being read and reviewed by at least two people; and there are two people who understand the code. And it focuses on communication: pairs teach and learn about the project's domain and about programming tricks. As pairs are dynamic, learning spreads out in the whole team.

Automated testing is vital in XP, and treated in detail in Chapter 2 and 5. Automated testing is testing carried out by the computers, not by humans. Computers do not make mistakes nor get tired of executing the same suite of 500 tests every ten minutes, and they execute them fast. By executing automatic tests on the software system often, the system itself gives feedback about its health and quality, and developers and customers can measure progress. These tests must all pass at all times. To get feedback on the "fitness of purpose" you make **small releases** frequently: once every month, every two weeks, or even daily. A small release is a working, but functionally incomplete, system. At the beginning of a project it may serve primarily as a vehicle to discuss fitness of purpose with the end users and customers; however as quickly as possible it will be deployed into production and used with ever growing functionality. **Continuous integration** means that the development effort of each pair programming team are continuously added to the project to ensure that the developed code is not in conflict with that of the other pair teams. Integration testing is on of the learning foci of Chapter 8.

Customers' and users' requirements are captured as **stories** which are short stories about user-visible functionality. These stories are written on index cards and given short headlines, like "Allow stop of video recording after a specified time interval", "Align numbers properly in columns", "Add currency converter tool to the spreadsheet", "Compute account balance by adding all deposits and subtracting all deductions," etc. Finally, they are put on a wall or some other fully visible place. Also each story is estimated for cost in terms of hours to implement. This allows developers and customers to choose those stories that provide the best value for the least cost. Stories should be formulated so they are achievable within a 4–16 hour effort of a pair. Once a pair has selected a story, it is further broken down as will be describe in Chapter 5 on test-driven development.

Exercise 1.4: Compare the short outline of XP above with the list of development activities in the start of the chapter. Classify XP practices in that framework: requirements, design, implementation, etc.

XP is a highly **iterative** development process. Work is organized in small *iterations*, each with a well defined focus. At the smallest level, an iteration may last a few minutes, to implement a simple method in a class. At the next level, it may last from a few hours to a day, to implement a feature or a prerequisite for a feature, and ends in integrating the feature into the product. And at the next level again, the small release defines an iteration, lasting from weeks to a month, to implement a coherent set of features for users to use. It follows that it is also an **incremental** development process: systems are grown piecemeal rather than designed in all detail at the start.

The strong focus in XP on people interactions means it is suited for small and medium sized teams. Large projects with many people involved need more rigor to ensure proper information is available for the right people at the right time. This said, any large project uses a "divide and conquer" strategy and thus there are many small teams working on subsystems or parts of a system and these teams can use XP or another agile method within the team.

1.4 Summary of Key Concepts

A software development process is the organization and structuring of indiviual activities (or steps) in development. Any project must consider and organize activities like requirements, design, implementation, testing, deployment, and maintenance.

Agile methods adhere to the agile manifesto that emphasizes four values: *interacting individuals*, *working code*, *collaboration with customers*, and *responding to change.* Extreme Programming is an early and influential agile process. It is light-weight, iterative and incremental. It manifests the agile values in a set of practices many of which have been adopted or adapted in later methods. Key practices are automatic testing, small releases, and continuous integration to ensure timely feedback to developers of both the product's reliability as well as its fitness to purpose. Pair programming is adopted to ensure learning and interaction in the team. Pairs are dynamic and form to implement stories: stories define a user-visible unit of functionality. Stories are written on index cards and usually put on a wall visible to all developers. Each story is estimated for work effort.

Fowler (2005)'s online article *The New Methodology* is a short and easy introduction to agile methods. For further reading on XP, you should consult *Extreme Programming Explained–Embrace Change*, available in a first and second edition, by Beck (2000) and Beck (2005). Another good overview is given in *Extreme Programming Installed* by Jeffries et al. (2001).

1.5 Selected Solutions

Discussion of Exercise 1.2:

Doubling the number of programmers on a project means there is a lot of learning of the domain that has to be made by the new staff before they become productive. The only source of information is usually the existing programmers thus productivity will actually fall for an extended period of time until the new people are "up to speed." More people also means more coordination and more management which drain resources for the actual implementation effort.

This said, adding more people may in the long run be a wise investment. XP advises to start projects in small teams and then add people as the core system becomes large enough to define suitable subprojects for sub teams to work on.

Discussion of Exercise 1.3:

Well, typical university course design or programming exercises are no different from the culture in industry. Projects specify *time:* "deliver by the end of next month", *cost:* "to be developed in groups of three students", and *scope:* "design and implement feature X, Y, and Z." Thus students typically are left only with the quality parameter to adjust workload.

1.6 Review Questions

Outline the activities or steps that any software development method must include. What is the goal of each activity? Discuss the difference between a waterfall method and an agile method.

What are the four aspects that are valued in the agile manifesto? Explain the argumentation for each of the four aspects.

Describe the model for software development proposed by Kent Beck: what are the four parameters and which are fixed and which are free in many traditional development projects. What parameter does XP propose to fix instead?

Explain the four values in XP. Describe the aspects in key practices such as pair programming, collective code ownership, stories, automated testing, small releases, and continuous integration.

Chapter 2

Reliability and Testing

Learning Objectives

Learning to make software *reliable* is important to become a competent and successful developer. Much of this book is devoted to develop the mindset, skills, and practices that contribute to build software that does not fail. The learning objective of this chapter is to develop the foundation for these practices and skills by presenting the basic definitions and terminology concerning reliable software in general, and testing as a technique to achieve it, in particular.

Specifically, this chapter

- Introduces you to the concept of *reliability*.
- Introduces you to definitions in testing terminology: what is testing, a test case, a failure, etc., that are used throughout the book.
- Introduces a concrete Java tool, JUnit, that is a great help in managing and executing automated tests.

2.1 Reliable Software

In the early days of computing, programs were often used by the same persons that wrote them. For instance, a physicist may write a program to help with numerical analysis of the data coming from an experiment. If the program crashed or misbehaved then the damage was rather limited as the program just had to be updated to fix the error. Modern mass adoption of computing has over the last couple of decades changed the requirement for reliable programs dramatically: today the ordinary computer user is not a computer programmer and will not accept software that does not work. Today users expect software to behave properly and *reliability* is a quality to strive for in producing software. Reliability can be defined in several ways but I will generally use definitions from the ISO 9126 (ISO/IEC International Standard 2001) standard.

Definition: **Reliability (ISO 9126)**

The capability of the software product to maintain a specified level of performance when used under specified conditions.

The central aspect of this definition is … *to maintain a specified level of performance*. In our setting "performance" is the ability to perform the required functions without failing: letting the users do their work using the software product. Informally developers state that the "system works."

Exercise 2.1: The aspects of the definition that deals with … *under specified conditions* also have implications. Give some examples of the same software system being considered reliable by one user while being considered unreliable by another because the users have different "specified conditions."

Reliability is one of many qualities a software system must have to be useful. Another quality may be that it must execute fast and efficiently so responses to the users do not take too long, that it must be useable so the users can understand and use the software efficiently, etc. The next chapter discusses "maintainability" as an important quality. However, reliability is a central quality as many other qualities becomes irrelevant if the software is unreliable e.g. it is of little use that a system responds quickly if the answer is wrong.

Thus reliability is a highly desired quality of software and both the research and industrial communities have produced numerous techniques focused on achieving reliability. Some examples are:

- Programming language constructs. Modern programming languages contain a lot of language constructs and techniques that prevent tricky defects that were common in the early days of machine code and early programming languages. As an example, the original BASIC language did not have local variables, and therefore you could ruin a program's behavior if you accidentally used the same variable name in two otherwise unrelated parts of the program.
- Review. Reviews are more or less formalized sessions where reviewers read source code with the intention of finding defects. Designs and documentation can also be reviewed. Reviews have the advantage that you can find defects in the code that are not visible when executing it. Examples are defects like poor naming of variables, poor formatting of code, misleading or missing comments, etc. The technique's liability is that it is manual and time consuming.
- Testing. Testing is executing a software system in order to find situations where it does not perform its required function. Testing has the advantage that is can to a large extent be automated, but its liability is that it can only detect defects that are related to run-time behavior.

I assume Java or a similar modern object-oriented programming language and therefore get the many reliability benefits these languages have over older languages like C, Fortran, and BASIC. Review is an important technique that can catch many types of defects, but I will not discuss it any further in this book. Testing is a well-known technique but has been revitalized by the agile software development movement, in particular by the test-driven development process. As I will emphasize testing and test-driven development, the rest of this chapter is devoted to the basic terminology and tools that form the foundation for understanding and using these techniques.

2.2 Testing Terminology

I will introduce basic concepts and terminology of testing through a small example. Consider that I am part of a team that has to develop a calendar system. One of my colleagues has developed a class, Date, to represent dates[1], whose constructor header is reproduced below:

Fragment: chapter/reliability/handcoded-test/Date.java

```
public class Date {
  /**
   * Construct a date object.
   * @param year the year as integer, i.e. year 2010 is 2010.
   * @param month the month as integer, i.e.
   *             januar is 1, december is 12.
   * @param dayOfMonth the day number in the month, range 1..31.
   * PRECONDITION: The date parameters must represent a valid date.
  */
  public Date(int year, int month, int dayOfMonth) {
```

Now I have been asked to extend it with a method, dayOfWeek, to calculate the week day it represents. The method header should be:

Fragment: chapter/reliability/handcoded-test/Date.java

```
  public enum Weekday {
     MONDAY, TUESDAY, WEDNESDAY,
     THURSDAY, FRIDAY,
     SATURDAY, SUNDAY };
  /**
  * Calculate the weekday that this Date object represents.
  * @return the weekday of this date.
  */
  public Weekday dayOfWeek() {
```

How do I ensure that my complex algorithm in this method is reliably implemented? I choose to do so by testing:

Definition: Testing

Testing is the process of executing software in order to find failures.

[1]The java libraries already have such a class. The best path to reliable software is of course to reuse software units that have already been thoroughly tested instead of developing new software from scratch. For the sake of the example, however, I have chosen to forget this fact.

This definition requires we know what a failure is:

> Definition: **Failure**
>
> A failure is a situation in which the behavior of the executing software deviates from what is expected.

Thus, if I call this method with a date object representing 19th May 2008, it should return `MONDAY`, as this date was indeed a monday.

```
    Date d = new Date(2008, 5, 19); // 19th May 2008
    // weekday should be MONDAY
    Date.Weekday weekday = d.dayOfWeek();
```

If, however, it returns, say, `SUNDAY` then this is clearly a failure. A failure is the result of a defect:

> Definition: **Defect**
>
> A defect is the algorithmic cause of a failure: some code logic that is incorrectly implemented.

Informally, defects are often called faults or "bugs".

☞ Find out why it is called a bug on wikipedia or other internet sources.

In other words, to test I have to define three "parameters": the method I want to test, the input parameters to the method, and the expected output of the method. This is called a *test case*:

> Definition: **Test case**
>
> A test case is a definition of input values and expected output values for a unit under test.

A single test case cannot test all aspects of a method so test cases are grouped in test suites.

> Definition: **Test suite**
>
> A test suite is a set of test cases.

A short and precise way of representing suites of test cases is in a **test case table**, a table that lists the sets of test cases defining the input values and expected output. In my case, the dayOfWeek method test cases can be documented as in Table 2.1.

A test case may **pass** or **fail**. A test case passes when the computed output of the unit under test is equal to the expected output for the given input values in the test case. If not, the test case fails. The terms are also used for test suites: a test suite passes if all its test cases pass but fails if just one of its test cases does. A failed test is often also referred to as a *broken test.*

I have written the test cases above for testing a single method, but in general, testing can be applied at any granularity of software: a complete software system, a subsystem, a class, a single method...Therefore test cases are defined in terms of the *unit under test*:

Table 2.1: Test case table for dayOfWeek method.

Unit under test: dayOfWeek	
Input	Expected output
year=2008, month=May, dayOfMonth=19	Monday
year=2008, month=Dec, dayOfMonth=25	Thursday
year=2010, month=Dec, dayOfMonth=25	Saturday

Definition: **Unit under test**

A unit under test is some part of a system that we consider to be a whole.

OK, I have made an implementation, I have written my test case table, but the real interesting aspect is to conduct the test. In my case, the Date class is part of a calender program, and I can do the testing by following a short manuscript like: *"Start the application, and scroll to May 2008. Verify that May 19th is marked as Monday"*. To execute this test and verify that the software does as expected, I have to spend time executing this manuscript manually. This is called manual testing.

Definition: **Manual testing**

Manual testing is a process in which suites of test cases are executed and verified manually by humans.

Usually developers do informal manual testing all the time when developing: add a few lines of code, compile, and run it to see if it "works".

Consider that I have tested the dayOfWeek method and as all tests pass I am confident that the implementation is correct and reliable. Some time later, however, customers complain that the calendar program is too slow when scrolling from one week to another. Our analysis shows that it is caused by the slow algorithm used in the dayOfWeek method—it has to be redesigned to compute weekdays much faster. The question is how I validate that the improved, faster, algorithm is still functionally correct? The answer is of course to repeat all the tests that I made the last time. This process has also its own term:

Definition: **Regression testing**

Regression testing is the repeated execution of test suites to ensure they still pass and the system does not fail after a modification.

One of the reasons that test case tables or other formal documents that outline test cases are important is to serve in regression testing and of course also in the final quality assessment before releasing a software product.

There is a large body of knowledge concerning how to pick a good set of test cases for a given problem: few test cases that have a high probability of finding the defects. Chapter 34 provides an introduction to some of the basic techniques. Until then we will rely on intuition, common sense, and the experience we gain from actual testing.

2.3 Automated Testing

Instead of manual testing, I can utilize my computer to handle the tedious tasks of executing test cases and comparing computed and expected values. I can write a small test program (only the last test case from test case Table 2.1 is shown):

Listing: chapter/reliability/handcoded-test/TestDayOfWeek.java

```
/** A testing tool written from scratch.
*/
public class TestDayOfWeek {
  public static void main(String[] args) {
    // Test that December 25th 2010 is Saturday
    Date d = new Date(2010, 12, 25); // year, month, day of month
    Date.Weekday weekday = d.dayOfWeek();
    if ( weekday == Date.Weekday.SATURDAY ) {
      System.out.println("Test case: Dec 25th 2010: Pass");
    } else {
      System.out.println("Test case: Dec 25th 2010: FAIL");
    }
    // ... fill in more tests
  }
}
```

This simple program shows the anatomy of any program to execute test cases: the unit under test (here dayOfWeek) is invoked with the defined input values, the computed and expected output values are compared and some form of pass/fail reporting takes place. This is *automated testing*:

Definition: **Automated testing**

Automated testing is a process in which test suites are executed and verified automatically by computer programs.

☞ Download the source code from http://www.baerbak.com and try to run the test. Why does it pass? Please observe that the production code is strictly made to demonstrate testing!

Automated testing requires me to write code to verify the behavior of other pieces of code and this effort results in a lot of source code being produced: code that will be defining the users' product and code that will define test cases that test it. Often we need to distinguish these two bodies of code and I will call them:

Definition: **Production code**

The production code is the code that defines the behavior implementing the software's requirements.

In other words, production code is what the customer pays for. The code defining test cases, I will call:

Definition: Test code

The test code is the source code that defines test cases for the production code.

Both manual and automated tests have their benefits and liabilities. Manual tests are often quicker to develop (especially the informal ones) than automated tests because the latter require writing code. However, once automated tests have been made it is much easier to perform regression testing compared to manual tests. Second, manual tests are also extremely boring to execute leading to human errors in executing the manual test procedures—or simply not executing them at all. The sidebar 2.1 describes a war story of how it may go. In contrast automated tests can be executed fast and without the problems of incorrect execution.

Exercise 2.2: In some sense automated testing is absurd because I write code to test code—should I then also write code to test the code that tests the code and so on? Or put differently: if a test case fails how do I know that the defect is in the production code and not in the test code? Think about properties that the test code must have for automated testing to make sense.

2.4 JUnit: An Automated Test Tool

Writing the testing program by hand, as I did above, works fine for a very simple example, but consider a system with several hundred classes each with perhaps hundreds or thousands of test cases. In this case you need much better support for executing the large test suite, for reporting, and for pinpointing the test cases that fail. **Unit testing tools** are programs that takes care of many of the house holding tasks and let you concentrate on expressing the test cases. In the Java world there are several tools but I will use JUnit 4 in this book. The fundamental concepts and techniques are the same in all the tools so changing from one to another is mostly a matter of syntax.

The test case for the dayOfWeek method can be written using JUnit like this:

Listing: chapter/reliability/junit-test/iteration-0/TestDayOfWeek.java

```
import org.junit.*;
import static org.junit.Assert.*;

/** Testing dayOfWeek using the JUnit 4.x framework.
*/
public class TestDayOfWeek {

  /**
   * Test that December 25th 2010 is Saturday
   */
  @Test
  public void shouldGiveSaturdayFor25Dec2010() {
    Date date = new Date( 2010, 12, 25);
    assertEquals( "Dec 25th 2010 is Saturday",
                  Date.Weekday.SATURDAY, date.dayOfWeek() );
  }
}
```

Sidebar 2.1: SAWOS RVR

I worked a few years in a small company that designed SAWOS systems: Semi-Automatic Weather Observation Systems. These are used in airports by meteorologists to generate reports of the local weather systems to be used by pilots and flight leaders.

One of the important factors when landing an aircraft is RVR: Runway Visual Range. It is the maximal distance at which the bright lamps in the runway can be seen. For instance, RVR = 25 m means you cannot see the lamps until you are only 25 meters away indicating a very heavy fog.

Usually each runway has instruments that measure RVR at both ends of it. Our SAWOS had a display showing the runway with each end colored according to its status ranging from green (no fog) to dark red (heavy fog).

We did not have automated tests and were a bit lazy about the manual ones, so at one time Aarhus airport reported a defect in their SAWOS: They saw a small patch of heavy fog drifting along the runway and when it passed the eastern end of the runway, the display marked the western end with a red color. I had implemented that part and had simply mixed up the assignment of instrument reading to the display and had missed this defect during my own manual testing.

I was on holiday when this defect was reported, but my colleagues fixed it "quick and dirty" as they did not quite understand my design, and sent a new version to Aarhus airport which cleared the problem.

When I returned home I was annoyed by their bug fix as it did not respect my design, so I remade it the "right" way. This change was then sent to the airport some weeks later as part of another software upgrade. Some months later, a technician phoned me and asked why RVR readings once again were showing at the wrong end of the display? I had once again forgotten to run the manual tests...

The first thing to note is the import statements. JUnit is a set of Java classes that you need to import in order to use it. The two imports in the source code listing is all I need for now.

Test cases are expressed as methods in JUnit which makes sense as methods contain code to be executed, just as test cases must be executed. To tell JUnit that a particular method defines a test case you have to mark it using the **@Test** annotation. JUnit will collect all methods with the @Test annotation and execute them, one by one. Thus a set of @Test methods in a class becomes a test suite.

A test case may pass or fail depending on equality between computed and expected value. JUnit provides a set of comparison methods that you must use, all named beginning with assert. In my example, I need to compare the computed weekday from dayOfWeek with the expected value Weekday.SATURDAY. The statement

```
Date date = new Date( 2010, 12, 25);
assertEquals( "Dec 25th 2010 is Saturday",
              Date.Weekday.SATURDAY, date.dayOfWeek() );
```

tells JUnit to compare the expected value (second parameter) with the computed value (third parameter). The first parameter is a string value that JUnit will display in case the test case fails in order for you to better understand the problem. For instance, if the above test fails, JUnit will print something similar to:

Sidebar 2.2: Asserts in JUnit

JUnit contains several assert methods that come in handy. Below I have shown them in their terse form; remember that you can always provide an extra, first, argument of type String that JUnit will print in case a test case fails.

assert	**Pass if:**
assertTrue(boolean b)	expression b is true
assertFalse(boolean b)	expression b is false
assertNull(Object o)	object o is null
assertNotNull(Object o)	object o is not null
assertEquals(double e, double c, double delta)	e and c are equal to within a positive delta
assertEquals(Object[] e, Object[]c)	object arrays are equal

If a method should throw an exception for a given set of input values, you provide the exception as argument to the @Test annotation. For instance, if method doDivide should throw ArithmeticException in case the second argument is 0:

```
 @Test(expected = ArithmeticException.class)
  public void divideByZero() {
    int value = calculator.doDivide(4,0);
  }
}
```

```
java.lang.AssertionError: Dec 25th 2010 is Saturday
     expected:<SATURDAY> but was:<SUNDAY>
```

In all JUnit's assert methods, you may omit the string parameter and only state the comparison, like

```
  Date date = new Date( 2010, 12, 25);
  assertEquals( Date.Weekday.SATURDAY, date.dayOfWeek() );
```

in which case JUnit prints a less descriptive failure report:

```
java.lang.AssertionError: expected:<SATURDAY> but was:<SUNDAY>
```

The assertEquals method takes two parameters, the expected and the computed value. If the parameters are objects, they are compared using their equals method; if they are primitive types, they are compared using ==. There are other assert methods you can use in your test cases, sidebar 2.2 describes some of them. For a complete list, consult the JUnit documentation on the web.

JUnit does not care about the name of the test case method but test cases are meant to be read by developers and testers so I consider naming important. I like to name my test cases starting with should... as test cases should verify that the unit under test does its job properly. So—dayOfWeek *should* give Saturday for December 25th in 2010.

Now—I have a JUnit test case and the next issue is to execute it to see if it passes or fails. JUnit 4 is a bare-bones tool that is primarily meant to be integrated into development environments like BlueJ (2009) or Eclipse (2009) but it does provide textual

Sidebar 2.3: JUnit Setup

You can download the JUnit framework from the JUnit Web site: `www.junit.org`. I have used JUnit version 4.4 in this book. To compile the JUnit test cases the compiler must know where the JUnit class files are located. You do this by including the provided jar, named `junit-4.4.jar`, on the classpath. This will look like this on Windows:

```
javac -classpath .;junit-4.4.jar *.java
```

Similar, you will need to tell the Java virtual machine to use this jar file when executing:

```
java -classpath .;junit-4.4.jar
  org.junit.runner.JUnitCore TestDayOfWeek
```

You will have to type the above on a single line in the command prompt/shell. If you run on a Linux operating system the ";" must be replaced by ":".

The source code for this chapter as well as compilation scripts for Windows and Linux are provided at the book's web site (http://www.baerbak.com). You need to download the zip archive, unzip it and then you will be able to find the source code in the folders listed above each source code listing. The JUnit jar is provided along with the example code so you do not need to download JUnit unless you want to browse the documentation and guides. The compile script is named "compile.bat" (Windows) or "compile.sh" (Linux bash). The run script is named "run-test.bat" or "run-test.sh".

output. You can find information on running JUnit from a command prompt/shell in sidebar 2.3.

The output of executing the test suite simply looks like this:

```
>java -classpath .;junit-4.4.jar
   org.junit.runner.JUnitCore TestDayOfWeek
JUnit version 4.4
.
Time: 0,031

OK (1 test)
```

JUnit is terse when all test cases pass: it prints a dot for every test case executed and finally outputs OK and some statistics.

OK—I will add another test case from my original test case table. I add a second test method for 25th December 2008 to the TestDayOfWeek class:

Fragment: chapter/reliability/junit-test/iteration-1/TestDayOfWeek

```
  @Test
  public void shouldGiveThursdayFor25Dec2008() {
    Date date = new Date( 2008, 12, 25);
    assertEquals( "Dec 25th 2008 is Thursday",
                  Date.Weekday.THURSDAY, date.dayOfWeek() );
  }
```