Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Desenvolvimento de Software Guiado por Testes de Aceitação Usando EasyAccept

Osório Lopes Abath Neto

Campina Grande, Paraíba, Brasil
Agosto de 2007

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Desenvolvimento de Software Guiado por Testes de Aceitação Usando EasyAccept

Osório Lopes Abath Neto

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Jacques Philippe Sauvé
(orientador)

Campina Grande, Paraíba, Brasil
Agosto de 2007

A119d
2007 Abath Neto, Osório Lopes.
Desenvolvimento de software guiado por testes de aceitação usando EasyAccept/Osório Lopes Abath Neto. ─ Campina Grande: 2007.
116f.: il

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientador: Dr. Jacques Philippe Sauvé.

1. Engenharia de Software. 2. Testes de Aceitação. 3. Acceptance Test-Driven Development. I. Título.

CDU 004.41(043)

"DESENVOLVIMENTO DE SOFTWARE GUIADO POR TESTES DE ACEITAÇÃO USANDO *EASYACCEPT*"

OSÓRIO LOPES ABATH NETO

DISSERTAÇÃO APROVADA EM 30.08.2007

PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Orientador

PROFª PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D
Examinador

PROF. JAELSON FREIRE BRELAZ DE CASTRO, Ph.D
Examinador

CAMPINA GRANDE – PB

Resumo

O Desenvolvimento de Software Guiado por Testes de Aceitação – Acceptance Test Driven Development (ATDD) – é uma metodologia de desenvolvimento de software ágil que apresenta vários benefícios, que incluem confiança no software em desenvolvimento, sincronização automática entre análise e código, redução de problemas de comunicação no projeto e foco dos desenvolvedores nos requisitos do cliente. É particularmente adequada para projetos terceirizados e para ensinar desenvolvimento de software a estudantes de Ciência da Computação. Entretanto, como é uma metodologia nova, ainda falta para ela uma cobertura adequada na literatura. Além disso, a área de padrões para ATDD ainda precisa ser iniciada. Esta dissertação envolve a realização de um estudo investigativo sobre melhores práticas e padrões para ATDD, a definição de como aplicar a metodologia sob o ponto de vista de um processo de desenvolvimento de software, e um resumo da experiência adquirida com ensino de desenvolvimento de software utilizando ATDD. Como resultado da realização destas atividades, foi escrito um texto introdutório sobre ATDD, que esperamos sirva não só para que novatos aproveitem o máximo da metodologia, mas também para divulgar seus benefícios. Os exemplos do texto usam EasyAccept, uma ferramenta de testes de aceitação com scripts, como meio de exposição da metodologia.

Abstract

Acceptance Test Driven Development (ATDD) is an emerging agile methodology to develop software which has a number of advantages, including confidence in the software being developed, automated synchronization between analysis and code, reduction of project communication problems and developer focus on client requirements. It is particularly suited to outsource projects and to teach software development to Computer Science students. As it is new, however, there is still a lack of proper coverage of this methodology in the literature. Furthermore, the area of patterns for ATDD still needs to be started. This dissertation involves performing an investigative study on best practices and patterns for ATDD, defining the application of the methodology with a software development process point of view and summarizing gathered experience with ATDD as a means of teaching software development. The result of these activities was an introductory text on ATDD, which we hope will serve not only to help newcomers yield more from the methodology, but also to divulge its benefits. The examples in the text use EasyAccept, a scripted acceptance testing tool, as a means of exposing the methodology.

Agradecimentos

Ao meu orientador Jacques Sauvé, pela reiterada confiança em mim, por me ter aberto as portas e me conduzido nos caminhos que tornaram esse trabalho possível.

A meus queridos tios Lucas e Marinete, pelo segundo lar de que me deixaram fazer parte por tantos anos e do qual recordarei com muito carinho para sempre. Espero poder retribuir-lhes algum dia tudo aquilo que recebi de coração aberto.

A Walfredo Cirne, meu co-orientador durante grande parcela do programa, pelos importantes comentários e sugestões, e por ter me acolhido com tanta amizade no LSD.

Às amigas Roberta e Ayla, que ajudaram a dar forma a idéias que estão contidas nesse trabalho.

A todos os colegas do DSC e do LSD, pela agradável convivência durante minha permanência em Campina Grande. Em particular, gostaria de agradecer de forma especial àqueles que foram mais amigos em momentos diferentes: Alexandre e Eloi na antiga sala da CHESF; os dois Rodrigos e Filipe na sala do Bottom Line; Raquel, Ana, Lívia e Iury na Dona Bica do LSD; Lauro, nunca na mesma sala, mas sempre presente.

A Ana e Vera, meu muito obrigado pela ajuda de sempre e também meu pedido de desculpas por ser um mestrando tão trabalhoso.

Finalmente, gostaria de agradecer a meus pais e minha família, que me apoiaram em todos os sentidos nesse trabalho, mesmo sem saber exatamente o que estive fazendo; a minha irmã Roberta, iniciadora dessa idéia; e a meus amigos que silenciosamente me sugestionam a tomar os caminhos corretos, segundo aquilo que combinamos antes de eu chegar aqui.

Sumário

Índice de Figuras

Índice de Tabelas

Lista de Abreviações

ATDD – Acceptance Test-Driven Development
FiT – Framework for Integrated Testing
IDE – Integrated Development Environment
TDD – Test-Driven Development
XP – Extreme Programming

1

padrões de software; experiência com a aplicação da metodologia em projetos de desenvolvimento de software; análise de massas de testes de aceitação gerados por equipes de desenvolvimento que aplicaram a metodologia; e finalmente experiência com o ensino da metodologia a alunos de graduação em Ciência da Computação.

Esse trabalho é relevante no sentido em que atua para avançar o estado da arte da metodologia ATDD e promover sua divulgação e adoção. Como resultado da investigação, foi dado início à catalogação de padrões e melhores práticas para a metodologia, gerando um conjunto inicial de artefatos que resume a experiência que nós e outros tivemos com a metodologia. Esse conjunto inicial serve como um repositório colaborativo que tende a crescer ao longo do tempo, e se torna importante tanto para aqueles que desejam adotar a metodologia ATDD quanto para aqueles que já a utilizam.

Como resultado adicional da investigação, incluímos neste trabalho um exemplo de aplicação da metodologia no contexto de um processo de desenvolvimento ágil, além de um resumo de nossa experiência no ensino da metodologia. Estes artefatos servem como guias para participantes de um projeto de desenvolvimento de software e professores universitários, respectivamente. A dissertação, além de servir o propósito adicional de introduzir neófitos à metodologia ATDD, resume todo o conhecimento produzido como resultado do trabalho investigativo e, tendo boa parte de seu conteúdo sido escrito em inglês, funciona como instrumento de mais ampla divulgação da metodologia.

Esta dissertação está estruturada em duas seções. Na primeira seção, são apresentados os fundamentos da metodologia ATDD, da seguinte forma: o capítulo 2 expõe os fundamentos da análise executável, o primeiro pilar do ATDD, incluindo uma discussão histórica sobre Engenharia de Software e quais os problemas que a análise executável se propõe a resolver. No capítulo 3, a metodologia é apresentada em detalhes através de um exemplo completo: todo o ciclo de desenvolvimento de um projeto hipotético para um software simples é mostrado, desde a análise, passando pela criação dos testes, até o código final escrito. O capítulo 4 define como o ATDD pode ser usado no contexto de um processo de desenvolvimento baseado em Extreme Programming (XP) *[Beck99]*, apresentando que adaptações precisam ser realizadas no processo para acomodar as novas técnicas e a participação do cliente.

Na segunda seção, é feita a discussão dos padrões levantados para a metodologia. A seção se inicia com uma discussão geral sobre padrões, seguida da exposição de como se deve criar uma *linguagem de script*, o vocabulário que define os comandos que serão usados nos scripts de teste. O restante da seção discute cada um dos padrões de criação e organização de testes de aceitação, além de padrões de aplicação dos testes dentro da metodologia ATDD.

Ao final da dissertação, há três apêndices. O primeiro deles consiste em um manual de referência da ferramenta EasyAccept, que foi usada nos exemplos ao longo do texto. O segundo apêndice trata em mais detalhes, com exemplos adicionais, a questão da sincronia entre testes de aceitação e de unidade dentro da metodologia ATDD, e serve como complemento ao capítulo 3. O terceiro apêndice resume a experiência que tivemos no ensino da metodologia a alunos de Ciência da Computação.

Após os apêndices, encontra-se a conclusão da dissertação, incluindo a discussão das suas limitações e contribuições, bem como o levantamento de trabalhos futuros.

2

There is one further problem with software charts and diagrams, as opposed to a house plan, for example. In software development, as in any other human endeavor involving many people, communication plays an essential role and is a common source of errors. Everyone involved in the project must keep focused on the same goals, and thus everyone must understand exactly what must be done according to what the client wants. What is meant by a house construction plan is taken for granted, as it visually displays the measures, the spaces and details the final house will have. With typical software diagrams, however, despite the greater precision and rigor they compel (as compared, for example, to mere textual descriptions), there still seems to be plenty of room for misunderstanding due to multiple views and interpretations.

The breakdown stems from gaps in the communication between various actors involved in software development:

- Clients may not correctly express what they want (or, as stated above, may not even know what they want concretely);
- Analysts may not completely understand what the client wants even if they do express it well;
- The translation of requirements and rules into diagrams may not be effective;
- Clients may fail to understand the diagrams (as they require technical background to be understood) and misleadingly approve them;
- Developers may overlook details or interpret the artifacts in a wrong way, resulting in bugs, unmet requirements and feature creep;

In order to reduce the effect of such communication gaps in software development, the direction that has been historically given to software development was that of reducing more and more the separation of roles between analysts and programmers. Analysts and programmers had formerly well-defined non-overlapping roles. Analysts talked to the clients to capture requirements and handed them to programmers, who created working software. People concluded that blurring the two roles into a single common “developer” denomination, with mixed responsibilities, allowed less information to be transferred and thus less miscommunication would arise. However, it is hard to find professionals that excel in both roles. The profile of a typical analyst is that of an expansive, talkative professional specialized in negotiating with clients and getting the most out his communication skills to capture software requirements. On the other hand, programmers tend to be technically-oriented, more comfortable with machines than with people.

Another solution to reduce communication problems would be to find a method that can integrate the language of clients, analysts, developers and testers using a non-ambiguous, verifiable, automatable, expressible and readable artifact:

- *Non-ambiguous* because there mustn't be multiple interpretations of what the artifact represents;
- *Verifiable* because the artifact must correctly reflect software characteristics and behavior the client wants, so there must be a way to check for such correctness;
- *Automatable* because it is not feasible to do a complete verification of software manually, given its complexity and the frequency with which testing must be done. Furthermore, with the possibility of knowing on the fly if the code they are writing is correct (through the execution of a full set of automated tests), developers lose the fear of changing and refactoring their code, improving its quality;
- *Expressible* because there mustn't be requirements or rules that can't be written or that are too difficult to write with the artifact;

- *Readable* because the client, a non-technical actor involved in the process, must be able to understand, discuss and approve the content of the artifact.

We call such a method "executable analysis". The output of the analysis activity - which involves requirements capture and processing, elicitation of business rules, etc. - is executable analysis artifacts that can be used to test and verify software in an automated way. If you make your analysis executable, you can address all those issues discussed previously.

Furthermore, as executable analysis allows requirements to be captured unequivocally, an efficient separation of roles between analysts and developers can be finally established. Analysts can capture requirements efficiently and translate them into executable requirements. Programmers need not have as much direct contact with the client as analysts and simply create the code that meets the executable requirements. Maybe software development will revisit its roots?

The stage is set for this to happen, because executable analysis has entered the realm of feasibility with the emergence of artifacts that enable it. But then, what is the ideal executable analysis artifact?

Using Client-Readable Acceptance Tests As Executable Analysis Artifacts

Examining the possible artifacts one could use to represent requirements, on the one extreme of the range of possibilities are formal, mathematical expressions that describe requirements in a rigorous way. On the other extreme are loose textual descriptions written in the client's natural language. Textual descriptions are the most expressible and readable of all possible artifacts, but are also highly ambiguous, hard to verify given the current state of the field of natural language processing, and not at all automatable. On the other hand, formal mathematical descriptions are highly precise, verifiable and automatable, but lack readability and especially expressiveness. There is clearly a tradeoff to be explored between client-side virtues of the artifact (expressiveness and readability) and development-side needs (nonambiguity, verifiability, automatability). A compromise must be found between the two sides.

Client-readable acceptance tests can serve well as this compromise.

First, let's state what acceptance tests are in isolation.
Per se, acceptance tests – sometimes also called functional tests – are black-box system-level tests. While not necessarily readable by the client, they are used by him to check whether or not the software does what it should. That's the origin of the name: the client either "accepts" or "rejects" the software produced depending on the test results. Such tests involve functions that make sense to the client. For him, it doesn't matter *how* such functions under test are implemented (tests are black-box).

By nature, acceptance tests comply with the developer-side requirements for an executable analysis artifact (non-ambiguous, verifiable and automatable). To comply with the remaining requirements (expressiveness, readability), acceptance tests should be written in a format that clients can easily grasp. That makes acceptance tests client-readable: in addition to automating software testing, their representation can be read, understood and thus discussed with the client. To make acceptance tests client-readable implies that such tests mustn't be

written using programming language code, or involve programming entities like classes, objects, data structures, etc., because a regular client doesn't understand these concepts.

Now, let's state what client-readable acceptance tests are not.
They are not tests for the user interface; even though most of the acceptance testing tools used in the industry involve the client interacting with a user interface to record his activities, the point of acceptance tests is not testing the user interface. Rather, they are used to test the underlying business logic of the program under test (that is, the rules, the calculations, the workflow of actions - in summary, functional requirements for the software), even if, to do so, they access user interface elements like buttons and drop-down menus to capture useful client examples. We don't argue that one should not test the user interface, though; quite on the contrary, the final "acceptance" of a program includes the client's satisfaction with the user interface, and testing it is an essential step toward this satisfaction. However, we believe testing the user interface is a separate concern from testing the business logic.
They are not tests for low-level software code; even though one could use an acceptance testing tool to test specific low-level pieces of software code, like the inner workings of a data structure or class, that generally is not a client concern (unless they involve a client's business rule, in which case he might want to discuss the situation), so that shouldn't be incorporated in the test suite as an acceptance test. Rather, programmers should write those tests as *unit tests* to help them gain confidence in the code they write.

Acceptance Testing Tools

In the last few years, a number of tools have emerged to help create and run client-readable acceptance tests. Any of those tools, as long as they really deal with *client-readable* acceptance tests, can be used to yield the benefits of executable analysis.

The most widely known of these tools is FiT (Framework for Integrated Testing), created by Ward Cunningham and others. It uses acceptance tests represented as tables enclosed in HTML files. The rationale behind this approach is that tables are easily understood by clients and can be created in widely available spreadsheet editors. Upon running FiT, the tables are linked to the program being tested using hookup code called *Fixture* code. Fixtures are provided by developers and are composed of attributes and methods that correspond exactly to the column titles in the test tables.

Most other tools use a text-driven approach, in which the tests are described using a format that resembles natural language - mostly, a sequence of sentences with verbs and nouns that serve as commands. Commands are then matched to methods in the hookup code that allow the program under test to be accessed.

Throughout this text, we will use examples of acceptance tests written in the format of EasyAccept, a text-driven acceptance testing tool created by Sauvé *et al*. EasyAccept takes acceptance tests enclosed in plain text files. Tests are expressed as a script consisting of a sequence of lines, most containing a command to access the business logic of the program being tested (there are also comments, for example). Clients and developers jointly define the set of commands that will be used to express the tests. To run the tests, you give EasyAccept the text files containing the tests, and an "entry point" to the program under test containing methods that match the commands defined. EasyAccept then runs the script line by line and reports back to the user the results of the tests executed.

Using Acceptante Tests to Test Existing Software

With tools like EasyAccept or FiT, one can easily probe existing software with a Façade or Fixtures, which do not change existing software code, and create useful acceptance tests for current functionality as well as for features that will be added. As the mass of tests evolves, developers not only gain more and more confidence that the new functionalities work, but also lose the fear of changing and refactoring existing code.

Using Acceptance Tests to Drive Software Development

An even more powerful application of client-readable acceptance tests is to use Acceptance Test-Driven Development (ATDD), an approach that yields the full range of benefits that executable analysis has to offer. In short, it consists in developing software from its inception driven by acceptance tests, in a write-tests-first approach.
Chapter 3 outlines a simple example of how this works with EasyAccept, and chapter 4 discusses ATDD from a software development process point of view.

The Need for Acceptance Testing Patterns

Patterns are effective solutions to solve common problems in a given field of application. Although originally conceived by Christopher Alexander, an architect, they have been mostly influential in the IT industry. In software engineering, the patterns movement (to find and spread patterns to solve software development problems) has grown with the popularity of *design* patterns. Singleton, Factory Method and Façade have become common words for developers in the software industry. Design patterns have even been integrated into the Computer Science curriculum of top universities.

Patterns have short, suggestive names to help convey and memorize the solution they enclose. Furthermore, they provide a common vocabulary for discussing solutions for the problems. They emerge from the observation that a certain recurrent problem can be solved effectively with a simple, but not always obvious solution. The identification and propagation of patterns is useful because of the common vocabulary and also to lead inexperienced people away from cumbersome solutions.

As the number of people using client-readable acceptance tests grows, the need and search for related patterns begin. What is the best way to create acceptance tests? How to effectively organize the tests, so that you boost communication through the tests? How to make the best use of tests in software development? This text provides an initial set of patterns that we hope will evolve with time.

3

folder you choose, make sure *easyaccept.jar* is included in the CLASSPATH, so that EasyAccept can be found by the Java compiler.

EasyAccept is run from the command line (or within an IDE or ant) with the following syntax:

```
java easyaccept.EasyAccept <FacadeName> <test file 1>.. <test file n>
```

Instead of specifying multiple test files, a folder that includes the test files can be referenced. In that case, EasyAccept runs all files it finds in the folder structure.
Examples of EasyAccept usage:

```
java easyaccept.EasyAccept ApplicationFacade us1.txt
java easyaccept.EasyAccept ApplicationFacade us1.txt us2.txt us3.txt
java easyaccept.EasyAccept ApplicationFacade acctests
```

(wherer `acctests` is a folder that contains many acceptance test files)

A Programming Session

In order to illustrate how EasyAccept is used, we have chosen to describe a programming session in which a client and a developer work together to create a simple program, a poll manager. This program, albeit unpretentious, has a good enough size to serve the aim of showing how the approach works.

Poll Program Outline

Alice is a client who wants a simple poll program to be created. With this program, Alice, an outspoken and highly sociable person, wants to create polls to know even more about the opinions of her thousands of friends. A typical poll consists of a question and a number of answer choices. The program users – Alice's friends, henceforth called *voters* – pick one among the set of possible answers for any given poll. Among the features Alice wants the program to have are the following: management of multiple polls (create, read, update, delete or CRUD operations), generation of reports of the poll results, voter authentication with passwords, and so on.

Although Alice has no trouble operating computers, she has no programming background whatsoever, so she hired Bob to do the job. He is the software developer who will materialize Alice's ideas. An experienced but open-minded developer, Bob had already realized the advantages of adopting test-driven development coupled with acceptance tests (Acceptance Test-Driven Development, or ATDD) to create software with focus and confidence. Since Alice is relatively "available" as a client, i.e., she can be easily contacted and can give prompt feedback over any doubt Bob might have regarding requirements, Bob suggested the usage of an ATDD approach to develop the poll program.

Alice thought of many features she wants her poll program to have. However, Bob asked her to choose the minimum functionality she needs implemented in the first *iteration* of development. At the end of this iteration, Bob will give her back fully working software – for

the set of features chosen. After giving the issue some thought, Alice comes up with the following list of *user stories*[3] for the first iteration of the poll program:

User story 1 – Let the user create a poll. The user must provide one poll question and at least 2 answer choices.

User story 2 – Let the user cast a vote for a poll. He chooses a poll and one of the possible answers.

User story 3 – Generate a report with the results of a poll, including the total number of votes for each option and the corresponding percentage.

Knowing the user stories he needs to implement, Bob chooses the first one to tackle. He tells Alice that, before he can write any code, some acceptance tests must be created. Such tests aren't meant to test the user interface, which she may well consider later (command line, windows, buttons, etc.), but rather they serve to test the program's business logic, that is, the rules, requirements and processes that go hidden in the inner workings of the program and often are taken for granted when people only test the user interface. Even though EasyAccept can potentially be used to test a user interface (if you associate script commands with GUI component operations, for example), the final acceptance of the interface by the user is mostly visual and manual, consisting in the user operating the program and evaluating the "look and feel" of the interface.

Bob now opens a text editor and creates a new file named *poll-us1.txt*. In this file, he'll write some acceptance tests for the first user story. As an experienced EasyAccept test writer, he won't bother calling a testing specialist to help him with the test creation, considering that this is a simple program. For this first user story, he knows the testing pattern **Creator** will work just fine to write good tests.

In order to write the tests, Bob needs to invent a number of commands to manipulate the program under test. For the first user story, it is immediately apparent from the user story description (based on the verbs Alice used to express the feature) that he will need a *doer* command, `createPoll`, and some *getters* to check if the poll was created successfully. When discussing how to create a script language, i.e., the set of commands or verbs you use to test software, we'll talk about doers, getters, preparers and the process Bob used to choose the right commands to express tests. For now, suffice to say that doers are actions a user of the program would typically execute, and getters are commands to get information back from the program under test. If you'd like to have a peek at that now, feel free to refer to *Creating a Script Language* in Section 2.

A First EasyAccept Test Script

Bob starts writing tests for the first user story. Its description states that a poll must have a question and at least two answers. "But then", Bob ponders, "don't polls have names?". He talks to Alice, explaining that should this information not be used in the program, the poll's

[3] User stories are an integral part of agile software development and Extreme Programming (XP), in particular, on which ATDD is based. If you are not familiar with XP or need to understand user stories in more depth, please refer to chapter 3, which details the ATDD process.

question itself could work as its name. Alice says she really won't need a poll's name. Using the text editor, Bob then writes the following lines into *poll-us1.txt*:

```
1 # user story 1 - Let the user create a poll. You need one poll question
2 # (which is also the name of the poll) and at least 2 answer choices
3 createPoll name="Do you like apples?" answers={yes,no}
```

The numbers put at the beginning of each line are there to easily identify lines when we refer to them in the text. They aren't actually part of the script. Let's analyze these first lines Bob has written. Notice the first two lines begin with a '#' character. Those are comment lines that are ignored by EasyAccept when running the script. They serve the purpose of communication among developers and clients and should be used extensively to explain and clarify what the tests are doing. Bob included in the comments the user story description and the clarification on polls' questions serving as names. Every time the developers have doubts regarding how the software should work, they should ask the client for clarifications and include them in the scripts as comments – see the patterns **Client Assertion** and **Commentor**, in Section 2.

Line 3 consists of a single command, `createPoll` (the doer command invented by Bob). He wants to create a poll with the name (which is also the question to be answered) "Do you like apples?", and two answer choices: "yes" or "no". After writing these lines, Bob saves *poll-us1.txt*. Before he can run the script *poll-us1.txt* with EasyAccept, though, he will need to write a Façade for the (still nonexistent) program under test; otherwise he won't even be able to call EasyAccept. If he tries, by typing in the command line (actually, a developer will probably use this line within and IDE, or with a tool like ant):

```
java easyaccept.EasyAccept PollFacade poll-us1.txt
```

he gets back the following message:

```
Facade not found: PollFacade
```

The Façade also helps when Bob implements a user interface for Alice in the future. He can use its methods to directly access the underlying program's business logic in a clean way, separating user interface and business logic concerns automatically from the very beginning of development. He then opens his Java editor, creates a class named `PollFacade` and simply leaves it blank, as in the code below, just to enable EasyAccept to be run:

```
public class PollFacade { }
```

Now that `PollFacade` exists, Bob runs the script using EasyAccept for the first time, typing again in the command line:

```
java easyaccept.EasyAccept PollFacade poll-us1.txt
```

That calls EasyAccept to test the poll program through the Façade `PollFacade` using the script *poll-us1.txt*. The result of the run was, unsurprisingly, unsuccessful. This is the output of EasyAccept:

```
Test file poll-us1.txt: 1 errors:
Unkown command: createPoll name="Do you like apples?" answers={yes,no}
Command producing error: <createPoll name="Do you like apples?"
answers={yes,no}>
```

The reason for the error is that Bob still has a blank Façade. Every command found in the script must correspond to a method in the Façade with a matching signature, but EasyAccept couldn't find a corresponding method for `createPoll`. Bob then edits `PollFacade` to include the method that matches `createPoll`. The resulting class follows:

```
public class PollFacade {
      public void createPoll( String name, String answers ) { }
}
```

Now Bob's Façade includes a method `createPoll` which takes a `String` as the name of the poll and a `String` of answer choices for the poll, formatted as a comma-separated list of items enclosed in curly brackets – "`{`" and "`}`" (more on this syntax below).

Upon running EasyAccept once again, Bob gets the following message as a result:

```
Test file poll-us1.txt: 1 tests OK
```

He has just run his first successful EasyAccept script for Alice's poll program with no errors. Bob could be happy if only the script were testing anything useful. The script must be improved, so Bob adds a few more lines. For now, it simply consists in a poll being created; but there must be tests to assess if the poll was created successfully. Just for the record, Bob is applying the **Creator** pattern, which will be discussed in section 2. This is the point where getters come into play. Those are new commands invented by Bob to create useful tests in the script. The resulting *poll-us1.txt* follows:

```
1 # user story 1 - Let the user create a poll. You need one question
2 # (which is also the name of the poll) and at least 2 answer choices
3 poll1=createPoll name="Do you like apples?" answers={yes,no}
4 expect "Do you like apples?" getPollName poll=${poll1}
5 expect "{yes,no}" getPollAnswers poll=${poll1}
```

Let's introduce a few new elements of an EasyAccept script that Bob has used to compose these tests. First observe the new commands he has introduced – the getters `getPollName` and `getPollAnswers`, which are combined in an EasyAccept script with the *built-in command* `expect`. This command compares the result returned by a script (business logic) command to an expected result declared in the script. When EasyAccept is run, if these two values don't match, the user is informed of the divergence that was found. In line 4, Bob is stating that he wants the program to return "Do you like apples?" as the result of a call to the getter `getPollName`. Likewise, in line 5, the collection composed of the answers "yes" and "no" must be returned when `getPollAnswers` is called (remember that the syntax Bob used to declare the answers consists of comma-separated items enclosed in curly brackets). We'll see in a few paragraphs the outcome of a failing `expect`-composed command, but let's first explain one more feature found in this script.

Notice that line 3 now begins with the statement `poll1=`. This is an EasyAccept *variable assignment*. Within a script, EasyAccept can take strings of characters returned by a command and store them into variables declared by the test writer with the symbol "=". These

variables can be referenced in other points of the script using the symbol "$" followed by curly brackets enclosing the variable name. In Bob's script, he declares the variable `poll1` in line 3 as receiving a string value returned by the command `createPoll`. The exact string value returned by the command `createPoll` remains hidden to the test reader. This value could be any string, but Bob's intention (as he also turns out to be the developer of the program) is to provide a mechanism of attributing unique identifiers to business objects created within the poll program and later accessing them. This is an acceptance test creation pattern called **Business Object Reference** (BORef, for short). We will talk more about keys and how they can be used to reference business objects when discussing how Bob implemented the program. Additionally, a discussion of the BORef pattern can be found in Section 2, should you want to take a look at it now.

After saving *poll-us1.txt* once again, Bob runs the tests again. As he hasn't implemented the new methods in the Façade yet, EasyAccept shows 2 errors in the script:

```
Test file poll-us1.txt: 2 errors:
Line 4, file poll-us1.txt: Unkown command: getPollName poll=
Command producing error: <expect "Do you like apples?" getPollName poll=>
Line 5, file poll-us1.txt: Unkown command: getPollAnswers poll=
Command producing error: <expect "{yes,no}" getPollAnswers poll=>
```

To overcome the errors accused by EasyAccept, Bob defines the getter methods in `PollFacade`. It now becomes:

```
public class PollFacade {
      public void createPoll( String name, String answers ) { }
      public String getPollName( String poll ) { return ""; }
      public String getPollAnswers( String poll ) { return ""; }
}
```

Notice that Bob still hasn't coded the actual implementation of a poll being created and its attributes being returned as the getters are called. He works in little steps, correcting each little error at a time. "Didn't EasyAccept complain that it couldn't find the getter commands?", he said, "Then let's create them". "Now, let's see what else it wants …" The result of the new run of EasyAccept was:

```
Test file poll-us1.txt: 2 errors:
Line 4, file poll-us1.txt: Expected <Do you like apples?>, but was <>
Command producing error: <expect "Do you like apples?" getPollName poll=>
Line 5, file poll-us1.txt: Expected <{yes,no}>, but was <>
Command producing error: <expect {yes,no} getPollAnswers poll=>
```

Now EasyAccept recognizes the getters. The new problem is that it found divergences between expected results and actual results output from the program. The program returned blank strings, when "Do you like apples?" and "{yes,no}" were expected as the name and answer choices for the poll supposedly created.

"Big deal", he thinks. "I'll just make the getters return what the test script is asking." He modifies the getters so that they return the correct answers:

```
public String getPollName( String poll ) { return "Do you like apples?"; }
public String getPollAnswers( String poll ) { return "{yes,no}"; }
```

Now look at the output of EasyAccept:

```
Test file poll-us1.txt: 3 tests OK
```

Ok, nice … But wait! Bob just cheated with EasyAccept, didn't he? This won't make him produce working software … Never mind, please wait just a few more paragraphs and you'll understand the way the technique works. Bob must always use the simplest code to make the tests pass. He strives to see the message above ("… tests OK") for the tests he is currently working on (and also for all preceding tests that already pass), before making big changes to the code. He can easily get lost if he doesn't follow this simple rule. As more test lines are dealt with and coding progresses, he will be forced to create the correct code (see below).

Bob adds a few more test lines to *poll-us1.txt*. The script now creates two polls and checks that they were created. Check out the new additions below:

```
1 # user story 1 - Let the user create a poll. You need one question
2 # (which is also the name of the poll) and at least 2 answer choices
3 poll1=createPoll name="Do you like apples?" answers={yes,no}
4 poll2=createPoll name="Do you like oranges?" answers={sure,nope}
5 expect "Do you like apples?" getPollName poll=${poll1}
6 expect "{yes,no}" getPollAnswers poll=${poll1}
7 expect "Do you like oranges?" getPollName poll=${poll2}
8 expect "{sure,yikes}" getPollAnswers poll=${poll2}
```

Upon running EasyAccept, the results are:

```
Test file poll-us1.txt: 2 errors:
Line 7, file poll-us1.txt: Expected <Do you like oranges?>, but was <Do you
like apples?>
Command producing error: <expect "Do you like oranges?" getPollName poll=>
Line 7, file poll-us1.txt: Expected <{sure,nope}>, but was <{yes,no}>
Command producing error: <expect {sure,nope} getPollAnswers poll=>
```

Errors were detected because the program is always outputting the same results (the first poll's attributes) for any poll, and divergences were found when the test script asked for the second poll's name and answers.
Bob must now stop "cheating" with EasyAccept, if he wants all these tests to pass. He must do some serious coding and implement the actual poll creation, so that the getters can return the correct poll names and answers. Before starting with Bob's implementation, though, we must introduce in a few lines some concepts that will be needed to fully understand what he'll do.

Business Objects

Let's talk a bit about *business objects*. They are logical entities manipulated by the program that *make sense to the client*, i.e., clients can understand what they mean and represent. They are also the domain names a client employs when discussing software requirements with the developers. A supplier, an order, a product item in a sales program; a player, a board place, a die in a Monopoly game; in Alice's program, a poll, a voter, a vote.

Business objects may or may not (but typically do) correspond to actual single software objects. Even if they don't (e.g., if a number of interrelated objects and structures as a whole

actually serve as a single business object), that remains hidden to the client. The client doesn't care *how* a particular business object is actually internally represented in the program.

Thinking the other way around, the only kind of object that should appear in an acceptance test script is the business objects kind (that's the **Only Business Objects** pattern, see section 2). A client doesn't understand non-BOs and will get confused if they start appearing in a script. For example, specific data structures like trees, queues, linked lists; database connections, middleware components; mock objects and design pattern-related objects. In most cases, those are low-level concepts that are out of a client's grasp and will likely turn a script into a complicated and unintelligible mess.

However, we don't mean to say that there shouldn't be tests for these structures. On the contrary, everything must be fully tested to ensure the code will work. What we mean is that programmers should employ *unit tests* (especially with test-driven development) to gain confidence in their low-level code. When a programmer runs an EasyAccept script and some test fails, he must know where the error lies. If a lower level unit test captures the exact point of failure, the better for the programmer. Bob will employ unit tests in the poll program's code in a few paragraphs. For now, let's make sure you focus on another concept: business object references.

Business Object References

A reference to a business object is a unique identifier with which it can be unequivocally referred to in a script. As we have introduced a few pages above, *string-valued keys*[4] are used to provide unique identifiers to BOs within a script. Test writers write tests using variables, such as `poll1` and `poll2`, and the value that is stored in these variables is what determines the uniqueness of a business object. Programmers must code the mechanism of unique attribution to an object, as you will see in Bob's implementation below. This is the **BORef** pattern: use keys as variables coupled with an internal uniqueness mechanism to create references to business objects in a script.

Implementing the First User Story

Bob starts coding "for real", in order to comply with the new user story 1 tests. He will need a longer step this time to make all the tests pass because he needs to implement the full creation process of a poll. Bob always begins the thought process in a top-down fashion starting with the Façade; as the Façade must access the main program entities through a simple interface, it is a good opportunity to figure out which such entities he'll need.

For a simple program such as Poll, Bob simply makes calls to a main class he named `PollSystem`. The Façade's methods simply push the responsibility of doing the job further to `PollSystem`. Bob's first implementation of `PollFacade` turns out like this:

```
package poll;

import util.MalformedStringException;
import util.StringParser;
import java.util.Collection;
```

[4] Keys are not necessarily string-valued

```
public class PollFacade {

	private PollSystem pollSystem;

	public PollFacade() {
		pollSystem = new PollSystem();
	}

	public String getPollName(String poll) {
		return pollSystem.getPollName(poll);
	}

	public String getPollAnswers(String poll) {
		Collection answers = pollSystem.getPollAnswers(poll);
		return StringParser.collectionToString(answers);
	}

	public String createPoll(String name, String answers)
		throws MalformedStringException {
		return pollSystem.createPoll(name, StringParser
				.stringToCollection(answers));
	}
}
```

In addition to forwarding the calls to corresponding methods in `PollSystem`, `PollFacade` also deals with the conversion and formatting of string collections to the syntax used in the test script (comma-separated items enclosed in curly brackets) via the utility methods of the class `StringParser`: `collectionToString` and `stringToCollection`. The former takes a `Collection` and returns a `String` in the format required, and the latter takes a `String` and parses it into a `Collection`. The exception `MalformedStringException` is thrown by `StringParser` when one tries to convert into a collection a string with an invalid format.

This makes a perfect example of the point we wanted to make previously when discussing business objects. `StringParser` is an inner program entity that isn't accounted for in the acceptance tests. Should there be tests for the string parsing/formatting? Of course! Everything must be fully tested. But should those tests be in `poll-us1.txt`? Only if that makes sense to the client. Bob believes that this is too much detail for Alice, but he needs to create tests for the string conversion and formatting anyway. This is where *unit tests* come in handy.

In order not to divert this chapter from the goal we set (and also because a number of the readers might already be well acquainted with unit tests), we have removed the discussion of this step to appendix II. Feel free to read that chapter now if you know absolutely nothing about unit tests; alternatively, you can believe that `StringParser`'s code works just fine and skim that appendix later.

As a side note, also observe that these conversion functionalities could well be EasyAccept's job. As a matter of fact, by the time you read this, collection utilities including conversions will likely be a built-in feature of EasyAccept. We decided to exclude the automatic conversion from this text to illustrate how Bob merged development with acceptance tests and unit tests.

Bob then codes `PollSystem`, since `PollFacade` can't be compiled due to numerous references to that still nonexistent class. While sorting out and filling in the methods of

`PollSystem`, Bob takes the opportunity to think out the low-level design of the classes he'll need. "A poll system handles polls …" – he ponders – "For now, the system just needs to create them and allow access to their attributes. Let me create a `Poll` class to be instantiated as needed in a `PollSystem`."

Bob only codes what is needed. Even though he knows the `Poll` class will have multiple responsibilities in the future, such as counting votes and handling access to specific poll answers, this first version simply does what the tests demand. `Poll` then turns out to be a simple class (for the moment) that takes a name and a collection of answers and stores that information for later retrieval.

```
package poll;

import java.util.Collection;
import java.util.List;
import java.util.LinkedList;
import java.util.UUID;

public class Poll {

	private String name;
	private List<String> answers;
	private String id;

	Poll( String name, Collection<String> answers ) {
		this.name = name;
		this.answers = new ArrayList<String>( answers );
		/* get a randomized unique universal identification for id */
		this.id = UUID.randomUUID().toString();
	}

	String getName() { return name; }

	List getAnswers() { return answers; }

	String getID() { return id; }

}
```

In the constructor, Bob implements the unique identification mechanism for a poll. He employs the Java class `util.UUID` to generate a random unique String in the program, and uses it as a poll's id. Additionally, he included the `getID` method, so that `PollSystem` can access at any time the poll's id.

Bob sets out to complete `PollSystem`, now that he's got a `Poll` class. See the code for `PollSystem` below, followed by comments.

```
package poll;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

public class PollSystem {

	private Map<String, Poll> polls;
```

```
        PollSystem() {
              polls = new HashMap<String, Poll>();
        }

        String createPoll(String name, Collection<String> answers) {
              Poll p = new Poll(name, answers);
              polls.put(p.getID(), p);
              return p.getID();
        }

        String getPollName(String poll) {
              return getPoll(poll).getName();
        }

        Collection getPollAnswers(String poll) {
              return getPoll(poll).getAnswers();
        }

        Poll getPoll(String pollID) {
              return polls.get(pollID);
        }
}
```

`PollSystem` stores polls into a `Map`, so that a poll can be easily retrieved given its id (that is done in the method `getPoll`). The methods `getPollName` and `getPollAnswers` use `getPoll` to find a specific poll and return the desired attribute. Finally, the method `createPoll` takes a poll's name and answers, creates the corresponding `Poll` object and puts it into the `Map` using its id as the key.

You may have noticed how little Bob is handling errors and exceptions in all of the preceding code. Please take that for granted for now, because he'll do so in a few moments when tests for errors are introduced in the script. In a typical round of user story implementation, Bob would have introduced those tests for errors in his very first and full-fledged version of *poll-us1.txt*. However, we asked him to co-write tests and code in little chunks so that our exposition of EasyAccept's syntax and the ATDD methodology would be softened.

Bob runs EasyAccept once again and gets the message:

```
Test file poll-us1.txt: 6 tests OK
```

The code Bob just wrote passes all tests, i.e., the poll program successfully creates polls and is able to get information back from them. However, this functionality is far from fully tested. A good testing script must submit the program to as many error, limit and special situation checks as possible.

Testing for Errors

Bob adds a few more test lines to *poll-us1.txt*. He uses an acceptance test creation pattern called **Command Errors** to cover the tests he'll need. This pattern consists in analyzing, for each script command created, error conditions such as invalid arguments or invalid usage of the command. For the command `createPoll`, Bob recalls that a poll can't be created without a name, and at least two answer choices must be provided (those are requirements stated in the user story's description). For the getters, he finds required tests would be to check if a valid existing poll is passed as an argument to the command.

```
1 # user story 1 - Let the user create a poll. You need one question
2 # and at least 2 answer choices
3 poll1=createPoll name="Do you like apples?" answers={yes,no}
```

```
Command producing error: <expectError "Poll does not exist" getPollName
poll=abc>
Line 20, file poll-us1.txt: Expected the error message <Poll does not
exist>, but the error message was <(no message: exception =
java.lang.NullPointerException)>
Command producing error: <expectError "Poll does not exist" getPollAnswers
poll=abc>
```

EasyAccept reported either no errors happening in the first four tests or different errors than expected in the last two tests (uncaught `NullPointerException`s were thrown). As we previously said, Bob hadn't worried about exceptions in the first place; now he must modify the code to cope with them.

The methods that must throw exceptions are those in `PollSystem`: `createPoll` and `getPoll`. Bob changes `createPoll` to add lines testing for the error conditions required in the script (we will deal with `getPoll` in a few paragraphs). The new code follows:

```
public String createPoll(String name, Collection<String> answers)
            throws PollCreationException {
      if (name == null || "".equals(name))
            throw new PollCreationException("Poll must have a name");
      if (answers.size() < 2)
            throw new PollCreationException(
                        "Poll must have at least two answers");
      Poll p = new Poll(name, answers);
      polls.put(p.getID(), p);
      return p.getID();
}
```

Bob then creates the corresponding exception classes in a hierarchy of specific exceptions to the poll program: `PollCreationException` extending a generic `PollException` superclass.

```
package poll;

public class PollException extends Exception {
      public PollException(String message) {
            super(message);
      }
}
```

```
package poll;

public class PollCreationException extends PollException {
      public PollCreationException(String message) {
            super(message);
      }
}
```

The Façade must also be changed. As it should simply forward to EasyAccept exceptions thrown in the inner program, the only modification he makes to the Façade is the inclusion of Java `throws` clauses to `createPoll`. Recall that `MalformedStringException` is related to `StringParser` and is part of the `util` package (refer to appendix II for more detail). The Façade method's signature now become:

```
public String createPoll(String name, String answers) throws
MalformedStringException, PollCreationException
```

Now an exception is thrown every time one tries to create a poll with a blank name or with less than two answers. Bob runs EasyAccept and checks if the modification works. The output was

```
Test file poll-us1.txt: 2 errors:
Line 19, file poll-us1.txt: Expected the error message <Poll does not
exist>, but the error message was <(no message: exception =
java.lang.NullPointerException)>
Command producing error: <expectError "Poll does not exist" getPollName
poll=abc>
Line 20, file poll-us1.txt: Expected the error message <Poll does not
exist>, but the error message was <(no message: exception =
java.lang.NullPointerException)>
Command producing error: <expectError "Poll does not exist" getPollAnswers
poll=abc>
```

Ok, that means the first 4 tests for errors (relating to a poll's creation) passed. The remaining lines relate to the lack of an exception being thrown in the method `getPoll` of `PollSystem` when trying to access a poll that doesn't exist. That this should happen is obvious to Bob, a seasoned programmer. However, suppose he doesn't have a clue of the exact point in the code where the error lies. He only got from EasyAccept the information that a `NullPointerException` was thrown somewhere. Couldn't EasyAccept be a little more specific? Certainly, and that's what we will introduce in the next subsection.

Debugging with EasyAccept

EasyAccept has a built-in command called `stackTrace` for debugging. The programmer can place it in the script before any command that results in an exception, and when EasyAccept is run, it prints out the full stack trace of the exception or error that occurred. Knowing this, Bob puts a `stackTrace` command at the beginning of line 19, as below:

```
19 stackTrace expectError "Poll does not exist" getPollName poll=abc
20 expectError "Poll does not exist" getPollAnswers poll=abc
```

When he runs EasyAccept, instead of receiving a simple "`NullPointerException` was thrown" message, he gets the following message (abridged here, because the full Java stack trace is really long-winded):

```
Test file poll-us1.txt: 2 errors:
Line 19, file poll-us1.txt:
easyaccept.EasyAcceptException: Line 19, file poll-us1.txt: Expected the
error message <Poll does not exist>, but the error message was <(no
message: exception = java.lang.NullPointerException)>
...
Caused by: java.lang.NullPointerException
      at poll.PollSystem.getPollName(PollSystem.java:30)
      at poll.PollFacade.getPollName(PollFacade.java:16)
...
Command producing error: <stackTrace expectError "Poll does not exist"
getPollName poll=abc>
Line 20, file poll-us1.txt: Expected the error message <Poll does not
exist>, but the error message was <(no message: exception =
java.lang.NullPointerException)>
```

```
Command producing error: <expectError "Poll does not exist" getPollAnswers
poll=abc>
```

Reading the stack trace, Bob can spot that the `NullPointerException` was thrown in the method `getPollName` from `PollSystem` at code line 30, which turns out to be

```
return getPoll(poll).getName();
```

That is, a `NullPointerException` is being thrown because the method `getPoll` is returning `null` and the code is trying to call a method from it. Taking a look at `getPoll`'s code, Bob immediately sees the problem. Bob now changes the code of `getPoll` to throw an exception when the poll requested is not found:

```
public Poll getPoll(String pollID) throws NonexistentPollException {
      Poll p = polls.get(pollID);
      if (p == null) throw
            new NonexistentPollException("Poll does not exist");
      return p;
}
```

Bob must also create the `NonexistentPollException` class extending `PollException`:

```
package poll;

public class NonexistentPollException extends PollException {
      public NonexistentPollException(String message) {
            super(message);
      }
}
```

Additionally, he updates the Façade with `throws` clauses in the methods that use `getPoll`:

```
public String getPollName(String poll) throws NonexistentPollException

public String getPollAnswers(String poll) throws NonexistentPollException
```

That not only solves the problem of line 19, but also that of line 20, which stemmed from the same cause. If he runs EasyAccept again, he gets the message:

```
Test file poll-us1.txt: 1 errors:
Line 19, file poll-us1.txt:  (No exception thrown.)
Command producing error: <stackTrace expectError "Poll does not exist"
getPollName poll=abc>
```

"Oops, I forgot to remove the `stackTrace` command from line 19", he ponders. That's exactly what happened. He left a `stackTrace` at line 19, but no exception was thrown. After he removes the `stackTrace` statement from line 19, Bob runs EasyAccept once again and verifies that all tests now pass:

```
Test file poll-us1.txt: 12 tests OK
```

Bob treats himself to a nice cup of coffee.

Testing for Differences

The tests for the first user story are not complete, though. "What happens", Bob thinks, "when I try to create a poll with an already existing name?" He is not exactly sure if this should be allowed or result in an error, so he gives Alice a call. She answers that this is not an error, because two polls could have the same name, even the same answer choices, but still be different (for example, she could want a given poll to be repeated every week). Bob promptly adds the clarification to the test script in the form of a comment (this is the **Commentor** pattern, as detailed in section 2) and includes tests for this case in *poll-us1.txt*. Below are the lines that were added.

```
21
22 # polls may have all attributes equal and still be distinct polls;
23 # for the tests below, even though they have the same name and answer
24 # choices, polls 1 and 3 must be different
25 poll3=createPoll name="Do you like apples?" answers={yes,no}
26 expectDifferent ${poll1} echo ${poll3}
```

Let us explain two more EasyAccept built-in commands used by Bob in the preceding tests. The command `expectDifferent` has a syntax similar to that of the `expect` command, but the test writer specifies a value *not* expected instead. The test passes if any value *except* for the one specified is returned by the business logic command that follows. In the case of line 25, the command that follows is the built-in command `echo`. It does what the name implies – it simply returns the concatenation of its arguments. It has been used there because of `expectDifferent`'s syntax: it requires a command after the value not expected.

In the script, Bob is using `expectDifferent` in line 25 to make sure the ids `poll1` and `poll3` are different and thus represent two different business objects. Bob had already implemented this when he devised the internal key mechanism. He runs EasyAccept again and gets the following result:

```
Test file poll-us1.txt: 14 tests OK
```

Completing User Story 1

Bob considers one more special case. "Can there be repeated answers in a poll?" – He asks Alice, and she replies – "No, there shouldn't be. It doesn't make sense. But couldn't the program simply ignore repeated answers and act as though the user typed them inadvertently? ". "That could be done", says Bob. He adds some more lines to *poll-us1.txt*, which we outline below.

```
27
28 # repeat answers within a poll should be ignored
29 poll4=createPoll name="Do you like me?" answers={yes,yes,no}
30 expect "{yes,no}" getPollAnswers poll=${poll4}
```

In line 29, Bob creates a poll with repeated answers but, as Alice requested, no error should be thrown. The program should creat the poll and throw repeated answers away. Line 30 checks that the poll program is actually eliminating the repeated answer.

Now this hasn't been implemented yet. When Bob runs EasyAccept, it complains of line 30.

```
Test file poll-us1.txt: 1 errors:
Line 30, file poll-us1.txt: Expected <{yes,no}>, but was <{yes,yes,no}>
Command producing error: <expect {yes,no} getPollAnswers poll=89f04fb7-
72c5-4940-8696-827bb3ee7492>
```

Just as a side note, observe the huge id that was returned by `createPoll` and assigned to the variable `poll4`. That's the sort of unique identifier created by Java's `util.UUID` class.

To make this new test work, Bob makes the simplest change he can come up with (of course, without cheating). He decides to change the `answers` data structure, which is stored in the `Poll` class, from a `List` to a `Set`. That way, element repetition can be handled automatically when an answer is added to the data structure. In Java, both interfaces, `List` and `Set`, are derived from the generic interface `Collection`, which is referenced throughout the entire code and passed as argument to and from `PollFacade` and `PollSystem`. The actual implementation is held in the class `Poll`. That's the only class he needs to change. The new code for the `Poll` class is depicted below, and the places where changes were made are in bold.

```
package poll;

import java.util.Collection;
import java.util.Set;
import java.util.LinkedHashSet;
import java.util.UUID;

public class Poll {
      private String name;
      private Set<String> answers;
      private String id;

      Poll( String name, Collection<String> answers ) {
            this.name = name;
            this.answers = new LinkedHashSet<String>( answers );
            /* get a randomized unique universal identification for id */
            this.id = UUID.randomUUID().toString();
      }

      String getName() { return name; }

      Set getAnswers() { return answers; }

      String getID() { return id; }
}
```

The implementation of the answers data structure changed from an `ArrayList` to a `LinkedHashSet`. After a new run of EasyAccept, he gets good news:

```
Test file poll-us1.txt: 16 tests OK
```

Bob can't think of any more tests to include in *poll-us1.txt*. At this point, Bob grins and gets more coffee, since he made the first user story of the poll program fully functional. The poll program handles error situations when the user tries to create invalid polls, and even handles repeated answers. It creates polls successfully and stores their information correctly in a single program session. However, Bob hasn't implemented a persistence mechanism yet;

we'll talk about persistence testing when we present the pattern **Persistence Tester** in section 2.

He shows testing results to Alice, who is also excited at the news.

At this point, Bob might do some *refactoring*, if needed, before moving on to implement the second user story. Two rules are followed when refactoring: 1) all tests must pass before refactoring; 2) all tests must pass after refactoring. That is, Bob only changes the code from a working state to another working state. At any time, he might also have ideas for new tests, as the process of test creation is not always completed in one sitting. Alice herself can also create tests if she wants, and they should be swiftly integrated with the test base.

Whenever ideas for new occur to Bob, if he has doubts on what should happen, he must ask Alice for clarification (she herself might not know what she really wants!!). She either approves or disapproves the test (and this step is of utmost importance, or else Bob might be introducing requirement bugs in the tests and consequently in the program). Bob and Alice must bear in mind that the tests must always be improving – this is part of the *test maintenance* activity in acceptance test-driven development. The better the tests, the more confident everyone becomes that the program does what it should correctly.

Concluding remarks before user story 2

This illustration of ATDD techniques to implement user story 1 served the purpose of introducing the methodology and EasyAccept's syntax in a stepwise manner. However, there are some differences from what Bob would actually do in practice. First of all, most of the test writing would be done upfront, and not in parallel with the coding of user story 1. From the beginning, Bob already had a good idea of which test conditions, errors, etc. he would need to include in the test script; furthermore, he knows test creation patterns well, so most of *poll-us1.txt* would be complete even before the first line of code was written.

A second remark: although this wasn't demonstrated in this user story, errors in the test script occur frequently. They are mostly typos, however, and are immediately identified when one runs EasyAccept. More serious acceptance test bugs (which are requirements bugs) can be prevented if the test writer always gets feedback from the client and/or the client reviews the test scripts frequently. Programmers must also always ask the client for clarifications, and not introduce tests on his own.

Implementing the second user story

Bob now turns his attention to the second user story. The sequence of steps followed is the same as for the first user story. Before starting to code, Bob needs acceptance tests. Based on the user story description, new commands are added to the script language. Such commands are combined with existing commands (created for previous tests) and EasyAccept built-in commands to create the set of tests. Alice reviews the tests, suggesting modifications and clarifying issues, and then Bob starts to code driven by the revised script.

User story 2 – Let the user cast a vote for a poll. He chooses a poll and one of the possible answers.

This turns out to be a straightforward user story. Bob only needs a *doer* command to vote a given answer for a given poll, and a *getter* to check if the voting action actually casts votes correctly. The commands defined are:

```
vote poll=<String> answer=<String>
getNumberOfVotes poll=<String> answer=<String>
```

In addition, he devises a *preparer* command called `clearSystemData` to be used at the beginning of the test script. This command is necessary to make the tests for user story 2 independent from other tests. For example, polls created in other test scripts, like *poll-us1.txt*, might introduce noise in the test script for user story 2 and be a potential source of errors.

The test script Bob produces consists in issuing some votes for a given poll, and checking if the number of votes for the answers increases accordingly (he is using the **Test Flow** pattern). Furthermore, he uses the test creation pattern **Command Errors** on the commands `vote` and `getNumberOfVotes` to assure that bad voting conditions result in errors. He saves the script in file *poll-us2.txt*. As we said, Bob usually writes a test script in an upfront manner, so we reproduce it in its entirety below:

```
1  # user story 2 - Let the user cast a vote for a poll.
2  # He chooses a poll and one of the possible answers.
3
4  clearSystemData
5
6  poll1=createPoll name="Do you like apples?" answers={yes,no}
7  expect 0 getNumberOfVotes poll=${poll1} answer=yes
8  expect 0 getNumberOfVotes poll=${poll1} answer=no
9
10 vote poll=${poll1} answer=yes
11
12 expect 1 getNumberOfVotes poll=${poll1} answer=yes
13 expect 0 getNumberOfVotes poll=${poll1} answer=no
14
15 vote poll=${poll1} answer=yes
16 vote poll=${poll1} answer=no
17
18 expect 2 getNumberOfVotes poll=${poll1} answer=yes
19 expect 1 getNumberOfVotes poll=${poll1} answer=no
20
21 # using the pattern Command Errors with the commands
22 # vote and getNumberOfVotes
23
24 expectError "Poll does not exist" vote poll=abc answer=yes
25 expectError "There's no such answer for this poll" \
26    vote poll=${poll1} answer=maybe
27 expectError "Poll does not exist" getNumberOfVotes poll=abc answer=yes
28 expectError "There's no such answer for this poll" \
29    getNumberOfVotes poll=${poll1} answer=maybe
```

Alice reviews the tests and approves them. Bob then implements the code that will make the tests pass. He runs EasyAccept, now including *poll-us2.txt* as an argument (note that *poll-us1.txt* must also be part of the EasyAccept run, to make sure new code doesn't introduce bugs to the code of the first user story). He runs EasyAccept as follows:

```
java easyaccept.EasyAccept poll.PollFacade poll-us1.txt poll-us2.txt
```

Alternatively, Bob could have used a mask like in

```
java easyaccept.EasyAccept poll.PollFacade poll-us*.txt
```

Alternatively, he can move the acceptance tests to a separate folder and call EasyAccept with a reference to the folder. For example, if he put the tests in a folder called *acctests*, the call would be

```
java easyaccept.EasyAccept poll.PollFacade acctests
```

EasyAccept reports errors for unrecognized commands `clearSystemData`, `getNumberOfVotes` or `vote` (they are not included in `PollFacade` yet).

```
Test file poll-us1.txt: 16 tests OK
Test file poll-us2.txt: 15 errors:
Unknown command: clearSystemData
Command producing error: <clearSystemData>
Line 6, file poll-us2.txt: Unexpected error: Unknown command:
getNumberOfVotes poll=d8c07e1d-2c81-45d7-8300-735d8a68d89d answer=yes
Command producing error: <expect 0 getNumberOfVotes poll=d8c07e1d-2c81-
45d7-8300-735d8a68d89d answer=yes>
...
```

Bob then adds those commands as methods in `PollFacade`, whose code simply directs the corresponding requests to `PollSystem`, as below:

```
	public void clearSystemData() {
		pollSystem.clearSystemData();
	}

	public void vote(String poll, String answer) {
		pollSystem.vote(poll, answer);
	}

	public int getNumberOfVotes(String poll, String answer) {
		return pollSystem.getNumberOfVotes(poll, answer);
	}
```

In `PollSystem`, he keeps following the rule of thumb in test-driven development: use the simplest code to make tests pass. It simply does nothing until a test breaks.

```
	public void clearSystemData() { polls = new HashMap<String, Poll>();
}
```

As for the remaining command, Bob makes the method `vote` in `PollSystem` do nothing, and makes `getNumberOfVotes` always return 0. With that simple implementation, he reduces the errors from 15 down to 7 when running EasyAccept:

```
Test file poll-us1.txt: 16 tests OK
Test file poll-us2.txt: 7 errors:
Line 12, file poll-us2.txt: Expected <1>, but was <0>
```

```
Command producing error: <expect 1 getNumberOfVotes poll=61ac33ea-899b-
41c6-b408-beadf937ea0b answer=yes>
...
```

That is, the first error occurs when a vote has been cast but was unaccounted for. Bob must now implement the actual voting mechanism, which will as a consequence trigger a cascade of passing tests in the next run.

Votes are cast within a poll to a given answer. Bob needs to change the `Poll` class to deal with answers, which must now be treated not as mere strings but as objects with responsibilities (they store votes, they are voted for, etc.). Polls must maintain a set of answers, process their search and control their access. The final `Poll` class is depicted below. Modifications to the class are in bold.

```
package poll;

import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.LinkedHashSet;
import java.util.UUID;

public class Poll {

	private String name;
	private Set<Answer> answers;
	private Map<String, Answer> answerMap;
	private String id;

	Poll( String name, Collection<String> answers ) {
		this.name = name;
		this.answers = new LinkedHashSet<Answer>();
		this.answerMap = new HashMap<String, Answer>();
		/* get a randomized unique universal identification for id */
		this.id = UUID.randomUUID().toString();

		/* initializes individual answers and adds them to the Map */
		Set<String> uniqueAnswers = new LinkedHashSet<String>(answers);

		Iterator<String> it = uniqueAnswers.iterator();
		while(it.hasNext()) {
			String oneAnswer = it.next();
			Answer ans = new Answer(oneAnswer);
			this.answers.add(ans);
			answerMap.put(oneAnswer, ans);
		}
	}

	String getName() { return name; }

	Set<Answer> getAnswers() { return answers; }

	String getID() { return id; }

	Answer getAnswer(String answer) throws NonexistentAnswerException {
		Answer ans = answerMap.get(answer);
		if(ans == null) throw new NonexistentAnswerException(
```

```
                                "There's no such answer for this poll");
            return ans;
      }

      int getVotes(String answer) throws NonexistentAnswerException {
            return getAnswer(answer).getNumberOfVotes();
      }

      void vote(String answer) throws NonexistentAnswerException {
            vote(getAnswer(answer));
      }

      void vote(Answer ans) { ans.vote(); }

}
```

The basic structural modification made to the `Poll` class is that the set of answers is now not simply a `Set` of `Strings`, but a `Set` of `Answer` objects. Naturally, the code won't compile because there is no such `Answer` class (but Bob will do this in a minute). To ease access to a specific answer from the set given its name, Bob also creates an answer `Map`. In addition to the required methods `vote` and `getVotes`, which `Poll` forwards to the specific `Answer`, Bob created a method `getAnswer`, in which an exception is thrown should an answer not exist in the set.

Bob then codes the `Answer` class. An answer has a name and a number of associated votes. It relates to a poll by composition (a poll *has* several answers) and its contents are thus accessed within a poll. The script command `vote` will ultimately trigger the `vote` method in the specific `Answer` object related to the `Poll` object referred to in the script. The same applies to getting the number of votes for a given answer.

```
package poll;

public class Answer {

      private String name;
      private int numberOfVotes;

      Answer(String name) {
            this.name = name;
            this.numberOfVotes = 0;
      }

      String getName() { return name; }

      int getNumberOfVotes() { return numberOfVotes; }

      public String toString() { return getName(); }

      void vote() { numberOfVotes++; }

}
```

In order for the new code to work, `PollSystem` must also be updated. The methods `vote` and `getNumberOfVotes`, instead of doing nothing and returning 0, are updated to include simple, forward-to-the-expert-class code:

```
int getNumberOfVotes(String poll, String answer)
      throws NonexistentPollException, NonexistentAnswerException {
      return getPoll(poll).getVotes(answer);
}

void vote(String poll, String answer)
      throws NonexistentPollException, NonexistentAnswerException {
      getPoll(poll).vote(answer);
}
```

Bob finishes preparing the code, saves everything and runs EasyAccept.

```
Test file poll-us1.txt: 16 tests OK
Test file poll-us2.txt: 15 tests OK
```

All tests from both test files are working. He tells Alice the good news and sets off to implement user story 3, after a cup of coffee, of course.

Implementing the third user story

User story 3 – Generate a report with the results of a poll, including the total number of votes for each option and the corresponding percentage.

Bob reads the description of user story 3. A full report, including final layout as presented to the user of the program, will not be tackled by Bob right now. Alice still has not decided the report layout or the medium she wants the data to be presented in (plain text, graphs, pdf files, etc.). If she said, for example, that she wanted the report as a text with a given format, Bob could use the pattern **Template Tester** to compare the program's output with a template of the expected report text. If you want to look now how Bob could test this, refer to the discussion of **Template Tester** on section 2.

However, this user story first needs to test the program's business logic. The beautifully presented report would be worthless if the output included wrong data, like totals and percentages. "Voting totals can be caught with `getNumberOfVotes` for each answer, but for the percentages I'll need a new command", thinks Bob. He then comes up with the following command,

```
getPercentageOfVotes poll=<String> answer=<String>
```

, which returns a double precision number. Bob thinks about the tests he'll need. He's got a new command, so he throws in some **Command Errors** tests for `getPercentageOfVotes`. To check if totals and percentages are being calculated correctly, he devises a simple scenario with a two answers poll (yes or no) and `expect` commands in various points between a series of votes for different percentage values. The script he came up with is depicted below (he names it, naturally, *poll-us3.txt*):

```
1 # user story 3 - Generate a report with the results of a poll,
2 # including the total number of votes for each option and
3 # the corresponding percentage.
4
5 poll1=createPoll name="Do you like apples?" answers={yes,no}
6 expect 0 getNumberOfVotes poll=${poll1} answer=yes
```

```
 7 expect 0 getNumberOfVotes poll=${poll1} answer=no
 8 expectWithin 0.000001 0.0 getPercentageOfVotes poll=${poll1} answer=yes
 9 expectWithin 0.000001 0.0 getPercentageOfVotes poll=${poll1} answer=no
10
11 vote poll=${poll1} answer=yes
12 expect 1 getNumberOfVotes poll=${poll1} answer=yes
13 expect 0 getNumberOfVotes poll=${poll1} answer=no
14 expectWithin 0.000001 100.0 getPercentageOfVotes poll=${poll1}answer=yes
15 expectWithin 0.000001 0.0 getPercentageOfVotes poll=${poll1} answer=no
16
17 vote poll=${poll1} answer=no
18 expect 1 getNumberOfVotes poll=${poll1} answer=yes
19 expect 1 getNumberOfVotes poll=${poll1} answer=no
20 expectWithin 0.000001 50.0 getPercentageOfVotes poll=${poll1} answer=yes
21 expectWithin 0.000001 50.0 getPercentageOfVotes poll=${poll1} answer=no
22
23 vote poll=${poll1} answer=yes
24 expect 2 getNumberOfVotes poll=${poll1} answer=yes
25 expect 1 getNumberOfVotes poll=${poll1} answer=no
26 expectWithin 0.1 66.7 getPercentageOfVotes poll=${poll1} answer=yes
27 expectWithin 0.1 33.3 getPercentageOfVotes poll=${poll1} answer=no
28
29 vote poll=${poll1} answer=yes
30 vote poll=${poll1} answer=yes
31 vote poll=${poll1} answer=yes
32 vote poll=${poll1} answer=yes
33 vote poll=${poll1} answer=yes
34 expect 7 getNumberOfVotes poll=${poll1} answer=yes
35 expect 1 getNumberOfVotes poll=${poll1} answer=no
36 expectWithin 0.1 87.5 getPercentageOfVotes poll=${poll1} answer=yes
37 expectWithin 0.1 12.5 getPercentageOfVotes poll=${poll1} answer=no
38
39 # using the pattern CommandErrors with the command getPercentageOfVotes
40 expectError "Poll does not exist" getPercentageOfVotes \
41    poll=abc answer=yes
42 expectError "There's no such answer for this poll" \
43    getPercentageOfVotes poll=${poll1} answer=maybe
```

In this script, Bob uses yet another EasyAccept built-in command: `expectWithin`. This command is used to expect a floating-point number, such as a percentage of votes, within a given precision. The syntax is `expectWithin <precision> <expectedValue> <businessLogicCommand>`.

After Alice reviews the script, Bob starts to implement the new functionality. EasyAccept doesn't understand `getPercentageOfVotes`, so errors are reported at line 8:

```
Test file poll-us1.txt: 16 tests OK
Test file poll-us2.txt: 15 tests OK
Test file poll-us3.txt: 12 errors:
Line 8, file poll-us3.txt: Unexpected error: Unknown command:
getPercentageOfVotes poll=15e41905-8aaa-4573-8363-b26b3b53e0c8 answer=yes
Command producing error: <expect 0,0 getPercentageOfVotes poll=15e41905-
8aaa-4573-8363-b26b3b53e0c8 answer=yes>
...
```

He starts by implementing `getPercentageOfVotes` in `PollSystem`:

```
	double getPercentageOfVotes(String poll, String answer)
	throws NonexistentPollException, NonexistentAnswerException {
		Poll p = getPoll(poll);
		if( p.getTotalVotes() != 0 ) {
			return (double) p.getVotes(answer) * 100.0
			/ (double) p.getTotalVotes();
		} else {
			return 0.0;
		}
	}
```

To get the percentage of votes for a given answer, Bob needs the total votes cast in a given poll. That's why there's a call to the (still nonexistent) method `getTotalVotes` in the `Poll` class. This method sums votes for all answers:

```
public class Poll {
	...

	int getTotalVotes() {
		int totalVotes = 0;
		Iterator<Answer> it = answers.iterator();
		while(it.hasNext()) {
			Answer ans = it.next();
			totalVotes += ans.getTotalVotes();
		}

	}

}
```

Finally, he needs to add the corresponding method to `PollFacade`:

```
	public double getPercentageOfVotes(String poll, String answer)
			throws NonexistentPollException,
NonexistentAnswerException {
		return pollSystem.getPercentageOfVotes(poll, answer);
	}
```

In order to come up with the preceding working code, Bob had to run EasyAccept a few times to resolve bugs in the process. For example, his first implementation threw a divide-by-zero exception at line 8 (a line where no error was expected). The problem was in the method `getPercentageOfVotes` in `PollSystem`. When no votes had been cast for any answer, the total votes for the poll would be 0, and that value was being used as a denominator in the calculation of the percentages without being previously treated. However, no matter how long he takes to code and how many mistakes he makes in the process, he is sure to have a correct code in the end, given that the tests are correct.

The final EasyAccept run confirms that all tests pass for the three user stories:

```
Test file poll-us1.txt: 16 tests OK
Test file poll-us2.txt: 15 tests OK
Test file poll-us3.txt: 32 tests OK
```

Bob now has fully working software to show Alice. Of course, Alice still needs a user interface to use with the poll program, but Bob can do that in little time with visual tools hooking the graphical components to the Façade methods (that's one of the reasons why one would code a Façade to a system, anyway). Or, if she is satisfied with a simple command-line interface, it would be even quicker: Bob could use a single class with a main method as the command-line interface. As Alice stated the three user stories would be the minimum useful functionality she would need, she could be using the program right away if she wanted, after this first programming session.

Concluding words

This first example illustrated how ATDD works and how you can easily apply the methodology with EasyAccept. In the next chapter, ATDD is reconsidered more completely. We explain the methodology in a software development process' point of view, depicting activities performed, actors involved and practices used. That closes this first section on the ATDD rationale.

4

Clients	Developers	Both
- Write user stories	- Code user stories to make acceptance tests pass	- Define a script language
- Prioritize user stories	- Refactor existing code to improve quality	- Write acceptance tests (though developers generally do this)
- Review acceptance tests (helped by developers)	- Help find errors in the acceptance tests	- Help identify new acceptance tests

Table 4.1 – Project Responsibilities

ATDD is composed of three classes of activities: planning activities, activities that occur during iterations, and maintenance activities. Most planning activities take place at the beginning of the project (though iteration planning occurs at the beginning of iterations); a series of iterations follow, during each of which a series of activities occur. Maintenance activities occur during the entire lifecycle of the process. Figure 4.1 depicts an outline of the activities involved in ATDD. The remainder of this chapter explains and details them.

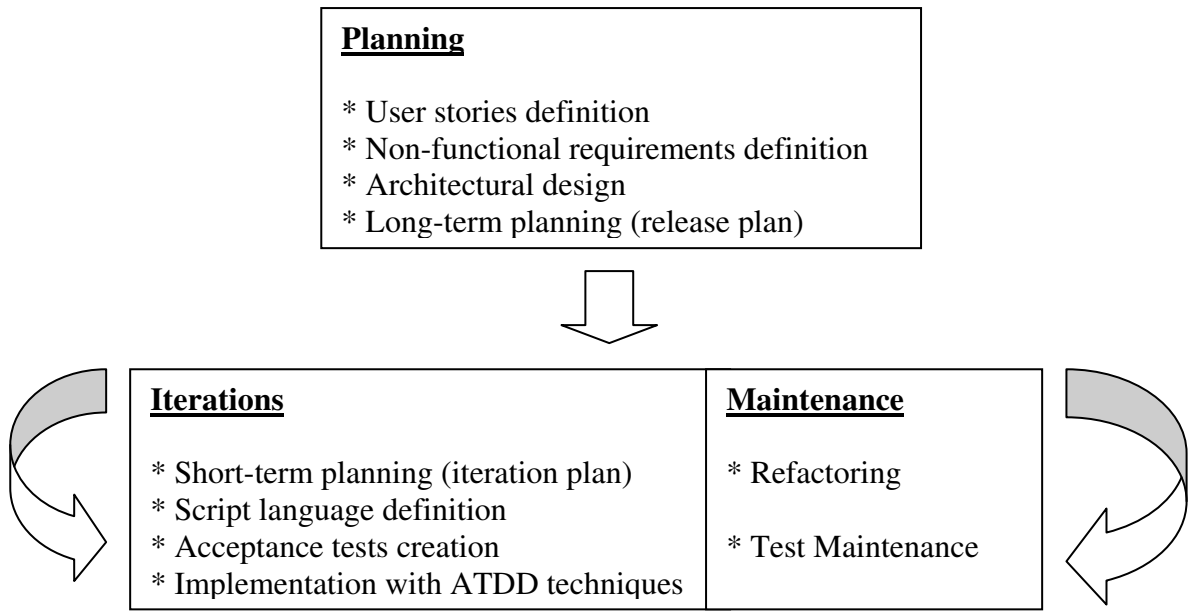


Figure 4.1 – ATDD Process

Project Planning

The process begins with a few meetings and talks between the client and one or more developers (or analysts, if your project has clear job distinctions). Not many people should be involved in these talks. Ideally, the client and at most a couple of developers should do the job. Too many people don't help overall understanding and hinder the flow of discussion. The objectives of these initial meetings are to grow a common overview of the software that will be created and to have a *user story list* written and analyzed, as well as a list of *non-functional requirements*. However, the final prioritized user story list will depend on some extent on a

loose *architectural design* being done by the developers because they need to provide the clients with estimates of time needed to complete each user story (and thinking out the architecture of the program helps them come up with better estimates). Subsequently, a *release plan* can be formulated that will dictate how and when the software will be created. We have grouped all these activities into a class called project planning, which occurs prior to software implementation and includes part of what is commonly called "analysis" in other processes. Planning should not last more than a few (say, two or three) weeks[5], even if it is not yet clear for the client exactly what he wants the software to do. As an agile process, ATDD handles change with ease and the client can be allowed to make up his mind later on to some extent.

User story list definition

At first, the planning meetings serve the purpose of discussion and understanding. A glossary of relevant terms needed to understand the area of application and a conceptual model depicting important entities may also be built during these meetings, to aid understanding. In the course of the planning period, before or during the meetings, the client writes a set of *user stories*. User stories are short descriptions in direct sentences, no more than one or two paragraphs, which summarize a feature that must be implemented.

In addition to these descriptions, the client must also *prioritize* the user stories, ranking higher those features that have more *business value*. This step in needed to define the order in which user stories will be developed. Along with their business value, the client decides the order of user stories based on the developer's feedback in the form of *time estimates* and user story *dependencies* after some sessions of *architectural design* that occur in tandem with the user story list definition.

Time estimates give the client additional information to prioritize the user stories (if a certain user story takes longer to complete, the client may prefer to have a series of shorter user stories implemented first). The estimates may be hard to produce, as they depend on developers' prior experience, but even rough estimates help achieve a better release plan. Dependencies between user stories also affect the order with which they will be implemented.

At any time during the software development lifecycle, the client may add, remove or modify user stories, as long as he keeps the user stories ranked and is warned that user stories are implemented one at a time, respecting the order of priority. Modification of a user story that is already implemented, however, is considered to be a new user story. Thus, it requires a new time estimate and prioritization, as the time needed to realize the modification can differ a lot from the original time required to implement the story from scratch.

Activity breakdown
Who does it: the client, helped by a few developers (analysts);

When it begins: as soon as possible, after the first meetings; user stories can be added / deleted / modified during the course of the project as desired;

[5] Of course, larger projects may require more time since coming up with a viable architecture is not always straightforward.

When it ends: as soon as enough user stories have been written and prioritized to allow a release plan to be created.

Architectural design

The activity in which developers make strategic high-level technical decisions to define the internal structure of software is called *architectural design*, or high-level design. Based on the client's needs (as described in the user stories), developers come up with an *architecture*, a description of the program's higher-level modules and components and the relationships between them. The architecture is often presented in the form of a variety of diagrams, expressing different views of the system characteristics, and/or in the form of a text description which includes key design decisions and justifications for them.

Key design decisions often include the resolution of tradeoffs and choices between non-overlapping options, each with advantages and drawbacks. For example, when creating an architecture, developers must decide on questions like: Will the system be centralized or distributed? How will we modularize the system? Are we buying or developing from scratch a particular module? How will we interface between the modules? What persistence data model will we use? How will we integrate the legacy? These are issues that must be decided early on in software development, because they are costly to change after implementation has begun. Even though ATDD is an agile, change-friendly process, we believe architectural design is an essential activity to be done in an upfront manner.

Activity breakdown
Who does it: a few of the developers (maybe specialized *architects*);

When it begins: after the first user stories are written by the client;

When it ends: a rough architecture must be produced before the user story list is prioritized because developers need to provide the client with time estimates and dependencies; a refined architecture must ideally be decided upon before implementation begins.

Non-functional requirement list definition

Non-functional requirements are conditions raised by the client that the software must meet but that do not correspond to a *function* of the software, i.e., a user interaction that can be described in a user story. Non-functional requirements are generally constraints or restrictions on response time, performance, throughput, needs to support legacy or to use specific software or hardware equipment, software development cost, legal restrictions, etc.

Although non-functional requirements are sometimes hard to meet, one of the main reasons for which being the inability to write automated tests for such requirements, the good news is that some of them can be tested. Performance, throughput and response time restrictions can sometimes be automatically checked with acceptance tests.

Although they can be associated with a particular user story, non-functional requirements often refer to the whole software. This way, tests for such requirements should ideally be written *before* any user story is implemented, during the first iteration, so that such tests can also drive development.

During the weeks of planning, clients must create a list of non-functional requirements. The part that is automatable with acceptance tests is identified and separated by the developer, and the rest is kept for reference and discussed with management for the appropriate project preparations.

Activity breakdown
Who does it: the client (non-technical requirements), architects (technical requirements);

When it begins: as soon as possible, after the first meetings; some non-functional requirements must be agreed upon early on and may not (easily) be changed during development – they are costly changes; non-costly changes may be requested even during implementation;

When it ends: when the client is satisfied with the list, before design begins; as stated above, some non-functional requirements may be changed during development.

Creation of a release plan (Long-term planning)

ATDD is an iterative software development process based on *releases* and *iterations*. A release is a set of user stories that together define a portion of the program that is useful to the client (i.e., it has *business value*). The program being developed encompasses several releases, so that the final program is ultimately delivered to the client in an incremental fashion, as each subsequent release adds more and more functionality and business value to the program.

An iteration is a timeframe measured in weeks (typically a few weeks, no more than two or three) during which a certain number of user stories is implemented. During an iteration, a whole release can be implemented, or a release can encompass several iterations (iterations are blocks of fixed time, whereas releases are sets of user stories). That is the case, for example, for the first release. It usually takes several iterations to complete, because during the first iterations you are starting the program from scratch and normally can't release anything of value.

In possession of a prioritized user story list, clients and developers can plan how and in what order the user stories will be implemented. Such planning is done in two levels. The planning activity done at the beginning of the project is called *long-term planning*, and produces a *release plan* as a result; this plan contains a description of each release.

Several refined *short-term planning* activities are done later to refine and detail the time-blocked activities that occur during an iteration. Short-term planning occurs at the beginning of each iteration and produces an *iteration plan* as a result.

The rationale that supports this separation of planning activities in two levels is that it is not feasible to conceive full upfront detailed planning of activities likely to work out over time, because the ordering of user stories depends on estimates of time that have a huge variability on the long-term.

Activity breakdown
Who does it: the client and developers;

When it begins: after an initial user story list has been created;

When it ends: when the client is completely satisfied with the release plan.

Iterations

As soon as the release plan is ready, the first iteration decided upon can be set on course.

Important disclaimer: as we said in the beginning of the chapter, we are not indicating some common important activities found in software development. Just as a reminder and a suggestion, however, we find it important to state at this point that, before the first iteration begins, it is advisable that developers do some more thinking on aspects of software development that are not change-friendly. We already mentioned architectural design, which must be dealt with early on not only because it is costly, but also because developers need an architecture to produce good time estimates. Another costly activity is database modeling, if a database will be required; the reason is that changing a database over time is usually a very time-consuming activity. Time invested in these activities beforehand may be beneficial to the project.

As we said when discussing the release plan, the first release generally takes several iterations. Subsequent iterations may or may not deliver releases (releases can be partitioned over several iterations, depending on user story dependencies and time required for implementation).

At the beginning of each iteration, developers create an *iteration plan* during short-term planning. The iteration plan includes details on how the iteration will be executed: it is basically a breakdown of micro-activities needed to implement each user story along with the time needed for each one of them (typically a few hours). The iteration itself is time-blocked (spans a fixed amount of time).

Each iteration encompasses the implementation of some user stories from the list (a complete release or part of a release). An iteration follows with the *definition of a script language* that will be used to *create acceptance tests* for the user stories chosen; then, the code that will make the acceptance tests pass is *implemented* with ATDD techniques; the iteration ends when all the acceptance tests for such user stories have passed. Acceptance tests for an iteration must also include tests for non-functional requirements, such as performance. That way, such requirements can be automatically attended as development progresses. Also during an iteration, *maintenance* activities that occur throughout the software development life cycle take place.

In each iteration, the following steps are performed:

Defining a script language

Since we use in this text EasyAccept as the acceptance-testing tool, we refer here to a *script language* – a set of script commands that one writes the acceptance tests with. However, if you use any other tool to automate your acceptance tests that requires another format, you can

replace “script language” in the title of this subsection with the basic elements that compose the tests. For example, if you use FiT, you will need to define the *table templates* (table types, column labels and disposition) that will be used to create the tests. No matter which tool you use, this definition of the test *vocabulary*, so to speak, is always necessary and must not be done in an off-the-cuff manner – even though the vocabulary definition is generally a simple, quick and straightforward activity – because ill-thought commands can make tests cumbersome and thus hinder client review and test maintenance.

In what regards the ATDD process, we need to refine the script language at the beginning of each iteration, as new features that will be implemented may require new commands to express the corresponding acceptance tests. However, this activity tends to grow ever shorter as iterations are completed, because commands already defined for previous user stories are frequently reused.

Activity breakdown
Who does it: clients and developers (analysts/testers);

When it begins: at the beginning of the iteration, along with the creation of acceptance tests;

When it ends: whenever all acceptance tests for the iteration have been written; moreover, if new commands must be added/modified, this can be done after implementation has started.

Creating Acceptance Tests

In an ideal world, clients are resourceful beings who are always available to write the acceptance tests themselves. People have created EasyAccept and other acceptance testing tools to materialize this idea. However, it most often is the case that the client is not willing to write acceptance tests himself. It’s not that clients are not available in the usual agile processes sense – they are around, clear up doubts, hint on this or that matter; the problem is that writing acceptance tests is a somewhat burdensome task they don’t want to embrace (at least, that’s the sort of situation we have most often come across).

Clients not writing the acceptance tests themselves, however, is not really *that* essential for the process to work, anyway, as long as they basically help with two things: 1. provide developers with the business rules for a given user story; 2. review the acceptance tests created by the developers;

The actual creation of the tests can then done by a developer specialized in writing acceptance tests (i.e., anyone who has read section 2, on acceptance test creation patterns), based on business rules gathered during talks with the client. *The client must review all acceptance tests*. He must read the tests and judge if they really reflect what he wants.

When does this step finish? There is no rule of thumb, but generally, the test writing effort should be employed to produce tests as thoroughly as possible before implementation (using some of the acceptance test creation patterns of section 2 can help enforce more complete tests in a systematic way). Naturally, with implementation under way, new test cases are created as new business rules are captured or tricky scenarios are brought to attention.

It is general knowledge that tests are inherently incomplete. Tests can prove the existence of a defect, but not their absence. This is especially true for testing techniques based on examples

(input values entered, expected outputs checked), just like the ones tools for acceptance testing are based on. At any time, new test cases and scenarios can be found for any preceding or ongoing user story under implementation (and this is dealt with in a separate continuous activity called acceptance test maintenance – see subsection below). The final acceptance of the software is therefore subjective to some extent, depending on the client's satisfaction with the acceptance tests created. Moreover, even when the software is delivered, as it is used, new user stories and tests are added in future versions. That is why the test creation effort (and implementation to make tests pass) actually never *ends* during the project's lifecycle.

Activity breakdown
Who does it: most often, developers (analysts/testers), and the tests are reviewed by clients; new tests can be created during the software lifecycle by any developer (see test maintenance below);

When it begins: at the beginning of an iteration, along with the definition of the script language;

When it ends: every user story must have at least one acceptance test; the main effort is done perhaps during the current iteration, but the test creation continues until the end of the development lifecycle; at any time new acceptance tests for user stories already implemented may be added to the test base (see test maintenance, below).

Implementation based on the acceptance tests

In possession of automated acceptance tests, developing software becomes much easier and gratifying. Developers gain control of the code correctness (a mouse click away), focus on what must be done, have automatic separation of concerns (business logic independence from user interfaces is enforced by EasyAccept), and many other advantages that have been discussed in the preceding chapters.

The approach developers should use for implementing software swiftly is to follow some simple rules, iterating for each test:

1. Write the simplest code that makes the acceptance test pass – first, code the methods in the Façade that correspond to the commands in the script, if they don't already exist; then, from the Façade methods, design the program's classes, and continue from this point down to the low-level code.
2. If you think there is an error in an acceptance test, submit it to the client for review (finding errors in the acceptance tests forces developers to think about business rules and requirements, promoting discussion and consequently understanding of what needs to be done);
3. Don't change correct code unless you have proved it is wrong by creating a new acceptance test case that breaks it (and this test must be reviewed by the client), or if you are refactoring code (see the maintenance activity Refactoring, in a section below)
4. If, to make the acceptance test pass, you need to create untested structures not viewable or testable by any acceptance test (because it is not a business rule, or a client's concern), first write unit tests for the structure, then code what you need, using the preceding rules substituting "acceptance" by "unit", and disregarding references to the client.

Activity breakdown
Who does it: developers;

When it begins: as soon as the first acceptance tests for the user stories are ready.

When it ends: for the current iteration, when all acceptance tests created for all user stories (under development and already completed) have passed. Final project is complete when all acceptance tests for all user stories have passed.[6]

Maintenance activities

Despite the fact that at the end of an iteration a number of user stories are implemented, two maintenance activities forces developers to come back to the code of already "completed" user stories. *Refactoring* deals with code quality control and prevention to help cope with change. *Test maintenance*, in addition to ensuring and strengthening software correctness, is a facet of change itself.

Refactoring

Merely writing the simplest code that makes acceptance tests pass in the long run can make the code messy and even incorrect, if developers do not "see beyond" the tests. As features are implemented, test runs for preceding user stories may start to fail, new acceptance tests that must be complied with may become harder to make pass because the existing code is hard to modify, and especially non-functional tests, such as performance tests, may become a big issue.
The solution to alleviate this is to *refactor* code and design, making extensive use of coding best practices and design patterns (things which will not be discussed in this text, due to lack of space). Such techniques boost reuse and prepare software for change and additions.

Refactoring is a continuous activity in the process that occurs during iterations. Developers must always be refactoring, as the need is felt or as a result of specific practices, such as design review, code review, pair programming, among others. We refrain from suggesting how you will do code and design quality control, as there are as many techniques as there are arguments brought up by their respective supporters.

No matter which techniques developers use, however, they must follow these rules:

1. Before refactoring, all tests must pass;
2. After refactoring, all tests must pass.

Activity breakdown
Who does it: developers;

When it begins: in a loose statement, whenever developers feel the need for it (actually, this activity must be regulated by project management); developers must always refactor;

---

[6] Of course, there are some other things, like the client being satisfied with the user interface, but for what we're discussing here, the statement is valid.

When it ends: never, over the program's lifecycle.

Test Maintenance

This activity involves keeping the quality of acceptance tests, making them more understandable and accurate. This is accomplished by:

1. Finding additional relevant test cases;
2. Removing irrelevant/repeated test cases;
3. Improving test documentation (in the form of comments inserted in test scripts);
4. Organizing tests (see the section on acceptance test organization patterns).

Test maintenance can be viewed as refactoring for the acceptance tests, with the exception that it also involves the client, who not only helps providing new test cases or exemplifying business rules, but also reviews the tests cases created by developers or clarifies their doubts. Furthermore, when the client uses the software, he will almost inevitably find defects. These defects are captured either as new tests or as modifications of old tests.

A major risk for the project occurs when broken acceptance tests (those that don't correctly reflect business rules) start to appear in the test database. Thus, it is extremely important that developers always do three things to cope with this risk:

1. Question tests they think are wrong;
2. Suggest what they think might be the correct test to the client;
3. Only incorporate suggested tests to the test base after the client's review.

A famous sentence from Kent Beck, the author of the book Extreme Programming Explained, states that "*untested code is nonexistent code*". Similarly, in the realm of acceptance testing, we must stick to the following rule: "a*cceptance tests not reviewed by the client are*

Section 2

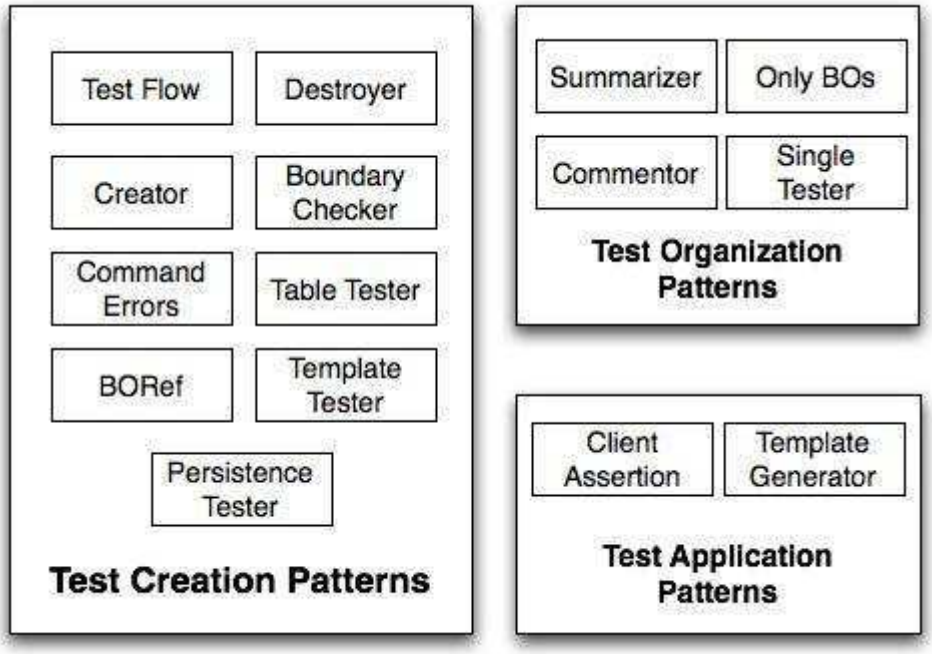


Figure 5.1 – Overview of acceptance testing patterns

The list of user stories follows, but don't feel forced to read through this list in its entirety now. You can skim it and report back whenever you feel you need details to understand an example. If anything, read through the first two user stories to recall how most rules work.

Monopoly is not a difficult program to implement, but it has many little rules that can be misinterpreted and result in bugs if not dealt with attention. Nevertheless, whoever has played this board game will probably understand the examples without effort.

User story 1

future user stories). If a player falls on a property or railroad with an available title deed, he buys it automatically, if he has enough money. After all results of a turn are resolved (and informed on the screen), the player's turn ends and the next player's turn begins, likewise: a message is shown indicating who the current player is, which actions are available, and the program asks the user to choose an action.

Rules for a Monopoly turn:

- The board is made up of 40 places, including the starting point ("Go!"). Mediterranean Avenue is place number 1, and the starting point is number 40. Place numbers, names, colors, prices and rent values (when applicable) can be found in Table 5.1:

Pos	Place Name	Price	Rent	Pos	Place Name	Price	Rent
1	Mediterranean Avenue	60	2	21	Kentucky Avenue	220	18
2	Community Chest 1			22	Chance 2		
3	Baltic Avenue	60	4	23	Indiana Avenue	220	18
4	Income Tax			24	Illinois Avenue	240	20
5	Reading Railroad	200		25	B & O Railroad	200	
6	Oriental Avenue	100	6	26	Atlantic Avenue	260	22
7	Chance 1			27	Ventnor Avenue	260	22
8	Vermont Avenue	100	6	28	Water Works	150	
9	Connecticut Avenue	120	8	29	Marvin Gardens	280	24
10	Jail – Just Visiting			30	Go to Jail		
11	St. Charles Place	140	10	31	Pacific Avenue	300	26
12	Electric Company	150		32	North Carolina Avenue	300	26
13	States Avenue	140	10	33	Community Chest 3		
14	Virginia Avenue	160	12	34	Pennsylvania Avenue	320	28
15	Pennsylvania Railroad	200		35	Short Line Railroad	200	
16	St. James Place	180	14	36	Chance 3		
17	Community Chest 2			37	Park Place	350	35
18	Tennessee Avenue	180	14	38	Luxury Tax		
19	New York Avenue	200	16	39	Boardwalk	400	50
20	Free Parking			40	Go!		

Table 5.1 – Monopoly Board Positions

- Places on which specific consequences occur are the starting point ("Go!"), properties, railroads, and taxes, only; all other positions must be treated as Free Parking – nothing happens;
- Every property is part of a group with an associated color. The four railroads compose a separate group, as well as the two utilities (Water Works and Electric Company);
- Every player begins the game with $1500;
- Any number of players can be at a same position in a given turn;
- If a player's token falls on a property owned by another player, he must pay the owner the rent value (see table);
- If a player falls on a railroad owned by another player, he must pay the owner the "ride", which is calculated by multiplying the value of the dice by a factor that depends on the number of railroads in possession of the owner: $25 if the owner has a single railroad, $50 if the owner has two railroads, $75 for three, and $100 for all four railroads;
- If a player falls on the Income Tax, he must pay $200 to the bank;
- If a player falls on the Luxury Tax, he must pay $75 to the bank;

- Every time a player falls or passes by the starting point (“Go!”), he gets $200 from the bank as salary;
- If a player falls on Free Parking or on any of the (for now) unimplemented places, nothing happens;
- If a player goes bankrupt (his cash falls below 0), he is automatically excluded from the game, and all of his belongings are returned to the bank and become available to be bought by other players; debts that exceed what the bankrupted player has been able to pay are lost; if the cash is exactly 0, the player remains in the game.
- The game ends when a single player remains.

User story 3

Creating a Script Language

A *script language* is the set of allowed commands that can be used to write acceptance tests for a program. In other words, it can be regarded as a *vocabulary* that specifies the *verbs* or *words* that will be combined to describe acceptance tests. The script language must be formally defined so that the tests can be automatically interpreted by a testing program, such as EasyAccept. Even if you don't use scripts as the means of doing acceptance testing (for example, when using FiT, you write tables in HTML files), an equivalent activity that defines the allowed "building blocks" for the tests, like table templates and diagrams, must exist.

As you recall from the ATDD activity breakdown, the script language definition must be done before any acceptance test can be written for a given iteration. The activity itself, as you will see, is pretty straightforward. In fact, the definition of most commands follows directly from a well-described user story. However, new script commands can – and typically will – be defined as needed throughout iterations, as clients and test creators (and maintainers) gain insight on new test cases and on improving current tests.

So, how do you choose the script language commands?
First, let's take a look at the *types* of script commands that one must use to create acceptance tests.

Types of script commands

There are three fundamental types of script commands: *getters*, *doers* and *preparers*.

Getters are commands that capture a current characteristic or state of a program's entity or element. In a test script, they are typically coupled with an `expect` built-in command to check if the returned value corresponds to the expected value. For the sake of standardization, always name getter commands beginning with "get". Examples of getter commands are:

```
getCurrentPlayerMoney
getEmployeeWage employeeName="John Doe"
getCustomerName customerId=01526
```

Doers are commands that correspond to actions taken in the program's business logic. Generally, a doer is coupled to an actual operation issued by a user when operating the software, or to a step in the sequence of operations that, when combined, correspond to such a user action.
Doers modify the state of the program and, as such, return values can be associated with them.
Thus, doers can either be used standalone in the script, coupled with `expect` commands or stored into variables that can be referenced in subsequent points of the script. Examples of doers:

```
rollDice firstDieResult=1 secondDieResult=3
payEmployee employeeName="John Doe"
createOrder customerId=01526
```

Preparers are ancillary commands in the script language. They exist only to serve the purpose of facilitating the writing of test cases, by setting up special situations that would only be obtained through a large number of doers.

Consider, for example, that you want to create tests for a simple information system that uses databases. Instead of having to use several `insertSomeDataYouWant` commands to create a sample database and later test if it were correctly setup (and this particular database creation sequence will probably be used to test several other user stories), you would instead want to have a command called `createSampleDatabase` that could be used as a shortcut to the entire sequence. Additionally, such a command could have an argument to specify a text file with sample data, so that multiple sample databases could be used.
Another example: in the Monopoly program, it would be tedious to check if a player has gone bankrupt having to strictly follow only commands users actually issue in the game (doers), like rolling dice and buying title deeds. To deal with this limitation, one could create a preparer command called `reducePlayerMoneyToOneDollar` to adjust a player's money low enough so that in the next move he falls on a tax place that leads him into bankruptcy. This is actually a test organization pattern, called **Summarizer**. Observe that neither preparers, `createSampleDatabase` nor `reducePlayerMoneyToOneDollar`, correspond to a useful action that would be issued in the final working program. They exist only to ease test creation.

Translating user stories into a script language

Let's try to identify commands to create a script language for Alice's Monopoly program.

Consider user story 1: "
Allow the user to create a Monopoly game. The program must ask for the number of players, which must be between 2 and 8. Then, for each player, a name and a token color must be provided. Allowed token colors are black, white, red, green, blue, yellow, orange and pink. When the game begins, all player tokens are placed on position #40 on the board, labeled "Go!" and receive $1500 in play cash.
"
With the user story on hand, you can derive some script commands directly from the description.

First of all, try to identify verbs that express actions a user would perform when operating the program. Those are candidate for doers. Moreover, pay especial attention to the key nouns of the description. They relate to elements of potential testing interest in the program. Generally, they will be coupled with a getter and/or serve as an argument for a doer.

For user story 1, the key action is to create a game. Then, let a command be called `createGame`. What information is needed in order to create a game? As stated in the user story description, one needs the number of players, the names of the players and the token colors chosen by each player. These pieces of information must be arguments to the command `createGame` when using it in a script. Number of players can be passed as an integer number (it must be between 2 and 8); player names and token colors are lists of Strings.[7]

[7] Currently, EasyAccept does not support the treatment of collections, such as lists of Strings, passed as arguments to a script command (but may do so in the near future). Lists must be expressed using a notation of your preference and your Façade must treat them explicitly. We suggest using the one we adopt in the examples of this text: comma-separated items enclosed in curly brackets. In appendix 1 you will find a discussion on unit tests and how they fit into an ATDD approach. Code provided in this appendix can be used to parse lists of Strings such as the ones used in this text's examples.

The command definition then becomes:

```
createGame numberOfPlayers=<int> playerNames=<String> tokenColors=<String>
```

From the user story description, we see that no other user action is described. The user story basically consists in player data being entered and the program setting up a game automatically. However, to fully test it, one needs a series of getters, as the description mentions lots of key nouns.

Key nouns found in the user story description are number of players, player name, token color, position on the board, label of a board position (place) and cash. The first three nouns are arguments when creating a game, the remaining are not. They all represent attributes of testing interest in the script, except perhaps for player names, which can be tested implicitly if you use them as primary keys for a player – they are tested when you use them as arguments in the script. To fully check if a game was created successfully, one needs to check if all the above-mentioned characteristics result as expected. Thus, we need getters for them. The resulting commands are:

```
<int> getNumberOfPlayers
<String> getTokenColor playerName=<String>
<int> getPlayerPosition playerName=<String>
<int> getPlayerCash playerName=<String>
<String> getPlaceLabel position=<int>
```

Notice that getters always have a return value, indicated at the beginning of each line. Also observe that every "entity" of the problem domain that needs to be tested (i.e., every Business Object, BO for short) has an associated primary key, which must be given as an argument to a getter. In the case of the BOs player and board place, their primary keys are a player's name and the position (number) of a place on the board.

For this particular user story, there is no immediate need of a preparer command. This is not be the case for user story 2, though. Let's briefly consider this user story now.

Despite the fact that user story 2 has a long description, you will notice that the only real doers (actions a user of the program issues) are rolling dice and quitting the program. Even though the end program user simply rolls dice and random face numbers result, the test command that Bob creates receives each die's value as arguments in order to test the program in a controlled fashion. Each die value must be passed individually as an argument because there are rules that depend on comparing them. As for the rest of the description, it merely discusses consequences of a player's dice throw, e.g., a player *automatically* buys deeds when his token falls on a property or railroad place, the player is *automatically* excluded from the game when he goes bankrupt, etc. Thus, Bob need only define the following doer commands for user story 2:

```
rollDice firstDieResult=<int> secondDieResult=<int>
quitProgram
```

As for the getters, you may anticipate that a good number of them will be needed. Testing user story 2 involves checking that a board was correctly set up with the appropriate place labels, colors, groups, prices and rents, when applicable. There will also be tests for each one of the business rules included in the description: testing if a player earns the correct salary

when cycling the board; testing if a player pays the correct rent or ride depending on his and other players' status (who owns which deeds or railroads); testing if a player goes bankrupt and is excluded from the game when he runs out of money, and so on. For all those tests, Bob will need to use new getters in addition to those that are already part of the script language. They are listed below:

```
<String> getPlaceName boardPosition=<int>
<String> getPlaceGroup boardPosition=<int>
<int> getPlacePrice boardPosition=<int>
<String> getPlaceOwner boardPosition=<int>
<int> getPropertyRent boardPosition=<int>
<String> getCurrentPlayer
<String> getPlayerDeeds playerName=<String>
<String> getPlaceOwner playerName=<String>
```

The commands defined so far are enough for Bob to test user stories 1 and 2, as they encompass all actions a user of the program issues and all pertaining queries needed to check the program's state.
However, as you may have anticipated, the tests for user story 2 can to be cumbersome, to say the least. Suppose that Bob wanted to test that a player really goes bankrupt when he runs out of money. With the doers `createGame` and `rollDice` alone, Bob would have to: start a given game (all players begin with $1500), issue a series of `rollDice` commands (one for each turn of each of the players) making players fall on specific places to make a given player slowly lose his money (say, by always falling on income taxes or paying rent to other players), and finally checking what he wanted: that a player went bankrupt and was removed from the game (see the test script below).

```
# testing for Alice's bankruptcy; Bob simply cycles through corners,
# Alice repeatedly falls on taxes (income and luxury); thus, his cash
# decreases $75 per cycle
createGame numberOfPlayers=2 playerNames={Alice,Bob}
playerTokens={white,black}
expect 2 getNumberOfPlayers
# Alice and Bob begin at position 40 (Go) with $1500; Alice plays first,
falls
# on Income Tax and pays $200
rollDice firstDieResult=2 secondDieResult=2
rollDice firstDieResult=5 secondDieResult=5
rollDice firstDieResult=3 secondDieResult=3
rollDice firstDieResult=5 secondDieResult=5
# Alice is now on Free Parking
rollDice firstDieResult=5 secondDieResult=5
rollDice firstDieResult=5 secondDieResult=5
rollDice firstDieResult=5 secondDieResult=5
rollDice firstDieResult=5 secondDieResult=5
# Alice falls on Luxury Tax and pays $75
rollDice firstDieResult=4 secondDieResult=4
rollDice firstDieResult=5 secondDieResult=5
# Alice falls on Income Tax and pays $200, after receiving salary ($200)
rollDice firstDieResult=3 secondDieResult=3
...
# after 10 cycles, we come to a point where Alice is on the verge of
# becoming bankrupt; the final step we needed to test is
expect 2 getNumberOfPlayers
expect false playerIsBankrupt playerName="Alice"
rollDice firstDieResult=4 secondDieResult=4
expect 1 getNumberOfPlayers
```

```
expect true playerIsBankrupt playerName="Alice"
expect false playerIsBankrupt playerName="Bob"
```

Further down in the script, Bob wants to test the "four railroads" rule: when a player owns all four railroads, other players must pay him $100 times the combined dice result when they fall on any of his railroads. To create this test, Bob would have to set up a sequence of turns being taken by players so that the same player buys all railroads and then other players fall on his railroads. Using such an approach, this and the preceding test cases would be really long-winded, hard to understand and maintain.

That's when preparers come in handy. With preparers, Bob can find shortcuts in the usual workflow of a program's execution and set up scenarios for the specific test cases he needs. He comes up with two such commands:

```
setPlayerPosition playerName=<String> position=<int>
setPlayerCash playerName=<String> cash=<int>
```

Compare how the test sequence for a player bankrupt becomes cleaner now that preparers were introduced in the script language:

```
# Testing for Alice's bankruptcy
createGame numberOfPlayers=2 playerNames={Alice,Bob}
playerTokens={white,black}
expect 2 getNumberOfPlayers
setPlayerCash playerName="Alice" cash=100
rollDice firstDieResult=2 secondDieResult=2
expect 1 getNumberOfPlayers
expect true playerIsBankrupt playerName="Alice"
```

For the "four railroads" test, yet another preparer command could be devised to make the test even cleaner: a command that gives a deed to a player. This command would avoid a player's need to fall on each of the railroads to buy them automatically.

Defining preparers is meant for the tester's convenience. The rule of thumb is to define as many preparers as it takes to make tests understandable and easy to maintain. There is a tradeoff to explore in this area because too many specific preparers, while they may make a test script more readable, may make them too complicated to maintain. The script below is easy to understand, but most of the commands are not usable in most tests.

```
createGame numberOfPlayers=2 playerNames={Alice,Bob}
tokenColors={blue,yellow}
giveAllRailRoadsToPlayer playerName=Alice
goToNextRailroad playerName=Bob
expect 1000 getPlayerCash playerName=Bob
goToRailroadWithMaximumDiceResult playerName=Bob
...
```

Now Bob and Alice have defined a working set of commands to write tests for the first two user stories. When discussing the patterns **Test Flow** and **Creator**, you will see how to use these some of the commands in scripts that test aspects of this user story. If you want to check all tests for Monopoly, check out EasyAccept's homepage (http://easyaccept.org).

Test Flow

Once you have defined a script language with the commands you need to test a user story, you are ready to begin writing the actual tests. So, how do you begin?

This and the next three patterns are what can be regarded as *general* testing patterns. Whenever you are creating a test (an acceptance test, unit test or any other kind of test), you will find yourself applying one of these patterns, even if you don't realize it. These patterns, for instance, have been used in chapter 3, when Bob wrote tests for Alice's poll program. They will be used to test Alice's Monopoly program, too.

The first of these patterns is used when you want to verify that a given software characteristic results as expected after issuing a given action in a particular situation – a common reason for writing a test. This "results as expected" statement could be either "the program produces the correct output" or "a program error occurs".

The basic framework is to use a three-step sequence:

(1) **build** a scenario that puts the program into the initial state you need for the test;
(2) **operate** the program using the desired action; and finally
(3) **check** if the new program state results as you expected.

Take the following test from Alice's poll program:

```
1 clearSystemData
2 poll1=createPoll name="Do you like apples?" answers={yes,no}
3 vote poll=${poll1} answer=yes
4 expect 1 getNumberOfVotes poll=${poll1} answer=yes
5 expect 0 getNumberOfVotes poll=${poll1} answer=no
```

This is a simple test that checks if voting for a given answer (action) results in a vote being counted for the desired answer and not for the other answers. Lines 1 and 2 represent the build phase – creating a poll with the desired answers – that sets the stage for the desired operation; line 3 alone contains the operation: casting a vote for the answer "yes"; and lines 4 and 5 do the checking: there must be one vote for the "yes" answer and zero votes for the "no" answer.

Another example, from Alice's Monopoly program: testing if a player has gone bankrupt using the preparer `setPlayerCash`:

```
1 # Testing for Alice's bankruptcy
2 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
3   playerTokens={white,black}
4 expect 2 getNumberOfPlayers
5 setPlayerCash playerName="Alice" cash=100
6 rollDice firstDieResult=2 secondDieResult=2
7 expect 1 getNumberOfPlayers
8 expect true playerIsBankrupt playerName="Alice"
9 expect false playerIsBankrupt playerName="Bob"
```

The build phase is found from lines 2 to 5 – creating a game and setting a player with low cash; the operation consists of a `rollDice` in line 6 (so that the player falls on income tax and loses all his remaining money), and lines 7 to 9 convey the check (only one player should be left in the game, and this player should be Alice).

Note that the build phase may also contain checks, like in line 4. Before checking if one player will remain after the other one is removed from the game, we must check if a game with two players was created in the first place. This was necessary in this particular test because the creation of a game is still untested. If a game creation were already tested, line 4 would not be necessary (and, in fact, undesired). Let's take this for granted in the next example:

```
1 # checking if the luxury tax of a Monopoly game takes $75 from a player
2 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
3   playerTokens={white,black}
4 # place Alice on "Jail: Just Visiting" and set her with $1000
5 setPlayerCash playerName="Alice" cash=1000
6 setPlayerPosition playerName="Alice" position=30
7 # doer: make Alice fall on luxury tax
8 rollDice firstDieResult=4 secondDieResult=4
9 # check: Alice should have lost $75
10 expect 925 getPlayerMoney playerName="Alice"
```

Observe that the build phase is generally larger than the operate phase (which generally consists of a single line) and the check phase (whose size depends on the number of characteristics you want to check). If, instead of checking property values, you want to check if an error occurs when operating the program, the operate and check phases are combined in a single line. See below, for example, a test for user story 4 of Alice's Monopoly game.

When a player is sent to jail, he becomes free when he throws a double in one of his next turns. Alternatively, he can pay to get out of jail or use an optional "Get Out of Jail" card, if he has one. In the test below, we use a preparer command, `sendToJail`, to send Alice to jail. When her turn comes, she won't have the option of using a "Get Out of Jail" card, since she doesn't own one.

```
1 # error when trying to use a jail card
2 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
3   playerTokens={white,black}
4 # send Alice to Jail
5 sendToJail playerName="Alice"
6 # operate and build phase combined: error when operating: using card
7 expectError "Player doesn't have a jail card" useJailCard
```

The build phase consists of lines 2 to 5, and the operate and check phases are combined in line 7.

Even when a test is made up of many lines, with many commands, the three-step sequence should be easy to identify. If not, this may be the sign of a bad test: one that tests many things simultaneously or one that is too verbose or redundant. Later on, we will present two patterns that help cope with those bad tests: **Single Tester**, which identifies and separates multiple combined tests and **Summarizer**, which deals with test redundancy.

Pattern outline

Context: you need to verify that a given software property results as expected after a given action.

Problem: the lack of a basic structured test sequence gives rise to tests that are complicated, hard to understand and maintain.

Forces: clients don't have a testing background and need an approachable way to understand tests without much effort. Furthermore, as reviewers of acceptance tests, clients must gain a testing culture to understand how software testing is done.

Solution: **Test Flow** provides the basic structure for an acceptance test, consisting of three steps. The first step is to build a scenario – a particular state of software taking into account the elements you want to test. In this step you find shortcuts in the logic flow of the program to the scenario you want to test. Once the scenario is built, you perform the specific actions that modify the current state of the program to the state you want to check. Finally, you check if the program state has resulted as expected, comparing values of properties of interest to values you want. If the expected outcome of the action is an error, the check step is frequently coupled with the operation step.

Rationale: the 3 steps sequence is easy to remember, apply and identify when reading a test script. This helps alleviate the learning curve of reviewing acceptance tests for the clients.

Creator and **Destroyer**

Creator and **Destroyer** are patterns used in the common situation of "creating" and "destroying" Business Objects – software elements of business interest. Every time you introduce a Business Object in a program, you must test it to make sure it was successfully created. Likewise, when you are done using it, you must make sure the process of destruction was completed adequately.

Many times when testing software, you need to create and destroy multiple BOs. If these operations aren't tested, a potential bug in them will interfere with many correct tests. Furthermore, if you don't use a systematic approach to test the creation and destruction of BOs, a number of redundant or unnecessary tests may appear scattered throughout your test script.

Creator and Destroyer provide a simple, short and direct one-shot test sequence for creating and destroying BOs. You need a Creator for every script command that creates a BO, and a Destroyer for every command that destroys one.

This is the profile for Creator: for every command that creates a BO:

(1) check that the BO doesn't exist;
(2) create the BO using the script command you defined;
(3) test if the BO was correctly created;
(4) make sure the same BO can't be created again (if applicable)

In order to accomplish the first step, expect an error when trying to access the given BO with any of its getters. Step 3 involves checking all pertinent properties of the BO that define it as being correctly created. Let's see an example:

```
1 # Using Creator to test an employee's creation
2 expectError "Employee does not exist" \
3   getEmployeeSalary employeeId=1434
4 createEmployee id=1434 employeeName="John Doe" salary=1500
5 expectError "Employee with this id already exists" \
6   createEmployee id=1434 employeeName="John Doe" salary=1500
7 expect "John Doe" getEmployeeName employeeId=1434
8 expect 1500 getEmployeeSalary employeeId=1434
```

In this example, an employee with an id of 1434, a name John Doe and a salary of $1500 is created. The employee's id is the BO's primary key, so his name and salary are the pertinent properties that need to be checked. The command `getEmployeeSalary` is used prior to creation to assert that an error is thrown. As a primary key, no other employees can be created with the same id. This is reflected in the test comprised in lines 5 and 6, which makes up for step 4.

Suppose you had defined an additional command to create an employee. Say, a command in which, along with an id, a name and a salary, another employee data (e.g., birth date) was passed as an argument. This other command would need a Creator test, too, in addition to the Creator used in the example above.

When you use an internal unique identifier as the primary key of a BO (i.e., when you apply the pattern **Business Object Reference**), you can't tell the key value beforehand. In this case, you can refer to the EasyAccept variable that you'll use, like in the example below, taken from Alice's Poll program:

```
1 # using Creator to test the creation of a poll
2 expectError "Poll does not exist" getPollName poll=${poll1}
3 poll1=createPoll name="Do you like apples?" answers={yes,no}
4 expect "Do you like apples?" getPollName poll=${poll1}
5 expect "{yes,no}" getPollAnswers poll=${poll1}
```

Observe that step 3 is longer in this case, because to properly test if a poll was created, we need to check its name and its answers (in lines 4 and 5), given that the primary key is an assigned identifier. Also observe that this Creator has no step 4. You can always create a new poll, even if another one with the same name and answers already exists in the system.
Now let's analyze another example, taken from Monopoly:

```
1 # using Creator to test a Monopoly game creation
2 expectError "game does not exist" getNumberOfPlayers
3 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
4   playerTokens={white,black}
5 expect 2 getNumberOfPlayers
6 expect "{Alice,Bob}" getPlayerNames
7 expect "{white,black}" getPlayerTokens
8 expectError "a game already exists" createGame numberOfPlayers=2 \
9   playerNames={John,Mary}playerTokens={blue,green}
```

In the case of Monopoly, no primary keys are necessary, as only one game can be created at a time. That's why even a completely different game couldn't be created in lines 8 and 9.

After you've applied the **Creator** pattern, you won't need to be checking BO creation properties in other parts of the script, in the middle of tests for other purposes. One important remark follows. Note that the mere application of a **Creator** pattern does not guarantee that the creation of a BO is fully tested, because it says nothing about error situations that should be raised when trying to create a BO. That's why you should also apply the **Command Errors** pattern for every creation command.

Now let's see how to test the destruction of a BO. See the profile of the **Destroyer** pattern: for every command that destroys a BO:

(1) make sure the BO exists (optional);
(2) destroy the BO using the command you defined;
(3) check that it doesn't exist anymore;
(4) check that it can't be destroyed again

Let's use the same examples we did for **Creator**.

```
1 # Using Destroyer to test an employee's removal
2 expect 1500 getEmployeeSalary employeeId=1434
3 removeEmployee employeeId=1434
4 expectError "Employee does not exist" \
5   getEmployeeSalary employeeId=1434
```

```
6 expectError "Employee does not exist" \
7   removeEmployee employeeId=1434
```

Line 2 alone contains the first step, checking that the employee with the id 1434 exists. Depending on the point in the script where you apply the **Destroyer**, this first step may be optional. For example, if the **Destroyer** is placed right after the corresponding Creator the first step is unnecessary. Line 3 contains the destructor command, `removeEmployee`. Checking that this employee was actually removed is accomplished by the command found on lines 4 and 5. Finally, lines 6 and 7 check that employee 1434 can no longer be removed.

The next example shows a **Destroyer** being applied to the poll program, considering that `poll1` has previously been created in the script and checked with a **Creator**, so that the first step of **Destroyer** is not needed.

```
1 # using Destroyer to test the destruction of a poll
2 destroyPoll poll=${poll1}
3 expectError "Poll does not exist" getPollName poll=${poll1}
4 expectError "Poll does not exist" destroyPoll poll=${poll1}
```

Since, in Monopoly, only one game can exist at a time, we also take for granted that a functional game has previously been created in a script.

```
1 # using Destroyer to test a Monopoly game destruction
2 endGame
3 expect true gameIsOver
4 expectError "Game is already over" endGame
```

Note how we've taken a different approach for step 3: we used a command that tells if a game is over, but we could have used an `expectError` coupled with a `rollDice` instead.

Make a habit of systematically using Creator and a Destroyer for every creation and destruction operations on a BO. Separate them from the rest of the script, and don't forget to use **Command Errors** on each of these operations. Creator and Destroyer alone don't guarantee that the operations are fully tested.

Pattern outline

Context: you need to test if a Business Object has been successfully created or removed in the program; those are common operations in test scripts.

Problem: the lack of a systematic repeatable sequence to test Business Object creation and destruction hinders test understanding and may leave these operations incompletely tested.

Forces: clients don't have a testing background and need an approachable way to understand tests without much effort. Furthermore, clients must gain a testing culture.

Solution: whenever you are creating Business Object (a new client, a new player, a new purchase order) that will be used in a test, first make sure that it doesn't exist (expecting an error), then create it, make sure it exists and finally check that it can no longer be created (if duplicates are not allowed). When testing the Business Object's destruction, first check that it

exists; then destroy it, check that it doesn't exist, and finally check that you can no longer remove it.

Rationale: this is a clear-cut, easy-to-follow sequence of steps that contains all of the important tests for business object creation and destruction.

Related Patterns: **Creator** is often used in conjunction with **Business Object Reference**.

Command Errors

Testing is an exercise in active search for the unusual, the strange, and the abnormal. It involves much more identifying the situations where a program must *not* work properly than those where everything should work fine. In ATDD, we employ user-defined commands to do so. However, an important catch of the approach is that the commands can be themselves source of bugs.

When you create a new command to use in a test script, you must make sure developers implement it correctly. Otherwise, bugs can be inadvertently introduced in the program due to typos or other errors in the tests.

What should you do? Apply a systematic approach and make it a habit. Every time you create a new command, open a script and type right away a series of tests for the command. Think about what the command does, its scope, its limitations, and contexts or situations where it should not be used. Generally, the tests will consist in error declarations using `expectError` for the illegal uses, because the normal use will come with the regular application of the command in the test script.

Let's see a clarifying example. In the simple employee management program, suppose we had just devised a new command, called `setEmployeeSalary`, which takes an employee id as an argument and changes the employee's salary. Suppose you didn't use **Command Errors** and the command is left untested. An inattentive developer didn't bother coding for, say, a negative salary as invalid. Some days later, hours are lost in an attempt to resolve a bug that ultimately lied in a typo: somewhere in the script, `setEmployeeSalary` was used with a negative salary, which resulted in a test further down in the script persistently breaking.

An error should have been thrown when a negative salary was issued. This would break the test with the typo, had the developer coded it. But to ensure he had coded for this error, an explicit test indicating it should appear in the script.

This is what **Command Errors** advocates. The best moment to code errors for a command is when you create it, when your mind is fresh with ideas on how you will use it. An additional advantage of the pattern is that thinking about a newly created command's limitations often makes you find useful tests for the program under test. In the example, a negative salary should not be allowed in the underlying program.

For `setEmployeeSalary`, the following tests pack up the initial set of command errors.

```
1 expectError "employee doesn't exist" setEmployeeSalary \
2   employeeName="Nonexistent John"
3 expectError "salary must be positive" setEmployeeSalary \
4   employeeName="John Doe" salary=-15
5 expectError "salary must be positive" setEmployeeSalary \
6   employeeName="John Doe" salary=0
```

The tests check for three error situations. The first case (lines 1 an 2) is when you try to set the salary for a nonexistent employee. The second and third cases (lines 3 to 6) occur due to invalid salaries. Note that these errors apply for illegal uses of the command within a test

script, but also correspond to illegal uses of the function of changing a salary in the final program. Naturally, a number of other illegal salaries could apply depending on specific rules that are incorporated later in the script.

A useful habit that helps better organize test scripts is to group the command errors in a separate section. This improves readability and helps pinpoint test bugs when running the scripts. In chapter 3, Bob used **Command Errors** when writing tests for the first user story of the poll program and does just that. In this user story, he devised the command `createPoll` and two getters, `getPollName` and `getPollAnswers`. In a separate section defined in the final lines of the script, he includes tests that cover illegal uses of these commands. See below an excerpt from the script:

```
10 # using the pattern Command Errors on the command createPoll
11 expectError "Poll must have a name" createPoll name="" answers={yes,no}
12 expectError "Poll must have at least two answers" \
13   createPoll name="Do you like apples" answers={yes}
14 expectError "Poll must have at least two answers" \
15   createPoll name="Do you like apples" answers={}
16 expectError "Poll must have at least two answers" \
17   createPoll name="Do you like apples" answers=""
18 # Command Errors on the getters
19 expectError "Poll does not exist" getPollName poll=abc
20 expectError "Poll does not exist" getPollAnswers poll=abc
```

Also observe from this example that testing a command begins by validating the arguments it receives. The program under test should throw an error every time one tries to use a command with an invalid argument. In the case of `createPoll`, invalid arguments are an empty name or a number of answers less than 2, a business rule. The getters, in their turn, must throw errors whenever nonexistent polls are passed as arguments.

Pattern outline

Context: you have created a new command to incorporate in the script language and will start writing tests with it. The new commands can be used in the wrong way.

Problem: addition of new commands involves interacting with new, untested software operations that can be the source of bugs.

Forces: test coverage is difficult to attain if you don't approach testing systematically; when thinking of a new command to add to the script language, you often think about tests that will involve this command, including anomalous uses; the program under test should be able to identify a misuse of its operations and may even give clues about the problem cause

Solution: every time you create a new command to incorporate in the script language, you must systematically write tests that devise a "fire curtain" of error tests that exhaust the possibilities of errors: think about the command's scope, limits, restrictions and invalid situations that it could cause. Additionally, put the command error testing in a separate section of the script for clarity.

Rationale: writing tests to cover misuses of a command early saves debugging time later and helps finding useful tests.

Boundary Checker

"Tests prove the existence of bugs, but not their absence." This famous quotation from Edsger Dijkstra is a most discouraging truth of software testing. You can only prove a program is 100% correct if you exhaust all possibilities of combinations of data entered and operations issued in all scenarios (or if you mathematically guarantee that this holds). This is always very difficult if not impossible to do even if you use automated generation of tests, let alone involving clients to verify test cases.

Other than in real-time systems or programs that need close to perfect reliability, this obsessive-compulsive approach is fortunately not generally necessary in most information systems.

What is done in practice is to use testing by example. What does this mean? In most cases, you only need a single test (example) to represent a huge set of invalid operations. For example, take the command `setEmployeeSalary`. A single line is enough to state that a negative salary is invalid, using a single value, e.g., -1.

```
expectError "must be positive" setEmployeeSalary \
  employeeName="John Doe" salary=-1
```

You (the tester) don't need to repeat the same line over and over again to wear out other orders of magnitude of negative salaries, because the developer can get the point. He will simply write down a condition of salary < 0 throwing an error in the code. This is a rather simple example, of course, but it makes the point: give one example for every group of tests.

This rule of giving examples applies particularly well to the case when you need to test ranges of values. The pattern **Boundary Checker** states that, when you need to test ranges of values, you need to give examples of the boundaries. If the range has a minimum value, demarcate the boundary by giving an example with such a minimum value being accepted, and an example of the next lower value throwing an error. Do the same for the maximum value.

Let's analyze the pattern applied to the first user story of Monopoly. It states that a Monopoly game must accept a number of players between 2 and 8. How do you test if the program accepts the range correctly? According to **Boundary Checker**, you need four tests: that a game with 2 players is accepted, that a game with 8 players is accepted, and that neither a game with 1 nor with 9 players is accepted. The excerpt from the script follows:

```
1  expectError "Too few players" createGame \
2    numberOfPlayers=1 playerNames={John Doe} tokenColors={black}
3  expectError "Too many players" createGame \
4    numberOfPlayers=9 playerNames={a,b,c,d,e,f,g,h,i}\
5    tokenColors={black,white,green,red,blue,yellow,brown,gray,pink}
6  createGame numberOfPlayers=2 \
7    playerNames="{John Doe,Mary Donna}" tokenColors={black,white}
8  finishGame
9  createGame numberOfPlayers=8 \
10    playerNames={a,b,c,d,e,f,g,h}\
11    tokenColors={black,white,green,red,blue,yellow,brown,gray}
```

Lines from 1 to 5 represent the tests for the outward limits of the range, the ones that need to throw errors. Lines 6 to 11 make sure inward limits are accepted. At this point, you might think: do you really need to place lines 6 and 7 here? After all, these lines will be repeated in the **Creator** for a Monopoly game. Thinking even further: do you need to place lines 1 to 5 here? They will also be part of the **Command Errors** for `createGame`.

Well, this is a concern of the most painful aspect of ATDD: organizing tests. There is a tradeoff between redundancy and understandability to explore in acceptance testing, because of the involvement of the client. Even if the tests indicated are repeated elsewhere in the script, it is undoubtedly clearer if they are kept together.

Later in this section, we will see some patterns that help cope with test organization. For now, suffice it to say that the test writer's convenience will dictate whether or not he will use redundant tests.

As you may have noticed, **Boundary Checker** is a general testing pattern that can be used in conjunction with other patterns. In the Monopoly example above, you saw how it helps find command errors when thinking about a new command.

When you need to check limits for floating point values that are an *output* of the program's execution, the application of **Boundary Checker** can be simplified, because EasyAccept has a built-in command for this function: `expectWithin`. The range can be stated as a precision around an expected value, and limit inward and outward values are automatically taken care of upon running the script. In the line below, the employee's salary must have the value expected ($2345.67) with a precision of 1 cent.

```
expectWithin .01 2345.67 getSalary employee=employee1
```

Pattern outline

Context: you need to test a business object or program flow that has limits (ranges or sets of allowed values).

Problem: unchecked limits are common sources of software bugs and the lack of a systematic approach to testing limits leaves room for unchecked limits.

Forces: testing by example, although not necessarily complete, is generally enough when it comes to communicating valid and invalid situations to the developer. From the special cases, the developer can generalize the working code.

Solution: acceptance testing involves giving "examples" of valid and invalid software operations or business object's properties. When writing tests, always demarcate precisely limits by giving examples of inner and outer bounds. Make sure values immediately out of bounds throw errors and inbound values don't.

Rationale: in general, tests for the immediate limits are enough to indicate to the developer that further values aren't acceptable, even if the tests don't explicitly list them all.

Related patterns: you often use **Boundary Checker** in conjunction with **Command Errors** to increase test coverage.

Table Tester

One of the drawbacks of scripted testing is the verbosity that follows in two situations: when testing long-winded sequences of data, and when testing multiple related examples that use the same structure.

See below an example of a long-winded sequence of data. In Monopoly's user story 2, Bob needs to test if the board was correctly set up. It is composed of 40 places, each of which has a bunch of characteristics. This is not a case of a range of values that can be summarized with an example, so each one of the places must be tested for every characteristic.

```
1 expect "Mediterranean Avenue" getPlaceName placeID=1
2 expect "purple" getPlaceGroup placeID=1
3 expect "bank" getPlaceOwner placeID=1
4 expect 2 getPropertyRent placeID=1
5 expect 60 getPlacePrice placeID=1
6 expect "Community Chest 1" getPlaceName placeID=2
7 expect "chest" getPlaceGroup placeID=2
8 expectError "This place can't be owned" getPlaceOwner placeID=2
9 expectError "This place doesn't have a rent" getPropertyRent placeID=2
10 expectError "This place can't be sold" getPlacePrice placeID=2
11 expect "Baltic Avenue" getPlaceName placeID=3
12 expect "purple" getPlaceGroup placeID=3
13 expect "bank" getPlaceOwner placeID=3
14 expect 4 getPropertyRent placeID=3
15 expect 60 getPlacePrice placeID=3
…
```

For every place, a total of five lines are used to test its characteristics: name, group (if applicable), owner, rent (if the place is a property), and price (if it can be sold). A total of 200 lines are necessary to test all 40 places. This long-winded list is cumbersome to look at and analyze. However, if you translate it into a table, see how much more readable the data becomes, with one line per place (only the first 3 lines are shown, for simplicity, but you may refer to the complete table in the beginning of this section):

Pos	Name	Group	Owner	Rent	Price
1	Mediterranean Avenue	Purple	Bank	2	60
2	Community Chest 1	Chest	!"This place can't be owned"	!"This place doesn't have a rent"	!"This place can't be sold"
3	Baltic Avenue	Purple	Bank	4	60

Cells beginning with an exclamation mark mean that an error with the message that follows is expected. This is equivalent to the `expectError` command in the script.

Let's see another example, taken from the employee management program. Suppose that the client has a business rule in the way of a somewhat complicated formula. It calculates an employee's wage based on a number of variables, like number of children, amount of sales and social security due.

The client states the rule as follows: “employees are paid a base salary plus 1% of his sales in the month minus the social security, which is a base value times the number of members of his family”. The way formulas are tested is through examples. The examples must not be redundant, but must be comprehensive, with one example for every isolated combination of variables.

Due to lack of space, we won’t present here the full list of examples, but let’s analyze two of them to compare the tabular vs. sequential approaches:
“if the salary is $1500, employee sells $50000, and has 2 kids, with a base social security value of $20, we owe him $1940”
“if the salary is $1000, employee sells $20000, and has 3 kids, with a base social security value of $20, we owe him $112”
Presenting the examples in a tabular format, we have this:

employeeName	Salary	maritalStatus	numberOfKids	socialSecurity	Sells	Wage
Employee1	1500	Married	2	20	50000	1940
Employee2	1000	Married	3	20	20000	1120

The final column expresses the wage due to the combination of characteristics for the employee described in the line. If the examples were translated into a sequential script, it would look like:

```
1 # first example
2 id1=createEmployee employeeName="Employee1" \
3   salary=1500 maritalStatus=married numberOfKids=2 socialSecurity=20
4 # for a more typical scenario, there could be multiple
5 # employeeSale, but let's say the total was from a single sale
6 employeeSale id=${id1} saleValue=50000 date=05/02/06
7 expect 1940 calculateEmployeeWage employee=${id1}
8 # second example
9 id2=createEmployee employeeName="Employee2" \
10   salary=1000 maritalStatus=married numberOfKids=3 socialSecurity=20
11 # for a more typical scenario, there could be multiple employeeSells,
12 # but let's say the total was from a single sale
13 employeeSale saleValue=20000 date=05/02/06
14 expect 1120 calculateEmployeeWage employee=${id2}
```

That is, you create an employee with the characteristics you want to test, and then compare the expected wage with the one calculated by the program.

These are examples of tests that are inherently *tabular*. They take advantage of a tabular format of presentation which is the format used in tabular acceptance testing tools, like FiT. However, how do you cope with tabular tests in scripts? EasyAccept provides a way of describing tabular data through the built-in command `processThisLoop`. Using this command, you can describe a fragment of a script using variables, and then substitute the variables with cell values on a table. For the first example of the examples, the script using `processThisLoop` is this:

```
 1 processThisLoop
 2   expect ${placeName} getPlaceName placeID=${placeId}
 3   expect ${placeGroup} getPlaceGroup placeID=${placeId}
 4   expect ${placeOwner} getPlaceOwner placeID=${placeId}
 5   expect ${placeRent} getPropertyRent placeID=${placeId}
 6   expect ${placePrice} getPlacePrice placeID=${placeId}
 7 forThisData placeId placeName placeGroup placeOwner placeRent placePrice
 8   1 "Mediterranean Avenue" purple bank 2 60
 9   2 "Community Chest 1" chest !"Can't be owned" !"Doesn't have a rent" \
10     !"Can't be sold"
11  3 "Baltic Avenue" purple bank 4 60
12  4 "Income Tax" tax !"Can't be owned" !"Doesn't have a rent" \
13     !"Can't be sold"
14  5 "Reading Railroad" railroad bank !"Doesn't have a rent" 200
15  6 "Oriental Avenue" "light blue" bank 6 100
16 endLoop
```

The syntax of `processThisLoop` is general and can be applied to virtually any repeatable script fragment. The first 6 lines of the example describe the script fragment that will be looped. In this case, a sequence of 5 expects, each one for one of the place characteristics that need be tested. Line 7, which begins with the reserved word `forThisData`, represents the headers of the table, with the sequence of columns that must match the value of cells. Lines 8 through 15 contain the lines of the table, each line with a number of cells corresponding to the headers. If, in a cell, an error is expected instead of a value, the character "!" must be used, following with the expected error message enclosed in "".

```
 1 processThisLoop
 2   ${id}=createEmployee employeeName=${name} salary=${salary} \
 3   maritalStatus=${maritalStatus} numberOfKids=${kids} \
 4   socialSecurity=${socialSecurity}
 5   employeeSale id=${${id}}
 6   expect ${wage} calculateEmployeeWage employee=${${id}}
 7 forThisData id name salary maritalStatus kids socialSecurity wage
 8   employee1 1500 Married 2 20 50000 1940
 9   employee2 1000 Married 3 20 20000 1120
10 endLoop
```

The tabular format is certainly clearer to understand in these cases, but it is still not very client-friendly, especially the description of the fragment after `processThisLoop`. In the future, EasyAccept will have an IDE through which a table can be visually edited.

Whenever you have tests that are inherently tabular, like the two examples we gave, use the tabular format. This is the pattern **Table Tester**. They become not only clearer to understand, but also much easier to pick and modify later. Every additional place in the Monopoly test or example in the employee test means a bunch more lines in the sequential script. Moreover, should you need to modify the test sequence, you only have to update it once, in the table description, and not in multiple points scattered throughout the long-winded sequence of tests.

Pattern outline

Context: you need to test extensive lists of features for multiple business objects of the same kind, or multiple examples to test *formulas* (calculations); clients want to declare business rules in a formulaic manner.

Problem: sequences of commands become too verbose when it comes to testing extensive lists of properties or calculation examples. This makes a script distracting, hard to review and maintain.

Forces: developers need an algorithmic translation of business rules so that they can be tested in an automated way.

Solution: use tables for testing in the fashion table based tools do, either by using an IDE that translates tables to scripts automatically, or by employing tailored built-in commands that simulate tables (like EasyAccept's `processThisLoop` command). Formulaic statements should be added to the script as a comment for the sake of understandability.

Rationale: Tables ease up reading long-winded lists of properties and formulas and, in the case of formulas, hides its associated algorithm from the script while still keeping the algorithm accessible to the developers in the code that links the tests to the software being tested. Tables also help when you need to modify the script.

Template Tester

Suppose we have to create a direct mail program for the Sell-It-All Department Store. It keeps record of clients, including personal data such as interests, favorite food, and so on. Then, based on some criteria, the program automatically generates personalized mail to clients so as to lure them into thinking they are unique and buying some stuff.

For example, the customer John Doe, who lives in Los Angeles, likes gardening. The program automatically generates the following mail to be sent to John Doe (personalized data are indicated in italics):

"Dear *John Doe*,
Knowing you as well as we do, we recognize quality is very important to you. You have style and a unique taste for the very best products. That's why we are pleased to offer, only to special customers like you who like *gardening*, this magnificent WATER-O-GIZMO (see picture below), that will make your flowers blossom like no other. Please pay us a visit at our local store in *San Francisco*.
Best regards,
Sell-It-All Department Store"

Based on John's interest (gardening), the program matches a suitable product and recommends a store near his address.

This functionality poses two problems to accepting testing using scripts. The first is that it is awkward to assert the full message with an `expect` command. The second is that there is no way to attach a picture to a script. See below the test script for this functionality using the conventional way of testing (excluding the picture tests, naturally).

```
1 id1=createCustomer name="John Doe" interest="gardening" address="L.A."
2 expect "Dear John Doe,\nKnowing you well as we do, we recognize \
3   quality is very important for you. You have style and a unique taste \
4   for the very best products. That's why we are pleased to offer, \
5   only for special customers like you who like gardening, this \
6   magnificent WATER-O-GIZMO (see picture attached to this message), \
7   that will make your flowers blossom like no other. Please pay us a \
8   visit at our local store at Los Angeles.\nBest regards,\n \
9   Sell-It-All Department Store" generateMail customer=id1 \
10  output="johnDoeResult.txt" pic="johnDoePic.jpg"
```

There is a single `expect` in the script: lines 2 to 10 represent only one test. For other tests, texts of similar length are employed.

In these cases, you can use the pattern **Template Tester**. It consists in diverting the output of the program to an external file and then comparing this file with a template. See below what the direct mail test script using **Template Tester** looks like:

```
1 id1=createCustomer name="John Doe" interest="gardening" address="L.A."
2 generateMail customer=id1 output="johnDoeText.txt" pic="johnDoePic.jpg"
3 equalFiles johnDoeText.txt templates/johnDoeTemplate.txt
4 equalFiles johnDoePic.jpg pictures/waterOGizmo.jpg
```

In line 2, the command `generateMail` generates both the personalized text and the picture of the product offered to John Doe. They are stored in the files `johnDoeText.txt` and `johnDoePic.jpg`. In lines 3 and 4, these output files are compared with templates conveniently placed in specific folders via the command `equalFiles`.

An additional advantage of using **Template Tester** is that templates are isolated from the test script and can be modified without changing the test structure. This is well suited to the approach of usage test case generation. We will talk in more depth about this when discussing the **Template Generator** pattern. But, in general terms, it consists in making the client or end user operate the program when it is partially complete and recording his actions. When the results of the program's execution appear, the client either asserts or rejects them. These results can then be used as templates to test the program.

Pattern outline

Context: you need to test massive textual or non-textual content.

Problem: massive textual content is cumbersome when included in a script because it may render the script unintelligible; non-textual content can only be directly included in a script through an IDE.

Forces: templates for test cases can be generated automatically from the customer using partially complete software.

Solution: divert the contents that need to be tested to a convenient place (a file) outside the script and compare it to a template.

Rationale: the solution allows non-textual data to be tested within a script and hides the massive textual data from the script.

Related Patterns: **Template Generator** can be used in conjunction with **Template Tester**

Persistence Tester

A fundamental requirement of virtually any program is to persist the data it manipulates. But how can you test persistence in a test script, if to do it you need to execute the program multiple times? The solution is to apply a sequence of steps just like you do with **Test Flow**. The steps are as follows:

(1) make sure the information you need to test for persistence is cleared from the program;
(2) enter the information you need to test and optionally test if it was entered correctly;
(3) make the program under test save data and close the program;
(4) restart the program and run the persistence test.

The first step is needed to guarantee that data entered in this test session is the one that persists. Steps 2 and 3 are easy to follow. Step 4 can be accomplished in two ways: you can either quit the testing tool and run a separate persistence test script in a following test session, or you can restart the program via the testing tool and run the persistence tests in the same script. Step 3 may not be needed if persistence is automatically performed in the program under test with a database, for example.

For example, in the employee management system, you need to check if John Doe's salary persists after his data is entered in the system. Using the first approach, you need two scripts. The first one is as follows:

```
1 clearEmployeeDatabase
2 id1=createEmployee employeeName="John Doe" salary=1500
3 expect 1500 getEmployeeSalary employee=${id1}
4 saveEverythingAndCloseProgram
5 quit
```

In this script, you clear all employee data, create the employee John Doe with a salary of $1500, check that his salary is correct, save the data and close the program. Finally, the command `quit` is used to terminate the execution of EasyAccept.

In a second script, you would simply check that the data persisted. There must be an employee with the name John Doe, and his salary must be $1500 (see below).

```
1 id1=getEmployeeByName name="John Doe"
2 expect 1500 getEmployeeSalary employee=${id1}
```

Observe that you need to create a new variable and use a special command to capture back which employee goes by the name "John Doe". This would not be necessary in the other approach. In it, you use a single testing session to test for persistence. In order to do so, you need to restart the testing tool. This is accomplished in EasyAccept by issuing he command `restart`. It terminates the program's execution and re-executes it, keeping all variables stored in the session. See below how the script using this second approach looks like:

```
1 clearEmployeeDatabase
2 id1=createEmployee employeeName="John Doe" salary=1500
3 expect 1500 getEmployeeSalary employee=${id1}
```

```
4 saveEverythingAndCloseProgram
5 # restart creates a new Façade, essentially restarting the program under
test
6 restart
7 # checking persistence
8 expect 1500 getEmployeeSalary employee=${id1}
```

Pattern outline

Context: you need to test that data entered in the program persists in future sessions.

Problem: the lack of a systematic approach to testing persistence can hinder understanding (particularly by the client) and test coverage.

Forces: setting up a scenario for persistence can often be reused as other tests.

Solution: testing persistence consists in two parts: in the first, the program under test is run, data is cleared, some information is entered and checked, and then the program is closed. The second part runs the program and checks if the information entered in the first part is still there.

Related Patterns: **Persistence Tester** is a special case of **Test Flow**.

Business Object Reference

A core concept in Computer Science is that of a primary key. A primary key is a *unique* identifier for an entity (tables in databases, objects in programs, etc.). Two distinct objects, for example, can have the exact same values for all their attributes *but* for the primary key. Two customers of a department store can live at the same address, have the same birth date, have the same tastes; they can even have the same name, but they cannot have the same id card number or social security registration – these could serve as primary keys for a customer. In practice, however, even these numbers aren't generally used as primary keys due to practical reasons. An illegal immigrant without a visa won't have an id card number (but the department store is certainly interested in increasing its income with that immigrant's shopping); likewise, if the id card number of tourists is represented by their passport numbers, the program would need to cope with different codifications for different countries.

That's why, in most cases, a primary key is implemented in software with a specific identification, which is generally called an *id*. Its value is usually a number taken from a sequence, or a random number or String.

When using acceptance tests, we also need primary keys to create references to Business Objects. Suppose we were testing the sales program of the abovementioned department store. At some point in the script, a method `createCustomer` is used in a test (say, inside a **Creator** pattern). The customer's name can't be used as a primary key, so an internal id must be assigned to the customer, like in the usage example below:

```
createCustomer name="John Doe" birthDate=09/12/1975 id=1435728
createCustomer name="Mary Doe" birthDate=05/23/1977 id=2693913
```

Whenever customers are referenced in the script, their id is used:

```
expect "John Doe" getCustomerName id=1435728
expect "Mary Doe" getCustomerName id=2693913
```

But wait … Could these tests be any fuzzier? If you need to manipulate 10 customers in the tests, you have to memorize all ids? In addition to the meaninglessness of these numbers, especially to the client, the mechanism of id attribution, random or not, is a low-level, programmer concern. Suppose programmers realize that they need to change the way ids are generated and represented. Depending on the type of change, all tests may have to be modified.

The solution for this is to hide the id attribution mechanism from the tests by using script variables – test writers use variables to store Business Object references in the form of ids generated by the program. The mechanism employed to generate the unique ids is up to the programmers. The only requirement on their part is to return the generated ids in the methods that create the Business Objects. In a program using a database to store data persistently (most do), the database itself can provide the unique key value. The programmer only has to return this reference. Let's apply the pattern on the example above to clarify things:

```
id1=createCustomer name="John Doe" birthDate=09/12/1975
id2=createCustomer name="Mary Doe" birthDate=05/23/1977
```

The ugly numbers are gone. Now, when referring to the customers:

```
expect "John Doe" getCustomerName id=${id1}
expect "Mary Doe" getCustomerName id=${id2}
```

The variable names are `id1` and `id2`, but could be `john` or `mary` as well – anything that makes the script easier to read. When the tool runs, it replaces ${ } with the content of the variable enclosed (this is not actually seen in the script):

```
expect "John Doe" getCustomerName id=748239568492890543
expect "Mary Doe" getCustomerName id=178423748923789024
```

BORef (short for **Business Object Reference**) can also be used when creating Business Objects that don't use an attribute-based primary key in the final program. For example, in Alice's poll program, she will never choose a poll by the id assigned to it or use that id in a report or anywhere. However, as you can see in the tests of chapter 3, the pattern **BORef** has been used because different polls must be referenced throughout the script.

A witty reader may argue that there is another way of providing Business Object references: why don't you simply assign objects (program entities) to variables? That way, the programmers wouldn't need to bother coding any support mechanism … Well, this could be a solution, but the problem with it is that clients would now have to deal with object orientation and we have tried to avoid that. As this text is being written, EasyAccept doesn't have this feature, but it may be an interesting user story to incorporate in the future.

Pattern outline

Context: you are writing tests for Business Objects that need to be uniquely referenced in the script.

Problem: assigning primary keys explicitly in a script introduces "magic numbers" that make the script hard to follow. Furthermore, Business Objects that don't require explicit ids still need to be referenced in a script.

Forces: primary key attribution mechanisms are the concern of developers; assigning objects (software concept) in the test script relieves developers from coding id attribution mechanisms, but require clients to deal with object orientation, a concept not likely to be familiar to them.

Solution: use script variables in a test script to refer to Business Objects; the value of the variable is the result of a unique id attribution chosen by the program.

Rationale: the actual id attribution remains hidden from the test script and is now solely a concern of programmers, as it should be.

Related Patterns: **BORef** is often used in conjunction with **Creator**.

Only Business Objects

ATDD emphasizes the need for understandability of tests. However, the concept of understandability differs between clients and developers. Examine the two tests below, for instance:

```
expect 25 getBTreeFatherNodeIndex
expectError "Unexpected middleware fault" getRMISkeletonMiddlewareProperty
```

Even though they are written in EasyAccept test format, one must be a developer to understand what they refer to – a B-Tree is a kind of hierarchical data structure for programs, and an RMI (Remote Method Invocation) Skeleton is a middleware (communication support software) structure.

The problem with such tests, however, is not only the jargon, but also the mismatch of testing concerns. Generally, acceptance tests involve problem domain concerns in which the client's opinion is of utmost importance. The choice of data structures or middleware is generally not a concern of the client because he doesn't even know what they are. Even if he does, does he care how developers implement the program, as long as all acceptance tests pass?

One may argue that some low-level choices may affect non-functional requirements, like performance, and thus could be the concern of the client. If this is the case, the best thing to do is to create performance tests at the acceptance test level (which can be understood by the client) and leave implementation choices to developers.

Ok, but what happens when clients are programmers themselves and thus understand all tests? Even so, there is still the argument of different test concerns. What happens when you mix problem domain testing with implementation details? Tests become a mess, hard to understand and maintain, which generates fear of changing the tests and ultimately a program with lower quality. Furthermore, mixing test levels creates coupling, forcing tests to change when internal implementation details change. Remember that, in the programming world, "coupling" is a four-letter word.

The solution to this problem is quite simple. Only Business Objects should be exposed and manipulated in an acceptance test script. Whenever you find a non-Business Object (B-Tree? Skeleton?), it probably is not a client concern and should be treated in a lower level as a unit test.

Observe that this is not affected by the tools you use. EasyAccept, an acceptance testing tool, can be used to create unit tests, as the example above suggests. Likewise, any unit testing tool like the xUnit family can be used to create acceptance tests, provided that the client understands Java or the corresponding programming language.

In chapter 3, you saw an example of the application of **Only Business Objects**. During the development of the poll program, Bob felt the need to test a support structure he needed: a converter of strings to and from collections. This structure needed to be tested because if there were a bug in it, acceptance tests would start to break even if the problem domain logic was correct.

The detailed discussion of this step, including the corresponding unit tests that Bob created, can be found in appendix II.

Pattern outline

Context: non-business objects are included in the tests.

Problem: when non-business objects are included in the script, the tests related to them become unintelligible to the client, who typically is not a technical person and doesn't understand or care about non-business objects.

Forces: when clients don't understand tests, their commitment to keep reviewing tests is lowered; testing non-business objects is important and must be done as much as the testing of business objects; even if clients understand tests, unit and acceptance tests are used at different testing levels.

Solution: only business objects should be tested in an acceptance test script. Remove all tests for non-BOs and make them unit tests instead.

Rationale: Non-BOs are still tested (in TDD, everything must be tested) but, as they are not a client concern, they are hidden from him.

Client Assertion

Bob was tackling user story 4 of the Monopoly game and found the following test sequence:

```
 1 createGame numberOfPlayers=2 playerNames={john,mary} \
 2   tokenColors={black,white}
 3 expect john getCurrentPlayer
 4 expect false playerIsInJail playerName=john
 5
 6 rollDice firstDieResult=5 secondDieResult=5
 7 expect 10 getPlayerPosition playerName=john
 8 expect john getCurrentPlayer
 9 expect false playerIsInJail playerName=john
10
11 rollDice firstDieResult=4 secondDieResult=4
12 expect 18 getPlayerPosition playerName=john
13 expect john getCurrentPlayer
14 expect false playerIsInJail playerName=john
15
16 rollDice firstDieResult=3 secondDieResult=3
17 expect 24 getPlayerPosition playerName=john
18 expect mary getCurrentPlayer
19 expect true playerIsInJail playerName=john
```

It tests the rule that sends a player to jail when he throws doubles three times in a row. Looking at line 17, he realizes he may have come across a test bug. The user story clearly states that, when a player throws doubles three times in a row, his turn immediately ends, before the token advances. "How come John ends up at board position 24 after throwing double 3's?" He is not entirely sure why that line was included. "Is this a test bug, or did Alice tell me a different rule that I don't remember?"

Suppose Bob decided this was not a bug, after all, and simply left the tests as they are (when actually line 17 *is* a bug[8]). The fact that a player may complete his turn before going to jail may change the outcome of the game. This is not what Alice wanted.

On the other hand, suppose that Bob decided to change line 17, but that this was not a test bug. Alice really asked him to change the rule, so that players are only sent to jail after completing their turn. In both cases we have the same problem: divergence between actual requirements and tests.

Every time Bob doubts over a test or a requirement, he uses the pattern **Client Assertion**. It simply consists in asking Alice for a clarification and committing the clarification to the tests. This closes the communication gap that could be introduced in the process, resulting in software bugs.

Of course, one must use common sense in using the rule. Having the client assert every immediately obvious test bug slows down the pace of development. Furthermore, in cases

[8] The correct line 17 would have expected 30 as the player's token position, as this is the board position of the Jail.

where clients are not always available, a non-critical doubt may be left aside for some time until it can be clarified (but developers must mark it in the script with comments or using some other mechanism).

The application of **Client Assertion** has the additional advantage of promoting discussion during software development. It forces clients to examine tests and reflect over requirements, and makes developers focus their attention on the clients' needs. This rule is broken by developers all the time because they *think* they know what the client wants or needs. Frequently, they

Rationale: **Client Assertion** avoids developers introducing errors in the test suite and can serve as a means of making the client review the tests.

Related patterns: **Client Assertion** should always be used in conjunction with **Commmentor**.

Commentor

Acceptance tests must be created with understandability in mind, in order to successfully involve clients in the game. Be that as it may, test logic is frequently far from simple and may even become convoluted. Sometimes this is due to lack of simplicity values on the part of analysts, who create overly complicated tests, but other times the complexity is inherent to the test itself – it is intricate, involves lots of interrelated rules or interdependent sequences of actions.

If the test is inherently complex, there's not much that can be done to improve its understandability other than explain in detail what it does. However, when the understandability problem stems from bad tests, some patterns can be used to help refactor them. **Single Tester** and **Summarizer** are patterns used for this purpose and will be detailed in the next section.

The pattern **Commentor** advocates that comments help improve understandability in scripts and are fundamental to establish effective communication among developers and between developers and clients. Requirements inevitably change during software development, and so must tests. **Commentor** helps you keep them in sync.

The pattern requires that tests be commented when they are created and suggests particular situations when comments should be improved or expanded. **Commentor** states that comments should be added to the script to enhance the comprehension of a test in the following situations:

1) a test has been modified;
2) a test is difficult to understand (inherently complex) or insufficiently commented;
3) the pattern **Client Assertion** has been used (even if no tests are modified);

Situation #1 requires **Commentor** because otherwise comments and test become inconsistent. In situation #2, improving comments may be the only way to increase test understandability, because of the inherent complexity involved. In situation #3, the application of the **Client Assertion** pattern itself may already result in a test update, which justifies the co-application of **Commentor**. But, even if it does not (for example, a potential bug was actually a misunderstanding), a comment on the clarification should be included in the tests so that other people will not have the same wrong interpretation or the same doubts when reading the test. Think of it this way: if the tests were not clear enough to start with and required clarification, then whatever was clarified should be added to the test so that a future developer won't need to clarify again.

A simple example of situation #1 in the Monopoly game follows:

```
1 # Testing Income Tax; Alice begins with $1000 and should lose $200
2 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
3   playerTokens={white,black}
4 rollDice firstDieResult=2 secondDieResult=2
5 expect 800 getPlayerCash playerName=Alice
```

In this test, the income tax rule is tested. When a player's token falls on income tax, the player should lose $200. In the test case, Alice begins the game with $1000, falls on income tax (position 4 on the board) and ends up with $800.

Now suppose the income tax rule changes to increase the tax value to $300. Bob updates the test but overlooks the need (or simply forgets) to update the comment. What happens now? Another programmer (or even Bob himself, at another opportunity) will be confused when he reads the test. Is there a bug in the tests, or a requirement has changed and the tests were not updated? Remember that, in ATDD, acceptance tests are not only tests but also a representation of requirements themselves.

Now see a simple example of situation #2. Analyze the following test.

```
createGame numPlayers=2 playerNames={player1,player2} \
  tokenColors={black,white}
setPlayerCash player=player1 cash=1000
setPlayerCash player=player2 cash=3000
rollDice firstDieResult=2 secondDieResult=3
rollDice firstDieResult=2 secondDieResult=3
expect 925 getPlayerMoney playerName=player1
expect 2875 getPlayerMoney playerName=player2
rollDice firstDieResult=5 secondDieResult=5
rollDice firstDieResult=5 secondDieResult=5
expect 1225 getPlayerMoney playerName=player1
expect 2375 getPlayerMoney playerName=player2
rollDice firstDieResult=5 secondDieResult=5
rollDice firstDieResult=5 secondDieResult=5
expect 1775 getPlayerMoney playerName=player1
expect 1625 getPlayerMoney playerName=player2
rollDice firstDieResult=5 secondDieResult=5
rollDice firstDieResult=5 secondDieResult=5
expect 2575 getPlayerMoney playerName=player1
expect 625 getPlayerMoney playerName=player2
quitGame
```

Can you tell what this sequence tests? Now read its commented version and see for yourself how much clearer it becomes.

```
# testing railroads; a player that falls on a railroad must pay
# the owner the result of the dice throw times the number of railroads
# the owner has in his possession times $25
createGame numPlayers=2 playerNames={player1,player2} \
  tokenColors={black,white}
setPlayerCash player=player1 cash=1000
setPlayerCash player=player2 cash=3000
# player1 falls on Reading Railroad and buys it automatically ($200)
# player2 falls on it and pays $125 (dice throw 5 x 1 railroad x $25)
rollDice firstDieResult=2 secondDieResult=3
rollDice firstDieResult=2 secondDieResult=3
expect 925 getPlayerMoney playerName=player1
expect 2875 getPlayerMoney playerName=player2
# player1 falls on Penns. Railroad and buys it automatically ($200)
```

```
15 # player2 falls on it and pays player1 $500 (10 x 2 railroads x $25)
16 rollDice firstDieResult=5 secondDieResult=5
17 rollDice firstDieResult=5 secondDieResult=5
18 expect 1225 getPlayerMoney playerName=player1
19 expect 2375 getPlayerMoney playerName=player2
20 # player1 falls on B & O Railroad and buys it automatically ($200)
21 # player2 falls on it and pays player1 $750 (10 x 3 railroads x $25)
22 rollDice firstDieResult=5 secondDieResult=5
23 rollDice firstDieResult=5 secondDieResult=5
24 expect 1775 getPlayerMoney playerName=player1
25 expect 1625 getPlayerMoney playerName=player2
26 # player1 falls on Short Line Railroad and buys it automatically ($200)
27 # player2 falls on it and pays player1 $1000 (10 x 4 railroads x $25)
28 rollDice firstDieResult=5 secondDieResult=5
29 rollDice firstDieResult=5 secondDieResult=5
30 expect 2575 getPlayerMoney playerName=player1
31 expect 625 getPlayerMoney playerName=player2
32 quitGame
```

The third situation where you must use the pattern in when you apply **Client Assertion**. An example of it can be found in the discussion of that pattern.

Pattern outline

Context: test scripts are difficult to understand or will be refactored.

Problem: comments are becoming inconsistent with the associated tests; tests are difficult to understand.

Forces: comments in the tests serve an important communication role between clients and developers, as they further clarify the script, an artifact that reconciles the languages of both.

Solution: add explanatory comments to tests that are hard to understand in the script; update comments whenever a test changes, doubts are clarified or requirements and business rules evolve.

Rationale: comments are an integral part of the test suite and serve as an additional means of communicating how the program should behave. They improve tests understandability.

Related patterns: **Client Assertion** is always used in conjunction with **Commentor**.

Summarizer

Repetition is very common in acceptance tests. Although redundancy is sometimes necessary to improve test's understandability, most often it is a source of distraction to the reader. The pattern **Summarizer** tackles this problem by diverting repeated test sequences to separate scripts or by hiding them into special preparer commands.

Let's see an example to figure things out. Suppose that, when testing Monopoly, Bob needs to analyze a series of special situations that have in common the fact that the game has many players and is in an advanced stage, i.e., players have cycled through the board multiple times so that all deeds for properties and railroads belong to them.

Good testing practices advocate that test cases should not depend on each other (see the pattern
Single Tester). Thus, for each single test case, Bob has to issue a series of shortcut commands – he devised the `giveDeedToPlayer` command – to prepare the scenario for the special tests (this preparation represents the build phase of **Test Flow**, one particular place where repetition plagues the scripts).

Such a sequence follows:

```
createGame numPlayers=8 playerNames={p1,p2,p3,p4,p5,p6,p7,p8} \
  tokenColors={black,white,red,green,blue,yellow,orange,pink}
giveDeedToPlayer playerName="p1" deed="St.Charles Place"
giveDeedToPlayer playerName="p1" deed="New York Avenue"
giveDeedToPlayer playerName="p1" deed="Park Place"
giveDeedToPlayer playerName="p2" deed="Baltic Avenue"
giveDeedToPlayer playerName="p2" deed="Kentucky Avenue"
giveDeedToPlayer playerName="p2" deed="Marvin Gardens"
giveDeedToPlayer playerName="p3" deed="States Avenue"
giveDeedToPlayer playerName="p3" deed="Pacific Avenue"
giveDeedToPlayer playerName="p3" deed="Boardwalk"
...
giveDeedToPlayer playerName="p8" deed="Connecticut Avenue"
giveDeedToPlayer playerName="p8" deed="Tennesse Avenue"
giveDeedToPlayer playerName="p8" deed="Ventnor Avenue"
```

In the sequence, a game with 8 players is created and deeds are equally distributed among the players. Having to repeat this sequence for each test case is just not nice, even if you copy/paste it all over the script. So, applying **Summarizer**, what does the script become? The way Bob chose was to create a different script, say `advancedGameBuild.txt`, and whenever he needs to perform the build in a script, he runs it.

In EasyAccept, the way to do it is to use the built-in command `executeScript`. This command is used to run a script from within another, after pausing the current script's execution. It gives the user the option of running the new script in a new thread of execution (with the argument `newThread`), but that will not be necessary in the example we are discussing. If you need an example of the usage of `executeScript` with threads, please refer to EasyAccept's manual in appendix I.

Bob's test cases become much cleaner after **Summarizer**, observe:

```
# Test case 1
executeScript newThread=false advancedGameBuild.txt
# rollDice, expect, etc. follow

# Test case 2
executeScript newThread=false advancedGameBuild.txt
# rolDice, expect, etc. follow
```

For every test case that uses the build, a simple reference to `advancedGameBuild.txt` is made, reducing the number of lines that appear in the script. Moreover, should Bob need to modify the build, there's only one place to change.

You can use **Summarizer** in an additional way. Instead of placing the test sequence in a separate script, you can simply delegate it to the program's code by creating a script command. This way, you don't need to use `executeScript`. In the Monopoly example, Bob could create a command called `buildAdvancedGame` and use it in the script, like below:

```
# Test case 1
buildAdvancedGame
# rollDice, expect, etc. follow

# Test case 2
buildAdvancedGame
# rolDice, expect, etc. follow
```

Using this approach, the build sequence is diverted to the program's code. In addition to summarizing and isolating the sequence, which simplifies future updates, an advantage of it over the `executeScript` approach is that arguments can be used to tailor the sequence for specific needs. For example, the `buildAdvancedGame` command could have an argument for the number of players you want the game to be created with, or an argument to state whether or not you want railroads and utilities' deeds to be distributed, in addition to properties' deeds.

The downside of the approach is that the sequence becomes "inaccessible" to the client (in the sense that clients don't understand code); thus, programmers alone must assure no test bugs are found; additionally, it only makes sense to summarize *build* and *operate* phases of a **Test Flow**, not *check* phases (the various `expect` and `equalFiles` commands), unless you treat the check phase as a unit test (see the proper acceptance vs. unit test discussion in appendix II).

Pattern outline

Context: too many repeated tests are found in the script.

Problem: repetition of the same test sequence in multiple tests hinders understanding and makes the scripts harder to maintain.

Forces: test sequences can often be summarized in a single expressive command without hindering the understanding the remainder of the test.

Solution: when multiple tests use the same test sequence, set it aside in a separate script and have all tests that use it refer to this script. Alternatively, create a shortcut command if you

need to hide the test sequence content from the script (only developers will have access to what it does, in the program code).

Rationale: tests become cleaner and easier to follow when script repetition is dealt with.

Related patterns: **Table Tester** also deals with script repetition, but in circumstances where test sequences have the same structure, differing only in the values.

Single Tester

One important principle of test design states that tests should be independent from one another. Each test should focus on only one business rule, or even part of it. The reason is the need to cope with test change, which is directly linked to requirements change. If two business rules are tested together and one of them changes, the other test will need change as well.

The **Single Tester** pattern is used in such a situation. Whenever you find a test that involves multiple business rules, refactor it in a way that results in each test referring to a single business rule. Consider, for example, the following cumbersome Monopoly script that tests at the same time the income tax business rule, the salary rule and one of the community chest cards (Lose $15):

```
 1 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
 2   playerTokens={white,black}
 3 # Alice falls on income tax and loses $200
 4 rollDice firstDieResult=1 secondDieResult=3
 5 expect 1300 getPlayerCash playerName=Alice
 6 rollDice firstDieResult=4 secondDieResult=6
 7 # Alice is put on #35 and falls on #2; she gets a new salary for having
 8 # cycled the board, but receives the CC card to lose $15
 9 setPlayerPosition playerName=Alice position=35
10 rollDice firstDieResult=3 secondDieResult=4
11 giveCommunityChestCardToPlayer card="Lose $15"
12 expect 1485 getPlayerCash playerName=Alice
```

What is the problem with this test case? Suppose that Alice decides that position #2 on the board will no longer be a community chest place. In order to test the rule that makes a player lose $15 when getting the “Lose $15” community chest card, Bob will have to change the test, making the player fall on the new community chest position, say, position #13. Doing so, however, interferes with the salary test, now that the player won’t cycle the board.

Another situation: what if the income tax rule changes? For example, if its value drops from $200 to $150. Alice’s cash will need recalculation at line 12 in addition to line 5, which makes no sense, since line 12 should have nothing to do with income tax testing.

This was an example of a bad test case that needs refactoring. For each test, there should be only one business rule, even if the total number of lines is greater. See how the reworked script becomes cleaner. First, the income tax test:

```
1 # testing the income tax rule
2 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
3   playerTokens={white,black}
4 # Alice falls on income tax and loses $200
5 rollDice firstDieResult=1 secondDieResult=3
6 expect 1300 getPlayerCash playerName=Alice
7 quitGame
```

Now the salary rule:

```
1 # testing the salary rule
2 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
3   playerTokens={white,black}
4 # Alice is put on #35 and cycles past #40 (Go!) to get a salary
5 setpPlayerPosition playerName=Alice position=35
6 rollDice firstDieResult=2 secondDieResult=4
7 expect 1700 getPlayerCash playerName=Alice
8 quitGame
```

Finally, the community chest card rule:

```
1 # testing the salary rule
2 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
3   playerTokens={white,black}
4 # Alice falls on a Community Chest and receives a card that takes $15
5 rollDice firstDieResult=1 secondDieResult=1
6 giveCommunityChestCardToPlayer card="Lose $15"
7 expect 1485 getPlayerCash playerName=Alice
8 quit
```

Observe that, even though there are considerably more test lines, they are now clearer and independent from one another. Changes in one of the business rules now won't affect the tests of the others.

Pattern outline

Context: the script includes tests that evaluate more than one business rule at once.

Problem: a change in one of the business rules interferes with the tests of the others.

Forces: you may have no choice but to make tests dependent when they are inherently complex.

Solution: try to refactor tests on a one business rule to one test basis, so that they become independent.

Rationale: when tests are isolated, they can be updated without fear interfering with other tests.

Related patterns: in order to refactor tests, you need to use **Client Assertion**.

Template Generator

Suppose the development of Monopoly is well under way. Bob has completed a number of user stories and Alice can already play the game with most of the rules.

Bob then invites Alice for a Monopoly gaming session. He starts a new game through a tentative user interface with rudimentary dialog boxes and a low-resolution image representing the board.

“This board is ugly”, says Alice. “Don’t worry”, Bob replies, “this is not the final user interface. I guarantee you will have a beautiful board in the end to play with”.

“Let’s see”, he continues. “Two players. Name of the first one: Alice. Which token color do you want?”

“Pink”, she replies. “Pink it will be. For me, Bob, token color is blue … Ok, you begin.”

Alice starts playing by clicking on a button labeled “Roll Dice”. A message is shown stating that the result of the dice throw is 4 (1 + 3). “Bad luck, Alice. You fell on income tax and just lost $200. That’s how much should lose, right?”. “Yes”, she answers. “Let’s see if you really lost $200. Click on the “Status” button.” A dialog box appears showing that Alice now has $1300.

Now it’s Bob’s turn. He clicks on the “Roll Dice” button and gets 10 (5 + 5) as a result. He falls on “Jail – Just Visiting”, and nothing happens. As he threw doubles, he gets an extra turn. Luckily for him, he throws doubles once again 10 (5 + 5)! But that makes him fall on the Free Parking, and again nothing happens.

“Wait!”, says Alice, “There’s so many *do nothing* places in the board ... I just got an idea for a new rule that I want you to put in the game.” She then explains the “jackpot” rule. Whenever a player falls on the Free Parking, he should receive a certain amount of cash, called the jackpot, which corresponds to 5% of the cash of the player who has the most cash in the moment.

Bob writes down the description of this new user story so that he can implement it later. The game continues. Bob rolls dice one more time, the result is 5 (2 + 3). He falls on B & O Railroad and buys it automatically. Now it’s Alice’s turn. From the income tax (place #4), she rolls dice and gets a 6 (3 + 3). She falls on “Jail – Just Visiting” and gets and additional throw. She rolls dice once again and gets a 7 (4 + 3), falling on the second Community Chest place. As a result, a dialog box appears showing the card that was drawn: “Advance to the nearest railroad.” Her token is placed on place #25, B & O Railroad, whose owner is Bob.

“Now this is a situation I’ve never seen”, says Bob. He laughs and says “How much should you pay me, since you arrived at the railroad by teleportation?”

Alice clarifies stating that she should only pay for the results of her last throw: 7 times $25 = $175, given that Bob is the owner of a single railroad. After her turn, Alice should remain with $1300 – 7 x $25 = $1125. She reckons the result, and Bob writes it down. There was not

a test for this case, and Bob feels the program is probably wrong, which he confirms in the game's status: Alice now has $925.

Bob stops the game at this point because he needs to analyze the script that resulted.

During the entire game session, Bob was applying the pattern **Template Generator**. Before Alice arrived, he coded a background mechanism to record every action that was taken in the game, along with the player's cash values. Thus, a script was generated in the end. By noting down Alice's comments and replies during the game, Bob can now process the script that was generated automatically and from it write relevant new tests. This processing is needed because Alice must assert the templates (expected values and errors) that were generated. Let's analyze the example to clear this out. This was the script generated by the background mechanism (observe that other game's properties could be included, but we chose to show you only the cash so that the script didn't end up cluttered):

```
 1 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
 2   playerTokens={white,black}
 3 rollDice firstDieResult=1 secondDieResult=3
 4 expect 1300 getPlayerCash playerName=Alice
 5 expect 1500 getPlayerCash playerName=Bob
 6 rollDice firstDieResult=5 secondDieResult=5
 7 expect 1300 getPlayerCash playerName=Alice
 8 expect 1500 getPlayerCash playerName=Bob
 9 rollDice firstDieResult=5 secondDieResult=5
10 expect 1300 getPlayerCash playerName=Alice
11 expect 1500 getPlayerCash playerName=Bob
12 rollDice firstDieResult=2 secondDieResult=3
13 expect 1300 getPlayerCash playerName=Alice
14 expect 1300 getPlayerCash playerName=Bob
15 rollDice firstDieResult=3 secondDieResult=3
16 expect 1300 getPlayerCash playerName=Alice
17 expect 1300 getPlayerCash playerName=Bob
18 rollDice firstDieResult=4 secondDieResult=3
19 giveCommunityChestCardToPlayer card="Advance to nearest railroad."
20 expect 925 getPlayerCash playerName=Alice
21 expect 1675 getPlayerCash playerName=Bob
```

The values indicated in lines 20 and 21 are wrong and should be $1125 and $1475. Based on this "raw" test script, Bob applies **Single Tester** and comes up with a useful test for the railroad teleportation, which he had overlooked.

```
 1 createGame numberOfPlayers=2 playerNames={Alice,Bob} \
 2   playerTokens={white,black}
 3 # We give B & O Railroad to Bob and put Alice on Jail - Just
 4 # Visiting, so that she falls on Community Chest next
 5 giveDeedToPlayer playerName=Bob deed="B & O Railroad"
 6 setPlayerPosition playerName=Alice position=20
 7 rollDice firstDieResult=4 secondDieResult=3
 8 giveCommunityChestCardToPlayer card="Advance to nearest railroad."
 9 # Alice should pay Bob $175 = 7 (dice throw) x 1 railroad x $25
10 expect 1325 getPlayerCash playerName=Alice
11 expect 1675 getPlayerCash playerName=Bob
```

Template Generator helps finding new tests, promotes discussion on current features, allows clients to think about new ideas for features. Moreover, **Template Generator** helps in the validation of the software's user interface. Having a user feel the interaction with the software is a good opportunity to find out whether or not the UI works for him.

Pattern outline

Context: development is under way and partially working software is available; you need to find more test cases.

Problem: as development progresses, it becomes harder to find test cases other than the more direct examples of software functions.

Forces: automation speeds up test creation; software usage by the client can reveal bugs that would otherwise not be considered; however, providing a recording mechanism requires extra effort by the developers.

Solution: have the client or end user operate the partially working software and provide a background mechanism to automatically generate a test script based on his actions (by recording the sequence of actions). When the client is done with a given operation, he examines the results that were presented and either accepts or rejects it. This result then function as a template for a test consisting in the sequence of actions performed by the client.

Rationale: the pattern provides an automated way of capturing new tests from the client.
Some of these tests could be overlooked if you only generate tests manually.

Appendix I

stackTrace	Built-in command which is used to obtain a stack trace when debugging. This is useful when unexpected exceptions occur and one wishes a stack trace to see what is happening.
quit	Built-in command to quit EasyAccept.
executeScript	Built-in command that executes a given script. The current script's execution is paused until the new script executes to completion. Either the current thread or a new thread taken from the thread pool may be used to execute the new script.
repeat	Built-in command that is used to repeat a given command's execution a specific number of times.
threadPool	Built-in command that creates a thread pool to execute scripts.
echo	Built-in command that returns the concatenation of its parameters.

Examples

Executing business logic

A script is written in a text file. A command is written on a single line. A command is simply executed. For example:

```
createUser id=id1 name="John Doe" birthdate=1957/02/04
```

will simply call the `createUser` business method passing three parameters to it.

Checking business logic execution

Special built-in commands can be used to check that a (business logic) command worked correctly. For example:

```
createUser name="John Doe" birthdate=1957/02/04

expect "John Doe" getUserName key=key1

expectWithin .01 2345.67 getSalary key=key1
```

In the above two lines, the first line calls a business logic command. EasyAccept will accept that it has functioned correctly if it does not produce an error (an Exception, in programmer parlance). The next line also calls business logic (`getUserName` with parameter `key1`) but checks that it returned the string "John Doe". The third line checks that the salary is correct, with a precision of one cent.

Using variables

It is sometimes necessary to obtain the result returned by a command in order to use it in the test script. For example, suppose that the `createUser` command chooses a record key internally (say a database OID) that is unknown to the tester. The following script shows how to deal with the situation using variables (assuming that `createUser` returns the chosen record key):

```
key=createUser name="John Doe" birthdate=1957/02/04

expect "John Doe" getUserName key=${key}
```

The syntax `${varName}` is substituted by the variable's value.

The scope of a variable is the set of scripts being executed, that is, from the time of variable definition until the end of the current EasyAccept execution.

Here is another example that checks whether two users with the same attributes generated different keys in the database:

```
key1=createUser name="John Doe" birthdate=1957/02/04

key2=createUser name="John Doe" birthdate=1957/02/04

expectDifferent ${key1} echo ${key2}
```

Checking error conditions

A special built-in command can be used to check that a (business logic) command produces an error (using an exception). For example:

```
expectError "Unacceptable date." createUser name="John Doe"
birthdate=1957/02/20
```

In the above line, a business logic command is called `createUser`. EasyAccept will accept that it has functioned correctly if it produces an error (an `Exception`, in programmer parlance) and if the `Exception's` error message is "Unacceptable date."

Checking voluminous output

When you want to use the expect built-in command but the string to be checked is large, it may be better to leave the string in a text file and have the business logic command produce output in another file. Then, the built-in command `equalFiles` can be used to check the command's output.

```
# this shows that John Doe exists
expect "John Doe" getUserName id=id1
produceReport id=id1 outputFile=rep.txt
equalFiles file1=expected-report.txt file2=rep.txt
```

In the above example, the command `produceReport` will produce a report concerning John Doe and the report will be left in file `rep.txt`. The next line checks that the `rep.txt` file is equal to the `expected-report.txt` file. This last file (the expected report) should be produced beforehand (by hand, for example) and should contain exactly the output desired for the `produceReport` command.

Executing scripts

With EasyAccept you can execute a script from within another. The current script’s execution pauses until the new script executes to completion.

```
# runs the script script2.txt from within the current script using the
current thread
executeScript newThread=false scriptFile=script2.txt
```

If you want the new script's execution to be run in a new thread, you can flag the argument `newThread` with true. EasyAccept takes a new thread from the thread pool, which must have been previously created with the threadPool command.

```
threadPool poolSize=5
# runs the script script2.txt in a new thread taken from the thread pool
executeScript newThread=true scriptFile=script2.txt
```

Debugging

When any command produces an unexpected exception and you would like to examine a stack trace of the situation that led to the exception, use the `stacktrace` command as shown below.

```
# the following command produces an exception
someCommand param=someValue
```

In this case, in order to see details of the exception that was produced, temporarily use the following command during debugging:

```
# the following command produces an exception
stackTrace someCommand param=someValue
```

Miscellaneous commands

The `repeat` command can be used to execute a given command a specific number of times. The `quit` command closes EasyAccept.

```
resetPlayerScore
# adds 6x200 points to the player's score
repeat numberOfTimes=6 playerScores points=200
expect 1200 getPlayerScore
# quits EasyAccept
quit
```

The language

A script resides in a file. Each line consists of name=value pairs. The following are examples of acceptable name=value pairs:

```
name=value
name
name=
name=""
=value
```

In the second and third cases, the value is null; in the fourth case, the value is empty; in the fifth case, there is no name.

In a line, the first name-value pair represents a command to be executed (the name) and typically does not provide a value. The name of the command must match a method available in the business logic. Parameters are passed as given in the other name=value pairs. In reality, since Java does not provide parameter names through reflection, the order of the parameters in the test must match the order in the business logic. The names themselves are not used and serve only as documentation in the tests. *Remember that the parameter order is important.* A method appropriate to the parameter types given will be found in the business logic. The following parameter types are acceptable and automatic conversion will be provided from a string to the parameter value of appropriate type:

`String`, `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`

A line starting with # is a comment. The line continuation character is \. A \ character itself must be given as \\.

The default string delimiter is ". This may be changed (see below).

Built-in commands

EasyAccept has several built-in commands used to perform special testing actions. They are described below.

echo

```
echo any string
```

This command returns the concatenation of its arguments. It is typically used to examine command results, variable names, etc.

equalFiles

```
equalfiles OKfile fileToTest
```

This command is a test of file contents. It receives two files, and the test passes if the two files have identical contents. By convention, the first file contains the correct (expected) output and the second file contains the file to be tested. This command is typically used after a business logic command that has produced its output in a file. There are two ways of using `equalfiles` command:

1) With Relative path (relative to directory where EasyAccept is executed)

```
equalfiles                          ./src/easyaccept/script/test/script1.txt
./src/easyaccept/script/test/script1.txt
```

2) With AbsolutePath:

```
equalfiles c:/projetos/script1.txt c:/script2.txt
```

expect

```
expect expectedString businessLogicCommand paramName=paramValue ...
```

This command executes the `businessLogicCommand` passing the specified parameters. The `businessLogicCommand` must return a string, which is compared with `expectedString`. The test passes if the strings are equal. No errors (exceptions) may occur.

expectDifferent

```
expectDifferent stringNotExpected businessLogicCommand paramName=paramValue
...
```

This command executes the `businessLogicCommand` passing the specified parameters. The `businessLogicCommand` must return a string, which is then compared with `expectedString`. The test passes if the strings are different. No errors (exceptions) may occur.

expectError

```
expectError expectedErrorString businessLogicCommand paramName=paramValue
...
```

This command executes the `businessLogicCommand` passing the specified parameters. The `businessLogicCommand` must return an error (produce an exception, in Java) using an error string. This string is compared with `expectedErrorString`. The test passes if the strings are equal. If no exceptions are thrown, the test does not pass.

expectWithin

```
expectWithin precision expectedValue businessLogicCommand
paramName=paramValue ...
```

This command executes the `businessLogicCommand` passing the specified parameters. The `businessLogicCommand` must return a value of type double, which is compared with `expectedValue`. The test passes if the values are equal, within the given precision. No errors (exceptions) may occur.

stackTrace

```
stackTrace <any other command, including built-in commands>
```

This command executes the command indicated and, if an exception is thrown, a full stack trace is printed. This is useful for debugging and will not normally be used permamently in scripts.

stringDelimiter

```
stringDelimiter delimiter_character
```

This command changes the string delimiter to the given character. By default, the string delimiter is ".”

quit

```
quit
```

This command ends EasyAccept execution.

executeScript

```
executeScript newThread scriptFile
```

This command is used to execute a script from within another. The new script can either be executed in a new thread (first parameter=”true”) or using the current script’s thread (first parameter=”false”). The new script is executed using a subroutine return mechanism.

repeat

```
repeat numberOfTimes anyCommand
```

This command is used to execute a script command a given number of times.

threadPool

```
threadPool poolSize
```

This command creates a thread pool with a definite pool size that EasyAccept uses to execute scripts concurrently.

Instructions for the Programmer

In order to expose the business logic of your program, you must write a façade. EasyAccept will instantiate the façade once and all public methods contained in the façade will be callable from a test script. Remember to separate the business logic from the user interface. Your façade should not print anything anywhere. It should communicate itself with the outside by accepting parameters, returning results or throwing exceptions. Parameters and return values cannot be objects.

In order to call EasyAccept to test a program, use the following syntax:

```
java -classpath ... easyaccept.EasyAccept <FacadeClass> <scriptFile>
[<scriptFile>] ...
```

where:

- <FacadeClass> - corresponds to the full class name (ex: easyaccept.script.TestFacade) of facade of he system to be tested.
- <scriptFile> - corresponds to the full name of a file or a directory. If the argument is a file, it can have any extension and should be composed by test commands written in EasyAccept script language. However, if it is not a EasyAccept script, EasyAccept will

still try to execute each line of the file – resulting in many errors. If the argument is a directory, EasyAccept will look for all files in this directory and its subdirectories, include them in a queue and execute all test commands of each file found.

A EasyAccept exit code of 0 implies that all tests have passed.

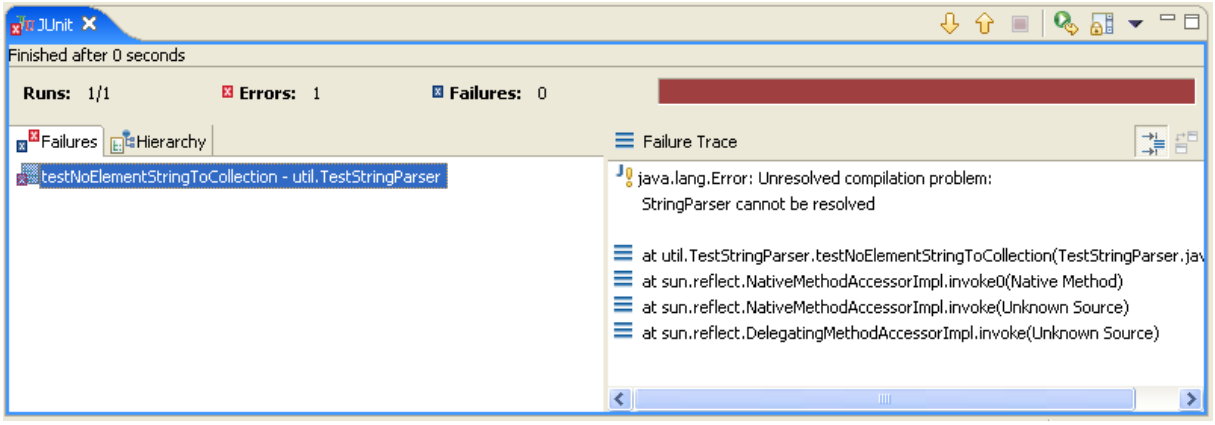
Appendix II

```
package util;

import junit.framework.TestCase;
import java.util.Collection;
import java.util.ArrayList;

public class TestStringParser extends TestCase {

      /** Tests if the string {} is parsed correctly
       */
      public void testNoElementStringToCollection() {
            String s0 = "{}";
            Collection c0 = (Collection)
               StringParser.stringToCollection(s0);
            Collection c0ok = new ArrayList();
            assertEquals(c0, c0ok);
      }

}
```

Let's analyze the test code. First of all, Bob creates a separate package called `util`, where he will store `StringParser` and any other utility class not directly related to the poll program. He then creates the `TestStringParser` class, which is a subclass of `TestCase`, the abstract testing class of the JUnit framework.

Each unit test for `StringParser` is enclosed in a method within `TestStringParser`. The first test Bob writes is the test he called `testNoElementStringToCollection`, which does just what the name suggests. It creates a `String` that represents a `Collection` with no elements: "{}", and expects that the method `stringToCollection` returns a Collection with no elements. He does so through the `TestCase` method `assertEquals`, which compares the two objects. In the test code, Bob chooses an `ArrayList` as the `Collection` implementation.

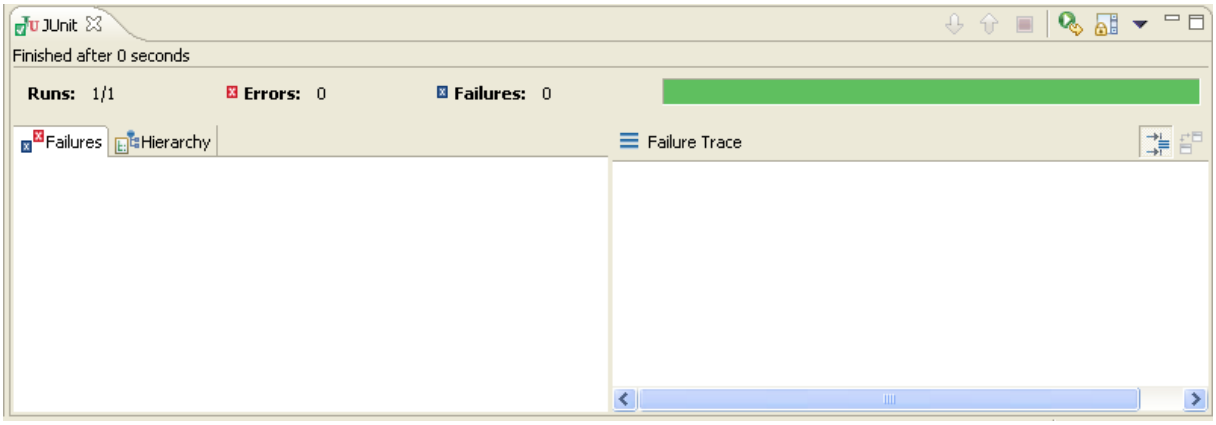
Bob runs `TestStringParser` with `JUnit` for the first time. The result of its execution is shown below:



Unsurprisingly, the only test resulted in an error because `StringParser` doesn't even exist yet. Bob then sets out to implement the simplest code he can to make the test pass. Such a code follows:

```
package util;

import java.util.Collection;
import java.util.ArrayList;

public class StringParser {

	public static Collection stringToCollection(String source){
		Collection result = new ArrayList();
		return result;
	}
}
```

The `StringParser` class simply returns an empty `ArrayList`, which should exactly match what is expected by the current test. This is the result of running JUnit again:

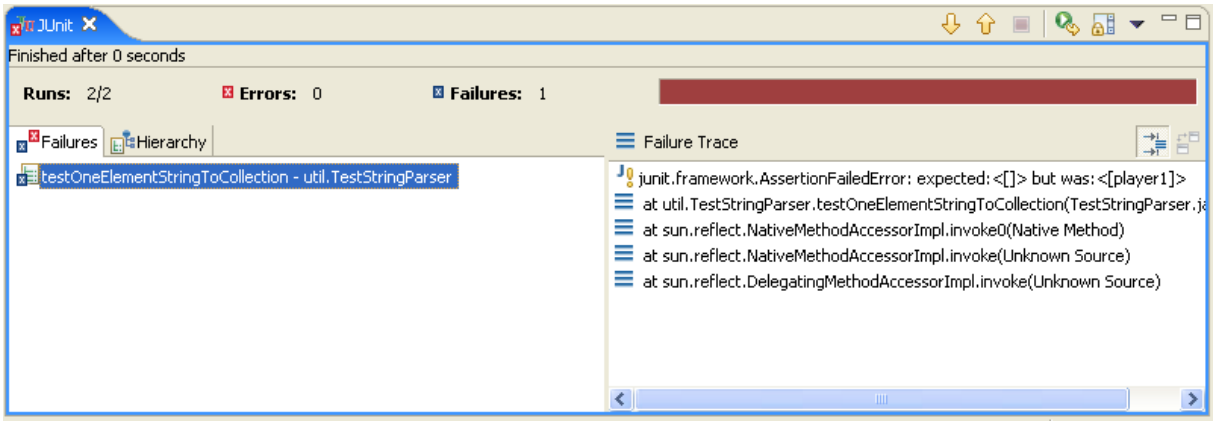


Green bar! All tests are working! Big deal, Bob. Now do some serious testing, will you?
Bob increments `TestStringParser` with one more method (that is, one more test), depicted below (just the method):

```
	/** Tests if the string {player1} is parsed correctly
	 */
	public void testOneElementStringToCollection() {
		String s1 = "{player1}";
		Collection c1 = (Collection)
		   StringParser.stringToCollection(s1);
		Collection c1ok = new ArrayList();
		c1ok.add("player1");
		assertEquals(c1, c1ok);
	}
```

This new test takes the `String` "{player1}" and must return a `Collection` (implemented as an `ArrayList`) consisting of the `String` "player1". To do so, the test creates a `Collection` with such a `String`, and compares it via `assertEquals` to the String return by the `stringToCollection` method.

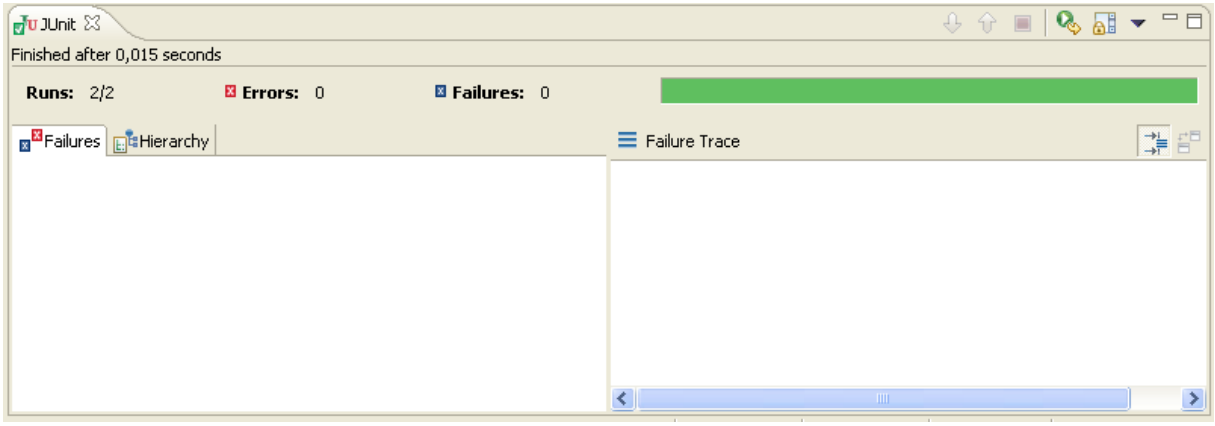
When Bob runs JUnit again, it shows an assertion error in this new test:



Bob must implement the code to make both tests pass. He does so by using a `StringTokenizer` in the `stringToCollection` method, as follows:

```
package util;

import java.util.Collection;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class StringParser {

      public static Collection stringToCollection(String source){
            Collection result = new ArrayList();
            StringTokenizer st =
               new StringTokenizer(source, "{,},\"", false);
            while (st.hasMoreTokens()) {
                  String s = st.nextToken();
                  result.add(s);
            }
            return result;
      }
}
```

The `StringTokenizer` breaks a `String` in a number of tokens determined by the separators listed in its creation. The code of the method then adds to the resulting `Collection` each of the tokens. After running JUnit, Bob gets … green bar!



Now that the `String` works with one element and with no elements, Bob thinks of one more test for the normal usage. He creates a test for the creation of 3 elements. There is a rule of thumb in Computer Science that states: "if it works with 3, it works with *n*" .

```
/** Tests if a string with 3 tokens is parsed correctly
 */
public void testThreeElementsStringToCollection() {
            String s3 = "{a,b,c}";
            Collection c3 =
               (Collection) StringParser.stringToCollection(s3);
            Collection c3ok = new ArrayList();
            c3ok.add("a");
            c3ok.add("b");
            c3ok.add("c");
            assertEquals(c3, c3ok);
}
```

Without touching the code of `StringParser`, Bob runs JUnit and it results in green bar! (we will not show you the green bar image anymore, ok? It's basically the same window).

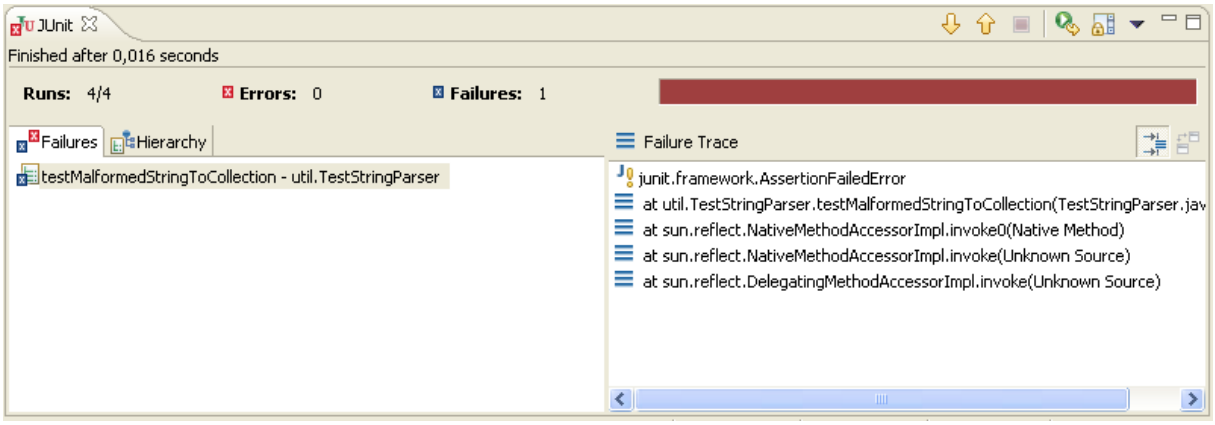
Bob is basically satisfied with the code for the normal behavior of `StringParser`. However, now he must do the real testing – abnormal situations. He begins by testing if a simple malformed `String` (missing " }") results in an exception thrown by `StringParser`. He method is:

```
/** Tests if a malformed string is detected
 */
public void testMalformedStringToCollection() {
      String bogus = "{";
      try {
            StringParser.stringToCollection(bogus);
            fail();
      } catch (MalformedStringException e) { /*ok*/
}
```

In order that this test works, the class `MalformedStringException` must be created. It will even be referred to in the poll program's Façade. Bob codes it right away, it is a simple class:

```
package util;

public class MalformedStringException extends Exception {
      public MalformedStringException(String message) {
            super(message);
      }
}
```

After updating the other methods so that `MalformedStringException` is thrown, Bob runs JUnit, which produces the following result:



Now he must update `StringParser`'s code so that it throws `MalformedStringException` when it finds malformed `Strings`. For this first case, he code resulted as follows (additions in bold):

```
public static Collection stringToCollection(String source)
      throws MalformedStringException {
      if (!source.startsWith("{") || !source.endsWith("}"))
            throw new MalformedStringException(
      "Collection must be a comma-separated list enclosed in { }");
      Collection result = new ArrayList();
      StringTokenizer st =
            new StringTokenizer(source, "{,},\"", false);
      while (st.hasMoreTokens()) {
            String s = st.nextToken();
            result.add(s);
      }
      return result;
}
```

But a number of other malformed `Strings` exist, so Bob creates a test for each one of it, using the same process: first he writes the test, runs it, and only then writes the simplest code just enough to make the test pass. In the end, this is the method for the malformed `Strings` test:

```
/** Tests if a number of malformed strings are detected
 */
public void testMalformedStringToCollection() {
      String bogus[] = new String[9];
      bogus[0] = "{";
      bogus[1] = "}";
      bogus[2] = "}";
      bogus[3] = "{,}";
      bogus[4] = "{,,}";
      bogus[5] = "{player1,,player2}";
      bogus[6] = "{,player1}";
      bogus[7] = "{player1,}";
      bogus[8] = null;

      for(int i=0; i<9; i++) {
            try {
                  StringParser.stringToCollection(bogus[i]);
                  fail();
            } catch (MalformedStringException e) {} /*ok*/
      }
}
```

Below is the corresponding code that made all tests pass:

```
package util;

import java.util.Collection;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class StringParser {

	/**
	 * Creates an ArrayList from a String in the form of a
	 * comma-separated list of names enclosed in { }
	 */
	public static Collection stringToCollection(String source)
			throws MalformedStringException {
		String errorMessage =
		  "Collection must be a comma-separated list enclosed in {}";
		if( source == null )
			throw new MalformedStringException( errorMessage );

		if (!source.startsWith("{") || !source.endsWith("}"))
			throw new MalformedStringException( errorMessage );
		Collection result = new ArrayList();
		StringTokenizer st =
			new StringTokenizer(source, "{,},\"", false);
		int countCommas = 0;
		for (int i = 0; i < source.length(); i++)
			if (source.charAt(i) == ',')
				countCommas++;
		if (st.countTokens() > 0 && st.countTokens() <= countCommas)
			throw new MalformedStringException( errorMessage );

		else if ((st.countTokens() == 0) && !source.equals("{}"))
			throw new MalformedStringException( errorMessage );

		while (st.hasMoreTokens()) {
			String s = st.nextToken();
			if (s.equals(""))
				throw new MalformedStringException( errorMessage );

			else
				result.add(s);
		}
		return result;
	}
}
```

Now, Bob decides he has fully tested the `stringToCollection` method. He still has the other method to test, `collectionToString`. By now, you should have already understood how the approach works. We will refrain from repeating the exposition of the step-by-step actions Bob took. Let's just skip to the final tests for `collectionToString` (they are also part of the `TestStringParser` class):

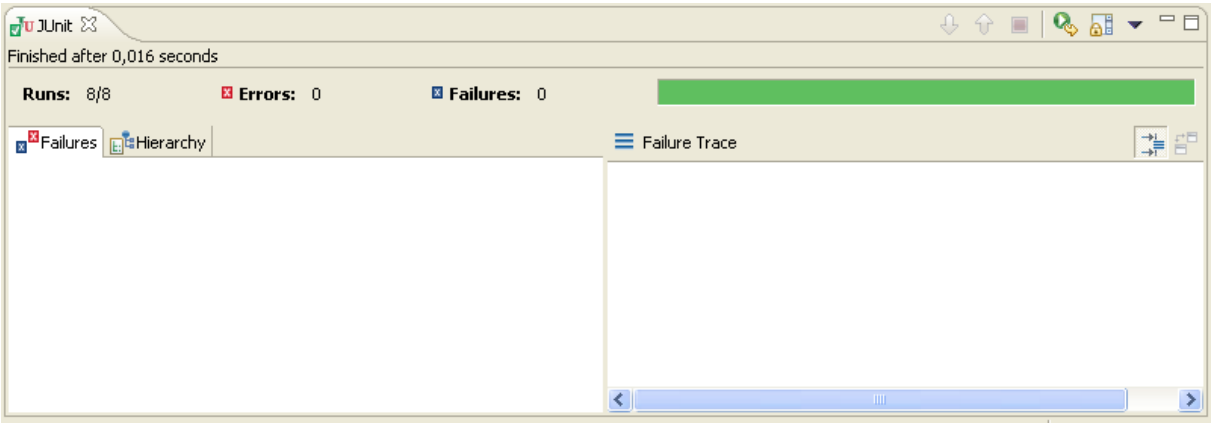
```
	/** Tests if a null Collection forms the right String
	 */
	public void testNullCollectionToString() {
		Collection c0 = null;
```

```
        String s0 = "{}";
        String s0ok = StringParser.collectionToString(c0);
        assertEquals(s0, s0ok);
    }

    /** Tests if an empty Collection forms the right String
     */
    public void testEmptyCollectionToString() {
        Collection c0 = new ArrayList();
        String s0 = "{}";
        String s0ok = StringParser.collectionToString(c0);
        assertEquals(s0, s0ok);
    }

    /** Tests if a Collection with one element forms the right String
     */
    public void testOneElementCollectionToString() {
        Collection c1 = new ArrayList();
        c1.add("player1");
        String s1 = StringParser.collectionToString(c1);
        String s1ok = "{player1}";
        assertEquals(s1, s1ok);
    }

    /** Tests if a Collection with three elements forms the right String
     */
    public void testThreeElementsCollectionToString() {
        Collection c3 = new ArrayList();
        c3.add("a");
        c3.add("b");
        c3.add("c");
        String s3 = StringParser.collectionToString(c3);
        String s3ok = "{a,b,c}";
        assertEquals(s3, s3ok);
    }
```

Notice that there are fewer tests this time, because there are no malformed syntaxes to cope with. Finally, this is Bob's code for the `collectionToString` method, which is included in the `StringParser`:

```
    /**
     * Creates a String representing a Collection of names
     */
    public static String collectionToString(Collection source) {
        if (source == null)
            return "{}";
        Iterator it = source.iterator();
        StringBuffer result = new StringBuffer("{");

        while (it.hasNext()) {
            Object obj = (Object) it.next();
            result.append(obj.toString());
            if (it.hasNext())
                result.append(",");
        }
        result.append("}");
        return result.toString();
    }
```

Bob has decided that, if a null `Collection` is received, instead of a null `String`, the `String` representing an empty `Collection` is returned ("{}").
In the end of all this process, Bob can finally rest (but only for a brief moment, because he has the full poll program to implement). This was JUnit's result for his last test run (green bar! 8 tests passing!):



Appendix III

This group of projects was compared with a sample of student projects taken from semesters *after* EasyAccept (in which students were given the acceptance tests). This time, as the students had to code their own Façade to begin with, it was just a matter of running the tests.

The two groups were *not* assigned the same project (we assign a new project every semester), but students were taken from the same course in the curriculum and had equivalent skills; the projects had also equivalent complexity, and the course was taught by the same teacher.

In Table III.1, below, we see the results for the first group of projects, taken from a class which had the assignment of building a framework for cell phone games. Project correctness was measured as the percentage of user stories that were completed successfully (without errors when running the acceptance tests with EasyAccept).

	Acceptance Test Set #	**Project 1**	**Project 2**	**Project 3**	**Project 4**
"Hungry" Game	US 1	failed	ok	failed	failed
	US 2	ok	ok	ok	ok
	US 3	ok	ok	ok	ok
	US 4	ok	ok	ok	ok
	US 5	ok	ok	ok	ok
	US 6	ok	ok	ok	ok
	US 7	ok	failed	ok	ok
"Gula" Game	US 8	failed	ok	failed	failed
	US 9	ok	ok	ok	ok
	US 10	failed	ok	ok	ok
	US 11	failed	ok	ok	ok
	US 12	ok	ok	ok	ok
	US 13	ok	ok	ok	ok
	US 14	ok	ok	ok	ok
	US 15	ok	ok	ok	ok
	US 16	ok	ok	failed	ok
"Magic" Game	US 17	failed	ok	failed	failed
	US 18	ok	ok	ok	failed
	US 19	ok	ok	ok	failed
	US 20	ok	ok	ok	ok
	US 21	ok	ok	ok	ok
	US 22	failed	failed	failed	failed
	US 23	failed	ok	ok	failed
	US 24	failed	failed	ok	failed
	US 25	failed	ok	ok	ok
	US 26	ok	ok	ok	ok
	US 27	failed	ok	failed	failed
	US 28	failed	failed	failed	failed
	%acceptance	**60.71**	**85.71**	**75**	**64.28**

Table III.1 – Correctness of sample projects *before* using EasyAccept

In table III.2, you can find the results for sample projects taken from a class which was assigned a direct mail project and received the acceptance tests in the beginning.

User story	Acceptance test Description	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
Letter generation for a single client	**1** - Setup testing: do we have the expected database?	ok	ok	ok	ok	failed	ok	ok	ok	ok	ok	ok
	2 - Setup testing: do we have the expected templates?	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	3 - Sending mail with 1 client and no tags	ok	ok	ok	ok	failed	ok	ok	ok	ok	ok	ok
	4 - Database errors testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	5 - Template errors testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	6 - Tag errors testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	7 - Sending errors testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
Mail sending to a single client via mail system	**8** - Setup testing: do we have the expected database?	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	9 - Setup testing: do we have the expected mail systems?	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	10 - Mail sending to a file	ok	ok	ok	ok	ok	failed	ok	ok	ok	ok	ok
	11 - Mail sending through a mail system	ok	ok	ok	ok	ok	failed	ok	ok	ok	ok	ok
Database record keeping	**12** - Creation and reading testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	13 - Alteration testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	14 - Removal testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	15 - Persistence testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
Selection according to criteria	**16** - Selection and sending testing	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
	%acceptance	**100**	**100**	**100**	**100**	**87.5**	**87.5**	**100**	**100**	**100**	**100**	**100**

Table III.2 – Correctness of sample projects *after* using EasyAccept

The gain in correctness is evident. Theoretically, there should not even be failing tests in the projects for the second group. We are still trying to find why some student groups every now and then insist on delivering incomplete projects. Real people wouldn't do that, right?

In the end of a semester, we are pleased to congratulate students for their compliance with the project's requirements. In addition to making this important point for students, we are also happy to have them introduced to good software development practices from which they will benefit during their career. We are also relieved because the process of correcting and grading the projects is so much easier – it basically takes us to run EasyAccept for each projects and check if tests pass.

However, the downside is that you must put a little effort in the beginning of the semester, in order to create the tests. Let's see some hints on how to do this easily.

How to create the tests

Once you decide to use acceptance tests in your project assignments, the first time around you will have to create the tests yourself, based on what you've learned from this text. With time, as classes come and go, the job can be easied up considerably because you can get other people to write the tests for you. The easiest way is interacting with the software analysis teacher and students from late semesters. Get the software analysis teacher to read this text and teach some testing techniques and patterns to his students. This completes the cycle on ATDD teaching, because they will experience how the "other side" works. As these students have already had experience with ATDD and EasyAccept in your classes, they will certainly be able to write tests. They could be assigned to write tests for the project that will be used in the next semester's software design course.

No matter how the tests are created, make sure to review them the first time around, and follow the recommendations below.

Leave some typos and doubtful points in the scripts on purpose. This will force students to contact you for clarification, as they can't change the tests. It also serves as a feedback for you. If in a month's time no one has contacted you yet, it is unlikely that any student is doing progress in the project.

In addition to the test cases you provide the students with, warn them that you have *secret* tests that will be used for grading. That way, students will be forced to think beyond the tests they receive. Furthermore, they will always have to code the general case out of the examples the test suggests, not just the code that makes the test pass.

How to assign the projects

Once you have the acceptance tests, explain the project to your students in the first class. You should give them the tests, the user stories, and maybe some other artifacts that will help them understand the project, just for reference. We give them a conceptual model and a glossary for this purpose.

A nice thing that we always do is to code the first user story in the first few classes with the students, explaining how the ATDD technique works. Make sure to include in this first user story a point where you can show a unit test being written, too. During the exposition, you should also explain patterns like **Client Assertion**, which students will need to apply. Remember that you are the student's client, so be available to clarify their project doubts during classes or via email.

How to grade students

Always use your own test files when testing student's projects, because they may cheat by including modified tests in the project. In order to avoid cheating and plagiarism, we also recommend applying MOSS (Measure Of Software Similarity), an online service provided by Stanford University (http://moss.stanford.edu).

We use software correctness (measured as the percentage of acceptance tests that pass) as the paramount weight in the student's project grade. We support the decision by telling students that, if they can deliver software that does what the clients want correctly, they have accomplished the single most important goal of software development. A few semesters ago we used to grant 70% of the grade for software correctness, but now we have reduced it to 50%. This was decided mainly because some students wouldn't do enough hard work to improve software code and design once they had gone through all the tests.

You can divide the remainder of the grade between design, code, documentation, and any other aspect you want to evaluate. Code is particularly difficult to grade because it involves subjective elements to some extent, but you can systematize it by using some more objective criteria like code smells (bad variable names, magic numbers, code repetition, deep nesting of loops, cascading if clauses, usage of instanceof, etc.). Evaluating for code repetition can be automated if you use MOSS, because it also compares code inside the same project.

6

6.2 – Contribuições

Acreditamos que o texto dessa dissertação será bastante útil para os leitores em potencial. Em particular, acreditamos que a dissertação terá grande valia para neófitos em desenvolvimento de software, como estudantes de graduação em Ciência da Computação, uma vez que ele inclui exemplos passo a passo com código Java que podem ser usados de forma complementar em um curso de projeto de software ou até mesmo de programação. Também parece ser de fácil compreensão – pelo menos até onde as listagens de código não estão envolvidas – para pessoas sem conhecimento técnico, o que o torna potencialmente válido como texto introdutório para clientes que participarão de projetos de desenvolvimento de software.

O levantamento de padrões e práticas foi um primeiro passo na direção de algo que esperamos se torne bem maior – um catálogo extenso de conhecimento legitimado e comprovado pelo uso todos os usuários da metodologia. O exemplo de aplicação da metodologia dentro de um processo XP serve como um guia rápido que pode ser usado para manter o foco dos participantes de um projeto de desenvolvimento de software no que cada um deve fazer.

6.3 – Limitações

Seguem algumas considerações sobre o escopo e as limitações deste trabalho.

A metodologia ATDD tem aplicabilidade associada ao contexto de processos ágeis de desenvolvimento *[Beck99]*, o que exige uma filosofia baseada no cliente e práticas centradas em testes e produção rápida de funcionalidade para feedbacks freqüentes. Não temos experiência da aplicação da metodologia em conjunto com processos "pesados", apesar de acreditarmos que a aplicabilidade do ATDD não está restrita a processos ágeis.

Além disso, a experiência que temos com a metodologia envolve apenas projetos de pequeno porte, com equipes envolvendo não mais que 10 pessoas, em ambiente universitário (ou seja, equipes incluem estudantes de Ciência da Computação e professores universitários, além de desenvolvedores profissionais). Isso é refletido no fato de o texto ser introdutório, com exemplos tirados de projetos de software simples.

Finalmente, os exemplos de teste de aceitação usados na dissertação estão escritos no formato adotado pelo EasyAccept e os exemplos de código utilizam a linguagem de programação Java. Apesar de o EasyAccept ser facilmente modificável para funcionar com outras linguagens, decidimos usar exemplos em Java por ser esta linguagem mais amplamente adotada em ambiente universitário, de onde vem um importante público-alvo para o livro.

6.4 – Trabalhos Futuros

A escrita de um texto introdutório sobre a metodologia ATDD usando EasyAccept dá um apoio inicial na evolução da ferramenta e da metodologia. Entretanto, há muito ainda o que fazer para que se possa lucrar ainda mais com a utilização de ambas.

Uma primeira frente de trabalho envolve levantar mais padrões e práticas para a metodologia ATDD, e desenvolver e refinar os já levantados. Os padrões que estão incluídos nessa dissertação foram discutidos no EuroPLoP 2007, uma conferência específica entre pesquisadores de padrões, e geraram um interessante debate que levou à conclusão de que há ainda muito mais padrões a serem escritos. Repositórios de padrões e práticas têm natureza comunitária e colaborativa, de forma que estes não são validados apenas pelo trabalho isolado de uma pessoa ou grupo. Somente através uso repetido e legitimado por toda uma comunidade podem padrões e práticas ser credenciados como tal. Assim, um trabalho interessante seria envolver-se com as comunidades de padrões, com os grupos que utilizam a metodologia, participar das demais conferências com submissão de artigos, etc., para ajudar a aprimorar o estado da arte da área.

A segunda frente envolve evoluir a ferramenta EasyAccept em si. Idéias incluem a criação de uma IDE gráfica, além de novas funcionalidades para tornar o uso da ferramenta ainda mais simples. Facilidade de uso é talvez a principal característica que se quer obter com a evolução do EasyAccept, então tudo que puder facilitar a vida do cliente – revisores e potenciais criadores de testes – deve ser incorporado à ferramenta. As idéias sobre novas funcionalidades para o EasyAccept podem ser encontradas na lista de discussão sobre a evolução da ferramenta, cuja URL é http://groups.google.com/group/evolution-of-easyaccept.

7

[Reppert] Reppert, T.