Bachelor's thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

MATLAB Mock Library

Vladimir Perić

May 2013
Supervisor: Ing. Petr Pošík, Ph.D.

Acknowledgement / Declaration

I would like to thank my mentor, whose wholehearted aid made this work possible, and all of my friends, who made slacking off so enjoyable.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl večkeré použité informační zdrove v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 24.5.2013

Abstrakt / Abstract

Tato práce popisuje knihovnu pro vytváření maket (mock) objektů v MATLABu, její návrh a implementaci. Makety objektů jsou důležitým nástrojem při testování softwarových jednotek (unit testing) a žádný podobný nástroj pro MATLAB dosud neexistoval. Vzorem implementované knihovny je mockito, knihovna pro vyvtváření maket objektů v Javě. V práci jsou rozebrány ty vlastnosti jazyka MATLAB, které mohou mít vliv na návrh a implementaci. Na základě této analýzy je navrženo rozhraní (API) celé knihovny. Vytvářené makety umožňují poskytovat testované jednotce nepřímé vstupy (stubbing) a kontrolovat její nepřímé výstupy (verification); při obojím lze navíc využít podmínky na argumenty (argument matchers). Výsledkem projektu je funkční knihovna maket objektů vhodná pro praktické použití.

Klíčová slova: MATLAB, programování řízené testy, makety objektů.

Překlad titulu: Knihovna maket objektů (Mock) pro MATLAB

A library for creating and using mock objects in MATLAB, its design and implementation, is described in this thesis. Mock objects are an important tool for unit testing, and no such tool for MATLAB existed. This library is inspired by and modeled after mockito, a Java mock library. MATLAB language features affecting the library design are analyzed. Based on this analysis, the library API is designed. The mock objects created with this library support stubbing, behavior verification, and argument matchers. The final result of the project is a functional mock library implementation, suitable for practical use in other software projects.

Keywords: MATLAB, test-driven development, mock objects.

Contents

1 Introduction 1
1.1 Mock objects 1
1.2 Motivation 2
2 MATLAB specifics 4
2.1 Value and handle classes 4
2.2 Reflection 4
2.3 `subsref` and `substruct` 4
2.4 Method calls 5
2.5 Packages 5
2.6 Variable number of inputs / outputs 5
2.7 Stack traces 6
3 Design 7
3.1 Stubbing 7
3.1.1 *mockito* 7
3.1.2 *mockito-python* 8
3.1.3 *mmockito* 8
3.2 Verification 10
3.2.1 *mockito* 10
3.2.2 *mockito-python* 11
3.2.3 *mmockito* 11
3.3 Matchers 12
3.3.1 *mockito* 12
3.3.2 *mockito-python* 13
3.3.3 *mmockito* 13
3.4 Spying and partial mocks 14
3.4.1 *mockito* 14
3.4.2 *mockito-python* 14
3.4.3 *mmockito* 14
3.5 Strict and Tolerant mocks 15
4 Usage 16
4.1 Problem specification 16
4.2 Tests 16
4.3 Summary 18
5 Implementation 19
5.1 Internal classes 19
5.1.1 Invocation 19
5.1.2 InvocationPattern 19
5.1.3 VerificationError 20
5.2 Matcher 20
5.2.1 Any matcher 20
5.2.2 AnyArgs matcher 20
5.2.3 ArgThat matcher 21
5.2.4 ArgEqualTo 21
5.2.5 NumberBetween, StringContaining matchers 21
5.3 Mock 21
5.3.1 Constructor 22
5.3.2 subsref 22
5.3.3 when 22
5.3.4 verify 23
5.3.5 verifyZeroInteractions 23
5.4 InOrder 23
5.4.1 Constructor 23
5.4.2 subsref 23
5.4.3 verify 23
6 Current limitations 24
6.1 Method calls 24
6.2 Methods with no arguments 24
6.3 Mocking abstract classes 24
6.4 Return arguments 25
6.5 Unmockable method names 25
6.6 Autocomplete 25
7 Conclusion 26
7.1 Future work 26
References 27

Chapter 1
Introduction

Test-driven development (TDD) is a software development process with a focus on writing tests: for each new feature, a test is first written, code is then added until the test passes. The developer is then free to refactor the code as much as needed to get it to the required coding standards, after which the process is repeated. Running the tests after each change assures that no new bugs will be introduced, assuming the process is followed constantly and all features are properly tested. The process was popularized by Kent Beck in his 2002 book, *Test-Driven Development by Example* [1].

In *Growing Object-Oriented Software, Guided by Tests* [2], it is claimed that following the test-driven development process gives constant feedback on the quality of both the implementation and the design. Writing tests first forces developers to precisely specify the required feature, encourages the writing of loosely coupled components and prevents over-design of unnecessary features, thus contributing to a better design. Running tests prevents the appearance of regressions, thus assuring the quality of the implementation. Small refactorings can then be done repeatedly without fear, eventually significantly improving code quality. The authors summarize the 'Golden Rule of TDD' as: Never write new functionality without a failing test.

Tests are generally grouped in several categories, through precise definitions vary. They can broadly be categorized into:

- unit tests, which are detailed tests of individual objects and their methods;
- integration tests, which test the collaboration between multiple objects;
- acceptance tests, which are driven by end-user requirements and are generally higher level; and
- regression tests, are all tests which once passed successfully, assuring that previously working features will always work.

Internally, all test types follow the same structure: setup the system under test, execute the code, validate the results and cleanup test system state.

As the test-driven development process advocates the constant running of tests, software tools are used to automate this process; a popular unit testing framework is *xUnit*, implementations of which are available in many languages. Such frameworks provide test validation capabilities and result reporting.

1.1 Mock objects

One of the key tenants behind the test-driven development process is to maintain a clean separation of object responsibilities: such objects can then be easily tested and thus changed. Problem arise when objects need to interact with other objects, their 'neighbors'. Unit testing such objects would then require relying on the implementation of other objects, which is inconvenient. In these cases, it is possible to use *mock objects* to facilitate proper testing by standing in for these 'neighbors'. Mock objects are substitute implementations for testing how an object interacts with its neighbors. [2]

It is important to note that there is a degree of confusion on the terminology of these substitute objects. There have been attempts to name all helper objects used in testing 'test doubles', which are then divided into:

- dummy objects, which provide no functionality;
- stub objects, which return canned values to certain calls;
- spies, which record information allowing for more detailed verification of state; and
- mocks, which are objects pre-programmed with expectations of the calls they are about to receive.[3]

However, in the common programming vernacular, the term 'mock object' is used to refer to stubs, spies and mocks, and different libraries use the same naming for different objects. This thesis will use the above definition of mock objects, which covers all four object subtypes just specified.

Regardless of the name used, there are two basic capabilities mock objects have: stubbing and verification. *Stubbing* is the ability to define method calls which will return canned values, in order to emulate the behavior of another object. Stubbed calls have to be precisely defined, and mocks will not respond to other calls, or exhibit some sort of default behavior. *Verification* refers to the ability to check if some expected interactions have been invoked on the mock object; more precisely, this feature is called behavior verification, checking if tested objects behave in a certain way. State verification, checking if a variable is in a certain state, can be accomplished without mock objects.

Although there are many mocking frameworks available, this thesis focuses on *mockito*, a Java mocking framework. Inspired by other popular Java libraries like easyMock and jMock, *mockito* tries to offer a simpler and more intuitive approach: questions about interactions are asked after execution; it is possible to verify only relevant interactions. It aims to be simple to setup and is designed primarily for ease of use. [4]

1.2 Motivation

MATLAB is a proprietary, high-performance language for scientific and technical computing, developed by MathWorks. MATLAB wasn't originally designed as an object-oriented language, but has adapted with time and now offers many object-oriented programming features; its users might also benefit from the test-driven development process. MATLAB has had an open-source xUnit test framework since 2008 [5] on the MATLAB File Exchange [6], the website where most third party MATLAB toolkits are published and distributed from. This implementation was written by a MathWorks employee, but distributed under the open-source BSD license and widely used. Since release R2013a [7], an alternative implementation, the `matlab.unittest` package, is available as an official part of MATLAB, and the development of the original xUnit framework has been discontinued. As such, this project was developed using the new `matlab.unittest` module.

No mocking frameworks for MATLAB currently exist. **The goal of this thesis is to develop a functional mocking library for MATLAB** as a basis for future research and use. The *mockito* Java library will serve as the main inspiration. Once a satisfying amount of features is implemented, this library will be open-sourced and made available to other MATLAB users via File Exchange, to further evaluate the chosen approach and its general usability. The intended audience are those already familiar with test-driven development and mocking; the advantages and disadvantages of this development process will not be covered further.

Chapter 2 covers MATLAB language specifics, which need to be accounted for when designing the library. The design is covered in chapter 3, while chapter 4 contains a short practical demonstration of mock object use. The implementation is covered in detail in chapter 5 and framework limitations are listed in chapter 6. Chapter 7 summarizes the results and suggests possible future work.

Chapter 2
MATLAB specifics

Mocking frameworks have been ported to many languages, and each implementation has had to deal with language-specific issues. The currently used and preferred MATLAB object-orientated syntax is available since release 2008a [7]; an older implementation is also present but not widely used anymore. Although many things are now possible, MATLAB's language isn't as feature-complete as say Java or .NET, so it was important to investigate specific issues to determine if a mocking library is at all feasible in MATLAB.

This chapter will discuss some of the more important identified issues and their solutions. The list is presented in no specific order.

2.1 Value and handle classes

MATLAB provides two basic types of classes: value and handle. Value classes are simple, immutable objects which do not need to be unique (for example, numbers); there are no interesting differences between two instances with the same contents. When changed, a new instance of the class is created and its content is set to the new value. In comparison, handle classes work like pointers or references in other programming languages, and provide just a reference to a specific object. If copied, the handle is copied but not the data associated with it. Mock objects will have to be handle classes, as they need to be able to change dynamically; there is no alternative.

The question is, do we mind not being able to mock value classes? The answer is simple — value classes should not be mocked, as per [2]. Since they ought to be immutable anyway, the correct approach is to just create an instance and use it directly.

2.2 Reflection

In Java and Python, it is possible to use reflection to modify the structure and behavior of programs at runtime, and this language feature is used by both *mockito* and *mockito-python*, while MATLAB only has limited support for introspection. The MATLAB `meta.class` class contains all information on a class, including among others all methods it has (both directly defined and inherited) and its properties and their statuses. However, while this information can be inspected, it is not possible to dynamically construct a `meta.class` object and then create a class from it. Therefore, an alternative way of constructing mock objects needs to be devised.

2.3 `subsref` and `substruct`

The key requirement of a mocking framework is to be able to handle arbitrary objects. In more powerful languages (eg. Python), it might be possible to dynamically define a method, but this is not the case in MATLAB. Instead, the special `subsref` method

can be used. Per the documentation, `B = subsref(A, S)` is called by MATLAB for the syntax `B = A(i)`, `B = A{i}` or `B = A.i` when `A` is an object.[1])

In the above `subsref` call, `S` is a special structure which is passed to and required by overloaded `subsref` and `subsasgn` methods. It has a `type` field, which can be one of `'.'`, `'()'`, or `'{}'`, and a `subs` field which contains the subscript values we are referencing (a field name or a cell array of index vectors). In the case of nested calls, `S` is an array of such structures. A structure with the appropriate form can be created either manually, or by using the `substruct` method. Therefore, by overloading the `subsref` method and carefully inspecting the contents of the received substruct `S`, we can process the desired *mockito*-like syntax and even nest statements to an arbitrary depth.

2.4 Method calls

MATLAB offers two common ways of calling a method on an object: `method(obj)` and `obj.method`. `method(obj)` calls do not pass through `subsref` and as such cannot be processed by a custom, overridden `subsref`. Simple benchmarking has shown that `method(obj)` calls are somewhat faster than `obj.method`, and not calling `subsref` might be a major part of the reason.

2.5 Packages

In MATLAB, there are two ways of making custom classes and functions accessible: adding them to the MATLAB search path or importing them from packages. If a folder is on the path, all of the classes defined in it are always available; conversely, this clutters the namespace and can lead to accidental overriding of MATLAB builtin functions (as the userpath is checked before MATLAB toolboxes are imported). On the other hand, if packages are used, their parent folder needs to be added to the search path but then they need to be explicitly imported, or called with their fully-qualified name.

While it is possible to use mocks interactively, most usage will be in unit tests, which are generally composed out of many functions, each representing a single unit. Unfortunately, MATLAB imports are local — they need to be repeated in every function and method which needs them. This needlessly makes the tests longer and harder to read. Therefore, a compromise was chosen - the most commonly used classes will be provided directly in a root directory, while internal functions will be hidden inside a package. Another package will contain some less commonly used functions and classes.

2.6 Variable number of inputs / outputs

While other languages always work with a precisely known number of arguments and just one return value (even if any of those can be objects containing an arbitrary number of other objects), MATLAB has built-in support for variable number of inputs and outputs, via the `varargin` and `varargout` special arguments. When mocking, this feature also needs to be supported: users might wish to stub a function which accepts a variable number of inputs, or returns more than one output.

[1]) `subsasgn` is a related special method, which is called when interpreting indexed assignment statements. Its use is not required in *mmockito*.

2.7 Stack traces

When verification fails, it is important to show an informative stack trace, so that the point of failure in code can be quickly reached, rather than the error in the mock object. A MATLAB `MException` object can be created to contain any information necessary, though a stack trace is included by default. Other than the general purpose `throw` method, it is also possible to use the `throwAsCaller` method to throw the exception as if by the calling function. However, it us not possible to chain these calls to 'skip' more than the first called method.

It is possible to use the `dbstack` command to return the function name, line number where the command occurred in the source code, name and line number of the caller and so on until the top-most function is reached; the command also supports an optional argument to skip the first n frames of the call stack. While this is exactly the sort of information needed by mock objects, a suitable way of adding the needed data has yet to be found.

Chapter 3
Design

The main inspiration behind this project is *mockito*, an open-source Java mocking framework. The key difference between it and other mocking frameworks is the ability to verify the behavior of the system under test without establishing expectations beforehand [4]; as such, it is possible to focus on just the relevant interactions instead of having to explicitly define complete behavior. Its focus is on simplicity — only one class of mocks and only one way of creating mock objects is provided; the same object supports both stubbing and verification. *mockito* in general tries to be as simple as possible and provide a slim Application Programming Interface (API). This simplicity was a key design consideration, as users of MATLAB can be expected to be scientists before programmers, and might not appreciate a difficult to learn new system.

Being a Java library, *mockito* is generally more verbose than equivalent MATLAB code. Following the exact syntax would go against the simplicity design goal, as well as impacting usability, as making tests harder to read makes them less valuable. As such, its port to Python, *mockito-python* was often considered when deciding on the exact syntax to use, as it is in general cleaner; Python is dynamically-typed language (as opposed to statically-typed Java), which more closely mirrors MATLAB's language. Additionally, being a younger project, *mockito-python* does not provide as many features as *mockito* — yet it is safe to assume that the most important features *are* implemented, covering the vast majority of use cases. By considering both, it is easier to intelligently limit the scope of this project.

In honor of the ideas provided by *mockito*, this project is named *mmockito* (with permission from the author of *mockito*), which stands for MATLAB *mockito*.

This chapter will discuss the basic features, stubbing and verification, their syntax in both *mockito* and *mockito-python*, the specific features supported, the chosen syntax (with regard to limitations imposed by MATLAB) and the set of features to be supported by *mmockito*. All syntax examples are taken from the *mockito* [8] and *mockito-python* [9] documentation.

3.1 Stubbing

A basic requirement during testing is the ability to provide predetermined answers to specific calls; such objects are generally called stubs. While stubs can be created directly, it is often more convenient to use a mocking framework; in addition, creating extra classes is harder in MATLAB than in other languages, as they have to be in separate files and on the path. Therefore, it is useful to provide a way of quickly generating stubs. Other commonly available features are throwing exceptions and stubbing consecutive calls (referred to as 'iterator-style stubbing' [8]).

3.1.1 *mockito*

mockito offers a simple way of creating basic stubs, seen in the following code excerpt from its documentation:

```
LinkedList mockedList = mock(LinkedList.class);

when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

System.out.println(mockedList.get(0)); // prints "first"
System.out.println(mockedList.get(1)); // throws RuntimeException
System.out.println(mockedList.get(999)); // prints "null"
```

Stubbing consecutive calls follows the same principle, with the last function call, **thenReturn**, repeated as many times as required. If the method is called more times than stubbed, the last stubbed value is returned.

```
when(mock.next()).thenReturn(1).thenReturn(2).thenReturn(3);

//Alternatively
when(mock.next()).thenReturn(1, 2, 3);
```

Finally, due to Java limitations regarding void methods[1]), a separate family of methods exists to handle these cases: **doThrow()**, **doAnswer()**, **doReturn()** and others. These can be used to replace all calls to the **when** method if desired. However, since MATLAB does not suffer from the same limitation these were not considered during design; they offer no new features compared to those already shown.

3.1.2 *mockito-python*

mockito-python offers essentially the same feature set, though the syntax is slightly different: instead of defining mocked calls as if called on the mock object, they are instead called as a method of the **when** object. The following code illustrates this:

```
mockedList = mock()

when(mockedList).get(0).thenReturn("first")
when(mockedList).get(1).thenRaise(Exception())

print mockedList.get(0) # prints "first"
print mockedList.get(1) # raises exception
print mockedList.get(999) # prints None
```

Stubbing consecutive calls is also possible, in the same vein as *mockito*:

```
when(mock).next().thenReturn(1).thenReturn(2).thenReturn(3)
```

As Python is a flexible language, with *mockito-python* it is also possible to mock static methods, so that all later created instances of the class will have them. None of these features have been deemed critical and might not even be implementable in MATLAB, so they were not considered when designing the API. In *mockito-python* it is also possible to call an **unstub** method, which unregisters all stubs. Since mocks shouldn't be used interactively, there is no compelling case for the implementation of this feature.

3.1.3 *mmockito*

Since the stubbing features offered by *mockito* are relatively simple, the biggest design issue was deciding on the appropriate syntax. *mockito* has syntax of

[1]) The compiler throws errors when processing void methods inside brackets.[8]

the form `when(mock.method()).then...`, while in *mockito-python* the syntax is `when(mock).method().then...`. Initial design called for the creation of a `when` method of mock objects, which would then be followed by a number of chained method calls. Unfortunately, this approach failed — the `when` method call works only for the exact arguments it is defined for, and cannot be followed by an arbitrary number of chained method calls.

An alternative approach was attempted: 'trick' MATLAB into accepting a `when(mock)` syntax by creating a `when` class, which would then just pass the details of its `subsref` call to a method of the `Mock` class. While this would add overhead, the trade-off was deemed worthwhile, as testing is not considered performance-critical. Confusingly, this approach also didn't work: while MATLAB was happy to allow this syntax when used from the command prompt or in a function, it threw an error[1]) when used in a script. This inconsistent behavior was reported as a bug, but no confirmation was received as to whether the bug is the error in one case, or the fact that it works at all. Nevertheless, it is not good design to rely on inconsistent behavior. Finally, the syntax `mock.when.method().then...` was chosen as a compromise solution implementable in MATLAB.

Naming the methods which define the return values `thenReturn` and `thenThrow` was a clear choice; an additional `thenPass` method, as a shorthand for `thenReturn(true)` was also added — these three methods are collectively referred to as `then*` methods in further text. Unlike Java or Python, these cannot be defined as methods in MATLAB, as such chained calling is not supported. They need to be implemented as special 'keywords' which would then be handled by the appropriate internal methods; this also means no auto-completion can be offered for them.

```
mockedList = Mock();

mockedList.when.get(0).thenReturn('first');
mockedList.when.get(1).thenThrow(MException());

mockedList.get(0) % ans = 'first'
mockedList.get(1) % throws exception
mockedList.get(999) % ans = []
```

In Java, functions can return only one value or object; in Python it is possible to return multiple objects in a tuple and MATLAB does something similar by returning a cell array. As such, multiple return values need to be supported by `thenReturn`.

```
mockedList.when.get(2).thenReturn('third', 3);

 [a, b] = mockedList.get(2) % a = 'third'  b = 3
```

Stubbing consecutive calls is also an important feature and should be supported. There is no reason to deviate from the syntax offered by *mockito*, so an arbitrary amount of `then*` calls will be allowed, with the last stubbing considered the relevant one once the others are exhausted. An additional keyword was also added, `times(n)`, which can follow any `then*` method and specifies the number of times the stubbed value will be returned. In such a way it is possible to override the 'infinite' stubbing of the last value — once all stubs are exhausted, the mock will act as if the method wasn't stubbed at all.

[1]) The exact error message shown is:
Static method or constructor invocations cannot be indexed.
Do not follow the call to the static method or constructor with any additional indexing or dot references.

```
mock.when.next().thenReturn(1).thenReturn(2).thenReturn(3);
mock.when.next().thenReturn("ok").times(3).thenThrow(MException());
```

3.2 Verification

The second key feature of mocking frameworks is the ability to verify that interactions happened. The simplest form is to check if a specified interaction was invoked on the mock; it is also possible to verify the number of invocations made. *mockito* further offers the options of checking that no invocations were made at all, or that all interactions were already verified. Finally, in-order verification is supported as a special case, in case the specific order of calls is also important.

3.2.1 *mockito*

Mocks will remember all interactions performed on them, and performing simple verifications is trivial. There is a syntax inconsistency with `when` and `verify`: when stubbing, the stubbed methods are called as if on the mock object; when verifying, they are called as if they were functions of some `verify` object. While a small difference, it is an inconsistency which should be avoided.

```
mockedList.add("one");
mockedList.clear();

//verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

It is also easy to verify the number of calls made, by adding another parameter to the `verify` call. Supported arguments are `times(n)`, `never()`, `atLeast(n)` and `atMost(n)`, with `times(1)` being the default and `never()` an alias for `times(0)`.

```
mockedList.add("once");
mockedList.add("twice");
mockedList.add("twice");

verify(mockedList).add("once");
verify(mockedList, times(2)).add("twice");
verify(mockedList, never()).add("never happened");
verify(mockedList, atMost(5)).add("twice");
```

It is also possible to verify no interactions happened on a mock, or that all interactions were already verified:

```
//only mockOne is interacted
mockOne.add("one");

//verify that other mocks were not interacted
verifyZeroInteractions(mockTwo, mockThree);
```

```
mockedList.add("one");
mockedList.add("two");

verify(mockedList).add("one");

//following verification will fail
verifyNoMoreInteractions(mockedList);
```

Finally, in-order verification is supported with the use of a helper class, `InOrder`: so that interactions could be checked across multiple mocks, a separate object must be used.

```
firstMock.add("was called first");
secondMock.add("was called second");

//create inOrder object passing any mocks that need to be verified
InOrder inOrder = inOrder(firstMock, secondMock);

//following will make sure that firstMock was called before secondMock
inOrder.verify(firstMock).add("was called first");
inOrder.verify(secondMock).add("was called second");
```

Any stubbed calls can also be verified, though it is suggested that this is bad test design and should be avoided.[9][8]

3.2.2 *mockito-python*

The feature-set offered by *mockito-python* is essentially the same, as is the syntax; the only difference is that number qualifiers are passed as named arguments and not as separate objects:

```
mockedList.add("once")
mockedList.add("twice")
mockedList.add("twice")

verify(mockedList, times=2).add("twice")
```

In-order verification is implemented somewhat differently: *mockito-python* seems to keep a global list of all invocations made on mocks, so no creation of a separate InOrder object is required. It also allows all verification number qualifiers to be used, which is a feature not mentioned in *mockito* documentation.

```
firstMock.add("was called first");
secondMock.add("was called second");

inorder.verify(firstMock).add("was called first");
inorder.verify(secondMock).add("was called second");
```

3.2.3 *mmockito*

As both projects mentioned offer the same feature-set, they all need to be available in *mmockito* as well. As in stubbing, the `verify(mock)` syntax has to be replaced with `mock.verify`, due to MATLAB method call handling; the same applies for `verifyNoMoreInteractions` and `verifyZeroInteractions`. This syntax implies that the verification number qualifiers will have to appear at the end of the call, which does not negatively impact readability, and might even improve it. The supported qualifiers are `times(n)`, `never()`, `atMost(n)` and `atLeast(n)`, with the same meanings as in *mockito*. If necessary, more short-hands can be added (such as `between(m,n)` or `atLeastOnce()`); the qualifiers should be available for both in-order and standard verification. Again, these 'keywords' will not be available for auto-completion.

```
mockedList.add('one');
mockedList.clear();
```

```
mockedList.verify.add('one');
mockedList.verify.clear();
```

```
mockedList.add('once');
mockedList.add('twice');
mockedList.add('twice');

mockedList.verify.add('once');
mockedList.verify.add('twice').times(2);
mockedList.verify.add('never happened').never();
mockedList.verify.add('twice').atMost(5);
```

```
mockTwo.verifyZeroInteractions;
```

In-order verification will be handled with the creation of a separate object, which will be able to combine an arbitrary number of mocks. The `InOrder` object will have to be created after the desired calls on mocks are performed. Additionally, it needs to take a mock as an argument, so that it can differentiate when more than one mock is involved in the verification. Following *mockito* design principles [4], not all interactions have to be verified, just those the user is interested in.

```
firstMock.add('was called first');
secondMock.add('was called second');

inOrder = InOrder(firstMock, secondMock);

inOrder.verify(firstMock).add('was called first');
inOrder.verify(secondMock).add('was called second');
```

3.3 Matchers

Although stubbing and verifying exact argument values is useful, extra flexibility is often required. This can be achieved with the use of argument matchers, which allow the user to specify a rule or pattern the argument must match, instead of the exact argument value. Commonly used matchers might be `anyInt()`, `containsSubstring()` or `greaterThan()` — the possibilities are vast. Once a matcher is created, it can be used both when stubbing and verifying.

3.3.1 *mockito*

mockito offers some built-in matchers, and an interface to the popular *Hamcrest* library [10]. Most matchers offered by default are of the `any*` variety, where `*` is a standard Java type (for example, `anyInt()`, `anyFloat()`, `anyString()` etc.[1]); the interface to Hamcrest is provided by the `argThat` matcher, and a few other commonly used matchers are available: `contains()`, `startsWith()` and `endsWith()` work with Strings and do direct checking, while the `matches()` matcher can match a regex on a String. An important limitation is that if matchers are used, all arguments need to be provided by matchers; this can make tests more verbose than necessary. The syntax is the same as with normal use, with the Matcher object inserted instead of an argument:

[1]) This is due to Java enforcing type-checking

```
when(mockedList.get(anyInt())).thenReturn("element");

System.out.println(mockedList.get(999)); // prints "element"

verify(mockedList).get(anyInt());
```

3.3.2 *mockito-python*

Although the *Hamcrest* library has a port to Python as well, *mockito-python* does not provide an interface to it and offers just two matchers: `any()` and `contains()`. The `any()` matcher can match any single argument, or an argument of a given type; the `contains()` matcher is used for substring matching. As in *mockito*, matchers can be used both in stubbing and verification. Syntax examples:

```
when(mockedList).get(any(int)).thenReturn("element")

print mockedList.get(999) # prints "element"

verify(mockedList).get(any(int))
```

3.3.3 *mmockito*

mmockito will also have to offer matchers, as they vastly expand usage possibilities. They might be considered the 'killer feature': although stubbing and verification are the basic functionality, without matchers they are hard to use effectively. Although there is no *Hamcrest* port to MATLAB, the `matlab.unittest` package available since release R2013a [7], contains `Constraints`, which effectively act the same as matchers. However, as their main focus is testing, they contain additional methods for diagnostic purposes, and the naming scheme is somewhat unfortunate in the mocking context. Therefore, it was decided to provide custom matchers. Also, there is no MATLAB-specific reason to prohibit the mixing of constant arguments and matchers as in *mockito*, so this shall also be supported.

Comparing the matchers offered by *mockito* and *mockito-python*, as well as considering possible use cases, it was decided to offer two basic matchers: `Any()` and `ArgThat()`. As MATLAB, like Python and unlike Java, doesn't force type checking, the `Any()` matcher can act like the one from *mockito-python*, matching either any single argument or an argument of a given class. The `ArgThat()` matcher is an interface to the `Constraint` interface[1]), so that existing `constraints` can be easily reused. Writing custom matchers should also be easy — it should be enough to just provide a single method, `matches()`. This ease ought to compensate for the few matchers present by default; if need arises, it will be trivial to distribute more matchers.

```
mockedList.when.get(Any('numeric')).thenReturn('element');

mockedList.get(999) % ans = 'element'

mockedList.verify.get(Any('numeric'));
```

[1]) The word 'interface' has no direct meaning in MATLAB; it is used in this text to refer to a class with only abstract methods

3.4 Spying and partial mocks

Instead of creating a new object, it can be useful to start with an existing object and only stub some calls; verifying calls on real objects is also sometimes necessary. In some literature, mocks which just verify existing objects are called spies, while those which perform stubbing are called partial mocks. In *mockito*, both of these features are offered via the same API, with the naming difference stemming from intent — as with stubbing and verification on 'normal' mocks, spying and partially mocking the same object is considered bad test design. Generally, the authors of *mockito* note that spies are useful, but should be used carefully and occasionally [8] — acceptable use being mocking of 3rd party interfaces or other legacy code. For the sake of clarity, this feature will be referred to as 'spying'.

3.4.1 *mockito*

In *mockito*, it is possible to create spies of real objects: using the spy will then call the real methods, unless they were also stubbed. They are created specifically as a `spy` object, not a `mock` object, but otherwise behave the same as mocks:

```
List spy = spy(new LinkedList());

when(spy.size()).thenReturn(100);

spy.add("one");
System.out.println(spy.get(0)); //prints "one"
System.out.println(spy.size()); //prints 100

verify(spy).add("one");
verify(spy).size();
```

3.4.2 *mockito-python*

mockito-python also supports spying of real objects, with the usual syntatic differences. The documentation is less detailed on possible limitations, other than a warning that some features will not work as expected and that real object spying shouldn't be relied upon.

3.4.3 *mmockito*

From inspecting the previous two projects, it is obvious that spying offers additional implementation challenges. Nevertheless, it is a useful feature and an attempt should be made to support it. Instead of creating a new object class with similar features, it was decided to offer the capability through the single existing `Mock` class; the argument passed to the constructor will be the real object to be mocked. When stubbing methods which already exist, the stubbed call will be preferred. All other features will be available to these real mocks as well.

```
spy = Mock(realInstance);

spy.when.size().thenReturn(100);

spy.realSetter('arg');
spy.realGetter(); % ans = 'arg'
spy.size(); % ans = 100
```

```
spy.verify.realSetter('arg');
spy.verify.size();
```

3.5 Strict and Tolerant mocks

Since verification is to be offered through the same class as stubbing, it is important how the `Mock` object will behave when called with a method it does not know about: throwing an error might be the expected behavior if stubbing the method, yet this would mean that all verifications would have to be stubbed beforehand. For verification to function as expected, mocks would have to silently pass on unknown method calls. To reconcile these two different needs, it was decided to allow two modes of operation for `Mock` objects: `strict` `Mocks` will throw an error when called with an unknown methods, while `tolerant` `Mocks` will just silently pass. For verification to work as expected, `Mocks` will be `tolerant` by default, which is also the behavior offered by *mockito*.[8][1])

Although modern mocking frameworks like *mockito* have moved towards using tolerant mocks by default, there are still use-cases for strict mocks as well. Since not all interactions have to be verified in *mockito*, strict mocks can be useful as a form of sanity-check against typos and other programmer errors. MATLAB's equivalent of null is an empty array, `[]`, and if working with such values it might also be useful to prefer a strict mock. Nevertheless, tolerant mocks are the default and will be preferred in most cases.

In *mmockito*, this property of `Mocks` will be set as a string parameter to the constructor. This parameter should come last, after the optional argument of the class to be stubbed. If omitted, `tolerant` is assumed.

```
m = Mock();
m2 = Mock(RealClass, 'strict');
```

[1]) In *mockito*, such mocks are referred to as 'nice Mocks'.

Chapter 4
Usage

In order to demonstrate usage on a practical example, a use case was constructed. The presented use case is a simplified version of a real world program, partly designed to quickly showcase some capabilities of mock objects. The algorithm presented is simple enough and no domain knowledge is necessary.

4.1 Problem specification

Our goal is to implement a new algorithm, the *First Improvement Local Search*, which can be expressed with the following pseudocode:

```
currSol <- initSol
currCost <- evaluate(currSol)
logger.logInitialized;

while not termCondition():
    newSol <- perturb(currSol);
    newCost <- evaluate(newSol);
    if newCost < currCost:
        currSol, currCost <- newSol, newCost
        logger.logImproved;

logger.logFinished;
return currSol, currCost
```

The algorithm requires five input arguments: an initial solution, function handles to methods `evaluate`, `perturb` and `termCondition`, and a logger object. The logger should implement an interface, `IOAlogger`[1]), with three methods: `logInitialized()`, `logImproved()` and `logFinished()`.

If we were to program the above as a function, we could use it directly and receive a result. However, let us assume we also want to be able to call it programatically, and step by step. Therefore, we will implement it as a class, with four methods: `initialize()`, `step()`, `run()` and `getBestSolution()`. The implementation will not be tied to any particular problem space, in order to avoid increasing its complexity; the solution objects are not further specified for the same reason.

4.2 Tests

Tests are written using the new `matlab.unittest` framework: the test class inherits from `matlab.unittest.TestCase` and the `assertEqual` method it provides is used. The tests are run with the following command:

```
run(TestSuite.fromFile('FirstImprovementLocalSearchTest.m'))
```

[1]) IOA stands for Iterative Optimization Algorithm

Our first requirement is that, if the terminating condition is fulfilled immediately, `getBestSolution()` returns the initial solution. This requirement can be formulated as a test in the following manner:

```
function returnsInitSolution_whenTermConditionTrue(testCase)
    % Given
    m = Mock();
    m.when.termCond().thenReturn(true);
    mockLogger = Mock();

    alg = FirstImprovementLocalSearch('firstSol', @m.evaluate, ...
            @m.perturb, @m.termCond, mockLogger);

    % When
    alg.initialize();
    alg.run();
    [sol, c] = alg.getBestSolution();

    % Then
    testCase.assertEqual(sol, 'firstSol');
end;
```

At first we create a tolerant mock, which allows us to call any method on it. We can then pass the algorithm constructor handles to these non-existing methods. Stubbing is used to return `true` in the appropriate function call. A separate mock object is passed to stand as the logger, but is unused. This test case could also be written without using mock objects, by replacing the `@m.termCond` function handle with an anonymous function always returning true, `@()true`.

The second requirement we have for our class is that the logger object must be properly called during initialization and finalization. The test may look like this:[1])

```
function verifyLogger_whenTermConditionTrue(testCase)
    % Given
    m = Mock();
    m.when.termCond().thenReturn(true);
    mockLogger = Mock();

    alg = FirstImprovementLocalSearch(0, @m.evaluate, ...
            @m.perturb, @m.termCond, mockLogger);

    % When
    alg.initialize();
    alg.run();

    % Then
    inOrder = InOrder(mockLogger);
    inOrder.verify(mockLogger).logInitialized();
    inOrder.verify(mockLogger).logFinished();
end;
```

While the stubbed method `termCond()` can again be replaced with an anonymous function, the in-order verification feature is used to assure that the logger correctly re-

[1]) While the two tests could be done in the same function, separating them in this way allows us to to test different functionalities separately, so that failure of one of them does not affect the test for the other.

ceives messages from the algorithm. This functionality could also be tested without the use of mock objects, but would require additional programming effort. The advantages of the library are beginning to show.

A further requirement is that the algorithm correctly informs the logger an improvement to the initial solution was found.

```
function verifyLogger_whenResultImproved(testCase)
    % Given
    m = Mock();
    m.when.termCond().thenReturn(false).times(3).thenReturn(true);
    m.when.evaluate(Any()).thenReturn(2).thenReturn(1);
    mockLogger = Mock();

    alg = FirstImprovementLocalSearch(0, @m.evaluate, ...
            @m.evaluate, @m.termCond, mockLogger);

    % When
    alg.initialize();
    alg.run();

    % Then
    mockLogger.verify.logImproved().times(1);
end;
```

This example shows more clearly the advantages of mock objects: the method `termCond()` is stubbed to allow for three iterations of the algorithm, that is `false` is returned thrice and only then `true`. The `evaluate` method is stubbed, using the matcher `Any()`, to accept a call with any argument and return first `2` and then `1` for all subsequent calls. We have thus easily prepared the required state: we know that several interactions will be conducted and that an improvement will be made during the run. The final verification on the `mockLogger` objects controls that the logger has received the appropriate message exactly once.

Reimplementing this test without mocking tools would be much harder: we would need to program a separate function for the termination condition, which would use a persistent variable to track the number of calls made; similarly for the `evaluate` function. The logger would have to be a separate class, which implements at least the `logImproved()` method, and also count the number of times it was called. All that would be far more work than using the above methods of mocks.

4.3 Summary

The test cases presented do not cover all the requirements for the class, which is not even implemented fully. They represent some tests from an interim stage, where some features are present but the functionality is not yet finished. They were designed to show that while testing without a dedicated mocking framework is possible, mock objects make writing tests much easier.

Chapter 5
Implementation

The project was implemented while mostly following test-driven development practices: acceptance tests were written first, then the features required were added. If the feature was more complex to implement than expected, additional unit tests were added to test for all corner cases. As such, most features are well tested, though tests are mostly missing for error handling code paths. A general guiding principle was to offer a more limited feature set and expand on this once the implementation is better understood; with the implication that it is better to add features later on than to remove them if they prove infeasible.

Initial code had just a single special class, `Mock`, with just an overridden `subsref` method; following the test-driven development process, a class hierarchy was slowly developed. In this way, the final resulting code is not over-engineered or over-designed, but there is still a reasonable degree of separation to make current code easier to understand and further programming simpler. The presence of thorough tests meant that there would be no regression in functionality during this refactoring, which further sped-up development.

Most commonly used classes are intended to be added directly to the MATLAB search path. Some less common matchers are provided in a separate `mmockito.matchers` package while classes not intended for end users are kept in a `mmockito.internal` package, per MATLAB tradition.

This chapter discusses all the classes in the current, final codebase.

5.1 Internal classes

5.1.1 Invocation

`Invocation` is a value class encapsulating the `substruct` which defines a method invocation. There is no difference between two `Invocations` created with the same `substruct`. The only method provided is a constructor, which checks that we have defined a valid method call. Invocation is a convenience class, to make working with `substruct` information easier. The constructor checks are a form of sanity check for the rest of the codebase — the user is not expected to directly create `Invocations` when creating mocks.

5.1.2 InvocationPattern

`InvocationPattern` converts a given `Invocation` to use `Matchers`. The constructor accepts an `Invocation` (which assures data correctness), and inspects its arguments. If the arguments are already `Matchers`, it leaves them as is; otherwise, they are converted to a `Matcher` matching a constant, which is the most common use-case. The special case of a method call with no arguments is also handled. Whereas `Invocation` encapsulates all data, the `InvocationPattern` constructor disregards irrelevant data (which are

uniform already due to passing through `Invocations'` constructor) and keeps just the function name and the list of `Matchers`.

The only other method provided is `matchedBy`, which returns true if a given `Invocation` matches the pattern represented by the `InvocationPattern`. It compares the function name and number of arguments, before passing control to `Matchers`. An `Invocation` is considered matched when these checks are passed and each individual `Matcher` is satisfied.

5.1.3 VerificationError

VerificationError is a simple value class which inherits MException. It is intended to be thrown on a verification failure and to contain any relevant information; currently, it just returns a MException object with the appropriate error message. While this might go against the goal of avoiding over-design, we know more functionality will be required from the class in the near future.

5.2 Matcher

`Matcher` is the interface for comparison in mocked arguments. It contains just one abstract method, `matches`, which should return true if a given actual value satisfies the constraints represented by the `Matcher`. The `Matcher` interface is modeled on the `Constraints` interface in the new `matlab.unittest` module, taking one abstract method from it and removing the other, which is used for diagnostics and is not relevant for *mmockito* usage. Removing that method makes the creation of custom matchers easier; a further advantage is that we can provide our own matcher names, which make more sense in our domain-specific language. Not strictly depending on the new module also allows the framework to be usable in older MATLAB versions.

While some commonly used matchers are provided, users are expected to create their own matchers for more complex uses. It is possible in general to emulate the Hamcrest library,[10] which is recommended in order to provide commonality with other mocking frameworks. Boolean combining of matchers is not currently supported, as it would unnecessarily complicate the syntax — equivalent functionality can be achieved by writing an appropriate custom matcher.

`Matchers` can be used both when creating stub methods and when verifying method calls, in line with the *mockito* design philosophy of general-purpose objects.

5.2.1 Any matcher

`Any` is a versatile matcher providing two distinct modes of operation: when created with no arguments, `Any` will match any argument; when called with a class name (as a string) or a `meta.class` object, it will match any argument of that class; more precisely, it will match an object fulfilling an `isa` relationship. Both of these are available in a single matcher to allow for better readability when stubbing calls. Internally, `Any` mimics the code of the `IsInstaceOf()` constraint from the `matlab.unittest` module. Although the name is very close to the MATLAB builtin `any`, they are used in completely different contexts and a collision is very unlikely to occur; following existing *mockito* naming was deemed more important.

5.2.2 AnyArgs matcher

`AnyArgs` is a matcher which can match an arbitrary number of arguments, including zero. Since MATLAB functions commonly accept many arguments via the `varargin`

special attribute, it was necessary to provide a way to match such calls. The `AnyArgs` matcher can be combined with other matchers, but always has to be the last matcher used, to prevent it from just catching all the arguments. `AnyArgs` is somewhat of a special case since its implementation is hidden inside the InvocationPattern class, so users cannot create matchers like it.

5.2.3 ArgThat matcher

`ArgThat` is an adapter to `matlab.unittest.constraints` — as matchers and constraints are fundamentally the same, it is useful to have a way to repack any `Constraint` as a `Matcher`. This allows the user to reuse any of the existing `Constraints` (of which there are over 30), as well as any custom ones they might have written.

When initially implementing the class, much thought was given to avoiding the import of unneeded `Constraints`, with the goal of avoiding performance penalties; the author had encountered similar issues in another project. After several unsuccessful attempts, a benchmark was written to actually test the import speed — it turned out that importing a single class or a whole package takes the same amount of time. This is a useful example showing that prematurely trying to optimize for speed should be avoided.

5.2.4 ArgEqualTo

`ArgEqualTo` is a special case, because it is kept in the `mmockito.internal` package and is not intended for direct use by end users. When an `InvocationPattern` is passed a constant, it needs to convert it to a `Matcher`: the easiest solution is to use an `ArgThat(IsEqualTo(...))` construct, however, this is not available under earlier MATLAB versions. Instead, it creates an `ArgEqualTo` matcher which tries creating a `Constraint`, but falls back to a simpler matcher if `matlab.unittest.constraints` are not available.

This simpler matcher is implemented directly inside `ArgEqualTo` and handles only primitive types, as MATLAB doesn't by default support comparisons of cells, structs or other more complex objects. It is intended as a last-resort, to provide at least some functionality in older MATLAB releases; if more detailed comparison is required a custom matcher can be programmed.

5.2.5 NumberBetween, StringContaining matchers

The `mmockito.matchers` package provides two additional matchers, which provide the same features as some constraints. Their purpose is to demonstrate the creation of custom matchers as two simple examples. They show how it might be useful to validate constructor parameters and hint at a readable naming scheme (`StringContaining` matcher versus the `ContainsSubstring` constraint). If need arises, further matchers can be put in the same package and released together with *mmockito*.

5.3 Mock

Mock is the most important class, providing almost all the features implemented, including both stubbing and verification. The key method is the overridden `subsref`; occasionally, control is passed to helper functions in order to simplify the code and make it more readable. Defining separate methods also allows MATLAB to offer autocompletion for them. Various properties are also defined, though some are relevant

only when verifying and some only when stubbing.[1]) None of these methods can even be called directly, as `subsref` will intercept all calls; properties can be accessed if necessary.

The key property for stubbing is `mockery`, which is a cell array of 3-tuples: (`InvocationPattern`, `result`, `numberOfCalls`), where `InvocationPattern` represents the stubbed method call, `result` is a cell array of what should be returned in case it is matched and `numberOfCalls` is how many times the call should be matched (accepting `inf` for infinite stubbing). The `mockeryLength` property is a helper variable for readability, containing the size of `mockery`.

For the purposes of verification, all method calls made on the mock object are stored as `Invocations` in the cell array `allInvocations`, alongside their `invocationID`, which is a unique identifier across all mocks, and is used in in-order verification. The `strict` property is a boolean, `true` if the Mock is strict (and should accept only stubbed methods) and `false` otherwise. In case a real object is mocked, the `realMocked` property is set to true and the `mockedObj` property will be a reference to that objects handle.

5.3.1 Constructor

The Mock object constructor takes either zero, one or two parameters: the first parameter is the real object mocked, and can be omitted if not mocking an existing object; the second parameter is either the string `'strict'` or `'tolerant'`, with tolerant mocks being the default if omitted. This is implemented as an if-then-else chain, but should probably be rewritten to use MATLAB's `inputParser` class, especially if more (optional) parameters are needed.

5.3.2 subsref

As noted, the `subsref` method contains most of the functionality. It checks the passed substruct and passes control to the appropriate method if the first argument is one of `when`, `verify` or `verifyZeroInteractions`. Otherwise, it tries to handle it intelligently: all calls are transformed to an `Invocation` and added to the `allInvocations` list for verification purposes; a persistent variable is used internally to assign a unique, ascending identifier to all calls across all created mocks for in-order verification. Then, `subsref` checks if the requested `Invocation` can be matched by one of the stubbed calls in `mockery` — if yes, the appropriate result is returned; multiple return values are supported.

If no matching entry in `mockery` is found, `subsref` will defer to the builtin `subsref`, either of the real mocked object if one exists, or the `subsref` of the `handle` superclass. If an exception is generated, it will be rethrown if the `Mock` is strict and ignored otherwise.

5.3.3 when

The `when` method is used for handling stubbing calls. It converts the first of the passed substruct to an `InvocationPattern`, and then accepts an arbitrary number of `thenReturn`, `thenPass` or `thenThrow` keywords, each of which can optionally be followed by a `times` keyword specifying the number of times the stubbed result will be provided for a matching `Invocation`. Implicitly, the last stubbed call is mocked infinitely many times, unless specified with the `times` keyword. Once read, the appropriate tuple is inserted into `mockery`.

[1]) Using the same Mock for both stubbing and verification is usually a sign of bad test design

5.3.4 verify

When verifying, the `verify` method creates an `InvocationPattern` from the substruct information and finds the number of calls in `allInvocations` which can be matched by it. It then checks for the existence of one of the keywords, `times`, `atLeast`, `atMost` or `never` and compares them to the number of matching calls; if no keyword is given, `atLeast(1)` is assumed. Successful verifications will pass silently, while failures will throw a `VerificationError`.

5.3.5 verifyZeroInteractions

A special case, `verifyZeroInteractions` passes if the `allInvocations` property is empty.

5.4 InOrder

The `InOrder` class is provided to support in-order verification. Although `Mock` can verify, in-order verification also needs to be supported across multiple `Mocks` so another class had to be created. Class design is modeled after `Mock`, with an overridden `subsref` and helper methods, though they are simpler as `InOrder` objects have just one responsibility. The only two properties are `allInvocations`, which is similar to the same property in `Mocks`, except each row also contains a reference to the `Mock` it was copied from; and `currentPos`, which notes how far down the `allInvocations` list have we gotten, so that proper in-order verification can be performed.

5.4.1 Constructor

The constructor takes an arbitrary number of `Mocks` and inspects their `allInvocations` properties. Each row is appended a reference to its `Mock` and added to `allInvocations` property of the `InOrder` object. This list is then sorted based on `InvocationID`, using the MATLAB builtin method `sortrows`. As `InvocationID` is unique and always ascending across all `Mocks`, sorting on it gives us a canonical ordering and allows in-order verification.

5.4.2 subsref

The `subsref` method has no need to do any processing itself, just pass control to the appropriate sub-function. However, it needs to be implemented, as the contents of the passed `substruct` are important and cannot be accessed otherwise. Although all methods are simpler and shorter compared to Mock, and could conceivably be kept in a single method, it was deemed that mimicking the design of `Mock` is a better choice from a maintainability viewpoint.

5.4.3 verify

The `verify` method handles most of the work, by inferring which `Mock` needs to be verified, creating an `InvocationPattern` and searching through the `allInvocations` list. The main difference compared to `Mock` is that instead of checking from the start every time, it only checks from the last reached position, which is stored in the `currentPos` property. This allows for verification to proceed in order. Not every method call has to be verified, but those that are must be called in order; furthermore, `times` and other special keywords supported by `Mock` currently cannot be used on `InOrder objects`. If no mock with a matching method call is found, or the end of the list was already reached, a `VerificationError` is thrown.

Chapter 6
Current limitations

Despite much effort, the current architecture has several limitations. These are not features which have not been implemented yet, of which there are a few, but rather issues which cannot even be solved with current MATLAB capabilities, or whose solving would require a complete rewrite of the codebase. While some are just minor inconveniences, others can lead to unexpected results occasionally. Most of these limitations could be protected against with better placed error messages, but few such checks are currently implemented.

This chapter describes known limitations, in no particular order.

6.1 Method calls

As covered in section 2.4, there are two ways of calling methods in MATLAB: `method(obj, args)` and `obj.method(args)`. As only the latter passes through `subsref`, it is the only syntax supported by *mmockito*. There is no way of using mock objects with the `method(obj, args)` syntax. It is not the intention of the author to force a specific coding style on *mmockito* users, but supporting both styles is not feasible.

6.2 Methods with no arguments

If a method has no arguments, it can be called with both `obj.method` and `obj.method()`, while the first method also ambiguously referring to a property as well. With the current implementation, calls to `subsref` with length of one will not be checked against the mocked methods list. Automatically assuming all such calls were to a method without arguments would make it impossible to access any property of the `Mock`, since they would never get passed to the builtin `subsref` method; this is especially worrisome when creating a mock from an existing object. When working with a tolerant mock, such a call can pass silently, returning an empty array. While this result might be expected by those familiar with the implementation details, it could also confuse users.

As there is no single behavior which would satisfy all use cases, it was decided not to support method calls of the `obj.method` type, instead requiring all method calls to be made with brackets, `obj.method()`.

6.3 Mocking abstract classes

When creating a `Mock` from an existing object, an actual instance has to be passed, so that unmocked methods can be called on it. It is not currently possible to mock classes with abstract methods. This is partly a design issue — does mocking an interface mean only the abstract method declared can be mocked, or something else? Nonetheless, equivalent functionality can be achieved by generating a 'standard' mock.

6.4 Return arguments

Although the current limitation supports the stubbing of methods with multiple return arguments, these methods have to be called with the exact number of return arguments they are defined with. If called with fewer arguments, only as many results will be returned — this can lead to subtle errors, especially if the function returns differing number of outputs based on the input. Currently, an error is thrown if the expected number of results does not match the actual.

Conversely, the current implementation assumes only one result will be returned if calling the builtin `subsref`, which happens if no appropriate stubbed method is found. This can lead to problems when working with partial mocks.

6.5 Unmockable method names

Since the custom `subsref` method has special handling for specific method names, these methods cannot be mocked.[1] This applies for `when`, `verify`, `verifyZeroInteractions` and `verifyNoMoreInteractions`, as well as any methods of `Mock` which might be added in the future. The same limitation does not apply for other used keywords, such as `times`, `never`, `thenPass`, `thenReturn`, and others. Overcoming this limitation would require a complete overhaul of the current architecture, which is not worth the effort and will not be attempted.

6.6 Autocomplete

As most modern integrated development environments (IDEs), the MATLAB editor provides autocompletion for variable names, method calls, properties and others. After typing the first few letters, MATLAB will automatically offer some suggestions on what it thinks the user means; this can significantly speed up programming. Since *mmockito* uses custom processing of method names via `subsref`, this feature will only work for the methods `when`, `verify`, `verifyZeroInteractions` and `verifyNoMoreInteractions`, and none of the other keywords will be. Chained method calls are not fully supported by MATLAB, so even defining a custom method would not allow it to be autocompleted unless its the first method. There is no way to override the MATLAB autocomplete functionality.

[1]) Neither verified, nor stubbed.

Chapter 7
Conclusion

The goal of this thesis, to write a functional mocking library for MATLAB, was successfully accomplished. The programmed library, *mmockito*, provides most features offered by mocking frameworks in other languages, while taking into account the specifics of the MATLAB language. The existing limitations can be worked around and represent a minority of possible uses. As such, the library is considered mature enough to be used in other research projects.

Although alternative mocking philosophies exist, I consider the chosen approach to be an excellent first point. More than ten years after such practices were introduced in software development, the test-driven development process can finally be applied fully when using MATLAB with the advent of this library. It is now possible to further investigate its advantages and disadvantages in the context of scientific computing and MATLAB in general; making the code open-source will further expedite this process.

7.1 Future work

The code is now at a state in which it can be published and distributed for wider use. A suitable open-source license should be found and the necessary legal steps performed to allow for the open-sourcing of the produced work. All code, current issues and developer documentation should be made available in a suitable, open, manner. Once these steps are made, the code should be distributed through the appropriate channels. After use in real-world projects, further development direction can be decided upon.

References

[1] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1st edition, 2009.

[3] Martin Fowler. Mocks aren't stubs, January 2007. `http://martinfowler.com/articles/mocksArentStubs.html`, URL accessed on 2013-03-27.

[4] *mockito* [software]. Documentation: Features and Motivations. `https://code.google.com/p/mockito/wiki/FeaturesAndMotivations`, URL accessed on 2013-04-24.

[5] Steve Eddins. MATLAB xUnit test framework [software], 2008. `http://www.mathworks.com/matlabcentral/fileexchange/22846-matlab-xunit-test-fram ework`, URL accessed on 2013-05-20.

[6] MathWorks. File Exchange. `http://www.mathworks.com/matlabcentral/fileexchange/`, URL accessed on 2013-05-20.

[7] MathWorks. MATLAB release notes. `http://www.mathworks.com/help/matlab/release-notes.html`, URL accessed on 2013-04-03.

[8] *mockito* [software]. Documentation: Mockito API. `http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html`, URL accessed on 2013-05-14.

[9] *mockito-python* [software]. Wiki documentation. `https://code.google.com/p/mockito-python/w/list`, URL accessed on 2013-05-14.

[10] *Hamcrest* [software]. `http://hamcrest.org/`, URL accessed on 2013-05-15.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Vladimír Perić

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: MATLAB Mock Library

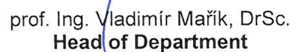
Guidelines:

1. Describe shortly the test-driven development process and the purpose and use of mock objects.
2. Design and implement a Mock library for MATLAB with a focus on creating a "test spy" framework that includes support for stubbing and behaviour verification.
    a. Review multiple APIs commonly used with other Mock libraries and decide which of them will be used for the library.
    b. Equip the library with a comprehensive set of unit and integration tests.
    c. Document the library usage thoroughly.
3. Evaluate the library in the context of a real-world project and assess its contributions.

Bibliography/Sources:
[1] Gerard Meszaros - xUnit Test Patterns: Refactoring Test Code - 2007
[2] Steve Freeman, Nat Pryce - Growing Object-Oriented Software, Guided by Tests – 2009

Bachelor Project Supervisor: Ing. Petr Pošík, Ph.D.

Valid until: the end of the winter semester of academic year 2013/2014


prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2013

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Vladimír Perić

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Knihovna maket objektů (Mock) pro MATLAB

Pokyny pro vypracování:

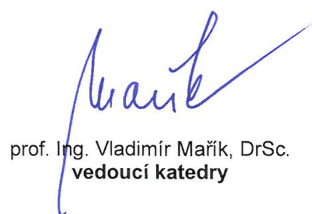
1. Seznamte se s procesem vývoje softwaru řízeným testy a s účelem a použitím maket objektů (zástupných objektů, mocků).
2. Navrhněte a implementujte knihovnu maket objektů pro MATLAB. Zaměřte se na podporu tzv. stubbingu a verifikace chování.
   a. Seznamte se s různými API, která se běžně používají v podobných knihovnách pro jiné jazyky, a vyberte mezi nimi API pro vytvářenou knihovnu.
   b. Důkladně knihovnu otestujte prostřednictvím sady jednotkových a integračních testů.
   c. Vytvořte ke knihovně důkladnou dokumentaci.
3. Vyzkoušejte knihovnu v rámci reálného projektu a posuďte její přínosy.

Seznam odborné literatury:

[1] Gerard Meszaros - xUnit Test Patterns: Refactoring Test Code - 2007
[2] Steve Freeman, Nat Pryce - Growing Object-Oriented Software, Guided by Tests – 2009

Vedoucí bakalářské práce: Ing. Petr Pošík, Ph.D.

Platnost zadání: do konce zimního semestru 2013/2014



prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2013