MoCaml: the Mock Test Pattern for OCaml

Mike McClurg

Kellogg College
University of Oxford

A dissertation submitted for the
MSc in Software Engineering

Abstract

Object mocking is a technique to assist programmers in writing unit tests, by replacing hard-to-test dependencies with “mock” implementations of those dependencies. A mock object allows the developer to set certain expectations about how the system under test will interact with the dependency, and then verify that these expectations hold true at the end of the test. While this technique was developed as a test design pattern for object oriented languages, it can also be useful for functional programming languages, especially for applications in which hard-to-test side effects can’t easily be factored out of impure functions. This dissertation will explore the current state of the art tools and techniques for unit testing and mock object generation in object oriented languages. It will make the case that these techniques can be of use in functional programming languages, and it will describe the implementation and use of a new mocking library for the OCaml programming language.

This thesis is dedicated to my wife, without whom I likely would never have started writing.

Contents

1 Introduction **5**
- 1.1 Motivation of this dissertation . . . 5
  - 1.1.1 Why write unit tests . . . 5
  - 1.1.2 Unit testing terms . . . 6
- 1.2 Test doubles and the mock pattern . . . 7
- 1.3 Current implementations of the Mock Object Pattern . . . 8
- 1.4 Objectives and expected contribution . . . 9

2 Background **12**
- 2.1 A brief tour of OCaml and its module system . . . 12
  - 2.1.1 OCaml basics: values, functions, and types . . . 12
  - 2.1.2 The OCaml module system . . . 15
  - 2.1.3 The rest of OCaml . . . 17
- 2.2 Test Double patterns in detail . . . 17
  - 2.2.1 Fake Object pattern . . . 20
  - 2.2.2 Dummy Object pattern . . . 20
  - 2.2.3 Test Stub pattern . . . 20
  - 2.2.4 Test Spy Object pattern . . . 20
  - 2.2.5 Mock Object pattern . . . 21
- 2.3 Tools for software testing in functional langauges . . . 23
  - 2.3.1 xUnit implementations . . . 23
  - 2.3.2 Specification-based random testing with QuickCheck . . . 25
  - 2.3.3 Criterion: Haskell benchmarking library . . . 26

3 Application **30**
- 3.1 Dependency injection methods in OCaml . . . 30
- 3.2 Design considerations for mock modules . . . 32
- 3.3 The simplest mock module implementation . . . 32
- 3.4 Specifying a mock's expectations . . . 34
  - 3.4.1 Module organisation . . . 35
  - 3.4.2 Generalised Algebraic Data Types . . . 35
  - 3.4.3 Verifying expectations . . . 35
- 3.5 Automatically generating mock modules . . . 36

4 Reflection **39**
- 4.1 Comparison of mocking frameworks . . . 39
  - 4.1.1 Notes . . . 39

List of Figures

2.1 Taxonomy of Test Double patterns 18
2.2 Indirect Inputs 19
2.3 Indirect Outputs 19
2.4 Criterion Output 27

1 Introduction

This dissertation will explore the use of a unit testing technique called the "Mock Object Pattern" in the strongly, statically typed functional programming language OCaml. Mocking (or object mocking, as it is more commonly known) is method of replacing hard-to-test dependencies in object-oriented code with automatically generated objects in order to make it easier to write unit tests. The technique was first applied for use in Java, which is an object oriented language with static types and run-time instrumentation facilities.

The dissertation is organised as follows. Chapter 1, Introduction, briefly describes the topic of software testing and introduces the topic of Mock Objects. Chapter 2, Background, describes software testing in more detail, including different levels of testing, common test design patterns, and tools commonly used for testing. Chapter 3, Application, describes the implementation of a tool, MoCaml, to assist OCaml developers in automatically generating mock dependencies, and compares tests written using this tool versus other similar tools. Chapter 4, Reflection, describes the experience of writing tests using MoCaml to write tests.

1.1 Motivation of this dissertation

1.1.1 Why write unit tests

Automated testing is an important part of software development. At its most basic, software testing is a means of demonstrating the quality of software – and the costs of poor quality in software can be huge. In 2002, the US National Institute of Standards and Technology estimated that poor software testing practices may cost the US between \$22.2 and \$59.5 billion USD per year (not including the costs of "mission critical" software whose failure could lead to catastrophic failure or loss of life). Furthermore, studies have shown that the cost of fixing a software defect in production may be up to two orders of magnitude greater than if the bug had been fixed in the development phase. [18] [3] While this "orders of magnitude" statement has been part of the software engineering lore since Boehm first made the observation in 1988, more recent analysis [2] [4] indicates that the ratio of cost between fixing a bug in development versus production may be closer to 1:5 rather than 1:100. Nevertheless, this is still a strong motivation to resolve software defects as early as possible in the development cycle.

Software tests are typically grouped into three different levels: system tests, component tests, and unit tests [13]. These categories represent a spectrum of granularity. Systems tests represent the broadest granularity, and are meant to test the entire system as a whole. On the other end of the spectrum are unit tests, which test functionality at the finest level of granularity: typically a method or function. Component tests sit somewhere in between the two ends of the spectrum, and might test interactions between modules or classes of an application, or perhaps the interaction between distinct services of a larger application. Some textbooks, particularly those describing Agile software development practices, prefer the terms acceptance and integration tests over system and component tests [8] [11]. While these terms are perhaps more descriptive of the objective of the different types of tests, in this dissertation we will prefer to use the terms system and component test, which better demonstrate *what* is being tested.

Recent trends in Agile development, such as continuous integration [11] and test driven development [1], have placed a greater emphasis on unit testing. Consequentially, many tools and libraries have been developed to assist programmers in writing these tests. Many of these libraries, such as JUnit [15] for Java, NUnit [23] for .NET, Test:Unit [28] and RSpec [27] for Ruby, and OUnit [25] for OCaml, follow the xUnit patterns of test development described by Meszaros [19]. This family of libraries typically provides a set of convenient assertion functions to help the user write test cases, annotations to help organise these test cases into larger test suites, and test harnesses to handle test running and status reporting.

1.1.2 Unit testing terms

To understand mock objects, we should first familiarise ourselves with the xUnit Pattern Language [19]. A pattern language is a high-level, often visual, framework for describing patterns that occur in software engineering. The xUnit Pattern Language has a vocabulary that aids in describing the patterns that commonly occur in unit testing. Primarily, it contains a vocabulary that covers most aspects of software engineering and software testing. The following is glossary of terms we will use in the following chapters.

System Under Test (SUT) The part of the software that is being tested. Confusingly, this doesn't necessarily refer to the system as a whole, but rather the component of the system that is actually being tested. For instance, if we are testing a particular HTTP handler of a web application, the SUT is the HTTP handler itself, not the web application as a whole.

Depended-On Component (DOC) A component of the software which is a required dependency of the SUT. For instance, the HTTP handler we are testing may make a database query through an object relational mapper (ORM). In this case, the ORM is a DOC. A SUT may have more than one DOC.

Dependency Injection (DI) The act of configuring a SUT's dependencies, either at compile-time or run-time. A dependency which is configured at compile-time is often described as a "hard-coded" dependency. Various dependency injection frameworks exist which provide facilities for configuring dependencies at run-time. For instance, a DI framework might instruct the ORM to map to a test database when a program is run in test mode, but would map to the real database when the program is run in production. This prevents the components of the software which interact with the database through the ORM from having to handle special-cases for testing purposes.[1]

Hard-to-Test Code For many reasons, some software components will be hard to test. A component may depend on external data that isn't available at test time. A component may require certain specialised hardware to run which isn't available at test time. A component may be difficult to access programmatically (such as a GUI or web interface). Or the software may be written in a way that will make tests brittle and prone to breaking (such is the case with tightly coupled components). Refactoring may help make tightly coupled code easier to test, but some components may just be intrinsically hard to test.

Test fixture Tests often require certain infrastructure to be set up prior to a test run. A test database may need to be instantiated, or the program may need to be guided into a certain state before a particular test is run. The component which handles setting up this

[1]Dependency injection is not necessarily a part of the xUnit Pattern Language, but it is defined here because dependency injection plays an important role in the construction of tests that use the mocking technique.

infrastructure is known as the test fixture (sometimes referred to as a test harness). Test fixtures often use a form of dependency injection to set up tests.

Test double It may be impossible to test the SUT because it contains a DOC which is hard-to-test. In this case, the test double pattern can be used. A test double is a replacement for a hart-to-test DOC which makes it possible to test the SUT. xUnit defines the following test doubles: Test Stub, Test Spy, Mock Object, and Fake Object. In addition to these types of double, test doubles may either be configurable at run time, or have their configuration hard-coded. Test doubles will be further discussed in section 2.2, where we will discuss the mock pattern in much greater depth.

xUnit Test Patterns also defines the four phases of the testing life cycle:

1. **Setup** Initialise the required state so that the unit test is ready to be run. Depending on the test case, this may involve starting an HTTP server, initiating a connection to a database and loading test data, or simply creating a test double and injecting it into the SUT. The point here is to provide context for the SUT to be executed.

2. **Exercise** This step simply executes the SUT in question within the context provided by the setup phase.

3. **Verify** Determine whether the test passed or failed by comparing the results of the execution phase to the expected results. This is typically accomplished by calling an `assert` function provided by the unit test framework being used, although it could be a more complex step such as verifying new database records, or calling a mock object's verification method.

4. **Teardown** Cleanup after the test has finished. This involves undoing anything that was done in the setup phase, and perhaps cleaning up any unwanted output generated by the SUT in the exercise phase, such as log files, etc.

1.2 Test doubles and the mock pattern

For various reasons, some code is just hard to test. Often this manifests itself in the form of hard-to-test dependencies of the SUT. Examples of these DOCs may include queries to databases, side-effecting code that is infeasible to run in a test environment, or, as in the case of Test-Driven Development [1], we are testing a SUT which has a DOC that has not yet been fully implemented.

A SUT may be hard to test not only because of a hard-to-test DOC, but because it is difficult to verify that the SUT has interacted with the DOC in an appropriate way. For instance, we may want to test that the SUT has sanitised its inputs before calling out to the ORM, but to do so would require intercepting calls to the ORM, or modifying the ORM specifically for the test case. The solution to this type of testing problem is known as Behaviour Verification [19].

The Test Double Pattern is a solution to the hard-to-test DOC problem. A test double is an implementation of the DOC's interface which can be used as a drop-in replacement for the DOC, thereby making the SUT easier to test. Test doubles come in five flavours: Dummy Objects, Test Stubs, Test Spies, Mock Objects, and Fake Objects. While each of these types of test double can be used in place of a hard-to-test DOC, each has a specific purpose that makes it better suited for particular applications.

A Mock Object has unique capabilities that distinguish it from the other test double patterns. A Mock Object implements the interface which we wish to replace, but allows the programmer to specify the expected behaviour of the Mock during the test. These expectations may specify both the expected inputs to a mocked method, as well as its expected outputs. Mock Objects provide a verification method which allows the test case to verify that the SUT interacted with the Mock Object in the expected way. Mock Objects are also typically automatically generated by the testing framework, and therefore are considered to be dynamically configured test doubles. Because of these properties, Mock Objects are very well suited for performing Behaviour Verification.

1.3 Current implementations of the Mock Object Pattern

As implied by its name, the Mock Object pattern is a technique that originated from the Object Oriented programming language community. One of the first tools implemented to assist programmers in writing Mock Objects was JMock [14] for the Java programming language. Mocking frameworks for other Object Oriented languages exist as well, such as Moq [22] for C# and .NET, and RSpec [27] for Ruby. These mocking frameworks each support the automatic generation of mock implementations of objects through the use of run-time reflection and code generation.

Reflection is a language feature which allows programs to perform introspection at run time. In the case of mocking frameworks, reflection is used to inspect a given class type (in Java's JMock and .NET's Moq) or object (in the case of Ruby's RSpec) and generate an object which implements the same interface as the original class type or object. The framework provides functions for overriding the methods of this generated object with user-provided expectations. The new object can be used in place of the DOC in the SUT, and after the test has executed, the mock's expectations can be verified.

Currently there is no implementation of a dynamically configurable mock framework for OCaml that automatically generates mock implementations of interfaces. The closest OCaml library for implementing the Mock Object pattern is Xavier Clerc's Kaputt unit testing tool [16], which provides function combinators for assisting programmers in writing hard-coded mock implementations of functions. Compared to the dynamically configurable mocks generated by the likes of JMock and RSpec, this is quite cumbersome. Furthermore, the library only provides facilities for mocking individual functions, meaning that if the user wants to write a mock implementation of an entire module or object, then the user must implement the entire interface manually, rather than just providing expectations for the few functions exercised by the test case.

Although it has neither run-time reflection nor run-time code generation[2] capabilities, OCaml still has the potential to host a dynamically configurable mocking framework, which could fully implement a mock of a given interface and allow a programmer to provide verifiable expectations for that mock. Among its many benefits, OCaml has two important features we need: a strong, static type system, and a powerful preprocessor and code generation facility called camlp4 [5]. These two features will allow us to write a fully-featured, dynamically configurable mocking framework for OCaml.

[2]While OCaml has excellent code generation facilities in the form of the `compiler-libs` library, it cannot generate at run-time code which could be executed by the same run-time environment.

1.4 Objectives and expected contribution

The objective of this dissertation is to create a dynamically configurable mocking framework for OCaml, and to describe in detail both its implementation and evaluate its use as a testing tool, in comparison with mocking frameworks for other languages (both object oriented and functional), and in comparison with other testing tools for OCaml. In the background section, we will cover software testing theory in more detail, and fully describe the Test Double patterns and their use and implementation. In addition, we will discuss the OCaml language, and explain how its features aid and hinder the implementation of a mocking framework. Finally, we will discuss the usefulness of the Mock Object pattern in a functional language compared with its usefulness in an object oriented language.

2 Background

In this section we will provide a brief tour of the OCaml programming language, focusing especially on its powerful module system. The chapter follows with a discussion of xUnit's Test Double patterns and their implementation in OCaml and Java. The chapter concludes with a summary of the current state-of-the-art testing tools for functional languages such as OCaml and Haskell.

2.1 A brief tour of OCaml and its module system

OCaml is a strongly, statically typed functional programming language based on the ML family of languages. It may be interpreted in a read-eval-print loop called the OCaml "toplevel," or it may be compiled to an OCaml specific bytecode language or to a native executable. [17] It has a very good generational garbage collector tuned for sequential programs. [9]

What follows is a *very* brief tour of the OCaml programming language. This should hopefully be enough to allow the reader to follow along with the coding examples later in this chapter and the next. For more assistance with OCaml, the reader is directed to the OCaml Language Specification [17], and Real World OCaml [20][1]

2.1.1 OCaml basics: values, functions, and types

Values

In OCaml, values are assigned to variables using the `let` keyword.

```
(* This is a comment *)
let num = 42
let str = "hello world"
let list = [1;2;3;4]
let tuple = (1,2)
```

Variables in OCaml are immutable, unless they are specially defined as references, or mutable fields of a record type.

```
let num = ref 42
Printf.printf "num was %d\n" !num
num := 24
Printf.printf "num is now %d\n" !num
```

Functions

Functions are created with the keywords `fun` or `function`, and assigned names using `let`, just like other values. Functions defined using the `function` keyword can only take one argument, but that argument can be pattern-matched.

[1]Real World OCaml is available for free online: `https://realworldocaml.org`

```
let f = fun a b -> a + b
let g = function
        | [] -> true
        | _  -> false
```

Functions can also be created using a shorthand `let` syntax. The function `f` below is equivalent to the function `f` above.

```
let f a b = a + b
```

Because functions are first-class values just like `int`s and `string`s, we can pass them to other functions. For instance, the function `List.map` takes a function `f` and a list `ls`, and returns a new list containing the result of applying `f` to each element of `ls`.

```
let f x = x + 1
let ls = [1;2;3;4]
List.map f ls          (* returns the value [2;3;4;5] *)
```

Types

OCaml is a strongly, statically typed language. The compiler will infer the type of values, and will return an error if a type constraint is violated. For instance, the following function `f` has type `int -> int -> int`, meaning that it takes two values of type `int` and returns a value of type `int`. If a `string` or `float` are passed to this function, a compilation error is thrown.

```
let f a b = a + b      (* has type int -> int -> int *)
f 40 2                 (* this type-checks *)
f "4" "2"              (* this fails to type-check *)
f 41.8 0.2             (* this also fails to type-check *)
```

OCaml allows the user to define complex types.

```
(* Tuples *)
type position = int * int

(* Variant types *)
type colours =
  | Red of int
  | Green of int
  | Blue of int

(* Record types *)
type book = {
  author : string;
  title  : string;
  mutable inventory : int;
}
```

Polymorphism in OCaml data types is expressed with type variables. In the following example, `'a` is a type variable which can represent any type, making `tree` a generic data type.

```
type 'a tree =
  | Leaf of 'a
  | Node of 'a tree * 'a tree
```

Polymorphic variants

OCaml has a feature called *polymorphic variants*. These are similar to atoms or symbols in languages like Erlang or Scheme. Polymorphic variants are distinguished from regular variants by the backtick (`) preceding the label. Unlike regular variants, polymorphic variants do not need to be capitalised, but this is still common practice. A common use of polymorphic variants is to encode inputs or outputs of functions; for instance, the below function uses polymorphic variants for encoding success and failure cases.

```
let f = function
  | `String v ->
    Printf.printf "Got a string: %s\n" v;
     `Success
  | `Int v    ->
    Printf.printf "Got an int: %s\n" v;
    `Success
  | `Float _  ->
    print_endline "Wasn't expecting a float.";
    `Failure
```

Polymorphic variants can be used in the same way as regular variant types, except that they don't need to be explicitly declared. Furthermore, any polymorphic variant with the same name is considered to be the same type (unlike regular variants, where identically named constructors may be protected inside of different modules, and therefore would be different types).

This gives rise to useful patterns in API development, such as extensible API types. For instance, the Yojson library has an encoding of standard JSON, as well as common, useful extensions to the JSON spec. The standard JSON spec is encoded in `Yojson.Basic.json`, whereas the extensions to the standard are encoded in `Yojson.Safe.json`. The `json` type in both modules uses polymorphic variants to define the json grammar, and "Basic" JSON can be easily upgraded to "Safe" JSON. A full treatment of this library can be found in Chapter 15 of Real World OCaml [20].

Pattern matching

One of the most powerful features of languages like OCaml is pattern matching. OCaml allows one to match over the structure of defined data types. Pattern matching can be accomplished with the `match value with | patt -> expr` expression, or in functions created with the `function` keyword. The following two functions are equivalent.

```
let rec num_leaves_1 = function
  | Leaf _      -> 1
  | Node (l, r) ->
    (num_leaves l) + (num_leaves r)

let rec num_leaves_2 tr =
  match tr with
  | Leaf _      -> 1
  | Node (l, r) ->
    (num_leaves l) + (num_leaves r)
```

Note the use of the underscore (_) in the `Leaf` pattern. In OCaml, underscore means "match anything." It can also be used in variable assignment, and is often used in module initialisation code.

Exceptions

OCaml has exceptions that follow the usual "try/catch" pattern. OCaml exceptions have no "finally" clause, but this can be implemented in a library using first-class functions.[2] Exceptions

[2]cf. `protect` and `protectx` in Jane Street's Core library: `https://ocaml.janestreet.com/ocaml-core/111.03.00/doc/core/#Exn`

are declared similarly to variant data types.

```
exception Not_found
let rec find x = function
  | [] -> raise Not_found
  | (k,v)::ls -> if x = k
                 then v
                 else find x ls
```

As with variants, exceptions can also carry data:

```
exception Failure of string
exception Error of [`Invalid_request of string | `Permissions]
raise (Error (`Invalid_request "foo"))
raise (Error `Permissions)
```

"try/catch" blocks in OCaml are similar to those in other languages, except OCaml pattern matches on the "catch" block using the `with` keyword.

```
try
  raise (Error (`Invalid_request "foo"))
with
  | Error `Permissions ->
    print_endline "Bad permissions"
  | Error (`Invalid_request req) ->
    Printf.printf "Invalid request: %s\n" req
```

2.1.2 The OCaml module system

Modules, signatures, and compilation units

A distinguishing feature of OCaml is its powerful module system. A module in OCaml is the basic compilation unit. Each compiled file is assigned its own module. For instance, the definitions inside a file called `foo.ml` would be placed into a module `Foo`.

```
type t = int
let f a = a
```

From outside of this module, function `f` would be referenced as `Foo.f`. Module `Foo` has the following type:

```
type t = int
val f : 'a -> 'a
```

The types of modules can be restricted using `mli` files. When an `mli` file exists with the same basename as an `ml` file, the module type of the module created for that `ml` file is restricted to that of the signature specified in the `mli` file. For instance, if `foo.mli` is:

```
type t = int
val f :    a  ->    a
```

then the resulting type of module `Foo` would be:

```
type t
val f : t -> t
```

Notice that `type t` is now abstract (meaning that we can't know its implementation outside of module `Foo`, and function `f` is now restricted to have type `t -> t`, instead of the more general `'a -> 'a`.

OCaml modules can also be defined independently of compilation units.

```
module Bar =
struct
  type t = int
  let g a = a + 1
  let h ls = List.map g ls
end
```

We can also create module type signatures, which can restrict the resulting type of a module in the same way as an `mli` file.

```
module type BAR =
sig
  type t
  val h : int list -> int list
end
```

We can limit the type of `module Bar` to the signature `BAR` by restricting it:

```
module Bar2 = (Bar : BAR)
```

We can also restrict the original definition of `module Bar`:

```
module Bar1 : BAR =
struct
  type t = int
  let g a = a + 1
  let h ls = List.map g ls
end
```

Modules `Bar1` and `Bar2` are now restricted by the signature `BAR`. The value `g` is now hidden, and `type t` is now abstract.

Functors

In most languages, a module system is primarily meant to provide a namespacing system or a way to refer to compilation units (cf. Haskell's module system [10], or Python's module system [21]). OCaml, however, has a much more powerful module system, including the ability to define *functors*.

A functor is a relation from modules to modules. A functor can take a module as an argument, and use it to specialise another module. For instance, here is an example adapted from the OCaml language "Introduction to OCaml" [17] that creates a `Set` functor.

```
module type ORDERED_TYPE =
sig
  type t
  val compare: t -> t -> int
end

module Set =
  functor (Elt: ORDERED_TYPE) ->
    struct
      type element = Elt.t
      type set = element list
      let empty = []
      let add x s = ...
    end
```

`Set` is a functor that takes some module with same type signature as `ORDERED_TYPE`, and returns a module which is specialised with that element type. For instance, we may create a new `StringSet` module using the `Set` functor and the `String` module from the OCaml standard library.

```
module StringSet = Set(String)

let set = StringSet.add "foo" StringSet.empty in
StringSet.mem "foo" set   (* => true *)
```

For brevity, OCaml supports and alternative syntax for functor definition which is more concise. For instance, we could have defined the `Set` functor as: `module Set(Elt : ORDERED_TYPE)= struct ... end`. This comes in handy when one is defining a functor which takes multiple arguments: `module F(A : X)(B : Y)= struct ... end`.

Functors are useful for creating generic data types, such as the `Set` type we defined above. We will soon see that they can also be useful for dependency injection.

First-class modules

OCaml 4.00 introduced the feature of *first-class modules*. A first-class module is analogous to a language having first-class functions: a module in OCaml can be packaged into a value, passed to functions, and returned from functions. First-class modules provide a similar capability as functors do, but are much more light-weight. Refactoring a module into a functor can be a significant amount of work, but adding a module argument to a function is a much simpler change.

```
module type DATABASE = sig ... end

module Db = struct ... end

let f db =
  (* Turn a module value into a module *)
  let module Db = (val db : DATABASE) in
  ...

(* And convert a module into a value *)
let _ = f (module Db : DATABASE)
```

2.1.3 The rest of OCaml

In addition to the module language described above, OCaml also has a rich class language and object system. Object orientation is a useful addition to OCaml, but most OCaml code in practice eschews the use of objects and classes. This dissertation will ignore the object oriented features of OCaml and instead focus on its functional core and module system. Most of the literature regarding software testing assumes an object oriented implementation language; this dissertations contributions lie elsewhere.

This was a whirlwind tour of OCaml. It was meant to cover just enough syntax to get the reader through the OCaml examples in the next section, and to understand the implementation in the following chapter. For more extensive coverage, please see the OCaml language spec [17], and the excellent Real World OCaml book [20].

2.2 Test Double patterns in detail

Test Doubles allow developers to write tests that might have been difficult or even impossible to write without a Test Double. As their name suggests, Test Doubles are meant to stand in for

[3] Adapted from Meszaros [19]

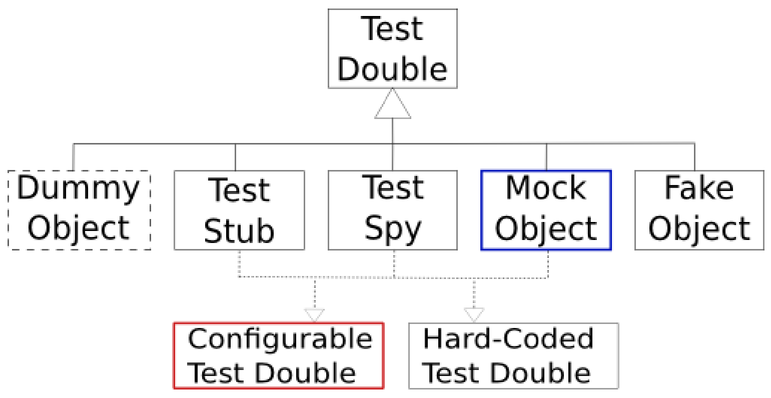


Figure 2.1: Taxonomy of Test Double patterns[3]

	Implements Interface	Indirect Inputs	Indirect Outputs	Configurable	Behaviour Verification
Dummy	No	No	No	No	No
Stub	Yes	Yes	No	Yes	No
Fake	Yes	No	No	N/A	No
Spy	Yes	No	Yes	Yes	No
Mock	Yes	Yes	Yes	Yes	Yes

Table 2.1: Comparison of Test Double pattern features

a DOC of the SUT. Their name is analogous to "stunt doubles" used in films to stand in place for the main actor during dangerous scenes [19].

Meszaros describes three problematic scenarios for which Test Doubles may be an appropriate solution.

Untested Requirement A particular SUT may have a system requirement that needs to be exercised in a unit test, but testing this behaviour requires observing interactions between the SUT and one of its DOCs. If the DOC doesn't provide a mechanism for observing these inputs and outputs, a Test Double may be used in place of the DOC to observe the indirect inputs and outputs between the SUT and the DOC.

Untested Code A SUT may depend on a DOC which does not provide the test case enough control over its inputs to properly exercise the desired behaviour in the SUT. A Test Double for the DOC may be used in order to put the SUT into the state necessary to exercise this behaviour.

Another instance of this issue occurs when the DOC has not been written yet, but a test for the SUT still needs to be written. This case often occurs in Test Driven Development (TDD) [1], where tests are usually written either before or during the development phase.

Slow Test Slow tests can be a major problem because they discourage the use of unit tests as part of the build process. A test may be slow because a particular DOC takes a long time to perform its task; perhaps it is performing a major calculation, or is downloading a file over a slow network connection. In this case, a Test Double may be used in place of the DOC in order to speed up the test cases which require this DOC.

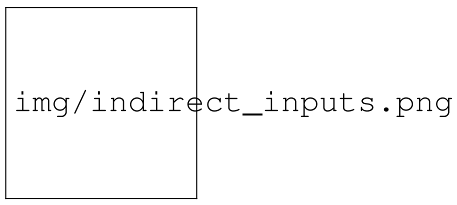


Figure 2.2: Indirect inputs to a SUT[4]

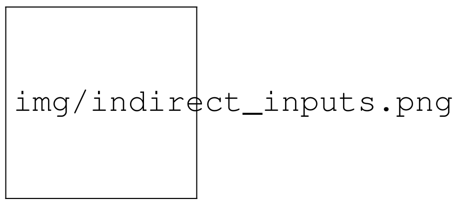


Figure 2.3: Indirect outputs from a SUT[5]

Indirect inputs and outputs

Test Doubles have the benefit of being able to provide *indirect inputs* to the SUT, and to receive *indirect outputs* from the SUT.

When exercising the SUT, we often need a way to control the execution path of the SUT from the test case. In many cases, this is not directly possible, because the SUT bases its execution path from inputs received from one of its DOCs. These inputs are known as *indirect inputs*. In order to provide indirect inputs to the SUT via the DOC, we can replace the DOC with a Test Double which support indirect inputs (see table 2.1).

In order to verify that the SUT performed as expected during the test, we may need to record the SUT's interactions with it's various DOCs. These interactions are called *indirect outputs* because they are not directly visible from the test case itself. If a DOC does not provide a way for the test case to read these indirect outputs in a way that makes it possible to verify the correct behaviour of the SUT, then a Test Double which supports indirect outputs may be used in place of the DOC (see table 2.1).

Common pitfalls of the Test Double patterns

Using the Test Double patterns can make it easier to write unit tests, but misuse of the patterns can cause problems. Because we are replacing a DOC with a component that is custom-built for a particular test case, we must be careful to avoid overly tight coupling between the Test Double and the SUT. This can lead to *Overspecified Tests*, which are a type of the *Fragile Test* anti-pattern [19]. A Fragile Test is one which may break due to changes in the SUT which do not affect the behaviour being tested.

Another potential pitfall of misusing Test Doubles is the possibility that the developer may accidentally replace the part of the SUT which is being tested. This can be a severe problem, because tests for a particular behaviour will show as passing, even though that behaviour has never been exercised by the tests. Care should be taken when writing tests to ensure that the test actually exercises the behaviour under test.

[4]Adapted from Meszaros [19]
[5]Adapted from Meszaros [19]

A note on terminology

The xUnit Test Patterns book refers to the pattern as a *Mock Object*. While OCaml does have object oriented features, we are primarily concerned with mocking OCaml modules. In the rest of this dissertation, we will use the term Mock Object when referring specifically to the pattern described in the xUnit book, although we mean to apply this pattern to OCaml's modules. When speaking of a module that follows the Mock Object pattern, we may specifically refer to this as a *Mock Module*.

2.2.1 Fake Object pattern

Fake Object is a simpler version of the DOC which performs the same function with less complexity.

A Fake Object is a simple replacement for a complex DOC. It does not provide any test-specific features, such as the ability to deal with indirect inputs or outputs. A Fake Object should be used when a SUT has a complex DOC that makes it infeasible to run unit tests. For instance, a web service DOC may be replaced by an object which returns hard-coded results without the need to access the network. A database may be replaced by a simpler data store such as a hash table or other in-memory database.

2.2.2 Dummy Object pattern

Dummy Object is a special case of Test Stub which applies to value types.

A *Dummy Object* is a special case of a Stub Object. It is used specifically when the SUT does not interact with the DOC at all during the test, but a value for the DOC is still required in order to initialise the SUT. For instance, the SUT may be a method of a class which requires certain values to be constructed, but the SUT itself doesn't make use of those values during the test. In a language with nullable types, such as Java, a Dummy Object may simply be the `null` value. In a language like OCaml, a Dummy Object may be a value, such as a simple type constructor. A more complex type may have a "default" or "empty" value that can be used. This is often the case with record types, for which programmers often create an "empty" value which they can extend to create more complex values. Many standard library containers have either an empty default value, or a simple constructor, for instance `Set.empty` or the value constructor `Buffer.create`, which takes a single integer value.

2.2.3 Test Stub pattern

Test Stub replaces a DOC in order to allow the test case to provide *indirect inputs* to the SUT.

The Test Stub pattern is the first of the Test Double patterns which provides functionality specifically for the test case it is used for. A Test Stub is an implementation of the DOC's interface which can be programmed to provide indirect inputs to the SUT during the test. This mechanism is used by the test case to provide indirect inputs to the SUT via the DOC.

To use the Test Stub, the test case will configure a new Test Stub, either at run-time by using stub generator from a testing framework, or by creating a hard-coded Test Stub. The test stub is then inserted into the SUT using some form of dependency injection. When the test is run, the SUT receives indirect inputs from the test case via the Test Stub DOC.

2.2.4 Test Spy Object pattern

Test Spy replaces a DOC in order to record the SUT's *indirect outputs* and provide them to the test case.

The Test Spy is used to to replace the DOC in the SUT and record the interactions that the SUT makes with the DOC. It is analogous to the the Test Stub, but it works in the opposite way: instead of providing indirect inputs from the test case to the SUT via the DOC, it provides indirect outputs from the SUT to the test case via the DOC.

The Test Spy is created in the same fashion as the Test Stub, and inserted into the SUT in the same way. After the SUT has been exercised, the test case may retrieve the SUT's indirect outputs from the Test Spy and proceed with the evaluation of the test.

2.2.5 Mock Object pattern

Mock Object combines the features of the Test Stub and Test Spy to provide both *indirect inputs* to the SUT and record the SUT's *indirect outputs*, as well as perform *behaviour verification*.

A Mock Object combines the features of a Test Stub and a Test Spy. It is typically configurable at run time – Mock Objects are usually not hard-coded, at least in the languages for which there are Mock Object frameworks. Mock Objects are capable of both providing indirect inputs to the SUT, as well as recording indirect outputs from the SUT.

The single distinguishing feature of the Mock Object, which makes it different than just a Test Double which behaves as both a Stub and a Spy, is that it is capable of performing *behaviour verification*. While a Test Spy simply performs the task of recording the indirect outputs of the SUT for later reading by the test case, a Mock Object provides facilities for verifying the correctness of the indirect outputs from the SUT. A Mock Object may be either *strict* or *lenient*. A strict Mock will raise a test failure if it receives the wrong outputs (too few, too many, or an unexpected output), as well as out-of-order outputs. A lenient Mock may allow the test to pass if it receives at least the outputs it was expecting, and may also allow out-of-order outputs. This behaviour would be configurable by the the mocking framework.

Listing 2.1: Example of JMock, a Java Mocking framework

```java
import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnitRuleMockery;

import org.junit.Rule;
import org.junit.Test;
import org.junit.Ignore;

import static org.junit.Assert.assertEquals;

// JMock generates mocks from interfaces, not classes
interface IOracle {
  int getAnswer();
  int setAnswer(int i);
}

class SUT {
  private final IOracle oracle;

  //constructor (receives DOC)
  public SUT(IOracle o) {
    oracle = o;
  }

  //sut_method (uses DOC)
  public int askTheQuestion() {
    return oracle.getAnswer();
  }

  public int setAnswer(int i) {
    return oracle.setAnswer(i);
  }
}

public class JMockTest
{
  @Rule public final JUnitRuleMockery context = new JUnitRuleMockery();
  final IOracle oracle = context.mock(IOracle.class);

  @Test
  public void satisfiesExpectations_OracleUsedOnce () {
    //final IOracle oracle = context.mock(IOracle.class);

    // Set up test and install the mock
    SUT sut = new SUT(oracle);

    // Specify the expectations
    context.checking(new Expectations() {{
      allowing(oracle).getAnswer();
      allowing(oracle).getAnswer();
      allowing(oracle).setAnswer(43);
    }});

    // Exercise the SUT
    sut.askTheQuestion();
    sut.askTheQuestion();
    sut.setAnswer(43);
  }

  @Test
  @Ignore
  public void doesNotSatisfyExpectations_OracleUsedOnce () {
    SUT sut = new SUT(oracle);

    // Specify the expectations
    context.checking(new Expectations() {{
      oneOf(oracle).getAnswer();
    }});

    // Exercise the SUT
    sut.askTheQuestion();
    //sut.askTheQuestion(); // BUG IN TEST: This will fail verification
  }
}
```

JMock is a mocking framework for the Java language. It is highly configurable, and provides a domain specific language (DSL) for configuring expectations [7]. JMock's expectation language provides many features beyond what is discussed in the xUnit book. Listing 2.1 is a small example of using JMock, a Java mocking framework, in a test case.

```
invocation-count(mock-object).method(argument-constraints);
  inSequence(sequence-name);
  when(state-machine.is(state-name);
  will(action);
  then(state-machine.is(new-state-name));
```

Above is a pseudo-code notation of the expectation format that JMock supports [8]. In addition to invocation counts and method argument matchers and value returners, JMock provides both *sequences* and *state machines* for use in mocks.

A sequence is a way of specifying that a series of invocations listed in a mock's expectations are meant to occur in the specified order (a *strict* mock). Unless an invocation is added to a sequence, the order of execution is not significant (a *lenient* mock). JMock also allows expectations to define simple state machines, so that expectations can keep track of the state that the SUT should be in during the test, and behave accordingly.

2.3 Tools for software testing in functional langauges

While much of the literature about and libraries for writing unit tests are targeted at more-popular object oriented languages, many functional languages also have testing libraries. Here is a short summary of some of these libraries.

2.3.1 xUnit implementations

Unit testing frameworks for many languages follow the xUnit pattern of testing ([15] [23] [28]), and languages like Haskell and OCaml are no different [12] [25]. OCaml's OUnit is one of the more popular unit testing frameworks for the language. It provides assertion functions that can be used to verify that a test case has passed or failed, combinators for grouping test cases into suites of tests, and helpers for enabling or disabling individual test cases.

Listing 2.2: Example of OUnit usage

```ocaml
open OUnit

(* This is our SUT *)
let divider x y = x / y

(* A pair of simplistic test cases *)
let test_div_by_zero () =
  assert_raises
    Division_by_zero
    (fun () -> divider 1 0)

let test_1_div_2_eq_0 () =
  assert_equal
    ~msg:"Test int division"
    (divider 1 2)
    0

(* Group our test cases into a suite *)
let suite =
  "Example suite" >::: [
    "test_div_by_zero"  >:: test_div_by_zero;
    "test_1_div_2_eq_0" >:: test_1_div_2_eq_0
  ]

(* Our main function runs the test suite *)
let _ = run_test_tt_main suite
```

OUnit and HUnit both follow the xUnit Test Patterns guide for unit tests, and therefor have the same feel as unit test libraries for other languages, such as JUnit and NUnit, which are also based on the xUnit patterns. The only major differences between these libraries is that the OCaml and Haskell unit test libraries both use functional combinators for building test cases and suites (the `>::` and `>:::` operators in listing 2.2). Combinators such as these are idiomatic to both of these languages.

Kaputt is another full-featured unit testing library for OCaml [16]. In addition to supporting xUnit-style assertion tests, it also supports QuickCheck-style specification testing.

Listing 2.3: Example of Kaputt usage

```
open Kaputt.Abbreviations

(* This is our SUT *)
let divider x y = x / y

(* A simplistic test case *)
let test_div_by_zero =
  Test.make_simple_test
    ~title:"test division by zero"
    (fun () ->
     Assert.make_raises
       (function Division_by_zero -> true | _ -> false)
       Printexc.to_string
       (fun () -> divider 1 0))

let () = Test.run_tests [ test_div_by_zero ]
```

As you can see from listing 2.3, Kaputt's basic usage is similar to that of OUnit, if not slightly more verbose. Kaputt also supports specification-based testing, in the style of QuickCheck, as shown in listing 2.4.

Listing 2.4: Example of Kaputt's specification-testing features

```
open Kaputt.Abbreviations

(* Precondition *)
let is_int : int Spec.predicate = fun _ -> true

(* Postcondition *)
let one_greater (a,b) = b = a+1

let spec_example =
  Test.make_random_test
    ~title:"Test succ function"
    ~nb_runs:128
    Gen.int
    succ                           (* SUT *)
    [ is_int => one_greater ]    (* pre- implies post-condition *)
```

Kaputt also provides support for creating Mock functions, but this part of the library is not as well developed as its assertion and specification-based testing capabilities. Kaputt provides functions for creating Mock functions which can record the number of times the Mock has been called with a certain input value, and which can output specified responses. It does not provide the ability to automatically generate Mock Modules, nor does it provide capabilities for injecting a Mock into a SUT. While it can record a SUT's indirect outputs, it does not provide the ability to perform behaviour verification; this must be performed manually by the test case. In this respect, Kaputt provides features to aid developers in writing their own hard-coded Mock functions, but little more.

Listing 2.5: Annotated example of Kaputt's mock features[6]

```
open Kaputt.Abbreviations

let mock_example = Test.make_simple_test
    (fun () ->
     (* eq_int_list tests that two lists are equal *)
     let eq_int_list = Assert.make_equal_list (=) string_of_int in
     (* Mock the standard library function 'succ' *)
     let f = Mock.from_function succ in
     (* Test input *)
     let i = [0; 1; 2; 0] in
     (* Exercise the mock *)
     let o = List.map (Mock.func f) i in
     (* Our expected outputs *)
     let o' = [1; 2; 3; 1] in
     (* Actual outputs match expected outputs *)
     eq_int_list o' o;
     (* Actual inputs match the inputs the mock received *)
     eq_int_list i (Mock.calls f);
     (* Mock was called the correct number of times *)
     Assert.equal_int (List.length i) (Mock.total f))
```

In the example in listing 2.5, the call to `List.map` is actually the SUT, and the function `succ` is our DOC, for which we create the Mock `f`. This is a simplistic example meant to demonstrate creating Mock functions using Kaputt, and not to demonstrate proper testing procedure.

2.3.2 Specification-based random testing with QuickCheck

Koen Claessen's and John Hughes' ICFP paper "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs" [6] described a method of *specification testing* that has since been ported to other languages, including OCaml [26] [16].

Specification testing is a method of describing the properties of the SUT as specifications, and using random generators to attempt to disprove the specification. For instance, say we want to test the `reverse` function.[7] This function has the following properties:

```
reverse [x]           = reverse [x]
reverse (xs++ys)      = reverse xs ++ reverse ys
reverse (reverse xs) = xs
```

We can easily specify these properties in Haskell.

```
rev_Unit x =
  reverse [x] == reverse [x]

rev_Append xs ys =
  reverse (xs ++ ys) == reverse xs ++ reverse ys

rev_Identity xs =
  reverse (reverse xs) == xs
```

[6]Un-annotated example adapted from Kaputt's manual: `http://kaputt.x9c.fr/distrib/kaputt.html#mock-example`

[7]This is an example from Claessen's and Hughes' QuickCheck paper [6]

And we can execute these specifications using quickCheck:

```
Prelude Test.QuickCheck> quickCheck rev_Append

+++ OK, passed 100 tests.
```

Specification testing is well suited to functional languages, especially those which restrict or eliminate mutability and side-effects, such as Haskell. It is common to write Haskell code which separates side-effecting functions (those that operate under the IO monad) from *pure* functions which have no side effects. These pure functions can then be tested using QuickCheck.

2.3.3 Criterion: Haskell benchmarking library

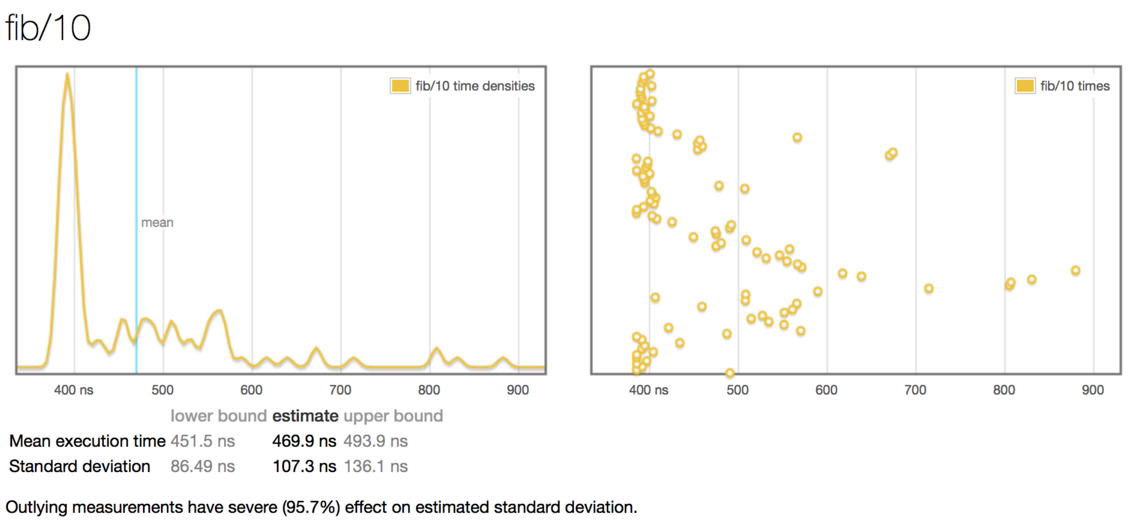
While it isn't a unit testing library like HUnit or OUnit, Criterion [24] is a useful Haskell library for running benchmarks. It is particularly interesting because of the attention to statistical detail that the library provides. Each benchmark run is calibrated against the system clock, and benchmarking runs with a high degree of variance are flagged to the user.

Listing 2.6: Example of Criterion benchmark library

```haskell
module Main where

import Criterion.Main

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main = defaultMain [
  bgroup "fib" [ bench "10" $ whnf fib 10
               , bench "20" $ whnf fib 35
               , bench "25" $ whnf fib 37
               ]
  ]
```

Below we see the command line output shown from running the code in listing 2.6, while figure 2.4 is part of the graphical output from running this program. In this image we see that Criterion gives us both the raw time data (graph on the right), and a kernel density estimate of the time data (graph on the left).

```
warming up
estimating clock resolution...
mean is 1.119713 us (640001 iterations)
found 65667 outliers among 639999 samples (10.3%)
  5404 (0.8%) low severe
  60263 (9.4%) high severe
estimating cost of a clock call...
mean is 58.35944 ns (7 iterations)

benchmarking fib/10
mean: 3.819773 us, lb 3.709738 us, ub 3.968785 us, ci 0.950
std dev: 653.0665 ns, lb 506.8601 ns, ub 834.0847 ns, ci 0.950
found 13 outliers among 100 samples (13.0%)
  6 (6.0%) high mild
```



	lower bound	estimate	upper bound
Mean execution time	451.5 ns	469.9 ns	493.9 ns
Standard deviation	86.49 ns	107.3 ns	136.1 ns

Outlying measurements have severe (95.7%) effect on estimated standard deviation.

Figure 2.4: Output from the benchmarking library Criterion

```
  7 (7.0%) high severe
variance introduced by outliers: 92.538%
variance is severely inflated by outliers
```

3 Application

We will now discuss the application of the Mock Object pattern to the OCaml programming language. In this chapter we will cover dependency injection techniques in OCaml, the manual creation of a mock module and expectations, a domain specific language for specifying expectations in OCaml, and the automatic generation of mock modules from a module interface.

3.1 Dependency injection methods in OCaml

Dependency injection (DI) techniques in object oriented languages typically focus on lifting hard-coded dependencies into a class's constructor so that dependencies can be injected at object creation time. In OCaml things are not much different, except that we will use language features such as functors and first-class modules for injecting module dependencies. Complex functionality typically found in more mature DI frameworks is not necessary for the rest of this work, and we will not describe their implementation here.

Listing 3.1: Example of dependency injection techniques in OCaml

```
(* Examples of depenency injection in OCaml *)

(* We have some dependency DOC *)
module DOC = struct
  let f x = x + 1
end

(* And a SUT which depends upon DOC *)
module SUT = struct
  let g x = DOC.f x
end

(* BUT we want to be able to use a different version of DOC, so we
   have to functorise SUT and specialise it with a new version of the
   DOC. *)

(* Create the module signature of DOC  *)
module type S = module type of DOC

(* SUT_functor now creates a version of the SUT which uses D instead of
    DOC *)
module SUT_functor(D : S) = struct
  let g x = D.f x
end

(* This is equivalent to the original SUT *)
module Original_SUT = SUT_functor(DOC)

(* This is a new SUT, with a different DOC *)
module New_SUT = SUT_functor(struct let f x = x + 42 end)
```

```
(* Downsides to functorisation: We have to do significant refactoring,
especially if the module in quesiton is at the toplevel of a
compilation unit, because ml files cannot be functors. A patter for
"lifting" a toplevel module into a functor is to surround the whole
file with a new module calld "Make": 'module Make (D : DOC) = struct
...', and at the end of the file, after we end the Make module, add
'include Make(DOC)', which will modify this module so that it contains
both a functor to create a new module, and the original contents of
the module. If the ml file is restricted by an mli file, it may be
simpler to just move the functor to a new ml file (say,
"SUT_func.ml"), and have the original ml file contain only the line
'include SUT_func.Make(DOC)'. *)

(* Alternatively, instead of using functors we could use first class
modules. We have a couple different options. We can introduce a first
class module at the toplevel of the SUT. Or we could intruduce a new
module parameter to the function which we wish to test. *)

module SUT_fc = struct

  (* Extract the module signature from the DOC *)
  module type S = module type of DOC

  (* First option: optionally pass a new DOC into the function we want
     to test. The DOC module is an optional argument, so we haven't  *)
  let g ?(m=(module DOC : S)) x =
    let module M = (val m : S) in
    M.f x

  let m = ref (module DOC : S)

  let h x =
    let module M = (val !m) in
    M.f x

end

(* Downsides to this method are that each function which needs to have
the option of dependency injection needs to be refactored to install
the new dependency. This may or may not be less invasive than
functorising the entire module. There is also a very slight
performance cost to this method, but it is so low an overhead that it
should be ignored for most purposes. *)

(* Another option for managing dependency injection using first class
modules would be to create a "module factory." Client modules could
ask this factory to create a particular module, and the module factory
would check whether it should return the production dependency or the
test dependency, depending on perhaps a command-line flag or some
configuration file. A major benefit to this method is that it removes
the need to refactor each of the functions that access the first class
module, because we can set this module at module initialisation
time. *)

module Factory = struct
  (* Always return the "fake" module, for simplicity *)
  module type S = module type of DOC
  let get_module () = (module struct let f x = x + 42 end : S)
end

module SUT_factory = struct
  module DOC = (val Factory.get_module ())
  let g x = DOC.f x
end

let main =
  (* Using the original SUT *)
  Printf.printf "Original:    g 42 = %d\n" (SUT.g 42);
  Printf.printf "Functor:     g 42 = %d\n" (New_SUT.g 42);
  Printf.printf "First-class: g 42 = %d\n" (SUT_fc.g 42);
  Printf.printf "First-class: g ~m:(module struct \
                 let f x = x + 42 end) 42 = %d\n"
                (SUT_fc.g ~m:(module struct let f x = x + 42 end) 42);
  Printf.printf "Factory:     g 42 = %d\n" (SUT_factory.g 42);
  ()
```

3.2 Design considerations for mock modules

Before we can automatically generate a mock module from a given interface, we must first describe how to create manually-written (hard-coded) mock modules.

3.3 The simplest mock module implementation

Note that this method does not provide any features we would expect from a full-featured mocking framework such as JMock. There is no automatic generation of the mock implementation. There is no expectation language, or even a means to specify expectations. And because there is no way to specify expectations, the test case must set up the mocked functions manually, and verify the behaviour of the SUT manually.

While this is a very simplistic method for creating a mock module, we will continually extend this method in the following sections, until we have automatically generated mocks with an expression language.

Listing 3.2: Example of dependency injection techniques in OCaml

```
(* Sig to mock *)
module type TURTLE =
  sig
    type t
    type direction = Left | Right
    type position = int * int
    val make : unit -> t
    val get_position : t -> position
    val turn : direction -> t -> t
    val move_forward : int -> t -> t
    val toggle_pen : t -> t
    val (>>) : t -> (t -> t) -> t
  end

(* The *real* TURTLE implementation *)
module Turtle : TURTLE = struct
  type direction = Left | Right
  type position = int * int
  type orientation = North | South | East | West
  type pen = Up | Down
  type t = {
    pos : position;
    pen : pen;
    dir : orientation;
  }
  let (>>) t f = f t
  let make () = { pos = (0,0); pen = Up; dir = North }
  let get_position t = t.pos
  let turn d t = match d with
    | Left ->
       let d = match t.dir with
               | North -> West | West -> South
               | South -> East | East -> North
       in { t with dir = d }
    | Right ->
       let d = match t.dir with
                | North -> East | West -> North
                | South -> West | East -> South
       in { t with dir = d }
  let move_forward i t =
    let x, y = t.pos in
    match t.dir with
    | North -> { t with pos = (x, y+1) }
    | South -> { t with pos = (x, y-1) }
    | East  -> { t with pos = (x+1, y) }
    | West  -> { t with pos = (x-1, y) }
  let toggle_pen = function
    | { pen = Up } as t -> { t with pen = Down }
    | { pen = Down } as t -> { t with pen = Up }

end

(* A partial mock of TURTLE (only mock's turn function). This works by
   including the to-be-mocked module in the mock itself, and
   reimplementing the functions we want to mock. This won't work if
   initialising the to-be-mocked module will have unwanted side
   effects. *)
module PartialTurtleMock = struct
  include Turtle

  (* We have to do this for every function we want to mock. *)
  let turn_args = ref []
  let turn_return : (direction -> t -> t) option ref = ref None
  let turn_count () = List.length !turn_args

  let turn t d =
    turn_args := (t,d) :: !turn_args;
    match !turn_return with
    | None -> turn t d
    | Some f -> f t d

end

(* Our system under test *)
module Draw(T : TURTLE) = struct
```

```
  (* Draw a simple shape *)
  let draw () =
    let open T in
    make ()
    >> turn Left
    >> toggle_pen
    >> move_forward 1
    >> turn Right
    >> move_forward 2
    >> toggle_pen
    |> ignore

end

let string_of_direction = function
  | Turtle.Left -> "Left"
  | Turtle.Right -> "Right"

(* A test case which uses the mock module *)
let test =
  (* Set up the test *)
  let module D = Draw(PartialTurtleMock) in
  let open PartialTurtleMock in

  (* Can provide indirect inputs to SUT *)
  turn_return := Some (fun d t ->
                       Printf.printf "Turning %s\n" (string_of_direction
                           d);
                       Turtle.turn d t);

  (* Run the test *)
  print_endline "* Excercising SUT";
  D.draw ();

  (* Analyse the indirect outputs from the SUT *)
  print_endline "* Analysing outputs";
  Printf.printf "Turn was called %d times\n" (turn_count ());
  Printf.printf "Last direction was: %s\n"
                (!turn_args |> List.hd |> fst |> string_of_direction)
```

3.4 Specifying a mock's expectations

Much of the Mock Pattern's power comes from having a lightweight expectation specification DSL. In this section we describe the design and implementation of an expectation DSL in OCaml which is similar to the JMock expectation DSL described in section 2.2.5.

Some considerations when implementing expectations. Expectations are used for two purposes: 1) to allow the mock implmentation to provide the appropriate indirect inputs to the SUT (through the expectation's actions), and 2) to provide indirect outputs to the test case from the SUT, in the form of invocation counts, as well as values passed to the mocked functions.

For expectations, we need to keep track of the expected invocation count, and the expected outputs of the function, whether they are values or expections. When (and if) we implement individual sequences, we need to keep track of which expectations are in which sequence.

While running the test case, the mock implementation needs to keep track of the invocations of each function, including the order in which they were called, as well as with what arguments they were called.

Note: a feature that we'd like to have is expectation pattern matching. We'd like to be able to set an expectation that a function will be called with some pattern-restricted set of arguments, and then either verify that the the function was invoked with arguments that match the pattern, or simply inspect the arguements passed to the function in the test itself. We might want to implement this feature by allowing the user to pass arbitrary funtions which match the type of the mocked function to the mock as the mock function itself, instead of generating an entire function using just the invocation arguments and expected return value.

3.4.1 Module organisation

We will be generating a lot of code when we generate mock modules. We'll have to generate an implementation of the to-be-mocked interface. We'll have to generate expectation language code which uses the same types as defined in the to-be-mocked interface, such that the types in the expectation langauge will unify with those in the mocked module. We'll have to generate code to verify the expectations that have been defined in the test case.

The organisation of generated modules must be chosen carfully, especially to satisfy the requirement that types between the expectation langauge and the mocked module must unify.

One solution to this is to put logically separate code into separate modules, and strategically include modules in "derivative" modules where necessary. (This is what we have currently done.) Doing this allows us to properly namespace the identifiers that we have generated, so that we don't have to mangle generated names and then redefine them later.

Another solution is to place all generated code into a single module that is not exported, but to then create separate modules (say `E` for the expecation language and `M` for the mocked module), where each of these modules is restriced by an appropriate signature. This should give us the most flexibility: no cumbersome module inclusions, while hopefully retaining types that are unifiable. Unfortunately, with this solution we will have to mangle our generated names and then redefine them later, because we may clash with the names in our to-be-mocked interface. We can minimise the names we have to mangle by keeping track of the identifiers defined in the to-be-mocked module, and only mangling clashing names. On second thought, this sounds far too complex; more complex than managing multiple modules.

3.4.2 Generalised Algebraic Data Types

We make use of Generalised Algebraic Data Types (GADTs) to help us encode the arguements for and return values of our expectations. GADTs are a relatively new feature in OCaml. They allow ...

3.4.3 Verifying expectations

To verify expectations, we need to compare them to the invocation record, and ensure that the invocations satisfy the expecatations. We need to ensure:

1. All invocation counts are satisfied. This includes insuring that *never* functions are not called, and that number-checked functions are called the correct number of times. We can skip testing for *allowing* functions, since we don't care whether they are called.

2. Sequenced expectations are invoked in the order specified. (We'll do this later, not as a first feature.)

To verify expectations, we need to iterate over each invocation, and imperatively update the list of expectations accordingly. We search for the current invocation in the list of expectations, and increment the invocation count. After we've interated through the invocations, we have an updated list of expectations that we can then verify (short-cut: we update the list of expectations at test-time, instead of recording invocations). Now we proceed through the expectations and verify that the invocation count satisfies the expectation. Later, when we implement sequences, we can also verify that the order of invocations is correct.[1]

We should be sure to proceed through the entire list of expectations, and not necessarily short-circuit at the first matching expectation. If that first matching expectation has already been satisfied, then we will need to proceed to the next potentially matching expectation. This means that we will need to perform verification at each invocation step, so that we can easily skip already-satisifed expectations.

3.5 Automatically generating mock modules

Mocks are only truly useful as a testing tool when they can be automatically generated from an interface. We will use Camlp4, an OCaml preprocessing tool, a new OCaml feature called *extension points* used for AST annotation, and the OCaml compiler's *compiler-libs* in order to generate and pretty-print OCaml code.

[1]Aha! We can simply attach an invocation sequence number to each expectation as we invock a matching function. This will allow us to verify that the expectations in each sequence have a monotonically increasing invocation sequence number.

4 Reflection

4.1 Comparison of mocking frameworks

Compare MoCaml, manual mocking with Kaputt, and JMock. Demonstrate pros and cons of each framework.

4.1.1 Notes

1. JMock can't tell at run-time whether an expecation type-checks. Make sure that we can do this!

2. `oneOf(mock).foo(); oneOf(mock).foo()` means **two** calls to `foo()`, not just one. (Note: this is okay because they occur in sequence. Two allowings with this behaviour would be a problem.)

3. JMock doesn't allow one to pass in methods or closures to implement mock functions, but we can (and should!) allow this.

Bibliography

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

[2] Barry Boehm and Victor R Basili. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, 426, 2005.

[3] Barry W Boehm. Understanding and controlling software costs. *Journal of Parametrics*, 8(1):32–68, 1988.

[4] L Bossavit. *The Leprechauns of Software Engineering*. Leanpub, Vancouver, BC, 2012.

[5] Camlp4. `http://pauillac.inria.fr/camlp4/`, March 2014.

[6] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[7] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 855–865. ACM, 2006.

[8] Steve Freeman and Nat Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[9] Garbage Collection - OCaml. `http://ocaml.org/learn/tutorials/garbage_collection.html`, March 2014.

[10] A gentle introduction to haskell: Modules. `http://www.haskell.org/tutorial/modules.html`, March 2014.

[11] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[12] HUnit – Haskell unit testing. `http://hunit.sourceforge.net/`, January 2013.

[13] Ieee standard glossary of software engineering terminology, Dec 1990.

[14] JMock: an expressive mock object library for Java. `http://jmock.org/`, January 2013.

[15] JUnit. `http://www.junit.org/`, January 2013.

[16] Kaputt. `http://kaputt.x9c.fr/`, January 2013.

[17] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rmy, and Jrme Vouillon. *The OCaml system release 4.01*. INRIA, 2013.

[18] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[19] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[20] Yaron Minsky, Anil Madhavaeddy, and Jason Hickey. *Real World OCaml*. O'Reilly, 2014.

[21] Modules — Python v2.7.6 documentation. `http://docs.python.org/2/tutorial/modules.html`, March 2014.

[22] Moq. `http://www.moqthis.com/`, March 2014.

[23] NUnit. `http://www.nunit.org/`, January 2013.

[24] Bryan O'Sullivan. Criterion, a new benchmarking library for Haskell. `http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/`, September 2009.

[25] OUnit. `http://ounit.forge.ocamlcore.org/`, January 2013.

[26] Ocaml-quickcheck – a mechanical translation of quickcheck to ocaml. `https://github.com/alanfalloon/ocaml-quickcheck`, March 2014.

[27] Rspec.info: home. `http://rspec.info/`, March 2014.

[28] test-unit - a Unit Testing Framework for Ruby. `http://test-unit.rubyforge.org/`, March 2014.