Injection testing backed refactoring

Thomas Mejstrik
University of Vienna
Vienna, Austria
thomas.mejstrik@gmx.at

Clara Hollomey
Austrian Academy of Sciences
Vienna, Austria
clara.hollomey@oeaw.ac.at

ABSTRACT

Injection-based testing for refactoring is a pattern that minimizes the need for manual editing when altering the internal behaviour of a code base which does not have unit tests yet in place. Neither does it rely on a compilation or a linking process nor does it make assumptions on the structure of the code. Thus, it can be particularly useful for refactoring code that has been written in scripting languages, and specifically targets the research and engineering context. We describe the pattern and propose a set of functions for its application. The applicability of code injection for refactoring is highlighted via specific examples for deriving unit and integration tests. Finally, we comment on customizing the pattern and give practical advice for its implementation.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**; **Software testing and debugging**; *Language features.*

KEYWORDS

refactoring, code instrumentation, legacy code, unit testing, scientific software, Matlab

ACM Reference Format:
Thomas Mejstrik and Clara Hollomey. 2018. Injection testing backed refactoring. In *25th European Conference on Pattern Languages of Programs (EuroPLop '22), July 6–10, 2022, Irsee, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3551902.3551966

1 INTRODUCTION

We introduce the pattern *Injection Testing* to facilitate the *refactoring* of existing code, i. e. changing the internal structure of the code without affecting its external behaviour [1]. The injection testing pattern enlarges the set of code bases to which testing provisions can be added retrospectively.

The pattern allows inferring the internal behaviour of code by defining designated interception points as comments in the source code. No strong requirements are placed on the programming style of the code, which is especially important for refactoring code does not posses testing provisions, in the following called *legacy code.*

Techniques for refactoring are well established, see e. g. [2–9], and often rely on the existence of an accompanying test suite [10].

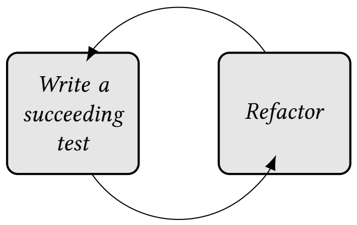

Figure 1: Refactoring cycles: (1) In refactoring, one often has to start with the *Refactor* step. (2) Using the injection testing pattern one can start with the *Write a succeeding test* step, and thus, make the refactoring step safe.
The refactoring step not only concerns the production code, but also the accompanying test suite.

The idea is to start from a succeeding test suite, apply the desired changes, and confirm that the observable behaviour did not change by running the test suite again, as depicted in Figure 1.

For code without testing provisions, however, this classical refactoring cycle indicates a chicken-and-egg problem: It is unclear how to write a test suite prior to touching the code when no such test suite is already present. Consequently, in the absence of tests there is always the risk of unwittingly changing the behaviour of the code, either by changing interfaces functions, or introducing new bugs, or – equally bad – by fixing existing bugs on which the code may rely. Thus, the challenge is intercepting a code‘s internal behaviour while at the same time minimizing the amount of manual editing.

1.1 Existing refactoring solutions

Common interception points arise from the compiler toolchain and have been summarized under the term "seams", *"…a place where you can alter behaviour in your program without editing in that place."* [10, Chapter 4].

Pre-processing seams can be accessed via a text replacement engine that automatically changes the source code just before interpretation or compilation. Besides the need for maintaining such an engine in languages that do not feature a pre-processor, the usage of such seams necessitates a new compilation/interpretation cycle for every change that is applied to the code.

Link seams can be used in languages where the compiler/interpreter produces an intermediate representation of the code. Intercepting calls to other intermediate representations allows for the execution of arbitrary code. Similarly to the pre-processor approach, an additional linking step is required after each refactoring iteration, impeding the practical applicability of the approach.

Object seams are not compilation-related and use overloading and polymorphism for intercepting calls to class methods. This requires

Journal[†]	Matlab submissions	unit tests	automatic unit tests
JOSS	25	9 (36%)	3 (12%)
ACM TOMS	19	9 (47%)	2 (11%)
JORS	22	8 (36%)	2 (9%)
J. Stat. Softw.	18	8 (44%)	1 (6%)
Softw. X	78	11 (14%)	5 (6%)
Softw. Imp.	13	2 (16%)	0 (0%)

Table 1: Number of publications of free Matlab software whose source code is still available, in selected scientific journals between 2015 and 2021, contrasted with the number of submissions additionally including unit tested code and automatic unit testing provisions.

[†] *JOSS*: Journal of Open Source Software, *ACM TOMS*: ACM Transactions on Mathematical Software, *JORS*: Journal of Open Research Software, *J. Stat. Softw.*: Journal of Statistical Software, *Softw. X*: Software X, *Softw. Imp.*: Software Impacts.

that both, the programming language, and the programming style of the code at hand, support object orientation and dependency injection at least to some degree.

All of the above approaches pose requirements on either the programming language or the programming style in which the code has been written. Not every code meets those requirements.

1.2 Code injection

Code injection and code instrumentation refers to changing the behaviour of a program at runtime by introducing (or injecting) new code into an existing computer program to alter the execution path. It is often used for debugging, to add logging capabilities, or to maliciously overtake other‘s computers.

1.3 Our motivation

Code for numerical computations in engineering, scientific, and educational contexts are often written incrementally and by frequently changing authors. This can lead to large code bases with insufficient testing provisions [11].

In fact, an informal survey of software papers published between 2015 and 2021 for Matlab, one of the most common programming languages in academia [12], indicates that only a small fraction of submissions includes tests, let alone automated tests using some specialized testing framework, as indicated in Table 1. This makes it hard to verify the correctness of the published results.

Also, such code is often written in scripting languages for which the seams in Section 1.1 are not applicable due to the absence of a pre-processor, a compilation and a linking step, and thus, makes it hard to refactor the software into a testable version.

1.4 Overview

In Section 2, we present our injection testing pattern and propose a set of functions for its efficient implementation

We show how the pattern can be applied to write *unit tests* in Section 3.1. The code injection pattern is further exemplified in Section 3.2, where we discuss how the injection testing pattern can be used to write *integration tests.*[1]

Finally, in Sections 2.6 and 4 we discuss some points regarding the implementation of our pattern, including the presentation of the Matlab/Octave[2] unit test framework *TTEST.*

2 INJECTION TESTING PATTERN

2.1 Context

Given a function which needs to be refactored, in the following also referred to as *system under test*, that has no or insufficient testing provisions. Furthermore, this function is written in a style rendering it “un-testable”.

2.2 Problem

Refactoring without having tests in place bears the risk of breaking the code. Thus, the challenge is to add tests to existing code while at the same time limiting the amount of behavioural changes to the code.

2.3 Forces

- In languages where there are no predetermined points for intercepting the behaviour of code, in the absence of sufficient testing provisions, one has to apply manual modifications to the code in order to make it testable. The manual editing of code with no sufficient testing provisions in place, however, bears the risk of unwittingly altering its behaviour. Therefore, the amount of manual edits to the code necessary for writing tests should be minimized and their potential impact should be mitigated.
- At the same time, it is desirable to be able inject at arbitrary positions in the code and to not be reliant on any particular programming style or other prerequisites for doing so.
- Also, tests should be loosely coupled to the system under test. In particular, unit tests should only use the SUT‘s public interface and there should be no test code in the production code.
- Running tests should need minimal manual intervention, this means in particular they are preferably written in the same language, not need special commands for executing them, and be portable; and thus not rely on third party code.

2.4 Solution

Add tests by injecting[3] test code at runtime, defining specific entry and exit points, and thus minimize the amount of code changes; optimally by using features directly incorporated in the used programming language, so that no external tools are necessary. We will call the corresponding interception method the *injection seam.*

To mitigate the risk of the injected code altering the behaviour of the code base in an unintentional way, we propose the following “I-triple-A” pattern:

Inject arrange - Act - Assert,

optionally supplemented by an *Inject setup* stage at the very beginning and a *Tear down* stage at the very end.

[1] *Unit tests* test well-defined, self contained parts of code, whereas *integration tests* test the code together with some of its dependencies with respect to specified functional requirements.

[2] *Octave* is a free implementation of the Matlab language.

[3] We use the term *code injection* instead of *code instrumentation*, to stress the point that we do not just add logging or similar functionality, but arbitrary code to an existing code base.

The injection testing pattern closely follows that of a classical unit test, the *AAA* (Arrange - Act - Assert) pattern [13, Chapter 3]. In the *Arrange* section, objects are initialized and data is passed to the system under test. In the *Act* section, the system under test is invoked with the arranged data. In the *Assert* section, the actions and results of the system under test is compared with the expected actions and results.

One key difference to the classical in unit tests is that the *Inject arrange* section does not arrange any objects before the system under test is invoked. The *Inject arrange* section only arranges the code which shall be executed when the system under test is running.

It is important that the *Assert* section is not injected into the system under test. Although possible, this would couple the unit tests and the system under test more strongly and increase the amount of injected code.

If the same part of the code shall be tested multiple times, then the recurring parts of the *Inject Arrange* stage may be factored out to the *Inject Setup* stage. Finally, if the unit test framework does not automatically clear injected code after the tests are finished, then all code injections need to be manually cleared in the *Tear down* stage.

2.5 Consequences

The refactoring of legacy code becomes easier and less likely to break existing code.

2.5.1 *Benefits.*

+ For most programming languages, injection testing does not pose any requirements on the coding style.
  Thus, the pattern can be applicable in situations where other approaches fail, most prominently those described in Section 1.1. This is especially important when refactoring legacy code.
+ Places where interception points can be placed using the injection seam depend on the programming language, but are in general quite arbitrary; usually at each code line. This enables the testing of parts of the code.
+ Using the injection seam can be faster than the pre-processor or link seam. No additional compilation/linking cycles are needed, but also code sections not relevant to the test can simply be skipped.

2.5.2 *Drawbacks.*

– Injection tests are more strongly coupled to the system under test than classical unit tests. In particular, variable names, line numbers, and/or labels may have to be spelled out explicitly in the injection test. Data that previously were private to functions (like local variables) become public to the injection test; In other terminology, the injection test becomes a friend of the system under test.
– Due to the strong coupling of the injection tests with the system under test, the injection tests themselves should be refactored after the SUT is refactored successfully.
– Without sufficient support from the programming language or the development environment, injection testing may not be possible or only partly possible, e. g. whenever a correspondence the set of key functions cannot be implemented fully.

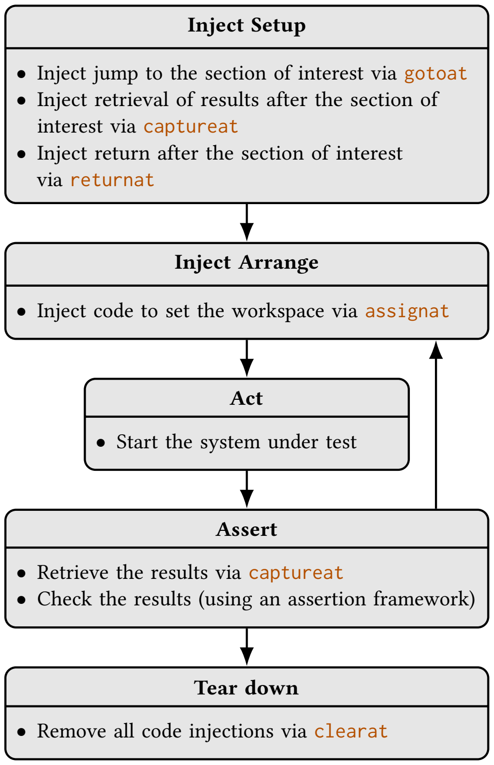
Inject Setup
• Inject jump to the section of interest via gotoat
• Inject retrieval of results after the section of interest via captureat
• Inject return after the section of interest via returnat
Inject Arrange
• Inject code to set the workspace via assignat
Act
• Start the system under test
Assert
• Retrieve the results via captureat
• Check the results (using an assertion framework)
Tear down
• Remove all code injections via clearat

Figure 2: Injection testing pattern: After an optional setup, in the *Inject Arrange* stage, the code to set the desired workspace is injected. After the execution of the system under test in the *Act* stage, the results are retrieved and checked in the `Assert` stage. The final tear down concludes the pattern. The used functions are described in Section 2.6.

– Code injection may degrade the system under test's performance, as (*a*) the code must be supervised by some means to allow code injection, and (*b*) the code can not be compiled with all possible optimizations.[4]
– Minimal manual editing of the code is still required, usually solely by adding comments, and thus, the risk of unwittingly altering its behaviour is not fully mitigated.

2.6 Implementation example

We will make use of the following set of key functions:

- `gotoat`: Jumps to an arbitrary line of code
- `assignat`: Assigns values to variables
- `captureat`: Stores the current state[5]
- `returnat`: Returns from the function
- `clearat`: Removes all injected code

The correspondence between the key functions and the stages of the injection testing pattern are depicted in Figure 2.

[4]Even in scripting languages, code is often just-in-time compiled to make it run faster.
[5]With *state* we mean any observable and for the SUT meaningful entity. For the rest of the paper we will only be concerned with variables and their values, although this pattern does not carry such a restriction.

The only necessary modification to the system under test may be the addition of comments to the code, indicating where to instrument the code for the injection tests. Comments are unlikely to break or alter the existing code[6], particularly when some special style (e. g. `<!TEST1!>`) is used, such that changes can be easily tracked and the comment‘s special role is clearly indicated. If comments were accidentally removed, the accompanying test suite would fail, thus indicating the problem.

Similar approaches are in use for tracing function calls in *Matlab* [15], and for implementing Mutation testing in *Java* [16].

Only some programming languages, e. g. *Java*, support code injection directly. When there is no language support, it is often possible to use the debugger for implementing code injection. The relevant features of the debugger are:

- Running a program step by step
- (Conditionally) stopping the program at a certain point
- Inspecting the current state
- Jumping to a certain point

Often, these features can be accessed from within the programming language, e. g. in interpreted languages like *Matlab*, *Python*, or *R*. Also often the debugger can be controlled using third party libraries, e. g. *libgdb* for *gdb* (discontinued 1993) [17], *lldb* for the *LLVM* toolchain [18], Windows Debugger (*WinDbg*) for *Windows* [19].

Failing that, one can usually still write macros in some scripting language for the debugger, e. g. *GDB\MI* for *gdb* [20].

3 EXAMPLES

In this section we give two examples on using our injection testing pattern for deriving a unit test and an integration test. Apart from the functions proposed in Section 2.6, we use the following tools:

- A method to easily save snapshots of some state
- A unit test framework for automated unit tests
- An assertion framework to easily write assertions

3.1 Example 1: Unit test

We demonstrate the injection testing pattern on the dummy legacy function `foo` given in Listing 1,[7] using Matlab style code. The task is to refactor parts of the body of `foo` out into a sub function while touching the code as little as possible. We only present the relevant lines of code.

Write a succeeding unit test. The function `foo` is not suited for classical unit tests, since the lines of code to be tested are not accessible via standard means.

```
function foo( a1, a2 )
  % lots of code
  sum = 0
  for i = 1:a1  % loop over i = 1 to a1 (both included)
    sum = sum + i; end
  % lots of code
```

Listing 1: Function `foo`

[6]An example where comments could break code are old *BASIC* dialects which used hard coded line numbers to specify the control flow.

[7]The “`:`”-operator generates a regularly spaced vector of numbers, and is usually used to define the bounds of a `for` loop.

In the first step we add the labels `<FOO:1>` and `<FOO:2>` which indicate where to instrument the code with injection tests, see Listing 2.

```
function foo( a1, a2 )
  % lots of code
  % <FOO:1>  % comment for testing
  sum = 0
  for i = 1:a1
    sum = sum + i; end
  % <FOO:2>  % comment for testing
  % lots of code
```

Listing 2: Function `foo` augmented

In this form, the code is ready for injection testing and we can write a unit test suite. An example unit test suite is given in Listing 3.

```
%% inject setup
  gotoat(    'foo', 'goto','<FOO:1>' )
  captureat( 'foo', 'at','<FOO:2>' )
  returnat(  'foo', 'at','<FOO:2>' )

%% test 15
  % inject arrange
  assignat( 'foo', 'at','<FOO:1>', 'a1',15 )
  % act
  foo()
  % assert
  X = captureat()  % obtain values
  EXPECT_EQ( X.sum, 120 )  % compare result

%% test 0
  % inject arrange
  assignat( 'foo', 'at','<FOO:1>', 'a1',0 )
  % act
  foo()
  % assert
  X = captureat()
  EXPECT_EQ( X.FOO2, 0 )

%% tear down
  clearat( 'foo' )
```

Listing 3: Unit test suite for `foo`

What happens in Listing 3. In the `% setup` part, we collect the code which is shared among both unit tests, `%% test 1` and `%% test 2`. The function `gotoat` injects code such that, after entering `foo` the control flow immediately continues at the line with the comment `<FOO:1>`. The function `captureat` injects code such that the value of the variable `sum` at the line with comment `<FOO:2>` is stored for later retrieval. The function `returnat` injects code so that the function `foo` returns to the caller site whenever the control flow reaches line `<FOO:2>`.

In the `%% test 1` part we arrange the data to be injected into the function foo; at the line with comment `<FOO:1>` the variable `a1` will be assigned the value 15. Afterwards we execute the system under test by calling it and retrieve the stored data by `captureat`. In the `%% assert` section we check whether the retrieved value of `sum` equals our expected value `120`. The second test `%% test 2` follows the same pattern.

In the `%% tear down` part, after the injection tests have finished, we clean up using `clearat` which removes all injected code from the function `foo`.

Refactor. Having our unit tests in place we can safely refactor the function `foo`, as given in Listing 4.

```
function foo( a1, a2 )
  % lots of code
  % <FOO:1>
  sum = sum0( a1 )
  % <FOO:1>
  % lots of code

function x = sum0( x )
  x = x * ( x + 1 ) / 2
```

Listing 4: Refactored `foo`

When we are sure that the refactoring did not change the behaviour, one should refactor the code and the unit tests once more, see Listings 5 and 6.

```
function foo( a1, a2 )
  % lots of code
  sum = sum0( a1 )
  % lots of code

function x = sum0( x )
  x = x * ( x + 1 ) / 2
```

Listing 5: Twice refactored `foo`

```
%% test foo
  EXPECT_EQ( sum0(15), 120 )
  EXPECT_EQ( sum0(0), 0 )
```

Listing 6: Refactored unit tests for `foo`

3.2 Example 2: Integration test

Another example for using the injection test pattern is the gradual refactoring of the function `bar`, whose functionality is not apparent to the programmer, see Listing 7.

Write a succeeding unit test.

```
function bar( a )
  % 1000 lines of code
  % <BAR:0>
  % another 1000 lines of code
  % <BAR:1>
```

Listing 7: Function `bar`

Using injection testing, we store the full state of the program at various locations when run in its initial form, i.e. before refactoring. After refactoring we compare the saved state with the new state. If they coincide, we can assume that the behaviour of the system under test did not change. An exemplary unit test suite is given in Listing 8.

What happens in Listing 8. The only substantial difference to our unit test suite for `foo` is the use of the helper function `CACHE`. This is just a thin convenience wrapper for storing data to disk, used as follows: When a file with name equal to its first argument does not exist, it stores the value of the second argument to disk. Otherwise, it discards the second argument and loads the stored data from disk.

```
%% inject setup
  captureat( 'bar', 'at','<BAR:0>' )
  captureat( 'bar', 'at','<BAR:1>' )

%% test 5
  % arrange
  a = 5
  % act
  bar( a )
  X = captureat()  % obtain values
  % assert
  EXPECT_EQ( CACHE('BAR0_5', X.BAR0), X.BAR0 )
  EXPECT_EQ( CACHE('BAR1_5', X.BAR1), X.BAR1 )

%% test 7
  a = 7
  % test continuos as above

%% tear down
  clearat( 'bar' )
```

Listing 8: Integration tests for `bar`

This time we inject code such that the whole workspace of the function `bar` is captured whenever the control flow reaches the lines with comments `<BAR:0>` and `<BAR:1>`. The call `X = captureat()` in the `% act` section then retrieves the stored data and stores it in `X`. The assert section now compares the two stored states with the snapshot taken from before refactoring.

4 IMPLEMENTATION IN MATLAB

We implemented the set of key functions proposed in Section 2.6 using Matlab's debugger and conditional breakpoints. Matlab's conditional breakpoints evaluate a string at run-time. If the result is *truthy*[8], the code run is stopped at that location; but when it is *falsy*, the code run continues normally. Using conditional breakpoints for code injection in Matlab has some restrictions:

(1) Conditional breakpoints only accept valid Matlab commands, but not anonymous functions.
(2) The injected code must return a *falsy* value in order to avoid the debugger stopping its execution.
(3) If the injected code throws an error, it is caught automatically by Matlab and the program run stops, i.e. the debugger starts.
(4) Injected code is always executed before the code at the injected line. Code cannot be injected between statements.

To execute anonymous functions, we store them in a persistent variable in some function, and generate a string which then executes that anonymous function. To ensure that the return value of the injected code is `false`, it gets wrapped in a function returning `false` and evaluated by `evalin( 'caller', __ )`[9]. Errors thrown by the injected code are caught inside the function which evaluates the string or anonymous function.

4.1 Example implementation of `evalat`

To illustrate the execution of anonymous functions in Matlab, and thus execution of arbitrary code, a minimum implementation of a

[8]A *truthy* value is a value which implicitly evaluates to `true`, for example in an `if` condition; e. g. `true`, 1 or an array with only non-zero values. Contrary, a *falsy* value implicitly evaluates to `false`; e. g. `false`, `0`, or an array with at least one zero.
[9]The function `evalin( 'caller', cmd )` executes a command `cmd` in the callers workspace, and in particular has access to the callers workspace. Note that `evalin` cannot be used recursively.

function `evalat` is given in Listing 9. The key functions `captureat`, `assignat` both can be derived from this one.

Note, to avoid parsing the inputs, the interface of `evalat` is different from the interface of the `...at` functions in the listings above.

```
function ret = evalat( fun, lne, h );
  persistent cache;
  if( nargin==0 );
    ret = cache;
    return; end;
  cache = h;
  h = ['returnfalse( 'assign(''handle'',evalat()) ) ||'...
      'returnfalse( handle() ); '];
  dbstop( 'in',fun, 'at',num2str(lne), 'if',h );

function ret = returnfalse( varargin );
  ret = false;

function ret = assign( name, value );
  try; ret = evalin( 'caller', [name ';'] );
  catch; ret = []; end;
  assignin( 'caller', name, value );
```

Listing 9: `evalat`

What happens in Listing 9. The function `evalat` accepts three arguments, `fun` is the function where code shall be injected, `lne` is the line number where code shall be injected, `h` is an anonymous function to be executed at the specified position.

Upon calling with three arguments[10], the anonymous function `h` is stored in the persistent[11] variable `cache`. A persistent variable retains its values between function calls. Then, the function `dbstop` adds in the function `fun` at the specified position `lne` a conditional breakpoint, which will execute the code listed in Listing 10; We put those lines of code in its own listing for better readability.

```
returnfalse( assign('handle',evalat()) ) || ...
  returnfalse( handle() );
```

Listing 10: Code of conditional breakpoint in `evalat`

Now, when the function `fun` is called and the program flow reaches line `lne` the function `evalat` is called without arguments. Thus, `evalat` returns the value of the persistent variable `cache`, this is exactly the anonymous function `h` we want to execute. The anonymous function `h` is passed to the function `assign`, which creates a variable with name `'handle'` in the workspace of the function `fun`. All of this code is wrapped inside a call to `returnfalse`[12] , which ensures that always `false` is returned.

A conditional breakpoint only stops when the injected code returns `true`. Since the first part of the injected code returns `false` the second part is evaluated. Now the just assigned variable `handle`, which is our anonymous function `h`, is executed. The result of the anonymous function `h` is passed again to `returnfalse` which again ensures that `false` is returned. Thus, the debugger does not stop the program.

[10]The function `nargin` returns the number of passed arguments. Note that, calling a function without parentheses is equivalent to calling it without arguments, i.e. `nargin` is the same as `nargin()`.
[11]Persistent variables retain their value between calls to the function. They get automatically initialized with `[]`.
[12]If the function's arguments are called `varargin`, then the variable `varargin` contains all passed arguments wrapped inside a cell array.

A usage example of `evalat` is given in Listing 11.[13,14,15]

```
>> evalat( 'surf', 1, @() disp('Hello World!') );
>> surf( membrane );
Hello World!  % and the Matlab membrane is plotted
```

Listing 11: Usage example of `evalat`

4.2 Example implementation of `returnat`

A more involved, but shorter example shows how to programmatically return early from a function. The idea is to provoke an error at a user defined position in a function and catch the exception. As already noted, simply throwing an error in some injected code would not work, since, whenever injected code throws, the debugger stops the program. Instead, we have to make sure that an error is thrown after the injected code was executed. This we achieve by some tough means; we clear the function's workspace, see Listing 12.

```
function ret = returnat( fun, lne );
  if( nargin==0 );
    evalin( 'caller', 'clear' );
    ret = false;
  else;
    dbstop( 'in',fun, 'at',num2str(lne), ...
            'if','returnat' );
    try; eval( fun );
    catch me; disp( me ); end; end;
```

Listing 12: `returnat`

What happens in Listing 12. The function `returnat` accepts two arguments, `fun` is the function which shall be executed, `lne` is the line number at which we want to return. When called with two arguments, `dbstop` adds a conditional breakpoint in the function `fun` at the specified position `lne`, which executes a call to `returnat` without arguments. Afterwards the function `fun` is called. When program flow reaches the specified location `lne`, `returnat` is called without arguments. Thus, the workspace of `fun` is cleared by executing `evalin( 'caller', 'clear' )`. Finally `false` is returned, so that the debugger does not stop the program. The next time a variable is accessed in the function `fun`, an error is thrown, therefore the `fun` returns. This error is caught in the `catch` block in the function `returnat`. In our example implementation we display the caught error, but any other code is equally possible. A usage example is given in Listing 13.

```
>> returnat( 'spy', 42 );
MException with properties:
identifier: 'MATLAB:refClearedVar'
   message: 'Reference to a cleared variable.'
      file: 'spy.m::42'
```

Listing 13: Usage example of `returnat`

[13]In the listing, the lines starting with "»" are supposed to be entered in the Matlab command window. The other lines present the expected result.
[14]Anonymous function are declared by an "@" character, followed by the list of arguments in parentheses.
[15]This example is tested with Matlab 2020a; with other versions it may not work.

4.3 *TTEST*

Functionality for implementing the code injection pattern, specifically the key functions `assignat`, `captureat`, `evalat`, `returnat`, are contained in the unit test framework *TTEST* for Matlab and Octave. *TTEST* has been written specifically with testing code in a scientific context in mind. It supports the testing of scripts, local and sub functions, has utilities for caching results for integration tests, and adds support for injection testing and partly for design by contract [21].

TTEST is published under a permissive open source license and available at *gitlab.com/tommsch/TTEST*. The full documentation of *TTEST*, together with a comparison of Matlab unit test frameworks, can be found in [22].

The following projects use *TTEST* (list non exhaustive): • *Auditory Modelling Toolbox* (Ver. 1.1) [23] • *ttoolboxes* [24] • *Large Time Frequency Analysis Toolbox* (Ver. > 2.4) [25]

5 CONCLUSION

We presented a pattern for the injection-based refactoring as a means for handling otherwise not testable code, along with a set of functions suitable for its implementation. We gave examples on its usage and practical advice for their implementation in scripting languages via making use of the debugger‘s functionalities. We provide a free implementation of the pattern‘s key functionality in our *TTEST* unit testing framework.

Further work comprises enhancements of the usability and customizability of the pattern by improving on the underlying functionality in the toolbox, e. g. by implementing a `gotoat` function, allowing for the direct execution of arbitrary sections in the code.

ACKNOWLEDGMENTS

This work has been supported by the Austrian Science Foundation (FWF) grant P33352-N, and by the European Union (EU) within the project SONICOM grant 101017743, RIA action of Horizon 2020. I want to thank my shepherd Eden Burton for taking the burden of shepherding me.

Last I want to thank my supervisor Maria Charina for her ever lasting support.

REFERENCES

[1] William Opdyke, *Refactoring Object-Oriented Frameworks*, Thesis, University of Illinois, 1992.
[2] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2018.
[3] Chris Karelis , Ioannis Megas, Apostolos V. Zarras, *How to Test the Extract Method Refactoring*, EuroPLoP '20, July 1–4, 2020, doi: 10.1145/3424771.3424793.
[4] Jehad Al Dallal, *Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics*, Inf. Softw. Technol., 54 (2012) 10, doi: 10.1016/j.infsof.2012.04.004
[5] Bian Yixin, Su Xiaohong, Ma Peijun, *Identifying Accurate Refactoring Opportunities Using Metrics*, In: ICSCTEA 2013, 141–146, doi: 10.1007/978-81-322-1695-7.
[6] Apostolos Ampatzoglou, Paris Avgeriou, Sofia Charalampidou, Alexander Chatzigeorgiou, Antonios Gkortzis, *Identifying extract method refactoring opportunities based on functional relevance* IEEE Trans. Softw. Eng., 43 (2017) July, doi: 10.1109/TSE.2016.2645572.
[7] V. Krishna Nandivada, Jyothi Vedurada, *Refactoring opportunities for replacing type code with state and subclass*, Proc. ICSE-C 2017, 305–307, doi: 10.1109/ICSE-C.2017.97.
[8] Zeba Khanam, Sam A. M. Rizvi, *A methodology for refactoring legacy code*, 2011 3rd International Conference on Electronics Computer Technology, (2011) 198–200, doi: 10.1109/ICECTECH.2011.5942080.
[9] Laurie J. Hendren, Soroush Radpour, Max Schäfer, *Refactoring MATLAB*, in: CC 2013, 224–243, doi: 10.1007/978-3-642-37051-9_12.
[10] Michael Feathers, *Working Effectively with Legacy Code*, Prentice Hall PTR, USA, 2004.
[11] James M. Bieman, Upulee Kanewala, *Testing scientific software: A systematic literature review* Inf. Softw. Technol., 56 (2014) 10, 1219–1232, doi: 10.1016/j.infsof.2014.05.006.
[12] Michelle Hirsch, The MathWorks, *How common is MATLAB in academia?*, mathworks.com/matlabcentral/answers/176635, 21-9-15.
[13] Vladimir Khorikov, *Unit Testing Principles, Practices, and Patterns*, O'Reilly, 2020.
[14] Gerard Meszaros, *xUnit Test Patterns - Refactoring Test Code*, Addison-Wesley Longman, 2007.
[15] Per Isakson, *tracer4m*, (2016), mathworks.com/matlabcentral/fileexchange/28929, 2022-02-14.
[16] David Schuler, Andreas Zeller, *Javalanche: Efficient mutation testing for Java*, ACM SIGSOFT (2009), doi: 10.1145/1595696.1595750.
[17] Cygnus Solutions, *libGDB – A library architecture for GDB*, sourceware.org/gdb/-papers/libgdb/libgdb.html, 2022-02-09.
[18] The LLDB Team, *The LLDB Debugger* lldb.llvm.org/, 2022-02-09.
[19] Microsoft, *Debugging Tools for Windows 10 (WinDbg)*, docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools, 2022-02-09.
[20] *GDB/MI*, sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html, 2022-02-09.
[21] Bertrand Meyer, *Applying 'design by contract'* in *Computer*, 25 (1992) 10, 40–51.
[22] Clara Hollomey, Thomas Mejstrik, *TTEST*, (2021), gitlab.com/tommsch/TTEST. 22-02-08.
[23] Piotr Majdak, Clara Hollomey, Robert Baumgartner, *AMT 1.0: the toolbox for reproducible research in auditory modeling*, submitted to Acta Acustica, http://amtoolbox.org.
[24] Maria Charina, Costanza Conti, Thomas Mejstrik, Jean-Louis Merrien, *Joint spectral radius and ternary Hermite subdivision*, Adv. Comput. Math., 47 (2021) 25, doi: 10.1007/s10444-021-09854-x.
[25] Zdeněk Průša, Peter L. Søndergaard, Nicki Holighaus, Christoph Wiesmeyr, Peter Balazs, *The Large Time-Frequency Analysis Toolbox 2.0*, Sound, Music, and Motion, Lecture Notes in Computer Science 2014, 419–442, http://ltfat.org.