Neil Ostrove
Chemical Abstracts Service

TFUI in Tcl/Tk

Exploiting the Humble Dialog Box

Table of Contents

TFUI in Tcl/Tk

Exploiting the Humble Dialog Box

Abstract

Test First User Interface (TFUI) design is a hot topic of discussion with a very active Yahoo newsgroup. I recently needed to develop a database editing program in Tcl/Tk and, having just read "Test-Driven Development: A Practical Guide" by Dave Astels, decided to use the unit tests from that book (suitably modified for language and application) to drive the editor design. The result was a small flexible application with an extremely clean *Model*/*View*/*Controller* (technically *Model*/*View*/*Presenter* [Fowler2004]) design and a comprehensive set of unit tests. I believe Tcl/Tk and [incr Tcl/Tk] to be friendlier to the Test-Driven Development (TDD) of user interfaces than the original Java example.

This paper re-implements the Movie List project from Dave Astels's book [Astels2003] using [incr Tcl/Tk] and the tcltest framework. In the course of this I discuss the various aspects of TDD concentrating on the issues most relevant to developing user interfaces primarily through unit tests written in the test first style.

Introduction

Why I Wrote This Paper

I have been doing Test Driven Development (TDD) for about five years and thought I understood it well. The practice caused a definite quality improvement in the code I write. But I hadn't been satisfied with the user interface code. Tcl/Tk is extremely powerful and convenient and it's very easy to make "smart" user interfaces. Too smart to test. Even when I tried to separate it out, there was too much untestable functionality directly mixed in with the GUI objects. Then I read *Test Driven Development: A Practical Guide* [Astels2003] and discovered that I had never really understood *Model*/*View*/*Controller*. For my next project I decided to use Dave's tests as a guide to my design to see if the ideas worked in Tcl/Tk the way they did in Java.

I was very happy with the result and decided to communicate this at Tcl 2005. To avoid using proprietary code, I'm re-implementing a portion of Dave's Java Project, a program to organize a list of movies with associated ratings and reviews. I find that it's working even more nicely, partly due to the choice of using [incr Tk] mega-widgets (which I hadn't used before).

The first part of this paper talks about the ideas of TDD and Test First User Interface (TFUI) development and the *Model*/*View*/*Presenter* design. The last (and longest) part is a blow by blow transcription of the Movie List project designed to try to communicate the experience of using the techniques. It parallels [Astels2003] but can be read independently. However, I definitely recommend reading [Astels2003] for its deeper treatment of the topics here.

Who Should Read This Paper?

This paper is for people interested in developing Tcl/Tk programs with rich user interfaces using Test First styles of software design, Test Driven Development (TDD) and Test First User Interfaces (TFUI). People who want to understand how to use the *Model*/*View*/*Presenter* pattern should also find the paper helpful.

Readers should be somewhat familiar with the Tcl programming language and the Tk widget set, but previous experience with [incr Tcl/Tk], TDD, and TFUI should not be required. The techniques shown here can be applied to any programming language.

TDD and TFUI

What is TDD?

Test Driven Development (TDD) is a style of programming in which (ideally) you don't write any code until you're forced to by a failing test. It ensures that you think about how a piece of code is going to be used *before* you create it. When you use TDD, you end up with not only the code implementing your application, but a comprehensive set of programmer tests as well. The method was originally called Test First Programming and its fundamental principle is: Write the Tests First.

TDD was developed by Kent Beck, and is an iterative method. Development proceeds in a rhythmic series of cycles, and each cycle follows the rules for Test Driven Development [Beck2003]:

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

When you follow these rules, the tests drive the design. This doesn't mean that you don't think about the design before you start; it does mean that you don't get yourself "locked" into a design and follow it slavishly. The tests, "code smells", and refactorings you'll encounter will often lead you away from your preconceived ideas. Let them. TDD works

best as an emergent design method. Once you get used to it, you'll be surprised how little "Big Design Up Front" (BDUF) you really need.

Refactoring [Fowler1999] may be an unfamiliar concept. The word can be used both as a noun and a verb. A refactoring is a small change that improves the design of a program without changing its behavior. Refactoring is the process of making those changes. How do you know that the behavior of a program doesn't change? That's where the tests come in. If you have a comprehensive suite of programming tests for your code, then you can be more confident that your refactoring didn't break anything. What's a good way to have a comprehensive test suite? TDD.

How do you know that you're actually improving the design when you refactor? For that there are Kent Beck's rules for simple design:

A program should

1. run all the tests,
2. contain no duplicate code,
3. express all the ideas the author wants to express
4. minimize classes and methods

These rules should usually be applied in order. You'll see them in action later.

TFUI

What is TFUI?

Test First User Interfaces (TFUI) is the application of TDD to user interface programming.

Why Are User Interfaces Hard to Test?

Traditional User Interface (UI) testing is a manual labor intensive procedure. A person needs to sit down at a workstation, exercise the software, and look for problems. Some of this is necessary. Only a person can tell that a UI is confusing or hard to use. But manual testing is tiring and error prone. Manual regression tests, to ensure that previously working software still works, are especially wasteful of human time.

Some GUI toolkits deny the programmer access to needed information about the "widgets" composing the user interface. They can also make it difficult to manipulate the interface, e.g. push a button, programmatically. These issues can be worsened if the interface is not actually displayed on a physical device. In some cases the GUI code is generated automatically by IDE "wizards" and can be hard to read.

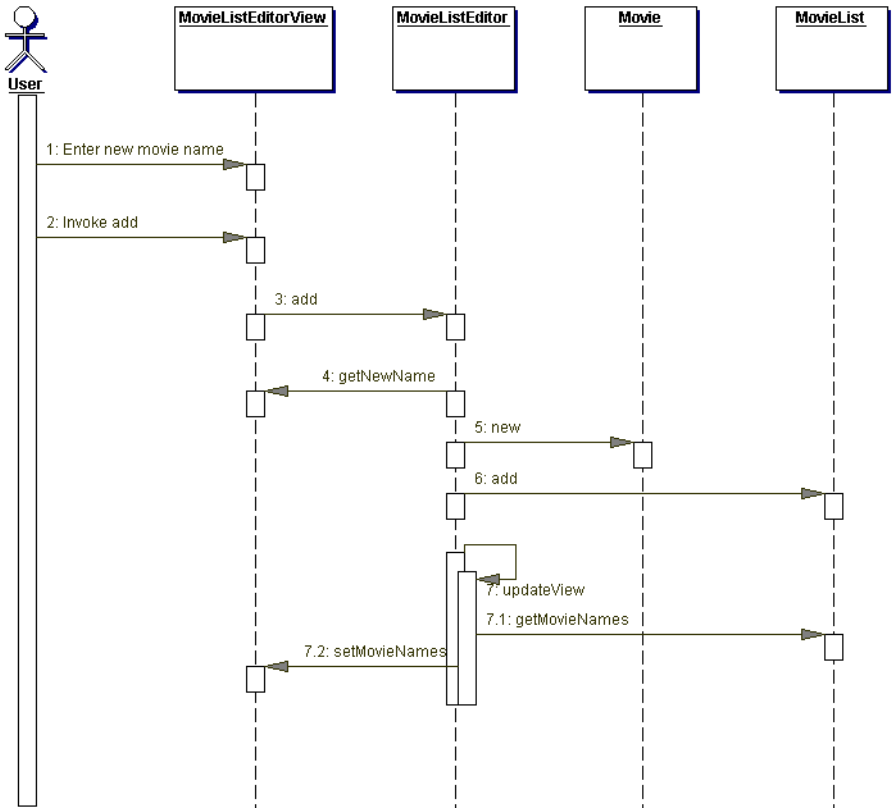
Traditional GUI testing tools use “capture and replay” to automate an initial manual test. This eases regression testing. Early versions of the tools captured button clicks and key strokes and did bit by bit compares on windows to detect differences from the original results. They were very brittle; moving a button (or even the position of a window on the screen) could invalidate a test script. Newer tools generally hook into the GUI toolkit and are more robust, but still usually test only the surface of the application

Model/View/Presenter

Using TDD does not make user interfaces easy to test, but it opens them up for testing. A design paradigm called *Model-View-Presenter* [Fowler2004], a variant of *Model-View-Controller*, can be used to reduce the amount of functional code bundled with the user interface. *Model-View-Presenter* divides software, like Gaul, into three parts. The domain specific parts of the software, the guts of the functionality, are in the *Model*. The user interface portion of the software is confined to the *View*. The *View* should know as little about the domain model as possible. The *View* displays the GUI elements and receives events such as button presses. Between the *Model* and the *View* is the *Presenter*, which is the glue that connects the UI with the functionality. The *View* translates the events it receives to responsibilities of the *Presenter*. This is the fundamental difference between *Model-View-Presenter* and *Model-View-Controller.*

In the Movie List example, to add a movie to the list you push the Add button. In the *View* (the MovieListEditorView class in the code), all that’s done is to delegate the functionality to the *Presenter* (the MovieListEditor class’s add method). The *Presenter* then asks the *View* for the name of the movie and tells the *Model* (Movie and MovieList classes) to create a new movie and add it to the movie list. The *View*/*Presenter* interaction is done completely through interface methods so that the *Presenter* does not see GUI specific information such as the actual event.

The following UML sequence diagram (extracted from the code developed in **Test007**) nicely illustrates what a *Presenter* does. The user pushes the Add button and the *View* (named MovieListEditorView) receives the event. MovieListEditorView simply calls the add method of the *Presenter* (MovieListEditor) which is the logical layer that decides how to add a movie. The *Presenter* needs the name of the movie to be added. It gets it by asking the *View*. Notice that the information was **not** included as an argument to the add method; that would have required the *View* to know the semantics of adding a movie and what information was needed. We’re trying to keep our *View* humble here. The *Presenter* then goes ahead and creates a new movie with the correct name and adds it to the movie list. Finally it makes sure the *Model* and the *View* are in sync.



The *View* thus consists of a set of user interface components (buttons, entry fields, etc.), and the ability to delegate any events (button pushes, carriage returns, etc.) to the *Presenter*. TDD allows you to test that the *View* contains the necessary components; that the *Model* implements the necessary functionality; and that everything is wired together correctly through the *Presenter*. Testing usability is still hard.

This style of *View* is often called the Humble Dialog Box [Feathers2002] and does not imply any reduction of features of the user interface. They're just implemented somewhere else. Using the Humble Dialog Box, it's possible for user interfaces to be both rich and thin.

The TFUI Testing Pattern

I've found, once the *Model*/*View*/*Presenter* separation is made, that adding a GUI feature test first using TDD/TFUI tends to follow a pattern.

1. Augment the *Model* to provide the feature functionality (writing the tests first).

2. Provide a way to invoke that functionality through the *Presenter* (writing the test first).

3. Add the required GUI elements to the *View* (writing the tests first).

4. Replace the *Presenter* invocation in the test with a *View* invocation.

5. Add behavior to the GUI elements to invoke the *Presenter* and make the test(s) pass.

6. Examine the GUI visually.

This is hard to describe clearly, but you can see it in action in tests **Test005**-**Test006**, **Test007**-**Test008**, **Test014**-**Test015**, and **Test016**-**Test017** where that pattern is repeated.

Interaction and State Testing

Interaction Testing

The tests developed with TDD are usually referred to as Programmer Tests and usually test the functional aspects of a design rather than other aspects like performance (though these aspects can often be tested with the same tools). The purpose of the tests is to lead you to a design. Programmer Tests are not the same as traditional Unit Tests, but thinking about the program units exercised by a test can be helpful.

It is often useful to reduce the "unit" tested by a Programmer Test, e.g., to speed up the test suite. This is often done using *Mock Objects*. A Mock Object satisfies the interface (responds to all the methods) of a production domain object, and stores its interactions for later verification. This is *Interaction Testing* and it allows you to simulate interactions with expensive or slower resources like a database or a GUI widget.

Dave Astels [Astels2003] uses Mock Objects representing Java Swing GUI elements for many of his tests. Several Mock Object frameworks exist for these and he compares them.

I am not aware of any Tcl/Tk Mock Object frameworks and I don't use Interaction Testing in this paper. I believe the reflection capabilities of Tcl would make such frameworks easy to implement, but I haven't found the need to do so. State Testing has been more than adequate.

State Testing

The classic Programmer Test sets up a situation and tests that the program state is what was expected. This is *State Testing* and I've found it sufficient for this paper. Details are covered with the initial **Test000** below.

The [incr Tcl/Tk] source is available. I try to keep enough context so you can see the evolution of the system. The source code for each test can be found in the directory named for that test.

Test000 - Hookup

Simple Test

[Astels2003] starts up immediately with Test 1 implementing the Movie List, but I like to start with a standard test just to make sure the environment is working before I start implementing. Ron Jeffries calls this initial test a Hookup.

Here's an example of a simple tcltest:

simple.test

```
#!/bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

package require tcltest

tcltest::test simple-test { actual should be expected } {
   set actual expected
} expected

tcltest::cleanupTests
```

The first three lines invoke the Tcl interpreter in a Unix environment (and will not be shown hereafter) and the next line specifies the tcltest unit test package. The actual test has a *name* (simple-test), a *description* (actual should be expected), a *body* (set actual expected) and a *result* (expected). If the value of the body matches the expected value, running the test produces

```
simple.test:  Total  1    Passed  1    Skipped 0    Failed  0
```

The statistics are provided by the cleanupTests command. A passed test is otherwise silent. If the value of the body has an unexpected value, running the test produces

```
==== simple-test actual should be expected FAILED
==== Contents of test case:

   set actual unexpected

---- Result was:
unexpected
---- Result should have been (exact matching):
expected
==== simple-test FAILED

simple.test:  Total  1     Passed  0     Skipped 0     Failed  1
```

This style of test is often used for traditional unit tests of Tcl and its libraries.

Boilerplate Tests

I like to see explicitly when my tests pass, and I often share setup and cleanup code among several tests. The tcltest framework provides these features, and I generally copy in the following "boilerplate" (slightly modified from an example in the tcltest man page) for my initial test

Something.test

```
package require tcltest

tcltest::configure -verbose pse

#source "./Something.tcl"

namespace eval ::Something::test {
namespace import ::tcltest::*

variable SETUP {
  # put common setup code here
}
variable CLEANUP {
   # put common cleanup code here
}

test passing-test {
   This test will pass trivially
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   lappend result 42
   set result
} -cleanup {
   eval $CLEANUP
} -result [list \
   42 \
   ]

test failing-test {
   This test will fail trivially
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   lappend result 3.14159
   set result
} -cleanup {
   eval $CLEANUP
} -result [list \
   2.71828 \
   ]

   cleanupTests
}
```

This test usually will be testing the code in Something.tcl and will usually be in a file called Something.test. Dave Astels occasionally uses other conventions and I will often follow his conventions to make it easier for people to follow this code with his commentary.

I use the verbose options

```
tcltest::configure -verbose pse
```

The 'p' verbose option causes tcltest to print an indication of passing tests (often called "green bar" from the JUnit GUI, a failing test is a "red bar"). The 's' option causes tcltest to tell you when a test has been skipped, and the 'p' option causes tcltest to print error information. Experiment with these and other options to find the ones that suit your individual style.

Wrapping the tests in a namespace

```
namespace eval ::Something::test {
namespace import ::tcltest::*
...
}
```

allows you to reuse the names of tests in different contexts (you'll see several tests named size) and the import statement removes the need for the tcltest prefix within the namespace.

The next section defines variables that are usable in all the tests within the namespace. Their most common use is to allow different tests to use the same *test fixtures*. A fixture is just a configuration of domain objects used by more than one test.

The *setup* section is evaluated before each test and the *cleanup* section after each test. The common variables SETUP and CLEANUP allow you to easily share setup and cleanup code among tests.

Most of my tests follow the pattern

1. Set up the objects to be tested (in setup as much as possible).

2. Manipulate the objects in the body and store values for comparison in the result list.

3. Put the expected value(s) in the result section (be careful of Tcl order of evaluation, the argument of result is evaluated before anything else).

4. Clean up the objects (and anything else) in the cleanup section. The tests should be as independent as possible.

This pattern is called *State Testing*. Also, while I'll be using [incr Tcl/Tk] and Object Oriented Programming and Design in this paper, the tests and test framework don't require them. They can be used just as well with imperative or functional style programming. *Interaction Testing* uses Mock Objects to capture the way objects interact. TFUI often employs interaction style testing (and Dave Astels has an excellent chapter on Mock Objects in his book), but I will not be using it in this paper.

I use a result list because I often have multiple items in one test. For example, In **Test008** I have a has-components test that checks that the user interface has a scrolling list for displaying movies and both a New Movie text entry field and an Add pushbutton for adding new movies to the Movie Lister. Some people believe that each individual test should test only one thing. Find a style that works for you.

The boilerplate tests produce the result

```
++++ passing-test PASSED

==== failing-test This test will fail trivially FAILED
---- Result was:
3.14159
---- Result should have been (exact matching):
2.71828
==== failing-test FAILED

Something.test: Total   2      Passed  1      Skipped 0      Failed  1
```

When there is more than one test file, as here, it is convenient to be able to run all the tests with a single command. The following script (taken from [tcltest])

RunAllTests.tcl

```
package require Tcl 8.4
package require tcltest 2.2
tcltest::configure -testdir [file dirname [file normalize [info script]]]
eval tcltest::configure $argv
tcltest::runAllTests
```

will run all the tests defined in files with the suffix .test (the tcltest::configure default) that are in the same directory as itself. You can customize it if you want a more complex source and test structure. Running the script gives the result

```
Tests running in interp:  /usr/tcl84/bin/tclsh
Tests located in:  .../Test000
Tests running in:  .../Test000
Temporary files stored in .../Test000
Test files run in separate interpreters
Running tests that match:  *
Skipping test files that match:  l.*.test
Only running test files that match:  *.test
Tests began at Fri Aug 26 18:29:49 EDT 2005
Something.test
++++ passing-test PASSED

==== failing-test This test will fail trivially FAILED
---- Result was:
3.14159
---- Result should have been (exact matching):
2.71828
==== failing-test FAILED

simple.test

Tests ended at Fri Aug 26 18:29:51 EDT 2005
RunAllTests.tcl:       Total   3       Passed  2       Skipped 0       Failed  1
Sourced 2 Test Files.
Files with failing tests: Something.test
```

Test001 – Size of empty list should be 0

Write a test

Once my boilerplate test is working, I edit it into my first “real” system oriented test.

MovieList.test

```
package require tcltest

tcltest::configure -verbose pse

source "./MovieList.tcl"

namespace eval ::MovieList::test {
namespace import ::tcltest::*

variable SETUP {
  # put common setup code here
}
variable CLEANUP {
   # put common cleanup code here
}

test empty-list-size {
   Size of empty movie list should be 0.
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   MovieList emptyList
   lappend result [emptyList size]
   itcl::delete object emptyList
   set result
} -cleanup {
   eval $CLEANUP
} -result [list \
   0 \
   ]


   cleanupTests
}
```

Notice that I wrote this test (actually Dave Astels wrote this test originally in Java) before writing any code. Writing the test requires making design decisions, for example, that there will be a class (type) called MovieList that has a method called size. This leads to the mantra “TDD is a design method, not a test method”.

Short digression on [incr tcl]: You create an object of a particular class by following the name of the class with the name of the object. Methods are like functions or procs that are associated with a particular class. Always call a method with an object of its class. You call a method by putting the object name before the method name and putting any

arguments to the method after the method name. Objects exist until you explicitly delete them.

```
MovieList emptyList
emptyList size
itcl::delete object emptyList
```

See [Smith2000] for more information.

By the way, I very much like the "…should be…" style of test description. It is well suited to state testing, which I use most frequently. The description makes sense when read either as the documentation for the test or as a reason for failure when printed with a failing test. (The size of an empty movie list should be 0 [*but it isn't*].)

Make it compile

When I run the test it not only fails, it doesn't even compile. The error message

```
couldn't read file "./MovieList.tcl": no such file or directory
```

tells me it's trying to source in the MovieList.tcl file and not finding it. I do the minimum amount necessary to fix the problem; I create an empty MovieList.tcl file.

The next error message is

```
invalid command name "MovieList"
```

which forces the creation of an empty MovieList class

MovieList.01.tcl

```
package require Itcl

itcl::class MovieList {

}
```

Filenames with a number (like MovieList.01.tcl) are intermediate stages toward getting a test to pass. They are included for reference with the source code for this paper. (In real life, I rely on my Change Management System or IDE for this.)

This implementation file produces the next error message

```
bad option “size”: should be one of …
```

This forces the creation of an empty size method

MovieList.02.tcl

```
package require Itcl

itcl::class MovieList {

    public method size { } {
    }

}
```

Run it to see that it fails

The test finally compiles, and immediately fails with the error

```
Result was
{}
Result should have been
0
```

to take me to the next step in the TDD cycle.

Make it run

I can now make the simplest possible change to get the test to pass, leading to

MovieList.tcl

```
package require Itcl

::itcl::class MovieList {

   public method size { } {
      return 0
   }

}
```

Now the test passes

```
++++ empty-list-size PASSED
MovieList.test: Total  1     Passed  1     Skipped 0     Failed  0
```

Love that green bar.

Could I have just written the class and method in the first place? Well, yes. This is simple enough that I could see how it was going to go. That is a testing pattern [Beck2003] known as "obvious implementation". However, for the first test I chose to do things with the smallest steps I could. This is another testing pattern, "fake it until you make it". Here you write no code except in response to a failing test. For this pattern you ask yourself "What is the simplest thing that could possibly work?" (Extreme Programming trademark mantra) and try to do only what is necessary for the existing tests.

Doing it this way has the advantage of maximizing your test coverage. Every line of code should be executed by some test. Path coverage metrics also look good when you use this discipline. It's nice to be able to say to your management that every line of your code has been tested. (I know, it's nowhere near sufficient, but it's still nice to say.) Can I say that I always write no code except in response to a failing test (or when refactoring, which covers a lot of ground)? Well, I can say it but, as we'll see later, I'd be lying. Even when I do follow that principle, I won't always be this pedantic about writing up every step.

Remove duplication

Beck's rules are used to guide refactoring

- All the tests pass.
- There is no obvious duplication in the MoveList class.
- There is a domain concept called MovieList with an associated size.

- There is only one class and only one method.

So there's not much to refactor at this point.

Technically, there is duplication. It is between the 0 in the size method and the 0 in the expected result of the test. That duplication will be removed later.

Test002 – Size of one item movie list should be 1

Write a test

The second test

MovieList.test

```
…
source "./MovieList.tcl"
source "./Movie.tcl"
…
test size-after-adding-one {
    Size of one item movie list should be 1.
} -setup {
    eval $SETUP
} -body {
    set result [list]
    MovieList oneItemList
    Movie starWars
    oneItemList add starWars
    lappend result [oneItemList size]
    itcl::delete object oneItemList
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    1 \
    ]
```

is placed in the file MovieList.test, after the first test but before cleanupTests. It also contains a number of design decisions. For example, a MovieList contains objects of type Movie and now has an add method. I am following along with Dave's design decisions by using his tests and (in most cases) following his implementation. However, TDD is not a mechanical method that always yields the same code from the same requirements the way Jackson System Design was supposed to (anyone remember Jackson System Design?) You can "drive" a design by choosing tests to make it go in a particular direction. For example, I could have had this test add movies by name with the line

```
oneItemList add "Star Wars"
```

and delayed (or possibly eliminated) the Movie class. In most cases I'll follow Dave's design so that this paper can be compared with his book (this paper is not meant to be a replacement for the book), but there will be a few places where I make different decisions. I might do this because Tcl/Tk has different ways of doing some things than Java, or simply because my aesthetic sense is different from his in places.

Make it compile

This test drives the design in ways similar to the last test. First create an empty Movie.tcl file, and then seed it with an empty Movie class.

Movie.tcl

```
package require Itcl

::itcl::class Movie {

}
```

Finally create an empty add method in MovieList (MovieList.01.tcl) and the test compiles.

Run it to see that it fails

It does. The message is what you'd expect (the size is still 0) so I'll skip it.

Make it run

What's the simplest change that could make the test pass? Just flag that add's been called.

MovieList.tcl

```
package require Itcl

::itcl::class MovieList {

   public method size { } {
      return $numberOfMovies
   }

   public method add { movie } {
      set numberOfMovies 1
   }

   private variable numberOfMovies 0
}
```

This now illustrates an [incr Tcl] class that not only has methods, but a member variable as well. The *private* modifier means that the variable should not be accessed from outside the class (though there are ways around that); the *public* modifier on the methods means that they are accessible.

Remove duplication

Again, there's not much duplication here other than the values (0 and 1) in both the code and tests. It doesn't need refactoring yet.

Test003 - Size of two item movie list should be 2

Next test. (Imagine the five-step rhythm yourself.)

l.MovieList.01.test

```
test size-after-adding-two {
    Size of two item movie list should be 2.
} -setup  {
    eval $SETUP
} -body  {
    MovieList twoItemList
    Movie starWars
    Movie starTrek
    twoItemList add starWars
    twoItemList add starTrek
    set result [list]
    lappend result [twoItemList size]
    itcl::delete object twoItemList
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    2 \
    ]


    cleanupTests
}
```

The number in the file name l.MovieList.01.test again indicates that this is an intermediate version of the test. The prefix **l.** (letter l) prevents it from executing with RunAllTests.tcl. These intermediate versions can be found with the source for reference.

The first compile error is a bit technical

```
command “starWars” already exists in namespace
```

Objects in [incr tcl] need to be explicitly deleted. I remembered to delete the MovieList oneItemList in the test size-after-adding-one, but forgot to delete the Movie starWars. The size-after-adding-two test failed when it tried to create the same object again. Once I added the delete object command, test size-after-adding-two gave me the “expected 2, got 1” failure I was expecting.

The simplest way to make the test pass is to count the movies added, and that gives

```
package require Itcl

::itcl::class MovieList {

    public method size { } {
        return $numberOfMovies
    }

    public method add { movie } {
        incr numberOfMovies
    }

    private variable numberOfMovies 0
}
```

I need to remember to add the explicit movie deletions to the new test.

MovieList.test

```
test size-after-adding-two {
    Size of two item movie list should be 2.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    MovieList twoItemList
    Movie starWars
    Movie starTrek
    twoItemList add starWars
    twoItemList add starTrek
    lappend result [twoItemList size]
    itcl::delete object twoItemList
    itcl::delete object starWars
    itcl::delete object starTrek
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    2 \
    ]
```

All tests pass. Time to refactor.

Test003 Refactored

Remove duplication

The duplication in the values 0 and 1 between the tests and the production code is no longer there. But that doesn't mean we're done. There's still a lot of duplication and it's all in the tests. For example, each test creates [and destroys] a MovieList. Two of them also create and destroy Movies. Move object creation into a common SETUP variable:

```
                                                    l.MovieList.01.test
variable SETUP {
    MovieList movieList
    Movie starWars
    Movie starTrek
}
```

I now get the error message

```
command "movieList" already exists in namespace
```

when the *setup* before the second test tries to create movieList while it still exists from the setup before the first test. Put in the deletes

```
                                                    l.MovieList.01.test
variable CLEANUP {
    itcl::delete object movieList
    itcl::delete object starWars
    itcl::delete object starTrek
}
```

Now test empty-list-size passes (it doesn't use any of the Movies) but the rest now fail because of duplicate Movie creation. They all need to pass.

First test empty-list-size is modified to use the SETUP MovieList

I.MovieList.02.test

```
test empty-list-size {
    Size of empty movie list should be 0.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    lappend result [movieList size]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    0 \
    ]
```

It still passes. Ideally, refactoring should not break any tests. I'll need to think about this after I get the other tests to pass. Just changing them to use the SETUP objects should be enough.

Test size-after-adding-one becomes

I.MovieList.03.test

```
test size-after-adding-one {
    Size of one item movie list should be 1.
} -setup  {
    eval $SETUP
} -body  {
    movieList add starWars
    set result [list]
    lappend result [movieList size]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    1 \
    ]
```

and passes. Test size-after-adding-two becomes:

MovieList.test

```
test size-after-adding-two {
    Size of two item movie list should be 2.
} -setup  {
    eval $SETUP
} -body  {
    movieList add starWars
    movieList add starTrek
    set result [list]
    lappend result [movieList size]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    2 \
    ]
```

and also passes.

OK, I'm safe. Time to think about what just happened. It's usually a good idea to have only one test failing at a time. When refactoring, all the tests should pass before and after each refactoring. When a number of tests break at once, it usually means I'm trying to do too much in one step. Kent Beck compares tests to the ratchet on a well crank that allows you to rest without losing ground while trying to raise a bucket of water. The heavier the bucket, the more closely spaced the ratchet teeth should be. When you're doing something that's harder, take smaller steps.

When the tests broke, I could have backed out the changes and restarted by adding the creation and destruction of just the movieList object to SETUP and CLEANUP. Then changed empty-list-size to use the common fixture. Then added the starWars object and changed size-after-adding-one. And finally added the starTrek object and changed size-after-adding-two. That probably would have kept all the tests passing throughout the process.

Test004 – Movie list should include added items

So far we haven't needed the Movie class at all. Adding a movie to the list just bumped a count and didn't require anything to keep track of the movies. To change that, I need a test that checks that an added movie is still in the list:

MovieList.test

```
test contents {
    Movie list should contain added items and not contain other items.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    movieList add starWars
    movieList add starTrek
    lappend result [movieList contains starWars]
    lappend result [movieList contains starTrek]
    lappend result [movieList contains starGate]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    1 \
    1 \
    0 \
    ]
```

This test postulates a method named contains that returns true (1) if its argument is in the movieList, and false (0) if it isn't. This test will drive the MovieList class design to keep track of its movies. Notice that I'm finally taking advantage here of the boilerplate result list to test more than one result item within the single test. I create a dummy contains method (MovieList.01.tcl) to get the test to compile. Now the new test (and only the new test) fails.

Change MovieList to keep around a list of movies (called movies), and change add to append a movie to the list with lappend and size to return the length of the list with llength. Notice that the old numberOfMovies increment is still in add. This keeps the old tests passing between changing add and changing size. As before, I don't want more than one failed test around at a time.

MovieList.02.tcl

```
package require Itcl

::itcl::class MovieList {

   public method size { } {
      #return $numberOfMovies
      return [llength $movies]
   }

   public method add { movie } {
      incr numberOfMovies
      lappend movies $movie
   }

   public method contains { movie } {
      return 0
   }

   private variable numberOfMovies 0
   private variable movies {}
}
```

All the old tests still pass and the new test contents still fails. To make it pass, implement contains with the Tcl list search function lsearch.

MovieList.03.tcl

```
   public method contains { movie } {
      return [expr { [lsearch -exact $movies $movie] >= 0 }]
   }
```

The –exact argument to lsearch forces an exact match of contents, and lsearch returns a zero based index to a found movie. It returns –1 if a match is not found. The boolean expression is evaluated with expr to return true (1) or false (0). All the tests pass.

The old numberOfMovies implementation is still there. Now that all the tests pass, it's safe to remove it.

MovieList..tcl

```
package require Itcl

::itcl::class MovieList {

   public method size { } {
      return [llength $movies]
   }

   public method add { movie } {
      lappend movies $movie
   }

   public method contains { movie } {
      return [expr { [lsearch -exact $movies $movie] >= 0 }]
   }

   private variable movies {}
}
```

All the tests still pass.

Test004 Refactored

All the tests in MovieList.test use the same setup and cleanup test fixture, but only two (size-after-adding-two and content) take full advantage of all the objects set up in it. One way to organize tests (there are many) is according to the program structure. Each class file like MovieList.test (assuming you didn't place more than one class in a file) would have a corresponding test file like MovieList.test. If a particular method required a large number of tests, those tests might be split into its own file named for the method. This is the method used by tools that automatically generate test stubs from code (sort of the opposite of test first programming).

Another way gathers together all tests that use the same test fixture, that is, the same objects and internal state created by the *setup* and *cleanup* sections. Dave uses this convention and divides the tests into the following separate files.

MovieListWithTwoMovies.test

```
package require tcltest

tcltest::configure -verbose pse

source "./MovieList.tcl"
source "./Movie.tcl"

namespace eval ::MovieList::test {
namespace import ::tcltest::*

variable SETUP {
    MovieList movieList
    Movie starWars
    Movie starTrek
    movieList add starWars
    movieList add starTrek
}
variable CLEANUP {
    itcl::delete object movieList
    itcl::delete object starWars
    itcl::delete object starTrek
}

test size {
    Size of two item movie list should be 2.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    lappend result [movieList size]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    2 \
    ]

test contents {
    Movie list should contain added items and not contain other items.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    lappend result [movieList contains starWars]
    lappend result [movieList contains starTrek]
    lappend result [movieList contains starGate]
    set result
} -cleanup {
    eval $CLEANUP
```

```
} -result [list \
    1 \
    1 \
    0 \
    ]


    cleanupTests
}
```

The way to get here is by refactoring. That is, make small changes and keep all the tests passing. In this case, I first copy MovieList.test to EmptyMovieList.test. Run all the tests with RunAllTests.tcl and they all pass. Now remove the empty-list-size test and the size-after-adding-one tests. Everything still passes. Move the add of starWars and starTrek to the *setup* section (one at a time, testing after each move of course). Rename the test size-after-adding-one to just size to remove the duplication of the name of the test with the name of the fixture. Everything still passes.

Do the copy and similar surgeries again to get

MovieListWithOneMovie.test

```
package require tcltest

tcltest::configure -verbose pse

source "./MovieList.tcl"
source "./Movie.tcl"

namespace eval ::MovieList::test {
namespace import ::tcltest::*

variable SETUP {
    MovieList movieList
    Movie starWars
    movieList add starWars
}
variable CLEANUP {
    itcl::delete object movieList
    itcl::delete object starWars
}

test size {
    Size of one item movie list should be 1.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    lappend result [movieList size]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    1 \
    ]


    cleanupTests
}
```

Finally, delete the one and two Movie tests from the original MovieList.test and rename it to get

EmptyMovieList.test

```
package require tcltest

tcltest::configure -verbose pse

source "./MovieList.tcl"
source "./Movie.tcl"

namespace eval ::MovieList::test {
namespace import ::tcltest::*

variable SETUP {
   MovieList movieList
}
variable CLEANUP {
   itcl::delete object movieList
}

test size {
   Size of empty movie list should be 0.
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   lappend result [movieList size]
   set result
} -cleanup {
   eval $CLEANUP
} -result [list \
   0 \
   ]


   cleanupTests
}
```

The refactoring is complete.

Test005 – MovieListEditor should send movie list from *Model* to *View*

I want a test that synchronizes the MovieList between the *Model* (the existing MovieList and Movie classes) and the *View* (as yet unnamed) via a *Presenter* (also unnamed). I will follow Dave and call this *Presenter* class MovieListEditor. When I know I'll need a new class but I'm not sure what it's going to look like, I often build the test up in pieces. First, this test forces the creation of the MovieListEditor class.

I.MovieListEditor.01.test

```
…
source "./MovieListEditor.tcl"
…
test list {
    MovieListEditor should send movie list from model to view.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    MovieListEditor editor
    itcl::delete object editor
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
     \
    ]
```

I sometimes leave in this test and call it creation, but all the MovieListEditor tests will necessarily test object creation, so a separate test is not really necessary. In any case, getting just this much to pass forced the creation of the empty MovieListEditor class. Now I force a MovieListEditorView class by changing the test to read

I.MovieListEditor.02.test

```
test list {
    MovieListEditor should send movie list from model to view.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    MovieListEditorView view
    MovieListEditor editor
    itcl::delete object editor
    itcl::delete object view
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
     \
    ]
```

To make the MovieListEditor the connection between the MovieList and the MovieListEditorView, I write a test with three movies in the MovieList, create a MovieListEditor, and verify that there

are three movies in the MovieListEditorView. (I really should test that they're the same three movies, but I don't remember at the moment how to compare lists in Tcl and I don't want to stop and look it up while I'm on a roll. This is enough to drive the design I want.) Now the [whole] test looks like

I.MovieListEditor.03.test

```
package require tcltest

tcltest::configure -verbose pse

source "./MovieList.tcl"
source "./Movie.tcl"
source "./MovieListEditor.tcl"
source "./MovieListEditorView.tcl"

namespace eval ::MovieListEditor::test {
namespace import ::tcltest::*

variable SETUP {
   Movie starWars
   Movie starTrek
   Movie starGate
   MovieList movieList
   movieList add starWars
   movieList add starTrek
   movieList add starGate
}
variable CLEANUP {
   itcl::delete object movieList
   itcl::delete object starWars
   itcl::delete object starTrek
   itcl::delete object starGate
}

test list {
   MovieListEditor should send movie list from model to view.
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   MovieListEditorView view
   MovieListEditor editor movieList view
   lappend result [llength [view getMovies]]
   itcl::delete object editor
   itcl::delete object view
   set result
} -cleanup {
   eval $CLEANUP
} -result [list \
    3 \
  ]
   cleanupTests
}
```

To get it to compile, I need to add a dummy getMovies method to MovieListEditorView, and add a dummy constructor with two arguments to MovieListEditor.

MovieListEditor.01.tcl

```
package require Itcl

::itcl::class MovieListEditor {

    constructor { movieList view } {
    }
}
```

The test now fails with the message

```
==== list MovieListEditor should send movie list from model to view. FAILED
---- Result was:
0
---- Result should have been (exact matching):
3
==== list FAILED
```

because the MovieListEditor constructor and getMovies don't actually do anything. I want the constructor to load the view with the movies from the movieList. Something like this.

MovieListEditor.tcl

```
package require Itcl

::itcl::class MovieListEditor {

    constructor { movieList view } {
        $view setMovies [$movieList getMovies]
    }
}
```

Running the test should now give me errors since the MovieList doesn't have a getMovies method and the MovieListEditorView doesn't have a setMovies method (not even dummy ones). I'm ready to add them, but I run the test for luck anyway. Surprise!

```
==== list MovieListEditor should send movie list from model to view. FAILED
---- Test generated error; Return code was: 1
---- Return code should have been one of: 0 2
---- errorInfo: invalid command name "movieList"
    while executing
"$movieList getMovies"
    while constructing object "::MovieListEditor::test::editor" in ::MovieListEditor::constructor (body line 2)
    invoked from within
"MovieListEditor editor movieList view"
    ("uplevel" body line 4)
    invoked from within
"uplevel 1 $script"
---- errorCode: NONE
==== list FAILED
```

What do you mean invalid command name "movieList"? The movieList is right there, created in the test. Namespaces strike again. The object movieList was created in the MovieListEditor::test namespace. It's being invoked as a command in the MovieListEditor class namespace. To use movieList as a command, you need to know its full name, including its original namespace. So the test needs to be changed to pass along to the editor the full name of the movielist (I.MovieListEditor.04.test) and the view.

I.MovieListEditor.05.test

```
…
    MovieListEditor editor \
        [namespace current]::movieList \
        [namespace current]::view
…
```

This test uses the namespace current command to prepend the current namespace to the object.

Running this test gives the

```
bad option "getMovies": should be one of...
```

I originally expected before the surprise. It might seem like the test has caused trouble, but what it did was provide an "early warning" to a namespace problem likely to come up in production code. That's a good thing.

OK, now add in the missing getters and setters. For the MovieListEditorView, we can go with the "obvious implementation"

MovieListEditorView.tcl

```
package require Itcl

::itcl::class MovieListEditorView {

    public method setMovies { moviesArg } {
        set movies $moviesArg
    }

    public method getMovies { } {
        return $movies
    }

    private variable movies {}
}
```

So, do I need separate tests (in a MovieListEditorView.test which doesn't yet exist) for this? Some people believe that accessor functions (getters and setters) don't need tests because they are "too simple to fail". Others feel that separate tests aren't needed because the accessors are adequately tested by the tests that caused you to generate them in the first place. Reasonable people can disagree. I consider the tests for a class as a specification of that class, so I tend to put them in. If they're not in an obvious place, the next programmer could wonder "Has this been tested?" and waste time determining what the behavior is supposed to be by writing new tests (this is less likely with simple accessors like this, but suppose there's a calculation or a cache). Finding my tests (and the one for the getMovies method of MovieList) is left as an exercise for the reader.

Notice that, at this point, there's no user interface code in MovieListEditorView at all. The "obvious implementation" is also a "fake it until you make it". This is a consequence of using state testing rather than interaction testing. In the latter, mock objects would have taken the place of the movieList and the view so that the file MovieListEditorView.tcl wouldn't need to be created at all at this point (see [Astels2003] page 220).

The test now passes. But it doesn't really test what I want it to. It's checking that the number of movies in the *View* is correct, but not that they're the same as the movies in the movieList. Now I go back and look things up [Welch2003] to see that the equality operator == is all I need. I was making things too complicated. I change the test

MovieListEditor.test

```
test list {
   MovieListEditor should send movie list from movieList to view.
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   MovieListEditorView view
   MovieListEditor editor \
      [namespace current]::movieList \
      [namespace current]::view

   lappend result [expr { [movieList getMovies] == [view getMovies] }]

   itcl::delete object editor
   itcl::delete object view
   set result
} -cleanup {
   eval $CLEANUP
} -result [list \
   1 \
   ]
```

and it still passes. Run all the tests. They still pass. Take a quick look at the code for duplication. There's some between MovieList and MovieListEditorView, they both have a getMovies method, but MovieListEditorView is still in "Fake it" mode so we'll wait on it.

Test006 - The GUI should have a list box and should display a list of movies

MovieListEditorView should have a list box

After all this time, there's finally going to be a user interface component. This is not unusual, a TDD design usually generates the functionality first. The Humble Dialog Box usually reduces the amount of actual user interface (widget) code actually needed as well.

I want to display the movies in a scrolling list box. Fortunately, [incr Tk] [Smith2000] already has a mega-widget (called Scrolledlistbox) for that so I don't need to reinvent one. I do need to turn my ordinary [incr Tcl] class into a [incr Tk] mega-widget and give it a Scrolledlistbox component. Also fortunately, this is easy. But I need a test for it.

I.MovieListEditorView.01.test

```
test has-scrolledlistbox {
    MovieListEditorView should have a place to display movies.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    lappend result [view component scrolledlistbox]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    \
   ]
```

This test asks the *View* if it has a component called scrolledlistbox. All [incr Tk] mega-widgets can be queried in this manner. The test will necessarily fail because I haven't said what the result should be, but just getting it to compile will take me in the direction I want to go. I often find it helpful to build tests up incrementally from incomplete tests when I'm not certain of the final result. The test fails with

```
bad option "component": should be one of...
```

because view is not a mega-widget and doesn't have a scrolledlistbox component. Making MovieListEditorView a mega-widget is easy; I just require the correct packages and make the class inherit from itk::Widget.

MovieListEditorView.01.tcl

```
package require Itcl
package require Iwidgets

::itcl::class MovieListEditorView {
    inherit ::itk::Widget

    …
}
```

Now the test fails with

```
---- Test setup failed:
bad window path name "view"
```

because mega-widgets, like ordinary Tk widgets, must have names that start with a period. So I change the view in the test to .view (I.MovieListEditorView.02.test).

Now MovieListEditorView is a mega-widget and it *can* have a component but it doesn't, yet.

```
name "scrolledlistbox" is not a component
```

Adding a component is also easy, I adapt the example code on page 350 of [Smith2000] to add a component to the MovieListEditorView constructor, use pack to geometry manage it, and call the [incr Tk] initializer itk_initialize to end the constructor.

MovieListEditorView.02.tcl

```
::itcl::class MovieListEditorView {
    inherit ::itk::Widget

    constructor { args } {
        itk_component add scrolledlistbox {
            ::iwidgets::scrolledlistbox $itk_interior.scrolledlistbox
        }
        pack $itk_component(scrolledlistbox)

        eval itk_initialize $args
    }

    public method setMovies { moviesArg } {
        set movies $moviesArg
    }

    public method getMovies { } {
        return $movies
    }

    private variable movies {}
}
```

This gives me the "good" failure

```
==== has-scrolledlistbox MovieListEditorView should have a place to display movies. FAILED
---- Result was:
.view.scrolledlistbox
---- Result should have been (exact matching):

==== has-scrolledlistbox FAILED
```

It's a good failure because everything compiled and the failure was because I didn't supply a result. I can make it pass by just copying in .view.scrolledlistbox to the *result* section. But that doesn't really test what I want. It tests that I have a component *named* scrolledlistbox, not that I have a component that *is* a Scrolledlistbox. To test that, I can ask the component what type it is like this

```
lappend result [[.view component scrolledlistbox] info class]
```

and then grab the result from the failure message and put it back in the test to get

```
                                        I.MovieListEditorView.03.test
test has-scrolledlistbox {
    MovieListEditorView should have a place to display movies.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    lappend result [[.view component scrolledlistbox] info class]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    ::iwidgets::Scrolledlistbox \
    ]
```

This test passes.

Copying the result from the failure message and blindly pasting it in is considered evil. I agree with that; the key word is blindly. In this case I used a test with a blank result to explore the behavior of a widget set, and then recorded the information I learned in the test. This use of *learning tests* is a good thing. (I've been hearing that phrase lately.)

This test is very specific as to implementation detail. That's not as good a thing, but often the requirements for the user interface do specify a particular type of interaction widget. It's not a problem at this point, but it does make the test more brittle than I'd like. I can't really see a way out at this point so I'll leave it as is.

MovieListEditorView should display a list of movies

There's now a Scrolledlistbox to display the movies, but it's not yet connected with the movies themselves. I could gradually convert the "faked" implementation to a real one by reimplementing the setMovies method to add the movies to the list box and the getMovies method to return the listbox contents. But many Tk widgets (and [incr Tk] mega-widgets) have a nice feature, the ability to coordinate with a variable. I already have the "fake" member movies that passes the tests, I can sync it with the Scrolledlistbox by changing the constructor to

```
constructor { args } {
    itk_component add scrolledlistbox {
        ::iwidgets::scrolledlistbox $itk_interior.scrolledlistbox \
                -listvariable [::itcl::scope movies]
    }
    pack $itk_component(scrolledlistbox)

    eval itk_initialize $args
}
```

The –listvariable option syncs the scrolled list with the variable, and the ::itcl::scope is more namespace magic that packages a class data member so it can be used by a Tk widget as if it were an ordinary (non class) variable.

Watch this, though. This is code that was put in without a failing test. That's always a red flag.

The has-scrolledlistbox test still passes. Now RunAllTests. TestMovieEditor.test now fails. It has two problems. First, since MovieListEditorView is now a mega-widget I need to start the view's name with a period here too. Second, mega-widgets are always created in the global namespace. Therefore I don't want the [namespace current] that I needed for the "fake" view. With these changes, all the tests now pass.

Visual Inspection

OK. There's now a feature with tests. I can create a movie list and display it in a scrolled window; and all the tests pass. But wait a minute, I haven't seen anything yet. I need to actually see a scrolling movie list before I believe it. There's more than one way to do this; I'll do it by writing a sample application. This will let me be sure that I can actually create a "main" program once my tests are complete. Here it is (including the code to invoke the interpreter)

MovieLister.01.tcl

```
#!/bin/sh
# the next line restarts using museTclTk \
exec tclsh "$0" "$@"

set script_dir [file dirname [info script]]

source "$script_dir/Movie.tcl"
source "$script_dir/MovieList.tcl"
source "$script_dir/MovieListEditor.tcl"
source "$script_dir/MovieListEditorView.tcl"

Movie starWars
Movie starTrek
Movie starGate

MovieList movieList
movieList add starWars
movieList add starTrek
movieList add starGate

MovieListEditorView .view


MovieListEditor \
   [namespace current]:: movieList \
   .view

pack .view
```

This will probably look a lot like my final application. I assume the Movie, MovieList, MovieListEditor, and MovieListEditorView will be in the same directory as MovieLister.tcl and source them in (I may end up putting them in a package or a more complicated directory structure, or may not). I create three movies (this won't be in the production version) and add them to the movieList. To me, the really nice thing about [incr Tk] is that it is easy to compose widgets into mega-widgets which then behave just like widgets. That means I can just pack the .view to make it visible and see

I expected to see this since it's what Dave saw at this point, so it's not surprising to me that this isn't what I want. I want to see the movie names ("Star Wars", "Star Trek", "Stargate") and not the variable names (starWars, starTrek, starGate). Of course, to do that I really should have included those names in the code somewhere.

So, movies should have names. Make a test.

Movie.test

```
test movie-name {
    starWars should have name "Star Wars"
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    Movie starWars "Star Wars"
    lappend result [starWars getName]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    "Star Wars" \
    ]
```

and make it pass (obvious implementation)

Movie.tcl

```
package require Itcl

::itcl::class Movie {

    constructor { movieName } {
        set name $movieName
    }

    public method getName { } {
        return $name
    }

    private variable name
}
```

Now we need to make the other tests pass by changing them to always create a movie with a name. Dave Astels has a good discussion of why just keeping the no argument constructor in Movie and papering over the problem with the tests leads to code debt. You should check it out ([Astels2003] page 224).

Check for duplication. There is some, but it all seems related to the minimum amount of code needed to implement setters and getters. If you see a way to avoid it other than public data members, give me a call.

Now change the movie lister

MovieLister.tcl

```
...
Movie starWars "Star Wars"
Movie starTrek "Star Trek"
Movie starGate "Stargate"

MovieList movieList
movieList add starWars
movieList add starTrek
movieList add starGate

MovieListEditorView .view


MovieListEditor \
   [namespace current]:: movieList \
   .view

pack .view
```

Run the changed movie lister and see…exactly the same thing. I haven't told MovieListEditorView to use the names yet. Dave does it by adding a toString() method to Movie which his movie objects implicitly call. I don't want to do it that way for two reasons:

1. Movie is a domain level class; I prefer to keep it out of MovieListEditorView and let MovieListEditor mediate.

2. Tcl doesn't use toString() the way Java does so it wouldn't work anyway.

I can tell MovieListEditor to send MovieListEditorView the names very simply. I just put a conversion loop in the MovieListEditor where it obviously belongs.

MovieListEditor.01.tcl

```
package require Itcl

::itcl::class MovieListEditor {

   constructor { movieList view } {
      set movies [$movieList getMovies]

      foreach movie $movies {
         lappend titles [$movie getName]
      }
      $view setMovies $titles
   }

   private variable movies {}
}
```

I fire up the movie lister and see

so I'm done.

Why I Shouldn't Add Code Without Tests

Or I will be as soon as soon as I run all the tests again. No problem since I didn't do anything major and I can *see* in the movie lister that everything's correct. So, RunAllTests.tcl and

```
==== list MovieListEditor should send movie list from movieList to view. FAILED
---- Test generated error; Return code was: 1
---- Return code should have been one of: 0 2
---- errorInfo: invalid command name "starTrek"
    while executing
"$movie getName"
    while constructing object "::MovieListEditor::test::editor" in ::MovieListEditor::constructor (body line 5)
    invoked from within
"MovieListEditor editor  [namespace current]::movieList  .view"
    ("uplevel" body line 4)
    invoked from within
"uplevel 1 $script"
---- errorCode: NONE
==== list FAILED

MovieListEditor.test:     Total  1     Passed  0     Skipped 0     Failed  1
```

Oh.

```
errorInfo: invalid command name "starTrek"
```

Namespace issues again (I can recognize the symptoms even if I can't remember to think ahead and avoid the problem. Why didn't MovieLister.tcl have this problem? Because all the objects there were created in global scope. Change the test (one object at a time [l.MovieListEditor.03.test - l.MovieListEditor.05.test]) to read

I.MovieListEditor.05.test

```
package require tcltest

tcltest::configure -verbose pse

source "./MovieList.tcl"
source "./Movie.tcl"
source "./MovieListEditor.tcl"
source "./MovieListEditorView.tcl"

namespace eval ::MovieListEditor::test {
namespace import ::tcltest::*

variable SETUP {
   MovieList movieList
   movieList add [namespace current]::[Movie starWars "Star Wars"]
   movieList add [namespace current]::[Movie starTrek "Star Trek"]
   movieList add [namespace current]::[Movie starGate "Stargate"]
}
variable CLEANUP {
   itcl::delete object movieList
   itcl::delete object starWars
   itcl::delete object starTrek
   itcl::delete object starGate
}

test list {
   MovieListEditor should send movie list from movieList to view.
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   MovieListEditorView .view
   MovieListEditor editor \
      [namespace current]::movieList \
      .view

   lappend result [expr { [movieList getMovies] == [.view getMovies] }]

   itcl::delete object editor
   itcl::delete object .view
   set result
} -cleanup {
   eval $CLEANUP
} -result [list \
   1 \
   ]
```

and I get

```
==== list MovieListEditor should send movie list from movieList to view. FAILED
---- Result was:
0
---- Result should have been (exact matching):
1
==== list FAILED

MovieListEditor.test:     Total   1      Passed  0      Skipped 0
```

In other words

I.MovieListEditor.05.test

```
…-
   lappend result [expr { [movieList getMovies] == [.view getMovies] }]
…
```

isn't true now. This test was checking to see the movies in movieList (which contained movie objects) was the same as the movies in .view (which should have contained movie names). If I had paid more attention to the test I would have realized there would be a problem even without my graphical movie lister or reading ahead in Dave's book.

To be fair, putting the comparison inside the test makes it difficult to see what's being compared. All you see is the 1 or 0 for match or not. There's a better way to do it. See the Appendix.

OK. I can make that test pass by just copying the loop from MovieListEditor

MovieListEditor.01.tcl

```
…
      foreach movie $movies {
         lappend titles [$movie getName]
      }
…
```

into the test and comparing the .view's contents against the list of movie names (titles). But that's worse than ugly. It's duplication.

So, what do I want to do? Well, if I had started by thinking about the test like I was supposed to, I would probably have made it look like this.

MovieListEditor.test

```
…
    lappend result \
      [expr { [movieList getMovieNames] == [.view getMovieNames] }]
…
```

Same format, just different names, different meaning, different intention. On the MovieListEditorView side, the change is easy; just change the names

MovieListEditorView.tcl

```
::itcl::class MovieListEditorView {
   inherit ::itk::Widget
…
   public method setMovieNames { movieNamesArg } {
      set movieNames $movieNamesArg
   }

   public method getMovieNames { } {
      return $movieNames
   }

   private variable movieNames {}
}
```

Rename Method [RenameMethod] is a simple refactoring and some development environments have tools that do it automatically, [usually] ensuring correctness and changing all uses, including the tests. I'm not aware of any for Tcl, so I do it manually and run the tests frequently to be sure everything got changed correctly.

I also add in the new method to MovieList

MovieList.tcl

```
...
   public method getMovieNames { } {
      set names {}
      foreach movie $movies {
         lappend names [$movie getName]
      }
      return $names
   }
...
```

and change the MovieListEditor's constructor to

MovieListEditor.tcl

```
package require Itcl

::itcl::class MovieListEditor {

   constructor { movieList view } {
      $view setMovieNames [$movieList getMovieNames]
   }
}
```

The movie lister still looks good and there's no obvious refactoring needed. The first story is really done and I have the tests to prove it. It took a lot longer to describe than to do.

Test007 – Add a movie

The second story is to be able to add a movie to the movie list (they have to get in somehow). The implementation starts with the *Presenter* MovieListEditor. When the logical layer is asked to add a movie it needs to do two things:

1. Request the movie name from .view.

2. Update the movie list

The descriptions will be a lot shorter now. Here's the test for the add method

I.MovieListEditor.01.test

```
test adding {
    MovieListEditor should add a movie to movieList from view.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    MovieListEditorView .view
    MovieListEditor editor \
        [namespace current]::movieList \
        .view

    set newMovieName "Lost in Space"

    set movieNamesWithAddition [movieList getMovieNames]
    lappend movieNamesWithAddition $newMovieName

    .view setNewName $newMovieName

    editor add

    lappend result \
        [expr { [movieList getMovieNames] == $movieNamesWithAddition }]

    itcl::delete object editor
    itcl::delete object .view
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    1 \
    ]
```

This test requires the new method add in MovieListEditor to provide the functionality and the new method setNewName in MovieListEditorView to simulate the user entering in a new movie name somehow (that "somehow" will be seen in the next test). The setNewName method is not strictly required for the functionality; it's there to allow the design to be tested. There are people who object to writing code that's needed "only for testing". I'm not one of them. Right now the MovieListEditorView is in "fake it until you make it" mode for this feature anyway, so all the method does is set a member variable (see MovieListEditorView.test and MovieListEditorView.tcl).

The MovieListEditor add method is the interesting one. It needs to ask the MovieListEditorView for the new movie name (to do this it needs to remember the view that was passed to the constructor), create a new Movie with that name, and ask the movieList (which it must also now remember) to add the movie. This leads to the code

MovieListEditor.01.tcl

```
package require Itcl

::itcl::class MovieListEditor {

   constructor { movieListArg viewArg } {
      set movieList $movieListArg
      set view $viewArg
      $view setMovieNames [$movieList getMovieNames]
   }

   public method add { } {
      $movieList add [namespace current]::[Movie #auto [$view getNewName]]
      $view setMovieNames [$movieList getMovieNames]
   }

   private variable movieList {}
   private variable view {}
}
```

(where #auto just generates a unique object name for the new movie) and

MovieList.tcl

```
package require Itcl

::itcl::class MovieList {
…
   public method add { movie } {
      lappend movies $movie
   }
…
```

All the tests pass but there's a lot of duplication, so it's refactoring time. The last line of the MovieListEditor constructor and the add method are identical and can be extracted into its own method updateView

MovieListEditor.tcl

```
…
   private method updateView { } {
      $view setMovieNames [$movieList getMovieNames]
   }
…
```

and the common fixture of the list and adding tests is put into SETUP/CLEANUP (MovieListEditor.test).

Test008 – GUI should support add movie functionality

Now, analogous to **Test006**, MovieListEditorView is enhanced to provide the real GUI for the add movie functionality (the "make it" part of "fake it until you make it"). This will be done in sections.

MovieListEditorView should have
  a) an entry field for New Movie Name, and
  b) a push button for add

This is straightforward. The has-scrolledlistbox test is changed to has-components

MovieListEditorView.test

```
test has-components {
   MovieListEditorView should have a place to display movies.
   MovieListEditorView should have a place for new movie names.
   MovieListEditorView should have an Add button.
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   lappend result [[.view component scrolledlistbox] info class]
   lappend result [[.view component moviefield] info class]
   lappend result [[.view component addbutton] info class]
   set result
} -cleanup {
   eval $CLEANUP
} -result [list \
   ::iwidgets::Scrolledlistbox \
   ::iwidgets::Entryfield \
   ::iwidgets::Pushbutton \
   ]
```

(you can use separate tests if you prefer) which is made to pass by augmenting the constructor

MovieListEditorView.01.tcl

```
...
::itcl::class MovieListEditorView {
   inherit ::itk::Widget

   constructor { args } {
      itk_component add scrolledlistbox {
         ::iwidgets::scrolledlistbox $itk_interior.scrolledlistbox \
            -listvariable [::itcl::scope movieNames]
      }
      pack $itk_component(scrolledlistbox)
      itk_component add moviefield {
         ::iwidgets::entryfield $itk_interior.moviefield \
            -labeltext "Movie Name:" \
            -textvariable [::itcl::scope newMovieName]
      }
      pack $itk_component(moviefield)
      itk_component add addbutton {
         ::iwidgets::pushbutton $itk_interior.addbutton \
            -text "Add"
      }
      pack $itk_component(addbutton)

      eval itk_initialize $args
   }
...
```

Of course, the code is added in two steps, one for moviefield and one for addbutton (I said the descriptions would be shorter).

The Add button should invoke MovieListEditor's add method

The moviefield just connects to the "fake it" member newMovieName and should work automatically. The behavior of the Add button will have to be specified.

I want MovieListEditorView to be as independent of the domain objects and logic as possible. Dave Astels let his MovieListEditorView contain a reference to the editor and hard coded in a call to its add method. In Java, that's the simplest thing that could possibly work. Tcl is a more dynamic language and it supports a pattern known as "Pluggable Behavior" [Beck1996] that's easy in Tcl but harder in Java. I will let the MovieListEditor tell the Add button what to do.

Let's start with a simpler behavior. Write a test to tell the Add button to just set a pushed flag to 1.

MovieListEditorView.test

```
test has-behavior {
    MovieListEditorView should  be able to set addbutton behavior.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    set pushed 0
    lappend result $pushed
    .view setAddButtonBehavior [list set [namespace current]::pushed 1 ]
    [.view component addbutton] invoke
    lappend result $pushed
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    0 \
    1 \
    ]
```

This test calls setAddButtonBehavior to tell the Add button to set the variable flag pushed (in the test's namespace, not the *View's*) to 1 and checks that the flag was 0 before the button was pushed and 1 afterwards. The method invoke is provided by [incr Tk] for a Pushbutton mega-widget. The method setAddButtonBehavior is surprisingly simple

MovieListEditorView.02.tcl

```
    public method setAddButtonBehavior { behavior } {
        $itk_component(addbutton) configure -command $behavior
    }
```

Now that I know I can do it, I want the MovieListEditor to give the Add button editor add functionality. I already have a test for the functionality. If I comment out the add method call and put in a call to invoke, that should test that the button works.

MovieListEditor.test

```
…

    #editor add
    [.view component addbutton] invoke
…
```

I left the commented line in to remind me what invoke is supposed to do, and to make it easier to have the test bypass .view later if necessary for investigation or debugging. A purist might write two tests; I'm not a purist.

I make it pass with

MovieListEditor.tcl

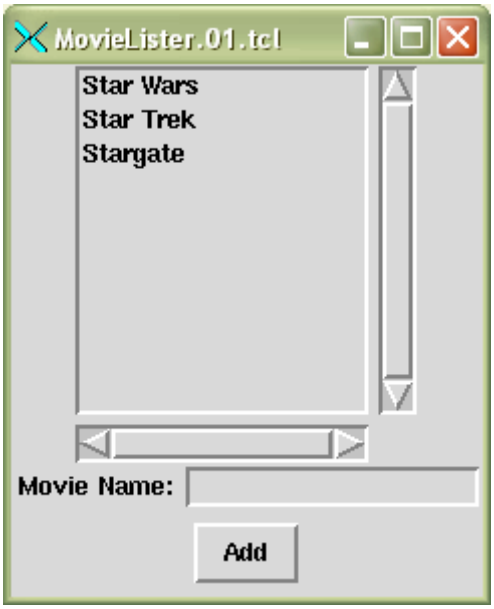
```
...
   constructor { movieListArg viewArg } {
      set movieList $movieListArg
      set view $viewArg

      $view setAddButtonBehavior [itcl::code $this add]
      updateView
   }
...
```

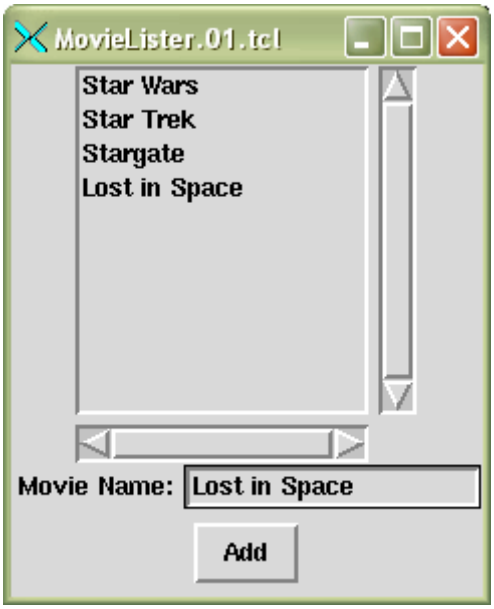
The itcl::code is more [incr Tcl] magic like itcl::scope. In the same way that itcl::scope packages class data members for use by widgets, itcl::code does the same for class methods.

Visual Inspection

All the tests pass, but does it work? Start up my trusty movie lister and I see

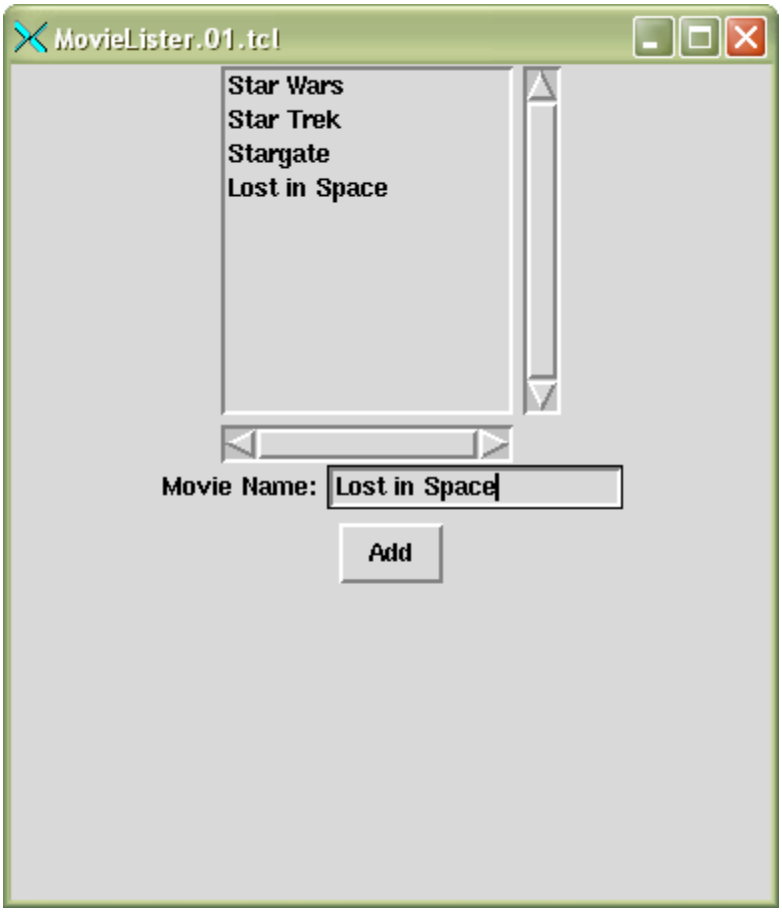


Type in a movie name and push the Add button and I get



The functionality worked, right out of the box.

Does this mean that exploratory testing is a thing of the past? Afraid not. Let me try to do something else a user might do, like make it bigger. Now I see



Not quite what I'd want to happen. I'd like the Scrolledlistbox to fill up all that extra space. I need to change the geometry management of the scrolledlistbox component to fill the space available (expanding in both directions),

MovieListEditorView.03.tcl

```
pack $itk_component(scrolledlistbox) -fill both -expand true
```

and the moviefield to fill the space available in the horizontal (x) direction,

MovieListEditorView.03.tcl

```
pack $itk_component(moviefield) -fill x
```

The pack in the movie list also needs to expand (MovieLister.tcl).

Now when I resize I get

which is how I want the space distributed.

This illustrates some of the tradeoffs in deciding what to test. I find there are three categories:

1. Things I can test, and do.

2. Things I can test, and don't bother.

3. Things I can't test.

Most of this paper is about the first category. An example of the second category would be that the text on the Add button says "Add". Usually it's not worth the trouble, but if I move away from hard-coded strings to using resources for internationalization I might wish that test were there. The resize behavior is in the third category. I don't know how to mechanically test that things "look nice". Sometimes things are in the third category because the GUI toolkit makes it hard to get information or select an item or push a button programmatically. I really like Tk widgets because I don't have that problem often.

A sample application is not the only way to visually examine your code. Phlip [Phlip] suggests the use of a "reveal" command that can be placed in the test. The command would normally be commented out, but when executed it would bring up the interface at that point and allow you to interact with it. In Tk, a reveal command can be as simple as a pack followed by a vwait [Welch2003]. If I had used one in the test, I wouldn't have had to create and explicitly load the movies in MovieLister.tcl and could have seen the visual state directly.

```
proc reveal { view } {
   set done 0
   wm protocol . WM_DELETE_WINDOW [list set done 1]
   pack $view -expand true -fill both
   vwait done
}
```

Refactoring

Time to refactor. The main thing that jumps out at me here is the constructor of MovieListEditorView. Not duplication, but Kent Beck's rule three (express all the ideas the author wants to express). Seems to me it's a lot clearer this way

MovieListEditorView.tcl

```
::itcl::class MovieListEditorView {
   inherit ::itk::Widget

   constructor { args } {
      buildScrolledlistbox
      buildMoviefield
      buildAddbutton

      eval itk_initialize $args
   }
...
}
```

Test009 – Rename movie should change movie name

Add a test

I.Movie.01.test

```
test renaming {
    starWars should have name "Star Trek" after renaming
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    Movie starWars "Star Wars"
    starWars rename "Star Trek"
    lappend result [starWars getName]
    itcl::delete object starWars
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    "Star Trek" \
    ]
```

Imagine the small steps, test failed for no rename method, test failed for dummy rename method that does nothing, hard-coded return value (I actually skipped that one), to get to

Movie.01.tcl

```
package require Itcl

::itcl::class Movie {

    constructor { movieName } {
        set name $movieName
    }

    public method getName { } {
        return $name
    }

    public method rename { newName } {
        set name $newName
    }

    private variable name
}
```

which passes.

Obvious duplication here, so refactor to

Movie.tcl

```
package require Itcl

::itcl::class Movie {

    constructor { movieName } {
        $this rename $movieName
    }

    public method getName { } {
        return $name
    }

    public method rename { newName } {
        set name $newName
    }

    private variable name
}
```

There's also a common fixture in the movie tests that needs refactoring (see Movie.test).

Test010 – Movie shouldn't be constructed with null name

This test illustrates one way to handle testing exceptions with tcltest. Since it doesn't use any of the existing fixtures, it gets its own file.

NullMovie.test

```
test construct-null-name {
    null name should throw exception
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    catch { Movie nullMovie {} } result
    set result
} -cleanup {
    eval $CLEANUP
} -result {null Movie name}
```

The catch command places the exception (if one is thrown) text in result. To make it pass, the constructor needs to do a check and throw an error. After the last refactoring the constructor just calls rename, so that's where I put the check.

Movie.tcl

```
public method rename { newName } {
  if { $newName eq {} } {
      error "null Movie name"
  } else {
      set name $newName
  }
}
```

Technically I should still put the check in the constructor since it may be that construction and rename could have different behaviors with null names. But I know what Dave's next three tests say. The YAGNI (you ain't gonna need it) principle says I should ignore that knowledge in the current implementation. So sue me.

Test011 – Movie shouldn't be constructed with empty name

In Java the null string and the empty string are distinct concepts. That's not really the case in Tcl. I can keep the design parallel with Dave's and write the parallel tests

NullEmptyMovie.test

```
variable exception "null or empty Movie name"

test construct-null-name {
   null name should throw exception
} -setup  {
   eval $SETUP
} -body  {
   set result [list]
   catch { Movie nullMovie {} } result
   set result
} -cleanup {
   eval $CLEANUP
} -result $exception
```

and

NullEmptyMovie.test

```
test construct-empty-name {
    empty name should throw exception
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    catch { Movie emptyMovie "" } result
    set result
} -cleanup {
    eval $CLEANUP
} -result $exception
```

(note the change in test file name). But all that's needed to pass both is to change the exception string (Movie.tcl).

Test012 – Movie shouldn't be renamed to null name

Test013 – Movie shouldn't be renamed to empty name

The tests here are obvious (they're in Movie.test) and they pass without any code changes needed.

Test014 – Selecting a movie should send that movie to the view

Here's a more complicated user interface behavior. When you select a movie in the scrolled list, that movie name should appear in the Movie Name entry field. This behavior only concerns the user interface so it can be encapsulated in MovieListEditorView. Right?

That's the traditional way of doing it and it leads to complicated and hard to test code in the user interface. That way leads to duplicated GUI code when you want your web page, rich client, and PDA interfaces to behave identically.

The dialog box needs to be kept humble. So the GUI behavior will be split into two parts. The *Presenter* (MovieListEditor) will implement the actual behavior with a select method (this test) and MovieListEditorView will simply invoke that method when a movie is selected (next test)

First step, write a test to select movie list item 1, the second item (Tcl, like C, starts counting at 0), and check that getNewName returns the selected choice.

I.01.MovieListEditor.test.

```
…
variable SETUP {
    MovieList movieList
    movieList add [namespace current]::[Movie starWars $starWarsName]
    movieList add [namespace current]::[Movie starTrek $starTrekName]
    movieList add [namespace current]::[Movie starGate $starGateName]
    MovieListEditorView .view
    MovieListEditor editor \
        [namespace current]::movieList \
        .view
}
…
variable starWarsName "Star Wars"
variable starTrekName "Star Trek"
variable starGateName "Stargate"
…
 test selecting {
    Selecting a movie should send that movie to the view.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]

    editor select 1
    lappend result [.view getNewName]

    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    $starTrekName \
    ]
```

Going through the small steps (I'm not going to list them, see MovieListEditor.01.tcl and MovieListEditor.02.tcl) leads to the code

MovieListEditor.tcl

```
    public method select { index } {
        set movieName [lindex [$movieList getMovieNames] $index]
        $view setNewName $movieName
    }
```

The select method could be done as a one liner, but I prefer to use an explaining variable (see [Fowler1999] page 124). And, what do you know, that setNewName method "that's needed only for testing" turns out to be useful in production code. Funny how often that happens.

One possible issue. When the behavior is encapsulated in MovieListEditorView, I know that it's called only by the scrolledlistbox and the select index argument is always valid. A public select method of MovieListEditor doesn't have that assurance. So I augment the test to see what happens with other valid and invalid arguments

MovieListEditor.test

```
test selecting {
    Selecting a movie should send that movie to the view.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]

    editor select 0
    lappend result [.view getNewName]

    editor select 1
    lappend result [.view getNewName]

    editor select 6
    lappend result [.view getNewName]

    editor select -1
    lappend result [.view getNewName]

    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    $starWarsName \
    $starTrekName \
    {} \
    {} \
    ]
```

Invalid arguments just give null names. Since previous tests catch that condition, I'm OK with this. I like putting in the additional conditions so that the tests act as a specification, but there's a fine line between sufficient testing and overkill. I'm not always on the correct side of it.

Test015 – Selecting a movie from the scrolledlistbox should send that movie to the view

The behavior works and is tested. Now it needs to be invoked from the view. This is similar to the case with the Add button; I need to comment out the call to editor select and replace it with a scrolledlistbox selection.

MovieListEditor.test

```
test selecting {
…
    #editor select 1
    set scrolledlistbox [.view component scrolledlistbox]
    $scrolledlistbox selection set 1
    eval [$scrolledlistbox cget -selectioncommand]
    lappend result [.view getNewName]
…
} -result [list \
    $starWarsName \
    $starTrekName \
    {} \
    {} \
    ]
```

I've learned from previous experience that setting a scrolledlistbox (or an ordinary Tk listbox) selection does not invoke the selectioncommand (Why? I don't know. But it doesn't.) so I need to cget the selection command and evaluate it explicitly.

MovieListEditorView should still be independent of the domain so I want to use "Pluggable Behavior" [Beck1996] again to make the test pass. Therefore I need a setMovieSelectBehavior method analogous to the setAddButtonBehavior one. But there's a complication. I'm not just invoking a method; I need to specify an argument telling me which element was selected. Fortunately, this problem has come up before. When you bind behavior to a mouse button, for example, it's useful to be able to specify which button was pressed (not everyone uses a Mac) and where it is. The way this is done is through magic event keywords %b, %x, and %y (see [Welch2003] page 449).

There's no magic keyword for the scrolledlistbox selection, but that's easily remedied. Again, first write a test to demonstrate the required behavior with a selected flag (analogous to the pushed flag in the button test). The magic keyword is designated by %currentSelection%.

```
MovieListEditorView.test
test has-select-behavior {
    MovieListEditorView should  be able to set movielist select  behavior.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]

    set selected 0
    lappend result $selected

    .view setMovieNames { one two three four }
    .view setMovieSelectBehavior \
     [list set [namespace current]::selected %currentSelection% ]

    [.view component scrolledlistbox] selection set 1
    eval [[.view component scrolledlistbox] cget -selectioncommand]

    lappend result $selected
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    0 \
    1 \
    ]
```

and now make it pass with

MovieListEditorView.tcl

```
::itcl::class MovieListEditorView {
   inherit ::itk::Widget
…
   public method setMovieSelectBehavior { behavior } {
      set selectionBehavior $behavior

   private method buildScrolledlistbox { } {
      itk_component add scrolledlistbox {
         ::iwidgets::scrolledlistbox $itk_interior.scrolledlistbox \
               -selectioncommand [::itcl::code $this doSelection] \
               -listvariable [::itcl::scope movieNames]
      }
      pack $itk_component(scrolledlistbox) -fill both -expand true
   }
…
   private method doSelection { } {
      set currentSelection [$itk_component(scrolledlistbox) curselection]
      regsub -all \
         "%currentSelection%" $selectionBehavior $currentSelection \
               behavior
      eval $behavior
   }
…
   private variable selectionBehavior {}
}
```

The method setMovieSelectBehavior just saves the behavior in the selectionBehavior data member. The scrolledlistbox then always executes the doSelection method when the selection changes. That method figures out the current selection, replaces all instances of %currentSelection% in selectionBehavior with the current selection, and then evaluates the result. A long way to go, but a great illustration of David Wheeler's immortal advice (as quoted in Butler Lampson's Turing Award lecture), *Any problem in computer science can be solved with another layer of indirection*.

After all that work, actually specifying the behavior we need is simple.

MovieListEditor.tcl

```
::itcl::class MovieListEditor {

    constructor { movieListArg viewArg } {
…
        $view setMovieSelectBehavior \
            [itcl::code $this select %currentSelection%]
…
    }
```

and all the tests pass. Just for luck I bring up the movie lister and select a movie. The Movie Name field is automatically filled in.

Test016 – Update should rename movie

Next story. It's time to add an Update feature. It's down to a rhythm now

1. Add the feature to the *Presenter* MovieListEditor, faking the GUI features (this test).
2. Add in the new GUI elements, replacing the fakery (next test).
3. Change the MovieListEditor test to invoke the feature through the GUI (also next test).

Here's the test

MovieListEditor.test

```
…
variable newStarTrekName "Star Trek I"
…
test updating {
    Updating should rename a movie
} -setup  {
    eval $SETUP
} -body  {
    set result [list]

    set movieNamesWithRename \
        [lreplace [movieList getMovieNames] 1 1 $newStarTrekName]

    editor select 1
    .view setNewName $newStarTrekName

    editor update

    lappend result \
        [expr { [movieList getMovieNames] == $movieNamesWithRename }]

    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    1 \
    ]
```

In order to get this to pass, the update method needs to know which movie was last selected. Have the select method save it in a new private variable.

MovieListEditor.tcl

```
    public method update { } {
        if { $selectedMovie != {} } {
            $selectedMovie rename [$view getNewName]
            updateView
        }
    }

    public method select { index } {
        set selectedMovie [lindex [$movieList getMovies] $index]
        set movieName [lindex [$movieList getMovieNames] $index]
        $view setNewName $movieName
    }
...
    private variable selectedMovie {}
}
```

On to the GUI.

Test017 – GUI should support update/rename movie functionality

Like **Test008**, enhancing MovieListEditorView to provide the update movie functionality will be done in sections.

MovieListEditorView should use a buttonbox

I could simply add in another Pushbutton mega-widget for the Update button, but then I'd need to be sure both buttons were the same size and evenly spaced. [incr Tk] has a mega-widget that will do that for me, the buttonbox.

I could go ahead and start adding the Update button but, when adding new functionality, a useful technique is to refactor first to make the functionality easy to add. So first get the Add button and all its tests to work with a buttonbox.

It would have been nice if the tests were independent of whether or not I use a buttonbox. Unfortunately, the buttonbox is a single component of the mega-widget and I can't use the component query to access the individual buttonbox pushbuttons. So the has-components test becomes

I.MovieListEditorView.01.test

```
test has-components {
    MovieListEditorView should have a place to display movies.
    MovieListEditorView should have a place for new movie names.
    MovieListEditorView should have an Add button.
} -setup {
    eval $SETUP
} -body {
    set result [list]
    lappend result [[.view component scrolledlistbox] info class]
    lappend result [[.view component moviefield] info class]
    lappend result [[.view component buttonbox] info class]
    lappend result [[.view component buttonbox] buttoncget addbutton -text]
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    ::iwidgets::Scrolledlistbox \
    ::iwidgets::Entryfield \
    ::iwidgets::Buttonbox \
    Add \
    ]
```

I check that there's a buttonbox and that the buttonbox has an addbutton component (labeled Add for a bit of overkill). To get the test to pass, I need to change buildAddbutton.

MovieListEditorView.01.tcl

```
private method buildAddbutton { } {
        itk_component add buttonbox {
            ::iwidgets::buttonbox $itk_interior.buttonbox
        }
        $itk_component(buttonbox) add addbutton -text "Add"
        pack $itk_component(buttonbox)
    }
```

Run the test and discover that setAddButtonBehavior no longer compiles. I comment it out (and all its calls) so that I can worry about behavior after the button is in place. I use a unique comment indicator (#???) so I can find them again when I need them. The test has-components now passes but has-addbutton-behavior now doesn't compile. Buttons in a buttonbox must be invoked with

I.MovieListEditorView.01.test

```
[.view component buttonbox] invoke addbutton
```

Now it compiles, but fails since the behavior is still commented out. Re-implement setAddButtonBehavior.

MovieListEditorView.01.tcl

```
public method setAddButtonBehavior { behavior } {
    $itk_component(buttonbox) buttonconfigure addbutton -command $behavior
}
```

and restore the calls. The MovieListEditorView.test tests now pass. Run all the tests. The adding test in MovieListEditor.test still has the old invoke. Change that and all the tests pass. I'm ready for the Update button. But use the movie lister to look at the application and try out the Add button anyway.

By the way, notice that while I don't mind having the tests know about the GUI internals of the *View*, when it comes to production code like MovieListEditor.tcl I always define a *View* interface method like setAddButtonBehavior to insulate the *Presenter* from Tk details.

MovieListEditorView should have a push button for update

Now adding the Update button should be easy. First add to the test.

MovieListEditorView.test

```
test has-components {
…
    MovieListEditorView should have an Update button.
…
    lappend result [[.view component buttonbox] buttoncget updatebutton -text]
…
    Update \
    ]
```

Then add to the implementation

MovieListEditorView.02.tcl

```
private method buildButtons { } {
…
      $itk_component(buttonbox) add updatebutton -text "Update"
…
   }
```

and the test passes. The "Rename Method" refactoring to the more expressive name (buildAddButton doesn't cut it anymore) actually came after the test passed with the old name. While I'm refactoring for more expression, refactor setAddButtonBehavior to

MovieListEditorView.02.tcl

```
public method setButtonBehavior { button behavior } {
   $itk_component(buttonbox) buttonconfigure $button -command $behavior
}
```

I really need to work on my YAGNI. But no, I'm refactoring to make adding the new Update button behavior easier so it's OK.

The Update button should invoke MovieListEditor's update method

This feature is following the pattern nicely. Now change MovieListEditor.test

MovieListEditor.test

```
…
   #editor update
   [.view component buttonbox] invoke updatebutton
…
```

The test fails until the behavior is set in the MovieListEditor's constructor

MovieListEditor.tcl

```
::itcl::class MovieListEditor {

   constructor { movieListArg viewArg } {
      set movieList $movieListArg
      set view $viewArg

      $view setButtonBehavior addbutton [itcl::code $this add]
      $view setButtonBehavior updatebutton [itcl::code $this update]
      $view setMovieSelectBehavior \
         [itcl::code $this select %currentSelection%]

      updateView
   }
```

All the tests pass. Time to refactor.

Refactoring

There are two main areas of duplication. One is the duplicated call to lindex in

MovieListEditor.01.tcl

```
   public method select { index } {
      set selectedMovie [lindex [$movieList getMovies] $index]
      set movieName [lindex [$movieList getMovieNames] $index]
      $view setNewName $movieName
   }
```

This was introduced in **Test016**. I got caught up in the rhythm and didn't do the refactoring I needed. Bad.

The second lindex call actually gets the name from the movie in the first lindex call; it could be replaced by a call to selectedMovie getName. But this causes a test failure since selectedMovie can be null. So it needs a check.

Even the first lindex call can be improved. A classic [code] smell is a method that seems more interested in a class other than the one it is in. This is called Feature Envy [FeatureEnvy]. The code seems to be calling out for a getMovie method for MovieList (or maybe I just saw it in Dave's code, but I think I got there independently). After adding the method (and the test), I have

MovieListEditor.tcl

```
public method select { index } {
    set movieName {}
    set selectedMovie [$movieList getMovie $index]
    if { $selectedMovie != {} } {
        set movieName [$selectedMovie getName]
    }
    $view setNewName $movieName
}
```

The code is definitely clearer. A good clarity test is to read the code aloud; the lindex calls were definitely not intention revealing. But now it's longer, and the check for null is duplicated. Maybe I need to introduce a null object [NullObject]. I think I'll wait until there's a third instance of the check before I do that. I don't want to just polish the polish.

Visual Inspection

Look at it with the movie lister.

1. Update button works
2. Add button works
3. Resize seems to work

a. Scrolledlistbox resizes to fill the space

b. Movie Name entry field expands horizontally but not vertically

c. Add and Update buttons are the same size and centered

4. Wrong! The title bar is the name of the file, MovieLister.tcl.

In Dave Astels's book, the title was "Movie List". It's easy enough to fix; just a one liner.

MovieListEditorView.tcl

```
::itcl::class MovieListEditorView {
   inherit ::itk::Widget

   constructor { args } {
      …
      wm title . "Movie List"
   }
   …
}
```

and it looks fine.

Maybe I should have had a test.

Acknowledgements

Beyond Kent Beck's and Dave's Astels's obvious influence, there were three main influences on this paper. Philip Craig Plumlee AKA Phlip started the TFUI Yahoo group [Phlip], Dossy Shiobara AKA Dossy published a Tcl version of the classic bowling score problem that enabled me to make sense of tcltest [Dossy], and Ron Jeffries showed me how to test the GUI "wiring" in [Jeffries2004].

Thanks also to Don Porter and Michael A. Cleverly for the help detailed in the Appendix. And to Larry Virden and Barton Browning for reviewing a draft.

None of these people are responsible for the way I've misunderstood and misinterpreted their work. Any errors are mine.

References

[Astels2003]
David Astels, *Test-Driven Development: A Practical Guide*, Prentice Hall PTR (2003)

[Beck1996]
Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall PTR (1996)

[Beck2003]
Kent Beck, *Test-Driven Development: By Example*, Addison-Wesley (2003)

[Dossy]
Dossy Shiobara, *what's the noun form of "bowling scoring"?*, http://dossy.org/archives/000031.html

[Feathers2002]
Michael Feathers, *The Humble Dialog Box*, http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf

[FeatureEnvy]
Feature Envy, http://c2.com/cgi/wiki?FeatureEnvy

[Fowler1999]
Martin Fowler et al, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999) (see also http://www.refactoring.com)

[Fowler2004]
Martin Fowler, *Model View Presenter*, http://www.martinfowler.com/eaaDev/ModelViewPresenter.html

[Jeffries2004]
Ron Jeffries, *Extreme Programming Adventures in C#*, Microsoft Press (2004)

[Martin1994]
Robert C. Martin and Robert S. Koss, *An Extreme Programming Episode*, http://www.objectmentor.com/resources/articles/xpepisode.htm

[NullObject]
Refactoring: Introduce Null Object, http://www.refactoring.com/catalog/introduceNullObject.html

[Phlip]
Philip Craig Plumlee, *Test First User Interfaces Yahoo Group*, http://groups.yahoo.com/group/TestFirstUserInterfaces/

[Plumlee2006]
Phlip Plumlee, *Test First User Interfaces : Developing Agile GUI Code*, Addison-Wesley Professional (2006)

[RenameMethod]
Refactoring: Rename Method, http://www.refactoring.com/catalog/renameMethod.html

[Smith2000]
Chad Smith, *[incr Tcl/Tk] from the Ground Up*, Mcgraw-Hill Osborne Media, (2000)

[tcltest]
tcltest - Test harness support code and utilities, http://tcl.tk/man/tcl8.4/TclCmd/tcltest.htm

[usenet]
seeking tcltest help, http://groups.google.com/group/comp.lang.tcl/browse_frm/thread/48a6de00a18f8465/b7c88d56a91e22ba?q=tcltest+help&rnum=1#b7c88d56a91e22ba

[Welch2003]
Brent Welch et al, *Practical Programming in Tcl and Tk*, Prentice Hall PTR; 4th edition (2003)

Appendix

In **Test006** I complained that comparing the editor and .view movie lists inside the tests made it difficult to see what the actual lists were. I felt I was forced to do it this way because of the Tcl evaluation rules which computed the argument for *result* before computing the *body* section of the test. I had a question about that placed on the comp.lang.tcl USENET group [usenet] and received excellent help from Don Porter and Michael A. Cleverly.

The custom comparison feature of tcltest [tcltest] can be used to solve the problem. Remember that the test

```
                                        l.MovieListEditor.01.test
test list {
    MovieListEditor should send movie list from movieList to view.
} -setup  {
    eval $SETUP
} -body  {
    set result [list]
    MovieListEditorView .view
    MovieListEditor editor \
        [namespace current]::movieList \
        .view

    lappend result [expr { [movieList getMovies] == [.view getMovies] }]

    itcl::delete object editor
    itcl::delete object .view
    set result
} -cleanup {
    eval $CLEANUP
} -result [list \
    1 \
    ]
```

gave the result

```
==== list MovieListEditor should send movie list from movieList to view. FAILED
---- Result was:
0
---- Result should have been (exact matching):
1
==== list FAILED

l.MovieListEditor.01.test:     Total   1      Passed  0      Skipped 0      Failed  1
```

If the test is rewritten to use a custom match comparison that does an indirect evaluation of the expected value [tcltest]

MovieListEditor.test

```
…
variable expectedResult {}

proc indirect { expectedRef actual } {
   set expected [uplevel 1 [list ::subst $expectedRef]]
   if { [expr [list $expected != $actual] ] } {
      error "\nExpected: $expected\nActual:   $actual"
   }
   return 1
}

customMatch indirect [namespace current]::indirect


test list {
   MovieListEditor should send movie list from movieList to view.
…
   set expectedResult [movieList getMovies]
   set result [.view getMovies]
…
} -match indirect -result $[namespace current]::expectedResult
```

then the output is the much more helpful

```
==== list MovieListEditor should send movie list from movieList to view. FAILED
---- Error testing result:
Expected: ::MovieListEditor::test::starWars ::MovieListEditor::test::starTrek
::MovieListEditor::test::starGate
Actual:   {Star Wars} {Star Trek} Stargate
==== list FAILED

MovieListEditor.test:   Total   1      Passed  0      Skipped 0      Failed  1
```

Unfortunately I learned this too late for this paper.