Using Design Patterns to Help Test Your Classes and Functions

Authors: Bruce Trask, Angel Roman

Complex software programs are comprised of many parts that work together. This started becoming most manifestly apparent with the introduction of the notion of subroutines. This notion of many modules working together in concert to achieve a particular goal continues in the object-oriented world of software development. Particularly with complex systems, frequently more than one developer is involved with a particular project and thus many modules are developed independently from each other. The notion of dependency is very familiar to any software developer, whether he/she knows it or not. If a piece of code makes a call on a function, then that code has a dependency on that other piece of code. This is not the only scenario that constitutes a dependency, but it is the simplest.

So, if code has a dependency on another module and that module is not available yet or is very complex to set up and run, how does one proceed with development of the code that is dependent on that as yet unavailable code? This becomes very apparent when one wants to *unit test* a particular module. "We often want to test the behavior of a module in isolation from the modules it uses."[1]

This is particularly needed in the world of Embedded Systems Programming (ESP) and Distributed Real-Time Embedded Systems (DRE) systems. In ESP, the hardware upon which software must run is frequently being developed in parallel with the software and thus is not available for use by the software developer until late in the development cycle. Time to market pressures prevent the development cycle from being serialized; the software developer "waiting" for the hardware to become available before software development can begin. Additionally, the distributed nature of DRE systems can make it very complex to bring up all distributed elements upon which a module depends, making it more complicated than it need be for testing a particular module.

The solution to this difficult problem is the combination of Test First Development techniques in combination with some simple design patterns. This technique is described in full in [1] and is sometimes called "spoofing", "mock objects" or "using factories as test fixtures"[1]. Our experience with this technique has shown it to be extremely useful in demonstrating to developers the efficacy of design patterns and the power of "programming to an interface, not an implementation" as propounded by the Gang of Four [2]. This concept sounds good but many times fails to hit home with developers and students until they try these techniques. Once they do, a whole bunch of principles come into full view:

1) The power of Test First Design
2) Programming to an interface, not an implementation
3) The Dependency Inversion Principle [1]
4) The Strategy Design Pattern [2]
5) The Abstract Factory Design Pattern [2]

6) How the Strategy design pattern and the Abstract Factory design pattern work together and why

The techniques involved include:

1) Test First Design
2) Refactoring (particularly refactoring to patterns[3])
3) Dependency Inversion
4) Design Patterns

All of this from a simple short example that will be illustrated below.

As alluded to above, there are 4 major participants to this app:

1) the test case
2) the class under test (CUT)
3) the class(es) upon which the CUT depends
4) the strategy design pattern
5) the abstract factory design pattern

As is the case when teaching any design pattern, it instructive to look at the state of the application before and and during the application of design patterns

In the case of ESP, the usual UML diagram is shown in Figure 1.

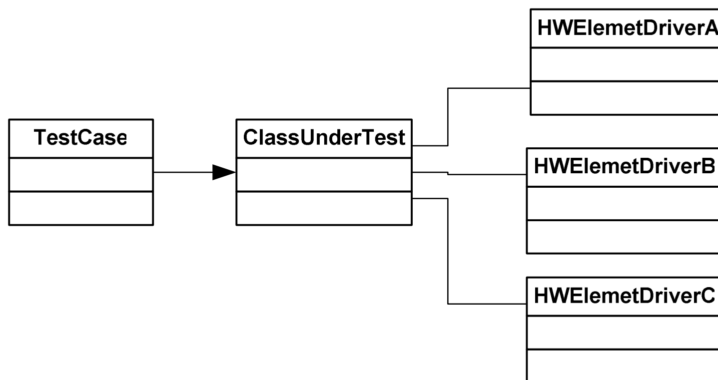


Figure 1

What are the potential problem with this setup?

1) The software element that communicates with the real hardware, the driver, or the real hardware itself, a particular processor board, may not be available at the time when the developer of ClassUnderTest wants to develop his code. Thus if the ClasUnderTest has direct dependencies upon those elements, he/she could be delayed.
2) As part of testing ClassUnderTest, the developer wants to ensure that the correct messages are being sent to the software hardware elements, A, B and

C, but the TestCase has no visibility to these are they may be called by protected or private method of ClassUnderTest.

3) The diagram violates the Dependency Inversion Principle, the Open Closed Principle and the principle of programming to an interface not an implementation. The ClassUnderTest has a hard and direct dependency to the HwA, B and C modules and thus it become very difficult to reuse CUT in other locations or with other implementations.

What is the solution? The most immediately solution is the application of the Dependency Inversion Principle, the “programming to an interface not an implementation” principle, and the strategy pattern leading to the following UML diagram:

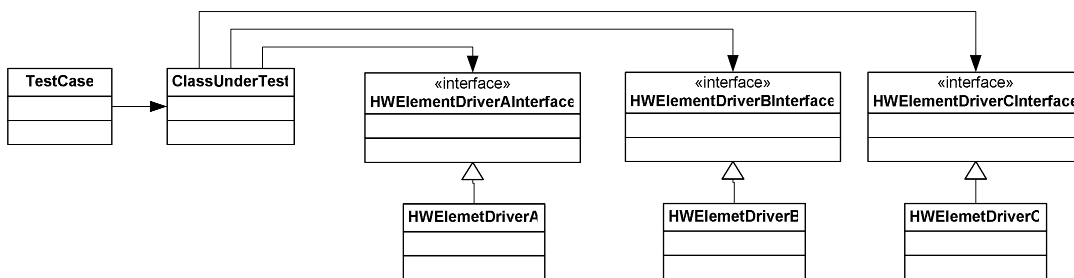


Figure 2

Here we are applying the strategy pattern to 3 different degrees of change in the software, A, B, and C. This will allow us to run ClassUnderTest with any module that supports the HWInterfaces not just the concrete implementations shown in Figure 1 as shown in Figure 3.

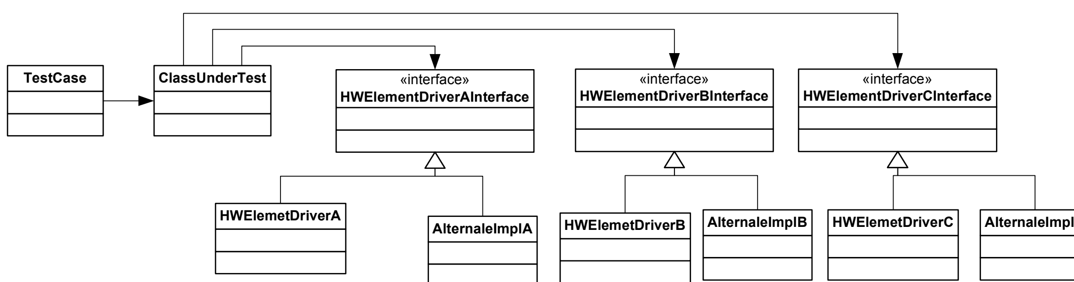
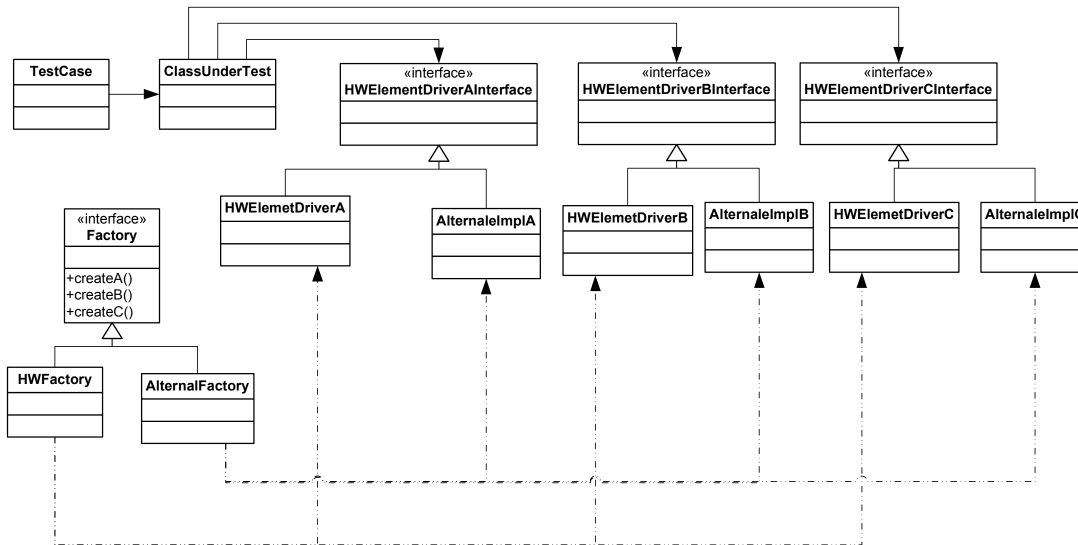


Figure 3

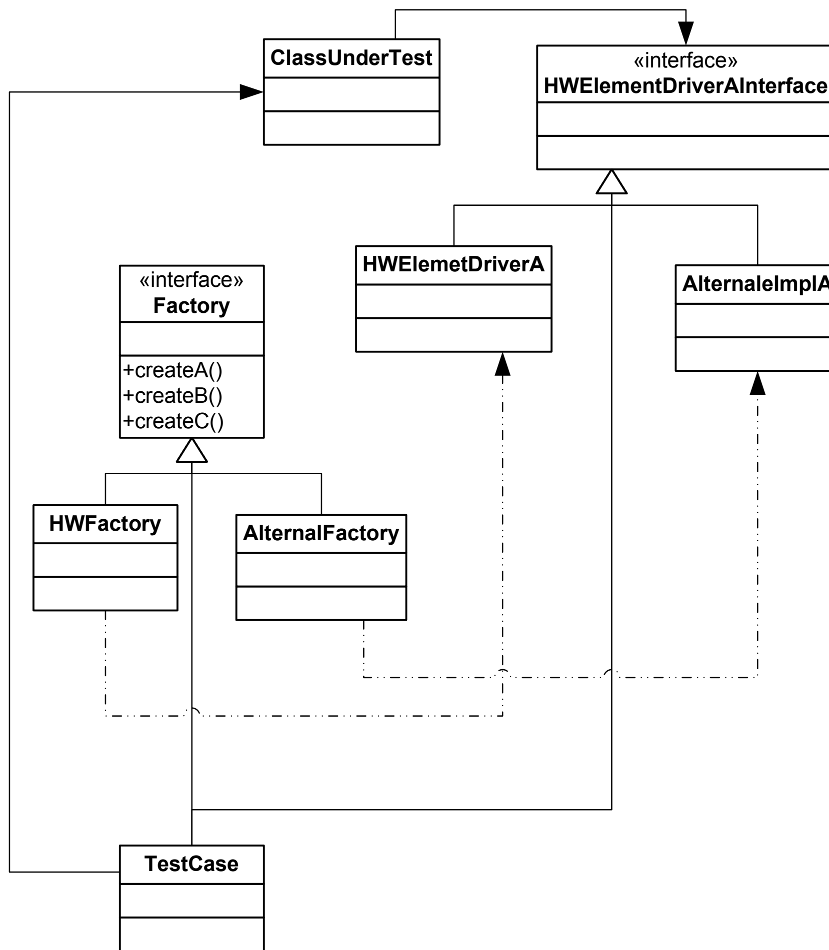
Two problems remain, one from the original list and *one brought about by the introduction of the Strategy Pattern*. Taking up the latter first: frequently when one introduces a design pattern to solve a particular “smell” that software might have, it solves a particular problem but simultaneously introduces a new one that most times requires the introduction of *another design pattern* to address it. In this case, it can happen that only certain concrete strategies can be combined with other concrete

strategies. In other words, there could be families of strategies that must work together. E.g. only the alternate strategies A, B, and C might work together. This introduces a configuration problem as to how to ensure that the ClassUnderTest is configured with the correct group of strategies. This force is the exact problem handled by the Abstract Factory design pattern: "a family of related product objects is designed to be used together, and you need to enforce this constraint."



The final piece of the puzzle is to solve how the TestCase can test the interaction of ClassUnderTest with all the strategies. This piece of the puzzle, if the developer or student really gets it, is the real eye opener and brings it all together.

What is involved is making the TestCase into not only a TestCase but also into a concrete factory and into concrete strategies as shown below (with some classes not included for simplification):



As the UML diagram shows, the TestCase IS-A Factory and IS-A HWElementDriverAInterface. The punch line in all of this is that the ClassUnderTest never knows whether he collaborating with the real hardware driver, an alternate implementation or a test case. When configured with a test case, the test case can "intercept" calls to the strategies to ensure they are correct based on calls uses as stimulus into the ClassUnderTest. The test case is "spoofing" both the factory and the strategies[1].

To summarize the benefits of this approach:

1) The ClassUnderTest can be developed in parallel with the other modules
2) The ClassUnderTest can be reused with other implementations
3) The TestCase can inject calls into ClassUnderTest and simultaneously ensure that the call coming out of ClassUnderTest are correct.
4) Almost more importantly, when students and developers have coded a simple example using the above technique, their understanding increases immensely for the main benefit of object-orientation, i.e. polymorphism. This is illustrated with a non-toy example that they can use thereafter for *every one* of their own classes from here on out.

5) Developers and students see how patterns work together and how a given design is refactored into using patterns based on seeing the exact problems they solve (and in some cases introduce)

[1] http://www.objectmentor.com/resources/bookstore/books/AgileSoftwareDevelopmentPPP

[2] http://www.aw-bc.com/catalog/academic/product/0,1144,0201633612,00.html
[3] http://www.aw-bc.com/catalog/academic/product/0,1144,0321213351,00.html