Testing Benefits of SOLID Principles

Easa El Sirgany

easaemad14@gmail.com

Abstract

SOLID design principles are well known for their impact on flexibility and maintainability of software development, but it is rare (if ever) that these principles are touted for their benefits with respect to testing.

In my quest to find a testing library to suit my needs for testing C++ libraries, I found that most frameworks were merely extensions of older C testing libraries, allowing for classes and member methods. The problem with this was that it failed to address some of the more important aspects of Object Oriented Programming (OOP) languages, specifically inheritance. Due to this shortcoming, any tests built for a library would only work for the library, and all derivations would need to implement their own test; this goes against everything software stands for.

After understanding the problem with building a framework that could do this for any library or base class, the following two-step solution was developed: build testing assertions into the source (something that should be done regardless of testing) and abstract test functions to work for any derivation of a base class.

This paper details the general approach taken in creating actionable steps for existing software to allow for proper software testing ideologies and strategies. The first was identifying SOLID design principle violations in the source and educating the developers why these violations are important (and how fixing them simplifies their lives), and the second was educating developers and SQA engineers alike on how to build templated testing functions in order to build proper unit tests.

Biography

Easa El Sirgany studied computer engineering with an emphasis in security (primarily encryption) and mathematics. With his love of breaking things in order to understand how to better them, the transition to a Software Quality Assurance Engineer was seamless.

More recently, he has spent his time building automated testing frameworks and tools for industrial-grade 3D printers.

1 Introduction

One day my boss tasked me with writing unit tests for a suite of C++ libraries under development at the time. I had limited knowledge in existing testing tools, most of which came from testing C and Hardware Description Languages (HDLs), due to my background in Embedded Systems. After some research, I concluded that a framework to handle my needs did not exist. The problem stemmed from the fact that the few frameworks that I looked into could not address one of the strongest benefits of OOP languages: inheritance.

The idea of testing libraries piqued my interest because building a test suite that was complete and verifiable seems impossible without having knowledge of the implementation. Firstly, it is possible for objects of a library to be abstract, i.e. have at least one pure virtual method. In such cases, it is not possible to instantiate the class for testing, since we will need to create a derivation in order to override the pure virtual methods.

Another problem with this approach was the fact that writing a unit test for a library would only apply to the methods of the base classes. For example, if I were to write a suite of tests for a base class with expected output of a virtual method, this test is no longer valid as soon as someone overrides the virtual method.

From the understanding of the two problems noted above, I implemented two different solutions. The first solution was the more obvious approach: write a program that generates all possible combinations of derivations, and test these against a framework of templates. This idea was very ambitious and required me to use C++ in ways I did not know were possible, but still failed even though it compiled and ran just as I expected.

The underlying problem of all of this was the code that I was testing was not built with testing in mind. Those crazy people that spend their days on the sidewalk preaching about Test Driven Development (TDD) were right all along. Even though I was able to automate the process of building derivations of the base class, I had no way to assert the success of the test run. This new understanding of the problem led to the second solution: documenting the problems of the source (libraries) using SOLID design principles.

SOLID design principles are composed of five design principles used by Object Oriented Programming (OOP) languages in order to increase flexibility and maintainability of software, but it also comes with some lesser-known testing benefits. The five paradigms are as follows: Single Responsibility Principle (SRP), Open/closed principle, Liskov's Substitution Principle (LSP), Interface-Segregation Principle (ISP), and Dependency Inversion Principle (DIP).

This paper will showcase the testing benefits a couple of the SOLID design principles provide, as well as a practical application to using these paradigms for testing; namely SRP and LSP. The example code used in this paper is written (not compiled nor tested) in modern C++, and is just that: example code. There are several guidelines and rules ignored in order to optimize for space and/or complexity.

2 Single Responsibility Principle

SRP is the easiest to implement and understand how this makes unit testing and TDD much easier to utilize. Robert C. Martin (Uncle Bob) states, "A class should only have one reason to change" (Martin 2003, 95). The idea was an expansion of a paper written by D.L. Parnas on the strengths of simplifying large solutions into smaller components, starting with components "which are likely to change" (Parnas, 1972). This has been a common idea well before OOD, and those familiar with the Unix Philosophy will know this to be very similar to the first rule: "Make each program do one thing well…." (McIlroy, Pinson, Tague 1978).

2.1 Implementing Single Responsibility Principle

The core idea behind SRP is that each change should require only one modification to a single piece of the system. For example, the following logger class is responsible for logging messages to a log file maintained by the class:

```
1. void Logger::log(const std::string& message, const std::string& level)
2. {
3.     if(lFile.is_open()) {
4.         std::lock_guard<mutex> lock(logMutex);
5.         lFile << "[" << level << "]" << message << std::endl;
6.         lFile.flush();
7.     }
8. }
```

The third line of this method checks to ensure that our log file is open, if not the logger does nothing. After this, the std::mutex is locked to save any threaded implementation of the logger. Line 5 is the area of interest covered in depth below, and the sixth line synchronizes the file with the underlying device in order to prevent any data loss in the event of an unexpected shutdown.

The first issue with line five is that any derivation of our logger will have to override the whole ::log() method in order to modify the formatting of the logged message; this breaks the open/closed principle. Every logger should log the same way, i.e. write to a file and flush to sync with the underlying device with mutual exclusion. Furthermore, there are at least two reasons to change the ::log() method above: *how* the logger class logs messages and *what* the logger logs to the log file. Separating the "how" and the "what" could look something like the following:

```
1. void Logger::log(const std::string& message, const std::string& level)
2. {
3.     if(lFile.is_open()) {
4.         std::lock_guard<mutex> lock(logMutex);
5.         lFile << formatMessage(message, level) << std::endl;
6.         lFile.flush();
7.     }
8. }
```

Going back to the fifth line of the above example, the ::formatMessage() method makes use of our two parameters passed to the ::log() method in order to generate a formatted log message. This allows the logger to format the message as needed, as well as any derivation of this class to override the formatting of a message (assuming ::formatMessage() is a virtual method).

Before moving on to analyzing testing benefits of SRP, I should note that this is a very simple example and differentiating between reasons for change is a common problem. Uncle Bob wrote a blog articulating this in more depth, stating the "reason is about people" (Martin 2014).

2.2 Testing Single Responsibility Principle

More importantly (for the purpose of this document) than the flexibility that SRP provides, breaking the formatting of messages into different methods simplifies testing strategies. It may be difficult to understand the full impact that SRP has on testing from the example above due to the simplicity of the ::log() method since there are only two operations that are performed, neither of which addressing points of failure. The following example initializes the logger class:

```
1.  bool Logger::init(bool verbose)
2.  {
3.      auto retVal{false};
4.      if(preInit()) {
5.          lFile.open(fileName);
6.          retVal = lFile.is_open();
7.      }
8.
9.      postInit(retVal);
10.     return retVal;
11. }
```

This example initialized our logger class, using a ::preInit() and ::postInit() method on lines 4 and 9, respectively. This is a common open/closed practice to allow derivations to extend upon the logger initialization, without modifying the inherent initialization process. Consequently, these two methods ensure the ::init() method conforms to SRP, assuming they are both virtual, since they allow changes to be contained to one method. If our pre-initialization succeeds, we attempt to open the (member variable) log file and set our return value before returning.

Testing the initialization of the logger, there are two conditions that are required to be true in order to assert success: ::preInit() returns “true” and we are able to open the file lFile. The beauty of conforming to SRP is that if we fail, we can differentiate between the operation that failed by checking ::preInit() directly. Assuming that Logger is a base class, ::preInit() should always return true if it is a virtual function to allow the least amount of restrictions if a derived class chooses to not override this method. Therefore, if the initialization fails, there is a direct correlation with opening the file.

Furthermore, if we override ::preInit() to always return false, we can double-check this against the result of testing the ::init() method. If ::preInit() returns false but ::init() succeeds, it can be stated with confidence that there is a bug in the initialization of the logger. This ties nicely into LSP since this test will work for any derivation of the logger.

3 Liskov Substitution Principle

The formal definition of LSP states the following subtype requirement: “Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) should be true for objects of type S where S is a subtype of T.” (Liskov and Wing 1994, 1812). I find this terminology confuses many (not completely from the usage of mathematical notation) due to the “ϕ(y) should be true” portion of the quote. This statement does not mean that every subsystem that returns true for type S should return true for type T, but rather *how* the subtype is determined should be true for all derivations. The following example demonstrates the use of LSP by calculating the area of rectangles:

```
1.  class Rectangle {
2.  protected:
3.      int width, height;
4.  public:
5.      Rectangle(int w, int h) { width = w; height = h; }
6.      virtual ~Rectangle() = default;
7.      int getWidth() { return width; }
8.      int getHeight() { return height; }
9.      long int calculateArea() { return width * height; }
10. protected:
11.     virtual void setWidth(int w) { width = w; }
12.     virtual void setHeight(int h) { height = h; }
13. };
```

The Rectangle class contains width and height private member integers, along with their public getters and protected setters. The setters on lines 9 and 10 are virtual because the Square class below will override the functionality of these methods. The ::calculateArea() method is important for testing and conforms to LSP; further explanation below. Since (mathematically) a square is a rectangle, the following example should be correct:

```
1.  class Square : public Rectangle {
2.  public:
3.      Square(int w, int h) : Rectangle(w, h)
4.      {
5.          if(w not_eq h) {
6.              throw invalid_argument("Sides not same size!");
7.          }
8.      }
9.      virtual ~Square() = default;
10.
11. protected:
12.     void setWidth(int w) override { width = w; height = w; }
13.     void setHeight(int h) override { height = h; width = h; }
14. };
```

Since a square is a rectangle with all four sides being the same length, the Square class requires both width and height to match, and a call to ::setWidth() and ::setHeight() set both values to ensure our square is always valid. The inherited ::calculateArea() method works for both the Square and Rectangle classes, so it conforms to LSP. The following test function tests this functionality for any type of rectangle.

```
1.  bool testArea(Rectangle& r)
2.  {
3.      r.setWidth(3);
4.      r.setHeight(4);
5.      return (r.getWidth() * r.getHeight() == r.calculateArea());
6.  }
```

If a base class Rectangle is passed to the testArea() function, it can be verified that the area is equal to the product of the width and the height (12 in this case). This validation is the same for a Square class, with the difference being that the width and the height always being equivalent. After line 3 in the test function, all sides of a square will have a value of three until the call on line 4 changes all sides to four. Regardless of order of setting width or height of a Square class (or the values used in the setters), it will always have correct implementation when tested with line 5.

The use of the rectangle and square classes is a popular example for LSP, and Uncle Bob wrote an article on LSP using a very similar example (Martin 1996, 3-7). Before moving onto testing strategies using LSP, it is important to understand why Uncle Bob and I disagree as to whether a square is a rectangle or not. The following is the test function used by Uncle Bob:

```
1.  void g(Rectangle& r)
2.  {
3.      r.setWidth(5);
4.      r.setHeight(4);
5.      assert(r.GetWidth() * r.GetHeight() == 20);
6.  }
```

The only difference between the testArea() and the g() test functions is the assertion (or Boolean return value of success) on line 5 for both functions. The problem with Uncle Bob's test function is due to the Rectangle class being incomplete or the implementation of the test is just wrong.

If the Rectangle class has requirements (depending on expectations of the class itself and how it will be used) to calculate the area of a rectangle, this **must** be built into the class and tested in the same manner as the testArea() function. If no such requirement exists, however, testing this functionality is not a valid test of any type of rectangle. Rewriting the test in order to ignore the calculation of area better demonstrates the requirements of the Rectangle class:

```
1.  void gPrime(Rectangle& r)
2.  {
3.      r.setWidth(5);
4.      r.setHeight(4);
5.      assert(r.GetWidth() == 5); // Square classes will fail here
6.      assert(r.GetHeight() == 4);
7.  }
```

gPrime() has the same error as the g() test function without the noise of area calculation, i.e. this test makes the assumption that the width will not be changed for any rectangle when setting the height. **If** this were a requirement of a Rectangle, **then** it would not be possible for a Square to be a Rectangle. Without this requirement, however, this test is invalid and should be two different tests: testing validity of setting the width and testing the validity of setting the height of a Rectangle. This could be also be done by switching lines 4 and 5 in the gPrime() test, which will work for any type of Rectangle.

3.1 Testing Benefits of Liskov Substitution Principle

One of the biggest benefits of LSP are the testing benefits provided by merely conforming to LSP. Since "ϕ(y) should be true for objects of type S where S is a subtype of T", any test written for an interface or base class can be used for all derived classes. Before moving onto testing our Rectangle class, it would be helpful to have a class that does some error checking:

```
1.  class Rectangle {
2.  protected:
3.      int width, height;
4.
5.  public:
6.      Rectangle(int w, int h)
7.      {
8.          if(not setWidth(w) or not setHeight(h) {
9.              throw invalid_argument("Invalid dimensions");
10.         }
11.     }
12.
13.     //…
14. protected:
15.     // …
16.
17.     virtual bool setHeight(int h)
18.     {
19.         auto retVal{false};
20.         if(h > 0) {
21.             height = h;
22.             retVal = true;
23.         }
24.         return retVal;
25.     }
```

Now that we have a Rectangle class that asserts valid dimensions, the following tests are valid for any type of rectangle:

```
1.  template<class R>
2.  bool testGetWidth(int w, int h)
3.  {
4.      auto retVal{false};
5.      try {
6.          R rect(w, h);
7.          retVal = (w == rect.getWidth());
8.      } catch(…) {} // This demonstration does nothing with caught exceptions
9.
10.     return retVal;
11. }
12.
13. //…
14.
15. template<class R>
16. bool testCalculateArea(int w, int h)
17. {
18.     auto retVal{false};
19.     try {
20.         R rect(w, h);
21.         retVal = (rect.getWidth() * rect.getHeight() == rect.calculateArea());
22.     } catch(…) {}
23.
24.     return retVal;
25. }
```

Each of these tests attempt to instantiate a Rectangle object and determine success of the method tested. The benefit of this approach, as compared to the assertion method used in the last section, is that the test allows for failures when expected. For example, if I run testGetWidth<Square>(-1, 3), this test will fail (for multiple reasons), but this is still a valid test that should be run to ensure that it fails gracefully.

The importance of the above test functions is that the testing implementation for any type of rectangle can utilize these for their testing strategies, since success or failure is not inherent within these functions. This is an abstraction layer between the implementation and the expectation of a Rectangle class object. The following example test will implement a discrete test for evaluating the area of a Square specifically, using the knowledge of the expected behavior of a Square:

```
1.  void validAreaSquareTest() {
2.      for(auto side{1}; side > 0; side++) { // Overflow will terminate
3.          if(not testCalculateArea<Square>(side, side)) {
4.              cerr << “Area test failed for square with side: ” << to_string(side) << endl;
5.          }
6.      }
7.  }
```

Since the expectation of any Square object is that this should pass for all positive width/height values, this test should pass for all values without having to modify the template test function. For example, in the event that requirements change and operating in two dimensions is no longer sufficient, these tests are still valid for testing anything that derives from a Square object (much like it did when testing a Square object derived from a rectangle base class).

4 Conclusion

In conclusion, implementing SOLID design principles not only benefits the flexibility and maintainability of the software itself, but also simplifies testing strategies by reducing redundancy of testing functions and allowing for inherent assertions of expected behavior.

Software contains more functions that are easier to understand when it conforms to SRP and, therefore, simplifies maintainability when deriving from base classes and interfaces. This allows tests to more thoroughly test the software and provide better feedback when the system does not perform as expected, while allowing specific tests to cater to expected behaviors of different types of objects.

Likewise, test functions built for interfaces and base classes are valid for all derivations when the software complies with LSP. Even though there will be a greater number of smaller test functions written (the side effect of SRP), these functions hold true for all children when abstracted from the implementation of the test, meaning fewer tests are written in total and redundancy is removed.

Since everyone loves a happy ending, I will share the outcome of the implementation of this work when applied to the project I was asked to work on. I wrote many stories and bugs relating to violations of SOLID design principles for a handful of existing libraries so that I could begin working on templated test functions. During the refactor of the libraries, developers found bugs that had existed for years, before I even started implementing test.

References

Martin, Robert C. 2003. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall

Parnas, D.L. “On the Criteria To Be Used in Decomposing Systems into Modules.” Communications of the ACM 15, no.12 (December 1972): 1058

McIlroy, Doug, Pinson, E.N., Tague, B.A. "Unix Time-Sharing System: Foreword" (PDF). The Bell System Technical Journal. Bell Laboratories. (8 July 1978): 1902–1903

Martin, Robert C. 2014 “The Single Responsibility Principle.” The Clean Code Blog. https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html (accessed June 20, 2019)

Liskov, Barbara H. and Wing, Jeannette M. “A Behavioral Notion of Subtyping”. ACM Transactions on Programming Languages and Systems 16, no. 6 (November 1994): 1811-1841

Martin, Robert C. 1996 “The Liskov Substitution Principle.” C++ Report. https://web.archive.org/web/20151128004108/http://www.objectmentor.com/resources/articles/lsp.pdf (accessed June 20, 2019)