Detangling a Code Base: Measuring Testability of Problematic Dependencies and Applied Refactorings

Erik Dietrich* Gitty Gottlieb*

*University of Illinois at Urbana-Champaign, {ediet2, gottlie3}@illinois.edu

Abstract

To many developers, testing currently occupies the realm of necessary evils; a task which must be done but is not the preferred choice of activity. We believe that much of the negative association with testing stems from code dependencies which complicate testing to the point of increasing its costs in terms of time and resources by several orders of magnitude. In this paper, we examine various forms these dependencies can assume and explore possible refactorings. We also propose a metric to evaluate the testability of a class based on its dependencies. Finally, we justify the metric by presenting its evaluation on three distinct code bases.

1. INTRODUCTION

Over the last decade or so, automated unit testing has steadily gained popularity. Once considered a luxury or a novel concept, it is now considered indispensible to most commercial software development operations. Methodologies such as Test Driven Development (TDD), Test-First Development, and Continuous Integration highlight the widespread appeal of the concept. Unit testing offers the benefits of providing a measure of confidence in the correct functionality of an application, guarding against regressions as new features or changes are introduced, and even serving as additional documentation of system requirements. The utility of the practice is quite commonly accepted.

In spite of the widespread acceptance and belief in the process, however, a sizable number of developers continue to neglect to use it. This may be for a number of reasons, i.e, working with legacy code that is not conducive to unit testing, a lack of time, a lack kof interest, a lack of training, etc. We feel that all these concerns can be boiled down to one core deterrent, namely, that the developer does not consider the perceived time and effort required worth the aforementioned benefits. In other words, unit testing is avoided because, for one reason or another, it is too hard.

As such, a methodology that could make uni testing easier is likely to be of value to the software engineering community. While the most obvious solution to this problem is to train people in unit testing, that is neither a software engineering problem to be solved nor is it necessarily practical. In lieu of improving the testers, one can focus on making the testing framework easier to use or making the System Under Test (SUT) easier to test. As of the time of this writing, there are already countless efforts to improve unit testing technology. Some of these include mocking frameworks such as Moles[1], automated test generation tools such as Pex[2], and testing frameworks such as NUnit[3] or MS Test[4]. This paper focuses on the remaining area of improvement, that of optimizing the SUT for testing.

Before approaching the work of facilitating easier testing of a SUT, one must define the criteria for measuring ease of testing. While some metrics for testability already exist[5], to the best of our knowledge there is no specific metric that addresses how easily it would be to write manual tests for a SUT, and manual test writing is the driving force behind methodologies like TDD.

As such, we define such a metric. To do this, we identify a number of coding patterns that tend to make unit testing difficult and determine specifically what about them poses the challenges to unit testing. We then identify a series of refactorings that help alleviate the testing difficulties. This work served as a guideline for developing a formal metric which, based on the evaluations we present, accurately predicts how easily manually testable a class is. Our work principally targets C# applications built in the .NET Framework 4.0, but we believe the metric can be extended to any object-oriented programming language.

We propose a metric, T(C), which is based on the relationship between a class, its total number of dependencies, it number of interface dependencies, its number of injected dependencies, its number of static state dependencies, and the T(C) scores of the classes from the SUT upon which it depends. We evaluate the metric by calculating the score of a class in a system, seeing how much code coverage, C(C), can be achieved on the class in a set amount of testing time, refactoring to improve T(C), and then comparing the original C(C) with coverage achieved in the same amount of test development time.

We demonstrate this evaluation on three different systems, a small sample application built for the purposes of this project, an open source CMS project called Rainbow[6], and a proprietary solution. We present the results of our findings, discussions, and conclusions.

The primary contributions of this paper are one, a taxonomy of dependency problems that create difficulties with manual unit testing and a taxonomy of refactorings that can eliminate these difficulties; two, a metric, T(C), for evaluating the testability of a SUT on a class-by-class basis; and three, an evaluation of T(C) as applied to actual software systems before and after relevant refactoring.

The remainder of this paper is organized as follows. Section two describes the problem, section three details a taxonomy of problematic dependencies, section four explores the relevant refactorings, section five covers our testability metric, section six presents an evaluation of our work, section seven contains a discussion of related work and what we have gained from work on this project, and section eight concludes.

2. THE PROBLEM

In this section we discuss in more detail the problems encountered in building a robust, automated unit test suite for production code. Enumeration of these challenges helps make obvious the motivations for our research and the considerations involved in generating our testability metric.

As mentioned above briefly, it is common for software development enterprises to avoid unit testing code in spite of overwhelming evidence that the suite of unit testing significantly improves software reliability and maintainability. The resistance arises for a variety of reasons, which generally boil down to the notion that the effort of creating and maintaining unit tests is not worth the benefits they provide. It may be that developers are pressed for time and consider the writing of unit tests to be a luxury or that they have grown used to and comfortable with a methodology that does not include unit testing. It may be that they lack the training in testing tools or understanding of the concepts involoved, or it may be that system upon which they work is not written in a way that is conducive to unit testing.

We believe that developer resistance in general stems from the difficulty of the task and perceived invasiveness of the new methodology into the status quo. If asked whether or not they would be willing to write an extra four lines of code in two minutes to dramatically improve the reliability of their software, few developers would protest. However, somewhere between that wonderful proposition and the reality of undertaking a unit testing approach, the line becomes blurred and the motivation wanes. Our contention is that the more closely the prospect of testing mirrors the former proposition, the more likely it is that unit testing will be adopted.

Unit testing consists of two main components: the System Under Test (SUT) and the testing apparatus, which includes the unit test code itself as well as any testing language extensions/libraries, mocking frameworks, and test running frameworks. It is important to note that while the items in addition to the test code are standard, none of them is strictly necessary for automated unit testing; a piece of external code that exercises the SUT and verifies requirements thereof is sufficient. We choose to focus on aspects of the SUT and how it can be constructed in such a way as to be more conducive to verification in the form of automated unit testing.

It is our belief that the principles of good object-oriented design correlate strongly with code that is easy to test. As such, we perceive a natural synergy between refactoring code toward better design and designing testable systems or making existing systems more testable. The inverse is true as well; there is a natural, vicious cycle that occurs as poorly designed systems with poor testability mature, starting out as relatively uncomplicated pieces of engineering and growing over time, as difficulties relegate testing to a “nice idea,” into behemoths that are increasingly difficult to test and maintain.

If the prospect of testing were made easier form the beginning, it is more likely that developers would start and stick with it on new projects, and the aforementioned natural synergy would tend to keep the code base cleaner and preserve testing practices. Once a project is far enough down a path of low testability, the likelihood of design improvements which would increase testability dramatically diminish. The code base becomes degenerative, prompting rewrites of the software which are often pointless, as the same lack of verification and good design in the process leads to the same result.

We believe that higher testability of a system leads to better, more maintainable software. As such, it is important to define system testability so that it can be assessed and preserved throughout development. If a software department is considering whether or not to adopt some sort of unit testing practice, it could be invaluable for individual developers to have a metric that tells them whether or not they are making the unit testing task harder as they write code. It would also be helpful for them to see which refactoring or design processes were beneficial to the testability of the system.

3. ASSESSING TESTABILITY: A TAXONOMY OF PROBLEMATIC DEPENDENCIES

In this section, we detail some common object-oriented practices that create testability problems. Testability problems are the direct result of the structure of dependencies within the SUT. The more interdependent components are upon one another, the more difficult it becomes to isolate them for testing purposes.

One of the most important distinctions between a small independent project and a large collaborative (often commercial) one is how complicated the dependencies involved are. For example, a student may write a utility to copy a series of files. Conceptually, the only dependencies here are on the OS file abstraction and any libraries of the programming language that the student needs. These are generally considered to be quite stable, and they are not internal to the student’s utility. That is, the student depends only on predefined abstractions.

In a large scale software enterprise with many collaborators, dynamics change. Consider the example of a company’s internal invoicing and customer tracking system. This system is likely to have several major components, one that interacts with the database, one that manages business logic, one that presents information to the users for review and edit, one that generates report files, etc. These components will, in turn, have their own sub-components and, depending on the architecture, there may be several more hierarchical or interactive organizations before getting to the level of individual classes and methods. In such systems, with many people collaborating and requirements frequently changing, there are obviously many, many more dependencies than the student writing a utility for himself.

Management of these dependencies is incredibly important for the maintainability and quality of the software. An application with a convoluted nest of internal, cyclic dependencies becomes brittle by virtue of the fact that changes to one component have unintended consequences for other components. Additionally, the individual components become more difficult to test because their state depends on the state of many other parts of the software. As such, it is important to carefully manage these dependencies and minimize them as much as possible when modifying or adding to a code base. In order to do that, one must first be able to recognize dependencies and the likelihood of them being problematic.

Below are some dependencies that are potentially problematic and an explanation of the problems that they might cause. All examples are in C#. it should be noted that a style of dependency being listed here does not mean that any give usage of it **is** problematic, just that it might be. After all, while loose coupling is an important goal of both software development and testability, having software components know about one another is unavoidable in functional systems.

3.1 Inline New

This occurs any time an object is instantiated inside of a class method or in the class level declaration. In the example below (Figure 1), any client of the car class (i.e, any class which instantiates a Car and uses it) now has immediate knowledge that Car has an internal Engine that performs its own operations. This is a hidden dependency. Changes to the Engine class can potentially cause changes to the behavior of Car, and nobody using Car has any way of knowing that without inspecting the code. This may be particularly problematic if Car and Engine are contained in separate libraries where clients cannot access their source code. This coupling also causes unit testing difficulties. Conceptually, it is not possible to test Car without also testing Engine. These classes are inextricably coupled.

```
class Car
{
 public Car()
  {}

  public bool StartEngine()
  {
    Engine myEngine = new Engine();
    myEngine.Start();
    return myEngine.IsStarted;
  }
}
```

Figure 1

3.2 Static Classes with State

A static class is one that cannot be instantiated and is loaded automatically by the .NET framework when its namespace is loaded. Because of this, the static class can be accessed at any time from any method. So, if this static class contains program state, clients of the class become tightly coupled to it. Similar to the inline new case, this PageLoader has a hidden dependency. Users of this class have no knowledge that it is accessing LoginInfo.

In this case, there is additional potential for problems. Because it is static and thus accessible from anywhere, and because it contains state, LoginInfo's members are essentially global variable. As such, a class with a dependency on LoginInfo has a logical dependency on any other class that interacts with LoginInfo. For instance, if someone modified an unrelated class called DatabaseReader to set LoginInfo's LoginID to null, PageLoader's LoadPage( ) methods would have different behavior, even though it does not interact with DatabaseReader or even with LoginInfo's LoginID property.

```
public static class LoginInfo
{
   public static string LoginId {
get; set; }
   public static bool IsUserLoggedIn
{ get { return
!string.IsNullOrEmpty(LoginId); } }
}


public class PageLoader
{
  public PageLoader() {}


  public void LoadPage()
  {
    if(LoginInfo.IsUserLoggedIn)
    { this.LoadPage(); }
    else
    { throw new
InvalidOperationException("You have
to log in to acccess this page!");
}
  }
  . . . .
}
```

Figure 2

3.3 Singletons

A singleton is a "Gang of Four"[7] design pattern that essentially creates a static/instance hybrid. The point of the singleton is twofold: it ensures that only one instance of an object can be created, and it provides a global point of access to that instance. It is the latter that presents a potential problem.

In many ways, the singleton suffers from the same issues as the static class – it is more or less a repository for global variables, and it allows methods and classes to hide their dependencies. It does allow a little extra flexibility that makes refactoring and reuse easier – singleton classes, because of their static/instance hybrid nature, can be interfaced and/or passed between instance classes and methods.

Singletons present some unique challenges of their own, however. Singletons are lazy-loaded, meaning that the first call to the singleton's instance will cause the singleton to load and subsequent calls will not. This means that a client class making the second or later calls to the singleton will behave differently in the application than it will if individually unit tested.

From a unit testing perspective, singletons are a disaster. While static classes may allow state to slip into them and create problems, singletons are **designed** to do this. As instance methods, they are not simple state repositories, but generally have instance behavior. The result is that each class that refers to a singleton becomes inextricably linked not only to the other classes that use the singleton, but to the **order** in which they use the singleton. With singletons in a code base, it frequently becomes impossible to test the majority of classes without the entire application running.

This tends to be a degenerative problem with code bases. Initially, a singleton might not create problematic dependencies – a logger, for example. However, over the course of time, they tend to turn into convenient shortcuts for communication between classes in different modules. The more classes that depend on them, the more tightly coupled and untestable code becomes.

```
public class LoginManager
{
  public string LoginId { get; set; }
  public bool IsUserLoggedIn { get { return !string.IsNullOrEmpty(LoginId); } }

  private static LoginManager _Instance;
  public static LoginManager GetInstance()
  {
    if(_Instance == null)
    {
      _Instance = new LoginManager();
    }
    return _Instance;
  }

  private LoginManager()
  {
    Initialize();
  }

  private void Initialize()
  {
     ....
  }
}

public class PageLoader
{
  public PageLoader() {}

  public void LoadPage()
  {
    if(LoginManager.GetInstance().IsUserLoggedIn)
    { this.LoadPage(); }
    else
    { throw new InvalidOperationException("You have to log in to acccess this page!"); }
  }
  ....
}
```

Figure 3

3.4 Law of Demeter Dependencies

The Law of Demeter[8] can be summarized loosely by saying that a method should only invoke methods on itself, its parameters, or property objects in its class. One of the more prominent conceptual violations would be a statement like “int x = a.b.c.GetX( );” In general, the additional objects that are invoked introduce additional, non-obvious dependencies to the class.

While this may seem straightforward and reasonable enough, this presents a problem. UniversityCourse maintains only a list of string representing the last names of student enrolled in the course. However, by exposing a Student object, it introduces an unnecessary dependency on the Student class. Now even though UniversityCourse’s only responsibility is to maintain a list of strings, changes to the Student class affect it. If someone changes Student's GetLastName( ) method, UniversityCourse.AddStudentName( ) may no longer work.

In addition to the tight coupling to Student, unit testing becomes slightly more burdensome. Testers are now forced to create Student objects to test UniversityCourse. If Student is a data transfer object, this might be only a mild inconvenience. However, if Student requires other objects, the tester must instantiate those and any other objects they require, and so on.

```
public class UniversityCourse
{
  private List<string> _enrolledStudentLastNames;


  public void AddStudentName(Student s)
  {
    _enrolledStudentLastNames.Add(s.GetLastName());
  }
}
```

Figure 4

3.5 Context Dependency

A context dependency occurs when a class intended to be a general class exposing an API depends on more specific application logic. In an application where basic components contain dependencies on application logic (see figure 5), coupling is tight between the abstract base class FileType and the FileReader. FileReader probably should be a class that depends on very little other than perhaps some File libraries. With a setup like this, it depends on the application logic and is not broadly reusable.. if we want to use the FileReader to read a new type of file, we have to create a file type and then also add logic to the file reader.

From a unit testing perspective, this creates additional problems. The logic for the low level task of reading a file can break with changes to what appear to be business objects.

```
public class FileReader
{
  public void ReadFile(FileType type)
  {
    if(type is CustomerFileType)
    { this.ReadCustomerFile(); }
    else if(type is a ManagerFileType)
    { this.ReadManagerFile(); }
  }


  private void ReadCustomerFile()
  {
    ...
  }
  private void ReadManagerFile()
  {
    ...
  }


}
```

Figure 5

3.6 Class Dependency

This is far more common and not as likely as some of the previously listed dependencies to be a problem, but it can present some challenges. It just means that a class depends on another class as opposed to depending on an interface. Sometimes this is acceptable or even desired. However, there are situations in which this creates unnecessary rigidity.

From an application flexibility perspective, this (figure 6) could be better. Perhaps there might later be a desire to migrate the database to PostgreSQL. That is not trivial to do with all of the service later classes hard-coded to use MySQL data access objects. While the constructor injection is good from a loose coupling and unit testing perspective, this is still rather rigid. Passing in an interface would make it much simpler to implement a new database connection.

The same logic applies to unit testing. If CustomerService took an IDao parameter instead of a MySQLDao parameter, it would be much simpler to test the service. The tester could implement the interface with a mock object that would supply whatever results he wanted to the service, and he could even throw exceptions to test exception handling.

```
public class CustomerService
{
  private MySqlDao _dao;
  public CustomerService(MySqlDao dao)
  {
    _dao = dao;
  }

  public int CreateCustomer(Customer newCustomer)
  {
     _dao.StartTransaction();
     try
     {
       _dao.AddCustomer(newCustomer);
       int myId = _dao.GetLastCustomerCreatedId();
       _dao.Commit();
       return myId;
     }
     catch { _dao.Rollback(); return -1; }
  }
}
```

Figure 6

3.7 Complex Interface Dependency

If a class depends on an interface with many methods, it is unlikely that anyone is going to implements the interface, making the default implementation the *de facto* dependency. In other words, an overly complex interface dependency may as well be a class dependency.

Here (figure 7), ComplicatedClient appears to be quite flexible and loosely coupled. It uses constructor injection of an interface. However, a look at IComplicated suggests that it is extremely unlikely that anyone is going to create another implementation besides whatever implementation it was extracted from (as this is likely what happened, since interface creators would likely break it up long before it got to this point if the interface were designed first).

From a unit testing perspective, this is a more muted effect. A determined tester might implement the interface and actually implement only the methods he needed. Still, this is a significant hindrance in that it may cause a tester to postpone testing to “when he has time,” which in the commercial software development world is often synonymous with “never.”

```
public interface IComplicated
{
  void Method1();
  void Method2();
  ...
  void Method150();
}

public ComplicatedClient()
{
  private IComplicated _complicated;
  public
CompliatedClient(IComplicated
complicated)
  {
    _complicated = complicated;
  }
}
```

Figure 7

4. ASSESSING TESTABILITY IMPROVEMENTS: A REFACTORING TAXONOMY

The following is a series of refactorings which can be used to eliminate or mitigate the problems detailed in section three from a testing perspective. These may be used independently or in conjunction with one another in order to improve testability. It should also be noted that there is no “silver bullet” when it comes to refactorings. None of these will eliminate all problems, even when used in combination.

Another consideration is that applying these refactorings is generally extremely dependent on the context of the design. These refactorings, applied properly, preserve application functionality for the most part, but often some manual tweaking is required. Addressing dependency problems which inhibit flexibility is context sensitive. Changes beyond the scope of these refactorings may be necessary.

Refactorings, whether manual or automated, also carry with them certain risk. In the following sections, we address the potential risks of each refactoring that may be undertaken to improve testability.

4.1 Promote Method Variable to Parameter

Promoting method variables to parameters involves taking any variable declared and instantiated within the scope of a method and instead having that variable passed into the method as a parameter. In the case of a literal variable, a literal is passed into the method. In the case of an object, a parameter is passed in as a reference.

This refactoring is a powerful way to get rid of hidden dependencies, but it does have some maintenance overhead in terms of efforts. Performing this on a method in use by other classes will cause a change in the other classes. In some cases, the resulting code will not compile. Some refactoring tools allow specification of a default parameter to be plugged in, specifiable by the user. However, even if this is the case, it is unlikely that the same parameter value will be applicable to all clients, meaning that some manual refactoring is likely necessary.

Another consideration is that in the case of a reference parameter, null dereference is a possibility. For instance, in the Car example below (figure 8), if a null Engine starter were to be passed in, a runtime exception would occur. As such, this refactoring may also need to be accompanied by manual checks for null dereferences.

Before:

```
public class Car
{
  private Engine _engine;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter = new EngineStarter();
    myEngineStarter.Start(_engine);
  }
}
```

After:

```
public class Car
{
  private Engine _engine;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start(EngineStarter starter)
  {
     starter.Start(_engine);
  }
}
```

Figure 8

4.2 Promote Method Variable to Class Member

This is similar to the first example except that the method parameter becomes a class level variable with an accessor and a mutator. This refactoring is less intrusive that the others in terms of compatibility issues with the client code. Since the method signature is unchanged, client calls to it would be unaffected. However, it is now up to the client to define the engine starter and inform the Car class about it, so functionality is at risk of being lost.

In addition the potential null dereference problem becomes more pronounced. Whereas before clients may or may not have passed in a null dereference, now, the reference is null by default. In some ways, this worse as the code will compile without issue but will throw runtime exceptions. Some refactoring tools may allow the user to specify a sensible default for the new class method, and doing so manually is not a substantial effort.

Before:

```
public class Car
{
  private Engine _engine;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter = new EngineStarter();
    myEngineStarter.Start(_engine);
  }
}
```

After:

```
public class Car
{
  private Engine _engine;

  private EngineStarter
_engineStarter = null;
  public EngineStarter Starter {
get; set; }

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
     _engineStarter.Start(_engine);
  }
}
```

Figure 9

4.3 Extract Method

This refactoring allows the user to select a chunk of code inside of a method and create a new method with it. This is generally useful for addressing overly long methods or methods that violate the single responsibility principle.

This refactoring is relatively low risk. Since a new method is being created, there is little risk for compilation error or even null dereferences. Some refactoring tools and IDEs are even sophisticated enough to figure out what parameters and return value, if any, are needed by the new method.

Before:

```
public class Car
{
  private Engine _engine;
  private OnboardComputer _computer;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter =
new EngineStarter();
    myEngineStarter.Start(_engine);

    _computer = new
OnboardComputer();
    _computer.Boot();
  }
}
```

After:

```
public class Car
{
  private Engine _engine;
  private OnboardComputer _computer;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter =
new EngineStarter();
    myEngineStarter.Start(_engine);

    StartComputer();
  }

  private void StartComputer()
  {
    _computer = new OnboardComputer();
    _computer.Boot();
  }
}
```

Figure 10

4.4 Extract Interface

This refactoring involves creating an interface based specifically on an existing class. The original class is modified only to implement the new interface and the interface is created with all of the public methods of the original class. This refactoring carries almost no risk. The functionality of the existing class is not altered in any way.

Before:

```
public class Car
{
  private Engine _engine;
  private OnboardComputer _computer;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter = new EngineStarter();
    myEngineStarter.Start(_engine);

    StartComputer();
  }

  public void StartComputer()
  {
    _computer = new OnboardComputer();
    _computer.Boot();
  }
}
```

After:

```
public class Car : ICar
{
 private Engine _engine;
  private OnboardComputer _computer;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter = new EngineStarter();
    myEngineStarter.Start(_engine);

    StartComputer();
  }

  public void StartComputer()
  {
    _computer = new OnboardComputer();
    _computer.Boot();
  }
}

public interface ICar
{
  void Start();
  void StartComputer();
}
```

Figure 11

4.5 Replace Class Member (setter/getter) with Constructor Setter

This refactoring involves taking a class variable, accessible via getter/setter (or possibly just an internal class variable) and setting it via constructor injection. It is not generally automated by any tools, to the best of our knowledge, and it is very error prone compared with other refactorings. If the accessor/mutator methods (properties in C#) are removed, this will break any client code that uses them. Additionally, the modification to the class's constructor will also create a need for manual refactoring in many cases (even assuming that the constructor calls are automatically seeded with some default value for the new parameter).

Before:

```
public class Car
{
  private Engine _engine;


  private EngineStarter
_engineStarter = null;
  public EngineStarter Starter {
get; set; }


  public Car(Engine engine)
  {
    _engine = engine;
  }


  public void Start()
  {
     _engineStarter.Start(_engine);
  }
}
```

After:

```
public class Car
{
  private Engine _engine;


  private EngineStarter
_engineStarter = null;


  public Car(Engine engine,
EngineStarter starter)
  {
    _engine = engine;
    _engineStarter = starter;
  }


  public void Start()
  {
     _engineStarter.Start(_engine);
  }
}
```

Figure 12

4.6 Replace Method Variable with Mutator Setting

Here, a method parameter is taken and promoted to a class level variable, accessible by getter and setter. This facilitates dependency injection without disturbing the existing code base too much.

However, there is a lot of potential for problems. While promotion to a class variable with accessor/mutator does not disturb any method signatures, existing code is still likely to break. Consider that the Start() function will now contain a null dereference. This could be addressed by setting the engine starter to a sensible, non-null default or by putting a check in for null in Start(), perhaps throwing an exception if the starter is null. Still, it is generally going to be necessary for clients to provide information about the engine starter to the car class – after all, that is the idea behind moving toward dependency injection.

Before:

```
public class Car
{
  private Engine _engine;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter =
new EngineStarter();
    myEngineStarter.Start(_engine);
  }
}
```

After:

```
public class Car
{
  private Engine _engine;
  private EngineStarter
_engineStarter;
  public EngineStarter Starter { get
{ return _engineStarter; } set {
_Engine

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    _engineStarter.Start(_engine);
  }
}
```

Figure 13

4.7 Replace New Method Call with Interface Class Parameter Passed via Constructor Injection

This refactoring involves replacing an inline parameter with a dependency injected interface. This combines the smaller refactorings of promoting to a class member, creating a dependency injecting constructor and extracting an interface, as well as some manual work.

Obviously, there is quite a bit of manual work here, and automating this would not be trivial. However, when all is said and done, the degree of invasiveness is not any higher than simply creating a dependency injected constructor. As mentioned earlier, interface extraction/implementation is a non-invasive refactoring. Likewise, replacing a concrete class member with an interface preserves functionality. So, manual work is involved here, but the effort and reasoning are not particularly complex. The resulting code base is substantially more testable and flexible.

Before:

```
public class Car
{
  private Engine _engine;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter =
new EngineStarter();
```

```
    myEngineStarter.Start(_engine);
  }
}
```

After:

```
public Interface IEngineStarter
{
  void Start(Engine e);
}

public class EngineStarter : IEngineStarter
{
  public void Start(Engine e)
  {
    ...
  }
  ...
}

public class Car
{
  private Engine _engine;
  private IEngineStarter _starter;



  public Car(Engine engine, IEngineStarter starter)
  {
    _engine = engine;
    _starter = starter;
  }

  public void Start()
  {
    _engineStarter.Start(_engine);
  }
}
```

Figure 14

4.8 Replace New Method Call with Interface Class Parameter Passed via Setter Injection

This is the same idea as the previous example, except that the dependency injection is setter injection. As with previous comparisons of refactoring to constructor versus setter injection, the upside is that this is non-invasive in terms of the code compiling. The down side is the chance of null dereferencing and the need to handle it.

Before:

```
public class Car
{
  private Engine _engine;

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    EngineStarter myEngineStarter = new EngineStarter();
    myEngineStarter.Start(_engine);
  }
}
```

After:

```
public Interface IEngineStarter
{
  void Start(Engine e);
}

public class EngineStarter : IEngineStarter
{
  public void Start(Engine e)
  {
    ...
  }
  ...
}

public class Car
{
  private Engine _engine;
  private IEngineStarter _starter;
  public IEngineStarter Starter { 
get { return _starter; } set { 
_starter = value; } }

  public Car(Engine engine)
  {
    _engine = engine;
  }

  public void Start()
  {
    _engineStarter.Start(_engine);
  }
}
```

Figure 15

5. TESTABILITY METRIC

This section describes the testability metric that we created and explains our justification for its particular content. It is based on our categorization in the previous two sections of dependency problems and the reason that certain refactorings help eliminate them.

5.1 Starting to Quantify Testability

After looking at the various dependency problems that hurt testability and potential solutions, some patterns emerge. Generally speaking, classes with more dependencies are more difficult to test. Additionally, classes that depend on static or global state are extremely difficult to test, while classes with injected dependencies or, better yet, injected interfaces, are easier to test.

Ther reason that we feel comfortable in making this parallel between flexible and decoupled design and ease of testing is that a manually written unit test is simply a use of a given class. The standard methodology for creating manually written tests is to create a series of methods that instantiate the class, perform one or more operations on it, and then make some assertions about the state of the class. Conventionally, the layout of unit testing projects is to have a class MyClassTest.cs for each class MyClass.cs. The test methods in MyClassTest.cs test only the behavior of MyClass.cs and not its dependencies, which are either frameworks, external classes, or other classes from the SUT that have their own tests.

Dependencies cause this paradigm to become more complicated. In order to ensure that the class under test is the only entity whose behavior is being tested, it is important to be able to cause the dependencies to be in a desired state. For instance, if a class Car throws an exception when its internal Engine's "state" variable is set to "NonFunctional," it is necessary to be able to create that condition when testing Car. If Engine is an injected dependency, this is simple to do. If Engine is a hidden, inline dependency, the only way to affect this behavior, if this is possible at all, is to manipulate Car until it causes its internal dependency to be in this condition. This sort of manipulation tends to be difficult and discourages testers.

Injected interface dependencies tend to be the most flexible. The reason for this is that, using the previous Car and Engine example, if a Car is given an IEngine and Engine implements IEngine, the test class can create its own implementation of IEngine and thus exert complete control over the interaction between its Engine and Car class. A class depending only on interfaces can be manipulated in any conceivable way by the test system.

On the opposite end of the spectrum, classes with static state create a cascading problem in the opposite direction. Testing a class with static state requires recreating the static state for the purpose of the test. If many classes contribute to static state, then it becomes nearly impossible to faithfully recreate that state for the class under test. The test assertions become rather meaningless as static state may be changing in as granular a fashion as while a method on the class is being executed or even during a property access. In an extremely interdependent application with static state, testing a class is virtually impossible without simply running the entire application.

Based on this reasoning about the usability of classes in a system by other classes, we composed the following list of guidelines. These dictated the generation of our testability metric.

1. The most testable class is one that has no objects at all, though such a class may be of limited usefulness in practice.
2. A class that receives all of its object instances from other classes is highly testable.
3. A class that instantiates its own objects is more difficult to test.
4. A class that instantiates objects which, in turn, instantiate their own objects becomes exponentially more difficult to test.
5. Static state introduces dependencies not just on other classes but on the order of operations of other classes.
6. A class that contains inline references to static state and/or singletons is likely to be *combinatorially* more difficult to test.
7. A class with interface members (not hard-coded to an instantiated implementer) has dependencies that can be mocked.

5.2 Scoring Testability

Based on our observations in 5.1, we created a metric that would provide us with a "testability score." A class with a low testability score is unlikely to be unit tested, or if it is unit tested, it is unlikely to be tested correctly, in a way that creates confidence in the results. After all, it is possible to write unit tests that do little or nothing, and it is likewise possible to achieve some degree of code coverage even with weak or meaningless assertions. Consider a class that contains 100 objects that it instantiates and multiple references to singletons or static state. In order to unit test this class, it would be necessary to create the static state (which means instantiating all classes that participate in the static state) and to allow the class under test to create all of its dependencies (which may involve including lots of additional references in the test project). The amount of effort required to recreate the SUT's static state may involve be comparable to the amount of work required to write the SUT itself. Developers are unlikely to do this, or, if they try to, they may just wind up creating integration tests rather than unit tests. Additionally, even if they do create an instance of the static state sufficient enough for the class under test not to throw exceptions or give meaningless results, they are now tasked with the overwhelming prospect of creating tests that vary all of that static state sufficiently.

On the opposite end of the spectrum, consider a class that has one interface that is injected into its constructor. This class would be highly testable as all of its operations either depend on nothing but itself or an object that can be created/mocked by a tester to behave in any way desired. This class is much more likely to be tested, as testing it is easy and the tests are much more likely to be meaningful, as the tests involve nothing besides the interaction of the class under test with its dependency or simply the behavior of the class under test.

5.3 A Concrete Metric

After consideration of the above, we propose the following metric for the testability of a class. Let C be a class, with its total dependencies represented by Cn. A dependency, for our purposes, is a non-literal object from the SUT not contained in the class. In general, it may be useful to consider objects from other assemblies or even objects from the .NET framework, but we are not in a position to evaluate those, and since the testability score is relative, disregarding these criteria will skew the results only slightly, if at all. Also, it is possible to use a mocking framework[1] to neutralize dependencies on framework or external classes. This would be a good candidate for future work or discussion.

In addition to C, the class, and Cn, the number of total dependencies in C, we define Cf, Cj, and Cs, the number of dependencies on, respectively, interfaces, injected objects, and objects containing static, global state. The sum of these will always be less than or equal to Cn. Also, let N equal the total number of classes in the SUT. Our metric, then, is as follows:

$$T(C) = \frac{1+5C_f+4C_j}{1+5C_n \cdot Log_2(C_n+1)\cdot(C_sN)!} \cdot \prod_{D\in C_n} T(D)$$

Figure 16

The rationale for this is as follows. A class's testability score will vary directly with the ratio of interfaces to total dependencies and also, to a lesser extent, with the ratio of injected dependencies to total dependencies. A class's score will vary inversely with total number of dependencies and with the ratio of non-interface/non-injected dependencies to total dependencies. It will also vary inversely, but in a combinatorial fashion, with the dependence on static/global state. Finally, a class's score will vary directly with the scores of its dependencies.

We chose the coefficients for the linear variance in such a way as to reflect our hypothesis on how easy it is to test classes with injected interface dependencies versus classes with injected concrete dependencies versus inline dependencies. The log base 2 coefficient reflects our hypothesis that the testability of a class degrades with more dependencies but not so much as to deserve linear consideration (i.e, testing a class with, say, two injected interfaces is not necessarily twice as difficult as testing a class with one injected interface).

As for the factorial appropriation to dependence on global, static state, the nature of such a dependency causes the class with the dependency to potentially be dependent on every other class in the SUT with access to the global state. This global, static state can be in the form of singletons or static classes that maintain state. In and of itself, this would create a simple dependency on every other class, meaning that we would simply have N dependencies for the class C. however, the nature of static state is such that not only to the other classes in the SUT affect it, but the order of instantiation affects it as well, suggesting order combinatorial. In general, we feel that this is effective at conveying and quantifying the devastating effect that static, global state has on testability.

A class's testability score will range from 0<T(C)<=1.

5. EVALUATION

To evaluate our metric, we chose the following testing paradigm. A class is selected, and its score, T(C), is calculated. From there, a test subject is given a fixed amount of time to write unit tests, with the goal being to achieve as much code coverage as possible while using meaningful asserts. Meaningful, in this context, implies that the unit test actually verifies some property or behavior of the class under test. The amount of coverage, C(C), is then recorded. This serves as the control result or baseline.

With the baseline established, we refactor the code toward looser coupling and a better T(C) score. An improved score is verified following the refactoring, the same amount of time is given for unit testing, and coverage assessment is re-evaluated. The new C(C) is then compared with the original coverage score as a function in the improvement in T(C). We claim that T(C) will vary directly with C(C).

For the purposes of our evaluation, we experimented on three code bases. The first is a proof of concept that we set up for experimental purposes. The second is an open source CMS system called Rainbow written in C# .NET 2.0. the final project is a proprietary application written in C# .NET 3.5 using WPF.

To conduct the testing experiments, we used the following tools:

- Visual Studio Express 2010
- Visual Studio Professional 2005
- NUnit v2.5.7 (Note, we were using 2.5.8, but it has a known integration issue with NCover)
- MS Test
- NCover v3.4.14.6908x86 (Trial Version)

Visual Studio was the IDE, NUnit/MS Test was the testing framework, and NCover provided results on code coverage. The proprietary application was ported to Visual Studio 2010 and .NET 4.0. To eliminate any impact the performance of the tools would have, we stopped our timing during the running of the tests and of the code coverage tool so that the hardware capabilities of the PCs being used and the execution speeds of the tools were irrelevant.

In each of the following evaluation charts, T(C) represents testability score, C(C) represents code coverage, Cf represents injected interface dependencies, Cj represents injected non-interface dependencies, Cs represents static state dependencies, Cn represents all dependencies, and T(d) represents the muliplicative combined score of the class's dependencies. Also to note is that any aggregation of dependencies, such as an array or collection, is considered a single dependency.

For the purposes of our evaluation, we set the time limit for our unit test coverage to 15 minutes in all cases.

6.1 Simple Candidate Evaluation

For the simple candidate evaluation, we explain the entire refactoring process, since the application is small. This helps illustrate the entire process of making an application or a module more testable on the whole. Here is the state of the selected classes from the application prior to refactoring.

Class	T(C)	C(C)	Cf	Cj	Cs	Cn	T(d)	Depends On	Comments
SongFilter.cs	.000	20%	0	1	0	1	.000	Song.cs	This class is well formed, but its only dependency references static state.
SongSorter.cs	.000	21%	0	1	0	1	.000	Song.cs	Also well formed but dependent on static state.
Playlist.cs	.000	24%	0	1	1	2	.000	Song.cs Logger.cs	Spent most of 15 minutes trying to figure out how to test without log file entries being written.
Song.cs	.000	0%	0	0	1	1	.000	Logger.cs	Instantiating this object causes writing to a file.
FileReader.cs	.000	14%	0	1	2	3	.000	Logger.cs FileReader.cs Playlist.cs	This class is a singleton - we can only instantiate it without file I/O
Logger.cs	.000	40%	0	0	1	1	.000	Logger.cs	We can actually run a decent amount of this class without incurring file I/O
MainWindow.cs	.000	0%	0	2	2	4	.000	FileReader.cs Logger.cs Playlist.cs SongsWindow.cs	This class is hopelessly intertwined with global state and File I/O
SongsWindow.cs	.000	0%	0	3	2	5	.000	FileReader.cs Logger.cs Song.cs SongSorter.cs SongFilter.cs	Also hopelessly intertwined with File I/O

Interestingly enough, all of the classes in the SUT have a testability score of 0. Many developers might see this (or see the SUT itself) and conclude that unit testing and loose coupling is an impossibility. However, it is interesting to note that several of the classes would have a decent to excellent score if not for depending on a class that has a poor score. For instance, SongFilter.cs and SongSorter.cs would have a score of .833 if not for the poor score of the Song.cs class.

It is our general experience with software development and unit testing that this sort of situation is more the rule than the exception. Quite frequently, there is a fine line between loosely coupled, good candidates for test, and tightly coupled, untestable classes. Often, especially when static state is involved, this can hinge on as little as a single line of code.6.1.2 Refactoring to Testability

We need to focus on refactorings that could improve the testability score. SongFilter and SongSorter both seem like good candidates to start as low hanging fruit. Neither of them depends on static state directly, and they each have only one dependency. One good refactoring pattern that would increase the score of SongFilter dramatically, for example, would be to create an interface for its dependency. Interfaces, by definition, have a T(C) of 1. The reason for this is that they have no implementation and thus do not depend on anything. So, by substituting an interface for a direct dependency, classes with an otherwise good score but being dragged down by the scores of their dependencies can eliminate this problem.

So, if we create an interface for the Song.cs class and have Song implement it, and then we replace Songs with ISongs in the SongSorter class, we get the following result for SongSorter.cs.

Class	**T(C)**	**C(C)**	**Cf**	**Cj**	**Cs**	**Cn**	**T(d)**	**Depends On**
SongSorter.cs	1.000	94%	1	0	0	1	1.000	ISong.cs

Figure 18

Nearly complete coverage was easy to achieve with an interface in 15 minutes by creating a mock Song class that did not depend on static state. Complete coverage could have been achieved in another 2 or 3 minutes.

Another approach that is relatively straightforward here is to fix the Song.cs class in terms of testability score. SongSorter (pre-refactoring) and SongFilter both have it as the only dependency, and

Class	**T(C)**	**C(C)**	**Cf**	**Cj**	**Cs**	**Cn**	**T(d)**	**Depends On**
SongFilter.cs	.833	100%	0	1	0	1	1.000	Song.cs
Song.cs	1.000	100%	1	0	0	1	1.000	ILogger.cs

Figure 19

their scores are affected by its dependence on static state. The easiest way to refactor the Song class to improve its score would be to replace its inline dependence on the logger with an injected dependency on the logger interface. We opted to use setter injection rather than constructor injection, to preserve the functionality of other client classes. Doing so has the result shown in

Figure 18 for the Song and SongFilter classes.

With the ability to mock out the logger class, the song class becomes very easy to cover completely. The song filter class is also easy enough to cover completely now that its dependency doesn't reference static state.

The playlist class is similar to the song class, in the way that it depends on the logger. If we apply the same refactoring to the playlist class as the song class, vis a vis the logger, we achieve the following (figure 19):

Class	T(C)	C(C)	Cf	Cj	Cs	Cn	T(d)	Depends On	Comments
Playlist.cs	.593	82%	1	1	0	2	1.000	Song.cs ILogger.cs	Still some room for improvement

Figure 20

While we do see substantial improvement from the class coverage as compared to when it had a static state dependency, we can realize further improvement by refactoring from a dependency on the Song class to the previously created ISong interface. The results of doing so are as follows (figure 20):

Class	T(C)	C(C)	Cf	Cj	Cs	Cn	T(d)	Depends On	Comments
Playlist.cs	.653	100%	2	0	0	2	1.000	Song.cs ILogger.cs	As well as can be done with 2 dependencies

Figure 21

At this point, the easiest things to tackle next might seem to be the file reader and the logger, since they have fewer dependencies than the window classes. However, since FileReader and Logger both self-depend by virtue of creating static, global state, this is not the case. Before singletons/static classes can be refactored to be more testable, everything depending on them must be decoupled. So, we turn our attention next to the window classes.

First, in looking at SongsWindow.xaml.cs, we already have the logic in place for replacing its dependencies on Song and Logger with dependencies on ISong and ILogger respectively. After we replace those dependencies with publicly settable interfaces, we can address SongSorter and SongFilter. These are currently inline dependencies which hurt the testability score.

A better implementation would be to create interfaces for filtering and sorting and have these injected into the constructor of the class. Finally, we need to get rid of the dependence on the FileReader singleton.

The same principle can be applied here - interfaced constructor injection. We create an IFileReader and inject it into the SongsWindow class. Now, the SongsWindow class scores as follows (figure 21) :

Class	T(C)	C(C)	Cf	Cj	Cs	Cn	T(d)	Depends On	Comments
SongsWindow.cs	.440	67%	5	0	0	5	1.000	IFileReader.cs ILogger.cs ISong.cs ISongSorter.cs ISongFilter.cs	A lot of dependencies drag down the score, but there is real progress here

Figure 22

Coverage is comparably difficult here. Not only do we have we have several different dependencies to contend with and mock up, but we also are dealing with a class that has an XAML component, which further inhibits code coverage in harnesses. Still, we are able to test some of this class, which is a large improvement.

Next up is the main window. The main window poses some interesting problems since it is the entry point for the application. Any application will require concreteness at some point, and the main window is where we require it, since we've abstracted it out of the other classes. Specifically, the main window is where we will need a concrete logger and a concrete file reader. Since we can't instantiate them for the time being because they're singletons, we'll settle for creating a reference to the singleton instance in one place and replacing it in the rest. With this in place, we can then remove the singleton implementation from our other classes and repair the break that this causes in that one place.

Now, because we've left the singleton implementation in place, and the singleton implementations cause file I/O, we still cannot unit test MainWindow at this point, despite the fact that we've improved its score by replacing concrete dependencies with interfaces. So, now is the time to kill the singletons and make FileReader and Logger more testable. The only dependency left on their singleton status is in the constructor of MainWindow.

To eliminate the singletons, the first thing we do is make their constructor public and remove their static instance methods. Any initialization logic not in the constructor is put into the constructor. Next, we treat the former singletons like any other class – we improve their score. The logger has no dependencies on other classes in the SUT, but the FileReader does depend on the logger. So, we make the Logger an interfaced, setter injected parameter for the file reader and set it where the logger is instantiated. It is worth noting that the benefit of having written our unit tests for most of the rest of the application no comes very much in handy as we make changes that could cause potential regressions.

With the singleton apparatus dismantled, we can now finish improvements to MainWindow and test the former singletons as well as MainWindow. First, we add a new constructor to MainWindow that takes an ILogger and IFileReader and use constructor chaining to default to the concrete implementations. This allows test classes to mock these components, but defaults them for the production version. Then, we write our unit tests, and the remaining three classes now have the following results:

Class	T(C)	C(C)	Cf	Cj	Cs	Cn	T(d)	Depends On	Comments
FileReader.cs	.388	38%	1	1	0	2	1.000	ILogger.cs	This still depends on Playlist, a concrete injected dependency, and playlist does not have a perfect score
Logger.cs	1.000	40%	0	0	0	0	1.000		No improvement in coverage only because of the file I/O constraint
MainWindow.cs	.121	53%	2	2	0	4	.261	IFileReader.cs ILogger.cs Playlist.cs SongsWindow.cs	Can now test all but GUI interaction with user

Figure 23

Now, there are other minor modifications possible to further enhance testability, and there design modifications that could be made to eliminate some dependencies altogether. In the case of the former, we feel that what we have done here is a rather faithful representation of a practical refactoring effort and that it won't be perfect or ideal when this is done in the field. In the case of the latter, optimizing program architecture is beyond the scope of our work.

After all refactoring is complete, here are the final results of the refactored code base, including before and after T(C) scores.

Class	Original T(C)	Original C(C)	Refactored T(C)	Refactored C(C)	Comments
SongFilter.cs	.000	20%	.833	100%	This class now contains only one interfaced, injected dependency
SongSorter.cs	.000	21%	1.000	94%	This class has more operations and so could not quite be covered.
Playlist.cs	.000	24%	.593	82%	
Song.cs	.000	0%	1.000	100%	
FileReader.cs	.000	14%	.388	38%	Because of its file I/O, complete coverage was not possible
Logger.cs	.000	50%	1.00	40%	Because of its file I/O, complete coverage was not possible
MainWindow.cs	.000	0%	.121	53%	Lots of dependencies, but very simple logic
SongsWindow.cs	.000	0%	.440	67%	This class has a GUI component that makes testing difficult, but a lot of dependencies made it difficult as well

Figure 24

What becomes apparent is that there does seem to be a correlation between T(C) and coverage achieved. In fact, for each .1 of T(C) improvement, on average, C(C) increased by 12.6%.

6.2 Rainbow Evaluation

Our first step in extending our metric beyond the bounds of academia was to identify an open source project which we could evaluate in the same vain in which we'd evaluated our POC. To do this, we located a C# CMS project called Rainbow which contains dependencies of the nature we target. We identified classes with a variety of dependencies and then created a testing framework. Several things became apparent immediately. Firstly, our metric is designed for classes which are not only written in C# but also follow the basic constructs associated with the object-oriented coding paradigm. A code base which carries the intents of procedural programming into an object-oriented project retains the tight coupling and rigidity which object-orientation is designed to eliminate. Writing unit tests for the collections of many long methods carrying out several tasks which assume the guise of a class would be tantamount to trying to test bits and pieces of a procedural project. While the root of the obstacles is a tangle of dependencies, the refactorings we'd chosen would need to be implemented to a degree such that we'd essentially be converting the whole system into an object-oriented model, a task which we were not prepared to undertake at this time. Additionally, we noted that the layers of dependencies embedded into a commercial system include components whose functionality, even for the purpose it serves in the directly in the class, cannot properly be mimicked by mock objects. Finally, dealing with the various IDEs and testing frameworks required by the application displayed the difficulty of porting between applications and in general applying a generic process to very real, specific instances. In deference to challenges encountered, we devised the following strategy for assessing the Rainbow codebase.

1. We identified several classes as containing dependencies on static state or on other classes in the SUT.
2. For each, we computed, as reliably as we could, T(C). The obstacle here was long chains of dependencies, as the testability of a class depends on the testability of its descendants. In addition, we encountered several cyclic dependencies which further complicated the process.
3. We then refactored each class, and in some cases, we chose to modify functionality for the purposes of testing. We feel that this is justified, as all it indicates is that the refactorings detailed in the paper may need complex application if the target project is written in the procedural model. The evaluation is not affected by these modifications, as the evaluation simply compares code coverage in respect to testability. Technically, two totally different classes could be compared, provided they have significantly different testability scores.
4. Each class was then re-evaluated for T(C) and C(C), but due to integration issues between .NET and testing frameworks, some C(C) values were calculated by hand, leaving a margin of estimability in

the results. What is presented here is a result of these efforts.

Another thing worth noting is that we used MS Test and its coverage metrics for this evaluation rather than NUnit/NCover, since

Class	Original T(C)	Original C(C)	New T(C)	New C(C)	Comments
MailHelper.cs	0	0			
HttpUrlBuilder.cs	0	0	0.55	90%	
EmailAddressList.cs	1	100%	1	100%	This class is fully testable and probably not a good candidate
ThemeManager.cs	0	0	1	35%	This class has File I/O that will limit testability
LanguageGrid.cs	0	0	0.55	70%	Some UI stuff that's probably hard to reach
UploadDialogTextBox.cs	0	0			This is inheriting a text box. Wouldn't be testable anyway
HTMLEditorDataType.cs	0	0	0.11	20%	This is a nasty class with few access points
Page.cs	0	0			
LogHelper.cs	0	0	1	100%	This is an excellent candidate
SimpleScheduler.cs	0	0			Not a good candidate - this is handling thread scheduling
BagFactory.cs	0	0			

Figure 25

this is what currently accompanies the proprietary code base. We don't consider the difference to be worth accounting for in any way. The evaluator is fluent in using both sets of test/coverage tools and they are fairly similar. Additionally, the T(C) does not depend on the testing tools and the purpose of our evaluation is to compare C(C) before and after a refactoring. So, as long as the same tool is used for the before and after C(C), we do not believe it matters.

6.3 Proprietary Evaluation

Because this is proprietary source code, we leave out the names of the classes in the final version. We do offer descriptions of the dependencies before and after, however, for proper evaluation consideration.

Additionally, as compared with the simple candidate, the scores tend to be a lot lower for both T(C) and C(C). We attribute this to the complexity of production code in a mature application versus a small example.

Here, we have the state of the SUT prior to any refactoring.

Class	T(C)	C(C)	Cf	Cj	Cs	Cn	T(d)	Depends On	Comments
Class1.cs	.000	5%	0	2	1	3	.000	Singleton class, two normal classes	Bad score resulting from singleton dependency
Class2.cs	.000	12%	0	2	0	4	.000	Class1, other classes	Bad score caused by Class1
Class3.cs	.000	28%	0	2	0	2	.000	Class1, other classes	Zero score caused by dependence on Class 1
Class4.cs	.000	8%	0	3	0	3	.000	Three normal classes	One dependent class has a singleton reference
Class5.cs	.000	14%	0	0	1	5	.000	Lots of things	Tough to refactor. Entry point for a subsystem
Class6.cs	.000	56%	0	2	0	3	.000	Three normal classes	One dependency directly references a singelton
Class7.cs	.000	0%	0	1	0	2	.000	Two normal classes	One dependency references a singleton
Class8.cs	.000	4%	0	2	0	3	.000	Three normal classes	One dependency references a singleton

While it may seem improbable that all of the classes we chose for refactoring have initial scores of zero, in reality, we believe this is common in code bases that make heavy use of static/global state and/or the singleton design pattern. Global state tends to creep into many classes, resulting in a code base where, even if one writes a class that does not actually depend on global state, one of the dependent classes does.

It seems that there is a bit of noise for classes with a score of zero. We attribute this to the fact that some classes that are linked into static state have it relatively isolated in their methods. In looking at Class 6 in particular, there is a class field that houses a static state reference, but it is used in less than half of the class methods. It also seems that, while static state clearly has a negative affect on testability, not all static state is created equal. This code base has extensive dependency on global/static state, but some of the classes that reference it do not in and of themselves inhibit testability. For instance Class 3 depends a great deal on a class with a singleton reference, but many of Class 3's methods that operate on that dependency do not cause it to reference its singleton.

Another thing to note about this particular code base is that the singletons use lazy instantiation and do not gracefully handle dependent state not existing. In other words, if one uses a class that refers to a singleton, the first reference to that singleton triggered by a test method causes the singleton's instantation logic to proceed, cascading with that of any other singletons it depends upon. This results in a nasty, unpredictable web of dependencies of all external resources the application uses. For example, just instantiating Class 7 somehow results in code searching for XML files, attempting database connections, and modifying the global exception handling scheme of the WPF framework.

However, in applications where global state is diligently managed, it may not always be the case that global state has this dramatic an effect. However, in our personal experience, global state is quite degenerative and thus it is only a matter of time before testability rots.

6.3.1 Refactoring to Testability

Unfortunately, we cannot really go into specifics here, but we can describe some general approaches. In some cases, we made modifications to the design that were not purely for the purpose of testability. That is, because this is production code, some of these refactorings also simplified and improved the design and object relationships to a more decoupled state. While this is a goal in and of itself, it does tend to have a positive impact on T(C), and so it is worth including it in the study, even if the primary objective was not to improve T(C).

The main approach that we did take specifically to allow better unit test coverage and thus improve T(C) was to create an interface for any class that has a reference to static state and to code to that interface. This allowed for the creation of mock objects and thus dramatically improved testability. It is also starting to pay reuse dividends in the code base, as use of these interfaces appears to be taking off.

Another approach that frequently came in handy was turning inline dependencies into injected dependencies. We did this wherever it fit in appropriately with the design, as it is not department policy to modify production code specifically with a mind toward testing. That is, if it is deemed inappropriate to expose a class's internals, we will not do that simply so that the class becomes easier to test.

The final approach that seemed to help and does not necessarily have an effect on the T(C) score as we have created it was to localize dependencies. In some cases, moving something from a class field to a single method parameter had a positive effect on coverage. In some cases, this improves T(C), such as if the class member was instantiated somewhere in the class logic. In other cases, it does not improve T(C) -- cases where the dependency was previously injected into the class and is now injected into a method.

Another interesting thing to note is that in the refactoring of the module in which these classes resided, we were able to create a situation where classes were not hindered by the scores of their dependencies. That is, these represent the 'interesting' classes that do the most work. In the case of most other processing classes, we were able to create dependencies exclusively on interfaces, or have a dependency only on the class under test.

Pictured here are the final results of the refactoring.

Class	**Original T(C)**	**Original C(C)**	**Refactored T(C)**	**Refactored C(C)**	**Comments**
Class1.cs	.000	5%	.404	47%	Replaced singleton with an interface, singleton was complicated so mocking took most of test time.
Class2.cs	.000	12%	.404	53%	In this case, we refactored a singleton to an interface and was also able to remove a dependency
Cass3.cs	.000	28%	.910	96%	This was a relatively simple class - coverage was originally impeded because of the singleton dependency causing exceptions in most methods
Class4.cs	.000	8%	.566	73%	This class was hindered mainly by a dependency reference to a singleton. Somewhat complicated testing this class as its at the third level of an inheritance hierarchy.
Class5.cs	.000	14%	.102	45%	Eliminating singleton via interface was a help, but this is a class with a GUI component, so full coverage is not expected with or without timeframe.
Class6.cs	.000	56%	.545	100%	Straightforward class that was easy to cover once it no longer depended on static state.
Class7.cs	.000	0%	.545	71%	Class contains complex conditional logic, so I ran out of time going for coverage, despite no exceptions.
Class8.cs	.000	4%	.404	82%	This class contains some sanity check exception handling that is unreachable without inheriting the class.

6.4 Results Summary

One might naturally expect that each tenth of a point increase in T(C) would correspond with 10% C(C), but the results indicate a slightly greater improvement in C(C). This would seem to indicate that there may be diminishing returns at higher T(C) values. That is, all of these classes were initially bogged down by static state in themselves or their dependencies. When starting from 0, the testability improvements were greater than what would be expected mathematically, which naturally implies that higher starting scores would yield smaller comparative improvements.

This seems to line up with one's expected experience. If a class has one injected dependency and one interface dependency, it would have a T(C) score of .910, assuming that its injected dependency had a 1.0 score. Generally, this class would be very easy to test - one object can be mocked at will, and the other has no significant dependencies, so it can easily be created. There is little reason to think that creating an interface for its one simple, non-interface dependency would provide the same kind of boost to code coverage as, say, what we did to class 5, which involved moving it from 5 dependencies with one singleton dependencies to 5 dependencies with no static state. This corresponds to about the same improvement in T(C), but in the case of Class 5, we are removing a huge barrier to testability.

Viewed from another angle, T(C) varies proportionally with the ratio of injected dependencies to dependencies, but varies in more complex fashion with static state and growing numbers of dependencies. As such, we should not expect a purely linear improvement in coverage with an improvement in T(C).

It is also worth noting that we did encounter some considerations that our T(C) metric did not account for. These include lack of coverage caused by file I/O, lack of coverage caused by UI interaction, and lack of coverage due to complexity. In the case of the first two, the metric could be tweaked by adding an additional inverse component for interaction with components likely to need mocking: GUI, database, files, etc. Alternatively, it is credible to leave the metric alone and stipulate that this metric applies only to classes with no outside factors (i.e. this is for service and business layer logic as opposed to GUI management or file and database I/O).

In the case of the score variance from what was expected based on logic complexity, this likely indicates room for improvement of the metric. We believed that this was accounted for by the nature of the fact that more dependencies will tend to correlate with more code complexity. However, in the case of the SongFilter and SongSorter classes, we had the same number of dependencies and the same score, but different amounts of code coverage. This is easily attributable to the fact that the sorter has significantly more logic. Extrapolating, a class may have 5 dependencies, but absolutely trivial logic on those dependencies (e.g. a "container" class that simply groups a series of objects), making it easy to test in spite of a poor score.

7. DISCUSSION

7.1 Threats to Validity

We noticed during evaluation that other factors have some bearing on the testability of the class. In particular, the Rainbow project had modules that consisted of long methods with cryptic naming conventions, lots of conditional logic, and a procedural style of programming. So, conditional complexity and code quality will have an impact on the testability of a class, and we do not address that in T(C).

We feel that this consideration is somewhat muted by the fact that the authors (or refactorers) of the code are generally the ones who write unit tests. So unfamiliarity with or unconventional styles of legacy code are going to be mitigated to some degree. In addition, these things do not necessarily impact the work needing to be done by a tester. A particularly complicated or poorly written class may be wrong or incomprehensible in its functionality, but also easy to test. Consider, for instance, a class with nothing but void methods and no exception throwing. Testing this class would be as simple as instantiating it and calling all of its methods for total coverage. The tester does not have to prepare other objects to give it or worry about its state. As such, we feel that, while the internal operation of a class is important to how meaningful and correct the tests may be, it is not as important to their C(C).

In addition, we realize that our sample size of developers who write tests for 15 minutes before and after a refactoring is quite small (i.e. the two of us). It might be said that we aren't accounting for different testing frameworks and practices as well as different ability levels. However, we feel that this is almost irrelevant to what we are trying to measure. We are interested in the relationship between dependencies and testing difficulties. So, the thrust of our experiment has been to find a way to measure a difference between two comparable pieces of code. The coverage achieved with a given tool, framework, and developer is measured only against coverage achieved by the same tool, framework and developer with a refactored SUT. Thus, we feel that our results about how refactoring improves testability are as universally applicable as we claim.

7.2 Future Work

Given the timeframe we had to do work and the relatively limited resources to which we had access, we are encouraged by our work while simultaneously believing that there is significant room for continuation. The first and most obvious way to do this would be to expand the number of systems upon which the testability metric, T(C) is evaluated. This would help flesh out the common relationship between T(C) and code coverage in many different scenarios. Not only would this boost confidence in the metric, but it might identify nuances that vary in different environments.

Along the same lines, the metric, T(C) would be subject to tweaking based on the findings. That is, if it were determined that using interfaces and mocking actually reduced coverage as compared to object injection, the coefficients for interfaces and injected dependencies could be tweaked to line up better with the relative code coverage scores.

Another potential line of future work would be to create a static analysis tool that could assess the T(C) score of a given class for developers. This could be used to identify testing difficulties as weaknesses in a system and it could provide a means of feedback during refactoring to alert the refactorers as to whether or not their efforts were helping to promote testability. It could also be calculated during the course of development to raise flags if developers were introducing too many dependencies, becoming dependent on global state, or if their classes were testable thanks to interfacing and dependency injection.

A final project that comes to mind would be to create automated refactorings specifically along the lines of improving the T(C) score. To the best of our knowledge, existing automated refactoring efforts are dedicated to performing correct refactorings at the explicit will of the user. We do not believe that any refactoring tool currently in existence is designed to operate based on improving a statically checked metric.

7.3 Related Work

One of the more relevant related works to ours was [5]. The idea for a testability metric was intriguing to us during the early phases of exploring this topic, and their treatment of metrics for the success of automated test generators was a good baseline. We drew inspiration from [9] and the papers that it referenced as well. This paper discussed a novel approach to automated refactoring involving using language extensions to preserve invariants about the source code during a refactoring. Since the ultimate aim of this line of research is to automate as much as possible improvements to code that is difficult to test, breakthroughs in the automated refactoring process hit close to home.

There were various other works that impacted our progress somewhat less directly, but served as a helpful backdrop for the state of testing, refactoring, and mocking. These included [1], [2], [10], [13] and [11]. Looking at various dependency injection frameworks was also helpful in providing background: [14], [15], [16]. And, finally, information about existing refactoring tools ([17], [18], [19]) was helpful, as was special information about singletons and static state: [20].

8. CONCLUSIONS

To the best of our knowledge, we have presented the first concrete metric for testability as far as manual generation of automated unit tests are concerned. We feel that this metric can be used to accurately predict the level of success or frustration that will be experienced by a developer tasked with unit testing a particular class. We have presented a justification of our metric based on concrete examples, standard design and coding practices, and frequently used refactorings. Furthermore, we have presented a detailed evaluation that supports the validity of our metric.

REFERENCES

[1] J. de Halleux and N. Tillman, Moles: Tool-Assisted Environment Isolation with Closures, TOOLS ’10, LNCS 6141, pp. 253 – 270, ‘10

[2] N. Tillman and J. de Halleux, Pex – White Box Test Generation for .NET, TAP ‘08

[3] NUnit: http://www.nunit.org/

[4]MS Test: http://msdn.microsoft.com/en-us/library/ms243147%28VS.80%29.aspx

[5] B. Daniel and M. Boshernitsan, Predicting Effectiveness of Automatic Testing Tools, Agitir Software, Inc.

[6] Rainbow: http://www.rainbowportal.net/site/1/home.aspx

[7] E. Gamma, R. Helm, R. Johnson, J.Vlissides, 1994, Design Patterns (the "Gang of Four" book)

[8] Law of Demeter: http://en.wikipedia.org/wiki/Law_of_Demeter

[9] M. Shafer, M. Verbaere, T. Ekman, and O. de Moor, Stepping Stones over the Refactoring Rubicon, ECOOP '09, LNCS 5653, pp. 369-393, '09

[10] C. Pacheco, S. K. Lahiri, and T. Ball, Finding Errors in .NET with Feedback-Directed Random Testing, ISSTA July '08

[11] M. d'Amorim, C. Pacheco, T.Xie, D. Marinov, M. D. Ernst, An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing

[12] R. E. Caballero and S. A. Demurjian Sr., A Graph-Based Algorithm for Automated Refactoring

[13] Spring: http://www.springsource.org/download

[14] Guice: http://code.google.com/p/google-guice/

[15] PicoContainer: http://www.picocontainer.org/

[16] ReSharper: http://www.jetbrains.com/resharper/

[17] NDepend: http://www.ndepend.com/

[18] CodeRush: http://www.devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/

[19] Google Singleton Detector: http://code.google.com/p/google-singleton-detector/