Resolving Cyclic Dependencies – Design for Testability

Yurii Boreisha, Oksana Myronovych

Department of Computer Science, Minnesota State University Moorhead, Moorhead, MN, USA

Department of Computer Science, North Dakota State University, Fargo, ND, USA

Abstract - This paper is dedicated to the issues of resolving cyclic/circular dependencies in contemporary software systems. Interface-based and event-based approaches are the best practices to deal with dependency problems and provide for creating software that’s easier to test. The main design patterns to support these practices are the dependency injection and observer/listener. It is demonstrated how the C# mechanism based on events and delegates can be used to implement the observer/listener design pattern to resolve cyclic/circular dependencies.

Keywords: Dependencies and layering, design for testability, acyclic dependencies principle.

1 Introduction

To produce high-quality software, developers must strive to ensure that their code is maintainable and testable. The code also should be adaptive to change. Agile development, the leading trend in the system development, helps keep system development projects responsive to change [6, 10].

Agile developers follow these steps to create contemporary high quality software [3, 4, 8]:

- Detect problems by following agile practices.
- Diagnose problems by applying design principles.
- Solve problems by applying appropriate design patterns.

SOLID is the acronym for a set of principles, patterns and practices that, when implemented together, make code adaptive to change. The SOLID practices were introduced by Bob Martin almost 15 years ago:

S – Single responsibility principle: “a class should have only one reason to change”.

O – Open/close principle: “software entities (classes, modules, functions, etc.) should be open for extension, but close for modification”.

L – Liskov substitution principle: “subtypes must be substitutable for their base types”.

I – Interface segregation principle: “clients should not be forced to depend upon methods that they don’t use; interfaces belong to clients, not to hierarchies”.

D – Dependency inversion principle: “higher-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend upon details; details should depend upon abstractions”.

Even taken in isolation, each of these principles are very useful. When used in collaboration, they give the code a structure that lends itself to change. SOLID patterns and practices are merely tools for you to use. Deciding when and where to apply any pattern or practice is a part of the art of software development.

In the context of software engineering, a broadly accepted definition for testability is “the ease of performing testing”. Testing is the process of checking software to ensure that it behaves as expected, contains no errors, and satisfies its requirements. The ability to test software, and in particular to test software automatically, is an aspect of extraordinary importance because automated tests give us a mechanical way to figure out quickly and reliably whether certain features that worked at some point still work after we make some required changes. In addition, tests make it possible for us to calculate metrics and take the pulse of a project, as well. Testing is an important part of the change [1, 2, 9].

Testable software is inherently better from a design perspective. Design for testability (DfT) was adapted to software engineering and applied to test units of code through tailor-made programs. DfT defines three attributes that any unit of software must have to be easily testable: control, visibility, and simplicity. When you apply control, visibility, and simplicity to the software development process, you
end up with relatively small building blocks that interact only via contracted interfaces [5, 7, 11].

To keep the code adaptive to change and testable one must manage dependencies effectively. This applies at all levels of the software – from architectural dependencies between subsystems to implementation dependencies between individual methods.

When the dependency structure is incomprehensible, a change in one module can cause a direct side effect in another, seemingly unrelated module. There are established patterns that help you arrange your application in the short term so that it can adapt to changes in the long term. Layering is one of the most common architectural design patterns, and MVC, MVVM, Web API, etc., provide examples of widely used implementations of the layering.

The following design principles also target the proper dependency management:

- **Coupling** is a qualitative measure of how closely the classes in a design class diagram are linked. A simple way to think about coupling is as the number of navigation arrows on the design class diagram. Low coupling is usually better for a system than high coupling. In other words, fewer navigation visibility arrows indicate that a system is easier to understand and maintain.

- **Cohesion** refers to the consistency of the functions within a single class. Cohesion is a qualitative measure of the focus or unity of purpose within a single class. Classes with low cohesion have several negative effects. First, they are hard to maintain. Second, it is hard to reuse such classes. Finally, classes with low cohesion are usually difficult to understand, their functions are intertwined and their logic is complex. High cohesion is the most desirable.

- **Indirection** – is a design principle in which an intermediate class (or component) is placed between two classes (or system components) to decouple but still link them. Inserting an intermediate object allows any variations in one system to be isolated in that intermediate object. Indirection is also very useful for many corporate security systems (for example, firewalls and proxy servers).

- **Acyclic dependencies principle** states that the dependency graph of packages or components should have no cycles. This implies that the dependencies form a directed acyclic graph.

There is a strict relationship between coupling and testability. A class that can’t be easily instantiated in a test has some serious coupling problems. If the problem of coupling between components is not properly addressed in the design, you end up testing components that interact with others, producing something that looks more like an integration test than a unit test. Integration tests are still necessary, but they ideally should run on individual units of code (for example, classes) that already have been thoroughly tested in isolation. Integration tests are not run as often as unit tests because of their slow speed and higher setup costs.

By keeping coupling under control at the design phase we improve testability. Conversely, by pursuing testability, we keep coupling under control and end up with a better design for the software.

Interface-based and event-based approaches are the best practices to deal with dependency problems and provide for creating software that’s easier to test. The idea of writing code against interfaces rather than implementations is widely accepted and applied. There is a number of options to accomplish this, for example, dependency injection (DI) and inversion of control (IoC) container [6].

This paper is dedicated to the issues of resolving cyclic/circular dependencies in contemporary software systems. We discuss these issues from the point of view of the event-based approach. It is demonstrated how the C# mechanism based on events and delegates can be used to implement the observer/listener design pattern to resolve cyclic/circular dependencies.

### 2 Dependencies and Layering

Layering is an architectural pattern that encourages you to think of software components as horizontal layers of functionality that build on each other to form a whole application. Components are layered, one on top of the other, and the dependency direction must always point downward. That is, the bottom layer of the application has no dependencies, and each layer upward depends on the layer immediately below it. At the top of the stack is the user interface. If the application is a service layer, the top layer will be the API that clients will use to interact with the system.

The primary reason for using layers (and tiers) is the Separation of Concerns (SoC). As an architect, you determine which layer talks to which layer, and you have testing, code inspection, and perhaps check-in policies to enforce these rules. However, even when two layers are expected to collaborate, you don’t want them to be tightly coupled. In this regard the dependency inversion principle (DIP) helps a lot.

DIP is the formalization of a top-down approach to defining the behavior of any significant class.
method. In using this top-down approach, we focus on the work flow that happens at the method level rather than focus on the implementation of its particular dependencies. At some point, though lower-level classes should be linked to the mainstream code. DIP suggest that this should happen via injection.

From the point of view of MS Visual Studio all classes and interfaces are contained in assemblies. When correctly organized, your assemblies will contain classes and interfaces that only pertain to a single group of related functionality [6].

Groups of two or more interrelated assemblies form components of the software system that is being developed. These components interact in a similarly well-defined and structured fashion. Components are not physical artifacts of deployment, like assembly dynamic-link libraries (DLLs), but are logical groupings of assemblies that share a similar theme.

In dependency management, components are no different from other programming constructs at lower levels. As with methods, classes, and assemblies, you can consider layers to be nodes in the dependency graphs. The same rules apply: keeping the dependency graph acyclic and ensuring a single responsibility.

There are several common layering patterns from which to choose for any project. The differentiating factor between the layering patterns is simply the number of layers used. The number of layers required correlates to the complexity of the solution; the complexity of the solution correlates to the complexity of the problem.

The difference between layers and tiers is the difference between the logical organization and physical deployment of code. Logically, you could separate an application in two or more layers, but physically deploy it into one tier.

With every tier you deploy to, you accept that you are crossing a network boundary, and with that comes a temporal cost: it is expensive to cross a processing boundary within the same machine, but it is much more expensive to cross a network boundary. However, deploying in tiers has a distinct advantage because it allows you to scale your application. If you deploy a web application that consists of a user interface layer, a logic layer, and a data access layer onto a single machine – thus a single tier – that machine now has a lot of work to do, and the number of users you can support will necessary be lower. Were you to split the application’s deployment into two tiers – putting the database on one tier and the user interface and logic layers on another – you could actually scale the user interface layer both horizontally and vertically.

To scale vertically, you just increase the power of the computer by adding memory or processing units. This allows the single machine to achieve more by itself. However, you can also scale horizontally by adding completely new machines that perform exactly the same task. There would be multiple machines to host the web user interface code and a load balancer that would direct clients to the least busy machine at any point in time. Of course, this is not a panacea to supporting more concurrent users on a web application. This requires more care with data caching and user authentication, because each request made by a user could be handled by a different machine.

3 Navigation Visibility and Dependency Relationships

Navigation visibility refers to the ability of one object to interact with another object. When building navigation visibility we should transform the problem domain model into the corresponding design class diagram. Here are a few general guidelines:

- One-to-many associations that indicate a superior/subordinate relationship are usually navigated from superior to the subordinate.
- Mandatory associations, where objects in one class can’t exist without objects of another class, are usually navigated from the independent class to the dependent class.
- When an object needs information from another object, a navigation arrow might be required, pointing either to the object itself or to its parent in a hierarchy.
- Navigation visibility arrows may be bidirectional, for example when an object might need to send a message to another object as well as the reverse.

To resolve the problem of bidirectional navigation visibility arrows one can apply the observer/listener design pattern. It is a design pattern in which an object, called the publisher, maintains a list of its dependents, called subscribers (observers, listeners), and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. This pattern is also a key part in the familiar MVC architectural pattern. The observer/listener pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.

The observer/listener pattern can cause memory leaks, known as the lapsed listener problem, because in basic implementation it requires both explicit registration and explicit deregistration, as in the dispose pattern, because the publisher holds strong
references to the observers/listeners, keeping them alive.

Herewith we provide an implementation of the observer/listener pattern based on C# mechanism of events and delegates. This implementation is free from the lapsed listener problem.

Let’s create class \texttt{Account} as it’s shown in Figure 1, with the related class \texttt{AccountEventArgs} like in Figure 2. The \texttt{Subscriber} class interface can look like in Figure 3. The related \texttt{Subscriber} classes should implement this interface as it’s shown in Figure 4. The \texttt{Publisher} class may look like in Figure 5.

This implementation is used to create and agreement/contract between the \texttt{Publisher} and \texttt{Subscriber} classes by determining the signature of the corresponding event handler method. The code in Figure 6 shows the \texttt{Main} class where a publisher object and a number of subscriber objects are instantiated.

From now on all subscribers will be informed about the changes made by the \texttt{Publisher} to the object of class \texttt{Account} (they will listen for the related event, \texttt{Published}, to fire); bidirectional visibility arrows will disappear.

```csharp
public class Account {
    public string AccountNumber { get; set; }
    public override string ToString() {
        return string.Format("Account {0}", AccountNumber);
    }
}

Figure 1: Class Account

public class AccountEventArgs : EventArgs {
    public Account MyAccount { get; set; }
}

Figure 2: Class AccountEventArgs

public interface ISubscriber {
    void OnPublished(object source, AccountEventArgs args);
}

Figure 3: Interface ISubscriber

public class Subscriber1 : ISubscriber {
    public void OnPublished(object source, AccountEventArgs args) {
        Console.WriteLine("Subscriber1 getting info about {0}", args.MyAccount);
    }
}

public class Subscriber2 : ISubscriber {
    public void OnPublished(object source, AccountEventArgs args) {
        Console.WriteLine("Subscriber2 getting info about {0}", args.MyAccount);
    }
}

Figure 4: Classes Subscriber1 and Subscriber2

public class Publisher {
    public void Publish(Account account) {
        Console.WriteLine("Working with: {0}", account);
        // To do: some stuff
        OnPublished(account);
    }
}
```
public event EventHandler<AccountEventArgs> Published;

protected virtual void OnPublished(Account account) {
    if (Published != null) // any subscribers?
        Published(this, new AccountEventArgs() { MyAccount = account });
}

Figure 5: Class Publisher

class Program {
    static void Main(string[] args) {
        Account account = new Account() { AccountNumber = "12345" };
        Publisher publisher = new Publisher();

        ISubscriber sub1 = new Subscriber1();
        publisher.Published += sub1.OnPublished;

        ISubscriber sub2 = new Subscriber2();
        publisher.Published += sub2.OnPublished;

        publisher.Publish(account);
    }
}

Figure 6: Class Main

The observer/listener pattern is also used to implement the two-way data binding operations in Universal Windows Platform applications to resolve cyclical dependencies between view and domain layers. The data binding does not know when the data to which it is bound has been changed. The domain layer object needs to inform the data binding of any modifications by sending a PropertyChanged event to the view layer. This event is part of the interface named INotifyPropertyChanged, and all object that support two-way data binding should implement this interface (it is defined in the System.ComponentModel namespace) as in Figure 7.

The OnPropertyChanged method raises the PropertyChanged event. The PropertyChangedEventArgs parameter to the PropertyChanged event should specify the name of the property that has changed. This value is passed in as a parameter to the OnPropertyChanged method as it’s shown in Figure 8 for the property EmailAddress.

public class Customer : INotifyPropertyChanged {
    ...
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName) {
        if (PropertyChanged != null) {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}

Figure 7: Domain layer class Customer
private string _emailAddress;
public string EmailAddress {
    get { return this._emailAddress; }
    set {
        this._emailAddress = value;
        this.OnPropertyChanged(nameof(EmailAddress));
    }
}

Figure 8: OnPropertyChanged method call

4 Dealing with Dependencies

UML design class diagram contains the final definition of each class in the final object-oriented software system. The primary source of information for this diagram is the problem domain model. The navigation visibility graph of the design class diagram should not contain cycles.

UML package diagram is a high-level diagram that allows designers to associate classes of related groups. The classes are placed inside the appropriate package based on the layer to which they belong. Dependency relationship is a relationship between packages, classes, or use cases in which a change in the independent item requires a change in the dependent item. Dependency direction must always point downward.

As we already mentioned above the interface-based and event-based approaches are the best practices to deal with dependency problems and provide for creating software that’s easier to test. The main design patterns to support these practices are the dependency injection and observer/listener (described in the previous section).

UML diagrams free from cyclic/circular dependencies leverage the software testability. As far as testing is concerned, one could say that there are the following main types of dependencies: those that you want to resolve (like cyclic/circular ones), those that you want to ignore, and those with which you want to interact but in a controlled manner. The following technique is used for the latter case.

According to the concept of the testing in isolation, it is highly recommended to isolate the class being tested from all its dependencies. This can happen only if the class is designed in a loosely coupled manner. In an object-oriented scenario, for example, class A depends on class B when any of the following conditions are verified:

- Class A derives from class B (inheritance).
- Class A includes a member of class B (aggregation, composition).
- One of the methods of class A invokes a method of class B.
- One of the methods of class A receives or returns a parameter of class B.
- Class A depends on a class that in turn depends on class B.

To neutralize such dependencies we use test doubles. If the classes under the test support the DI, providing test doubles is relatively easy [5].

A test double is a class you write and add to the test project. This class implements a given interface or inherits from a given base class. After you have the instance, you inject it inside the object under the test by using the public interface of the object being tested.

There are two main types of test doubles: fakes and mocks. A fake object is a relatively simple clone of an object that offers the same interface as the original object but returns hardcoded or programmatically determined values. A mock object does all that a fake does, plus something more.

In a way, a mock is an object with its own personality that mimics the behavior and interface of another object. Essentially, a mock accommodates verification of the context of the method call. With a mock, one can verify that a method call happens with the right preconditions and in the correct order with respect to other methods in the class. For the level of flexibility you expect from a mock, you need an ad hoc mocking framework.

5 Conclusions

There is a strict relationship between coupling and testability. A class that can’t be easily instantiated in a test has some serious coupling problems. If the problem of coupling between components is not properly addressed in the design, you end up testing components that interact with others, producing something that looks more like an integration test than a unit test. Integration test are still necessary, but they ideally should run on individual units of code (for example, classes) that already have been thoroughly tested in isolation.
Two major recent contributions to the software testing area are the definition of new frameworks for test execution (collectively referred as xUnit) and promote shorter cycles in the testing process, such as continuous integration (CI).

The basic idea behind CI is to commit, one or more times per day, all of the working copies of the software on which different developers or groups of developers are working. CI is related to the concept of automated test execution frameworks, in that a regression test suite should be automatically run against the code (ideally, prior to commit it) to help ensure that the codebase remains stable (i.e., no regression errors have been introduced) and continuing engineering efforts can be performed more rapidly. If some test fail, the developers responsible for the change must then correct the problems revealed by the tests. This approach is advantageous because it can reduce the amount of code rework that is needed in later phases of development, and speed up overall development time [9].

This paper is dedicated to the issues of resolving cyclic/circular dependencies in contemporary software systems. We discuss these issues from the point of view of the event-based approach. It is demonstrated how the C# mechanism based on events and delegates can be used to implement the observer/listener design pattern to resolve cyclic/circular dependencies.

6 References