# How to Prevent or Remove Coupled Code

When we talk about coupling, we are talking about code coupling. Code coupling can take various forms:

1. Build coupling – coupled items must be present at build or it will fail
   1. Class to Class coupling
   2. Library (dll) to Library (dll) coupling.
   3. File and folder coupling – files and folders must exist or the build will break
   4. Pre/Post action coupling – actions must succeed or the build will break
2. Runtime coupling – coupled items must be preset at runtime or it will fail
   1. Convention coupling – A convention must be followed or the runtime will fail
   2. File and folder coupling – touching files and folders can break runtime
   3. Temporal coupling – code has to run in a certain order or the runtime will fail
   4. External system coupling – an external system must exist or the runtime will fail

Note: The above is not an exhaustive list of coupling but is sufficient for this document.

Not all coupling is bad. For example, every C# application is coupled to the dotnet framework just as every Java application is coupled to the JRE.

How can we tell when coupling is good or bad? Also, is it always black and white? Or can some coupling be better or less bad than others?

## Build-time coupling vs Run-time coupling

When an application runs, everything needed to run that application must be present. This is called run-time coupling and this is the normal expected behavior. Run-time decoupling is possible in a sense, but is more difficult to achieve without first achieving build-time decoupling.

Build-time coupling, however, is code coupled at build time. This should not require every library that an application needs. Instead, the build-time coupling should be minimized or eliminated.

An application may need 100 dlls and 1 executable to run. But each of the 100 dlls could have 0 dependencies themselves. This is most easily thought of by Legos. You can have 100 Lego pieces and build a lot of different shapes. It doesn’t make sense to glue (Kragl 😊) Lego pieces together so they are permanently connected. Similarly, it doesn’t make sense to couple code or code libraries together so they are permanently connected. Instead use them as individual Lego pieces.

## Code Coupling

What is Code Coupling? Very similar to the term “dependency” which is the list of external requirements for one piece of code to build and run.

Code coupling is any external dependency that is required for code to work. The topic of this document is to focus on what is considered “bad” coupling, which is when the coupling makes the code harder to maintain, harder to keep bug free, harder to change, etc.

Before we talk about the types of coupling, let’s discuss the “when” of code coupling.

### The “When” of Code Coupling

Code coupling is really limited to two important times for the code.

1. Compile Time
2. Run Time
   1. Running Unit Tests (Only the unit under tests should be running)
   2. Running Integration tests – A partial or full application running.
   3. Running Application in Dev/QA/Production – The full application is running.

Coupling can be compile time, run time, or both.

Why is it important to distinguish these? Because usually when devs talk about bad coupling, we are talking about build coupling. Run Time coupling is usually not a problem.

### Types of Coupling

There are many other forms of coupling. Below is a list of other forms, but probably doesn’t cover them all. Also note, that not all forms of coupling are bad.

1. Class-to-Class Coupling – Almost always bad.
2. Library-to-Library Coupling – Can be either good or bad.
3. Exe-to-Library Coupling – Completely normal and good.
4. Temporal coupling – Action A must run before action B. Can be either good or bad.  
   Note: Temporal coupling isn’t always bad coupling. For example, a class must be registered with an IoC container before IoC can resolve an instance of that class. Is that bad coupling? No. But there are examples of bad temporal coupling.
5. Knowledge coupling – A.dll needs knowledge about B.dll even though they should NOT know anything about each other. – Almost always bad.
6. File/Folder coupling – Files and folders must exist. – Can be either good or bad.
7. External system coupling – you need an actual database or web server available to run. Usually good and normal.
8. Convention Coupling – Some convention (naming scheme, file paths, etc.) must be followed for code to either build or run. Usually good coupling.  
   Note: A good convention, when well documented can solve some hard problems. Conventions usually exist to decrease coupling elsewhere. For example, C# projects have a build output folder convention, and that isn’t bad, it is good.

Mostly, we are going to talk about the coupling that is considered almost always bad.

## Class coupling

Class to Class coupling is almost always bad. By bad, we mean code that is difficult to unit test, has increased cyclomatic complexity, and is less maintainable, causes bugs, cannot be closed for change, cannot be easily extended, cannot be injected, is not Single Responsibility. Or better said, class coupling quickly becomes problematic code.

This is very simple to prevent. There is a 1-step prevention process:

1. Use interfaces.
   1. Never have a class depend on another class. Instead, only have a class depend on interfaces.

### Class coupling - Bad Example

The below code is an example of bad coupling because class A is directly dependent on class B.

public class A  
{

private readonly B \_b;  
 public A(B b) { \_b = b; }  
}

public class B { /\*…\*/ }

Tight coupling leads to difficult to maintain code, increased complexity in unit tests, and many other issues that slow down development, which results in higher development costs.

A unit test by definition tests 1 unit. Any unit test that tests class A is prevented from testing 1 unit as it must test class B as well.

### Class coupling - Good Example

The below code is an example of good coupling because class A is not dependent on class B. Instead, class A only depends on an interface.

public interface IA { /\*…\*/ }

public class A : IA  
{

private readonly IB \_b;  
 public A(IB b) { \_b = b; }  
}

public interface IB { /\*…\*/ }

public class B : IB { /\*…\*/ }

The above is an example of good coupling because class A is not directly dependent on class B.

A unit test, by definition, should only test 1 unit. Any unit test that tests class A is properly only testing 1 unit, class A, because no other class is a dependency. As an interface, IB can be mocked or faked, making it very simple to test A.

### Why Static code is an outdated construct?

Look, no construction work builds new homes with a mallet and wooden pegs. They use nail guns. Some even use pre-formed nailed walls. Sure, a mallet and wooden pegs are still useful in certain limited scopes, but when it comes to building a wall in a house, they are pretty much never used.

One of the most well-known outdated coding constructs is the goto statement in modern languages. No modern developer uses it in any modern language. In fact, most developers would get mocked in a code review for using the goto statement. Why? Because it is a buggy code construct that can always be replaced with a less buggy coding construct. Also, a developer who uses goto would be demonstrating an extremely outdated skill set. Like the mallet and wooden pegs, there are rare times goto needs to be used in legacy languages, assembly languages, etc. But most developers should never use it.

Just like the mallet and wooden pegs, and the goto statement, most static code is outdated. Static code enforces coupling. Coupling is well-known to be problematic. Like with the goto statement, static code can always be replaced with a more stable and less buggy coding constructor. In this case, it is dependency injection.

Static enforces coupling. Static code cannot be put directly behind an interface. Extra effort is needed to wrap the static, which once done, makes the static useless. There are a lot of applications with a lot of bad static code. If it is dead code, leave it to die. If it is code you work on, replace it with non-static code as soon as possible.

The exception is non-volatile and simple code, such as string.Concat. All the data is in memory. Nothing is volatile. It is easier to write unit tests without mocking than with mocking. Just like wooden pegs might be still used today in special circumstances, such as joining corners, static can also be used in special circumstances. The problem occurs when we try to use them outside of those special circumstances.

Should a developer be mocked for writing new static code like they’d be mocked for writing new code using the goto statement? Yes. Yes, they should. For legacy, no. Nobody would mock a pioneer in the 1800s for using a mallet and wooden pegs, or a historical restoration person fixing a historical building. Similarly, some legacy code is so full of static, it can’t be helped. However, today, if some construction worked tried to tell his company he was going to replace his nail gun and nails with a mallet and wooden pegs, he be laughed off the job site. If a developer is writing new static code, they are demonstrating an outdated skillset and such code should never pass a code review.

## Library Coupling

Library coupling is different than class coupling in that there are many classes in a library. It only takes a reference of one class from one dll to one class in another dll to create library coupling.

Imagine you have two dlls, A.dll and B.dll. There are three possible ways these could be coupled. If either references the other, or they have a circular dependency, they are coupled.

1. **A coupled to B**A black background with a black square

   Description automatically generated with medium confidence
2. **B coupled to A**A black background with a black square

   Description automatically generated with medium confidence
3. **Circular**A black background with a black square

   Description automatically generated with medium confidence

**Note: Visual Studio will prevent circular** dependencies inside a solution when using project references, but if projects aren’t in a single solution, or the dll are referenced directly, not through projects, then can still occur.

Both A to B, B to A coupling are considered couplings, and whether they are bad coupling depends on what is happening, but as they aren’t interface-only dlls, they are likely bad. Circular dependencies are definitely just bad.

So what are some methods to avoid bad coupling?

1. Referencing interface-only dlls instead of implementation dlls.
2. No inter-reference at all, but instead they are joined via the composition root.

Because this example is simplistic, it may be difficult to see the problems that this coupling causes.

Imagine that B.dll gets a new dependency on C.dll. Because A.dll depends on B.dll, it also will inherit the dependency to C.dll, which is very problematic.

### Referencing interface-only dlls

The first thing to understand about referencing an interface-only dll is that you must first have an interface-only dll. Let’s demo this in a contrived example.

Example:

Interface dlls

* A.Interfaces.dll – Has an IA interface. No dependencies.
* B.Interfaces.dll – Has an IB interface. No dependencies.

Implementation dlls

* A.dll – Has class A. Depends on A.Interfaces.dll and B.Interfaces.dll.
* B.dll – Has class B. only depends on B.Interfaces.dll

It is very important to have interfaces and implementations in separate dlls. Implementations often reference more than one interfaces-only library, so one of the keys to decreasing or eliminating coupling is to have implementation dlls only reference interface-only libraries.

A group of blue and green squares

Description automatically generated

Because this example is simplistic, it may be difficult to see the benefits this provides over direct coupling.

Imagine that B.dll gets a new dependency on IC.dll. Because A is decoupled, it doesn’t inherit that dependency at build-time. In fact, B.dll can go through drastic changes and A.dll doesn’t even need to recompile as long as B.Interfaces.dll doesn’t change.

There is usually nothing wrong with A.dll depending on B.Interfaces.dll. However, it could be wrong if that dependency shouldn’t exist. There are other ways to make so A.dll isn’t even coupled to B.Interfaces.dll. Read the next section for steps to enable this level of decoupling.

### Free photo: Light Bulb - Bird, Boat, Bulb - Free Download - JooinnParadigm Shift - Concrete Implementations of Interfaces

Just because a dll has an interface defined, doesn’t mean the concrete implementation should exists in that same dll.

A dll not even having an implementation of an interface is completely normal and often preferred.

This concept is often hard for developers to understand, especially developers who have been coding in a coupled way for a long time. Interface-only dlls help with this because it helps a developer clearly see that the interface and the implementation don’t have to exist in the same library, and more importantly, they likely shouldn’t exist in the same library. This is one of the foundations of the D in SOLID, Dependency Inversion.

### Coupling only at the composition root

This is done by adding a third item, the composition root, to our diagram. A composition root is usually a place at the startup of an application where all the code is composed together to make the application run.

In the below diagram, you can see the new third box representing the Composition Root. Notice that neither A.dll nor B.dll reference each other. However, the composition root references both.

A black and blue square with yellow text

Description automatically generated with medium confidence

At first glance, this looks like coupling, and it is, so why it is better? Why is it freeing code to be decoupled when the diagram still shows coupling? Because it is minimized coupling. If you look at the above diagram, A has no arrows pointing from it. Same with B. Now imagine that you have library C.dll.

The composition root is in the runtime application. A.dll and B.dll are freed from build-time coupling to each other. Only the run-time application has all the dependencies.

However, the composition root has a build-time coupling to A.dll and B.dll. So while this is better, we don’t want to depend on implementations at build time. We only want to depend on implementations at runtime.

Taking this one step further, we can also have the composition root depend on only interfaces at build time.

A yellow and black sign

Description automatically generated

Interfaces have to have implementations at some point, so A.dll and B.dll are required at runtime, but while this is true, we don’t need them to be required at build-time. A.dll and B.dll are not needed to build the composition root.

The run-time dependency diagram will include an implementation of A.dll and B.dll whereas the build-time dependency diagram does not.

How is this done? When the application launches, some configuration can be made in the composition root so all interfaces in A.Interfaces.dll get implementations from A.dll and similarly all interfaces in B.Interfaces.dll get implementations from B.dll. The key is that this only happens at runtime. Usually an IoC container library makes this possible.

### Big O regarding dependencies

Consider Big O where we are adding one library to an application with N existing libraries.

In a coupled application, it could have A.dll and B.dll depend on C.dll, which would be two lines. And if we add a D.dll, and A, B, and C all depend on it, that is three lines. This demonstrates that when adding one library without the composition root, the coupling to that library could be Big O(n) or worse.

With a composition root, it is only Big O(1). To add one new library, a dev would only add one dependency line from the composition root to the C.dll. Same for D.dll. This demonstrates that adding one new library to N existing libraries in an application is Big O(1).

### Chained dependencies problem

Consider referencing a library that has a whole dependency tree of its own. Does your library really need all those dependencies. Probably not. However, they come along for the ride. This means that if LibraryA depends on LibraryB, but doesn’t need any of LibraryB’s dependencies, all those chained libraries are requirements of LibraryA for no reason.

### Decoupling with the Composition Root – The Process

OK, diagrams are nice, but as a developer, you need to make this happen, so you need me to get down deep in the weeds and tell you how you implement this successfully. There is a simple process for this.

The process is written as only two steps.

1. Make sure library A.dll doesn’t not reference library B.Interfaces.dll or B.dll.
2. Make any changes needed for these libraries to interact in the composition root.

However, step 2 isn’t yet in the weeds. Let’s go into the weeds next.

#### Types of Coupling

You should be able to describe the types of coupling yourself by answering this question:

*Why would A.dll ever need to be coupled to B.dll in the first place?*

There are only a few constructs in code that code in A.dll would need from B.dll.

1. Model Classes or classes without methods, includes enums.
2. A class with methods or just the method itself
3. Other – many languages have custom constructs

So if a class in A.dll needs one of the above constructs from another library B.dll, coupling occurs. Let’s go through these and how they can be decoupled using the composition root.

#### Decoupling Model Classes with the composition root

If there is a model class in B.dll that a class in A.dll needs, it might sound impossible to resolve. That is because, to start with, the entire premise is wrong. A.dll never needs a model from B.dll in the first place.

Note: Model classes are usually simple classes that have no methods. If model classes have methods, that is another antipattern that will not be discussed here, just know it is heavily frowned upon. Follow the rule: Models have no methods!

A.dll has a model need. B.dll has a model need. They may appear to be the same model, and in fact they may be identical when first created or even forever, however, A.dll and B.dll have separate responsibilities and as such should have separate models. The model in A.dll and the model in B.dll could benefit from being the same but mostly like won’t.

Note: Another option is to have a third library where a common model class lives. In fact, simple models are often so simple that they are included in interface-only dlls and as long as the “Models don’t have methods” rule is not broken, they are sufficiently simple and abstract. However, this still has the downside of A.dll and B.dll sharing the same model, which may not always fit for they separate responsibilities.

What about Don’t Repeat Yourself (D.R.Y.)? Wouldn’t having a duplicate model break the DRY principle. Maybe, but a model in A.dll and a model in B.dll are separate responsibilities, so they aren’t actually the same model. The confusion here is because they may have similar enough goals that models are identical or nearly identical. This isn’t a break of DRY, despite appearances. If you aren’t convinced still, then just try if for a while until you see the benefits.

So how do we decouple A.dll from needing a model in B.dll? Using a mapper in the composition root.

Let’s use an example:

**Person model in A.dll.**

Namespace NamespaceForA

{

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public int Age { get; set; }

}

}

**Person model in B.dll**

This could be exactly the same model, but for example purposes, we will make it different by having a birthday as a DateTimeOffset instead of an Age.

Namespace NamespaceForB

{

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public DateTimeoffset BirthDate { get; set; }

}

}

Now, in the composition root, we need a mapper. In C#, you may use AutoMapper, or you can write your own mapper. If you write your own mapper, it should be non-volatile. In C#, a static extension method could be used. Also, in C#, you can also use a generic with reflection, however, reflection can have performance issues, but those can be mitigated with caching a Type’s reflection info. Also Expressions could be use, again, this can be sped up by precompiling and caching precompiled Expressions.

For simplicity’s sake, we will use the static extension method.

public namespace NamespaceForMyCompositionRoot

{

using NamespaceForA.Person APerson;

using NamespaceForB.Person BPerson;

public static class PersonExtensions

{

public APerson Convert(this BPerson p,

DateTimeOffset? Now = null)

{

// Method injecting now value allows for:

// 1. Age at certain moments in time

// 2. Easy unit tests

now = now ?? DateTimeOffset.Now;

return new APerson

{

FirstName = p.FirstName,

LastName = p.LastName,

Age = p.BirthDate.ToAge()

}

}  
 }

}

Now, anytime Person data flows from B to A, we have a nice separation of concerns. B’s person and A’s Person classes are not coupled, neither is A.dll coupled to B.dll.

Now, you might be asking, where would this be called? If nothing is coupled to both A.dll and B.dll, and the composition root is only coupled to them for composing, how would this ever be called. The answer is that in a real application, there are usually methods along with models that need to be called. This example was for models and is perhaps too trivial to demonstrate this. Method examples come next.

Lastly, you can give models interfaces or simple models are approved to include in interfaces as models are a form of a contract. Then the composition root could still depend only on interface at build time and dynamically map the implementations only at runtime.

#### Decoupling Methods

There are multiple way to decouple methods. To understand them, we have to first understand the method type.

* 1 to 1 method types – one library calls a method in another library.
  + A.dll calls an method in B.dll
* 1 to many method types – one library calls 0 to N methods, Publish/Subscribe scenario.
  + A.dll calls any method that has subscribed

These cover the majority of method calls, but code is so unique and ever changing so new ones could be found.

#### Decoupling Methods using an interface implementation in the composition root

One of the key points of decoupling is that not all interfaces have to be implemented in the implementation dll. For example, imaging A.dll has an interface IGetPerson. But A.dll has no code to get the a Person. Maybe your application will eventually get the Person from a database, and that uses B.dll, not A.dll.

namespace NamespaceForA

{

public interface IGetPerson

{

Person GetPerson(long personId);

}

}

This can be implemented in the composition root. Let’s assume that there already exists a way to get the person in B.dll. We just inject that way in.

public namespace NamespaceForMyCompositionRoot

{

using NamespaceForA.Person APerson;

using NamespaceForA.IGetPerson IGetPersonA;

using NamespaceForB.Person BPerson;

using NamespaceForB.IGetPerson IGetPersonB;

public class PersonGetter : IGetPersonA

{

private readonly IGetPersonB \_personGetterB;

public PersonGetter(IGetPersonB personGetterB)

{

\_personGetterB = personGetterB;

}

public APerson GetPerson(long personId)

{

return personGetterB.GetPerson(personId)

.Convert(); // To APerson

}  
 }

}

Now you can use this implementation in your composition root. A.dll doesn’t know about B.dll and B.dll doesn’t know about A.dll, but the composition knows about both.

Note: This is one example of what is meant by the D in SOLID, dependency inversion. Instead of A.dll referencing B.dll, the dependency is inverted, and instead, other code (the composition root) references A.dll.

#### Decoupling Methods using Method Variables in the composition root

Methods can be variables. This is key element to decoupling. For example, imaging A.dll has an interface IGetPerson and an implementation that relies on a method injected as a constructor parameter.

Note: In C#, method variables used to be delegates, but that construct, while still available, was replaced with Action<> and Func<> classes which should be used instead.

Your application will eventually get the Person from a database, and there is already a method that does that somewhere else. That method can be injected into the implementation.

This can be implemented in the composition root. Let’s assume that there exists already a way to get the person, we would create an implementation that uses that.

public namespace NamespaceForA

{

public class PersonGetter : IGetPerson

{

private readonly Func<long, Person> \_personGetterMethod;

public PersonGetter(Func<long, Person> method)

{

\_personGetterMethod = method;

}

public APerson GetPerson(long personId)

{

return \_personGetterMethod(personId);

}  
 }

}

Now, the method you inject may have a problem, in that it might not return a NamespaceForA.Person object as it likely comes from B.dll. You can simply wrap it, and this would be done in the composition root.

public APerson GetPersonWrapper(long personId)

{

return personGetterB.GetPerson(personId)

.Convert(); // To APerson

}

That method can go anywhere in the composition root (make sure it is a place that makes sense) and then the GetPersonWrapper can be injected into NamespaceForA.GetPerson implementation. A.dll doesn’t know about B.dll and B.dll doesn’t know about A.dll, but the composition knows about both.

Note: This is not significantly different than the interface one. I prefer to use interfaces over method variables because interfaces are easier for new developers to use and understand and usually have a more obvious place for the interface implementation to live, whereas a method doesn’t seem to have a place to live.

Now, this is usually used for 1 to 1 method calls. In the above example, it wouldn’t make sense to get a Person object twice. However, in a more complex example, the Person object could be derived from multiple sources (imagine Age came from a different database than FirstName and LastName), and calling two sources to aggregate the data is totally possible. However, doing that would not be what is considered one library calls 0 to N methods as these are separate methods with separate signatures. When we discuss one library calling 0 to N methods, all the methods will have the same signature and can be added to the same list.

Note: This is one example of what is meant by the D in SOLID, dependency inversion. Instead of A.dll referencing B.dll, the dependency is inverted, and instead, other code (the composition root) references A.dll.

#### Decoupling Methods using a Publish/Subscribe the composition root

This is best understood as the publish/subscribe model or an event system.

Let’s say that A.dll can add a Person to our system. B.dll and C.dll would both like to know when a person is added to the system. They might also want to know about updates and deletes.

In coupled code, A.dll would reference both B.dll and C.dll and then call them directly in a coupled fashion whenever a Person is added. That is bad coupling. This is solved by using the Publish/Subscribe pattern.

It wouldn’t be terrible to have B.dll and C.dll reference A.Interfaces.dll. However, that isn’t required. Sometimes it makes sense to do that, but this could be done 100% in the composition root. Both ways have their pros and cons. However, in this instance, it is typical of consumers to reference the subscriber’s interface, so it is likely that B.dll and C.dll (or any consumer) references A.Interfaces.dll.

Note: This is one example of what is meant by the D in SOLID, dependency inversion. Instead of A.dll reference all subscribers, the dependency is inverted and all the subscribers referenced A.Interfaces.dll.

When A.dll and its consumers are in the same application, then the subscribing is done in the composition root.

Note: This model is often done in the publisher application (aka the server) with multiple subscriber applications (aka clients), and the subscribing should also be done in the composition root, but each client application would have its own composition root.

Here is what you might have in A.dll.

public namespace NamespaceForA

{

public class PersonNotifier : IPersonNotifier

{

private readonly HashSet<Func<Person>> \_subscribers;

public void Subscribe(Func<Person> func)

{

if (!\_subscribers.Contains(func))

\_subscribers.Add(func)

}

// An Unsubscribe would also be implemented.

// args would have the Person that was changed.

// args would have the change type, Add, Update, Delete

public void OnPersonChanged(IPersonChangedArgs args)

{

if (\_subscribers.Any())

{

foreach (var subscriber in \_subscribers)

subscriber.Invoke(args.Person);

}

}

}

}

Again, this is a contrived and simplistic example. Likely you would want to handle more CRUD actions, multithreading, and exceptions, etc. However, as you can see, the coupling is minimized to all subscribers needed to be coupled to 1 interface.

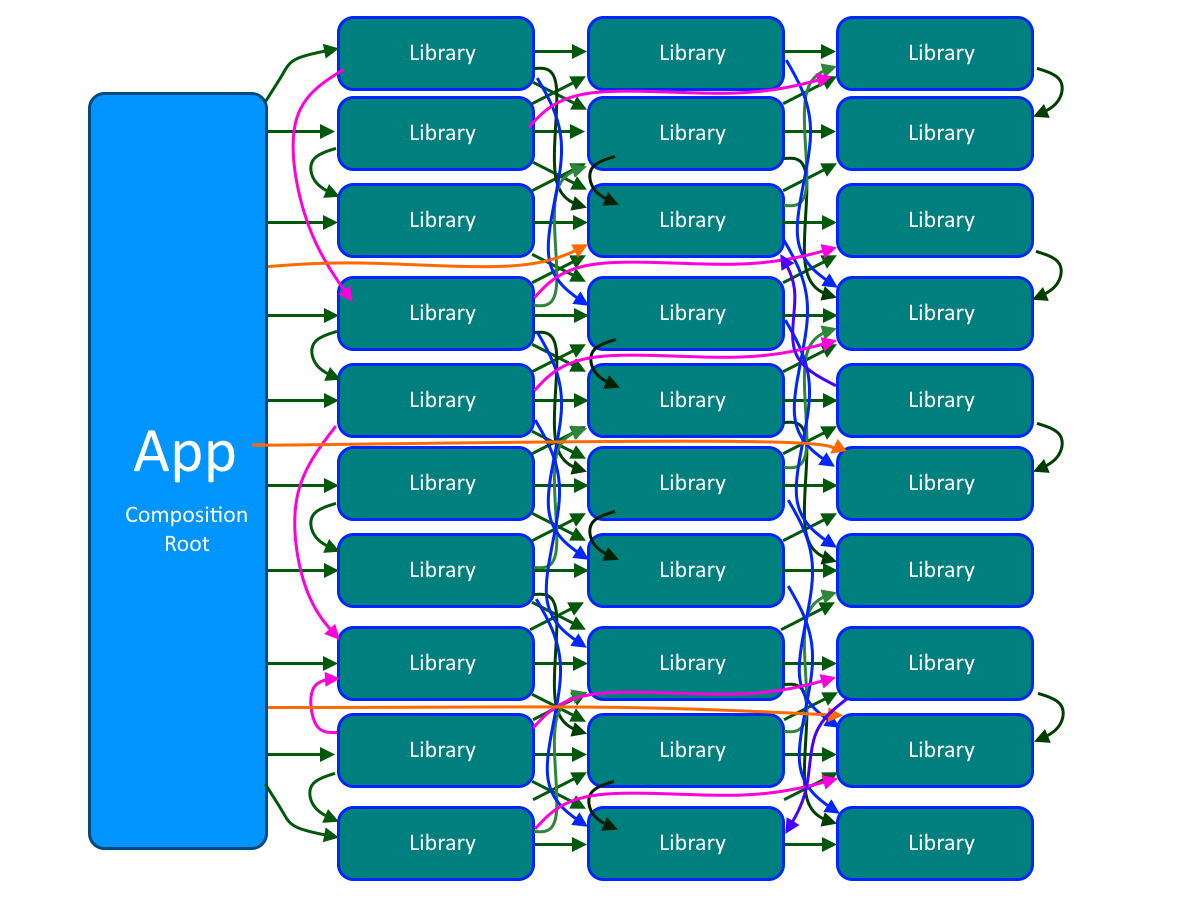
## Coupled vs Decoupled Library Architecture

A picture says a thousand words, so let’s look at two pictures. One of a bad library-to-library coupled architecture and one of a decoupled architecture.

### Coupled Architecture

Notice that in the coupled architecture, the arrows are impossible to follow. There are many projects out there where this is an over-simplification, and the reality is far worse. There could be 300 libraries and a dependency tree so crazy it looks like a black ink spot filling an entire image when trying to add all the libraries and arrows.

Also notice that there are far more libraries in the coupled architecture. That is because, as mentioned above in the [Chained Dependency Problem](#_Chained_dependencies_problem) section, coupled architecture leads to more library references than are actually needed.



### Decoupled Architecture using the App or Composition Root

Notice how in the decoupled architecture, the arrows are clean and easy to follow. Not every library is dependency free, but the few dependencies they have are easy to understand.

Notice how there are far fewer dependencies. That is because without coupling, unnecessary dependencies aren’t included if they aren’t used.

A screenshot of a computer screen

Description automatically generated

## Lab

We have created a GitHub repo with multiple branches to demonstrate some of these decoupling techniques.

* <https://github.com/rhyous/Examples.Decoupling/tree/1-CoupledCode>
* <https://github.com/rhyous/Examples.Decoupling/tree/1-CoupledCode-LibraryC>
* <https://github.com/rhyous/Examples.Decoupling/tree/2-InterfaceOnlyLibrary>
* <https://github.com/rhyous/Examples.Decoupling/tree/3-ZeroCoupling>
* <https://github.com/rhyous/Examples.Decoupling/tree/3-ZeroCouplingWithReturnModel>
* <https://github.com/rhyous/Examples.Decoupling/tree/3-ZeroCouplingWithReturnModelAutomapper>

### Preparation

Before doing the labs, to the following:

1. Clone this repo:  
   <https://github.com/rhyous/Examples.Decoupling.git>
2. Open the solution:   
   ./Example.Decoupling/src/Rhyous.Example.Decoupling.sln

### Lab 1a

1. Switch to this branch: (Either in VS or in the command line)  
   <https://github.com/rhyous/Examples.Decoupling/tree/1-CoupledCode>
2. Read each class file.
3. Draw the library and class coupling architecture.  
   Note: This is an example of bad coupled architecture.
4. Run the solution and step through it, using Step into.

### Lab 1b

1. Switch to this branch: (Either in VS or in the command line)  
   <https://github.com/rhyous/Examples.Decoupling/tree/1-CoupledCode-LibraryC>
2. Notice LibraryC was added.
3. Read each class new class file.
4. Update your architecture drawing to include LibraryC.  
   Note: Again, this is an example of bad coupled architecture, but it is getting worse.
5. Run the solution and step through it, using Step into.

### Lab 2

1. Switch to this branch: (Either in VS or in the command line)  
   <https://github.com/rhyous/Examples.Decoupling/tree/2-InterfaceOnlyLibrary>
2. Read each class file.
3. Notice, interface libraries were added.
4. Verify Build decoupling (i.e. LibraryA doesn’t rebuild with a change to LibraryB.)
   1. Make a minor change to LibraryB, any change, doesn’t matter, even just whitespace.
   2. Build
   3. Notice that a change to LibraryB doesn’t cause LibraryA to rebuild.
5. Draw the library and class coupling architecture.  
   Note: This is an example of architecture only coupled to interfaces. Better, but not best.
6. Run the solution and step through it, using Step into.

### Lab 3a

1. Switch to this branch: (Either in VS or in the command line)  
   <https://github.com/rhyous/Examples.Decoupling/tree/3-ZeroCoupling>
2. Read each class file.
3. Notice LibraryA only references LibraryA.Interfaces. It doesn’t reference LibraryB or LibraryB.Interfaces.
4. Notice the implementation of the event is in the composition root (or the App).
5. Verify Build decoupling (i.e. LibraryA doesn’t rebuild with a change to LibraryB.)
   1. Make a minor change to LibraryB, any change, doesn’t matter, even just whitespace.
   2. Build
   3. Notice that a change to LibraryB doesn’t cause LibraryA to rebuild.
6. Notice the unit tests are also decoupled.
7. Draw/Update the library and class coupling architecture.  
   Note: This is an example of good, decoupled architecture.
8. Run the solution and step through it, using Step into.

### Lab 3b

1. Switch to this branch: (Either in VS or in the command line)  
   <https://github.com/rhyous/Examples.Decoupling/tree/3-ZeroCouplingWithReturnModel>
2. Read each class file.
3. Notice that A has an interface for getting the Something model: ISomethingProvider.
4. Notice that the implementation of is in the composition root (or the App).
5. Notice that both A and B have their own models.   
   Why doesn’t this break the D.R.Y. principle?
6. Notice that LibraryB’s Something model is converted to LibraryA’s Something model.
7. Draw/Update the library and class coupling architecture.   
   Note: This is an example of good, decoupled architecture.
8. Run the solution and step through it, using Step into.

### Lab 3C

1. Switch to this branch: (Either in VS or in the command line)  
   <https://github.com/rhyous/Examples.Decoupling/tree/3-ZeroCouplingWithReturnModelAutomapper>
2. Notice that AutoMapper is now configured in Program.cs.
3. Notice that AutoMapper is injected into the ConcreteSomethingProvider and used to convert LibraryB’s Something model to LibraryA’s Something model.
4. Draw/Update the library and class coupling architecture.  
   Note: This is an example of good, decoupled architecture.
5. Run the solution and step through it, using Step into.

#### Lab 4A

1. Switch to this branch: (Either in VS or in the command line)  
   <https://github.com/rhyous/Examples.Decoupling/tree/4-ZeroCouplingMultipleImplementations>
2. This adds on to Lab 3. Read each class file.
3. Notice B has been changed to have two implementations.
4. Notice a MySettings now determines which instance of B to use.
5. Draw/Update the library and class coupling architecture.   
   Note: This is an example of good, decoupled architecture.
6. Run the solution and step through it, using Step into.
   1. Notice it uses LibraryB.Implementation2.B.
7. Notice there is a launchSettings.json file under App\Properties.
   1. Change the value from 1 to 2
   2. Run the solution again
   3. Notice it uses LibraryB.Implementation2.B now.
8. This is example of how to use the O in SOLID. Implementation1 and Implementation2 can be closed for change, but another implementation can be added. With a little more work, the setting could probably be enhanced to not need to be changed either, perhaps by adding an IoC framework and changing the setting to return the fully qualified type and resolving it.

#### Lab 4B

1. Switch to this branch: (Either in VS or in the command line)  
   <https://github.com/rhyous/Examples.Decoupling/tree/4-ZeroCouplingWithJsonSerializer>
2. This adds on to Lab 3. Read each class file.
3. Browse to various files to notice details and architecture:
   1. Notice A and B now only have a method to Serialize.
   2. Notice both have their own ISerializer interface.
   3. Notice that neither A.Interfaces, A, B.Interfaces, nor B reference a serializer of any type.
   4. Notice the implementations are in the composition root. See the Implementations folder.
   5. Notice that multiple implementations are easily created and used.
4. Run the application.
   1. Notice
5. This is example of how to use the O in SOLID. The serializer can be changed without touching A.Interfaces, A, B.Interfaces, or B. A and B code can be closed for change, but any serializer can be used, making it open for extension.
6. Add an Xml serializer implementation for both A and B.
   1. Right-click on the App project and choose Manage NuGet packages.
   2. Add this NuGet package to the App project: Rhyous.EasyXml.  
       <PackageReference Include="Rhyous.EasyXml" Version="1.2.7" />
   3. Create an implementation for LibraryA.Interfaces.ISerializer that serializes with Xml.  
      Hints:  
      using Rhyous.EasyXml  
      …  
      return Serializer.DeserializeFromXml<T>(serializedObject);  
      …  
      return Serializer.SerializeToXml<T>(obj);
7. In Program.cs:
   1. Create the two instances of your new xml serializers.
   2. Add a third instance of A using the Xml serializer.
   3. Add a third instance of B using the Xml serializer.
   4. Run again (it will fail).
   5. This fails because it cannot serialize an object with a constructor. Rhyous.EasyXml is just
8. Solve the error in the implementation.
   1. Suggestion: Use a simple model to and AutoMapper to solve this.
   2. If you need help see:   
      <https://github.com/rhyous/Examples.Decoupling/compare/4-ZeroCouplingWithJsonSerializerAndXml>

### Lab 5

1. Look at an example in your proprietary code base.   
   Hint: Often logging is a good example. Create a new library or look at an existing library. Consider adding logging without referencing the library with the logger. Instead have the App or Composition Root reference the logger.

## Conclusion

Bad coupling needs to be avoided in new applications. Bad coupling can be removed and replaced in old applications. The patterns and practices in this document can help guide you to accomplishing this.

* Classes only reference abstracts (usually interfaces)
* Interfaces and implementations should be in separate dlls
* Libraries rarely reference each other and when they do:
  + They only reference the interfaces dll
  + It is the composition root which is the only library that can reference everything

Bad coupling does not always need to be removed in dead, legacy code, that is in code freeze.

Bad coupling should be removed from living applications that must be maintained for many years to come. If you are currently working on the code, adding new features to the code, it is living code.

Bad coupling kills maintainability, making such code overly difficult (and more important, overly costly) to maintain.