Suggested Practices

Composition over Inheritance

Using composition makes for more reuseable and more testable code.

Inheritance

Is it not possible to create instances of abstract classes making testing just the abstract functionality difficult. On top of that you are not able to mock out the dependent functionality meaning every test over `ExecuteOperation` must also test `OnImportantFunction`

```
public abstract class MyAbstractBase {
    protected virtual void OnImportantFunction() {
        //do work here
    }
}

public class MyImplementation : MyAbstractBase {
    public override void ExecuteOperation() {
        //do work
        OnImportantFunction();
        //do more work
    }
}
```

Composition

Using composition makes testing `ImportantFunction` trivial as the object may be instantiated on its own. It is also possible to mock out the functionality of `ImportantFunction` when testing `ExecuteOperation` allowing for testing just the implementation's functionality while ensuring the dependent operation is also called.

```


public interface IProvideFunction {
    void OnImportantFunction();
}

public class MyFunctionProvider : IProvideFunction {
    public void ImportantFunction() {
        //do work here
    }
}

public class MyImplementation {
    private readonly IProvideFunction _function;

    public MyImplementation(IProvideFunction function) {
        _function = function;
    }

    public override void ExecuteOperation() {
        //do work
        _function.ImportantFunction();
        //do more work
    }
}
```

Dependency Injection over Service Locators (Singletons)

Service Locators create tightly coupled implementation that are more difficult to manage and test. Singletons can also cause threading issues by allowing contention on shared resources.

Service Locator (Singleton)

While service locators/singletons may be tested in isolation. Calling code and not mock out or easily replace the dependent call.

```


public class MyFunctionProvider {
    public static void ImportantFunction() {
        //do work here
    }
}

public class MyImplementation {

    public override void ExecuteOperation() {
        //do work
        MyFunctionProvider.ImportantFunction();
        //do more work
    }
}
```

Dependency Injection

Using dependency injection makes it much easier to test operations in isolation. It also allows for less contention in multi-threaded environments.

```
public interface IProvideFunction {
    void OnImportantFunction();
}

public class MyFunctionProvider : IProvideFunction {
    public void ImportantFunction() {
        //do work here
    }
}

public class MyImplementation {
    private readonly IProvideFunction _function;

    public MyImplementation(IProvideFunction function) {
        _function = function;
    }

    public override void ExecuteOperation() {
        //do work
        _function.ImportantFunction();
        //do more work
    }
}
```

Annotations over Marker Interfaces

Annotations (also known as Decorators or Attributes) are a means to provide metadata without effecting the type system.

Marker Interface

While this does have the advantage it may be easily accessed using the static typing provided by your language's type system. Many prefer to use annotations and refection for for metadata versus using the type system.

```
public interface IMyTag {
}

public class MyClass : IMyTag {
}
```

Annotations

Annotations provide a simple means of adding metadata to your class without impacting the type system or implementation of a given class. Annotations are easily access from reflection and help create a clean distinction from implementation.

```
public class MyTagAttribute : Attribute {
}

[MyTag]
public class MyClass {
}
```