Design and Code Review Checklist

Instructors Who Consult. | Consultants Who Teach.

BUILDING YOUR BLUEPRINT FOR DESIGN & CODE REVIEW

INTERTECH

A Quality CONSIDERATIONS

Did you know?

SOFTWARE VENDORS LOSE
AN AVERAGE OF **.63% OF THEIR MARKET VALUE**
ON THE DAY SECURITY-RELATED FLAWS ARE DISCLOSED.

.63% [YOU LOSE]

APPROXIMATELY **15% OF BUGS INTRODUCED**
DURING DEVELOPMENT GO UNDETECTED
PRIOR TO CODE RELEASE.

15% [BUGS]

SLOWING REVIEW RATES DOWN TO
200 LINES OF CODE PER HOUR
EFFECTIVELY IDENTIFIES NEARLY HALF
OF DESIGN DEFECTS AND MORE THAN HALF
OF THE BUGS.

[200 LINES]

1/2

B Structural PRIORITIES

Does the code satisfy these principles?

1 OPEN CLOSED PRINCIPLE:
- OPEN FOR EXTENSION
- CLOSED FOR MODIFICATION

2 DRY PRINCIPLE
- DON'T REPEAT YOURSELF

3 SINGLE RESPONSIBILITY PRINCIPLE
- ONLY ONE RESPONSIBILITY FOR EACH PROJECT

4 LISKOV SUBSTITUTION PRINCIPLE
- SUBTYPES MUST BE SUBSTITUTED FOR THEIR BASE TYPES

Consider delegation over inheritance, unless you need to change base class behavior.

C Maintainability ISSUES

YAGNI (you aren't gonna need it).

Beware of gold plating: when in doubt, leave it out!

EVERYTHING IN ITS PLACE!

- IS THE CODE **IN THE CORRECT PLACE?**
- IS THE CLASS/PROCEDURE/VARIABLE **SCOPED APPROPRIATELY?**
- WERE THE **RIGHT CLASS/METHOD/VARIABLE NAMES** USED?
- DOES **THE NAME SPACE** MAKE SENSE?
- ARE THE METHODS AND VARIABLES **TYPED CORRECTLY?**

FINALLY, **DON'T FORGET TO REVIEW UNIT TESTS.**

+800 .866 .9884 + 651 .288 .7000

INTERTECH

Instructors Who Consult. | Consultants Who Teach.

WWW.INTERTECH.COM

Design & Code Review Checklist - Table of Contents

This Intertech checklist provides a comprehensive compilation of design and code review principles for consideration during projects. There are items on the checklist that are outlined in detail further on in the document and a few where we've provided links from this document to quality design and review resources.

***Underlined Text* Links To Further Detail**

- ✓ Review Unit Tests
- ✓ Is the code in the right place?
- ✓ Does the name space make sense?
- ✓ Is the class / procedure / variable scope correct?
- ✓ Are the Classes, methods, and variables named correctly?
- ✓ Are the methods, and variables typed correctly?
- ✓ Look at the code.
- ✓ Review for OCP (Open Closed Principle - Open for extension closed for modification)
- ✓ Review for DRY Principle (Don't Repeat Yourself - abstract common things and put in single place).
- ✓ Review for SRP (Single Responsibility Principle - every object has a single responsibility. All the object's services should be focused on that responsibility).
- ✓ Review for LSP (Liskov Substitution Principle Subtypes must be substitutable for their base types).
- ✓ Consider Delegation over Inheritance. If you don't need to change base class behavior, consider delegating (handing over responsibility of a task) rather than inheritance.≠
- ✓ Consider Composition over Inheritance. Similar to delegation except the owner class uses a set of behaviors and chooses which one to use at runtime. When the delegating class is destroyed, so are all the child classes.
- ✓ Aggregation. Similar to composition except when the delegating class is destroyed, the child classes are not.
- ✓ Consider Polymorphism. Make a group of heterogeneous classes look homogeneous
- ✓ Consider generics.
- ✓ Testability considerations?
- ✓ YAGNI (You ain't gonna need it) When in doubt, leave it out!
- ✓ Does object wake up in a known good state (constructor)
- ✓ Consider Security.

Is the class / procedure / variable scope correct?

Below is a scoping/access modifier cheat sheet:

Classes:

- Public Class - Access is not restricted.
- Private Class - Only valid for nested classes
- Internal Class - only visible to the assembly

Members:

- Private Member - only available to the containing type
- Protected Member - Available to the containing type and anything derived from the containing type.
- Internal Member - available to current assembly.
- Protected Internal Member - Available to current assembly or anything derived from containing type.
- Public Member - Access is not restricted

Valid Member Access Modifiers:

- Enum - public
- Interface - public
- Class - public, private, internal, protected, protected internal
- Struct - public, private, internal

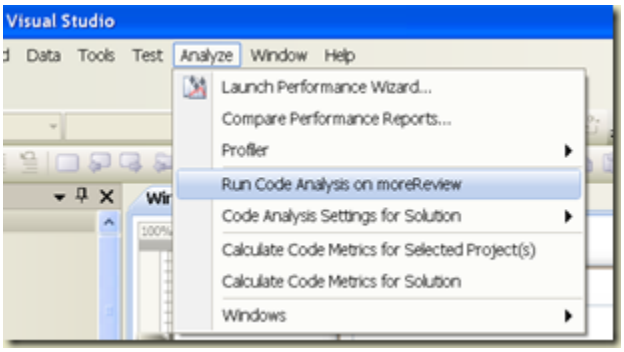
Inheritance:

- Abstract class - cannot be instantiated. May have abstract and non abstract methods. Derived class must implement all abstract methods.
- Sealed Class - Cannot be inherited
- Virtual Method - may be overridden
- Abstract method - no implementation, must be overridden
- Sealed method - Cannot be overridden
- Sealed Override Method - No longer may be overridden. Public sealed override MyMeth()

Look at the Code

- ✓ Is the code logically correct?
- ✓ Are we following best practices?
- ✓ Are we considering security?

Did we run a code review? If you have a team or enterprise developer edition, the code review may be found in the Analyze menu.



- ✓ Is the code maintainable?
- ✓ Is our code unnecessarily complex? I always favor simplicity until forced to do otherwise.
- ✓ Is the error handling effective?
- ✓ Will our code perform? I tend to assume it will until proven otherwise.
- ✓ Are our objects loosely coupled?

Open-Closed Principle (OCP)

SRP it is an easy concept to understand but breaking down a system into the “correct” objects is difficult to do well. It is imprecise and there is a lot of room for opinions. You seldom know the correct breakdown until you are finished. In this blog I want to take a look at the OCP or Open/Closed Principle. The Open/Closed principle states that Software entities (classes, modules, functions etc.) should be open for extension but closed for change. (“Agile Principles, Patterns, And Practices in C#” Robert C Martin and Micah Martin). In other words this means that we would like to be able to change the behavior of existing entities without changing the code or binary assembly. This concept is not as easy to understand. Did I just say to change something without changing it?

Let’s use a configurable rule engine as an example. Assume the simple rule engine has 2 types of entities, a Condition and an Action. The Action is simple, execute some logic and return true on success or false on failure. The condition is also simple, it checks some binary condition and returns true or false. The rules engine takes a list of conditions and actions and executes them. If an Action returns false, the rules stop executing and a rollback is performed. If a condition returns true, the next action is executed. If the condition is false, the next action is skipped and the following action is executed. Pretty simple so let’s look at some sample code:

Define our Conditions, Actions, and Rule Type with enums:

```
1.      public enum Actions
2.      {
3.          CreateOrder,
4.          CreateBackorder,
5.          CloseOrder,
6.          ShipOrder,
7.          StoreOrder,
8.          ReduceInventory
9.      }
10.       public enum Conditions
11.       {
12.           IsComplete,
13.           IsInStock,
14.           CanShip
```

```
15.      }
16.      public enum RuleType
17.      {
18.          Condition,
19.          Action
20.      \
```

Here is a very crude implementation of a rule engine

```
1. public class RuleEngine
2.     {
3.
4.         public void ExecuteRules(int rulesId)
5.         {
6.             //Gather all the conditions and actions
7.             //assume we have a Ruleset table that looks like this:
8.             //ID - Int
9.             //Index - Int
10.              //RuleType - Int (corresponds to RuleTypeEnum)
11.              //Rule - Int (corresponds to Conditions or Actions enum)
12.
13.              //Note, transaction logic, creating the parameter
14.              //Error handling, and safe DataReader.Read() logic
15.              //has been omitted for brevity
16.              using (SqlConnection connection
   = new SqlConnection("ConnectString"))
17.              {
18.                  using ( SqlCommand cmd = new SqlCommand
19.                      ("SELECT RuleType, Rule FROM Ruleset where ID = ?
   ORDER BY Index"
20.                      , connection))
21.                  {
22.                      SqlDataReader rules = cmd.ExecuteReader();
23.                      bool executing = true;
24.                      while (executing)
```

```
25.                     {
26.
27.                         rules.Read();
28.                         if ((int)rules[0] == (int)RuleType.Action)
29.                         {
30.                             executing = ExecuteAction((int)rules[1]);
31.                         }
32.                         else
33.                         {
34.                             if(ExecuteCondition((int)(rules[1])))
35.                             {
36.                                 executing = true;
37.                             }
38.                             else
39.                             {
40.                                 rules.Read();
41.                                 executing = true;
42.                             }
43.                         }
44.                         //Implement some exit strategy here
45.                     }
46.                 }
47.             }
48.         }
49.         public bool ExecuteCondition(int theCondition)
50.         {
51.             switch (theCondition)
52.             {
53.                 case (int)Conditions.IsComplete:
54.                     //Is the order complete?
55.                     return true;            //or false
56.                 case (int)Conditions.CanShip:
57.                     //Can we ship the order?
58.                     return true;            //or false
59.                 case (int)Conditions.IsInStock:
60.                     //Is this item in stock?
61.                     return true;            //or false
```

```
62.                     default:
63.                         throw new Exception("Unsupported Condition");
64.                 }
65.             }
66.
67.             public bool ExecuteAction(int theAction)
68.             {
69.                 switch (theAction)
70.                 {
71.                     case (int)Actions.CreateOrder:
72.                         //Execute order create logic
73.                         return true;            //or false
74.                     case (int)Actions.CreateBackorder:
75.                         //Execute backorder logic
76.                         return true;            //or false
77.                     case (int)Actions.ShipOrder:
78.                         //Execute shipping logic
79.                         return true;            //or false
80.                     case (int)Actions.StoreOrder:
81.                         //send to warehouse
82.                         return true;            //or false
83.                     case (int)Actions.CloseOrder:
84.                         //Execute order close logic
85.                         return true;            //or false
86.                     case (int)Actions.ReduceInventory:
87.                         //Remove item from inventory
88.                         return true;            //or false
89.                     default:
90.                         throw new Exception("Unsupported Action");
91.                 }
92.
93.             }
94.         \
```

Never mind the problems with error handling, transactions, and the lack of support for nested conditions, the point of the blog is OCP, not creating a rule engine. What we have will work but

does it satisfy the OCP? No, there are a couple problems with this type of implementation that will make it difficult to maintain. The first problem is that the logic for the Rule Engine, Conditions, and Actions are all in one class and therefore one assembly. Any system that wants to use any of this logic will be tied to all of this logic. The second problem is that any time you want the rule engine to do something new, you have to modify this assembly which is a violation of the Open/Closed Principle.

Let's take a look at a more robust design.

We will still use an enum to distinguish between Actions and Conditions:

```
1.      public enum RuleType
2.      {
3.          Condition,
4.          Action
5.      \
```

Now let's declare an interface:

```
1.      public interface ISupportRules
2.      {
3.          public bool Execute();
4.          public RuleType TypeOfRule();
5.      \
```

We will put both the enum and the Interface in an assembly called Rule.Types.
Now let's add a few classes:

```
1.       public class CreateOrderAction:ISupportRules
2.       {
3.           #region ISupportRules Members
4.
5.           public bool Execute()
6.           {
7.               //Order Create Logic Here
```

```
        }

          public RuleType TypeOfRule()
          {
              return RuleType.Action;
          }

          #endregion
      \
```

We will put this in an assembly called Rule.Actions.CreateOrder.

```
    public class CanShipCondition:ISupportRules
    {
        #region ISupportRules Members

        public bool Execute()
        {
            //Execute Can Ship Logic here
        }

          public RuleType TypeOfRule()
          {
              return RuleType.Condition;
          }

          #endregion
      \
```

We will put this class in Rules.Conditions.CanShip. In fact we will create a separate assembly for each condition and action we defined in the enums.

Here is our new rules engine which goes in the Rule.Engine assembly:

```
1.          public void ExecuteRules(List<ISupportRules> rules)
2.          {
3.              bool executing = true;
4.              int ruleIndex = 0;
5.              while (executing)
6.              {
7.                  if (rules[ruleIndex].TypeOfRule() == RuleType.Action)
8.                  {
9.                      executing = rules[ruleIndex].Execute();
10.                     ruleIndex++;
11.                 }
12.                 else
13.                 {
14.                     if (rules[ruleIndex].Execute())
15.                     {
16.                         ruleIndex++;
17.                         executing = true;
18.                     }
19.                     else
20.                     {
21.                         ruleIndex += 2;
22.                         executing = true;
23.                     }
24.                 }
25.                 //Implement some exit strategy here
26.             }
27.
28.         }
29.     \
```

Notice that the ExecuteRules method takes a generic list of type ISupportRules as a parameter but has no reference to any of the conditions or actions. Also notice that the condition and action classes have no reference to each other or the rules engine. This is key to both code reuse and extensibility. The refactored rule engine, condition, and action classes are completely independent of each other. All they share is a reference to Rule.Type. Some other system may use any of these assemblies independent of each other with the only caveat being they will

need to reference the Rule.Type assembly. The other thing we gained with this approach is we can now Extend the rule engine (make it execute new conditions and actions) by simply adding a new Condition or Action that implements ISupportRules and passing it into the ExecuteRules method as a part of the generic list. We can do all of this without recompiling the RefactoredRulesEngine which is the goal of the OCP. By the way, this design approach is called the Strategy Pattern.

If you haven't noticed yet I'm leaving out one major piece of the puzzle. How does the generic list of rules get generated? I'm going to wave my hands here a little bit and save the details for another blog. We would use a creational pattern (one of the factory patterns). If we assume we are consuming the table outlined in our first solution this factory would accept a Ruleset ID and magically return the generic List<IsupportRules> of rules. The implementation of the factory pattern could be written in such a way that each time you add a Condition or an Action the factory would need to be recompiled or we could use a provider pattern and use a configuration file to allow us to create these new Conditions and Actions without a recompile.

To summarize things a bit: conceptually we have this RulesEngine that is relatively complex (much more complex than I have written) and we want to write it, test it, and leave it alone. At the same time though we have this need to enhance the system by adding more rules. By using using the strategy pattern we now have this stable rule execution engine that can execute any condition or action that implements the ISupprotRules interface. Because we inject a list of conditions and rules into the ExecuteRules method we can do all of this without recompiling the refactored rules engine. Another approach we might have taken to satisfy the OCP is the Template Method pattern. In the template method pattern we would make use of an abstract class to define the skeleton of an algorithm, then allow the concrete classes to implement subclass specific operations.

Review for DRY Principle

DRY in the DRY principle is an acronym for Don't Repeat Yourself. While we are going to focus on applying the principle to code, it can and should be applied to more than just code. The can be applied to database schemas, tests, documentation, test plans, design documents, build scripts and more. Any time we start duplicating our work we are signing up for parallel maintenance and chances are we will eventually have the items fall out of synch.

We can use several techniques to avoid duplicate code. Some obvious things we can do include using classes and methods to organize common code. We can write an abstract or base class to implement common behaviors for multiple derived classes.

We can use properties to perform a validation in one place rather than performing the validation anywhere we want to use a member variable:

```
1.  [System.Diagnostics.DebuggerBrowsable(System.Diagnostics.DebuggerBrowsab
    leState.Never)]
2.          int aComplexInteger;
3.          public int AComplexInteger
4.          {
5.              get { return aComplexInteger; }
6.              set
7.              {
8.                  if (value == 0)
9.                      throw new ArgumentOutOfRangeException("AComplexIntege
    r");
10.                 if (value != aComplexInteger)
11.                 {
12.                     aComplexInteger = value;
13.                     //Maybe raise a value changed event
14.                 }
15.             }
16.          \
```

One of my favorite techniques is constructor chaining. If we have multiple constructors that perform similar logic, we should use constructor chaining to avoid duplicating code:

```
1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Text;
5.
6. namespace CicMaster
7. {
8.     class BetterConstructorLogic
9.     {
10.          #region WhyItIsBetter
11.          //No duplicate code, this is called constructor chaining
12.          //step through this
13.          #endregion
14.          string someString = string.Empty;
15.          int someInteger = 0;
16.          List<int> myIntegers = new List<int>();
17.
18.          public BetterConstructorLogic():this("A Default Value")
19.          {
20.              //someString = "A Default Value";
21.              System.Diagnostics.Debug.WriteLine("In Default
   Constructor");
22.          }
23.
24.          public BetterConstructorLogic(string aString):this(aString,123)
25.          {
26.              //someInteger = 123;
27.              System.Diagnostics.Debug.WriteLine("In one param
   constructor");
28.          }
29.
30.          public BetterConstructorLogic(string aString, int anInteger)
31.          {
```

```
32.             System.Diagnostics.Debug.WriteLine("In two param
   constructor");
33.
34.             someString = aString;
35.             someInteger = anInteger;
36.             myIntegers.Add(someInteger);
37.             myIntegers.Add(someInteger ^ 3);
38.             myIntegers.Add((int)(2 * 3.14 * someInteger));
39.         }
40.     }
41. }
42.
```

The final technique I would like to mention is use a factory to create all but the simplest objects. The following (admittedly nonsensical) code needs to execute maybe a half dozen lines of code to construct an Order object.

```
1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Text;
5. using System.Threading;
6.
7. namespace BusinessLayer
8. {
9.     class ObjectContext
10.     {
11.         public ObjectContext(String username)
12.         {
13.             //Look up the user permissions
14.         }
15.         public bool IsInTranaction { get; set; }
16.         public bool BeginTransaction()
17.         {
18.             //do some transaction logic
```

```
19.                 return true;
20.             }
21.         }
22.         class Order
23.         {
24.             public Order(ObjectContext theContext)
25.             {
26.
27.             }
28.
29.         }
30.         class Consumer
31.         {
32.             public Consumer()
33.             {
34.                 ObjectContext ctx
    = new ObjectContext(Thread.CurrentPrincipal.Identity.Name);
35.                 if (!ctx.IsInTranaction)
36.                 {
37.                     if (ctx.BeginTransaction())
38.                     {
39.                         //...
40.                     }
41.                 }
42.                 Order order = new Order(ctx);
43.
44.             }
45.         }
46.
47.  }
48.
```

Duplicating these few lines of code in a couple places is not that difficult. Now say the application is enhanced and grows for a few years and suddenly we see this code duplicated dozens or hundreds of times. At some point it is likely that we want to change the construction

logic, finding and changing all the code we use to create the order is difficult, time consuming, and a QA burden.

A better approach would be to encapsulate the logic required to build a new order. Here is an implementation using a simple factory. It is much easier to find, change, and test this code:

```
1.  static class OrderFactory
2.      {
3.          public static Order GetOrder()
4.          {
5.              ObjectContext ctx
    = new ObjectContext(Thread.CurrentPrincipal.Identity.Name);
6.              if (!ctx.IsInTranaction)
7.              {
8.                  if (ctx.BeginTransaction())
9.                  {
10.                     //...
11.                   }
12.             }
13.             return new Order(ctx);
14.         }
15.     }
16.     class DryConsumer
17.     {
18.         public DryConsumer()
19.         {
20.             Order order = OrderFactory.GetOrder();
21.         }
22.     \
```

If we recognize that we are duplicating even a few lines of code over and over we need to take a serious look at the code and figure out a way to encapsulate the logic.

Single Responsibility Principle

Back when I first started managing a team of developers one of the things I instituted was code reviews. We created some coding standards and a list of best practices to follow including naming conventions, error handling, and optimizations around performance and we inserted a review step into our development process. After a while writing code that followed our standards became second nature to the team so we would go into the reviews, validate the code, and wrap up the meeting in a very short period of time. Where the process fell short was we were focusing on the code, not the design. Writing good, clean, uniform, and optimized code is great but what is probably more important is to have a good design.

A good design has a lot of characteristics but one of the most important signs of a good design is a solid breakdown of class responsibilities. The Single-Responsibility Principle states that a class should have only one reason to change. ("Agile Principles, Patterns, And Practices in C#" Robert C Martin and Micah Martin).

Here is an example requirement:

When a file is dropped into a folder, attach some meta-data to it and move it to a secure location (say a file stream).

From a high level there are a few things we need to do:

- Monitor a folder looking for new files.
- Create meta-data.
- Associate the meta-data to the file.
- Stream the file to the database.

A first pass at defining the classes might be as follows:

- Create a FileHandler class responsible for monitoring the folder and moving the file to the secure location
- Create a Document class responsible for creating the meta-data and associating it to the file stream.

For the sake of this blog I'm only going to look at the file handling logic. I do think the FileHandler logic defined above is a defensible breakdown of tasks, the FileHandler object takes care of everything having to do with the files we want to process. What I don't like about the break down is we have put the file monitoring logic and file transferring logic in the same class. If we change the requirements down the road so that we only look for files with certain extensions we would have to change the FileHandler implementation. If we change the requirements and we need to support a different secure storage location (such as a third party document management system that doesn't support streaming) we would again have to change the FileHandler.

With the high level design I defined above we would have 2 reasons to change the class:

1. Because the file monitoring logic changes

2. Because the file transfer logic changes.

This is a violation of the SRP.

Still not convinced? Let's look at another change that in my opinion tips the scales in favor of separating the file monitoring and transferring logic. What if we want to allow documents to be imported into the system via a web service? We don't want to duplicate the file transferring logic so we would want to employ the services of the FileHandler object. We certainly don't want or need the file monitoring logic in our web service, therefore I favor putting the file monitoring logic and file transferring logic in different classes.

Review for Liskov Substitution Principle

The (Barbara) Liskov Substitution principle states:

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

That hurts my head. Let's again go with the definition from "Agile Principles, Patterns, And Practices in C#" Robert C Martin and Micah Martin: Subtypes must be substitutable for their base type.

There are two angles from which I want to look at this principle. Let's look at it first from a structural standpoint. I'm going to reuse the refactored code from my last post with a slight change. Rather than having the conditions and actions implement the ISupportRules interface, they will be derived from an abstract class named RuleBase. This means our RulesEngine's ExecuteRules method needs to accept a generic list of RuleBase.

The Code:

We have our enums :

```
1. namespace OpenClosePrinciple
2. {
3.     public enum Actions
4.     {
5.         CreateOrder,
6.         CreateBackorder,
7.         CloseOrder,
8.         ShipOrder,
9.         StoreOrder,
10.          ReduceInventory
11.     }
12.     public enum Conditions
13.     {
14.         IsComplete,
```

```
15.          IsInStock,
16.          CanShip
17.      }
18.      public enum RuleType
19.      {
20.          Condition,
21.          Action
22.      }
23.  }
24.
```

An Action derived from RuleBase:

```
1. namespace OpenClosePrinciple
2. {
3.     public class CreateOrderAction:RuleBase
4.     {
5.         #region ISupportRules Members
6.
7.         public override bool Execute()
8.         {
9.             return true;
10.         }
11.
12.         public override RuleType TypeOfRule()
13.         {
14.             return RuleType.Action;
15.         }
16.
17.         #endregion
18.     }
19. }
```

A Condition derived from RuleBase:

```
1. namespace OpenClosePrinciple
2. {
3.     public class CanShipCondition:RuleBase
4.     {
5.         #region ISupportRules Members
6.
7.         public override bool Execute()
8.         {
9.             return true;
10.          }
11.
12.          public override RuleType TypeOfRule()
13.          {
14.              return RuleType.Condition;
15.          }
16.
17.          #endregion
18.      }
19.  }
20.
```

RuleBase:

```
1. amespace OpenClosePrinciple
2. {
3.     public abstract class RuleBase
4.     {
5.         public virtual bool Execute()
6.         {
7.             return true;
8.         }
9.
10.          public abstract RuleType TypeOfRule();
```

```
11.       }
12.   }
13.
```

The Rule Engine.

```
1. using System;
2. using System.Data.SqlClient;
3. using System.Collections.Generic;
4. namespace OpenClosePrinciple
5. {
6.      public class RefactoredRuleEngine
7.      {
8.
9.          public void ExecuteRules(List<RuleBase> rules)
10.           {
11.               bool executing = true;
12.               int ruleIndex = 0;
13.               while (executing)
14.               {
15.                   if (rules[ruleIndex].TypeOfRule() == RuleType.Action)
16.                   {
17.                       executing = rules[ruleIndex].Execute();
18.                       ruleIndex++;
19.                   }
20.                   else
21.                   {
22.                       if (rules[ruleIndex].Execute())
23.                       {
24.                           ruleIndex++;
25.                           executing = true;
26.                       }
27.                       else
28.                       {
29.                           ruleIndex += 2;
30.                           executing = true;
```

```
31.                         }
32.                     }
33.                     //Implement some exit strategy here
34.                 }
35.
36.             }
37.         }
38.     }
39.
```

Our Base class simply supports an Execute and a RuleType Method. Both our condition and Action class derive from RuleBase and may be "passed around" as a RuleBase and therefore we have satisfied the structural idea of being able to substitute a derived class for its base class.
The second angle I want to look at this principle is from a behavioral perspective. Most examples I read demonstrating an LSP violation use the Rectangle base class and the square subclass. Proof of the LSP violation is based on setting the length and width of the Square to unique values then discovering that an assert in a unit test that calculates a rectangles area returns an incorrect result.
I agree that this is a violation. I believe the problem lies in the claim that a square "is-a" rectangle. Now don't go running to a dictionary and grab the definition of a rectangle and tell me that a square fits the definition of a rectangle – I'm not talking about the English language. The classic implementation of the rectangle when discussing the LSP principle is to expose a Width and a Height property and a CalculateArea method which returns Width X Height. This makes sense. The problem in claiming that the Square "is-a" Rectangle is that with a Square there is not a notion of a Width and a Height that are different. The Width and Height must be the same. It doesn't make sense to expose 2 properties, it is misleading and a consumer can logically assume that they would be independent properties. But in the implementation of the square this is not the case. Our Square is not (logically) a Rectangle because it does not have a Length and Width that are independent of each other.
To consider whether our design above has violated LSP from a behavior standpoint we have to take a look at the RuleEngine. At a glance it looks like we have a violation, notice that we have some logic in the engine that concerns itself with the RuleType.

if (rules[ruleIndex].TypeOfRule() == RuleType.Action) …

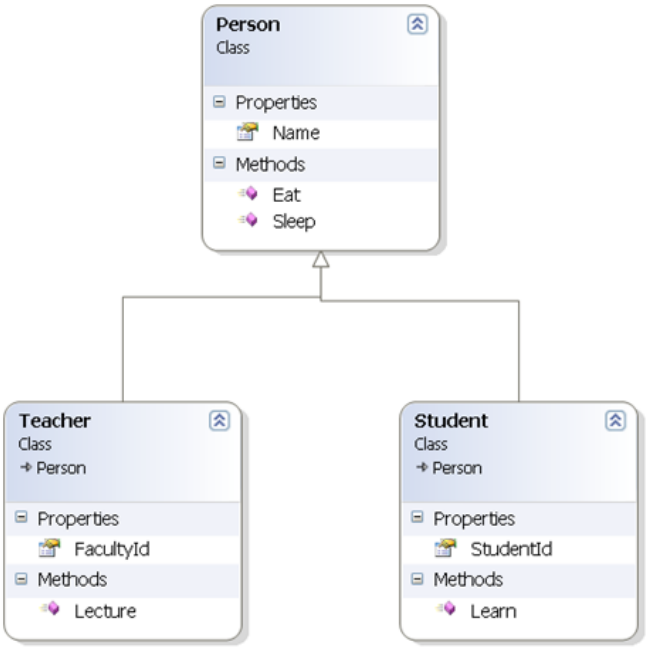
This does have the “smell” of bad code that we need to constantly look for but I maintain that if we take a closer look this is NOT a violation. When we cooked up the notion of the RuleEngine we decided we would support 2 types of rules, Conditions and Actions. It is reasonable to do this. We would never get our code out the door if we didn’t put some constraints on the types of rules we could support. The line of code in question is simply executing code based on whether it is a condition or an action.
An example of code that would violate LSP or OCP would be as follows:

```
if (TypeOf(rules[ruleIndex]) == CanShipCondition)
```

Here we are trying to execute logic based on the derived type which is a violation of OCP and LSP. The consumer of the base class would not have to worry about the derived type if the derived type was substitutable for its base class and by worrying about derived types, the rules engine is no longer open for extension.

Delegation vs. Inheritance

Consider delegation over inheritance. I have had some pretty heated debates on this topic in the past and I will even take it a step further and say the rule should state that we should favor delegation over inheritance. Let's consider the following classic (and intuitive) person/student/teacher example. A class diagram might look something like this:

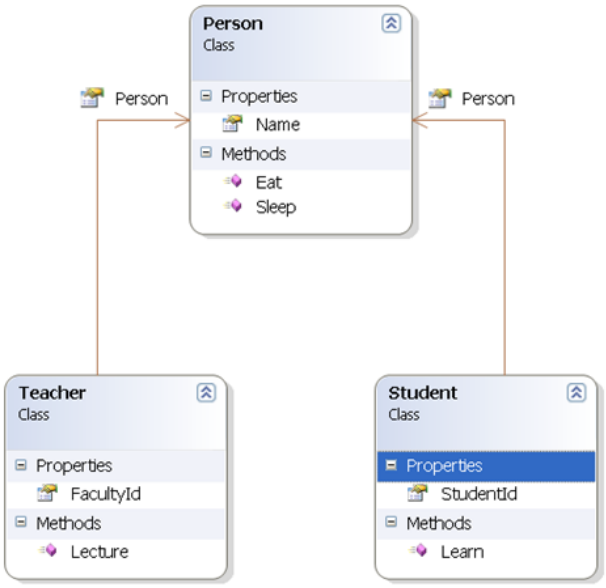


Person is our base class; Teacher and student derive from person. We can say that a student Is A person and that a Teacher Is A person.

The up-side of this design is it is simple, intuitive, and it will perform well. The down side of this approach is that there is a tight coupling between the derived classes and the base class, we are breaking encapsulation by making the functionality of the derived class dependent on the functionality in the base class, and we may even break encapsulation on our Person class by using protected or public access modifiers where we normally wouldn't do so.

The problem with breaking encapsulation is that making a change to one class (our base class) can cause unintended changes in other consuming classes (our derived classes). I can't begin to tell you how many times I have seen a seemingly small change to a base class break huge pieces of an application.

Now let's consider using delegation rather than inheritance. We will change our class diagram to look like the following:



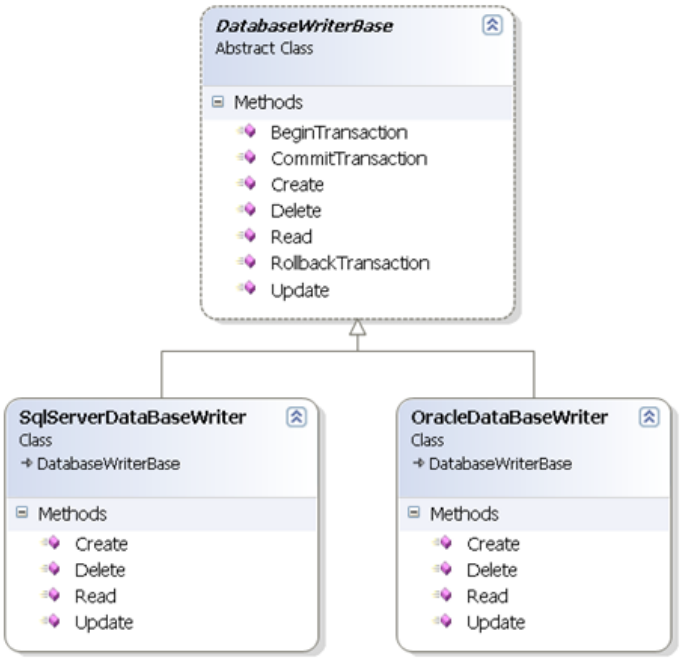
We can say a teacher Has A person and similarly student Has A person.

The down-side of this design is that we have to write more code to create and manage the person class and thus the code will not perform as well (though in most cases I suspect the performance hit is negligible). We also cannot use polymorphism - we cannot treat a student as a person and we cannot treat a teacher as a person. The up-side of this design is that we can decrease coupling by defining a person interface and using the interface as the return type for our Person property rather than the concrete Person class, and our design is more robust. When we use delegation we can have a single instance of a person act as both a student and a teacher. Try that in C# using the inheritance design!

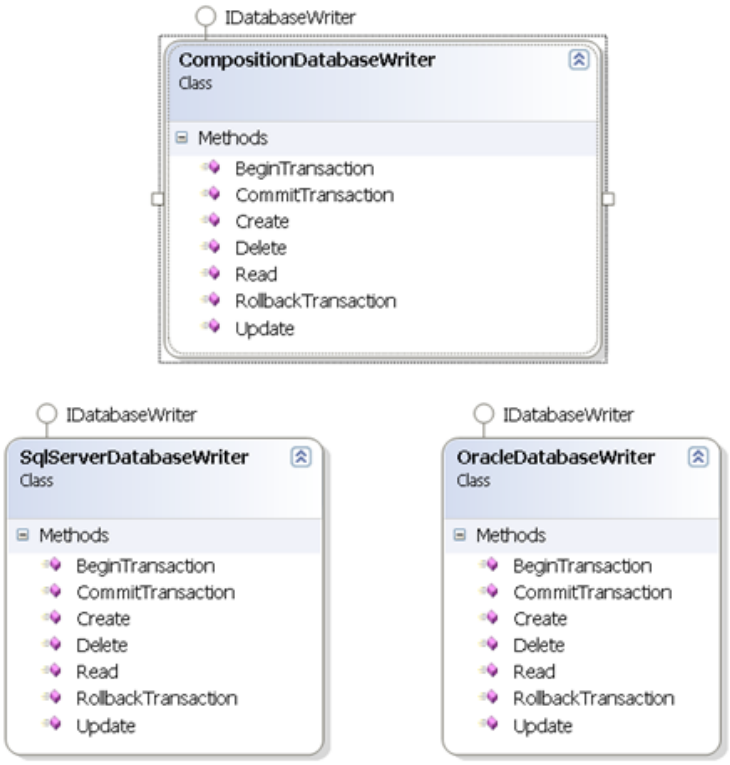
Composition vs. Inheritance

Composition is a method we use to combine simple objects into more complex objects. With composition we have an owner class that changes behavior by delegating the implementation of certain behaviors to smaller and simpler objects. When we destroy the owner object, all of these smaller objects the owner uses are also destroyed. If we were to use inheritance to achieve this same goal (changing behavior) we would use a subclass to change the behavior of the base class. As I mentioned in my last post, the base class and subclass are tightly coupled and this can tend to be a bit brittle.

Consider the very real case where we are writing an application that needs to work with both Oracle and SQL Server. We could use inheritance to define a base class and have one subclass for SQL Server and another for Oracle. Our application can figure out which object to instantiate at runtime and that will work pretty well. Here is what our class diagram might look like:



If we take a little different approach and use composition instead we might have a class diagram similar to the following:



The CompositionDatabaseWriter does not need to implement IDatabaseWriter but I did that because I think it makes the example easier to understand. The application will use the CompositionDatabaseWriter to do database work and CompositionDatabaseWriter will determine whether to use SqlServerDatabaseWriter or OracleDatabaseWriter at runtime perhaps by using a configuration file entry. When one of the CompositionDatabaseWriter methods is called, CompositionDatabaseWriter simply calls the corresponding method on the Sql Server or Oracle object.

Both designs allow us to interact with an oracle or Sql Server database which was our goal. Now here is where the flexibility of composition comes in; Suppose we now need to support a

to do to use it is deploy this single assembly and change a config file. Very powerful, because we used composition we can change the behavior at runtime via configuration.

The fact of the matter is that our sample is so simple we didn't even use composition at all so imagine the considerably more complex scenario where what we are doing is creating a persistence object that must transactionally work with the file system and a database. We define the persistence object and within the persistence object we delegate the responsibility of dealing with the file system to one class, and dealing with the database to another. The persistence object is composed of a FileSytemWriter and a DatabaseWriter. Our Persistence object has only one concern – coordinating database and file persistence. Our FileSystemWriter has only one concern – managing File System interactions, and finally our DatabaseWriter is only concerned about interacting with a database of one type. We have decomposed a fairly complex problem into a few smaller more managable problems, we came up with a nice robust design thanks to composition and we have managed to do it in such a way that there are no SRP violations either!

YAGNI – You Ain't Gonna Need It!

The YAGNI principle states that programmers should not implement functionality until it is necessary. It is a pretty simple principle but oddly enough it is a difficult principle to follow.

I'll take a crack at what I think the upside of breaking this rule is.

- It seems to me the biggest benefit to breaking this principle is that we can architect and design our code by taking into account both current and future features thus implementing a more robust design.
- Another big argument is that by implementing this feature our software will be more useful. If our software is more useful we may sell more licenses or we may be more productive.

Let's look at the most obvious negatives of implementing a feature before you need it. This is a longer list.

- If we don't need it yet, maybe we should be spending our time and money implementing something we do need now.
- If we implement a feature we should have some corresponding documentation describing the feature – this takes time.
- If we implement something we need to test it -every release and this takes a lot of time.
- When we add a feature in an agile/TDD shop, we need to write unit tests for the feature so we can automatically test it every release. Writing the tests can take as long as or longer than writing the feature and anyone who has to run unit tests before a check in will probably agree that adding time to this automated process is not good.
- When we add a feature we need to train the support crew and user base on how to use the feature.
- A feature is hard enough to implement correctly when it is needed, how do we expect to implement it optimally when we are still guessing whether or not we need it?

A quick summary of the negatives: We are spending time writing code for something we might need. We are writing the feature the way we think we will need it in the future. We are

spending time and money documenting and testing a feature that we are guessing we will need and by the way we are also guessing how to best implement it.

If all of those (mostly financial) arguments aren't enough to convince you, let's take a look at some down sides from an architect or developer's point of view.
If code is in a release you have to assume it is being used. When it comes time to change the code you think you needed, you are going to spend time figuring out how to change it without breaking the (imagined?) user base. This might involve migrating data, configurations, backup procedures, reports, integration work flows etc. When it comes time to change the code you really do need you must consider all of the code. This includes the code you don't realize that you don't need – if it is in production, how can you be confident in saying "oh, we really don't need that code". This really stifles our ability to modernize our code. Even if you do find a way to refactor the bloated code base not only will you have to change the existing code but you will have to change the tests, documentation, and training materials.

Let the business analysts, user base, and marketing folks decide what features your product needs and when it needs it. I would rather architect and design a system based on things that are known. I don't ever want to tell somebody that the system is the way it is because I thought we needed some functionality that turns out to be useless. When we do a good job of designing and reviewing our system we can be confident that we can refactor our code to implement major functional changes when they are understood, needed, and the highest priority.

Contact Information

Intertech, Inc.

1020 Discovery Road, Suite 145

Eagan, MN 55121

+800.866.9884

+651-288-7000

Ryan McCabe – Vice President of Sales

+651.288.7001

RMcCabe@Intertech.com

Intertech Background

Tom Salonek founded Intertech in 1991. Intertech is a leading Twin Cities-based software development and the largest software developer training company in Minnesota.

Intertech designs and develops software solutions for state government and mid-sized corporations. Intertech has created prepackaged software, software that powers Fortune 500 businesses, as well as systems for state government. Intertech works with NASA, Wells Fargo, Lockheed Martin, Microsoft and Intel and other major companies around the United States teaching and helping them use technology.

Intertech's technical team frequently publishes books in the *Intertech Instructor Series* through a partnership with Apress in Berkley, California. The *Intertech Instructor Series* includes best-selling technology training books on Amazon.com.

Growth

Intertech is frequently listed in "fast growth", "top", and "best" lists, including:

- **2013** Consulting Magazine, 8th Best IT Consulting Firms to Work For in North America
- **2013** Consulting Magazine, 1st Employee Morale

- **2013** Ernst & Young, CEO named Entrepreneur of the Year Finalist
- **2013** The Business Journal, Great Places to Work (nine time winner)
- **2013** Inc. 5000, One of the Fastest Growing Firms in America (six time winner)
- **2012** Minnesota Business Magazine, 100 Top Employers, #1 Mid-Sized Company Winner
- **2012** The Business Journal, Fast 50 growth firm (three time winner)
- **2012** The Business Journal, Great Places to Work (eight time winner)
- **2012** Inc. 5000, One of the Fastest Growing Firms in America (five time winner)
- **2012** Star Tribune, Top 100 Workplace (eighth place in category)
- **2011** The Business Journal, Great Places to Work (seven time winner)
- **2011** Inc. 5000, One of the Fastest Growing Firms in America (four time winner)
- **2010** "Healthiest Employer" by the Minneapolis/St. Paul Business Journal and OptumHealth
- **2010** PCI Entrex, 4th Quarter "Entrex Growth Awards"
- **2010** PCI Entrex, 3rd Quarter "Entrex Growth Awards"
- **2010** The Business Journal, Great Places to Work (six time winner)
- **2010** Minneapolis/St. Paul Business Journal and OptumHealth, Healthiest Employer
- **2009** The Business Journal, Great Places to Work (five time winner)
- **2009** Inc. 5000, One of the Fastest Growing Firms in America
- **2009** The Wall Street Journal, Winning Workplaces finalist (one of 35 in America)
- **2008** UpSize Magazine - Business Builder Awards, Communications Finalist
- **2008** PCI Entrex, Fastest Growing Privately held firm in US (Q4)
- **2008** PCI Entrex, Fastest Growing Privately held firm in US (Q3)
- **2008** The Business Journal, Great Places to Work (four time winner)
- **2008** Minnesota Work Life Champion, Awarded For Promoting Healthy Work and Life Balance
- **2008** Inc. 5000, One of the Fastest Growing Firms in America
- **2007** UpSize Magazine - Business Builder Awards, Community Impact Finalist
- **2007** The Business Journal, Great Places to Work (three time winner)
- **2007** Inc. 5000, One of the Fastest Growing Firms in America
- **2006** The Business Journal, Great Places to Work, 10th in Minnesota

- **2006** The Business Journal, Top 25 Software Development Firms
- **2005** The Business Journal, 40 Under 40, CEO named as one of the top business leaders under 40
- **2005** UpSize Magazine - Business Builder Awards, Finance & Operation Best Practices Winner
- **2004** The Business Journal, Great Places to Work, top 20 in Minnesota (ranking not released)
- **2003** Minnesota Technology Fast 50, 33rd Fastest Growing Firm in Minnesota
- **2003** The Business Journal, Top 25 Computer Training Firms, 6th Largest Firm in Minnesota
- **2003** Twin Cities Business Monthly, Top Computer Training Firms, 4th Largest Firm in Minnesota
- **2002** Minnesota Technology Fast 50, 16th Fastest Growing Firm in Minnesota
- **2002** The Business Journal, Top 25 Computer Training Firms, 9th Largest Firm in Minnesota
- **2001** Forbes ASAP, 440th Fastest Growing Firm in America
- **2001** Inc. 500, 286th Fastest Growing Firm in America
- **2001** Minnesota Technology Fast 50, 11th Fastest Growing Firm in Minnesota
- **2001** The Business Journal, Top 25 Computer Training Firms, 16th Largest Firm in Minnesota
- **2000** Forbes ASAP, 335th Fastest Growing Firm in America
- **2000** Inc. 500, 243rd Fastest Growing Firm in America
- **2000** Minnesota Technology Fast 50, 16th Fastest Growing Firm in Minnesota

Recognition

In six of the last seven years, Intertech was chosen from a field of over 200 as a winner in The Business Journal's Best Places to Work in Minnesota. In addition, Intertech has been featured in Fortune Small Business, Forbes, The Business Journal, Twin Cities Business Monthly, Upsize, Ventures, The Star Tribune, The Pioneer Press, and Inc. magazine.

Company & Staff Credentials

- Project Management Professionals (PMP®)
- Java Certified Programmer
- Java Certified Developer
- IBM Business Partner

- Microsoft Gold Certified Partner
- Custom Development Solution Competency
- Learning Solutions Competency
- Microsoft Certified Systems Administrator (MCSA)
- Microsoft Certified Professional (MCP)
- Microsoft Certified Systems Engineer (MCSE)
- Microsoft Certified Applications Developer (MCAD)
- Microsoft Certified Solution Developer (MCSD)
- Microsoft Certified Trainer (MCT)
- Certified Scrum Master

Intertech, Inc.

GSA Contract Holder Contract GS-35F-XXXXW

IBM Business Partner

Scrum Alliance

Microsoft Partner
Gold Application Lifecycle Management
Silver Web Development
Silver Software Development
Silver Learning

BBB MEMBER

Intertech Staff.

Sun Java Developer

PMP®

CSM ScrumAlliance Certified ScrumMaster

Microsoft Certified Trainer

Microsoft Certified Professional

Microsoft Certified Software Engineer

Sun Java Programmer

MSSE

amazon web services Partner Network

Microsoft Certified Solution Developer

Microsoft Certified Application Developer

MVP Microsoft Most Valuable Professional

Give Us A Call If We Can Be Of Assistance.

Ryan McCabe – Vice President of Sales

+651.288.7001

RMcCabe@Intertech.com