Design Pattern Automation: your compiler just got smarter

Introduction

Software development projects are becoming bigger and more complex every day. The more complex a project the more likely the cost of developing and maintaining the software will far outweigh the hardware cost.

There's a super-linear relationship between the size of software and the cost of developing and maintaining it. After all, large and complex software requires good engineers to develop and maintain it and good engineers are hard to come by and expensive to keep around.

Despite the high total cost of ownership per line of code, a lot of boilerplate code still is written, much of which could be avoided with smarter compilers. Indeed, most boilerplate code stems from repetitive implementation of design patterns. But some of these design patterns are so well-understood they could be implemented automatically if we could teach it to compilers.

Implementing the Observer pattern

Take, for instance, the Observer pattern. This design pattern was identified as early as 1995 and became the base of the successful Model-View-Controller architecture. Elements of this pattern were implemented in the first versions of Java (1995, Observable interface) and .NET (2001, INotifyPropertyChanged interface). Although the interfaces are a part of the framework, they still need to be implemented manually by developers.

The INotifyPropertyChanged interface simply contains one event named *PropertyChanged*, which needs to be signaled whenever a property of the object is set to a different value.

Let's have a look at a simple example in .NET:

```
public Person : INotifyPropertyChanged
{

  string firstName, lastName;
   public event NotifyPropertyChangedEventHandler PropertyChanged;

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

 public string FirstName
```

```
  {
   get { return this.firstName; }
  set
    {
       this.firstName = value;
       this.OnPropertyChanged(“FirstName”);
       this.OnPropertyChanged(“FullName”);
  }

public string LastName
  {
   get { return this.lastName; }
  set
    {
       this.lastName = value;
       this.OnPropertyChanged(“LastName”);
       this.OnPropertyChanged(“FullName”);
  }

  public string FullName { get { return string.Format( “{0} {1}“,
this.firstName, this.lastName); }}

}
```

Properties eventually depend on a set of fields, and we have to raise the PropertyChanged for a property whenever we change a field that affects it.

Shouldn’t it be possible for the compiler to do this work automatically for us? The long answer is detecting dependencies between fields and properties is a daunting task if we consider all corner cases that can happen: properties can have dependencies on fields of other objects, they can call other methods, or even worse, they can call virtual methods or delegates unknown to the compiler. So, there is no general solution to this problem, at least if we expect compilation times in seconds or minutes and not hours or days. However, in real life, a large share of properties is simple enough to be fully understood by a compiler. So the short answer is, yes, a compiler could generate notification code for more than 90% of all properties in a typical application.

In practice, the same class could be implemented as follows:

```
[NotifyPropertyChanged]
public Person
{

public string FirstName { get; set; }
public string LastName { get; set; }
public string FullName { get { return string.Format( “{0} {1}“,
this.FirstName, this.LastName); }}

}
```

This code tells the compiler *what* to do (implement INotifyPropertyChanged) and not *how* to do it.

Boilerplate Code is an Anti-Pattern

The Observer (INotifyPropertyChanged) pattern is just one example of pattern that usually causes a lot of boilerplate code in large applications. But a typical source base is full of patterns generating a lot of boilerplate. Even if they are not always recognized as “official” design patterns, they are patterns because they are massively repeating among a code base. The most common causes of code repetition are:

- Tracing, logging
- Precondition and invariant checking
- Authorization and audit
- Locking and thread dispatching
- Caching
- Change tracking (for undo/redo)
- Transaction handling
- Exception handling

These features are difficult to encapsulate using normal OO techniques and hence why they're often implemented using boilerplate code. Is that such a bad thing?

Yes.

Addressing cross-cutting concerns using boilerplate code leads to violation of fundamental principles of good software engineering

- The **Single Responsibility Principle** is violated when multiple concerns are being implemented in the same method, such as Validation, Security, INotifyPropertyChanged, and Undo/Redo in a single property setter.
- The **Open/Closed Principle,** which states that software entities should be open for extension, but closed for modification, is best respected when new features can be added without modifying the original source code.
- The **Don’t Repeat Yourself** principle abhors code repetition coming out of manual implementation of design patterns.
- The **Loose Coupling** principle is infringed when a pattern is implemented manually because it is difficult to alter the implementation of this pattern. Note that coupling can occur not only between two components, but also between a component and a conceptual design. Trading a library for another is usually easy if they share the same conceptual design, but adopting a different design requires many more modifications of source code.

Additionally, boilerplate renders your code:

- **Harder to read and reason with** when trying to understand what it's doing to address the functional requirement. This added layer of complexity has a huge bearing on the cost of maintenance considering software maintenance consists of reading code 75% of the time!

- **Larger**, which means not only lower productivity, but also higher cost of developing and maintaining the software, not counting a higher risk of introducing bugs.
- **Difficult to refactor** and change. Changing a boilerplate (fixing a bug perhaps) requires changing all the places where the boilerplate code had been applied. How do you even accurately identify where the boilerplate is used throughout your codebase which potentially span across many solutions and/or repositories? Find-and-replace...?

If left unchecked, boilerplate code has the nasty habit of growing around your code like vine, taking over more space each time it is applied to a new method until eventually you end up with a large codebase almost entirely covered by boilerplate code. In one of my previous teams, a simple data access layer class had over a thousand lines of code where 90% was boilerplate code to handle different types of SQL exceptions and retries.

I hope by now you see why using boilerplate code is a terrible way to implement patterns. It is actually an anti-pattern to be avoided because it leads to unnecessary complexity, bugs, expensive maintenance, loss of productivity and ultimately, higher software cost.

Design Pattern Automation and Compiler Extensions

In so many cases the struggle with making common boilerplate code reusable stems from the lack of native meta-programming support in mainstream statically typed languages such as C# and Java.

The compiler is in possession of an awful lot of information about our code normally outside our reach. Wouldn't it be nice if we could benefit from this information and write compiler extensions to help with our design patterns?

A smarter compiler would allow for:

1. **Build-time program transformation:** to allow us to add features whilst preserving the code semantics and keeping the complexity and number of lines of code in check, so we can automatically implement parts of a design pattern that can be automated;
2. **Static code validation:** for build-time safety to ensure we have used the design pattern correctly or to check parts of a pattern that cannot be automated have been implemented according to a set of predefined rules.

Example: 'using' and 'lock' keywords in C#

If you want proof design patterns can be supported directly by the compiler, there is no need to look further than the *using* and *lock* keywords. At first sight, they are purely redundant in the language. But the designers of the language have recognized their importance and have created a specific keyword for them.

Let's have a look at the *using* keyword. The keyword is actually a part of the larger Disposable Pattern, composed of the following participants:

- **Resources Objects** are objects consuming any external resource, such as a database connection.

- **Resource Consumers** are instruction block or objects that consume Resource Objects during a given lifetime.

The Disposable Pattern is ruled by the following principles:

1. Resource Objects must implement IDisposable.
2. Implementation of IDisposable.Dispose must be idempotent, i.e. may be safely called several times.
3. Resource Objects must have a finalizer (called *destructor* in C++).
4. Implementation of IDisposable.Dispose must call GC.SuppressFinalize.
5. Generally, objects that store Resource Objects into their state (field) are also Resource Objects, and children Resource Objects should be disposed by the parent.
6. Instruction blocks that allocate and consume a Resource Object should be enclosed with the *using* keyword (unless the reference to the resource is stored in the object state, see previous point).

As you can see, the Disposable Pattern is richer than it appears at first sight. How is this pattern being automated and enforced?

- The core .NET library provides the *IDisposable* interface.
- The C# compiler provides the *using* keyword, which automates generation of some source code (a *try/finally* block).
- FxCop can enforce a rule that says any disposable class also implements a finalizer, and the Dispose method calls GC.SuppressFinalize.

Therefore, the Disposable Pattern is a perfect example of a design pattern directly supported by the .NET platform.

But what about patterns not intrinsically supported? They can be implemented using a combination of class libraries and compiler extensions. Our next example also comes from Microsoft.

Example: Code Contracts

Checking preconditions (and optionally postconditions and invariants) has long been recognized as a best practice to prevent defects in one component causing symptoms in another component. The idea is:

- every component (every class, typically) should be designed as a “cell”;
- every cell is responsible for its own health therefore;
- every cell should check any input it receives from other cells.

Precondition checking can be considered a design pattern because it is a repeatable solution to a recurring problem.

Microsoft Code Contracts (http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx) is a perfect example of design pattern automation. Based on plain-old C# or Visual Basic, it gives you an API for expressing validation rules in the form of pre-conditions, post-conditions, and object invariants.

However, this API is not just a class library. It translates into build-time transformation and validation of your program.

I won't delve into too much detail on Code Contracts; simply put, it allows you to specify validation rules in code which can be checked at build time as well as at run time. For example:

```
public Book GetBookById(Guid id)
{
    Contract.Requires(id != Guid.Empty);

    return Dal.Get<Book>(id);
}

public Author GetAuthorById(Guid id)
{
    Contract.Requires(id != Guid.Empty);

    return Dal.Get<Author>(id);
}
```

Its binary rewriter can (based on your configurations) rewrite your built assembly and inject additional code to validate the various conditions that you have specified. If you inspect the transformed code generated by the binary rewriter you will see something along the lines of:

```
public Book GetBookById(Guid id)
{
    if (__ContractsRuntime.insideContractEvaluation <= 4)
    {
        try
        {
            ++__ContractsRuntime.insideContractEvaluation;
            __ContractsRuntime.Requires(id != Guid.Empty, (string)null, "id != Guid.Empty");
        }
        finally
        {
            --__ContractsRuntime.insideContractEvaluation;
        }
    }
    return Dal.Get<Program.Book>(id);
}

public Author GetAuthorById(Guid id)
{
    if (__ContractsRuntime.insideContractEvaluation <= 4)
    {
        try
        {
            ++__ContractsRuntime.insideContractEvaluation;
            __ContractsRuntime.Requires(id != Guid.Empty, (string)null, "id != Guid.Empty");
        }
        finally
        {
            --__ContractsRuntime.insideContractEvaluation;
        }
    }
    return Dal.Get<Program.Author>(id);
}
```

For more information on Microsoft Code Contracts, please read Jon Skeet's excellent InfoQ article here (http://www.infoq.com/articles/code-contracts-csharp).

Whilst compiler extensions such as Code Contracts are great, officially supported extensions usually take years to develop, mature, and stabilize. There are so many different domains, each with its own set of problems, it's impossible for official extensions to cover them all.

What we need is a **generic framework** to help automate and enforce design patterns in a disciplined way so we are able to tackle domain-specific problems effectively ourselves.

Generic Framework to Automate and Enforce Design Patterns

It may be tempting to see dynamic languages, open compilers (such as Roslyn), or re-compilers (such as Cecil) as solutions because they expose the very details of abstract syntax tree. However, these technologies operate at an excessive level of abstraction, making it very complex to implement any transformation but the simplest ones.

What we need is a high-level framework for compiler extension, based on the following principles:

1. **Provide a set of transformation primitives**, for instance:
    a. intercepting method calls;
    b. executing code before and after method execution;
    c. intercepting access to fields, properties, or events;
    d. introducing interfaces, methods, properties, or events to an existing class.

2. **Provide a way to express where primitives should be applied**: it's good to tell the complier extension you want to intercept some methods, but it's even better if we know which methods should be intercepted!

3. **Primitives must be safely composable**

    It's natural to want to be able to apply multiple transformations to the same location(s) in our code, so the framework should give us the ability to compose transformations.

    When you're able to apply multiple transformations simultaneously some transformations might need to occur in a specific order in relation to others. Therefore the ordering of transformations needs to follow a well-defined convention but still allow us to override the default ordering where appropriate.

4. **Semantics of enhanced code should not be affected**

    The transformation mechanism should be unobtrusive and leave the original code unaltered as much as possible whilst at the same time providing capabilities to validate the transformations statically. The framework should not make it too easy to "break" the intent of the source code.

5. **Advanced reflection and validation abilities**

By definition, a design pattern contains rules defining how it should be implemented. For instance, a locking design pattern may define instance fields can only be accessed from instance methods of the same object. The framework must offer a mechanism to query methods accessing a given field, and a way to emit clean build-time errors.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of concerns.

An *aspect* is a special kind of class containing code transformations (called *advices*), code matching rules (barbarically called *pointcuts*), and code validation rules. Design patterns are typically implemented by one or several aspects. There are several ways to apply aspects to code, which greatly depend on each AOP framework. Custom attributes (*annotations* in Java) are a convenient way to add aspects to hand-picked elements of code More complex pointcuts can be expressed declaratively using XML (e.g. Microsoft Policy Injection Application Block) or a Domain-Specific Language (e.g. AspectJ or Spring), or programmatically using reflection (e.g. LINQ over System.Reflection with PostSharp).

The *weaving* process combines advice with the original source code at the specified locations (not less barbarically called *joinpoints*). It has access to meta-data about the original source code so, for compiled languages such as C# or Java, there is opportunity for the static weaver to perform static analysis to ensure the validity of the advice in relation to the pointcuts where they are applied.

Although aspect-oriented programming and design patterns have been independently conceptualized, AOP is an excellent solution to those who seek to automate design patterns or enforce design rules. Unlike low-level metaprogramming, AOP has been designed according to the principles cited above so anyone, and not only compiler specialists, can implement design patterns.

AOP is a programming paradigm and not a technology. As such, it can be implemented using different approaches. AspectJ, the leading AOP framework for Java, is now implemented directly in the Eclipse Java compiler. In .NET, where compilers are not open-source, AOP is best implemented as a re-compiler, transforming the output of the C# or Visual Basic compiler. The leading tool in .NET is PostSharp (see below). Alternatively, a limited subset of AOP can be achieved using dynamic proxies and service containers, and most dependency injection frameworks are able to offer at least method interception aspects.

Example: Custom Design Patterns with PostSharp

PostSharp is a development tool for the automation and enforcement of design patterns in Microsoft .NET and features the most complete AOP framework for .NET.

To avoid turning this article into a PostSharp tutorial, let's take a very simple pattern: dispatching of method execution back and forth between a foreground (UI) thread and a background thread. This pattern can be implemented using two simple aspects: one that dispatches a method to the background thread, and another that dispatches it to the foreground thread. Both aspects can be compiled by the free PostSharp Express. Let's look at the first aspect: *BackgroundThreadAttribute*.

The generative part of the pattern is simple: we just need to create a Task that executes that method, and schedule execution of that Task.

```
[Serializable]
public sealed class BackgroundThreadAttribute : MethodInterceptionAspect
{
    public override void OnInvoke(MethodInterceptionArgs args)
    {
        Task.Run( args.Proceed );
    }
}
```

The *MethodInterceptionArgs* class contains information about the context in which the method is invoked, such as the arguments and the return value. With this information, you will be able to invoke the original method, cache its return value, log its input arguments, or just about anything that's required for your use case.

For the validation part of the pattern, we would like to avoid having the custom attribute applied to methods that have a return value or a parameter passed by reference. If this happens, we would like to emit a build-time error. Therefore, we have to implement the *CompileTimeValidate* method in our *BackgroundThreadAttribute* class:

```
// Check that the method returns 'void', has no out/ref argument.
public override bool CompileTimeValidate( MethodBase method )
{

  MethodInfo methodInfo = (MethodInfo) method;


  if ( methodInfo.ReturnType != typeof(void) ||
       methodInfo.GetParameters().Any( p => p.ParameterType.IsByRef ) )
  {
     ThreadingMessageSource.Instance.Write( method, SeverityType.Error, "THR006",
             method.DeclaringType.Name, method.Name );

     return false;
  }

  return true;
}
```

The *ForegoundThreadAttribute* would look similar, using the Dispatcher object in WPF or the BeginInvoke method in WinForms.

The above aspect can be applied just like any other attributes, for example:

```
[BackgroundThread]
private static void ReadFile(string fileName)
{
    DisplayText( File.ReadAll(fileName) );
}

[ForegroundThread]
private void DisplayText( string content )
{
```

```
    this.textBox.Text = content;
}
```

The resulting source code is much cleaner than what we would get by directly using tasks and dispatchers.

One may argue that C# 5.0 addresses the issue better with the *async* and *await* keywords. This is correct, and is a good example of the C# team identifying a recurring problem that they decided to address with a design pattern implemented directly in the compiler and in core class libraries. While the .NET developer community had to wait until 2012 for this solution, PostSharp offered one as early as 2006.

How long must the .NET community wait for solutions to other common design patterns, for instance INotifyPropertyChanged? And what about design patterns that are specific to your company's application framework?

Smarter compilers would allow you to implement your own design patterns, so you would not have to rely on the compiler vendor to improve the productivity of your team.

Downsides of AOP

I hope by now you are convinced that AOP is a viable solution to automate design patterns and enforce good design, but it's worth bearing in mind that there are several downsides too:

1. **Lack of staff preparation**

   As a paradigm, AOP is not taught in undergraduate programs, and it's rarely touched at master level. This lack of education has contributed towards a lack of general awareness about AOP amongst the developer community.

   Despite being 20 years old, AOP is misperceived as a 'new' paradigm which often proves to be the stumbling block for adoption for all but the most adventurous development teams.

   Design patterns are almost the same age, but the idea that design patterns can be automated and validated is recent. We cited some meaningful precedencies in this article involving the C# compiler, the .NET class library, and Visual Studio Code Analysis (FxCop), but these precedencies have not been generalized into a general call for design pattern automation.

2. **Surprise factor**

   Because staffs and students alike are not well prepared, there can be an element of surprise when they encounter AOP because the application has additional behaviors that are not directly visible from source code. Note: what is surprising is the *intended* effect of AOP, that the compiler is doing more than usual, and not any side effect.

   There can also be some surprise of an *unintended* effect, when a bug in the use of an aspect (or in a pointcut) causes the transformation to be applied to unexpected classes and

methods. Debugging such errors can be subtle, especially if the developer is not aware that aspects are being applied to the project.

These surprise factors can be addressed by:

I. IDE integration, which helps to visualize (a) which additional features have been applied to the source displayed in the editor and (b) to which elements of code a given aspect has been applied. At time of writing only two AOP frameworks provide correct IDE integration: AspectJ (with the AJDT plug-in for Eclipse) and PostSharp (for Visual Studio).
II. Unit testing by the developer – aspects, as well as the fact that aspects have been applied properly, must be unit tested as any other source code artifact.
III. Not relying on naming conventions when applying aspects to code, but instead relying on structural properties of the code such as type inheritance or custom attributes. Note that this debate is not unique to AOP: convention-based programming has been recently gaining momentum, although it is also subject to surprises.

3. **Politics**

Use of design pattern automation is generally a politically sensitive issue because it also addresses separation of concerns within a team. Typically, senior developers will select design patterns and implement aspects, and junior developers will use them. Senior developers will write validation rules to ensure hand-written code respects the architecture. The fact that junior developers don't need to understand the whole code base is actually the intended effect.

This argument is typically delicate to tackle because it takes the point of view of a senior manager, and may injure the pride of junior developers.

Ready-Made Design Pattern Implementation with PostSharp Pattern Libraries

As we've seen with the Disposable Pattern, even seemingly simple design patterns can actually require complex code transformation or validation. Some of these transformations and validations are complex but still possible to implement automatically. Others can be too complex for automatic processing and must be done manually.

Fortunately, there are also simple design patterns that can be automated easily by anyone (exception handling, transaction handling, and security) with an AOP framework.

After many years of market experience, the PostSharp team began to provide highly sophisticated and optimized ready-made implementations of the most common design patterns after they realized most customers were implementing the same aspects over and over again.

PostSharp currently provides ready-made implementations for the following design patterns:

- Multithreading: reader-writer-synchronized threading model, actor threading model, thread-exclusive threading model, thread dispatching;
- Diagnostics: high-performance and detailed logging to a variety of back-ends including NLog and Log4Net;
- INotifyPropertyChanged: including support for composite properties and dependencies on other objects;
- Contracts: validation of parameters, fields, and properties.

Now, with ready-made implementations of design patterns, teams can start enjoying the benefits of AOP without learning AOP.

Summary

So-called high-level languages such as Java and C# still force developers to write code at an irrelevant level of abstraction. Because of the limitations of mainstream compilers, developers are forced to write a lot of boilerplate code, adding to the cost of developing and maintaining applications. Boilerplate stems from massive implementation of patterns by hand, in what may be the largest use of copy-paste inheritance in the industry.

The inability to automate design pattern implementation probably costs billions to the software industry, not even counting the opportunity cost of having qualified software engineers spending their time on infrastructure issues instead of adding business value.

However, a large amount of boilerplate could be removed if we had smarter compilers to allow us to automate implementation of the most common patterns. Hopefully, future language designers will understand design patterns are first-class citizens of modern application development, and should have appropriate support in the compiler.

But actually, there is no need to wait for new compilers. They already exist, and are mature. Aspect-oriented programming was specifically designed to address the issue of boilerplate code. Both AspectJ and PostSharp are mature implementations of these concepts, and are used by the largest companies in the world. And both PostSharp and Spring Roo provide ready-made implementations of the most common patterns. As always, early adopters can get productivity gains several years before the masses follow.

Eighteen years after the Gang of Four's seminal book, isn't it time for design patterns to become adults?