POSTSHARP

# 10 Reasons You MUST Consider **Pattern-Aware Programming**

Developers spend up to 20% of their time writing repetitive code that machines could generate more reliably. Automate the boring side of programming and get a 19x ROI.

**Developers spend up to 20% of their time writing repetitive code that machines could generate more reliably. Automate the boring side of programming and get a 19x ROI.**

Within this document we discuss the problem of duplicated source code, also known as the notorious *boilerplate code* that stems from manual implementation of patterns.

We challenge the notion that patterns in software development are limited to architecture and design but do not apply to the implementation itself. Instead, we assert that the absence of support for patterns in programming languages is a chief cause of boilerplate code. We propose introducing support for patterns into mainstream programming languages—a concept we name *pattern-aware compiler extensions.*

This guide is written for all organizations for which software development is critical, but that struggle with high development cost, long time to market, poor quality (even random defects) and shortage of qualified engineers (with the need for engineers to do meaningful work, not repeating mundane tasks). It is especially written for CTOs, software architects and senior developers in software-driven organizations—specifically in financial, insurance, healthcare, energy and IT industries.

## Is there a real problem? Do I need to address this?

As a developer, how many times have you found yourself:

- copying and pasting blocks of code to implement a functionality (e.g. logging)?
- repeating the same pattern of code hundreds or thousands of times—and wishing so-called “modern” programming languages were a bit smarter?
- trying to understand the business logic behind a codebase cluttered with technical code such as exception handling, transaction management, audit or thread synchronization?
- struggling to add or modify functionality in an existing software?
- debugging random failures in multi-threaded applications?
- wondering why it’s so hard to build enterprise-grade software when the prototype seemed so simple?

If you’ve answered yes to any of the above questions, then you have a problem with boilerplate code.

What is boilerplate? Why is it so ubiquitous? And more importantly, what can we do about it? This is what you will learn in this white paper.

## The problem: existing compilers don’t support patterns

Whether it’s architecture, carpentry or mechanics, patterns are essential to all construction and engineering disciplines. Learning not only how to implement patterns, but also when and why to choose them, is an important part of the education of professionals of these disciplines.

Software engineering is no exception. Patterns have been successfully applied to software architecture and software design, two preliminary steps of software constructions. As a result, today’s software engineers are trained to *think* in terms of patterns. However, when it comes to *implementation,* developers don’t have the right tools.

Conventional programming languages miss a concept of *patterns.* Therefore, software developers are forced to implement patterns *by hand.* Just like artisan carpenters would manufacture dozens of almost (not totally) identical joints to build a cabinet, a software developer would add *almost* identical exception handling logic to hundreds of functions by hand. The difference, however, is that the artisan carpenter produces a piece of art, while the software developer builds a utilitarian application.

You would not hire artisan carpenters to build a utilitarian industrial building, would you?

Developers call this repetitive code *boilerplate.* Good developers consider boilerplate code as a boring but necessary part of their work. However, as this white paper will show, boilerplate code is not inevitable.

Before looking into alternatives, take a few minutes to think about the cost of boilerplate code for your organization.

# What is boilerplate costing you?

Boilerplate code is a major source of pain in enterprise development. It has the following consequences.

## 1. High development effort

- **High development costs.** Some application features require a large amount of repetitive code when implemented with existing mainstream compiler technologies. Source code is a liability and needs to be maintained. Empirical research shows that the total lifetime cost of a line of code is approximately $14 for medium enterprise projects[1]. This cost also applies to boilerplate code.

- **Long time to market.** The time developers spend writing repetitive code is time they cannot spend focusing on what actually matters. Businesses that come late to market can miss opportunities.

## 2. Poor quality software

- **High number of defects.** Every line of code has a possibility of defect, but code that stems from copy-paste programming is more likely than others to be buggy because subtle differences are often overlooked. Additionally, empirical research[2] shows that features that are implemented across a large number of artifacts are more likely to contain defects.

- **Lack of robustness.** The difference between a prototype and an enterprise-grade application partly lies in features such as exception handling, transaction management and auditor caching. These features typically exhibit the most repetitive patterns and are tedious and boring to implement; therefore, reliability features are often deliberately omitted, unintentionally forbidden in some parts of the applications or simply left untested and unreliable.

- **Multi-threading issues.** Conventional object-oriented programming languages allow programs to be simultaneously executed by several threads on multi-core CPUs, but these languages do not guarantee that the program execution is safe. As a result, developers have to build thread safety manually using low-level thread synchronization artifacts such as locks, events or interlocked operations.

  In any sizable application, managing locks becomes a nightmare. The problem occurs when developers forget a lock (and it *always* happens). No big red sign appears immediately, instead, the defect will hide in the code and will appear *randomly,* most of the time in production. This is why thread safety problems are so difficult to diagnose.

## 3. Difficulty to add/modify functionality

- **Unreadable code that's difficult to maintain.** Business code is often littered with low-level, non-functional requirements and is more difficult to understand and maintain, especially when the initial developer left. When you know that maintenance accounts for 55% to 95% of the total cost of a software system[3], and that developers spend *half of their time* on average trying to understand existing code, you realize why it is so crucial to keep source code succinct and readable.

- **Strong coupling.** Conventional programming languages don't give you the tools to properly decompose the pattern itself from its usage. As any software architect knows, poor problem decomposition results in duplicate code and strong coupling, and strongly coupled architectures are more difficult to maintain.

  For example, suppose you want to modify a pattern to change the details of an audit policy. If your audit policy requires you to add code to every single business function being audited, you will end up modifying thousands of files. This is why weak coupling and proper problem decomposition is so important.

## 4. Slow ramp-up of new team members

- **Too much knowledge required.** When new team members come to work on a specific feature, they often must first learn about caching, threading and other highly technical issues before being able to contribute to the business value—an example of bad division of labor.

- **Long feedback loops.** Even with small development teams, common patterns like diagnostics, logging, threading, data binding and undo/redo can be handled differently by each developer. Architects have to make sure new team members understand and follow the internal design standards and have to spend more time on manual code reviews—delaying progress while new team members wait to get feedback from code review.

1 One developer typically produces 600 lines of code per month (McConnell, Steve (2006). Software Estimation: Demystifying the Black Art. Pearson Education). We used the estimate that the yearly cost of a developer is $100,000.

2 Eaddy M., Zimmerman T., Sherwood K., Garg V., Murphy G., Nagappan N., Aho A. (2008). Do Crosscutting Concerns Cause Defects? IEEE Transactions on Software Engineering, Vol. 34, No. 4, July/August.

3 Roberto Minelli, Andrea Mocci and Michele Lanza (2015). I Know What You Did Last Summer—An Investigation of How Developers Spend Their Time. Published in 2015 IEEE 23rd International Conference on Program Comprehension.

# Existing technologies

Several different tools can be used for pattern automation although they have not been designed specifically for this purpose. These tools are often found under the categories of dependency injection framework, aspect-oriented programming framework, meta-programming framework and assembly post-processor.

## Dynamic Proxies and Dependency Injection Frameworks

A proxy is a common design pattern in which a class operates as a gateway to one or more other classes. A dynamic proxy is a class generated on demand by a framework. Because they are generated on-the-fly, dynamic proxies can add additional dynamic behaviors to objects, such as logging or transaction handling.

Dynamic proxies aren't flexible enough to implement all or even moderately complex patterns on all types of classes. Dynamic proxies also require special methods of instantiation, encourage the alteration of the architecture to conform to the limitations of dynamic proxies, and lack direct IDE support. While dynamic proxies provide some of the benefits of pattern-aware compilers, they fall far short of delivering on its full promise.

## Code Generators

Every developer who has used an IDE is familiar with the code generation paradigm, in which UI gestures or models, such as data schemas, are converted into executable code. Visual Studio and the .NET Framework define several components that attempt to turn such cross-cutting behaviors as multi-threading into a matter of code generation. This is of only limited effectiveness since it cannot be used to add new behaviors to existing code.

Tools that generate source code from a model generally cause more problems than resolutions. Although the initial source code does not need to be written by hand, it still needs to be fine-tuned and then maintained. In other words, these tools don't only generate source code, but also a lot of liabilities.

## Refactoring Tools

There are many good refactoring tools on the market. Some of those allow you to automatically generate code for a few patterns. Refactoring tools are productivity extensions to the code editor, not to the language itself. Refactoring tools allow you to write boilerplate code faster, but they don't allow you to reduce the amount of boilerplate code. Boilerplate code still needs to be maintained during years and can still hide defects.

## Meta-Programming

Meta-programming, which has been made popular by dynamic languages and more recently by the Roslyn compiler, allows programs to change their behavior extensively at runtime and/or build time.

Classic meta-programming tools operate at a very low level of abstraction, for instance at MSIL level for post-compiler tools or at syntax level for pre-compiler tools. Meta-programming is powerful, but complex to learn and implement. Developers who have sought to implement meta-programming as a comprehensive solution to the problem of pattern implementation have found themselves writing their own aspect-oriented framework (see below).

## Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) was invented at the Xerox Palo Alto Research Center (PARC) in 1997 when Xerox engineers designed AspectJ™, an AOP extension to the Java language. AspectJ is today the undisputed leader in AOP solutions for Java, and there are AOP extensions for most programming languages. AOP was initially designed to address the problem of cross-cutting concerns.

AOP is an excellent technology and is extremely useful to automate the implementation of patterns. However, AOP was not designed specifically with patterns in mind. AOP and design patterns were conceptualized independently. AOP needs to be better conceptualized and communicated so it can get the attention it deserves from the whole software development industry, not just computer science majors.

AOP alone is not sufficient to implement patterns. Additionally, we need the ability to analyze programs more deeply than AOP usually does.

## Static Program Analysis

Static program analysis is the analysis of a program without executing it. An example of such tools is Microsoft Code Analysis or Microsoft Code Contracts. Analyzing complex patterns requires advanced static analysis abilities. The better analysis you can do, the smarter the pattern automation can be, and the easier it will be for the developer using the pattern.

Today's mainstream compilers rely on fairly simple analysis. They were designed a few decades ago to perform fast enough within the constraints of the typical CPUs of that time. As typical development machines get more and more computing power, compilers can rely on more complex analysis and be more intelligent, saving more work from developers.

Wouldn't it be nice to produce high-quality, easy-to-maintain software with less development effort and faster ramp-up of new team members... without having to replace your existing compiler?

## The solution: pattern-aware compiler extensions

Pattern-aware programming *extends* conventional object-oriented programming with a concept of *pattern,* which becomes a first-class element of the programming language.

Are we arguing you should rewrite your applications and retrain your team in a totally new programming language? Of course not!

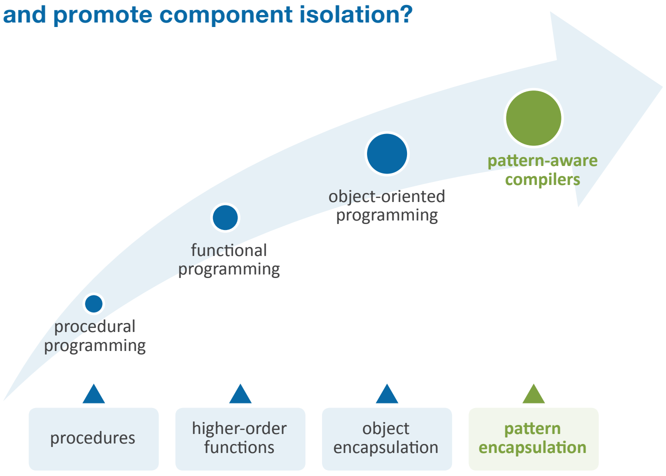
Most mainstream programming languages can be extended with a concept of pattern, avoiding the cost of rewriting applications in a new language.

Because patterns are supported by the compiler extension (100% compatible with your existing compiler), they do not need to be manually implemented as boilerplate code. Features such as data binding, logging, transactions are implemented in a cleaner, more concise way, making development and maintenance much easier.

Pattern-aware programming is a relatively new concept that defines the objective that programming languages should include a concept of pattern, so that the programming language matches the level of abstraction of human reasoning. In other words, pattern-aware programming acknowledges that programming is a human cognitive activity.

Programming languages should be designed for their users: humans. Therefore, they should be designed with regards to human cognitive habits and not solely for mathematical elegance or for the ease of implementing the compiler itself. The result of better designed programming languages is more succinct and more understandable source code.

### How do programming language avoid code repetition and promote component isolation?



## Why consider a pattern-aware compiler?

### 1. Stop writing boilerplate code and deliver faster

- **Fewer lines of code mean fewer hours of work**. Patterns are repetitive, with little or no decision left to the developer. However, repetition is exactly what computers are good at. Let the compiler do the repetitive work and save development time and costs immediately.

### 2. Build more reliable software

- **Cleaner code means fewer defects.** With a pattern-aware compiler eliminating the boilerplate, your code becomes easier to read, understand and modify, and contains fewer defects.

- **Reliability becomes much more affordable.** Because they no longer require so much manual coding, reliability features such as caching or exception handling are much easier and cheaper to implement, so you can spend your extra time building a more robust app.

### 3. Add/modify functionality more easily after the first release

- **Cleaner and shorter code is easier to understand.** After the initial release, too much development time is spent reading and analyzing source code, especially if the initial developer leaves. With minimized boilerplate code, developers can easily focus on business logic and spend much less time trying to understand the source code.

- **Better architecture is future-proof.** When using a pattern-aware compiler, features like logging, exception handling or transactions are no longer scattered among thousands of files but they are defined in one place, making it much easier and faster to modify when necessary.

### 4. Help new members contribute quicker

- **Achieve a better division of labor.** Using a pattern-aware compiler makes the introduction of new or junior team members less onerous since they can focus on simpler, more business logic-oriented tasks rather than having to waste time learning complex architectural structures.

- **Implement a tighter feedback loop.** A pattern-aware compiler can validate that hand-written code respects patterns or guidelines, and it can detect nonconformities *at build time* instead of during code reviews, testing, or in production.

## Top 10 features to look for

What should you look for when searching for a pattern-aware programming tool?

### 1. Ready-made pattern implementations

Would you build a graphing component or data grid yourself? Probably not. Unless you have very specific requirements, you would rather buy one tested by hundreds of customers from a specialized vendor instead of paying your own developers to reinvent something that has already been done.

The same thinking applies to patterns. While automating very simple patterns can be straightforward with most frameworks, implementing more advanced patterns can require a significant amount of work and testing. When choosing a pattern-aware compiler, check that there are ready-made implementations of most of the patterns you need—or at least documented samples.

### 2. Threading models

Starting new threads and tasks in .NET languages is simple, but ensuring that objects are thread-safe is not part of mainstream programming languages. Several design patterns called *threading models* have been described to guarantee your code executes safely even when used from multiple threads. However, threading models are typically too complex to be implemented manually.

Threading models raise the level of abstraction at which multi-threading is addressed. Unlike working directly with locks and other low-level threading primitives, threading models *decrease* the number of lines of code, the number of defects, and reduce development and maintenance costs—without having to have expertise in multi-threading.

Several threading models exist and each is suitable in a different situation. Some frameworks will force you into one specific threading model (like functional programming forces you into the immutable model) and claim it is a universal threading model. This is not true. Choose a framework that offers several threading models and allows you to choose the threading model of each class separately.

### 3. Ability to automate complex patterns

Let's face it. There will always be patterns that are specific to your project and for which no ready-made solution will be available. In this situation, you need a toolkit to build automation for your own custom patterns.

Most products make it possible to implement simple patterns requiring the interception of method invocations. For instance, transaction and exception handling are fairly simple to implement in any framework. It's definitively a good thing that simple requirements are simple to implement, but some frameworks won't let you implement more complex patterns because the framework itself is either simplistic or designed for another purpose (such as dependency injection).

When scouting for a pattern automation tool, look for the following characteristics:

- **Rich set of primitive transformations.** Besides method invocations, does the framework support interception of field accesses or event firing? Does it allow you to introduce new interfaces, methods or even custom attributes?

- **Patterns composed of several transformations.** Simple patterns like transaction scopes are composed of a single transformation. However, more complex patterns like data binding require several transformations to work closely together.

- **Pattern composition.** What happens when you add several patterns to the same class or method? Good frameworks let you express the pattern order and conflict in a consistent manner.

- **Simple API.** Some frameworks allow you to perform arbitrary transformations of MSIL code or source code AST but this is overly complex. In good frameworks, the complexity of the API matches the complexity of what you're trying to achieve.

### 4. Ease to add patterns to source code

Whether you plan to use a ready-made pattern or you've built a custom one, you need to tell the pattern-aware compiler to apply the pattern to the desired pieces of code. Depending on the situation, you will value different strategies:

- **Custom attributes.** Add a custom attribute to every single method, field, or class to which you want to add the pattern. This works great in situations where you need to hand-pick all targets, such as adding a caching pattern.

- **Multicast attributes.** What if you want to add logging to all public methods of a namespace and there are hundreds of them? Look for frameworks that allow you to do that in a single line of code.

- **Inheritance.** Some patterns such as NotifyPropertyChanged naturally apply to all children of a class when applied to the parent class.

- **XML file.** When you apply security policies to a code base, it is more convenient to do it in a central XML file instead of having the policy spread in several files. It makes the security review easier.

- **Programmatic (build-time).** To implement policies composed of several patterns, it is convenient to be able to add patterns to your code programmatically, by executing your code at build inside the compiler.

## 5. Compatibility with your existing codebase

Despite the hype around functional programming languages, C# and VB still remain an excellent platform for enterprise development. Pattern-aware compiler extensions respect your technology assets and will work incrementally with your existing code base—there is no need for a full rewrite or redesign.

- **Design neutrality.** Due to implantation restrictions, frameworks based on dependency injection can only inject behaviors at the boundary between a service client and a service implementation. They don't make it possible to add behaviors to private methods, fields, or any non-virtual and non-interface method. Therefore, if you try to use dependency injection frameworks to implement patterns, you will be tempted to fragment your code into more classes and interfaces than what would make sense for dependency injection.

  Dependency injection frameworks are great for dependency injection. Don't let them dictate every single aspect of your design.

- **Plain C# and VB.** Languages such as F#, Scala, Nemerle, Python, Ruby or JavaScript exhibit many pattern-aware features, but they require you to rewrite your code and retrain your staff. With a pattern-aware compiler extension, your code is still 100% C# and VB, and it is still compiled by the proved Microsoft compilers.

- **Cross-platform.** Check whether the tool supports all the platforms you need: .NET Framework, Windows Phone, WinRT, Xamarin, Portable Class Libraries, UWP, .NET Core, etc.

## 6. Build-Time Validation of Patterns

In a typical development structure, only a couple of developers will actually *create* new patterns. These developers are typically members of the "architecture" or "core" team. The large majority of developers just *use* patterns. One of the main responsibilities of the architecture team is to ensure that the rest of the developers are highly productive, and a key element in developer productivity is to *detect errors as early as possible.* The later a defect is detected, the more expensive it is to fix.

This is why it is so important that developers who build aspects also implement validation logic that verifies that the pattern has been applied to a legitimate target.

For instance, it is a good practice to only cache immutable data. If you build a caching aspect, you could enforce that the pattern is only added to methods returning immutable types. Otherwise, you may emit a build-time error.

When looking for a patter-aware compiler, look for the following abilities:

- Validate pattern targets declaratively, by adding custom attributes to the pattern class itself.

- Execute validation code at build-time and emit errors and warnings.

## 7. Architecture Validation

In a large team, it can be really challenging to get all developers to respect conventions. There may be several good ways to solve the same problem, but inside the same project, it is much better if the team agrees on *one* way and everybody respects it. Then this agreed way of doing things becomes a pattern for the whole team.

Traditionally, controlling the respect of conventions, guidelines and patterns was done manually during code reviews—a work-intensive process that is sometimes executed days or weeks after the original code has been written.

With a pattern-aware compiler, you can *automate* the verification of hand-written code against the rules of the pattern the team decided upon. If any anomaly is found, the build will just fail.

When looking for a pattern-aware compiler, look for the following abilities:

- Access assembly metadata at runtime using the familiar System.Reflection API.

- Execute complex queries in the whole assembly, for instance "find all types derived from class C" or "find all methods writing to field F".

- Access expression tree of method bodies.

## 8. Integration with Visual Studio

Many developers have reported being disappointed by free aspect-oriented programming frameworks because they were no longer able to understand and debug their code. After discussion, it appeared that the cause of their disappointment is not a defect of aspect-oriented programming itself, but rather the absence of proper integration of the language extension within the main development environment.

Developers expect the same level of integration for language extensions as for the main language itself. Check for the presence of the following features:

- **Editor enhancements.** One of the first questions you will have when executing pattern-enhanced code is: how do I know which patterns have been applied to this piece of code *without executing the program?* This is the most important question developers ask when they need to understand their code. They don't need to know *how* the patterns are implemented under the hood but *which* patterns have been applied. Pattern-aware compiler extensions can display this information using adornments in the code editor in addition to tooltips.

- **Aspect Browser.** The second question developers ask is: which pieces of code has this pattern been applied to? This is best answered using a tool window displaying all patterns in the solution and their targets.

- **Debugger enhancements.** When you are debugging business logic, do you want to step into caching logic? Probably not. But when you are debugging caching logic, you surely don't want the pattern implementation to be stepped over. To address these two scenarios, the pattern-aware extension should offer an option allowing steps either into or over pattern implementations.

## 9. Run-time performance

Start-up latency, execution speed and memory consumption matter. Whether you're building a mobile app or a back-end server, run-time performance is of paramount importance.

Look for the following features:

- **Build-time code generation.** Unlike proxy-based solutions, build-time tools modify your code at build time. No reflection is needed at run-time.

- **Build-time initialization.** Many patterns make decisions based on the shape of the code which they are applied. Make sure that you can analyze the target code at build-time and use this data at runtime.

## 10. Commercial support

Open-source AOP frameworks and metaprogramming tools pop up regularly. Most often than not, the authors of these project invest a few months into the project. Although it is tempting to believe that having the source code gives you a safety net in case the project is no longer supported, the reality is that such tools are very complex and exhibit a steep learning curve. Your team may be able to fix a bug if necessary, but it is definitively going to distract it from its goal.

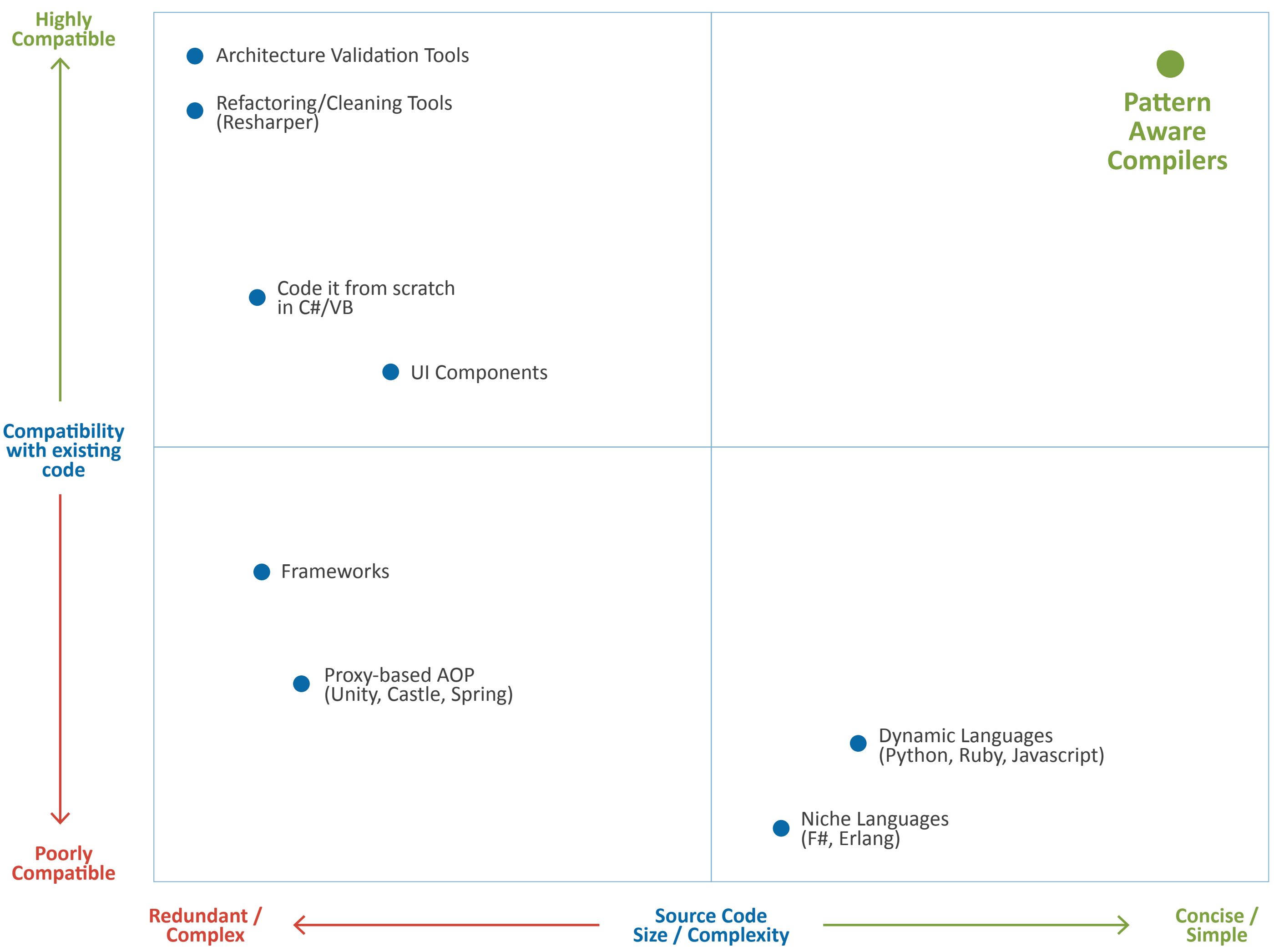
When choosing a framework, make sure it is backed by a stable company with full-time employees assigned to maintenance and support.

## Comparative Matrix

The following matrix shows some of the differences between pattern-aware compilers and alternatives:

| Benefit | Pattern Aware Compilers | Vanilla C# / VB | Refactoring Tools | Dependency Injection |
|---|---|---|---|---|
| Build automation for your own patterns with comprehensive toolkit | Yes | No | No | No |
| Ready-made standard design patterns implementations | Yes | No | No | No |
| Build thread-safe applications | Yes | No | No | No |
| Detect errors and non-compliance before code reviews | Yes | No | No | No |
| Leverage the skills of your most experienced developers to the whole team | Yes | No | No | No |
| Compatible with Pattern-Aware Compilers | Yes | Yes | Yes | Yes |

### The compatible way to write cleaner, simpler code



## What's Holding You Back?

Now that you've seen some of the advantages of pattern-aware compiles, what's holding you back?

Following are some of the typical questions:

### Will compilation be slower?

Yes, as pattern-aware compilers introduce additional steps into the compilation, there is a performance cost. This is the same for custom tools run before C# compiler is executed such as XAML compiler. How large this cost is mostly depends on how extensively the pattern-aware compiler transforms the original program, which depends on how much the patterns are leveraged. For comparison, the best pattern-aware compiler extensions are generally a few times faster than FxCop, which is frequently run for every build in larger companies.

Eventually, the dilemma is to decide what is more important to your organization: lower engineering time and higher quality on one side, or lower machine-processing time on the other side. Arguably, longer build time also means longer waiting times for developers. When the build time becomes too long, developers tend to switch to another task or get distracted, which can be a significant productivity hurdle.

If you fear your build will be too long with a pattern-aware compiler, chances are that it is already borderline at the moment, so you will need to find a solution to this problem anyway. You may want to consider at-build acceleration applications such as IncrediBuild from Xoreax, which parallels builds and distributes tasks over the network, allowing developers to use more CPU resources than the ones available on their own development machine.

Therefore, by combining a smarter but more CPU-intensive compiler with build acceleration technologies, you can win twice: reduce engineering time and reduce build waiting time.

### Do we have to replace our existing compiler?

No. Reputable pattern-aware compiler extensions should always be compatible with your existing Microsoft compiler, providing the tooling and user experience you're used to, including light-bulb integration or errors and warnings displayed in the Visual Studio Error List.

### Microsoft's compilers are not extensible. What is the trick?

It is technically correct that Microsoft's compilers are not extensible for program transformation. Therefore, compiler extensions don't integrate with the compiler itself but with the build tool, namely with MSBuild. Compiler extensions (including for instance Microsoft Code Contracts) transform the binary output of the C# or VB compiler. The format for this output is named MSIL, a well-documented specification (ECMA-335) that, unlike the C# and VB languages themselves, hasn't changed since 2012.

### The only right way to have thread safety is to use purely functional languages

Purely functional languages are thread-safe because they strictly follow the Immutable pattern. Pattern-aware extensions to object-oriented languages can also offer the Immutable pattern with other popular threading models.

A pattern-aware compiler does not force you into a specific programming model. Unlike functional programming, which is mainly popular in academic circles and in some specific industry niches, pattern-aware compilers follow a pragmatic approach where thread safety is achieved through a combination of build-time and run-time verification. Because their objective is not to reach 100% provable soundness at build time, threading models for pattern-aware compilers can focus on providing the maximum thread safety commercially realistic in a business setting.

Note that this approach to thread safety would be insufficient for operating system kernels, aeronautic/space software, real-time financial trading or control of nuclear power plants, but these pieces of critical software typically cost an order of magnitude more than typical business applications.

## What Kind of Returns Should You Expect?

Another thing to look for is the ROI you should expect.  In fact, the bottom line is: if it doesn't save you in development and training time, expense, and/or speed to market, it's not worth using.  ROI is not an abstraction, so let's analyze the results of a mid-sized enterprise development team working on a business software.

Let's consider a 20 person team working on a 3 year project. Suppose the average cost per team member and per year is $100,000, all taxes and overhead costs inclusive (an understatement in many countries). That means that the total project cost is $6,000,000. According to empirical research , a 20 person team would produce approximately 430,000 lines of code during 3 years. This gives us an approximate cost of $14 per line of code.

We cannot quantify all benefits of using a pattern-aware compiler, but empirical research[4] and customer data shows that the number of lines of code required to implement a same set of features decreases by 5% to 25%. Let's take a conservative 10% reduction. Since the cost of developing and debugging software is roughly linear to its number of lines of code, it means that we can expect the total project cost to be $600,000 lower with a pattern-aware compiler than with a traditional programming language.

Now let's compare this to software license costs. Although the whole team benefits from having a smaller code base, only

4 McConnell, Steve (2006-02-22). Software Estimation: Demystifying the Black Art. Pearson Education

developers (suppose there are 15 developers, 4 testers and 1 manager) need to purchase a license. Assuming this software is in a typical price range for development tools ($600-$2000 per developer for the first year, 30%-40% maintenance fee for the next 2 years), we can estimate that the total licensing cost is $20,000.

Let's add the training cost. Typically, just the architecture team (say 2 people) need to acquire a deep understanding (say 3 days). The rest of the development team (13 people) just need a superficial understanding of the concepts (say ½ day). Thus the cost of training the team would be under $11,000.

We have a total cost of adoption of $31,000 for a return of $600,000. Despite all the conservative projections, **this is still a 19x return on investment!** And we just took into account the quantifiable benefits. Hidden multithreading defects, late arrival to market or security leaks can cost you millions and even billions of dollars, and pattern-aware compilers help address these problems too.

Most software has nowhere near this kind of return—making pattern-aware compilers one of the most profitable approaches you can consider.

## What's Next?

Thank you for taking the time to discover the advantages of pattern-aware programming. This document was created by PostSharp Technologies, creators of the #1 pattern-aware compiler extension for C# and VB.

PostSharp started as an open-source project in 2004 and due to its popularity, it soon became a commercial product trusted by over 50,000 developers worldwide and over 1,000 leading corporations. More than 10% of all Fortune 500 companies including Microsoft, Intel, Bank of America, Phillips, NetApp, BP, Comcast, Volkswagen, Hitachi, Deutsche Bank, Bosch, Siemens and Oracle rely on PostSharp to reduce their development and maintenance costs.

With over a decade experience in boilerplate reduction, PostSharp is now the #1 best-selling pattern-aware extension to C# and VB and the only commercially-supported development tool for .NET.

"PostSharp is a pretty amazing piece of software. Abstractions like PostSharp are the whole point of what the computer is supposed to do for us, work that's not fun, like logging and transactions. So why not hide that?"

Scott Hanselman
Principal Program Manager at Microsoft

## PostSharp is trusted by over 50,000 developers worldwide....

| | | | | |
|---|---|---|---|---|
| Microsoft | ORACLE | intel | SolidWorks | KASPERSKY lab |
| pwc | avanade Results Realized | CGI | THALES | HITACHI |
| Bank of America | JPMorgan Chase & Co. | WELLS FARGO | Deutsche Bank | COMMERZBANK |
| SIEMENS | PHILIPS | Humana | GfK | McKESSON |
| TOTAL | CNPC | bp | HESS | HALLIBURTON |
| US AIRWAYS | EADS | Lufthansa | Volkswagen | HONDA |
| Moody's | mitchell | RGA | AON | AXA |
| SKANSKA | BOSCH | LIEBHERR | Schneider Electric | Roche |
| ebay | COMCAST | ZEISS | TESCO | THOMSON REUTERS |

Following are some resources to help you learn more:

- **Download PostSharp** and get a free 45-day trial at **https://www.postsharp.net/download**
- **Read customer stories** at **https://www.postsharp.net/customers.**
- **Register for a free one-to-one demo** at **https://www.postsharp.net/live-demo.**

**PostSharp Technologies**
Namesti 14 rijna, 1307/2
150 00 Prague 5
Czech Republic
**hello@postsharp.net**