Patterns in Software Development

An Idea Whose Time Has Come
Software patterns have been helping us design better software for many years. They capture and effectively transmit highly useful knowledge that was once solely in the minds of gurus.

Ah, Forget It
Most people who think they’re using patterns really aren’t. Tool support for patterns has been a failure, and the first truly great pattern language has yet to be written. If this is as far as we can get in 10 years, it’s time to pack it in.

“Alexander reminded the crowd that the real purpose of any pattern language must be nothing short of improving human existence. That was a lot for a stadium-sized crowd of engineers to swallow.”
— Joshua Kerievsky, Guest Editor

Opening Statement
Joshua Kerievsky

Not Yet “Drunk on God”: A Sobering View of Patterns Progress
James O. Coplien

Can Patterns Be Harmful?
Alan Shalloway

Context Is Key: The Power of Pattern Languages
Steve Berczuk

The Perfection of Informality: Tools, Templates, and Patterns
Dirk Riehle

Singing the Songs of Project Experience: Patterns and Retrospectives
Linda Rising and Esther Derby
Let me answer this question unequivocally. It depends. Figured out I’m a consultant yet? Actually, let me explain. It does depend. It depends on what you think patterns are. Many people have a misunderstanding of patterns, and for those people, patterns can be harmful. However, when people get past their early understanding of patterns and come to comprehend them in a richer way, patterns cannot be harmful. This article will explain both views, so I think that by the end of it, I will have answered the question unequivocally.

THE STRATEGY PATTERN

First, let’s start by examining why patterns can be harmful for someone with an incomplete understanding of design patterns in general. A common definition for a pattern is “a solution to a problem in a context.” In the software community, finding patterns means looking for commonly recurring problems that occur in similar situations (the context). If there is a set of high-quality solutions that take a similar approach to solving the problem, a pattern may be identified.

For example, the recurring problem that the Strategy pattern solves often occurs when your software has several algorithms available to use. If one object is tasked with both using the algorithms and choosing which one to use, that object might have to manage too many things. As more algorithms become available, the object becomes more complicated even though it really isn’t adding any additional functionality (its management of the varying algorithms is just getting more complex). The context is that the object using these different solutions should not be coupled to which one it is using at any particular time. This allows more flexibility from run to run and enables you to add new algorithms as needed. The common solution is to treat all of these algorithms in a similar way (making them interchangeable). This entails having all the algorithms implement a common interface and then invoking them through this interface.

The solution is often thought of as:

1. Making all the algorithms have the same interface.
2. Separating the tasks of “using algorithms” from “choosing which algorithm to use.” (In the pattern nomenclature, the using object is the Context and the object calling the Context is the Client. You transfer the role of choosing which algorithm to use from the Context to the Client.)
3. Making the Client pass a reference to an algorithm to the Context, thereby freeing the Context from any decisions about which algorithm to use.

This solution fixes the problem because you can now add more and more algorithms without increasing the complexity of the Context object. This is a proper use of the Strategy pattern. Later we’ll see a case of an improper use of the Strategy pattern. Examples of a Strategy pattern design and an implementation for a class that needs to be able to perform different kinds of encryption are shown in Figures 1 and 2, respectively.

In this example, we have a Configuration object that knows which algorithm to use. It could just as well have been the Client object that knows which algorithm to use. The Strategy pattern doesn’t explicitly state which object actually knows which algorithm to use, but it does imply that the Context object doesn’t know. This doesn’t add any complexity to the system, however. Somehow, the problem domain will give an indication of which algorithm to use. The Strategy pattern tells you to take advantage of this and to
put this logic outside the scope of the using object.

WHAT ARE PATTERNS REALLY ABOUT?

Many people believe that the Strategy pattern says, “When you have a set of potential algorithms to use, create an interface for all of the algorithms and have the class that uses the algorithms receive a reference to a specific algorithm from a Client object.” This shows the problem in a context and shows how the problem is solved. It boils the pattern down to a class diagram solution, detailing which class or object contains what.

The difficulty is that the pattern is not the sum of the problem, the context, and the solution — the pattern is really about how the forces in the problem occurring in this context relate to each other and can be resolved. It is the relationship of these three issues — problem, context, solution — and the possible resolutions of them; that is really what the pattern is about.

Christopher Alexander1 states as much at the end of The Timeless Way of Building when he says:

At this final stage, the patterns are no longer important: the patterns have taught you to be receptive to what is real [1].

In my classes I joke that instead of telling me this on page 545 of a 549-page book, Alexander could have told me on page 2 and saved me a lot of reading! However, I go on to say that by the time someone reads this, he or she already knows it to be true. The patterns aren’t important — the forces in the problem domain and how to resolve them are.

Of course, at this point, I haven’t shown anything wrong with using the Strategy pattern. Before I do, however, I would like to extend our knowledge of patterns. Then we can see both how misusing a pattern can be harmful and why it is truly a misuse that is causing the problem, not the pattern itself.

PATTERNS AS FORCES

As a first step in shifting our view from the implementation of the Strategy pattern to the forces of the Strategy pattern, let’s look at the primary mandates of the “Gang of Four.”2 Not surprisingly, the Strategy

---

1Christopher Alexander is the architect often credited with inspiring many experts in the software community to investigate patterns in software. His books A Pattern Language [2] and The Timeless Way of Building [1] offer tremendous insights to the software design patterns community.

2The “Gang of Four” is a sobriquet for Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, authors of Design Patterns: Elements of Reusable Object-Oriented Software [4].
pattern reflects all of these mandates:

1. Program to an interface, not an implementation.

2. Favor object composition over class inheritance.

3. Consider what should be variable in your design. Focus on encapsulating the concept that varies [4].

**Program to an interface, not an implementation.** Essentially this means considering the object you are dealing with as a black box. Do not consider how it does its tasks, just consider the interface (set of public methods) that it allows you to use. This is very consistent with Bertrand Meyer’s notions of “Design by Contract” [6]. The Strategy pattern follows this approach because each algorithm can only be accessed by its interface. The Context object is not aware of any algorithm’s implementation.

**Favor object composition over class inheritance.** When learning object-oriented design, many developers are taught that when an alternative way is needed to accomplish something, inheritance should be used. In other words, if you have a class Chip that gets a status of a hardware component, stores it in a string, encrypts that status and then transmits it, you can add a new kind of encryption by subclassing Chip and overriding the encrypt method (see Figure 3). This works well for handling one variation. However, if something else starts to vary (say, how you transmit the status or whether you need to compress it before encrypting), you get an exponentially increasing number of possible combinations. Using inheritance alone will require a correspondingly large number of classes (one for each combination). In older code, we often find an increasingly complex inheritance hierarchy handling many, many special cases and becoming more brittle and harder to change as time goes by.

On the other hand, if you use the Gang of Four’s suggestion, you would have the Chip object contain a reference to an object that handles the encryption for you (see Figure 4). This would allow you to use polymorphism (or any other method of handling variation) to handle different encryptions. If you have other functional variations as well, you could create a reference to each varying object. If you are also following the “design-to-interface” mandate, you can create as many references as needed without greatly increasing complexity.

Some people point out that both cases use inheritance, and that is correct. Notice, however, that in Figure 3 you use inheritance on Chip directly, while in Figure 4 Chip contains a reference to Encryption, which uses inheritance. Therefore, from Chip’s perspective, you achieve the variation in encryption methods through inheritance in Figure 3, while in Figure 4 you use containment on the Encryption object.

Essentially, in the first case you are using inheritance to specialize Chip, while in the second case you are using it to categorize the various encryption algorithms under a common type.

**Consider what should be variable in your design. Focus on encapsulating the concept that varies.**

![Figure 3 — Handling variation with inheritance.](image)

![Figure 4 — Handling variation using composition (and inheritance).](image)
This mandate was a bit confusing to me at first. To me, encapsulation meant “data hiding.” But to the Gang of Four, it means something else: encapsulating the concept that varies is encapsulation of type. Notice in Figure 1 how the Client object has no coupling to the fact that the different concrete implementations (AlgorithmOne, AlgorithmTwo, AlgorithmThree) even exist. Their existence is hidden (hence encapsulated) by Algorithm. Furthermore, the rules for encryption (i.e., which one should be used under what circumstance) are also encapsulated.

THE FORCES IN THE STRATEGY PATTERN

If the word “forces” is not clear to you, think of forces as the issues in your problem domain that you need to consider in order to best solve the problem you are facing. In the case of the Strategy pattern, the forces are as follows:

1. Many potential algorithms (business rules) exist.
2. There can be many ways the algorithms vary:
   - Minimal information required to do their tasks
   - Different ways to instantiate them
   - Different situations in which they apply
   - Different persistence implications
3. Our Client (application) should not be encumbered with having to know all of the business rules possible, as this would both add complexity and limit future variation.

Basically, the Strategy pattern says to:

1. Consider the cost of the Client’s knowing which algorithms it uses. This represents the potential savings of the pattern.
2. Consider the cost of making the algorithms interchangeable. This represents the cost of implementing the pattern.
3. Compare the cost to implement the pattern to the savings it will achieve.
4. Decide whether to use the pattern based on this comparison.

This actually a bit of an oversimplification. Depending on the software development practices you are following, both deciding on what the costs are and how you determine what to do based on them may vary considerably. This is because what you consider to be the savings of the pattern will vary. For example, if you take a “design up front” point of view, you may consider the savings to be all of the savings that will be achieved once the system is complete. In other words, you’ll consider the cost of the Client’s being coupled to the entire set of algorithms to be the cost of all algorithms that the Client will eventually need to know. However, if you are an Extreme Programmer and are following the mantra “Do the simplest thing that could possibly work,” you will likely consider the cost of only those algorithms that are present in this iteration (and there may be only one algorithm in this iteration).

This “pay now” or “pay later” philosophical difference is often at the heart of why some people say patterns can be harmful. Someone with a design-up-front point of view

---

3To be honest, I didn’t initially understand the one about composition either, but I wasn’t confused — just wrong.

4This is an example of why James Trott and I subtitled our book “A New Perspective on Object-Oriented Design” [7]. We didn’t mean that we had come up with a new perspective but that design patterns themselves create a new perspective. We were merely trying to explain it.

5The reader is, I hope, aware of Extreme Programming. If not, I highly suggest reading either Kent Beck’s Extreme Programming Explained [3] or Bob Martin’s Agile Software Development: Principles, Patterns, and Practices [5]. One of the mandates of XP is to do the simplest thing possible. Software is developed over short (one- to four-week) iterations in which only the most important function that can be done for a specific time frame is implemented. When done properly (it often isn’t), this iterative development is very fast, robust, and flexible. Other agile methods also take this approach.
may say, “Hey, I’ve got only one algorithm now, but I’m bound to get others later, so I’ll set up an interface now.” Such individuals believe that putting in effort now to assist where variations can emerge later will save time down the road. The problem with this, however, is that change can happen anywhere later. Following this approach everywhere can both overcomplicate your design and waste time because you often implement things that are never needed.

The Strategy pattern, however, doesn’t suggest this. The pattern describes forces. It is up to you to decide if implementing the pattern is called for.6 In other words, if you force-fit a pattern into a situation where the pattern doesn’t apply, you are, by definition, not following the pattern.

Using the Forces Without the Implementation

On the other hand, the Strategy pattern can be very useful to an XPer even when he or she writes no code that actually implements the pattern. Consider the previous situation if you were following XP practices.7 If, in your current iteration, you had only one algorithm to implement, you’d likely have your Client just call it directly. However, you might also recognize this as a likely place to use a Strategy pattern later if there are other algorithms present in future stories. This knowledge would suggest that you take steps now to make it easier to implement the pattern if it is needed later.

Admittedly, XP would say the same thing — its coding practices are geared toward easy code changes. However, the potential need for an implementation of the Strategy pattern underscores this idea and makes you more likely to follow the XP practices you should. Paying attention to this doesn’t take any extra work or code; it just reinforces standard XP practices.

Let’s consider two XP coding practices: “coding by intention” and “once and only once.” Coding by intention says that if, while writing one method, you find you need to implement a function, pretend a method that accomplishes that function already exists. Give it an intention-revealing name, write down an appropriate parameter list, and actually implement it later. “Once and only once” means just that — code things once and only once. In other words, avoid duplication.

Coding by intention will make it easier to implement the Strategy pattern because the object that uses the algorithm (i.e., the object that will become the Context object if the Strategy pattern is ever implemented) will likely end up having a method called something like “runAlgorithm().” Later, if you need to implement the Strategy pattern, you will simply change this to the reference and call required. The once-and-only-once rule will ensure that if there are several steps in calling this algorithm, you will have only one place containing these steps (otherwise you’d have duplication). Thus, knowing that new rules may become available merely reinforces things you should be doing in XP anyway. In other words, knowledge of the Strategy pattern reinforces the good coding rules that XP mandates. It wouldn’t require you to implement the pattern before it is called for.

Misapplying the Strategy Pattern

The whole idea of incremental development, which is at the heart of XP and all agile processes, implies that we should reject the idea that code must decay in favor of the notion that code can evolve. To do this, however, we must hold ourselves to a rigorous standard. Each time we change the code, we will make it at least a little bit better — and never any worse. This will lead us to use the Strategy pattern.

6To suggest a variant of a well-known theme, I will argue that hammers aren’t bad. Used properly, hammers can be very useful tools. Used improperly, they can be very destructive.

7I do not consider myself an XPer for reasons beyond the scope of this article. However, it does have many brilliant practices and values that I follow vigorously.
pattern when it improves the code in some way; for instance, by making the Context object simpler.

Ignoring the forces of the pattern, however, can lead to its improper use regardless of design philosophy. For example, if the algorithms require input data that is widely different for each algorithm, it may take as much (or more) work to resolve them all to a common interface. Handling varying returned data could be just as problematic. This would be akin to forcing a square peg into a round hole and would be counterproductive. Unfortunately, some people will say, “The pattern tells me to do this.” No, it doesn’t. The pattern tells you about forces and a potential implementation. Knowing patterns does not mean you no longer have to think. It just helps you decide what to think about.

To put it another way, the Strategy pattern is about externalizing a variation in order to encapsulate it. Doing so protects the Context object from the variation, but at a cost. If it’s harder to externalize the variation than it is to maintain it in the Context object, then the pattern does not apply. Of course, the pattern tells you this in the first place.

The reason patterns get misused is that people generally misunderstand what they are. They are not “reusable solutions,” even though Design Patterns discusses them as if they are [4]. The Gang of Four wanted to demonstrate that reusable designs (not necessarily reusable implementations) do indeed exist. They did not mean to imply that these designs should be force-fit into new problems. Their way of looking at object-oriented design overall (the “advice from the Gang of Four”) is manifested by the patterns — that is, the patterns are examples of the benefits of this new paradigm. The reason the “patterns aren’t important” is that we’d achieve the same results without them if we followed the principles they proceed from; we’d just take a little longer to get there.

---

**ANOTHER EXAMPLE OF PATTERNS AS FORCES: THE DECORATOR PATTERN**

The Decorator pattern comes into play when there are a variety of optional functions that can precede or follow another function that is always executed. For example, prior to sending a transmission (which is always done), it may be that you want to encrypt, compress, translate data, check, or do any number of other things in any order. How would you best do this? The Decorator pattern’s implementation tells you to make a linked list of the optional functions ending with the Transmission object. The “decorating” objects should all have the same interface as the Transmission object.

This implementation can actually be a very bad design. For example, let’s say that these “decorators” (i.e., the optional functionality) are created by different development groups. Let’s further state that certain exceptions may be thrown by the system and need to be handled by each of the decorators. In a perfect world, you could have confidence that each of these groups will do what they are supposed to do — catch these exceptions properly. However, what if they do not? What if an exception is thrown, and the group writing the code doesn’t catch it? The entire system may crash at this point.

Another solution is to have the Client object catch the exception. However, this loses some of the value of the Decorator pattern in that your Client object now needs to do more than it did before.

A more robust solution is to implement a collection object that has the same interface as the Transmission object. This calls the “decorators” and catches any required exceptions in case they don’t. In fact, this might have the added advantage of making it so the decorating objects no longer need to have the same interface.

The point is, you can view the Decorator pattern as having the following forces:

1. Several optional functions exist.
2. These decorators may or may not be following all of the rules they should.
3. You need some way to invoke these decorating objects, in different orders as needed, without encumbering the Client object.
4. You don’t want to encumber your application with knowing which of these to use (or even that they exist).

Thinking of the Decorator in this light makes a distinction between the purpose of the Decorator pattern and the implementation of the Decorator pattern.
The value of “named” patterns is more about the name than the pattern. Other professional domains (carpentry, masonry, metallurgy) have specific language that comes from hundreds or thousands of years of tradition. Software development has been around for 40 years at best. Named patterns give us a common vocabulary that helps us to move our activities to the same level of professionalism that these other domains enjoy.

**SUMMARY**

We have seen that while a pattern has an implementation, it is better not to think of the pattern as being its implementation. If we do think of a pattern as mandating solutions in particular situations, this is not unlike having a hammer and looking for nails. Patterns are about helping us resolve forces in our problem domain, about seeing these forces. Using patterns still requires us to think — in fact, maybe more so. However, knowing patterns helps us know what we should be thinking about. Patterns used improperly can cause damage, but then, in essence, we aren’t using the pattern but rather an incomplete knowledge of patterns. Patterns used properly can only help us, as they encourage us to pay attention to the important issues surrounding the problems we are trying to solve.

**REFERENCES**

1. Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press, 1979. (*Both a brilliant technical book as well as a book that is fun to read. I highly recommend reading this if you have any interest in architecture, philosophy, or software.*)


4. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (*Still the best reference on design patterns, even if it’s starting to become somewhat dated.*)


6. Meyer, Bertrand. *Object-Oriented Software Construction*, 2nd edition. Prentice Hall, 2000. (*Although the size of this book is a bit intimidating, it is a brilliant book. Read it a little at a time and reflect on what he has to say.*)

7. Shalloway, Alan, and James R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, 2001. (*At the risk of sounding self-serving, this is the best text on introducing a developer to what patterns are. Read before reading the Design Patterns book by the Gang of Four.*)

Alan Shalloway is the founder, CEO, and senior consultant of Net Objectives. His vision is effective software development without suffering. Mr. Shalloway is the primary author of Design Patterns Explained: A New Perspective on Object-Oriented Design and is currently coauthoring Emergent Design: Refactoring and Design Patterns for Agile Development with Scott Bain. He has a master’s degree in computer science from MIT.

Mr. Shalloway can be reached at Net Objectives, 25952 SE 37th Way, Issaquah, WA 98029, USA. Tel: +1 425 313 3065; Fax: +1 425 657 0027; E-mail: alshall@netobjectives.com.
Ensure That Your Enterprise Architecture Initiatives Are Successful

The demands on corporate IT have never been greater. Cutting costs and accelerating time to market for individual line-of-business projects are still priorities, but even that’s not nearly enough anymore. Companies are now looking for strategies to better leverage their whole IT infrastructure. They want IT to deliver sophisticated enterprise applications that can provide value across many lines of business and provide marked differentiation from their competitors.

The key to achieving a truly integrated enterprise is to build an architecture that transcends different technologies and provides application integration, consolidated views into the enterprise and common business processes. Cutter Consortium’s inhouse Enterprise Architecture Workshops are developed and delivered by its expert Senior Consultants and are designed to provide you with the strategic insight and technical foundation to build — and build upon — an architecture that delivers maximum business flexibility and growth. Bring Cutter Consortium’s Senior Consultants into your organization today, and benefit from the experience of the trailblazers in enterprise architecture.

Enterprise Architecture Workshops Available from Cutter Consortium:

- **Integrated Enterprise Architecture**
  Discover what concrete steps you need to take to become an integrated enterprise.

- **Enterprise Architecture for Senior Managers**
  Understand a common conceptual framework and advanced techniques for enterprise and system architecture.

- **Service-Oriented Architecture Basics**
  Learn how to start introducing service-oriented architecture (SOA) principles into your applications to enable your organization to take advantage of the benefits of new technologies like Web services.

- **Achieving a Service-Oriented Architecture with Web Services**
  Discover the concepts of an SOA, how Web services can help achieve one and the benefits of such an architecture.

- **Web Services and Service-Oriented Architectures**
  Get a real-world perspective of how Web services can be used and the issues — and possible solutions — that one can expect from the adoption of Web services for distributed system development.

- **Model Driven Architecture**
  Understand the basics of Model Driven Architecture (MDA) and how to start realizing its benefits in your organization.

- **Applying Model Driven Architecture to Web Services**
  Learn how to apply MDA techniques to Web services applications in order to maximize their cross-platform integration capabilities.

- **Architecting Web Services Applications**
  Combine new and existing J2EE GUI-based applications with new Web services applications.

- **Integrating J2EE and .NET Applications with Web Services**
  Discover a strategy for integrating J2EE and .NET applications with Web services.

For More Information:

For more information on Cutter Consortium’s Training Workshops, call Dennis Crowley at +1 781 641 5125, send an e-mail to dcrowley@cutter.com or visit www.cutter.com/workshops.
Cutter IT Journal

Topic Index

September 2003  Patterns in Software Development
August 2003  The New CIO Agenda
July 2003  EA Governance: From Platiitudes to Progress
June 2003  IT Metrics and Benchmarking
May 2003  Is Open Source Ready for Prime Time?
April 2003  Project Portfolio Management: Blueprint for Efficiency or Formula for Boondoggle?
March 2003  Critical Chain Project Management: Coming to a Radar Screen Near You!
February 2003  XP and Culture Change: Part II
January 2003  Ending “Garbage In, Garbage Out”: IT’s Role in Improving Data Quality
December 2002  Preventing IT Burnout
November 2002  Globalization: Boon or Bane?
October 2002  Whither Wireless?
September 2002  XP and Culture Change

Upcoming Issue Themes

Usability
Metrics and Benchmarking II
Killing IT Projects

Joshua Kerievsky, Guest Editor

Joshua Kerievsky, a Senior Consultant with Cutter Consortium’s Agile Project Management Practice, began his career as a professional programmer at a Wall Street bank, where he programmed numerous financial systems for credit, market, and global risk departments. After a decade at the bank, he founded Industrial Logic to help companies practice successful software development. Mr. Kerievsky has programmed and coached on small, large, and distributed XP projects since XP’s emergence. He recently pioneered Industrial XP, an application of XP tailored for large organizations. Mr. Kerievsky has been an active member of the patterns community and is presently completing a book entitled Refactoring to Patterns. He can be reached at jkerievsky@cutter.com.
BUSINESS-DRIVEN SOFTWARE DEVELOPMENT

Business-Driven Software Development is Net Objectives’ proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. This approach has a consistent track record of delivering higher quality products faster and with lower cost than other methods.

Business-Driven Software Development goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

Our approach integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management’s role altogether. These are critical aspects of all successful organizations. Here are some key elements:

- Business provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment.
- Teams self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed.
- Management bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality.

BECOME A LEAN-AGILE ENTERPRISE

Involve all levels. All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

WHY NET OBJECTIVES

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place, such as focusing on the team when that is not the main problem, or using the wrong method, such as using Scrum or kanban because they are popular.

Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban) and integrates business, management and teams. This lets us help you select the right method for you.
LEARN TO DRIVE DEVELOPMENT FROM THE DELIVERY OF BUSINESS VALUE
What really matters to any organization? The delivery of value to customers. Most development organizations, both large and small, are not organized to optimize the delivery of value. By focusing the system within which your people are working and by aligning your people by giving them clear visibility into the value they are creating, any development organization can deliver far more value, lower friction, and do it with fewer acts of self-destructive heroism on the part of the teams.

THE NET OBJECTIVES TRANSFORMATION MODEL
Our approach is to start where you are and then set out a roadmap to get you to where you want to be, with concrete actionable steps to make immediate progress at a rate your people and organization can absorb. We do this by guiding executive leadership, middle management, and the teams at the working surface. The coordination of all three is required to make change that will stick.

OUR EXPERTS
Net Objectives’ consultants are actually a team. Some are well known thought leaders. Most of them are authors. All of them are contributors to our approach.

SELECTED COURSES
Executive Leadership and Management
- Lean-Agile Executive Briefing
- Preparing Leadership for a Lean-Agile/SAFe Transformation

Product Manager & Product Owner
- Lean-Agile Product Roadmaps
- PM/PO Essentials

Lean-Agile at the Team
- Acceptance Test-Driven Development
- Implementing Team Agility
- Team Agility Coaching Certification
- Lean-Agile Story Writing with Tests

Technical Agility
- Advanced Software Design
- Design Patterns Lab
- Effective Object-Oriented Analysis and Design
- Emergent Design
- Sustainable Test-Driven Development

DevOps
- DevOps for Leaders and Managers
- DevOps Roadmap Overview

SAFe®-Related
- Implementing SAFe with SPC4 Certification
- Leading SAFe® 4.0
- Using ATDD/BDD in the Agile Release Train (workshop)
- Architecting in a SAFe Environment
- Implement the Built-in Quality of SAFe
- Taking Agile at Scale to the Next Level

OUR BOOKS AND RESOURCES

CONTACT US
info@netobjectives.com
1.888.LEAN-244 (1.888.532.6244)

LEARN MORE
www.NetObjectives.com
portal.NetObjectives.com