

Chapter 11

Refactor to the Open-Closed

Solving tricky problems can often involve changing your point of view. In this chapter we'll examine one particularly tricky problem – how to avoid overdesign without suffering the problems that often accompany an insufficient or naïve design. In the process, we'll rethink two hopefully well-known aspects of development: the Open-Closed Principle, and the discipline of Refactoring. We'll begin by examining these aspects as they are traditionally understood, and then repurpose them in a more agile way.

The Open-Closed Principle

The notion that systems have to accommodate change is not a new one. Long before objects and OO, Ivar Jacobsen said “All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version.” Of course, since this was before the invention of Object-Oriented languages and systems, he was focused on the particular nature of procedural code, and how one can make code more or less “changeable” depending on what one focused upon when writing it.

Jacobsen was one of many who promoted the idea of breaking up programming functionality into multiple, “helper” functions, called from a central location in the code, with the idea that change would be fundamentally easier to deal with if code was not written in large “blobs”. Without this, code would be hard to understand, hard to control in terms of side-effects, and difficult to debug. Hard to change, in other words.

We still believe in this notion, and in fact it has become known today as the practice of “Programming by Intention” (see our chapter on this subject for more details). Personally, we never write code any other way.

With the advent of OO the same notion, namely that we should expect our systems to change, took on a potentially different meaning. Bertrand Meyer, an early OO thinker and the creator of one of the most object-oriented language of the time, Eiffel, rephrased Jacobsen this way:

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

Initially this was seen as a natural outgrowth of inheritance. If we have an existing class, let's call it “ClassA” and we want to change some aspect of its behavior, it was pointed out that rather than making code changes to ClassA, we could instead create a new class based on it, through inheritance, and make the change(s) in the new class.

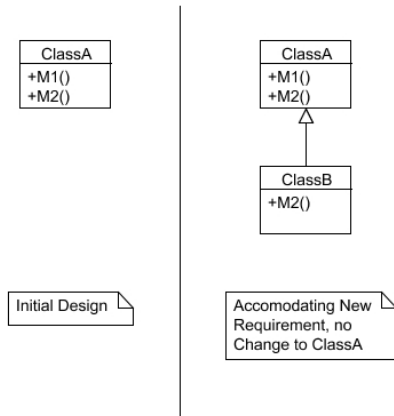


Figure 11.1
Open-Closed through direct inheritance

ClassA, here, is an existing class with existing behavior. Now, we need something similar, but with a variation in method M(), and so we’ve used inheritance to make a new class (ClassB) based on the existing one.

This was thought to be “re-using” the object, and seemed to be an admirable aspect of OO. Leave ClassA alone, and you’re not very likely to break it. A new class, ClassB, will only contain what is different and new (in this case, by over-riding the M2() method), and will therefore also be simpler and safer to work with. This is most likely what caused the creators of Java to use the word “extends” to indicate inheritance, as Java was created in the early-mid 90’s when just this sort of thinking was prevalent.

There were problems with this once developers started to see “inheritance for reuse” as the solution to essentially everything (see our chapter “When and How to use Inheritance” for more details on this). Patterns, and the general design advice they contain, tended to make us reconsider what it meant to be “closed for modification”, and to use class polymorphism to achieve it as shown in figure 4.1.2.

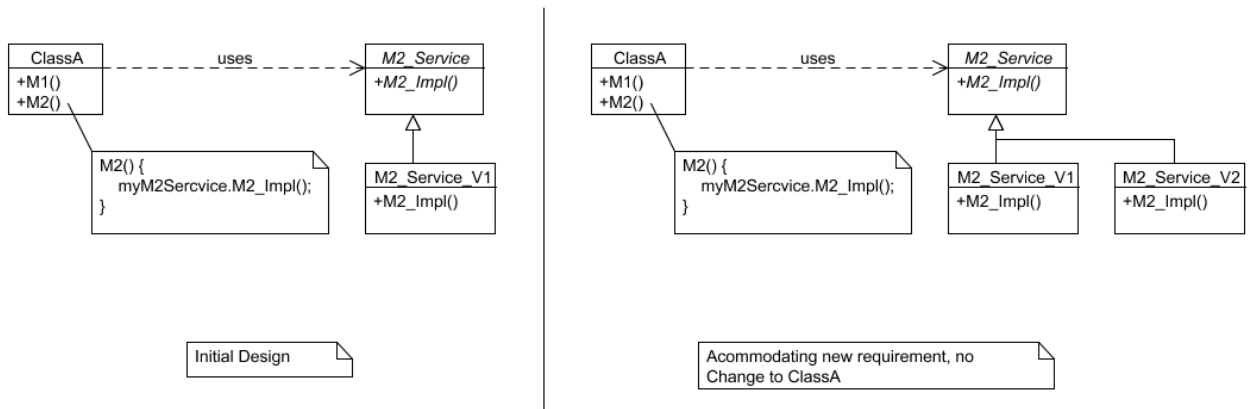


Figure 11.2
Open-Closed through class polymorphism

Now we can accommodate a new version of the `M2_Service` by adding another class, `M2_Service_V2`, and make no changes to the existing code in `ClassA`, or in `M2_Service_V1`. This means we are “open to extension” (adding a new class, in this case) and “closed to modification” (all the things we are not changing).

But even in the initial design, we have added a class (the abstract class `M2_Service`), and this is without considering how the initial “`M2_Service_V1`” class will be instantiated.

If `ClassA` contains the code “`new M2_Service_V1()`” within it, then we would not be able to add `M2_Service_V2` without changing that bit of code, and thus we would not be “closed to modification” of `ClassA`, which was our goal. In other words, the `M2_Service` abstract type does not really buy us much if client classes build concrete instances themselves. See our chapter “Separate Use from Construction” for a more detailed treatment of this topic.

To fix that problem we’d have to add yet another entity, one which was solely responsible for creating the instance in the first place, and we would thereby know that was the one and only place where a code change would have to be made. `ClassA`, and all other clients of this service (reuse being another primary goal here) would not change when the V2 version came along.

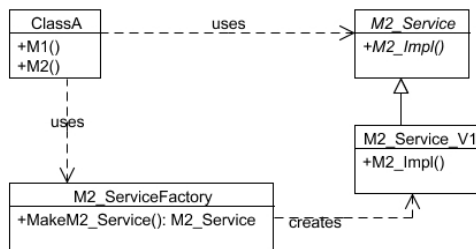


Figure 11.3

Adding an Instantiation Entity (“Factory”) to Build the Right Instance

This would seem to be adding complexity for something that may or may not happen. In the past, we would often say to add this sort of anticipatory infrastructure when the issue was “likely to change,” but there are few developers indeed who would claim to be able to make that estimation with any degree of confidence. Anticipatory guessing is not very reliable and not repeatable over time. We need something more than luck.

Open-Closed to Other Things

Furthermore, people often focus on the open-closed principle in terms of adding a new behavior to an existing system. The truth is that the principle can apply to any change at all.

- An object may start out using a single instance of a service, object and then later require more than one, for load balancing, or where there are different versions of a service and more than one version is needed. One->one vs. one->many is a potential change.
- An object may use a set of service objects in a given order today, and then alter the order tomorrow. The sequence of a workflow is a potential change.
- An object may be responsible for cleaning up the memory of the service object it holds, then later it may no longer be. Memory management is a potential change.
- An object may use the same service object it was initially given throughout its lifecycle, then later must have the ability to change the service object later. Static vs. dynamic relationships is a potential change.

...and so on. We should consider the open-closed principle as potentially applying to all of these circumstances. When any of these potential changes occurs, we'd like to be able to accommodate it by adding something new, and leaving as much of the existing system as-is.

But as before, the question is how can we know when any of these changes are likely enough that it is worth putting in the extra abstractions, factories, and so forth up front?

Open-Closed is a “Principle”

Principles can be thought of either being rules of nature (or programming in our case) or guidance to follow to best take advantage of these rules. Principles are always true, but how to use them depends upon the context in which you find yourself. The principle does not specify a specific tactic for achieving this, because there are many that could work, depending on circumstance.

Remember that the Open-Closed Principle comes from Jacobsen's notion that all systems will have to be changed, and that this was before the creation of objects and object-oriented languages and tools. So, what he was referring to was the basic way code is structured, using the finer-grained functions mentioned earlier.

For a concrete example of what Jacobsen was recommending, we can turn to something fun.

Several years ago, when IBM was trying to encourage some of their older, quite experienced developers to switch to an object-oriented point of view (in this case, by learning Java), they ran into some resistance. These guys felt that they could do what they needed to do without OO, and really didn't want to have to abandon their tried-and-true techniques in procedural code. This is understandable; good procedural code required a lot of discipline and experience, and so these guys had invested a lot of themselves into it.

So, IBM tapped into the generally competitive nature of most software developers. They came up with a game, called “Robocode”¹, which allowed developers to write Java classes that would actually fight each other, with explosions and all, in a graphical framework. The devs write the classes, the game lets them fight each other.

Here's an example of a *very* simple Robocode robot class:

```
import robocode.*;
public class Robbie extends Robot {
    public void run() {
        ahead(100);
        turnGunRight(360);
        back(100);
        turnGunRight(360);
    }

    public void onScannedRobot(RobotEvent re) {
        if(isNotMyTeam(re)) fire(1);
    }
}
```

¹ Robocode is more complex than we are intimating here, but this makes our point. For more, visit their site! It's wicked fun: <http://robocode.sourceforge.net/>

By extending Robot, Robbie can be up-cast to that type, and the game will be able to call methods like “run()” when it’s Robbie’s turn to move and “on ScannedRobot()” when Robbie sees an enemy, etc...

You might have noticed that neither of these methods really does much in and of itself. “run()” called other methods like “ahead()”, “turnGunRight()” and so forth. “onScannedRobot()” delegates to other methods as well. Jacobsen and others of the time called these subordinate methods “helper methods” and they suggested that this was a good practice to follow.

It’s good because adding more helper methods, removing existing ones, or change the implementation of a helper method, etc... can have very limited impact on the rest of the system. Contrast this to how such changes would play out if run() had all the code in it, in a large, complicated structure. Such structures will work, but they are very hard to change later on.

Would this design be more Open-Closed if these behaviors, like “fire()” were in separate objects? Sure. But how far do you go?

One extreme would be to pull everything out into its own object. This would create vast numbers of small objects, creating a lot of complexity and coupling. The other extreme is to code all behavior into a single method in a single object. Neither seems very attractive, so how do we find the best middle ground? One adds flexibility at the expense of complexity, the other loses code quality. How can we add flexibility when we need it? That’s what Refactoring to the Open-Closed will allow us to do.

Refactoring

It’s not uncommon for developers to make changes to code that cause no difference in the behavior of a system. Sometimes, for instance, we may change the name of a variable, or clean up the indentation in nest logic, or give a method a more intention-revealing name, or change a code structure to make it easier to read (a “for” loop replaced with “foreach” loop, etc...), and so on.

Sometimes these changes can actually alter a design or architecture, but again, they may not change the outward-facing functionality of the system from the stakeholder’s point of view, or in the way it interacts with or affects other systems.

When we make such a change, we are refactoring. We make this distinct from *enhancing* a system, *debugging* it, or taking any other action that would change its resulting behavior (including, of course *breaking* it).

In his ground-breaking book Refactoring², Martin Fowler defined a specific set of disciplines around what had up to that point been individual, ad-hoc practices. He defined his “refactoring moves” in a way quite similar to design patterns; take things we do repeatedly, give them specific names, and capture what we as a community know about them, so this knowledge becomes shared. We often think of refactoring moves as “patterns of change”. They establish a shared language that communicates our plans and expectations with a higher degree of fidelity.

“Extract Method”, for example, would be the refactoring move that we’d use to convert a “single method blob” of code into the “helper method” approach used in our Robot above. If we realized that one of those helper methods actually should be in its own class, we could do “Extract Class” if the class was a new one or “Move Method if we were moving it to an existing class. In each case, a clearly-defined series of steps is defined, making sure that we don’t miss

² TODO citation

anything critical, and enabling us to improve the code aggressively, and with high confidence. It's an enormously useful book.

That said, refactoring has gotten something of a bad reputation among those who pay for software to be developed.

If a development team is “refactoring the system” business owners know that they can expect no new functionality or improved performance from the system while this is going on. In fact, every refactoring move includes running a suite of tests to *ensure* that nothing has changed in the way the system behaves. The same tests must pass after the refactoring that passed before it.

This reputation is understandable. Refactoring improves the design of a system, in terms of its source code. Only the developers encounter the source code, so obviously refactoring is a developer-centric activity, and only delivers value to developers... or so it would seem.

Also, there is no end to refactoring, per se. No system design is ever perfect, and therefore any system could be refactored to improve its design. Making these improvements can be a very rewarding, satisfying, and intellectually-stimulating thing for a developer to do, and so it is possible to fall into the trap of obsessively refactoring a system.

Why Refactor?

As part of the discipline, Fowler included a set of “code smells” to help us to see where refactoring improvements are called for.

If we find that we're making the same change in many different places, he calls this “shotgun surgery”. If we find that a method in Class A is referring to the state or functionality of Class B to an excessive degree, he calls this “feature envy.” A class which has no functionality but does contain state is called “lazy”, etc... These smells do not necessarily indicate a problem, but rather a potential problem that should be investigated and refactored if appropriate.

These smells, in other words, help us to target parts of the system that could be refactored and, arguably, improved, but they do not tell us if the system *should be refactored now*. When is it worth it, and when are we, in fact, falling into the trap that has given refactoring a bad reputation among those who pay for software to be developed?

Debt vs. Investment

Let's make a distinction about this decision. Do we fix something in the design of the code, even though the code is working, or not?

If we see something we know is “not good code” and we decide not to fix it, we know two things:

1. If we do not fix it, we save time and can move on to adding new functionality to the system. However, this new functionality will be achieved at the cost of leaving the poor code in place.
2. If we fix the code at a later point in time, it will likely cost more. It will probably have gotten worse, more things will be coupled to it the way it is now, and it will be less fresh in our mind (requiring more time to re-investigate the issue).

Doing something which avoids payment now at the cost of a higher payment later is essentially “debt”. It's like buying something on your credit card; it is not that you don't *ever* have to pay for it; you just don't have to pay for it now. Later, when you do, there will be interest added and it will cost you more.

If, on the other hand, we fix the bad code without adding any outward business value to the system, we must acknowledge that this is not “free”; it costs money (our customer’s money). However, we also know:

1. Cleaner code is easier to change. Therefore, changes we are asked to make later will be easier to do, take less time, and therefore save money.
2. This can pay off over and over again, as one change leads to another, or as the business continually requires new functionality from the system.

In other words, this constitutes an investment. Pay now, and then get paid back over and over again in the future. It’s a little like the notion of the “money tree”. When you plant the tree, initially you don’t get anything for your effort. Once the tree grows into blossom, however, it pays off over and over again.

Businesses understand these concepts very well. They will accept debt consciously, but they know that holding a valuable investment is preferable. Sometimes it helps to speak the language of the person you’re trying to influence.

Refactoring and Legacy Systems

Most people think of refactoring in terms of “cleaning up” legacy code. Such code was often written with a focus on making the system as small and fast as possible. Computers in the earlier days often made this a necessity; sure, the code is incomprehensible and impossible to maintain when we come back to it later, but it was easier for the slower computers to deal with, and computers were the critical resource in those days, not developers.

Legacy systems that have remained in place, however, have probably proved their value by the very fact that they are still here. Fowler’s initial purpose was to take this code that has value in once sense (what it *does*), and to reshape it into code that is valuable in the more modern sense (we can work with it efficiently), without removing any of that exiting value... that is, without harming it.

Refactoring to the Open-Closed

So, refactoring is often thought of in terms of bad code, untested code, code that has decayed over time and that we have to suffer with now.

This is true as far as it goes. However, we also know that it’s very difficult to predict what changes will come along, and that change can appear almost anywhere in our process:

- Requirements can change
- Technology can change
- The marketplace can change
- Our organization/team can change
- We change (hopefully, we get smarter)
- etc...

So, another role for refactoring skills to play is when we have code that *was fine yesterday* but it not easy to modify in the light of a particular changing circumstance. In other words, we have something new that we need to add now to the code, but we cannot add it in an open-closed fashion as it stands.

Let’s use our Robocode Robot as a concrete example. The design so far is a simple, single class that either extends a type called “Robot”.

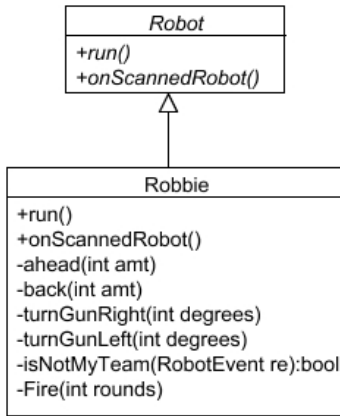


Figure 11.4
Robbie the Robot

This is not “Open-Closed” in terms of adding new classes to change behaviors. If we come up with more than one way of “firing”, for instance (maybe we’ll have more than one kind of gun in the future), we’ll have to change the code in the “fire()” method, perhaps adding a switch or other logic to accommodate the variation in behavior. Similarly, if we get new ways of turning, or moving, or determining whether another Robot in on “my team”, we’ll have to make code changes and we’ll add complexity when we do.

To make it “Open-Closed” in the senses that we usually mean today, we’d introduce polymorphism for all of these behaviors. The design would probably look something like this:

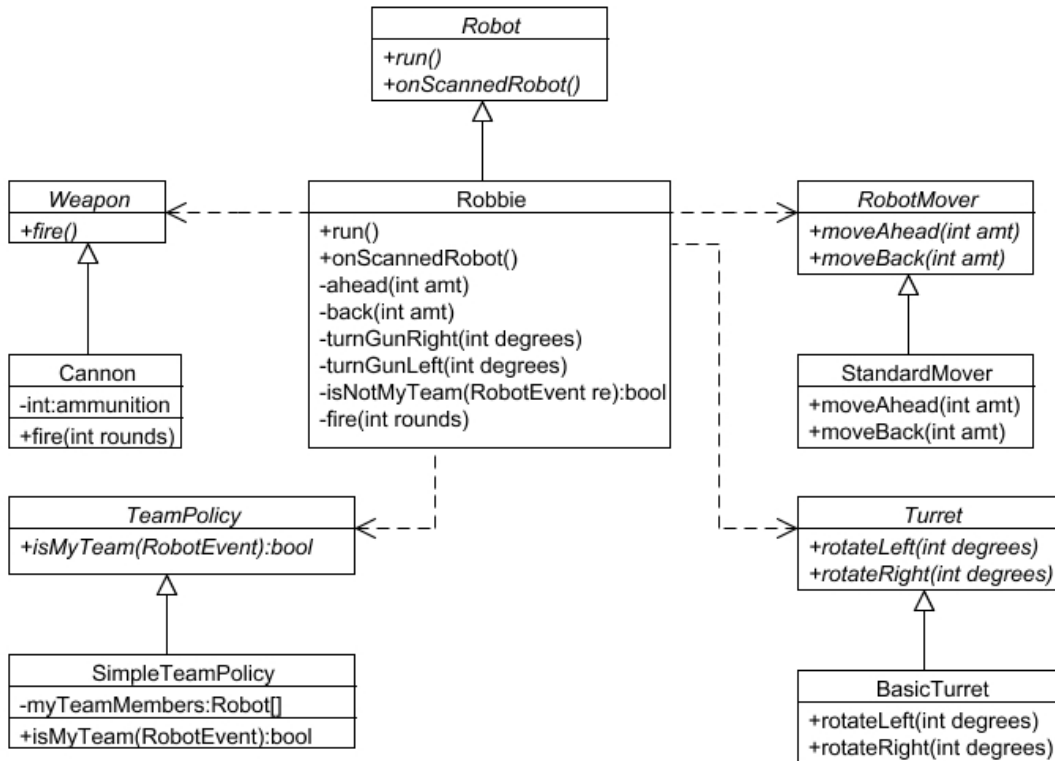


Figure 11.5*Everything is Open-Closed*

This would make it possible for me to add a different kind of weapons, turret, movement mechanism, or team structure without have to alter the code in Robot. But, given that none of these things is varying right now this is far more complex than I need, especially when you consider that I have not added those factories yet!

If you feel, given that none of these behaviors are varying at the moment, that this approach is over-doing it, I'd agree. If one pulls out all possible variations, as a rote practice, one will tend to produce designs that are overly complex.

Some would say that we should pull out issues that are not varying today, but are *likely to vary* in the future. The problem with this predictive approach is that we're likely to be wrong too often, and when we are we will have pulled something out that never varies (overdesign), and fail to pull thing out that need to be variable (failing to be open-closed).

In studying refactoring, we are learning how to make changes in a disciplined way. Each refactoring, though it may have been originally intended as a way to clean up bad code, can also be used to change code *just enough* to allow it to be changed in an open-closed way, once we know this change is necessary.

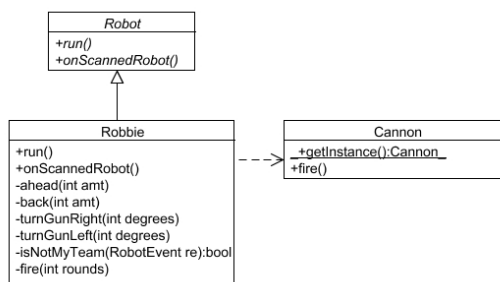
We call this “refactoring to the open-closed”.

Just-In-Time Design

Refactoring to the open-closed allows us to introduce design elements as they are needed, but not before. It allows design to emerge in a Just-In-Time way, which means that we can proceed based on what we know, when we know it, rather than through prediction.

Look back at the version of Robbie in figure 4.1.4. Let's say that we designed and coded it that way, then later a new requirement emerged... we need to be able to accommodate a new kind of “fire” mechanism (a new weapon, or a new way of firing in general). We could put a switch into the fire() method of Robbie, but this is not an open-closed change.

In the light of this new requirement (which is not a guess, but is actually being requested), we can do this in stages, using our refactoring skills. First, we do “Extract Class”³:

**Figure 11.6***Extract Class*

(You'll note that we've used a static getInstance() method to create the instance of Cannon. See “Separate Use from Construction” for more on this technique.)

³ “Refactoring: Improving the Design of Existing Code”, Martin Fowler, pp 149

The fire() method in Robbie now contains a call to the fire() method in Cannon. This is still not open-closed, but we’re getting closer. Next, we do Extract Interface:⁴

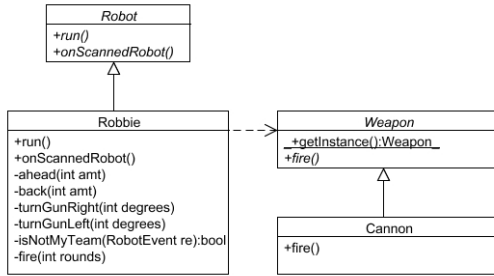


Figure 11.7
Extract Interface

We have not changed the behavior of the Robot, and if we had tests running they would still all pass in the same way. This is one nice thing about automated tests; they confirm that we are, in fact, refactoring, not enhancing or introducing bugs.

This is now open-closed to the new weapon, and we are done refactoring. Now we can enhance the system with the new requirement in an open-closed way:

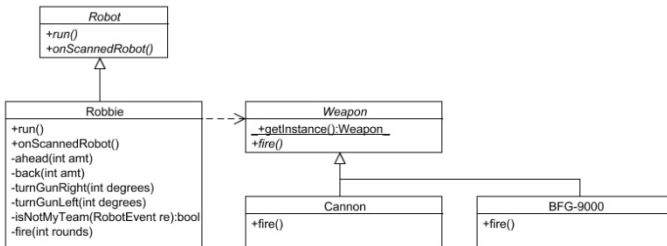


Figure 11.8
Extract Interface

Summary

Basing your decisions on prediction is setting yourself up to fail. Trying to design for every possible future change will lead to over-design, which is also setting yourself up to fail.

The refactoring discipline, used to enable this just-in-time response to changes allows you to make your decision based on what actually happens, not what you predict will happen, and also allows you to introduce design elements as you need them, avoiding over-design.

So what’s missing? Tests. Refactoring requires automated testing, because it is the tests that tell you whether you are, in fact, refactoring. See our chapter on “Define Tests Up Front” for more on testing.

⁴ “Refactoring: Improving the Design of Existing Code”, Martin Fowler, pp 341

Business-Driven Software Development (BDS) 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. BDS has built a reputation and track record of delivering higher quality products faster and with lower cost than other methods

BDS 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.

BDS 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. In BDS:

- **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

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 BDS 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 (e.g., the team when that is not the main problem) or using the wrong method (e.g., Scrum, just because it is popular). Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban, Scrumban) and integrates business, management and teams. This lets us help you select the right method for you.

<p>Assessments</p> <p>See where you are, where you want to go, and how to get there.</p> <p>Business and Management Training</p> <p>Lean Software Development Product Portfolio Management Enterprise Release Planning</p>	<p>Productive Lean-Agile Team Training</p> <p>Team training in Kanban, Scrum Technical Training in ATDD, TDD, Design Patterns</p> <p>Roles Training</p> <p>Lean-Agile Project Manager Product Owner</p>
--	---

