Agile Technologies

Module-5

Deliver Value: Exploit Your Agility, Only Releasable Code Has Value, Deliver Business Results, Deliver Frequently, Seek Technical Excellence: Software Doesn't Exist, Design Is for Understanding, Design Trade-offs, Quality with a Name, Great Design, Universal Design Principles, Principles in Practice, Pursue Mastery

Deliver Value

Your software only begins to have real value when it reaches users. Only at that point do you start to generate trust, to get the most important kinds of feedback, and to demonstrate a useful return on investment. That's why successful agile projects deliver value early, often, and repeatedly.

Exploit Your Agility

Simplicity of code and process are aesthetically pleasing. Yet there's a more important reason why agility helps you create great software: it improves your ability to recognize and take advantage of new opportunities.

You may discover a brilliant new technique that simplifies your code, or your customer may develop a new business practice that saves time and money.

Want to deliver real value? Delivering value to your customer is your most important job. Aggressively pursuing feedback from your customer, from real users, from other team members, and from your code itself as early and as often as possible allows you to continue to learn and improve your understanding of the project. It also reveals new opportunities as they appear.

Agility requires you to work in small steps, not giant leaps. A small initial investment of time and resources, properly applied, begins producing quantifiable value immediately.

As well, committing to small amounts of change makes change itself more possible.

Aggressively seeking feedback and working in small steps allows you to defer your investment of resources until the last responsible moment.

In Practice

XP exploits agility by removing the time between taking an action and observing its results, which improves your ability to learn from this feedback. This is especially apparent when the whole team sits together. Developing features closely with the on-site customer allows you to identify potential misunderstandings and provides nearly instant responses to questions.

Including real customers in the process with frequent deliveries of the actual software demonstrates its current value to them.

The short work unit of iterations and frequent demos and releases create a reliable rhythm to make measured process adjustments.

Beyond Practices

I worked for a small startup whose major product was an inventory management system targeted at a specific retail industry. We had the top retailers and manufacturers lined up to buy our project. We were months away from delivery when our biggest customer ran into a problem: the license for their existing point-of-sale system suddenly expired.

The software included a call-home system that checked with the vendor before starting. Our plans included developing our own POS system, but that was at least several months away.

Our current system managed inventory, but we hadn't done specific research on supporting various terminal types, credit card scanners, and receipt printers.

This was our largest customer, and without its business, we'd never succeed.

After discussing our options and the project's requirements with the customer, we decided that we could build just enough of the POS system within six weeks that they could switch to our software and avoid a $30,000 payment to the other vendor.

We shifted two-thirds of our development team to the POS project. Though we started from an open source project we'd found earlier, we ended up customizing it heavily.

After two weeks of development, we delivered the first iteration on a new machine we had built for the customer to test. We delivered weekly after that. Though it was hard work, we gradually added new features based on the way our customer did business.

Finally, the only remaining task was to change a few lines in the GUI configuration to match the customer's store colors. We saved our customer plenty of money, and we saved our account.

Only Releasable Code Has Value

Having the best, most beautiful code in the world matters very little unless it does what the customer wants. It's also true that having code that meets customer needs perfectly has little value unless the customer can actually use it. Until your software reaches the people who need it, it has only potential value.

Delivering actual value means delivering real software. Unreleasable code has no value.

Working software is the primary measure of your progress. At every point, it should be possible to stop the project and have actual value proportional to your investment in producing the software.

Agility and flexibility are wonderful things, especially when combined with iterative

incremental development. Throughput is important! Besides reducing thrashing and waste, it provides much better feedback, and not just in terms of the code's quality. Only code that you can actually release to customers can provide real feedback on how well you're providing value to your customers.

In Practice

The most important practice is that of "done done," where work is either complete or incomplete. This unambiguous measure of progress immediately lets you know where you stand.

Test-driven development produces a safety net that catches regressions and deviations from customer requirements. A well-written test suite can quickly identify any failures that may reduce the value of the software.

Similarly, continuous integration ensures that the project works as a whole multiple times per day. It forces any mistakes or accidents to appear soon after their introduction, when they're easiest to fix.

Beyond Practices

I was once on a project that had been running for four months and was only a few weeks away from its deadline when it was abruptly halted. "We're way over budget and didn't realize it," the manager told me. "Everybody has to go home tomorrow."

We were all contractors, but the suddeness of the project's cancellation surprised us. I stayed on for one more week to train one of the organization's employees—"Joe"—just in case they had the opportunity to pick up the code again in the future.

Although this wasn't an XP project, we had been working in iterations. We hadn't deployed any of the iteration releases, but the last iteration's result was ready to deploy.

Joe and I did so, then spent the rest of the week fixing some known bugs. This might have been a textbook example of good project management, except for one problem: our release wasn't usable. We had worked on features in priority order, but our customers gave us the wrong priorities! A feature they had described as least important for the first release was in fact vitally important: security. That feature was last on our list, so we hadn't implemented it. If we had deployed our interim releases, we would have discovered the problem. Instead, it blindsided us and left the company without anything of value.

Fortunately, the company brought us back a few months later and we finished the application.

Deliver Business Results

Someday that may happen to you. It may not be as dramatic as telling a recurring customer that he'll get better results if you don't write software, but you may have to choose between
delivering code and delivering business results.

Value isn't really about software, after all. Your goal is to deliver something useful for the customer. The software is merely how you do that. The single most essential criterion for your success is the fitness of the project for its business purposes.

For example, agile teams value working software over comprehensive documentation. Documentation is valuable—communicating what the software must do and how it works is important—but your first priority is to meet your customer's needs.

The primary goal is always to provide the most valuable business results possible.

In Practice

XP encourages close involvement with actual customers by bringing them into the team, so they can measure progress and make decisions based on business value every day.

Real customer involvement allows the on-site customer to review these values with end-users and keep the plan on track. Their vision provides answers to the questions most important to the project.

XP approaches its schedule in terms of customer value. The team works on stories phrased from the customer's point of view and verifiable by customer testing. After each iteration, the iteration demo shows the team's current progress to stakeholders, allowing them to verify that the results are valuable and to decide whether to continue development.

Beyond Practices

A friend—"Aaron"—recently spent a man-month writing 1,500 lines of prototype code that generated $7.8 million in revenue during its first demo.

As a graduate student, he interned with a technology company doing research on handwriting recognition with digital pens containing sensors and recording equipment. A customer made an off-hand remark about how useful it might be to use those special pens with printed maps.

Suddenly, Aaron had a research assignment.

The largest potential customer used an existing software package to send map data to field agents to plan routes and identify waypoints. Aaron modified the pen software to

send coordinate information on printed pages. Then he found a way to encode the pen's necessary calibration data on color laser printouts. The final step was to use the API of the customer's software to enter special pen events—mark waypoint, identify route, etc.

In effect, all of his code merely replaced the clunky mouse-based UI with the act of drawing on a custom-printed map, then docking the pen.

A few minutes into the first demo, the customer led the sales rep to the control room for a field exercise. After installing the software and connecting the pen's dock, the rep handed the pen and a printed map to one of the techs. The tech had never seen the product before and had no training, but he immediately circled an objective on the map and docked the pen. In seconds, the objective appeared on the vehicle displays as well as on the PDAs of the field agents.

The customer placed an order for a license and hardware for everyone at the location. That's business results.

Deliver Frequently

If you have a business problem, a solution to that problem today is much more valuable than a solution to that problem in six months—especially if the solution will be the same then as it is now.

Value is more than just doing what the customer needs. It's doing what the customer needs when the customer needs it.

Delivering working, valuable software frequently makes your software more valuable. This is especially true when a real customer promotes the most valuable stories to the start of the project. Delivering working software as fast as possible enables two important feedback loops.

One is from actual customers to the developers, where the customers use the software and communicate how well it meets their needs. The other is from the team to the customers, where the team communicates by demonstrating how trustworthy and capable it is.

The highest priority of any software project is to deliver value, frequently and continuously, and by doing so, to satisfy the customer. Success follows.

In Practice

Once you've identified what the customer really needs and what makes the software valuable, XP's technical practices help you achieve fast and frequent releases. Short iterations keep the schedule light and manageable by dividing the whole project into week-long cycles.

Beyond Practices

According to founder Cal Henderson,* the photo-sharing web site Flickr has practiced frequent delivery from its earliest days. There was no single decision to do so; it was just an extension of how its founders worked. Rather than batching up new features, they released them to users as soon as possible.

The most important component of this process is a group of strong and responsible developers who appreciate the chance to manage, code, test, stage, and deploy features. The rest of the work is standard agility—working in small cycles, rigorous testing, fixing bugs immediately, and taking many small risks.

The results are powerful. When a user posts a bug to the forum, the team can often fix the problem and deploy the new code to the live site within minutes. There's no need to wait for other people to finish a new feature. It's surprisingly low-risk, too.

Seek Technical Excellence

"What's the intellectual basis for design? What does it mean to have a good design?"

Unfortunately, many discussions of "good" design focus on specific techniques. These discussions often involve assumptions that one particular technology is better than another, or that rich object-oriented domain models or stored procedures or service-oriented architectures are obviously good.

Some folks describe good design as elegant or pretty. They say that it has the Quality Without a Name (QWAN)—an ineffable sense of rightness in the design.

My QWAN is not your QWAN. My Truth and Beauty is your Falsehood and Defilement. My beautiful domain models are uglier than your stored procedures, and vice versa. QWAN is just too vague. I want a better definition of good design.

Software Doesn't Exist

When you run a program, your computer loads a long series of magnetic fields from your hard drive and translates them into capacitances in RAM. Transistors in the CPU interpret those charges, sending the results out to peripherals such as your video card.

Yet none of that is software. Software isn't even ones and zeros; it's magnets, electricity, and light. The only way to create software is to toggle electrical switches up and down—or to use existing software to create it for you.

You write software, though, don't you?

Actually, you write a very detailed specification for a program that writes the software for you.
This special program translates your specification into machine instructions, then directs the computer's operating system to save those instructions as magnetic fields on the hard drive.

Once they're there, you can run your program, copy it, share it, or whatever.

The specification is the source code. The program that translates the specification into software is the compiler.

Design Is for Understanding

If source code is design, then what is design? Why do we bother with all these UML diagrams and CRC cards and discussions around a whiteboard?

All these things are abstractions—even source code, so we create simplified models that we can understand. Some of these models, like source code, are machine-translatable. Others, like UML, are not.

Early source code was assembly language: a very thin abstraction over the hardware. Programs were much simpler back then, but assembly language was hard to understand.

Programmers drew flow charts to visualize the design. Why don't we use flow charts anymore? Our programming languages are so much more expressive that we don't need them! You can read a method and see the flow of control.

Before structured programming:

```
1000 NS% = (80 - LEN(T$)) / 2
1010 S$ = ""
1020 IF NS% = 0 GOTO 1060
1030 S$ = S$ + " "
1040 NS% = NS% - 1
1050 GOTO 1020
1060 PRINT S$ + T$
1070 RETURN
```

After structured programming:

```
public void PrintCenteredString(string text) {
int center = (LINE_LENGTH - text.Length) / 2;
string spaces = "";
for (int i = 0; i < center; i++) {
spaces += " ";
}
```

```
Print(spaces + text);
}
```

Design Trade-offs

When the engineers at Boeing design a passenger airplane, they constantly have to trade off safety, fuel efficiency, passenger capacity, and production cost. Programmers rarely have to make those kinds of decisions these days.

The assembly programmers of yesteryear had tough decisions between using lots of memory (space) or making the software fast (speed). Now, we almost never face such speed/space trade-offs. Our machines are so fast and have so much RAM that once-beloved hand optimizations rarely matter.

In fact, our computers are so fast that modern languages actually waste computing resources.

With an optimizing compiler, C is just as good as assembly language. C++ adds virtual method lookups—requiring more memory and an extra level of indirection. Java and C# add a complete intermediate language that runs in a virtual machine atop the normal machine.

Ruby* interprets the entire program on every invocation! How wasteful. So why is Ruby on Rails so popular? How is it possible that Java and C# succeed?

What do they provide that makes their waste worthwhile? Why aren't we all programming in C?

Quality with a Name

A good airplane design balances the trade-offs of safety, carrying capacity, fuel consumption, and manufacturing costs. A great airplane design gives you better safety, and more people, for less fuel, at a cheaper price than the competition.

What about software? If we're not balancing speed/space trade-offs, what are we doing? Actually, there is one trade-off that we make over and over again. Java, C#, and Ruby demonstrate that we are often willing to sacrifice computer time in order to save programmer time and effort.

However, wasting cheap computer time to save programmer resources is a wise design decision. Programmers are often the most expensive component in software development.

If good design is the art of maximizing the benefits of our trade-offs—and if software design's only real trade-off is between machine performance and programmer time—then the definition of "good software design" becomes crystal clear:

A good software design minimizes the time required to create, modify, and maintain the software while achieving acceptable runtime performance.

Great Design

1. **Design quality is people-sensitive**. Programmers, even those of equivalent competence, have varying levels of expertise. A design that design quality relies so heavily on programmer time, it's very sensitive to which programmers are doing the work. A good design takes this into account.

2. **Design quality is change-specific.** Software is often designed to be easy to change in specific ways. This can make other changes difficult. A design that's good for some changes may be bad in others. A genuinely good design correctly anticipates the changes that actually occur.

3. Modification and maintenance time are more important than creation time. It bears repeating that most software spends far more time in maintenance than in initial development. When you consider that even unreleased software often requires modifications to its design. A good design focuses on minimizing modification and maintenance time over minimizing creation time.

4. **Design quality is unpredictable.** If a good design minimizes programmer time, and it varies depending on the people doing the work and the changes required, then there's no way to predict the quality of a design. You can have an informed opinion, but ultimately the proof of a good design is in how it deals with change.

Furthermore, great designs:

• Are easy to modify by the people who most frequently work within them

• Easily support unexpected changes

• Are easy to maintain

• Prove their value by becoming steadily easier to modify over years of changes and upgrades

Universal Design Principles

Universal principles—apply to any programming language or platform—that point the way.

The Source Code Is the (Final) Design

Any design that you can't turn into software automatically is incomplete. If you're an architect or designer and you don't produce code, it's programmers who finish your design for you. They'll fill in the inevitable gaps, and they'll encounter and solve problems you didn't anticipate. Follow your design down to the code.

Don't Repeat Yourself (DRY)

Don't Repeat Yourself is more than just avoiding cut-and-paste coding. It's having one cohesive location and canonical representation for every concept in the final design.

Eliminating duplication decreases the time required to make changes. You need only change one part of the code. It also decreases the risk of introducing a defect by making a necessary change in one place but not in another.

Be Cohesive

A cohesive design places closely related concepts closer together. A classic example is the concept of a date and an operation to determine the next business day. This is a well-known benefit of object-oriented programming: in OOP, you can group data and related operations into the same class.

You can improve cohesion by grouping related files into a single directory, or by putting documentation closer to the parts of the design it documents.
Cohesion improves design quality because it makes designs easier to understand.

Decouple

Different parts of a design are coupled when a change to one part of the design necessitates a change to another part.

Problems occur when a change to one part of the design requires a change to an unrelated part of the design. Either programmers spend extra time finding out these changes, or they miss them entirely and introduce defects. The more tenuous the relationship between two concepts, the more loosely coupled they should be.

Clarify, Simplify, and Refine

If good designs are easy for other people to modify and maintain, then one way to create a good design is to create one that's easy to read.

When I write code, I write it for the future. I assume that people I'll never meet will read and judge my design. As a result, I spend a lot of time making my code very easy to understand.

Fail Fast

A design that fails fast reveals its flaws quickly. One way to do this is to have a sophisticated test suite as part of the design, as with test-driven development. Another

approach is use a tool such as assertions to check for inappropriate results and fail if they occur.

Failing fast improves design by making errors visible more quickly, when it's cheaper to fix them.

Optimize from Measurements

Optimized code is often unreadable; it's usually tricky and prone to defects. If good design means reducing programmer time, then optimization is the exact opposite of good design.

Although well-designed code is often fast code, it isn't always fast. Optimization is sometimes necessary. Optimizing later allows you to do it in the smartest way possible: when you've refined the code, when it's cheapest to modify, and when performance profiling can help direct your optimization effort to the most effective improvements.

Eliminate Technical Debt

Despite our best intentions, technical debt creeps into our systems. removing technical debt, a team can overcome any number of poor design decisions.

Principles in Practice

These universal design principles provide good guidance, but they don't help with specific languages or platforms. That's why you need design principles for specific languages.

Consider the simple and popular "instance variables must be private" design rule. As one of the most widely repeated design rules, it often gets applied without real thought.

It's true that instance variables should often be private, but if you want to understand the rule and when to break it, ask why. Why make instance variables private? One reason is that private variables enforce encapsulation. But why should anyone care about encapsulation?

The real reason private variables (and encapsulation) are good is that they help enforce decoupling. Decoupled code is good, right? Not always. Appropriately decoupled code is good, but it's OK for closely related concepts to be tightly coupled.

However, closely related concepts should also be cohesive. They should be close together in the code. In object-oriented programming languages, closely related concepts often belong in the same class.

Pursue Mastery

A good software design minimizes the time required to create, modify, and maintain the software while achieving acceptable runtime performance.

The same is true of agile software development. Ultimately, what matters is success, however you define it. The practices, principles, and values are merely guides along the way.

Start by following the practices rigorously. Learn what the principles mean. Break the rules, experiment, see what works, and learn some more. Share your insights and passion, and learn even more.

Over time, with discipline and success, even the principles will seem less important. When doing the right thing is instinct and intuition, finely honed by experience, it's time to leave rules and principles behind. When you produce great software for a valuable purpose and pass your wisdom on to the next generation of projects, **you will have mastered the art of successful software development.**