NET OBJECTIVES EZINE
VOLUME 1, ISSUE 2 FEBRUARY 2004

Introducing the Integrated Theory

Dan Rawsthorne, Net Objectives, drdan@netobjectives.com

Alan Shalloway, Net Objectives, alshall@netobjectives.com

(A chapter of forthcoming book: "An Integrated Theory of Effective Software Development")

The Integrated Theory brings together many effective concepts from all aspects of software development. Essentially, agile process is married to effective methods for analysis, design, code and test to create synergy in the overall process.

Net Objectives is a consulting organization that assists software developers from around the world to build effective software solutions. We have one core purpose: to help developers and development teams gain the skills, confidence, and insights to develop software successfully in today's rapidly-changing environments.

This book is the product of the people at Net Objectives, and also others that we've encountered throughout the years. Although it is written and arranged primarily by Dan Rawsthorne and Alan Shalloway, we rely heavily on the thoughts and writings of others on the Net Objectives team and in the general software development community.

The motivation of this book is not that we know the *right* way to develop software – we don't believe that there is a single right way. Our motivation comes from realizing a certain synergy is available from seemingly disparate techniques that are coming to the forefront of software development. These include Agility (eXtreme Programming (XP), Scrum, etc.), refactoring, design patterns, Test-Driven-Development (TDD), Commonality-Variability Analysis (CVA) and others. It is the combination of these methods that we believe heralds a new possibility of developing software in an efficient, responsive way heretofore impossible in a repeatable manner.

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

The Software Development Problem

If you've been developing software you may have the feeling that the situation just doesn't seem right. Things take longer, are more difficult and the results are of lower quality than we feel they should be. We call this the software development problem. Before describing this more clearly, let's first make a statement:

> **A software developers' goal is to provide suitable solutions for users that consist of quality code and that don't cost too much**

Now the problem:

> **Unfortunately, they usually don't succeed**

This problem has three facets:

1. A *Suitable Solution for Users* means a software feature set that is both useful enough and usable enough for solving the user's need at the time of delivery and afterwards.
2. *Quality Code* is code that not only functions correctly, but also can be maintained and extended, as necessary.
3. *Cost* has both money and time components, and must be both predictable and controllable

Tension exists between these three facets because whenever you try to improve one facet, it may adversely affect the other facets. If you

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

try to improve the quality, you end up increasing the cost. If any of these four things receives too much attention at the expense of the others, the overall success of the project may be threatened.

However, in almost every project, delivering on all three fronts simultaneously is necessary for success of the project as a whole.

You must develop a comprehensive set of functionality, at an acceptable level of quality, for an acceptable amount of money and time.

According to The Standish Group (www.standishgroup.com), 66% of all IT-related software projects fail to come in on time, and/or within budget. You may not agree with how they came up with their numbers, but we think you will agree with the qualitative result – software projects typically take longer and cost more than we expected or wanted when the project started.

Why is our failure rate so high?

Three things common to many software development projects that tend to result in project failure are:

- a high rate of change of requirements and priorities
- a focus on building code quickly without paying attention to its quality
- an over-emphasis on control of the project as a means to avoid failure

Here's a story to illustrate these points:

Let's say John, a project manager, is told he has 6 months to build a small reporting system. He is given a list of 20 main functions that need to be implemented. He spends the first month detailing these features, writing detailed use cases for each of them. He then begins the next phase of development by designing an architecture that will support all of these features.

After 6 weeks from the start of the project he is very pleased with his team's progress. He has a full set of requirements and a completely designed architecture. He decides to split the functionality he needs to do into 2 phases. He picks half of the use cases somewhat randomly but in such as way that all aspects of his architecture get used. He then assigns these 10 use cases to his team, giving each one 2 months to complete them. His plan is to complete the other 10 use cases in the following 2 months and will still have 2 weeks to do a complete system test.

Weeks 7, 8 and 9 go pretty well and everyone seems to be on schedule. In another week they should have half of the functionality completed. John tells his customers that a review of the system would be helpful and they agree and schedule it with him.

Unfortunately, in the tenth week, when the use cases are supposed to have been implemented, testing shows them to be poorly coded and designed. John is furious. His developers had been telling him they were on schedule and would be done on time. Unfortunately, they were basing this on the fact that they were concentrating on coding the system. They really didn't know its quality was so bad.

John goes back and tells the customers it'll be another week since they have to test the system more thoroughly. He doesn't expect to have to slip the remaining schedule, however. He figures his team can work harder to make up this lost week. They should be able to do that because they now know more about the system and know they need to test it. Therefore, he figures they can do the remaining 10 use cases in the 9 weeks they have left for it, thinking that's only a week less than the week they had planned for it.

The team works extra hard and gets the first 10 use cases done. John shows the customers the system. They are delighted with his team's work. However, they now can see new things that they want that are more important than what they already asked for. They also see that much of what his team has done doesn't have much value. In fact, the customers pointedly ask John why he selected the use cases he did. Clearly, many of them weren't as important as ones still to be done – not to mention the new ones.

The customers want 5 more features. John points out that they can't reasonably be expected to just do more in the remaining time left. The customers agree. After complaining again that they would have preferred to trade off for 2 use cases he's already done, they agree to allow John not to complete 5 of the remaining use cases.

This highlights a common estimation mistake we've seen. Comparing what should happen against what was planned, not what actually happened. If John is correct and they can do it a week faster, it should be a week faster than 11 weeks – what it actually is taking for the first phase – not a week faster than the 10 weeks estimated. Unfortunately, there is no evidence it will be faster at all.

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

John goes back to his team and sees that he is in trouble. First, he realizes that he wasted 1 week analyzing the 5 use cases he no longer needs to build. Second, it'll take him a week to get details on the 5 new use cases. Thirdly, the architecture they built was more than they needed because the 5 use cases removed no longer need to be supported. Finally, the new functions are not supported well by the architecture.

Swapping out the old features for the new features essentially means that he has an extra week of analysis, a few more days in expanding his architecture and the added difficulty of supporting a broader system than is actually needed (although the customers say they didn't need the 2 low-priority use cases John's team already did, since they were written, they won't allow John to drop support of them.)

John complains bitterly to his team about how customers don't know what they want but that it is now up to them to get the system done. They take on the task and work hours and hours of overtime. The pressure and the now extended architecture take its toll and the quality of the code degrades. The team had built the system assuming they knew what they wanted and now are finding the way they coded things does not allow for modifications…

We think you can see where this is going. We also think that this scenario is much better than what really happens in the real world. At least John showed his customers the results half way through and accepted feedback. If he hadn't, his team may have been able to build the originally asked for software. But it clearly wouldn't have done the job the customers later realized they wanted. The technique of building software without asking for verification leads to what Dan calls "building yesterday's system tomorrow."

There was actually something much more fundamentally wrong going on. First, John assumed going in that what he needed to do could be defined as a significant analysis up front. Thus, he spent 4 weeks doing nothing but analyzing his requirements. Unfortunately, not all of these requirements were implemented. This means most of the time he spent on them was a waste. Secondly, John built an architecture that supported all of these functions but didn't account for others. This was again based on the assumption that he knew what to build. This resulted in both an architecture that wasn't complete enough and had more function than necessary to support features that were never built (thereby raising his costs). Third, the code John's team wrote was not designed to be modifiable, but just to implement what was needed (there was a general assumption on his team that quality code takes longer to write than code that just gets the job done). Finally, John never really knew where his team was because he had to base their progress on their guesses – not on what they could demonstrate was working.

These attitudes often lead to project failure because of the following factors:

- We often start with a high degree of uncertainty but act as if we had a high degree of certainty.
- We have an underlying belief that it takes longer to write quality code that can be maintained, modified, and extended easily.
- We believe we can control the software development process by breaking it down into independent steps.
- We believe that developers should be able to do software development in a fully predictable, controlled fashion.

In software, the needs of users can change very rapidly. Technology can advance quickly, too. So, in addition to starting out with a very high degree of uncertainty, we are constantly impacted by the environments in which we work. In addition, any frameworks the developers have elected to use are often more involved than they thought.

It is only after they get into the code and start working with a given framework that developers start to understand its use. These impacts can result in software developers needing to change code that had the intended

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

functionality, the way it was meant to work in the first place. Unfortunately, this code is often not very modifiable because the developers thought they knew what they needed and that it wouldn't have to change.

Stated plainly: We often have a high rate of change but act as if we don't

Unfortunately, developers' predilection for linear control makes them able to be manipulated by management (and each other) into taking on more than they should. This tendency often makes them inefficient through overwork. This occurs because management doesn't appreciate the technical issues of the software development process and developers feel they should be able to do what management is asking for. In other words, management and developers both feel they should be able to do as they predict. Management because they don't understand the complexities involved and developers because underneath it all, they feel they are supermen.

So how can software projects ever achieve success?

The best way we've found to successfully develop software in the face of uncertainty and changing requirements is by using the concepts of Agility.

We define Agility as the ability to respond with sufficient speed to the forces that are in your way, thus enabling you to remain efficient and effective. Put another way, Agility is the ability to effectively respond to things that can harm your project. Consequently, agility is contextual, as those things trying to knock you off track are distinctly yours.

In this book we act as if agility requires actual quickness. That is, we assume we want to move as quickly and efficiently as we can. However, that this is not always the case. Some industries move and change slowly, which means we can move at a slower, more stately, pace and still be successful. The concept of agility still pertains to these industries, but the implementations may be different. In particular, the code and requirements may not have to be so easily modifiable.

So, this book is written as if you want to move as quickly and efficiently as you can. However, the actual pace at which you move will depend upon the forces, or issues, in your project. By being able to respond to changing environments, priorities, and requirements, you will be able to develop more useful products. This can be a strong competitive advantage.

About the authors --

Dan Rawsthorne

Dan Rawsthorne has been developing software for over 20 years and is an accomplished manager, mentor, coach, consultant, and architect. The projects he has worked on run the gamut – from e-commerce, to databases, to military avionics. He has written and presented papers, chaired seminars at OOPSLA, and he has served as a columnist for C++ Report. He has taught courses on OO modeling and methods at both the University of Denver and the University of Washington. He currently serves on the Advisory Board of the Certificate Program in Objected-Oriented Analysis and Design using UML at the University of Washington. He has reviewed many books, authored the Requirements Modeling chapter in the "Handbook of Object Technology" (1999), wrote the afterword for Jeffries (et al)'s "Extreme Programming Installed" (2001), and contributed many sections to Adolph & Bramble's "Patterns of Effective Use Cases" (2003). He is a Certified Scrum Master, one of five people certified by Alistair Cockburn to teach Writing Effective Use Cases, and he has a PhD in mathematics from the University of Illinois

Alan Shalloway

Alan Shalloway is the CEO and senior consultant of Net Objectives. Since 1981, he has been both an OO consultant and developer of software in several industries. Alan is a frequent speaker at prestigious conferences around the world, including: SD Expo, Java One, OOP, OOPSLA. He is the primary author of Design Patterns Explained: A New Perspective on Object-Oriented Design and is currently co-authoring three other books in the software development area. He is a certified Scrum Master and has a Masters in Computer Science from M.I.T.

Volume 1, Issue 2
February 2004

NET OBJECTIVES

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

The Effective Solution

Quickly responding to changes in the middle of development can be difficult. It also takes a high degree of discipline because you have to coordinate several different activities. Agility is often thought of as the ability to respond quickly. Responding quickly has two possible meanings, however (and we mean both of them). First, it can mean how *early* can you respond? Second, it can mean, when you do respond, how *fast* is your response?

Thus, Agility means to respond as soon as possible to changes that occur in your project as fast as possible, with effectiveness. Of course, to be able to do this requires that you are aware of changes as they occur. This means that you both know accurately where you are and where you want to be. Therefore,

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

agility requires both feedback and validation. That is, we continuously get feedback to verify our assumptions (our stated course) and constantly validate that we are on our selected track (that what we have is working as it should). When where we want to be differs from where we are, we adapt. Additionally, frequent feedback also has a moderating affect on thrashing, which is often a result of responding without proper information. It enables us to respond when we need to and with sufficient information as our basis of action.

Requirements, the Code, Management, and the Team

There are two main activities of software development. First, you must decide what to build, then you must build it (and, after development, you must maintain it). Another way to state this is finding the problem and solving the problem. Things like gathering requirements, acceptance testing, etc., are activities involved with the problem. Other things like designing, architecture, coding, unit testing make up the solution part. Of course, these processes are not independent – they are very much intertwined with each other. We are therefore not suggesting we do one and then the other.

Management and team collaboration are other activities. Both are geared toward maximizing the efficiency and effectiveness[2] of the other activities.

All of these activities have importance in connection with the others. And each of these is *alive* in the sense that it is subject to constant change and it can evolve over time.

In a nutshell, there are four things you've got to do. First, you've got have requirements that are understandable, validated, and traceable. Second, you've got to write quality code that's resilient, robust and modifiable (i.e., changeable). Third, you've got to have a management process that allows you to change your mind and re-prioritize as often as needed. And fourth, you've got have a team that is willing and able to do the first three things. In other words:

- effective requirements
- effective code
- effective (agile) management
- effective team

What are Effective Requirements?

Requirements need to clearly tell us what to build and why. There are two types of requirements: *functional requirements* (which describe what the system must do) and *software requirements* (aka technical requirements, which describe what must be in the system). They must be validatable and easily updated/changed if proven invalid. If requirements do this, then they are effective. Some difficulties with achieving quality requirements (especially functional requirements) are:

1. It may be difficult for users to know what they want initially
2. Once users see a part of the system working, they may get new ideas that cause the requirements to change
3. Once a developer sees what the user is really after, they may get new ideas that cause the requirements to change
4. Even when requirements accurately reflect what is needed, they may not be clear
5. After a series of updates, it is often difficult to read a requirements document, find all the pieces, trace to all the relevant pieces when changes occurred
6. It is often difficult to predict how a change in requirements will affect existing code
7. Depending on the author, the functional requirements can be so intertwined with design decisions that

[2] *A short parable can clarify the meanings of the words "efficiency" and "effectiveness". Assuming you need to get to the roof of a building, efficiency relates to how quickly you can place the ladder against the wall so you can get to the roof. Effectiveness relates to – did you put the ladder on the right wall? (so you get to the right roof).*

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

the developer loses the freedom to design

Unfortunately, these issues have a kind of negative synergy. That is, the way they interact makes the quality of requirements degrade more than the sum of their affects individually.

Therefore, no matter what else you want to do, we feel you need to understand how to elicit and document effective requirements. We believe that the essence of this process is that they should be elicited, documented, and validated in an incremental way.

The best techniques that we have found for identifying, capturing, and documenting requirements are through use cases (using the Cockburn style), the Ever-Unfolding Story (we didn't really invent this term, but one of us was in the room when it happened), and Commonality / Variability Analysis (CVA).

Use cases document functional requirements by describing how the system is to be used from the perspective of the entities/users (not necessarily human) that will interact with the system.

The *Ever-Unfolding Story* identifies and names the software entities that must exist in order to make the use case's scenarios come true.

Commonality / Variability Analysis investigates the similarities, differences, and relationships among these entities. This allows the developer to bridge the gap between functional requirements and the software requirements that must be designed and coded.

Use Cases incorporate several other fundamental principles that improve the effectiveness and traceability of the requirements document. Some of these principles are Intention-Revealing Naming, Simple Grammatical Sentencing, Assumption of Success, and Multiple Precision Levels[3].

We like Cockburn-style use cases, documented in Writing Effective Use Cases) for several reasons, but primarily for these two:

- They are designed to be able to be documented easily and effectively in a wide range of software development cultures
- They allow for easy evolution of the use cases as our understanding changes (that is, they synergize with the Ever-Unfolding Story about to be described).

Employee Spotlight – Jeff McKenna

Net Objectives is pleased to announce the addition of Jeff McKenna as a Senior Consultant. Jeff will present courses and seminars in the San Francisco Bay area.

Jeff McKenna has been involved in the software industry since 1963 in programming, system design, architecture, project management, sales and marketing, and as a small business owner. In the last decade, Jeff has focused on the problem of rapid development of software using object-oriented technology and agile processes.

He has trained, coached, and mentored developers and customers to help them succeed using object-oriented technologies, facilitated implementation of agile processes, provided analysis, architecture, design expertise, testing, and defect tracking for companies large and small. He has given talks and presentations around the world on XP, development processes, software testing and object-oriented development.

[3] *For more information on these topics, see Alistair Cockburn's Writing Effective Use Cases, Addison-Wesley, 2000. ISBN 0201702258.*

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

The *Ever-Unfolding Story* is a process of expanding and modifying the functional scenarios inherent in our use cases to develop more accurate software requirements as we move from functional requirements to code.

After unfolding your use cases' scenarios you will see a direct connection between the needs of the user and the entities that are needed in your code. What's more, the connection is maintainable and changeable even as in the needs of the users evolve over time.

The Ever-Unfolding-Story is a process of starting with a minimalistic description of a use case and expanding it as needed. This enables a quick overview of requirements to work initially. We then expand (unfold) on these requirements when we need to know more about them in order to implement them. Some favorite reasons why we like "The Ever-Unfolding story" are.

- It allows us to unfold as needed and avoid paralyzing up-front analysis.
- It keeps our requirements model, especially our use cases, from being complicated by other requirement information that is better kept separate.
- It allows us to create a navigable requirements model that enables us to quickly and naturally find related information.

Commonality-Variability-Analysis (CVA) is a technique described in Jim Coplien's Multi-paradigm Design In C++[4]. We use it to analyze the entities identified in our domain and use-cases in order to organize them as abstractions and specific cases. This focus on what is varying facilitates the Ever-Unfolding as well. CVA enables us to move from entities found in the functional requirements to the software's structure. This allows us to design our software to easily allow for modifications without over-design or excessive re-work.

A few keys to the success of CVA are:

- It forces us to ask questions that we might otherwise miss.
- It can reduce the amount of redundant requirements we would write by allowing us to organize in a templated fashion.
- It reminds us that although we are given specific use cases and scenarios, we must look for the *abstractions* in our problem domain in order to build quality software

What is Effective Code?

To us effective code means code that does what it is supposed to do and can be easily modified to do something else if it is determined we need it to.

We think effective code is not the result of agility, but rather that it is an enabler of agility. The more effective your code, the faster you can move. The faster you can move, the more responsive to change you can be.

Three techniques that work very well to enable effective code are using Software Patterns, Test Driven Development, and Refactoring. Each of these techniques is a book in itself. However, we hope to give a good enough overview of them in the following chapters for you to see how they interact together and synergize in an Agile process.

Software Patterns, as we mean them, are an extension of the well-known design patterns as described in Design-Patterns: Elements of Reusable Object-Oriented Software[5]. Design patterns are proven, reliable, object-oriented programming techniques that are usually identifiable and are virtually always present (although not always implemented correctly).

[4] *James O. Coplien. Multi-Paradigm Design for C++, Addison-Wesley, 1999. ISBN 0201824671. This is a brilliant book that is really not just for C++ developers. The first half is really about domain analysis and is useful for all developers. An on-line copy of what is essentially the book is available at http://www.netobjectives.com/download/CoplienThesis.pdf*

[5] *Gamma, Helm, Johnson, Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994. ISBN 0201633612. (Still the best reference on design patterns, even if it's starting to become somewhat dated.)*

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

They are often described as solutions to recurring problems that occur in a context. In software we find that many problems occur again and again and that particular approaches work better than others. Unfortunately, design patterns are often overly focused on the design and implementation aspects of our problems.

If you study design patterns long enough, you will learn that there is a common theme to them. That is, there is a particular approach to design that virtually all of the design patterns follow. This approach enables variations in our problem domain to occur without degrading our code[6]. Studying design patterns can also teach us the qualities (characteristics) of good code (i.e., changeable, understandable code).

Used properly, design patterns also give strong insights into analysis, implementation and test. Used this way, the patterns tell us about forces in our problem domain that we should pay attention to (as well as suggestions on how to do that). Knowledge of design patterns gives us insights and a body of knowledge to draw on that can be useful even if the pattern itself is never implemented. Thus, design patterns lead to a deeper understanding of patterns in analysis, design, implementation and test. We call this expanded view of design patterns *Software Patterns*.

Test-Driven-Development (TDD) at first appears to be oxy-moronic – that is, at odds with itself. TDD essentially says to have the considerations of how we will test our system be the driving force in building our system.

There are different flavors of TDD. The most common one is eXtreme Programming's (XP) test-first approach. In it, you actually write your tests before you write your code. After some tests are written, you implement what you need to get them to work. This process results in code that does what you think it does (because all of the actions are tested).

Another way of doing TDD is to consider how to design your code based on making your tests easier to implement and be more complete in coverage. This flavor of TDD may not actually involve writing your tests first, but still has test considerations drive the design and development of your code.

Why does TDD work? In a nutshell, testability is directly related to strong cohesion, loose coupling, encapsulation, no redundancy, and clarity of code. These are actually the characteristics that design patterns achieve as well. In other words, code testability is strongly correlated with code quality.

Refactoring is the process of improving the design of existing code without adding functionality. Refactoring is an old discipline. However, it has recently been popularized by Martin Fowler's Refactoring: Improving the Design of Existing Code[7] It is an essential technique to use if your code is going to change. Refactoring is often used as a way to fix code smells – that is, poor quality code (e.g., code that stinks). Very often, after writing code, the coder will see a better way to have written it. Fowler suggests ways to improve the code at this point. Or, sometimes, after your code is tested, you may find it fails and it needs to be restructured to be more solid (this is technically not refactoring, but is closely related).

However, in any agile project, refactoring occurs in other situations as well. In particular, changing requirements often result in a need to change your design to accommodate these new requirements. In other words, before the requirements, your code was fine. However, now, the new requirements require your design to change. The discipline of refactoring would say to modify the design before adding the new functionality.

[6] *For more information on this, please refer to Design Patterns Explained: A New Perspective on Object-Oriented Design, by Alan Shalloway, James Trott, Addison-Wesley, 2001. ISBN 0201715945.*

[7] *Martin Fowler. Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999. ISBN 0201485672.*

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Software Patterns, Test Driven Development, and Refactoring work together. Although all three of these techniques work well on their own, they are particular effective when used together.

Software patterns help you organize your code to isolate (encapsulate) the implementation of a varying concept. The patterns also give us things to look for that might cause us difficulties. TDD helps you identify variations as well because it is easier to set up and test your code if different implementations can be treated (tested) in a conceptually similar manner. Finally, refactoring serves as a way to handle the inevitable code changes that result from our new requirements and our better understanding of our code as it evolves.

These work together because all three of these techniques are based on the same principles of strong cohesion, loose coupling, encapsulation and no redundancy.[8] Commonality-Variability-Analysis further contributes to these techniques because they help identify, and then isolate, variation – making code more flexible.

What is Effective Management?

Since we know the target is moving, and we've prepared our requirements and our coding processes to be able to react quickly, we especially need a management style that can cope with rapid change.

Fortunately, there are already several different agile project management methods. The most common ones are:

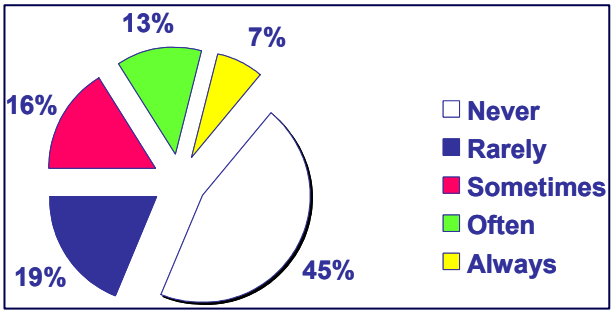
- eXtreme Programming (XP) - Kent Beck
- Scrum - Ken Schwaber and Mike Beedle
- Rational Unified Process (RUP) (when done properly) – Rational/IBM

Others include:

- Dynamic Systems Development Method - (DSDM)
- Crystal Methodologies - Alistair Cockburn
- Feature Driven Development (FDD) - Peter Coad
- Adaptive Software Development - Jim Highsmith

A common theme in all of these methodologies is frequent validation from feedback while incrementally moving toward a more and more complete project. They have the customers steer, so to speak, by picking the most important functions that are needed. This results in the most useful functionality being implemented first and prevents wasting effort on little-used functions or anticipated infra-structures.

The Standish Group (http://www.standishgroup.com) has done extensive research on software projects. One of their more astounding results is that, on average, 45% of software functionality is *never* used. The following graph summarizes their data about the use of software functionality.



Since the software development problem consists mainly of uncertainty and constant change, then the solution must involve continuous reassessment of what is needed and what is most important.

Iterate and validate. Iterate and validate. Plan, perform, and evaluate again. That is the mantra of the agile project manager.

By doing this repeatedly, it is possible to avoid big surprises (or at least not get them late in the game), and to capitalize sooner on

[8] *To see a graphical description of the relationships between these methods, the code qualities described and different approaches suggested, go to http://www.netobjectives.com/dpexplained/dpe_rs.htm*

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

any lessons along the way. This is different from what we normally do, which is to attempt to study our problems and then predict and control the outcomes. Agile methods are often thought of as less plan-based than others. However, they are actually very plan-based – just planned on accurate, up-to-date information, with the plan changing as needed. It enables a business to do its planning based on current truths, rather than outdated assumptions.

What is an Effective Team?

The hardest obstacle for software development teams to overcome is the obstacle of uncertainty and the necessity for change. That is why the main focus of development is centered around agility or reacting quickly to new information.

In order to accomplish this, a team must have a good attitude – the right values, and knowledge of the fundamentals that make agility possible. Fundamentally, the team must *focus on the product* and play to win. They must remain *validation centric* so that mistakes are caught and new information is learned as soon as possible. Finally, they must *work as a team* and work hard to enable communications, promote cooperation, and engender trust in one another.

Fundamental values of such a team are: respect for each other, honesty and a commitment to frequent communication. A team with these values has a chance to succeed. A team without them is sure to fail.

How it all Fits Together

Effective analysis, code, management and team allow for an adaptive process. This is critical for effective development. We believe it is usually a fantasy to assume that one can accurately predict and control what will happen in a software development project. This is especially true with today's, typically complex, projects. It is just too complicated a problem with too many unknowns. Although this is attempted time and again, we find few people who have actually ever had success with it. The closest we've seen is that people built the functionality that was originally asked for on schedule and within budget, but which wasn't the functionality wanted at the end of the project. However, we don't consider this to be a success.[9]

We believe the management of a software development project must be predicated on managing change – not trying to avoid it. To do this you must be validation centric and driven by feedback. In other words – decide on the most important thing to do and do it. Then see if: 1) it was done properly, 2) it was the right thing to do, and 3) it created new information about what to do next.

Given this new information, figure out what to do next and do it – repeating this process.

Difficulties do arise. Some problems with this approach are:

- It takes a good team to pull it off
- It means you can't know everything you want to know about what you are going to do (hence the use of use cases and the ever-unfolding story)
- Your architecture and code must be resilient to change (hence the use of CVA, software patterns, architecture patterns, infrastructure frameworks, TDD and refactoring)
- Your code must be of high quality and not allowed to degrade (hence the use of software patterns and TDD).
- You must continuously test your changing code (hence the use of automated testing techniques embedded in any TDD approach).

Fortunately, the methods we described (and will detail more in future chapters) will solve these challenges.

[9] *This continued attempt at doing things the same way while repeatedly achieving the same types of failure inspired our T-Shirt with a front that reads: "Insanity: Doing the same thing over and over and expecting a different result". On the back it says: "I used to be insane, now I am different." To get such a T-shirt, or any of our other shirts, go to http://www.netobjectives.com/resources/t_shirts.htm*

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

There are other challenges as well. How do we predict what our finished product will look like? What will its final cost be? We may not answer these questions sufficiently for you. However, we do suggest that current methods do not handle them either – they just pretend (fantasize) that they do. Furthermore, agility allows for you to select the functions you feel are most important for implementation while deferring effort on less important functions (thereby saving work and making you more efficient).[10]

By practicing these techniques and understanding why these techniques work so well together, you can enable your team to be more agile and to use the agile project management style – which best enables you to cope with change and mitigate uncertainty.

To illustrate this point, let us tell you another story.

Barry works as a project manager on a team that has adopted some of these agile techniques. He is told he has 6 months to build a reporting system with 20 features. Although his boss told him what to do, Barry is skeptical that these selected features are truly the ones his customers want. He has been burned in the past; given requirements with the assurance that they were complete and correct when they were, in fact, not.

Therefore, Barry spends 3 days getting a broad picture of the features he has been given. Essentially, writing little more than a name for the use case and describing in a few lines what the 'sunny day' scenario is for it. He then talks to his customers and asks which are the most important ones to implement. They prioritize them and he selects the top 5 for some further unfolding.

After another day, he has split the use cases up into 3-4 stories each. That is, each use case is now described as 3-4 functions, with each function being described in terms of how it would be used (hence the term story).

Barry goes back to his customers with his team this time. They all work together to create even more detail into how to implement the stories. At the same time, the customers decide which ones are the most important and which ones should be built first based on their estimated cost. The developers chime in with discussions of which stories are the most interesting from an architectural point of view. The meeting ends when the customers and developers have selected the most important stories (from both a business and architectural point of view) that Barry's team says can be implemented in one month.

Barry and his team then go off and develop these selected stories. Throughout this time, they talk to any customer they need to get additional information. They only worry about implementing the stories they have been given. They do not look ahead to future stories. This keeps their design as simple as it can be at this point. Since the stories they picked include architecturally interesting ones, these simple designs will lead to a robust architecture.

After a month, Barry has almost all of the selected stories completed. He sets up a meeting with his customers to show them what he has done and explains that two stories did not get completed. He explains that he didn't want to delay the meeting until they were done because the customers' feedback was too important to delay it. The incomplete stories can be done first in the next iteration if the customers still think they are important.

The customers are pleased with what they see. However, they now decide that there are other things that they would rather see done. Barry says – no problem. Just add them to the list but put them in the order of expected priority. Barry sees some earlier requests go to the bottom of the list. He knows they will never get to this in the 6 months time allotted and is happy that he didn't spend anytime getting more information on them or worrying about if his architecture would handle them.

He repeats the process he did before – meeting with the customers and his team and selecting the highest ranking use cases for unfolding and prioritization. Although he knows it will take some time to modify his code for this new functionality, he is not worried since his team used the lessons of software patterns to keep the code quality high and he knows they will use TDD and refactoring to make their changes in a disciplined way.

[10] *See earlier Standish Group graph.*

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Contrasting our stories

Both scenarios described in this chapter start the same way and have the same resources available However, in the first one, John acts as if he is told what the system truly needs to be. He then believes he knows where his team is without working code. His attitude is one of being able to see reality and to control it. Thus, when change happens, he thinks it is someone's fault.

Barry, on the other hand, trusts his team and his customers to be honest with him but doesn't believe that they are all-knowing. He assumes that what is true at the beginning of the project will be different from what is true in the middle or end of the project. Thus, he prepares for change because he knows it is going to happen. By doing as little analysis as possible, he is able to get feedback quickly to ensure he is on the right path. By insisting his coders write unit tests, he knows their code works. By coordinating his QA person with his customers, he both clarifies what his customer is asking for and validates that his team is building this.

Let's break this down into a simple table showing the techniques being used by John and Barry over time. Although this example is contrived it does a good job of highlighting the techniques and typical outcomes from these two approaches.

Technique / Activity	John	Barry
Analysis	Starts large up-front product definition and detailed feature list. Time frame: 1 month	Broadly outlines the features required and selected by customers. Time frame: 1 week
Prioritization technique	Does not use customers. Considers architectural needs.	Driven by customers. However, Barry pays attention to design risk of doing certain stories late.
Size of system to develop at a time	Large chunks.	Segments that are estimated to be completed in a month.
Timeframe for feedback	Midway through project.	Continuous throughout development.
Validation timeframe.	Customer review at start of project and QA Testing at end of project.	Monthly.
Architecture	Comprehensive up-front architecture that is designed to incorporate all needs of system.	High-level conceptual architecture driven by "interesting" stories that can be easily modified when new requirements are undertaken.
Testing	Non-developers test to see if quality was achieved.	Unit Testing is built into the process to make sure quality is always present, that code can be modified, and that testing costs are low due to automation.
Customers	Used to define requirements but are kept separate from development team.	Are considered an essential part of the development team.
New Requirements	Considered to be the result of process failure and are dreaded.	Considered to be an opportunity for a competitive edge and are welcomed.

Both teams have the same challenges. However, one tries to avoid the challenge while the other knows it is inevitable and adapts their process to meet it.

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Summary

We see from the example that agility is based on accepting what usually is unavoidable – things (or at least our knowledge of things) will change. Instead of trying to prevent it – we prepare for it. In our process, our team, our requirements, our code and our tests (and anywhere else we can think of). We do this by doing the following:

1. Have an effective process:
   - Be incremental and iterative – do a little, get feedback, correct, do a little more
   - Pay attention to your tools and your environment. Make effective use of your architectural patterns and infrastructure frameworks
2. Manage Requirement Effectively:
   - Defer requirements when possible
   - Document and track our current requirements with written use cases
   - Unfold these use cases as we go.
   - Use Commonality-Variability-Analysis to move towards design
3. Use design and programming techniques that are designed to accommodate the changes we know will occur:
   - Software patterns
   - Test-Driven Development
   - Refactoring
4. Have an effective team:
   - The team works well together and plays to win
   - The team seeks out, and values, feedback
   - The team is focuses on validating all products, all the time

None of these things is really new. However, by using them all together, we get greater value than by using each of them alone. The management style enables us to avoid confusion in the face of uncertainty and change. The ever-unfolding story allows us to evolve our requirements as needed. And our coding techniques are an enabler of the entire process.

To join a discussion about this article, please go to http://www.netobjectivesgroups.com/6/ubb.x and look under the E-zines category.

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Go to http://www.netobjectives.com/subscribe.htm to subscribe to our mailing list and receive future ezines and seminar announcements

Volume 1, Issue 2
February 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Net Objectives – What We Do

Net Objectives provides enterprises with a full selection of training, coaching and consulting services. Our Vision is "Effective software development without suffering". We facilitate software development organizations migration to more effective and efficient processes. In particular, we are specialists in agility, effective analysis, design patterns, refactoring and test-driven development.

We provide a blend of training, follow up coaching and staff supplementation that enables your team to become more effective in all areas of software development. Our engagements often begin with an assessment of where you are and detail a plan of how to become much more effective. Our trainers and consultants are experts in their fields (many of them published authors).

When you've taken a course from Net Objectives, you will see the world of software development with new clarity and new insight. Our graduates often tell us they are amazed at how we can simplify confusing and complicated subjects, and make them seem so understandable, and applicable for everyday use. Many of our students remark that it is the best training they have ever received.

The following courses are among our most often requested. This is not a complete list, though it is representative of the types of courses we offer

Agile Project Management - This 2-day course analyzes what it means to be an agile project, and provides a number of best practices that provide and/or enhance agility. Different agile practices from different processes (including RUP, XP and Scrum) are discussed.

Agile Use Case Analysis - This 3-day course provides theory and practice in deriving software requirements from use cases. This course is our recommendation for almost all organizations, and is intended for audiences that mix both business and software analysts.

Design Patterns Explained: A New Perspective on Object-Oriented Design - This 3-day course teaches several useful design patterns as a way to explain the principles and strategies that make design patterns effective. It also takes the lessons from design patterns and expands them into both a new way of doing analysis and a general way of building application architectures.

Test-Driven Development: Iterative Development, Refactoring and Unit Testing - This 3-day course teaches how to use either Junit, NUnit or CppUnit to create unit tests. Lab work is done in both Refactoring and Unit Testing together to develop very solid code by writing tests first. The course explains how the combination of Unit Testing Refactoring can result in emerging designs of high quality.

Effective Object-Oriented Analysis, Design and Programming In C++, C#, Java, or VB.net - This 5-day course goes beyond the basics of object-oriented design and presents 14 practices which will enable your developers to do object-oriented programming effectively. These practices are the culmination of the best coding practices of eXtreme Programming and of Design Patterns.

Software Dev Using an Agile (RUP, XP, SCRUM) Approach and Design Patterns - This 5-day course teaches several design patterns and the principles underneath them, the course goes further by showing how patterns can work together with agile development strategies to create robust, flexible, maintainable designs

If your user group or company is interested in Net Objectives making a free technical presentation for them, or if you would like to be notified of upcoming Net Objectives events, please visit our website, or contact us by the email address or phone number below.

www.netobjectives.com • mike.shalloway@netobjectives.com • 404-593-8375