

Traditional project teams take a (near) serial approach to development in which requirements are defined and documented early in the project, they may or may not be reviewed and accepted by the project stakeholders, and they're provided to developers. Requirements creep, also known as scope creep, is contained through a change management process (or more accurately, a **change prevention process**). Traditional change management processes typically involve one or more people, often referred to as a change control board (CCB), to act as a gateway through which changes are evaluated and potentially accepted. The goal is usually to minimize, if not prevent, requirements creep so as to stay on budget and schedule. In this article I argue that BRUF leads to **significant wastage**, that an **evolutionary approach to development** is must less financially risky than serial development, and that you should take an **agile approach** to requirements.

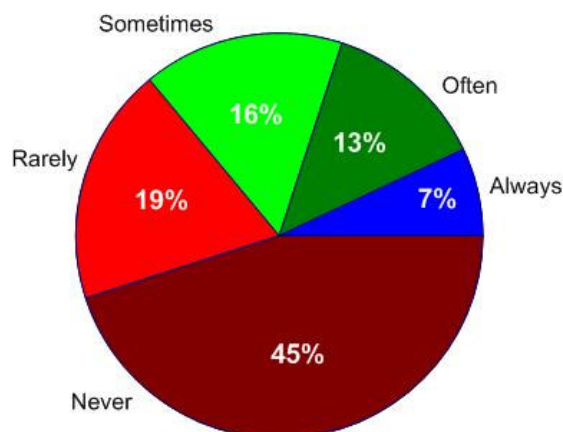
[Ads by Google](#)

## BRUF Leads to Significant Wastage

The **Chaos Report** shares some interesting statistics pertaining to the effectiveness this sort of approach. The Chaos Report, looks at thousands of projects, big and small, around the world in various business domains. The Chaos Study reports that 15% of all projects fail to deliver at all, and that 51% are challenged (they're severely late and/or over budget). However, the Standish Group has also looked at a subset of traditional teams which eventually delivered into production and asked the question, "Of the functionality which was delivered, how much of it was actually used?" The results are summarized in **Figure 1**: an astounding 45% of the functionality was never used, and a further 19% is rarely used. In other words, on these so-called "successful" projects there is significant wastage.

**Figure 1: The effectiveness of a serial approach to requirements.**

Average percentage of delivered functionality actually used when a serial approach to requirements elicitation and documentation is taken on a "successful" information technology project.



Source: Chaos Report v3, Standish Group.

Copyright 2005-2006 Scott W. Ambler

This wastage occurs for several reasons:

1. **The requirements change.** The period between the time that the requirements are "finalized" to the time that the system is actually deployed will often span months, if not years. During this timeframe changes will occur in the marketplace, legislation will change, and your organization's strategy will change. All of these changes in turn will motivate true changes to your requirements.
2. **People's understanding of the requirements change.** More often than not what we identify as a changed requirement is really just an improved understanding of the actual requirement. People aren't very good at predicting what they want. They are good at indicating what they sort of think that they might want, and then once they see what you've built then can then provide some feedback as to how to get it closer to what is actually needed. It's natural for people to change their minds, so you need an approach which easily enables this.
3. **People make up requirements.** An unfortunate side effect of the traditional approach to development is that we've taught several generations of stakeholders that it's going to be relatively painless to define requirements early, but relatively painful to change their minds later. In other words, they're motivate to brainstorm requirements which they think that they just might need at some point, guaranteeing that a large percentage of the functionality that they specify really isn't needed in actual practice.

Why do organizations choose to work this way? Often, they want to identify the requirements early in the lifecycle so that they can put together what they believe to be an accurate budget and schedule. This ignores the fact that in the past that this hasn't worked out very well (51% of projects are arguably challenged) but I guess hope springs eternal. Yes, the more information that you have the better your budget and schedule can be, but software development is so dynamic that at best all we can do is define a very large range at the beginning of a project and narrow it down as the project progresses. Although such an estimates may be soothing to the people desperately trying to manage the project, I personally wouldn't want to invest much effort in putting this estimate together, particularly when there are **better alternatives**.

Many organizations tend to treat custom software as if they were purchasing it off the shelf -- they treat it as a one time purchase. The reality is that you're building a system which will be maintained and evolved for many years to come, it's really a product (involving many releases over time, often implemented as a series of "projects") that you need to manage, not a project. Therefore, a fixed set of requirements with a budget tied to meeting those requirements is an unrealistic way to work.

Many organizations are convinced that software is very hard to change. This is true if it's difficult to redeploy newer versions of the software, but most organizations these days have automated their release process for the majority of their systems. The data community often convinces themselves that it's difficult to **refactor databases**, which is simply not true, or that they must negotiate a **one data truth** in order to proceed with development (also not true).

Worse yet, some people naively think that developers do their work based on what is in the requirements document. If this was true, wouldn't it be common to see printed requirements documents open on every programmer's desk, or displayed in a window on their screens? If you walk around your IT department one day I suspect you'll only need one hand to keep track the number of people actually doing this.

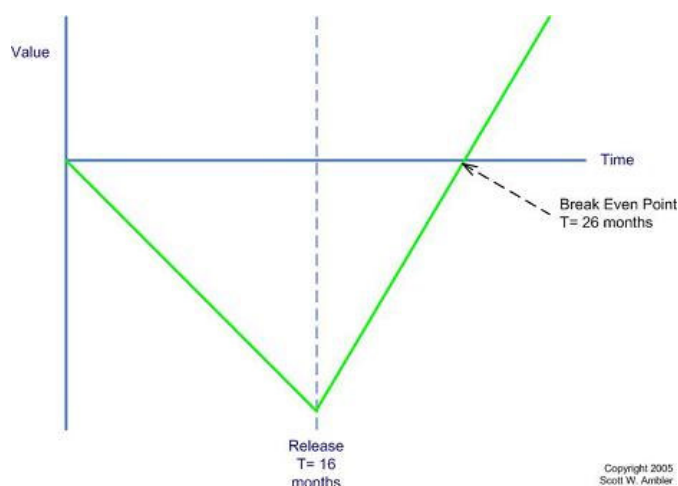
An interesting question that wasn't asked, to my knowledge, was how much required functionality wasn't delivered as the result of a **BMUF** approach. We have no measure, yet, of the opportunity cost of this approach.

To recap, many organizations are motivated to comprehensively define requirements early in a project in order to put together an accurate estimate and schedule. Ignoring the fact that most projects are challenged, even when this approach is successful as **Figure 1** shows roughly two thirds of your investment is a waste. Although this should be enough to motivate you to change your approach, it just gets worse.

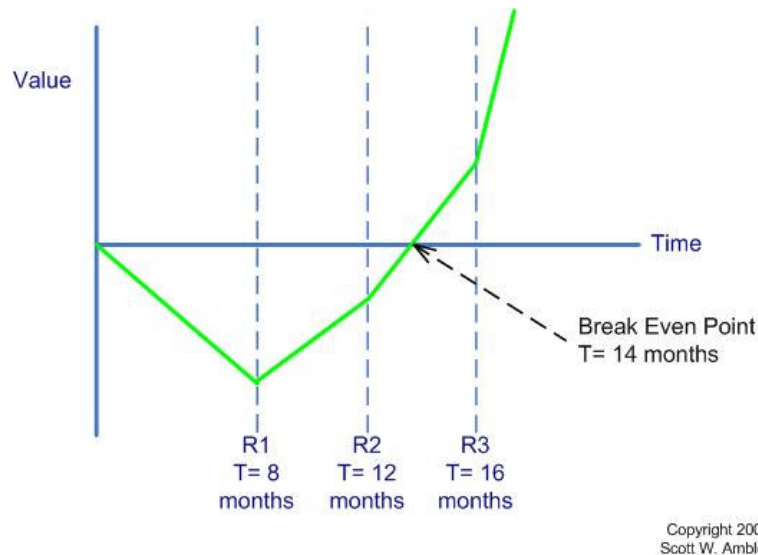
## Take an Evolutionary Approach to Development

Assuming that you can live with two-thirds wastage, as the majority of organizations seem to do, a serial approach to development is still a poor way to work. **Figure 2** depicts the break even graph for a fictional project team taking a serial approach to development whereas **Figure 3** shows the similar graph for a team taking an evolutionary approach. With an evolutionary approach you work iteratively and deliver incrementally. With an iterative approach, instead of trying to define all the requirements up front, then developing a design based on those requirements, then writing code, and so on you do a little modeling, then a little coding, then some more modeling, and so on. When you deliver incrementally you release the system a piece at a time. As you can see, the serial team developed the entire system over a 16 month period whereas the evolutionary team had an initial release after 8 months, then another 4 months later, then one more after another 4 months.

**Figure 2. Break even for a serial project.**



Copyright 2005  
Scott W. Ambler

**Figure 3. Break even for an evolutionary project.**

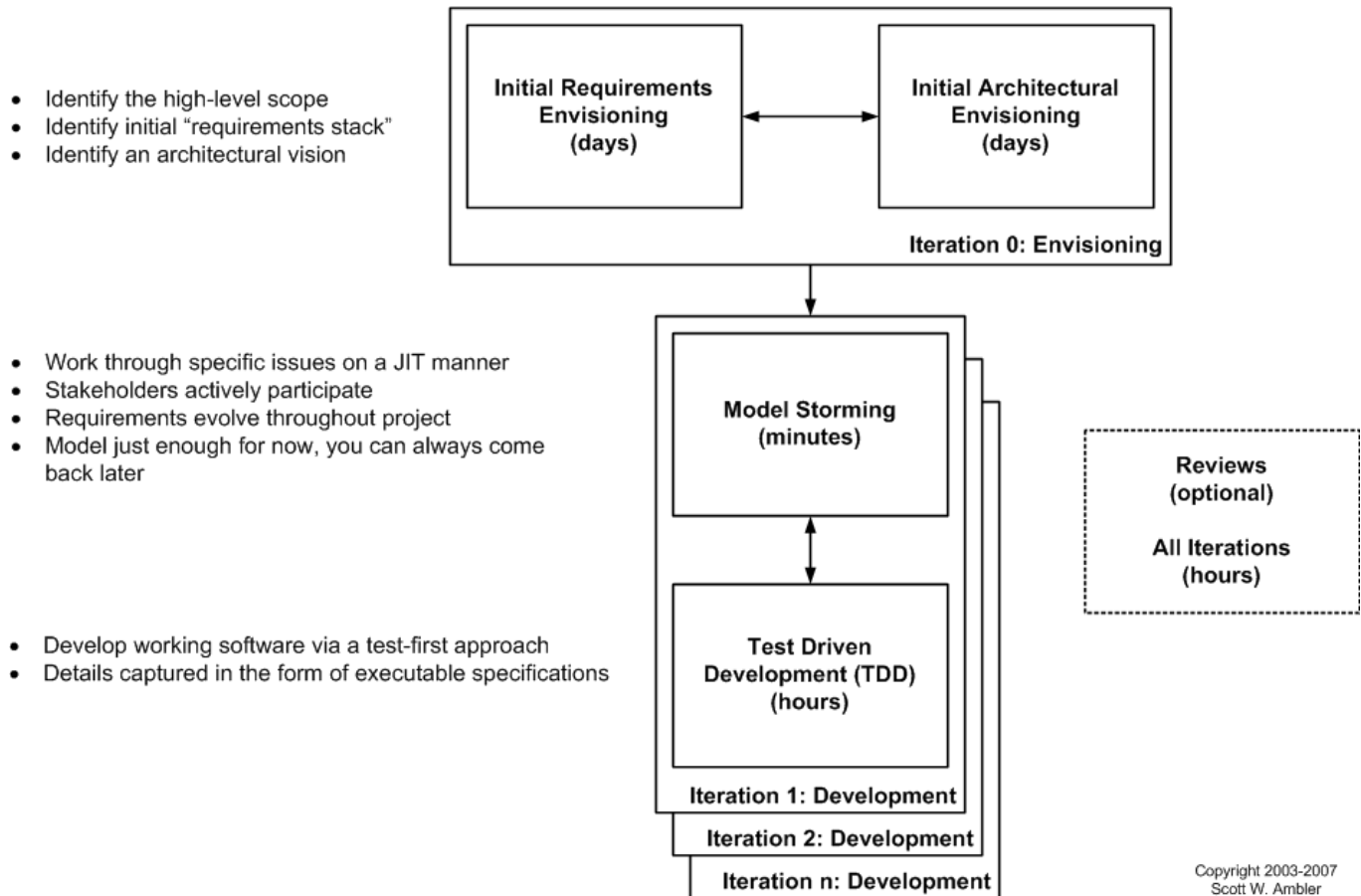
The serial team's break even point occurs 26 months after the start of the project, whereas the evolutionary team's achieves break even after 14 months. With the evolutionary approach you not only spend less money at first, you also start receiving the benefits of having working software in production earlier. Better yet, if you develop the highest priority requirements first (more on this in a minute), you gain a higher rate of return on your initial investment with the evolutionary approach. Furthermore, the evolutionary approach is less risky from a financial point of view. Not only does it pay off earlier, always a good thing, you can easily stop development if you realize that an interim release really does deliver the critical functionality that you actually need.

So, to recap once again, a traditional approach to software develop result in roughly **two-thirds wastage** in delivered software. Worse yet, this approach produces a longer break even point than does an evolutionary approach, increasing the risk to your project. In short, although waterfalls are wonderful tourist attractions they are spectacularly poor ways to organize software development projects.

## Consider an Agile Approach

So what is the solution? First, as **Figure 4** depicts you should take an agile approach to **requirements elicitation** where initial requirements are **initially envisioned at a very high level** at and then the **details are model stormed** on a just-in-time (JIT) basis. For the first release of a system you need to take several days to identify some high-level requirements as well as the scope of the release (what you think the system should do). Model storming sessions are typically impromptu events, one project team member will ask another to model with them, typically lasting for five to ten minutes (it's rare to model storm for more than thirty minutes). The people get together, gather around a shared modeling tool (e.g. the whiteboard), explore the issue until their satisfied that they understand it, then they continue on (often coding). Model storming is JIT modeling: you identify an issue which you need to resolve, you quickly grab a few team mates who can help you, the group explores the issue, and then everyone continues on as before. Extreme programmers (XPer) would call modeling storming sessions stand-up design sessions or customer Q&A sessions.

**Figure 4. The Agile Model Driven Development (AMDD) lifecycle for software projects.**

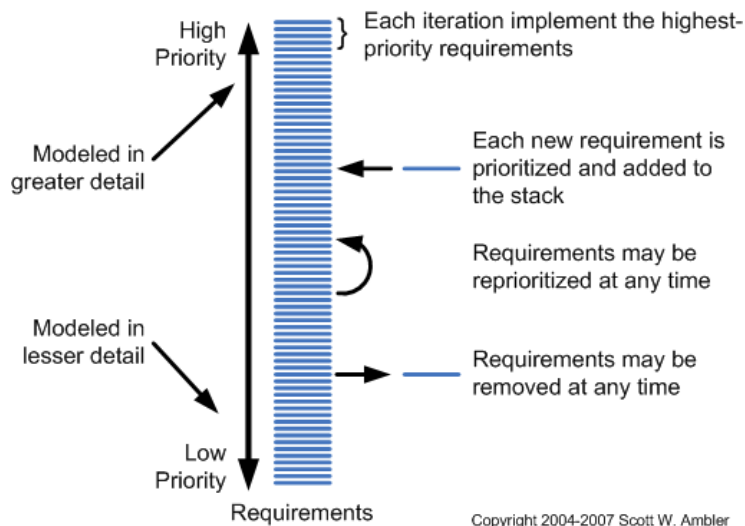


This streamlined, JIT approach to requirements elicitation works for several reasons:

1. **You can still meet your "project planning needs"**. By identifying the high-level requirements early you have enough information to produce an **initial cost estimate** and schedule.
2. **You minimize wastage**. A JIT approach to modeling enables you to focus on just the aspects of the system that you're actually going to build. With a serial approach, you often model aspects of the system which nobody actually wants, as **you learned earlier**.
3. **You ask better questions**. The longer you wait to model storm a requirement, the more knowledge you'll have regarding the domain and therefore you'll be able to ask more intelligent questions.
4. **Stakeholders give better answers**. Similarly, your stakeholders will have a better understanding of the system that you're building because you'll have delivered working software on a regular basis and thereby provided them with **concrete feedback**.

Because requirements change frequently you need a streamlined, flexible approach to requirements change management. Agilists want to develop software which is both high-quality and high-value, and the easiest way to develop high-value software is to implement the highest priority requirements first. Agilists strive to truly manage change, not to prevent it, enabling them to maximize stakeholder ROI. **Figure 5** overviews the **agile approach to requirements change management**, reflecting both **Extreme Programming (XP)**'s planning game and the **Scrum methodology**. Your software development team has a stack of prioritized and estimated requirements which needs to be implemented – XPers will literally have a stack of **user stories** written on index cards. The team takes the highest priority requirements from the top of the stack which they believe they can implement within the current iteration. **Scrum** suggests that you freeze the requirements for the current iteration to provide a level of stability for the developers. If you do this then any change to a requirement you're currently implementing should be treated as just another new requirement.

**Figure 5. An Agile Approach to Change Management.**



New requirements, including defects identified as part of your user testing activities, are prioritized by your project stakeholders and added to the stack in the appropriate place. Your project stakeholders have the right to define new requirements, change their minds about existing requirements, and even reprioritize requirements as they see fit. However, stakeholders must also be responsible for making decisions and providing information in a timely manner.

At first this strategy appears to be a radical approach to requirements management, but it has several interesting benefits:

1. **Stakeholders are in control of the project scope.** They can change the requirements at any point in time, as they need to.
2. **Stakeholders are in control of the project budget.** They can fund the project for as long as they want, to the extent that they want. When you reach the end of the budgeted funds, you either decide to continue funding the effort or you decide to put whatever you've got into production.
3. **Stakeholders are in control of the project schedule.** See point #2.
4. **Wastage is minimized.** Because the project is continually driven by the stakeholders' priorities, each increment provides the greatest possible gain in stakeholder value. Because stakeholders review each incremental delivery, progress is always visible and it's unlikely that the team will be sidetracked by unimportant issues or deliver something that doesn't meet stakeholder needs. In other words, developers **maximize stakeholder return on investment.**

The only major impediment to this approach is cultural inertia. Many organizations have convinced themselves that they need to define the requirements up front, that they need comprehensive documentation before project teams can begin development. They often do this in the name of minimizing financial risk, they want to know what they're going to get for their money, and as a result incur incredible wastage and actually increase financial risk by delaying the break even point of their deployed system. Choosing to succeed is one of the hardest things that you'll ever do.

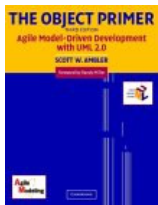
## Acknowledgements

I'd like to thank Phil Doherty, Daniel Hoey, Huet Landry, and Scott Preece for their feedback regarding this article.

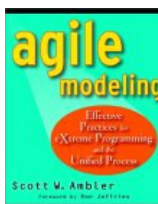
## Recommended Resources

- **Active Stakeholder Participation**
- **Agile Analysis**
- **Agile and Iterative Development: A Manager's Guide.** This book, written by Craig Larman, includes a very interesting chapter on the evidence for why traditional approaches don't seem to be that effective and why evolutionary approaches are. This is a great book in general, but the evidence chapter alone is well worth the money because it will really start to open your eyes.
- **Agile Requirements**
- **Agile Requirements Change Management**

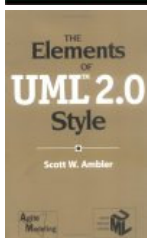
- **Big Modeling Up Front (BMUF) Anti-Pattern**
- **The "Change Prevention" Anti-Pattern**
- **Detailed Requirements Specification: A Worst Practice?**
- **Initial High-Level Architectural Envisioning**
- **Initial High-Level Requirements Envisioning**
- **Is Agile Crossing the Chasm?**
- **IT Projects Sink or Swim.** This paper by Andrew Taylor in the January 2000 issue of the British Computer Society's Computer Bulletin overviews a study of 1,027 IT projects cited scope management related to serial practices as the single largest contributing factor to project failure in 82% of the projects and was given a overall weighted failure influence of 25%.
- **Overcoming Common Requirements Modeling Challenges**
- **Lean Software Development** by Mary and Tom Poppendieck, describes how agile/lean techniques can minimize the wastage on software development projects. This book is a must read for anyone interested in SPI.
- **The "Model a Bit Ahead" Pattern**
- **The "One Truth Above All Else" Anti-Pattern**
- **The Process of Database Refactoring**
- **Rethinking How You View Requirements Management**
- **Rethinking the Role of Business Analysts**
- **The Unchangeable Rules of Software Change** (Brad Appleton, Robert Cowham, and Steve Berczuk)



**The Object Primer 3rd Edition: Agile Model Driven Development with UML 2** is an important reference book for agile modelers, describing how to develop 35 **types of agile models** including all 13 **UML 2 diagrams**. Furthermore, this book describes the techniques of the **Full Lifecycle Object Oriented Testing (FLOOT)** methodology to give you the fundamental testing skills which you require to succeed at agile software development. The book also shows how to move from your agile models to source code (**Java** examples are provided) as well as how to succeed at implementation techniques such as **refactoring** and **test-driven development (TDD)**. The Object Primer also includes a chapter overviewing the critical database development techniques (**database refactoring, object/relational mapping, legacy analysis, and database access coding**) from my award-winning **Agile Database Techniques** book.



**Agile Modeling: Effective Practices for Extreme Programming and the Unified Process** is the seminal book describing how agile software developers approach **modeling and documentation**. It describes principles and practices which you can tailor into your existing software process, such as **XP**, the **Rational Unified Process (RUP)**, or the **Agile Unified Process (AUP)**, to streamline your modeling and documentation efforts. Modeling and documentation are important aspects of any software project, including agile projects, and this book describes in detail how to **elicit requirements, architect**, and then **design** your system in an agile manner.



**The Elements of UML 2.0 Style** describes a collection of standards, conventions, and **guidelines** for creating effective **UML diagrams**. They are based on sound, proven software engineering principles that lead to diagrams that are easier to understand and work with. These conventions exist as a collection of simple, concise guidelines that if applied consistently, represent an important first step in increasing your productivity as a modeler. This book is oriented towards intermediate to advanced UML modelers, although there are numerous examples throughout the book it would not be a good way to learn the UML (instead, consider **The Object Primer**). The book is a brief 188 pages long and is conveniently pocket-sized so it's easy to carry around.

## Let Me Help

I actively work with clients around the world to improve their information technology (IT) practices as both a mentor/coach and trainer. A full description of what I do, and how to contact me, can be **found here**.



Copyright 2006-2007 **Scott W. Ambler**

Last updated: March 3, 2007

This site owned by **Ambysoft Inc.**

|| **Agile Data (AD)** | **Agile Unified Process (AUP)** | **Enterprise Unified Process (EUP)** | **My Writings** ||