Principles and Practices of Software Development

Daniel Huttenlocher[1,3] Daniel Spoonhower[2,3]
dph@cs.cornell.edu spoons@cs.cmu.edu

[1]Computer Science Department and
Johnson Graduate School of Management
Cornell University
Ithaca, NY 14853

[2]Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

[3]Intelligent Markets, Inc.
San Francisco, CA 94103

Draft of August 26, 2002

Abstract

In this paper, we aim to provide a new perspective on the methods of software development. We find most software development methodologies to be overly prescriptive; they focus primarily on what methods to use without adequate consideration of what problems are being addressed and without a comparison of different methods that apply to the same problem. We maintain that software developers can be most effective if they are provided both with a variety of methods from which to choose and with the understanding necessary to select the methods that are best suited to their project. To that end, we introduce a framework for describing the *principles* of software development, a vocabulary for characterizing and understanding the constraints under which software is being written.

We present some principles of software development, relating them to specific problems that occur in software projects and to practices that are used to address those problems. We observe that the practices of different methodologies can be understood in terms of how they weigh the relative importance of the underlying principles. We illustrate how identifying such principles can help in selecting the most appropriate practices. We note that some of the most significant differences among methodologies arise when principles provide conflicting viewpoints. We then discuss how iterative or incremental software development practices can be used to minimize the risks that result from conflicting principles.

1 Introduction

Most books and articles on developing software focus on the methods or practices of software development without much consideration for when those practices are applicable or how one set of practices relates to another. Some development methodologies do, however, present an underlying philosophy, either implicitly or explicitly. For example, Extreme Programming (XP) [1] [2] and the related Agile Software Development [6] methodologies place significant value in the role of the developer as a creative individual. This can be seen as a reaction against what advocates of these methodologies perceive as an implicit devaluation of the developer in software engineering approaches such as the Personal and Team Software Processes (PSP and TSP) advocated by the Software Engineering Institute (cf. [8] [10]).

Unfortunately, this combination of specific practices together with abstract philosophies does not provide a software developer or manager with much help in understanding which practices are most useful for improving their own development process. Worse yet, many software development methodologies claim that it is necessary to adopt all their practices; they can't or don't explain how individual practices might applied in isolation. In our own software development work (on an enterprise software product for securities trading) we grew frustrated by the difficulty of determining what practices might be useful for our project. As a result, we have begun developing a set of principles of software development that we hope will be of value to other software developers in evaluating and comparing various development practices.

The current state of the literature on software development can be caricatured as having two camps, the "scruffy" practitioners (for example, the advocates of the Agile methodologies) and the "neat" theoreticians, as exemplified by the Software Engineering methodologists. Those in the scruffy camp tend to come from the front lines of recent commercial software development projects and are attempting to generalize their experience to help other developers. Those in the neat camp tend to come from research laboratories or organizations that study software development and have often based their studies on large-scale government-sponsored software development projects (e.g. in space agencies or the military). The scruffy practitioners often find the methodologies recommended by the theoreticians to be difficult to apply in the field, while the neat theoreticians often criticize the approaches of the practitioners as unprincipled and dangerous. One of our goals is that by articulating explicit principles of software development, and relating those principles to the practices advocated in each camp, it will be possible to start a richer dialog that will assist software developers in all types of projects. Like the practitioners, we would like to relate some of the successes and failures of our experiences, but we would also like to begin creating a broader framework for understanding the implied tradeoffs.

1.1 Terminology

One analogy that we have found useful in thinking about software development is a comparison to the application of algorithms in programming. Just as an algorithm suggests a particular implementation in source code, a software development practice suggests a particular implementation for a given software project. Like an algorithm, a particular software development practice is only appropriate for solving certain problems. Just as different algorithms for the same problem can be understood in terms of certain underlying constraints and principles, we believe that different software development practices can be understood in terms of higher level abstractions. We further believe it is critical to recognize that, as in the application of algorithms to programming, there is no single correct set of software practices.

Drawing on this analogy, we try to be careful to distinguish between principles of software development, problems faced by developers, and practices used to overcome those problems. In particular, we believe it is important to distinguish between *principles*, *problems*, *practices* and *implementations*. We begin by defining each of these terms, with the goal of creating a common vocabulary that might be used to better understand the kinds of problems that a software development project encounters and make it easier to determine what methods might appropriate to that project.

- A *principle* is a comprehensive and fundamental law, doctrine, or assumption. Principles may be universal, or they may apply only to certain types of projects.

- A *problem* is something that can get in the way of rapidly developing high quality software that meets customer needs, while having fun doing it.[1]

- A *practice* is a way of acting or working so as to avoid or to alleviate problems (we use the term practice rather than process, because to us it more explicitly captures the notion of how things are actually done as opposed to how someone thinks they should be done).

- An *implementation* is a specific way of following a practice, and reflects a level of specificity that is most appropriate to a given software project. (Most implementations are at a level of detail that is beyond the scope of this paper.)

A good principle, like a law of nature, is more than a statement of something that is (or can be) true. It is a statement that helps expand our understanding of our environment. A good principle is predictive, broadly applicable, and can easily be related to prior experience. Together, a set of principles provide a context for a software project and help guide the choice of appropriate practices. Often, problems can be understood in reference to a small set of principles and can be solved, minimized, or avoided by following certain practices in accordance with those principles.

The discussion at the beginning of this article can be rephrased using this terminology, as follows. Approaches to software development (or software engineering) tend to focus on practices and implementations, rather than on problems and principles. In our view, it is important to understand all four of these aspects of software development and how they relate. Practices alone are often not particularly useful, because one wants to understand what methods are applicable before undertaking the effort of applying them; the cost of choosing the wrong practices can be catastrophic to a project or even an entire company. Our immediate goal in this paper is to articulate a number of principles about software development, relating those principles to problems that occur, and to practices that can address those problems.

Next we consider a classic software development problem, a late project, and discuss some principles and practices related to that problem. We then investigate two of the principles underlying this problem in more detail as those two principles provide opposing viewpoints. Similar competing principles can be seen to underlie many software development practices. In particular, incremental

[1] Some people may wonder what fun has to do with it, after all software development is serious business. We firmly believe that as a creative activity, software development should be fun. That doesn't mean it is always fun – writing an article isn't always fun either – but overall it is more fun than not.

or iterative approaches to software development have arisen as a means of balancing competing principles and minimizing the risk that comes from conflicting underlying needs and goals. Extreme Programming (XP), the Rational Unified Process (RUP) and Rapid Application Development (RAD) approaches are all examples of such iterative methodologies. We subsequently turn to a more detailed discussion of practices advocated by these methodologies and to some related problems and principles.

1.2 A Classic Example: The Late Project

One of the most well-known problems in software development, articulated by Brooks in his classic book on software project management, *The Mythical Man Month* (henceforth *MMM*) [5], is reflected by Brooks's Law:

> *Adding manpower to a late software project makes it later.*

In our terminology of principles, problems, practices and implementations, this "law" contains both a problem and a practice, but it does not constitute a principle of software development. The problem described in this statement is a missed deadline or milestone; the practice is adding more developers to the project. According to Brooks's Law, that particular practice is not a good way of addressing this problem. In *MMM*, Brooks discusses some effects of adding more people to a project and provides a context for understanding when adding more people is a bad way of getting back on schedule.

In order to address the problem of a late software project, there are several possible practices, including the one stated above, that can be applied:

- Add more people to the project.
- Make people work harder.
- Focus more on writing code, less on design and testing.
- Reduce the scope of the project.
- Create a more realistic schedule with a later delivery date.

A number of principles can be used to help understand which of these practices (individually or together) might be most applicable to a particular software project. While Brooks does not specifically state these as principles in *MMM*, he does discuss the first two.

Principle of Group Overhead Communication overhead grows faster than linearly with group size (perhaps quadratically).

Principle of Ramp-Up Slowdown Adding people to a project produces a temporary slowdown as those people learn about the project, and the work is re-distributed. This is a much larger effect than Group Overhead.

Principle of Intellectual Focus Developing software is an intellectual activity that requires focus. There is a limit to the length of time for which a person can stay focused.

Principle of Clear Statement A clear and concise statement of user needs generally results in the development of better software.

Principle of System Dependability The degree of confidence in the reliability of a software system should be proportional to the cost of its failure and the required repair.

Principle of Real Use Understanding the needs of users and validating that those needs have been met is best done with a real implementation of the software and real users.

If one accepts this set of principles, then they provide a basis for choosing among the various practices for addressing a late project listed above. Our experience is primarily with the development of enterprise software, and we have found these principles to be quite useful. Other kinds of software development projects, such as shrink-wrap desktop applications, web-based applications, or projects involving both hardware and software might find different principles applicable. Brooks's experiences related in *MMM* arose largely from developing mainframe operating systems, which shares considerable similarity with current-day enterprise software (though with more interaction with hardware development). It is no surprise that several of these principles also arise in his discussion.

The principles of Group Overhead and Ramp-Up Slowdown suggest that the practice of adding more people to the project will not, at least in the short term, help get a late project back on track. However, the Ramp-Up Slowdown is a temporary phenomenon, so for a long-term project it may be a good idea to add people if the project falls behind schedule in the early stages (contrary to Brooks's Law). In so doing, it is important to realize that this will temporarily make the problem worse, but then will eventually help as the new people get up to speed. In addition, the Group Overhead principle suggests that an increase in staff will result in permanently higher overhead.

The principle of Intellectual Focus suggests that the practice of making people work harder is only of value if people are not working hard already (this is usually not the case with late projects, but sometimes it is true). It can often be effective to have people work harder for a short time period (perhaps two weeks or less) particularly to complete a project or major milestone. However, we believe that there is a fine line between pushing people to get to the next goal and continually pushing people to the point where they are no longer focused and thus become less effective. There is often the temptation for managers to see longer work hours as the easiest way of getting back on track, the principle of Intellectual Focus should help one realize both that this is not a solution to a missed deadline, nor is it without potential damage to the quality of the product and to the team.

The principles of Clear Statement and System Dependability should generally dissuade one from the practice of focusing more on writing code and less on requirements and testing. While reducing non-coding activities may cause code output to increase, and in the short term cause goals to apparently be reached more quickly, these activities are critical to producing a system that is high quality and meets user needs. A word of caution here, however, to the tendency in some projects (encouraged by some software development methodologies) to gather requirements "forever," before getting to coding. The Clear Statement principle alone could be taken to support such extensive up-front analysis, however, the principle of Real Use provides a critical balancing view. The requirements for a system cannot be well validated without an implementation of the software, because the best validation occurs with real users experiencing a real system.

The Real Use principle further suggests that the practice of setting a later ship date has limited utility. What makes this practice problematic is that it postpones the time when real users can sit in front of a real system to provide feedback. Sometimes delaying the schedule is necessary, but in our experience, it is usually better to stick to (or close to) a date. The best way to do this is to cut the scope and to deliver some important and usable functionality. This allows users to get in front of a system, albeit one with less functionality than originally planned, and provide feedback. Cutting the scope is becoming recognized as the best practice for late software projects. Of course in the extreme, even this becomes problematic if the reduced scope offers little or no functionality that is of use and can be validated by users.

1.3 Competing Principles

The principles of Clear Statement and Real Use provide competing viewpoints; the former says that to build good software you need a clear understanding of the problem, whereas the latter says that you generally can't get a clear understanding until the software is in use. We view this pair of competing principles as so central to software development that we now consider them in more detail before turning to the practices of incremental development methodologies such as XP, RUP and RAD.

Most approaches to software development are driven by at least one of these two principles. Some of the earlier methodologies, such as the Waterfall Model [3], can be seen as drawing heavily on the Clear Statement principle. In the Waterfall Model, distinct stages follow in succession: requirements gathering, product design, system architecture, technical design, coding, integration, testing and deployment. In such a staged approach, detailed specifications of what the software should do are created prior to beginning the technical design or coding. The goal is to avoid wasted effort in designing, developing and testing something that is not clearly understood.

It is quite natural that early software development methodologies (such as the Waterfall Model) started with detailed specification practices related to the Clear Statement principle because software systems are so unconstrained compared with physical systems. This flexibility allows software to be built even though its design is inconsistent or poorly understood, yielding an end result that is often nearly useless. In contrast to software, in mechanical or electronic systems, a poor understanding will often lead to violating some physical constraint that reveals a problem or inconsistency much earlier in the design and development process.

In practice, however, the Waterfall Model has often been quite problematic for software development. While detailed and carefully validated requirements can be used to create a good understanding of a system before it is built, the resulting software is often still not actually useful. The Real Use principle suggests that this is because requirements gathered before a system is built will be less accurate than those gathered once it is available for use. This has led to many modifications of the Waterfall Model including the Spiral Model and modified waterfalls with overlapping stages of requirements, design and coding (a good survey can be found in Chapter 7 of [11]). These modifications still follow distinct stages of requirements analysis, product design, technical design, etc., but there is now feedback and refinement in each stage.

Applying such staged methodologies, however, still generally results in a large investment of effort before any usable software is built. On an enterprise software project it can be a year or longer from initial requirements to usable software. This runs counter to the Real Use principle: it is difficult to justify extensive effort by a substantial size team without validation from real users using a production system. Rapid prototyping and rapid development approaches to software development [11] arose in order to address this problem by quickly and repeatedly validating that software meets the perceived needs. However, rapid development methodologies often do not put much emphasis on the Clear Statement principle, instead using reactive approaches where user feedback from prototypes and beta tests is incorporated into both the software design and implementation. Rapid development methodologies have been used quite successfully with the release of “beta” or “preview” versions of software to thousands, and in some cases millions, of users. However, a lack of attention to the Clear Statement principle can still be problematic, as such software is often unwieldy in size or can have many complex features that are confusing to users.

Some methodologies are explicitly aimed at balancing the competing principles of Clear Statement and Real Use. These approaches are often referred to as iterative or incremental delivery, where the idea is to build a small but key subset of the desired functionality, get that in front of users as quickly as possible, and then add to it. XP is an example of one such methodology, which we use as the basis of further discussion below. Regardless of the approach that one takes to software development, however, we strongly believe that it is critical to pay constant attention to the tension between the Clear Statement and Real Use principles, getting real use as quickly as possible, while always striving for clear and concise statements of the desired functionality.

It is important to note that not all iterative development practices address the tradeoff between the Clear Statement and Real Use principles. Only those practices that encourage incremental *delivery of usable functionality* through the rapid delivery of working software to users address this problem. Feedback loops in a staged approach, such as a Modified Waterfall Method, are generally not concerned with incremental delivery of usable functionality. Similarly, feedback loops in technical design and implementation (as advocated in RUP) do not incrementally provide new functionality that is validated by users.

In considering the problem of a late software project, we have introduced several principles and discussed how they can be used to understand possible practices for addressing that problem. We have also revealed competing principles that govern these practices. In general, we believe that the success of software projects can be improved by recognizing underlying conflicts such as these and by choosing practices that balance them in a way that is appropriate for the given project. In identifying such competing principles, we are also able to relate the practices advocated by different software development methodologies to one another. Many of these practices can be seen as addressing one side of a conflict or the other, which often accounts for the great differences between methodologies.

2 Principles of Software Development

Several software development methodologies attempt to articulate driving principles. In some cases, these principles are not fundamental laws or assumptions, and in these cases, we would not actually consider them to be principles. In other cases, the principles tend to describe the development methodology rather than the development of software itself. While principles about methodologies are useful, we believe that they provide less insight than principles that characterize fundamental issues about software development. Also, in being concerned with a particular methodology, such principles are often less useful for comparing the practices of different methodologies. Here we briefly consider the principles articulated by Extreme Programming, the Agile Methodologies and the Personal Software Process, in order to better illustrate what we do and do not consider to be general principles of software development.

In the case of XP [1], the “principles” actually describe actions or sets of actions, and so we consider them to be practices rather than principles:

1. Get rapid feedback.
2. Assume simplicity.
3. Incremental change.
4. Embrace change.
5. Do quality work.

There are several issues with treating these statements as what we have termed principles. First, they do not provide any insight into why they should be followed or to what conditions might make them applicable. To us the most useful principles reveal something about the characteristics of team interaction, software development, or the use of software. Second, each of the above statements describes certain properties that a practice might have or support, but in doing so they characterize *ways of working* rather than *fundamental properties* of software development. To us, ways of working are practices, not principles.

These statements are also sufficiently broad that they would not constitute individual practices, rather they suggest sets of practices. For instance, there are many practices that can be used to "get rapid feedback", including focus groups, mock-ups, prototypes and incremental delivery. Some of these work better than others. The principle of Real Use, introduced above, illustrates how underlying principles can be used to compare such practices and choose among them. Real Use suggests that better feedback is obtained from real users and a real system. Thus practices such as prototyping and incremental delivery will tend to produce better feedback, assuming that the time requirements of the different methods are similar. We believe it is important that principles provide such guidance in choosing among different practices, rather than simply characterizing properties of possible practices. While we generally agree that the above statements describe important characteristics of good development practices, we have not found them to be particularly useful in evaluating and comparing practices for our own software development work.

The book *Agile Software Development* [6] discusses seven principles that motivate the Agile Methodologies of software development:

1. Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.
2. Excess methodology weight is costly.
3. Larger teams need heavier methodologies.
4. Greater ceremony is appropriate for projects with greater criticality.
5. Increasing feedback and communication lowers the need for intermediate deliverables.
6. Discipline, skills, and understanding counter process, formality, and documentation.
7. Efficiency is expendable in non-bottleneck activities.

In contrast with the XP "principles," these principles attempt to capture some more fundamental aspects of software development. However, several of these principles tend to characterize software development *methodologies* rather than software development itself. In addition, some of the Agile principles are almost tautological and don't serve to broaden our understanding of software development problems. This is particularly true of the second, third and seventh of these principles. In contrast, the first, fourth and fifth of these principles do have many of the properties that we are looking for in terms of broadly applicable principles of software development (however they are more concerned with development methods than with development itself).

Our principles differ from these in that we attempt to directly characterize the issues and constraints facing a software development project, rather than characterizing different kinds of methodologies used in software development. For instance, the fourth principle relates the "weight" of a set of development practices to the criticality of the project. In contrast, we articulated the principle of System Dependability, which states that the confidence in the reliability of a software system should be proportional to the cost of its failure and the required repair. Greater ceremony is one way of achieving the necessary degree of confidence, but there are other ways of doing so, such as through more extensive testing or by choosing developers with certain skills. Thus we prefer to have principles characterize the underlying constraints on software development itself, rather than the methodologies used to develop software. We believe that such principles provide more value in enabling developers to relate a broad range of different practices to a given problem, and to pick the practices that are best for their project.

The Personal Software Process (PSP) [8] is based on the following principles:

1. Every engineer is different; to be most effective, engineers must plan their work and they must base their plans on their own personal data.
2. To consistently improve their performance, engineers must personally use well-defined and measured processes.
3. To produce quality products, engineers must feel personally responsible for the quality of their products. Superior products are not produced by mistake; engineers must strive to do quality work.
4. It costs less to find and fix defects earlier in a process than later.
5. It is more efficient to prevent defects than to find and fix them.
6. The right way is always the fastest and cheapest way to do a job.

These principles focus primarily on planning, measuring progress, and defect prevention and reduction. Much of the emphasis is on measurement, to provide data for planning, evaluation and improvement. As with many of the Agile principles, the first two of these principles are more about software development methodologies than about software development itself. The emphasis that these principles place on measurement, planning and defect reduction, without consideration of other aspects of software development, is in our opinion much of what drives the "scruffy" XP and Agile methodologies to present themselves as a reaction against the software engineering approach. This is in part because measurement and planning, while presented here as individual practices, are often (mis)used by management to assign blame for late or failing projects. It is also in part because many of the measurement tools advocated by PSP and other measurement-heavy methodologies have considerable overhead and are difficult to implement effectively in practice. We return to these issues in the section called Planning and Tracking Progress.

Having an explicit statement of principles is valuable, because it allows one to understand whether or not those principles agrees with ones own experience and goals for their project. For instance, the

Agile and PSP principles both allow one to readily understand the applicability of the corresponding methodology based on what the principles identify as important. In our experience the fifth of the PSP principles, which states that it is more efficient to prevent defects than to find and fix them, is often not the case. Rather, we have found that it is possible to spend considerable effort attempting to prevent defects with little or no marginal return. In such cases it might well be more effective to try building something and see what kinds of defects it has. We return to this issue later, as in our view one of the critical decisions for any development project is how to evaluate the relative cost of finding and fixing defects as opposed to working around them in order to ship the product. Alpha and beta tests are one way of setting this "knob" toward shipping a product even if it has defects. The other extreme is never shipping the product because of the possibility of one more defect.

In addition to having an explicit list of principles, we believe that it is important to be able to readily relate individual principles to specific problems and practices in software development. This enables developers to pick and choose particular practices for their problems, based on the understanding articulated in the principles. In contrast, the principles discussed in this section primarily apply to entire methodologies rather than specific problems and practices. To that end, we discuss principles in the context of problems and of practices that are commonly used to address those problems. We first turn to a discussion of some widely used software development practices and then present some principles related to these practices.

3 Software Development Practices

In this section we review the practices of Extreme Programming, and relate them to practices advocated by other common development methodologies such as the Waterfall Model [3], the Rational Unified Process (RUP) [13], and Rapid Application Development (RAD) approaches [11]. Our goal in this section is to consider a specific set of practices in enough detail to provide a shared basis for the discussion of some problems and principles. We have chosen XP for consideration here because it is clearly presented, it is relatively simple to describe (there are twelve concisely stated practices) and many of the practices are closely related to those of other iterative approaches to development.

The XP practices have received considerable attention lately, along with a broader set of so-called Agile Software Development Methodologies [6]. XP is the best known of these methodologies, all of which advocate incremental delivery (with short two- or three-week delivery cycles), extensive customer involvement, and face-to-face communication rather than large numbers of project-related documents. While we use the practices of XP to organize the discussion, we have a mixed opinion of how well XP applies to enterprise software development projects. Although we agree with most of the principles underlying the XP practices as well as with many of the practices themselves, we find the high degree of dependence on verbal communication to be a drawback of XP. For those interested in extending XP for enterprise software development, we also recommend the writings of Martin Fowler [7].

The book *Rapid Application Development* [11] describes three iterative approaches to software development: Evolutionary Prototyping, Staged Delivery, and Evolutionary Delivery. These differ in the degree of specification that is done prior to development. In Evolutionary Prototyping, new or changed features are specified for each development cycle, and once the customer is satisfied the iterations cease and the result is the final product. In Staged Delivery, all the features are specified before development starts, much as in a traditional Waterfall approach, but the features are delivered incrementally. The Evolutionary Delivery approach is a hybrid of the two, where some up-front specification is done, but additional features or changes may be specified for each successive development cycle.

XP has been characterized by some people as being an Evolutionary Prototyping approach. We believe this is incorrect and see XP as Evolutionary Delivery, because the planning game need not be applied separately to each iteration. In fact, the book *Planning Extreme Programming* [2] suggests planning both at the level of individual two-week cycles and at the level of larger releases (a few months or more). These longer term plans are less precise and will not have all their stories created yet, nonetheless the process is not simply based on customer reactions to each in a succession of prototypes (as in Evolutionary Prototyping).

3.1 The Twelve XP Practices

We now list the XP practices, all of which fit our definition of practice. XP advocates twelve practices, and encourages the use of all twelve in conjunction with one another. We briefly summarize each XP practice below:

1. **Planning Game** Develop "stories" that describe what the software should do. Stories should be concise – fitting on a single card – and clear. The customer determines the scope, priority, and dates, while technical people estimate what can be done in the allotted time.

2. **Small Releases** Put a simple system into production quickly. Release new versions on a very short (two week) cycle.

3. **On-Site Customer** Have a real, live user on the team full-time to use the system as it evolves, help set priorities and answer questions.

4. **Metaphor** Guide all development with a simple, shared theory of how the system works.

5. **Refactoring** Restructure the system without changing its functionality (behavior), to remove duplication, improve clarity, simplify, or add flexibility.

6. **Simple Design** Design as simply as possible at any given moment. This applies both to product design and to technical design.

7. **Continuous Integration** Integrate and build the system many times a day, every time a task is finished.

8. **Continual Testing** Write unit tests which must run flawlessly; customers specify tests to demonstrate functionality is finished. Use continual regression testing to ensure no inadvertent loss of functionality as a result of changes.

9. **Pair Programming** Write all production code using two programmers at one machine.

10. **Coding Standards** Use common standards throughout the code to emphasize communication.

11. **Collective Ownership** Encourage anyone to improve code anywhere in the system.

12. **Sustainable Pace** (Formerly called 40 Hour Workweek). Work at a pace that can be maintained; avoid working weekends and excessively long days as a regular practice.

3.2 Relation to Other Practices

Most of the XP practices are related to practices of other methodologies, particularly methodologies that advocate iterative approaches to development such as the Rational Unified Process (RUP) [13], and various Rapid Application Development (RAD) methodologies [11].

1. **Planning Game** All software methodologies involve planning. The best development practices define product features and priorities based on needs identified by business people, and set the schedule based on difficulty estimates provided by technical people. The XP approach to planning is lightweight compared to most other methodologies, and is largely based on verbal communication. At the other end of the spectrum, the Personal and Team Software Processes (PSP and TSP) advocate extensive written planning and tracking of progress.

2. **Small Releases** Iterative approaches to software development are coming to be recognized as best practices (e.g. RUP [13], RAD [11]). An important aspect of XP that is not shared by all iterative methodologies is the incremental delivery of new functionality, both for tracking progress and for validating the software.

3. **On-Site Customer** Most rapid development methodologies encourage extensive interaction with customers or their representatives (e.g. Joint Application Development or JAD [11]), but this interaction is usually not continuous as it is with XP.

4. **Metaphor** Some other software methodologies recommend developing a high-level vision of how the product works, both as a guide to feature selection and as a guide for system architecture (e.g. RUP [13]).

5. **Refactoring** Evolutionary approaches to technical design and architecture have long been advocated by some development methodologies (as by members of the Lisp and Smalltalk communities), whereas other methodologies advocate doing technical design and implementation correctly the first time [3] [8].

6. **Simple Design** The practice of simple design is not generally feasible for methodologies that involve doing the design just once, because it is important to try to design for changing requirements in such approaches. For iterative methodologies, such as XP or RUP, simple design is a good practice because the architecture can always be changed to reflect new requirements when they arise.

7. **Continuous Integration** Most iterative software development methodologies such as RUP and RAD approaches advocate frequent integration, although generally not as frequently as XP which advocates that it be done as each programming task is completed and at least once a day.

8. **Continual Testing** Most methodologies advocate some form of continual testing, such as automated smoke tests. However there are considerable differences in terms of how different

methodologies balance developer-written unit tests, as favored by XP, versus functional tests, integration tests, smoke tests, and tests written by a separate QA group.

9. **Pair Programming** This practice is advocated primarily by XP.

10. **Coding Standards** Most methodologies encourage using common coding standards, although there are substantial differences in how they recommend getting participation from developers. PSP and TSP provide considerable focus on arriving at coding standards and having means of enforcing them.

11. **Collective Ownership** The practice of "egoless programming" is recommended by a number of methodologies, although collective ownership is more extreme because it advocates allowing any developer to change any piece of code at any time.

12. **Sustainable Pace** (Formerly called 40 Hour Workweek). A number of methodologies note that excessive overtime is generally leads to high turnover, but do not necessarily make limiting hours a key practice.

4 Problems, Practices and Principles

We now turn to a discussion of some problems and principles that relate to the twelve software development practices described in the previous section. As in the introduction, we consider specific problems together with principles and practices that relate to each problem. We do not attempt to exhaustively catalog either the problems or the principles that are related to these practices. Rather, we choose a few problems that arise in a wide range of software development projects, and discuss practices and principles related to each of those problems. As we saw in the introduction there are often several competing principles related to a given problem; we thus also focus on the tension between underlying principles.

In our discussion, we group the practices according to the problems that they address, and consider some principles related to each problem. We consider problems in four areas: (i) defining what the software should do (specification of the software), (ii) planning and tracking the progress of development, (iii) technical design of the software, and (iv) building the software. While testing might seem to be another natural problem area to consider, we see testing as a recurring theme that arises in all of these areas.

4.1 Defining What the Software Should Do (Specification)

The incredible range of needs that can be met with software often makes it hard to determine what a given software product should do. It can also be difficult to ensure that all the parties involved share the same understanding of what a product will do. Identifying the purpose of a software product is problematic for nearly every software development project, potentially resulting in substantial effort being spent to build software that ends up being of little use.

As we discussed in the Introduction, the principles of Clear Statement and Real Use are both critical to defining a software product. It is further important to consider what kind of software is being developed in order to understand how these principles apply. For instance, for some software products the functionality is simple to describe whereas for others it is not. As an example, a file compression tool such as WinZip is easy to explain from a functional perspective (even though its internal processing is highly complex), but in contrast an accounting system has a complex functional description (as well as complex internal processing). The implications of the Clear Statement and Real Use principles are quite different in these two cases. In the former case, a clear statement is easy to develop a priori, and Real Use is most important for testing that the software behaves as intended. In the latter case, creating a Clear Statement involves continual user feedback during the design and development of the product, and Real Use is thus important throughout the process.

We now turn to two specific problems of software specification, focusing primarily on the case of a product with complex functionality that is relatively difficult to describe.

Problem: The software is defined in a way that does not accurately capture the needs of users. (The specification of the product is inaccurate.)

The XP practices used to address the problem of inaccurate specification are On-Site Customer and Small Releases. They require new releases of a system to be provided with high frequency and a local domain expert to quickly give feedback on the added functionality. This combination of practices can be a powerful approach to quickly developing software when there is a good understanding of what is being built. These practices reflect the following principles:

Principle of Real Use Understanding the needs of users, and validating that those needs have been met, is best done with a real implementation of the software and real users [repeated from the introduction].

Principle of Domain Expertise Understanding the domain is critical to discovering, interpreting and explaining user requirements.

Principle of Early Validation Validating a product definition early in the project generally lowers the overall project cost.

Principle of Deployment Cost The cost of upgrading or replacing a piece of software is proportional to the degree to which the software is integrated with, or critical to, a customer's work process.

The principles of Real Use, Early Validation and Deployment Cost also provide conflicting viewpoints. When the deployment cost is high, it is difficult to justify Real Use, especially early in the development process. This is particularly true for systems that have hardware as well as software components. For instance, applying the Early Use principle to the development of avionics software for a new aircraft might require having a physical aircraft available early in the software development process, often an impossibility. Enterprise software systems also often have high deployment cost, due to data migration, integration and training issues.

The practice of On-site Customer is a valuable way of addressing high deployment cost for many kinds of software, by having a "local deployment" for the on-site users, rather than deploying to external customers. However, over time, on-site customers tend to become more part of the development team than of the user base, making them less representative of actual users in the field. Thus it is still important to deploy software regularly to actual users, though this can be less frequently than the two or three week delivery cycles advocated by the Small Releases practice. To be most effective we believe that the Small Releases practice requires not only knowledgeable on-site users but also regular releases to users in the field (at most every few months).

The principle of Domain Expertise states that the best software is developed by a team that is knowledgeable about the domain. The practice of having On-site Customers is both a good way of providing that domain expertise and a good way of transferring this knowledge to other members of the team, who learn from the On-site Customer. For instance, developing good software generally requires an understanding of common workflows so that the software naturally supports patterns of actions that users will take; supporting the workflows or common usage can be as important as providing the functionality. Usage and workflows are incredibly hard, if not impossible, to understand without a working product and without people who try to use that product as part of their own work (rather than using it to validate requirements or to test the product). To the extent that On-site Customers are able to do this, they can be incredibly valuable.

The principle of Early Validation conflicts with the principle of Real Use because it advocates validation regardless of how that validation is obtained. For instance, some software development methodologies encourage the use of system mock-ups, focus groups, and other means of getting early feedback from users or potential users. In our experience such early feedback mechanisms provide limited value because of the difficulty of getting accurate information about workflows without an actual working system. However, they can be useful for gaining some insight early in the product development cycle. XP takes the most extreme view of the Real Use principle, and obtains validation only from a working system (although initially with just a subset of the eventual functionality).

Problem: The software specification keeps changing, so software is not developed to a specification.

This problem manifests itself in two distinct ways, and many software projects suffer from one of the two instances. In the first case, because the implementation is gated on the specification, no software is actually written. This instance is more common among groups that practice the traditional Waterfall Model, where there is such an emphasis on up-front specification that development indefinitely remains part of the future. In the second case, an implementation is completed but without having a good description of what the software does, making testing and documentation difficult, if not impossible. This case can occur when using iterative prototyping methodologies, as well as non-methodologies such as code-and-fix, that do not focus on developing a clear understanding of what the product should do.

The XP practices that address the problems of software specification are the Planning Game, On-Site Customer, Small Releases and Continual Testing. The Planning Game is concerned with creating user-stories, which are clear and concise descriptions of functionality that is important to the user. One key aspect of the Planning Game is that each story is a concise description of the desired functionality that is clear to both the On-site Customer and to the developers (a story is intended to fit on an index card, although larger cards are often used). Another important part of the Planning Game is that the On-site Customer is the ultimate arbiter of whether a story and its implementation in software meet their needs. In particular, the On-site Customer is responsible for creating acceptance tests demonstrating that each Small Release functions as the stories intended. These tests serve the additional role of specifying the functionality of the system.

While XP and the other Agile Methodologies advocate producing little in the way of specification documents, in our opinion too little for most projects, their practices are aimed at developing a clear understanding of what the software should do. The Agile Methodologies simply takes the view that specification should be very light-weight, with a focus on clarity over detail. The principles underlying these practices are:

Principle of Changing Requirements Requirements change, both because the understanding of the needs of users has changed and because the needs themselves have changed.

Principle of Clear Statement (Revised) A clear and concise statement of user needs *and the corresponding functionality of the software* generally results in the development of better software.

Principle of Specification Cost There is a high cost to writing and maintaining detailed specification documents that are accurate and effectively convey understanding.

In addition, the principles of Real Use and Domain Expertise, stated earlier, underlie these practices. XP lives up to the name "Extreme" in its approach to the principle of Specification Cost. Rather than paying any cost for writing and maintaining specification documents, XP advocates writing only brief stories to describe the features being implemented, together with acceptance tests. The risk accepted in this approach is that such short descriptions of functionality afford more distinct interpretations than would be possible with more detailed written descriptions. For relatively simple features, the customer and the developer may develop a shared understanding through a verbal discussion and use the written story as a reminder of that discussion. However, for more complicated features, or for large sets of closely related features, the stories may not ensure a common understanding or consistent description of the product. This leads to two principles that are not accepted, or at least are not deemed important, by XP.

Principle of Unreliable Memory Personal memory is a poor substitute for a written document.

Principle of Adequate Specificity When customer needs admit many interpretations, precise descriptions of the software help ensure usability and quality.

These two principles run counter to the principles of Specification Cost and Changing Requirements, yielding another important case of competing principles in software development. Compared to XP, other software development methodologies take a very different view of the tradeoffs involved. Many methodologies advocate the creation of multiple documents describing what the software should do, such as Market Requirements (MRD), High Level Design (HLD), Functional Specification, User Interface Specification, Technical Specification, Relationship Diagrams, or Sequence Diagrams. Such documents are not only advocated by traditional methodologies such as the Waterfall Model, but are also advocated by iterative methodologies such as RUP, which is a highly document-oriented process. While all of these specification documents address problems associated with the principles of Unreliable Memory and Adequate Specificity, it is important to understand when such documents are necessary versus when they get in the way of developing a clear understanding and responding to changing requirements.

Different software projects need to balance the conflict in these principles in different ways. If a project involves complex functionality, relying on peoples' memory and having little written description, as advocated by XP, will be problematic. On the other hand, for many projects, highly detailed specification documents can cause more problems than they solve. The detail can cause loss of clarity, and the sheer difficulty of changing the documents can make the project unresponsive to important changes.

The fact that XP uses simple stories rather than highly detailed specifications has led some critics to lump XP in with code-and-fix approaches to software development (those in which software is built before developing a clear understanding of what the software should do). This is not necessarily a reasonable characterization of XP, as the stories coupled with On-site Customers can provide a good understanding of what the software is intended to do. XP's strong recognition that the principle of Changing Requirements acknowledges that this understanding may change substantially once initial versions of the software are built. Its practices are designed to address these changes through new and changed stories. XP also takes a strong view on minimizing the cost of writing and maintaining specification documents.

In our own work we have found it necessary to create more design documents than advocated by XP, paying more attention to the principles of Unreliable Memory and Adequate Specificity. However, in doing so we have tried to carefully respect the three competing principles of Changing Requirements, Clear Statement and Specification Cost because we believe that many software projects fail by not paying enough attention to one or more of these. We have found it important to write specifications but equally important to keep them as concise as possible, always keeping the goals of clarity and flexibility in mind.

4.2 Planning and Tracking Progress

In the introduction we considered the problem of a late project, something that occurs all too often in software development. Many software development methodologies address how to plan and measure progress, in an effort to avoid late projects. Even so, a large percentage of software projects are still late, either because they do not apply the practice advocated in such methodologies, or because the practices do not work for their projects.

Problem: There is poor visibility into how long it will take to complete the software (or the next release of the software).

Incremental delivery practices as advocated by XP and some RAD approaches (see [11]) are a common way of addressing this problem. In XP, the practices of Small Releases and the Planning Game are used to address schedule visibility. Releases are planned around new functionality, where each release implements additional stories. By providing new functionality every two to three weeks, these practices provide a means of measuring progress in terms of completed functionality.

Planning in XP is done relatively quickly and frequently. Each two- or three-week development cycle is generally planned in at most a day, and involves many if not all developers together with the On-site Customer [2]. Developers estimate how long each story will take to implement, and stories are chosen based on these estimates together with customer provided priorities. Estimates are made by comparing a story with other similar stories that have been implemented previously. Several developers participate in the estimation process. If they disagree about how long something will take, then the most optimistic estimate is used. (The goal of using the most optimistic estimate is to get all developers to be realistic in their estimates. However, early in a project this may lead to estimates that are substantially over-optimistic.) Once a story is completed and verified by an on-site customer, the actual time taken to implement the story is recorded and is used to help produce more accurate estimates for future stories.

Incremental delivery practices reflect the following principles:

Principle of Usable Progress Usable functionality is the most accurate way of measuring progress on a software project. The degree of certainty in progress estimates is proportional to the cost of producing those estimates.

Principle of Group Estimates The most accurate estimates are created by a group that includes both the developers who will do the work and those with a broader architectural perspective.

XP again lives up to the name "Extreme" in its planning and tracking practices. For planning, XP takes the view that the cost of obtaining accurate estimates is too high to be worth the cost, which is reflected in the fact that the Planning Game advocates spending very little time on estimation and scheduling. For tracking progress, XP advocates using tested functionality, via the practice of Small Releases, as the only measure of progress on a project. This is the most extreme view of the principle of Usable Progress, taking usable functionality to be the only measure of progress.

Other approaches to software development take a different view of the tradeoffs underlying estimation. For instance, processes like the Wideband Delphi method [4] are based on the premise that it is possible to produce reasonably accurate estimates by combining the estimates from many developers. In Wideband Delphi, individual developers produce independent estimates, and these estimates are compared and refined in search of a consensus for the group. This practice also advocates breaking down tasks to a level of detail where each task is a developer-day or less in duration. It treats individual developers as noisy sources and requires that differences in estimates are resolved by sharing understanding and proposed designs. Furthermore, Wideband Delphi views the uncertainty as lower if the subtasks are more detailed. This and other similar methods have the advantage of spreading knowledge around the group and increasing awareness and transparency into other previously unfamiliar parts of the project.

In the case of scheduling and tracking progress, XP uses Small Releases as the main mechanism and does little to address adjusting schedules or correcting estimates until after a two- or three-week development cycle is complete. Most other development methodologies explicitly address problems of misestimation and schedule adjustment by understanding and tracking dependencies between tasks and by understanding how other features may be shifted or swapped to continue to meet a deadline. XP advocates assessing the accuracy of estimates, but by recording how long a task took to complete and comparing that to the estimated time, rather than tracking ongoing progress or rescheduling tasks that are late. In estimating a new task, the task is compared to similar previously completed tasks, using the actual time that those tasks took. Such an approach is only practical if there is a high degree of independence between tasks, such that one late task does not delay many or all the other tasks.

For tracking progress, many development methodologies advocate the use of various other metrics for measuring progress. Such measurement practices, however, often mistake easily sampled, "objective" quantities for accurate measures. Measuring lines-of-code (e.g., [4]) or function points (e.g., [9]), for instance, correlates progress with measures that have a somewhat arbitrary relation to effort or difficulty. With such measures, "90% complete" can mean that anywhere between 5%

and 95% of the work actually remains to be done. By analogy, one might measure the progress of a skyscraper by its current height, neglecting the status of its internal structure. Other methodologies make individual developers responsible for reporting the progress of current tasks. In our experience, however, a developer's own estimate of task completion is often no more accurate than any other measure and should also be used with caution. Over time we have become firm believers in the fact that incremental completion and validation of functionality are critical to maintaining accurate schedules.

Some critics maintain that frequent delivery of new features is not always practical, because it makes it impossible to write software with features that take a long time to build. Such a view is contradictory to our experience building highly complex software, where we break down larger features so that some functionality can be delivered every couple of weeks. It may take months to fully develop complex new functionality, but we have found that in general it is possible to break it down in to smaller pieces. Delivering functionality incrementally, at least to "internal" users, provides good visibility into progress on larger features. This can be rewarding both for developers, who see their work realized more quickly, and to users and sponsors, who get to experience the new functionality as it emerges.

4.3 Designing the Software

In general, software does not directly implement the functionality of a system. Instead, it implements a set of abstractions or theories that provide the desired functionality. This is often referred to as the technical design or architecture of the software. A simple analogy is to a building, where the design includes certain internal structures for plumbing and other utilities. An abstraction in designing a building might be that the bathrooms are in the same location on each floor to simplify the routing of plumbing. Once such an abstraction is in place, breaking it is relatively expensive because additional infrastructure is required to locate a bathroom where it was not expected. Similarly, software has internal structures that enable functionality to be created less expensively. Unlike the architecture of a building, these structures are not constrained by any physical laws. Because they are more abstract, they tend to be more difficult to express to a nontechnical person than structures of a physical building. While software architecture is important to creating clean and maintainable code, some groups place too much emphasis on it, leading to the following problem:

Problem: There are ongoing design discussions within the development team about architecture, but the system is not actually getting built.

This is a common problem faced by development teams, particularly early in a large project. Sometimes the whole project is canceled before it ever gets beyond the technical design stage. This problem is exacerbated by methodologies that encourage doing all the design at the beginning of the project, rather than allowing the design to evolve over time. Methodologies that further advocate completing the design before writing any code, such as the classic Waterfall model, tend to make the problem even worse. In such circumstances it is natural to spend a lot of time trying to perfect the architecture, because the goal is to get things right before starting the implementation. In particular, it is common to try designing a highly flexible architecture that will be able to handle future requirements as they arise. In our experience, the most intractable architectural debates arise when one is attempting to provide flexibility for requirements that do not yet exist, because there are no objective means of assessing and comparing different designs.

Technical design documents can be highly useful in the design process. Written documents both allow a larger group of people to take part and offer criticism, and create a permanent record of the discussion for later reference. Written arguments tend to be more formal or at least more precise than verbal discussion, and through several iterations of feedback and revision, they tend to converge on an understanding shared by the group. By contrast, verbal discussion tends to converge only for a small subset of the participants, or not to converge at all. This reflects the principles of Unreliable Memory and Adequate Specificity for technical designs (which were discussed previously for functional specifications). However, written technical design documents can also become part of the problem when they are iterated endlessly, without converging on a design that can be implemented.

XP addresses the problem of never-ending design by taking an evolutionary approach to software architecture: the architecture is changed when necessary, in order to better support new or changed functionality. Similar evolutionary approaches to system design, involving building and re-abstracting code, have long been advocated in the Lisp and Smalltalk programming communities. Moreover, other commonly used iterative methodologies such as RUP and RAD also encourage iterative change to the technical design. There are two XP practices that reflect this approach. The first of these practices is Simple Design, which stipulates using the simplest possible architecture that will meet the business requirements. A key goal of doing the simplest possible design is to limit the architecture to supporting the currently articulated needs of the user, not attempting to foresee future needs. Simple Design strongly discourages doing any design or coding to provide flexibility for future and as yet unspecified requirements.

The second practice in the evolutionary architecture approach is Refactoring, which advocates redesigning and rewriting code so as to capture new commonality or structure without changing the functionality. Refactoring is done when code becomes overly complex due to the addition of new features or when nearly duplicated code arises in different parts of the system. If the simplest design results in overly complex or duplicated code, then the practice of Refactoring is used to address that. The idea is to separate designing for functionality, which should always use the simplest approach, and designing for code stability and maintainability, which can develop higher-level abstractions but based only on those features that have already been implemented (again, not potential, future features).

There are two competing principles that underlie the practice of evolutionary architecture in general and the practices of Simple Design and Refactoring in particular:

Principle of Build With a Design Software is generally more reliable, easier to maintain, and easier to extend when it follows a clear technical architecture.

Principle of Build to Understand The architecture of a software system can invariably be improved once the system has been implemented, tested, and used.

The principles of Build With a Design and Build to Understand suggest conflicting practices. The first says that software should have an architecture and not simply be built without having underlying abstractions identified, just as a physical building should not be built without planned infrastructure and architecture. The second says that the best architectures generally arise after one has already built a system, because the problems are then better understood. The evolutionary approach to architecture, advocated by the XP practices of Simple Design and Refactoring, is one way of addressing with this conflict. The architecture is improved where needed, and old code is rewritten using the new architecture. This is repeated throughout the development process whenever the architecture fails to meet the needs of new or changing requirements.

We believe that it is important to recognize that evolutionary design does not mean building without a design. Several critics, in our view mistakenly, state that XP advocates a code-and-fix approach to building without designing. A simple design should not be confused with a lack of design. In fact, in our experience, people who do not believe in simple design are often more interested in building elaborate and difficult to maintain "general and flexible architectures" than they are in delivering useful functionality to users. We believe it is important to realize that complex and un-maintainable code can come from two very different sources. In one case, it results from building without a design, but it also results from building with an overly complex and overly general design.

Evolutionary design exploits the fact that with software, building with a design does not necessarily require completing the design *before* building. While it is important to follow a design when building software, that design can evolve as the understanding of it evolves. It is worth noting how different this evolutionary design approach is from the design of most non-software artifacts. For example, imagine if it were possible to "refactor" a luxury office building to handle twice as many tenants without feeling cramped (adding floors would probably run afoul of zoning laws as well as exceed what the foundation could handle). This is precisely what evolutionary design is about.

The ability to come back and change the architecture when it really needs changing can usually help make developers less nervous about choosing a simple design. However, in applying evolutionary design it is important that development managers ensure that the scheduling process carefully considers what needs to be re-designed and allocates the time to do so. Doing Simple Design but not coming back to do Refactoring will eventually lead to a system that is indistinguishable from the "spaghetti code" that results from code-and-fix approaches.

Sometimes changing the technical design takes considerable time, or introduces substantial temporary instability into code. If an architectural change is big enough, it can be necessary to work on it in parallel with developing new features, or to implement it in stages so that it does not stop the development and delivery of new features. However, in our experience large architectural changes can be handled in this manner and still allow new features to be implemented on short development cycles. Moreover, we have found evolutionary design to work much better than submitting to the temptation to create the "perfect architecture" the first time around.

In addition to the above principles, the XP practice of Refactoring also reflects the principle of Changing Requirements. No matter how appropriate an architecture is, some new user needs will invariably not fit within that architecture. This is particularly true for the initial architecture of a new system, where the system has not yet been used and user requirements have in all likelihood not been well captured. By encouraging Refactoring, XP highlights the fact that changing requirements will sometimes require changes in the architecture, rather than encouraging one to stick with an architecture that has outlived its usefulness. In contrast, design-first approaches are poor at handling Changing Requirements, because designs can often be rendered untenable by changes in requirements. This can be a major source of morale problems for the development team if they have followed a methodology where a lot of effort was expended on the design, only to have the requirements change. Despite these problems, design-first approaches still seem to be advocated by many software methodologies and are attempted by many development teams.

Design patterns are another practice commonly used to simplify the design process. Design patterns are well-known templates for solving certain common technical problems. Many practitioners of XP, RUP or RAD approaches make use of design patterns. While we believe that design patterns serve a useful role as catalogs of possible architectural approaches, we grow concerned when applying patterns becomes a substitute for thinking about and understanding the problems being solved. Design patterns can be valuable in helping to apply Build With a Design. However, they can also be used to simply "build what someone said was good", without understanding how well that design serves the problem at hand. This can be problematic, both because the resulting code does not do what is actually needed, and because a particular common pattern may unnecessarily limit the design space or result in code that is actually more complicated than necessary. Finally, we want to emphasize that using patterns does not absolve developers of the responsibility to understand what they are doing. Not only can patterns be mis-used, the patterns themselves can simply be wrong.[2]

A second major architectural problem that occurs in software development projects, is that the time required to make changes to the software is often unrelated to how large the changes seem to be from a business or user perspective. A good software architecture is one that allows small changes to the functionality to be accomplished via small changes to the code, in effect the architecture mirrors how people interact with the system. However, this is often not the case, as reflected in the following problem:

Problem: Small changes to the software take a long time to implement.

The XP practice of Metaphor is used to address the problem of disproportionate cost of change. The idea of a system Metaphor is to have a guiding analogy or description that provides a high-level understanding of what a system does and how it works. For instance, a message-oriented middleware system might be characterized by the metaphor of a bonded courier: it is used to deliver important things that cannot be lost. Finding good metaphors can be difficult, but they are often useful in helping guide the architecture such that it effectively mirrors the functionality of the system. For a large system there may be several metaphors, and the more broadly they characterize the entire system the more useful they tend to be.

The practice of Refactoring is also important in addressing the problem of disproportionate cost of change. XP advocates what is referred to as "relentless" refactoring, constantly looking for places where the architecture does not allow the required functionality to be delivered using simple, non-duplicated code. One goal of this kind of approach is to continually refine the architecture as requirements change, so that the design always does a good job of mirroring the functional requirements. In that way, changes that appear small to the user will in general not require large changes to the code. In other words, by paying the overhead of constantly updating the architecture, one attempts to avoid the case of small changes taking a large amount of time. However, with the added overhead, the effect is that small changes now have a medium overall cost. Another advantage

[2] For example the so-called "Double Checked Locking" idiom or pattern for multi-threaded access to a singleton object in Java has been widely advocated but has been shown to be incorrect. It ignores import aspects of the Java memory model and causes intermittent bugs that are incredibly difficult to find.

of constant refactoring is that it can be used to help balance the development schedule, by making the architectural changes at times when certain developers do not have as much to do in terms of delivering new or changed functionality.

The problems of never-ending design and disproportionate cost of change reflect the following additional principles:

Principle of Domain Expertise (Revised) The best way to discover, interpret and explain user requirements, *and architect the software*, is to understand the domain.

Principle of Design Tension Technical designs created by an individual tend to be more self-consistent and clearer than those created by a group. However, designs created by a group have a better chance of meeting a broad range of needs.

The principle of Domain Expertise is reflected in two XP practices. First, a good metaphor captures the understanding of domain experts, and thus the XP practice of Metaphor brings domain expertise to the technical design process. Second, the practice of having an on-site customer brings domain expertise to the technical design process, by having a local expert involved in the design and testing of the product.

The principle of Design Tension is not really addressed by the XP practices, as they provide little in the way of guidance for balancing individual and group design. One practice that is advocated by many development methodologies, and that we have found useful in our own work, is that of holding design reviews. We have found that the best reviewers of a design are those that will have to use it. In particular, we are strong advocates of explicitly documenting the internal interfaces between modules within a system, and then having those who will use a given interface be reviewers of it. The perspective of developers creating an interface is generally very different from those using it, and it is important to catch such problems at the design stage. Such reviews both serve an educational role for the reviewers and provide feedback to the designer(s) from the “user” perspective.

The XP practice of Continuous Integration also reflects the principle of Design Tension. While on the surface this practice appears most relevant to building software, it also plays a role in evolutionary software architecture. Difficulties in integrating parts of the system, particularly when integration is done frequently, often point to limitations of the design as much as problems with the implementation. Thus difficulties in integration can serve as a good means of determining when a technical design is straining under the load of new requirements and functionality and a candidate for refactoring. We further discuss the practice of Continuous Integration when we consider problems related to software implementation.

4.4 Building the Software

Once a design has been agreed upon, the software must be realized in a way that is consistent with that design. The aspects of software development described above – requirements gathering, specification, scheduling and technical design – have all been aimed at giving the programmer the best possible chance at success. In the end, however, the software must still be constructed, and no matter what specifications and designs have been created, mistakes will be made during this construction. In this section, we focus on problems that occur in the actual building of software and on practices intended both to prevent mistakes from being made and to quickly find and correct errors once they have occurred.

In providing a clear and comprehensive architectural description, the technical design of most software will be (by nature) abstract. Part of the role of a software developer is to make this design concrete by realizing it in code. In rendering the design concrete, developers may make mistakes either in interpreting the design or in writing the code itself. We consider problems of misinterpretation versus mis-implementation separately, as the sets of practices used to address them are often disjoint.

By a "mistaken" interpretation we mean the case in which different developers have arrived at different understandings of the same design. Each of these interpretations may in fact be self-consistent when considered separately, and the same may be true of the code and tests based upon these disparate understandings. However when the different interpretations, and the resulting code, are brought together the system does not work. Thus the effects of such mistakes are often not realized until code is integrated with parts of the system written by other developers.

Problem: Code works when "unit tested" but not when integrated into a complete system.

In some cases, as discussed in the previous section, this problem is due to shortcomings in the architecture itself rather than to inconsistent interpretations of that architecture. Here we consider only problems, practices and principles related to the developers' *understanding* of the architecture.

Several common practices are used to address the problem of code that works in unit testing but does not work when integrated with the rest of the system. One set of practices involves investing the time and resources to carefully understand and validate the design, via design reviews and discussions. Such practices are generally done before coding is started, with the goal of catching inconsistent interpretations before the fact. Methodologies based on the Waterfall Model take this kind of approach, with extensive technical documentation and review. In practice, however, considerable understanding is also gained during the process of writing the code, so it is also useful to have design reviews and discussions after coding starts. Thus we believe it is important to review the design as the implementation progresses.

A second practice commonly used to address the problem of code that does not work with the rest of the system is to do frequent integration, so as to limit the amount of code written before finding the flaw. Most iterative approaches to software development advocate this latter kind of approach, viewing integration as an ongoing part of development rather than a phase at the end. We believe that frequent integration and ongoing design review are critical to any large software project. In our experience, this sort of evolutionary approach, as advocated by the principle of Build to Understand, is the key to building something which is both robust under current use and flexible enough to adapt to future needs.

Some software development methodologies combine both ongoing design reviews and frequent integration. XP explicitly advocates early and frequent integration via the practice of Continuous Integration, where code is incorporated into a complete running version of the system as each task is finished. While XP does not advocate written technical designs or thorough design reviews, it does maintain that all developers should be aware of and able to correct problems anywhere in the product (through the practice of Collective Ownership). By encouraging every developer to look at all of the code, there a good chance that one developer will catch the erroneous assumptions made by another. One drawback of Collective Ownership is that it does not scale very well. If a project has 20 developers and half a million lines of code, it is no longer practical to be aware of things anywhere in the product. Some more structured responsibilities are important in such cases, but practices that involve all developers take broader responsibility for the system than just the code that "they wrote" are important for successful software development.

The XP practice of Pair Programming is also (to some degree) applicable here. Compared with explicit design reviews, however, Pair Programming is less useful for addressing misunderstandings of the design, because both programmers in the pair are looking at the architecture from the same point of view. Many misunderstandings or differences in interpretation come from developers writing code in *different* parts of the system, each from a different side of an interface. This makes design reviews useful even with practices like Pair Programming.

These practices for addressing the problem of code that fails to work when integrated acknowledge the fact that people are capable of making mistakes in interpretation and look to prevent or correct them. These practices are reflections of the following principles.

Principle of Broad Scrutiny The larger the number of knowledgeable developers who use, understand, and extend the same code, the more likely it is that hidden assumptions, differing interpretations and duplication will be discovered.

Principle of Imperfect Simulation Running the actual product code is key to understanding its output: there is no comparable simulation of the behavior of either an individual component or the product as a whole.

Principle of Old Bugs Die Hard Flaws in technical design and implementation tend to be more more ingrained and harder to fix the longer ago they were introduced.

It is important to note that the principle of Broad Scrutiny does not necessarily imply either that the entire development team should be directly responsible for authoring each line of code or that anyone should change code in any part of the product without discussion. After all, the person suggesting a change may be the one who is confused. Rather, this principle implies that more people should be involved in the coding process at a level of detail that they can suggest changes, and in some cases, make them. The specific practices being followed will dictate if or how those changes are made.

The XP practice of Collective Ownership follows from the principle of Broad Scrutiny, encouraging everyone to change and improve code anywhere in the system. Collective Ownership takes Broad Scrutiny to an extreme, because it suggests that anyone is able to make changes anywhere and at any time. In our view this is not practical and can lead to a chaotic development process. While confusion or misunderstanding surrounding a piece of code is an indication that it should be simplified or better commented, encouraging developers to simply make changes where the meaning of the code is not clear is not an effective way of improving the situation.

It is worth considering the parallels between the principles of Broad Scrutiny and Design Tension. Broad Scrutiny applies to the development of code whereas Design Tension applies to the creation of technical designs, however both are concerned with the relation between the contributions of individuals to those of the group. As just discussed, we believe that encouraging developers to make changes anywhere in the code is detrimental to the stability of the project. Nonetheless, we strongly believe that a large group of developers can simultaneously work toward understanding and improving code through a variety of practices such as code and design reviews, and team structures where more than one person works on the same code. In the case of design and architecture, as opposed to writing code, there are also both positive and negative aspects to the participation of a larger group. A larger group tends to make the architecture less coherent, but as with code development, reduces the risk that the architecture overlooks something that will later cause problems.

The principle of Broad Scrutiny is also one of the major reasons for the success of open source software. By making the source code available to a large group of interested people, projects which use the open source model enlist the help of, and solicit improvements from, the software users. Most open source projects, however, do not allow any user to make changes to the code. There is a relatively small group of maintainers who are responsible for reviewing the changes suggested by others. The maintainers ensure that the changes are both correct and consistent with the overall design of the project. In instances where the maintainers are not responsive in including suggested changes, the project may split and a new set of maintainers may become responsible for a separate version of the code.

The practice of frequent integration, encouraged by many iterative methods including XP, is an application of both the principles of Imperfect Simulation and Old Bugs Die Hard. Thorough design documentation and review may catch many differences in interpretation, but some flaws may not be discovered until the offending components are run in the same environment. In the end, any human reader is only simulating the behavior of the machine that will run the code; the principle of Imperfect Simulation implies that this simulation is likely to be flawed.

The principle of Imperfect Simulation also applies to test harnesses. Because a test harness only reflects its author's understanding of the rest of the system, the harness is still based upon a *simulation* of how the system will behave. Only by testing the code in an integrated environment can a developer be completely confident in understanding how the product code will perform. This is not in any way an excuse for not having good test plans and doing thorough testing, as simply running the product code is by no means adequate for identifying problems. However, no matter how good the test plans, if the testing is done only in a test harness we believe that this is again not adequate testing.

While some methodologies allow developers to write a significant amount of code without running it, many iterative methods require that new code be integrated with the rest of the system after each task is completed. As suggested by the principle of Old Bugs Die Hard, this minimizes both the amount of erroneous code written and the risk that new code begins to depend on its incorrect behavior. It is becoming widely recognized, and we believe rightly so, that code should be run and integrated frequently during the development process.

As noted above, even if developers understand and correctly interpret the technical design, there is still a risk that errors are introduced as the software is written.

Problem: Despite developers' understanding of the architecture, local errors are made in the implementation.

All programming methodologies recognize that some errors will occur during implementation and have practices that concentrate on quickly detecting and correcting them. Common practices used to address implementation errors include frequent integration, testing and code reviews. As we discussed above, frequent integration can be used to validate the design and usability of the system. In addition frequent integration is useful for helping ensure the correctness of the implementation. As described above, many iterative practices, including XP, depend on frequent integration to discover both mistakes made in interpreting the design and in writing the code. Many methodologies also advocate peer review of code and devote significant amounts of resources to testing the product. Steve McConnell further recommends[12] that each of line code be executed in a debugger by the developer who has written it.

One of the most common practices used to address the fallibility of programmers is peer code review. Just as design reviews are important in communicating aspects of the system architecture, code reviews ensure the consistency of coding conventions and avoiding duplicated code. In addition to finding mistakes, code reviews also allow developers to show off well-written pieces of code and can serve as an opportunity for less experienced programmers to learn techniques from more experienced members of the team. There are many different kinds of code review practices. Perhaps the most common is having a small group of developers read and comment on the work of one developer. Another practice is to have assigned code readers, whose job is to read and understand the designs, interfaces and code that make up the product. Yet another practice is to have more experienced developers, or "technical leads" work closely with less experienced developers, reviewing one another's work.

In order for any of these practices to be effective, the code must be transparent and understandable to anyone in the development group. Time invested in making code readable "pays off" in terms of time spent by others reviewing the code. Well-constructed code also pays off for the author if the code is set aside for a long period of time before changes are required.

While XP does not have a structured code review process, the practice of Collective Ownership serves as a sort of informal peer review process. XP also uses Coding Standards to help ensure that everyone can understand what has been written. In addition to encouraging the entire development team to look for (and correct) errors in the code, XP takes a radical view of *preventing* programmer mistakes. The practice of Pair Programming focuses directly on discovering errors as they are introduced. Because Pair Programming requires more time and attention than practices like code reviews, developers are more likely to find mistakes; the time required to prepare for and participate in a code review is typically a fraction of that necessary to write that same code. Also, because the resulting code is such a clear deliverable in the act of programming, developers participating in Pair Programming are more likely to become engrossed in the process and remain "on guard" for errors that might arise. In contrast, reading or reviewing code are more passive activities with no concrete deliverables, allowing attention and focus of developers to drift and fail to detect errors.

The XP practice of Continuous Testing is also important in finding programmer mistakes. While we pointed out above that unit testing is not very effective in finding problems caused by differing interpretations of the design, it certainly can be useful in finding errors in the code. However, the risk of having *developers* design and write unit tests, as advocated by XP, is that the code already handles any problematic cases probed by such unit tests. If oversights were made by the developer in writing the code, the same oversights are likely to be present in unit tests written by that developer. These tests become more useful as code is changed as, for example, during refactoring; the benefit lies more in ensuring the stability of the behavior of the code over time rather than ensuring its correctness. This applies to pair programming as well as to individual programmers – tests will tend to focus on things that the pair has already considered in writing the code itself.

Other methodologies advocate separating the responsibility of testing from designing and implementing the software. In these practices an independent group is responsible for validating that the software implements the required functionality and that it complies with the system architecture. Techniques such as white box, unit and integration testing are all used to validate that the implementation is consistent with the design and satisfies the requirements. These practices do not suffer as much from the risks associated with developer-written tests, since it is less likely that the same oversights are made by both the developer and the tester.

Finally, the practice of Sustainable Pace tries to ensure that each developer is as effective as possible while at work. XP may be trying to be overly general in stipulating this practice, however. In many cases, different kinds of tasks may be interleaved to maximize the potential of a developer. Many methodologies advocate a mix of activities, including design, programming, review and testing. There is, of course, an upper bound to the number of simultaneous tasks that can be effectively handled, but overlapping different types of tasks helps to keep developers fresh and ready for new challenges.

These practices for addressing the problem of mis-implementation reflect the following principles, as well as reflecting some of the principles introduced above, particularly Broad Scrutiny.

Principle of Readable Code Source code is more than a set of instructions for a computer: it is also an embodiment of the programmer's understanding of the domain. In order to communicate this understanding, code needs to be readable by people, not just machines.

Principle of Intellectual Focus Developing software is an intellectual activity that requires focus. There is a limit to the length of time for which a person can stay focused [repeated from the introduction].

Principle of System Dependability The degree of confidence in the reliability of a software system should be proportional to the cost of its failure and the required repair [repeated from the introduction].

The principle of Broad Scrutiny, previously discussed with respect to mis-interpreted architecture, is also the basis for many of the practices just described, from code reviews and code readers to Collective Ownership and independent QA groups. The fundamental idea is that risk of mistakes made by individuals is mitigated by exposing the work of these individuals to the scrutiny of a larger group. Where the author brings a set of assumptions and a single perspective, others bring new perspectives and question the assumptions underlying the design. For example, where tests are designed by someone *unfamiliar* with the implementation, this lack of knowledge is a positive aspect of the practice: there is a better chance of finding a mistake in the code if a second (or third) person is validating the desired behavior.

Many of these practices also emphasize the importance of the human element of software development, in part reflecting the principle of Readable Code. This principle encourages practices such as documenting and commenting code, and having coding standards, in order to ensure that people can readily understand the code. Readable Code highlights the value of practices where code is read and understood by others, thus guaranteeing it is read by some*one* (and not just by a compiler). The principle of Readable Code is often applied along with Broad Scrutiny, such as in the practices of code reviews or code readers. These practices are intended to find problems such as boundary cases, duplicated code, or other issues that may not be revealed by a compiler or even by running the code. A program maybe perfectly valid as far as a compiler is concerned, but another human may be able to point out more abstract flaws in the thinking behind the code.

The principles of Imperfect Simulation and Readable Code say two very different things about the nature of source code. On one hand, Imperfect Simulation states that, for every input, there is a predictable output of a program and that this behavior is best understood through running the code in the real environment in which it will be used. At the same time, Readable Code states that there is more to a program than simply how it is interpreted by a machine. Understanding this less directly observable nature is important for two reasons. First, for most large programs, there are so many inputs and outputs that it's impossible to test all possible executions. Therefore a more abstract understanding is necessary to convince ourselves of the correctness of the code. Secondly, the current behavior of the program tells us nothing of how we may be able to change the code to adapt to changing requirements or different designs. Understanding the theories behind the code, the models and ideas used by the programmer to implement the current functionality, allow us to see how this code can be extended or reused. These theories can be communicated through the use of coding conventions, comments, patterns, or the types of abstractions afforded by higher level languages, all ways in which the code is made more readable.

While XP explicitly captures the principle of Intellectual Focus in the practice of Sustainable Pace, a similar practice is probably exercised in most successful software organizations. As discussed above and in the introduction, while it's possible to push developers to realize an important goal, understanding the effective limits of developers is a responsibility of the leadership and management of the group. This understanding is not only necessary to estimate current tasks and plan for the next set of milestones, but also to ensure the longer-term health of the team.

When considering any of the practices used to detect errors made by programmers, one must carefully weigh the costs and potential benefits. Because XP does not put a particularly high value on System Dependability for initial releases of new functionality, the on-site user is left testing the results of the integrated system and ensuring that the product meets the defined requirements. While this practice can reduce the cost of the project (in terms of people and time devoted to testing the product), it implicitly assumes these savings are higher than the the cost of failures once the product is deployed. In some instances, the correct behavior of the software may be so critical that significant resources need to be committed to review, testing and simulation.

The application of many of the principles in this section must be balanced by a need to ship the software. Principles such as Broad Scrutiny and Readable Code require time and effort beyond what is necessary to implement the specified functionality. The benefit of committing additional resources is (hopefully) a higher quality and more maintainable product, but the principle of System Dependability must be carefully applied to determine when there is a real need to allocate additional time or even delay the project. We believe that this tradeoff is one of the most poorly understood and difficult problems facing development teams today. That is, once a group has the capacity to produce software of known quality on a predictable schedule, how does that group decide exactly how much quality to deliver? We believe the answer lies in understanding the risks and costs of software failure, the expectations of customers, and the needs of the market. In some cases, the cost of failure may be low enough, or the benefit of new features high enough, that shipping a product with a potentially large number of flaws is acceptable. In other cases, customers may be unwilling to accept software that is not thoroughly understood and rigorously tested. Understanding these issues is critical to delivering a product that not only meets a real need, but also does so in a timely manner.

5 Conclusion

Our main goal in this paper has been to enable software developers to make more informed choices among development practices, by providing them with a means of understanding the broader principles that underlie software development problems. In drawing careful distinctions between problems, practices and principles of software development, we believe that it is possible for developers to compare practices from quite disparate methodologies and to choose the practices that are best suited to their project. We further hope that careful attention to principles, rather than just practices and problems, will also be of help to methodologists in their studies of software development.

We have identified about two dozen principles that we have found to be applicable to a broad range of software development problems and practices, and related those principles to some common problems that arise in software development projects. These principles are intended to describe comprehensive or fundamental laws, properties or assumptions in software development. Our goal is to help enrich our understanding of the problems encountered in software development and the practices being used to solve those problems. We do not claim that the principles presented here are either the best or only principles of software development. Instead, we aim to begin a dialog and to explicitly identify broadly accepted principles.

In our investigation, we have seen a number of sets of competing principles. We believe that these competing principles play a particularly important role in identifying the best practices for a development project, because the relative importance of these principles can lead to adopting very different practices. In our view, the existence of widely divergent practices, such as the Waterfall Model versus Extreme Programming, can be seen to arise from such conflicts in underlying principles. For instance, the conflict between the principles of Real Use and Clear Statement can lead to very different practices. In a pure software project, it is possible to give the Real Use principle the most weight and follow an incremental development approach as advocated by XP and the Agile Methodologies. On the other hand, in a project that involves an integrated system with both hardware and software components, it is more important to do careful up-front specification as in the Waterfall Model, weighing the Clear Statement principle more heavily than Real Use. This is particularly true when the hardware is not available until late in the project.

For projects that involve only software, we strongly believe that iterative practices are a good choice for dealing with competing principles, because iterative practices allow one to incrementally mitigate the risk that arises from the underlying conflict. This applies not only to the use of incremental delivery practices, as advocated by XP, but more generally to the use of iterative or incremental practices to address competing principles. While the use of incremental delivery has been widely argued for by supporters of the Agile Methodologies, the use of iterative approaches to scheduling and to software design do not seem to be as widely discussed.

Incremental specification and estimation practices arise from the conflict between the principles of Specification Cost and Changing Requirements on the one hand and the principles of Unreliable Memory and Adequate Specificity on the other. The former principles suggest limiting the amount of written specification whereas the latter suggest that it is difficult to develop software without a good written description. We believe that the best specification and estimation processes are again iterative, starting with high-level descriptions and coarse estimates that progressively get as detailed as necessary, with feedback both from customers and developers as more detailed passes occur. The higher-level specifications and estimates can be used to drive a longer term plan that sets direction, while the more detailed ones are used to drive development of features in short delivery cycles. The use of incremental estimation and scheduling are not as broadly recognized as the use of incremental delivery practices.

Evolutionary architecture arises from the conflict between the principles of Build With a Design and Build to Understand. The former principle suggests that a good technical design is necessary a priori whereas the latter suggests that it is often impractical to develop a good design before construction. Building a working system and then evolving the architecture of that system over time have been recognized as a good way of addressing this conflict. Such an approach to technical design runs counter to that for physical systems, where it is nearly always important to design first (and perhaps then refine the design but not evolve it to a totally different design). The use of evolutionary architecture is the subject of considerable debate with many claiming that it is tantamount to having no technical design. We strongly believe that evolutionary architecture is the best way to respond to the only true external constraints on software, which are the required functionality and level of performance.

While many methodologies have only laid out sets of practices, others have begun to enumerate sets of beliefs and assumptions that form a basis for the decisions that surround their practices. We continue this initiative in describing what we have termed the principles of software development. In so doing, we have attempted to relate individual principles to specific problems and practices, rather than enumerating principles that apply to an entire methodology. We believe that it is important to be able to consider the relative importance of relevant principles for each project, and to thereby guide the choice of practices. Seen in this light, XP is not so extreme in terms of the principles that it observes but rather in the degree to which it applies them. Given the examples we have discussed with respect to XP and other common methodologies, we hope that others will be able to relate practices that they are using or considering to these or other underlying principles. We also hope that these principles will broaden our understanding of existing practices and may form the basis for new methodologies.

References

[1] K. Beck, *Extreme Programming Explained: Embrace Change* Addison-Wesley, October, 1999.

[2] K. Beck and M. Fowler, *Planning Extreme Programming*, Addison-Wesley, October, 2000.

[3] B.W. Boehm, Software Engineering, *IEEE Transactions on Computers*, Vol. 25, No. 12, December 1976, pp. 1226-1241.

[4] B.W. Boehm, *Software Engineering Economics*, Prentice Hall, October 1981.

[5] F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, Addison-Wesley, August, 1995.

[6] A. Cockburn, *Agile Software Development*, Addison-Wesley, December, 2001.

[7] M. Fowler, Articles on XP and Agile Methods, http://martinfowler.com/articles.html.

[8] W.S. Humphrey, The Personal Software Process (PSP), Technical Report CMU/SEI-2000-TR-022, November 2000.

[9] J.J Marciniak, ed. *Encyclopedia of Software Engineering*, 518-524 Wiley, 1994.

[10] D.R. McAndrews, The Team Software Process (TSP): An Overview and Preliminary Results of Using Disciplined Practices, Technical Report CMU/SEI-2000-TR-015, November, 2000.

[11] S.C. McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, July, 1996.

[12] S.C. McConnell, *Software Project Survival Guide*, Microsoft Press, July, November, 1997.

[13] Rational Corporation, The Rational Unified Process, http://www.rational.com.