Building Quality In – A Model for Achieving High Quality on Any Type of Project

Kathy Iberle

Pacific Northwest Software Quality Conference 2007

Abstract

We've all heard that it's better to "build in quality" than to test in quality. Have you ever wondered how exactly *is* quality built in?

Most books about quality assurance say that following good practices will ensure that quality is built in. However, the quality assurance books describe only the "quality" practices – configuration management, testing, auditing – and not the practices which actually create the software. Software engineering books do describe the practices which create the software, but often without explaining how those practices build in quality, or under what circumstances they work best.

If you understand the fundamental principles that make some software engineering practices work in some circumstances and not in others, you'll be able to put together a set of practices which will build in quality for any situation. Once you understand the principles, you can choose practices, testing methodologies, and so forth to fit whatever your environment may be.

This paper explains the fundamental principles and demonstrates how to apply them. We'll take a look at identifying what type of mistakes are most frequently made in your organization, and at choosing from a toolkit of prevention and detection practices to address these mistakes. We also consider how to choose practices which will best fit your organization's software development process.

Biography

Kathy Iberle is a senior software quality engineer at Hewlett-Packard, currently working at the HP site in Vancouver, Washington. Over the past two decades, she has been involved in software development and testing for products ranging from medical test result management systems to inkjet printer drivers to Internet applications. Kathy has worked extensively on training new test engineers, researching appropriate development and test methodologies for different situations, and developing processes for effective and efficient software testing.

Kathy has an M.S. in Computer Science from the University of Washington and an excessive collection of degrees in Chemistry from the University of Washington and the University of Michigan.

Her website is at www.kiberle.com

Introduction

Have you ever wondered exactly how one goes about "building quality in?" Is it possible to avoid being constantly plagued with errors and rework, with endless "stabilization" phases, with compromise and customer dissatisfaction?

It is possible, but rarely well explained. Worse, there's a lot of misleading information circulating in the industry. Most of the sources prescribe solutions that are aimed at specific contexts – a particular type of business, or a particular type of product. The solutions do (usually) represent the application of fundamental principles to those contexts, but the principles themselves are either mentioned superficially or not explained at all. This makes it very difficult to apply lessons learned in one context to any other context.

If you're writing custom financial applications or client-server web-enabled apps, you're in luck – those are the two most common contexts so you're likely to be able to find and successfully use a prescribed solution. All the rest of the diverse software industry continues to struggle, trying to understand how similarly good results can be achieved when you simply don't have the ability to release every two weeks or to work directly with your customers.

This paper is for the rest of us. I will explain the fundamental principles that are behind most of the successful methods and processes touted in the industry. Once the principles are understood, you too can apply those principles and assemble your own high-productivity, high-quality software development methodology – one that will fit your environment and your business.

Fundamentals of "Building Quality In"

Software development is a creative process which is fraught with opportunity for error. Worse, defects, once created, tend to propagate, either becoming more and more expensive to fix or breeding more defects. [MCCO96]

Let's talk about defects. There are two things you can do about defects:

- Prevent them from being made in the first place.
- Find the defects very early, before they have a chance to devour time and money.

These are two fundamental principles that you will find behind many of the practices in successful software engineering and quality assurance methods. For instance,

"Prevent defects" can be found in
- ISO 9001's insistence on tracking training records (which implies training)
- CMM's emphasis on establishing and using known practices
- the inclusion of configuration management in quality assurance methods
- the use of code inspections to teach better coding skills
- the frequent re-assessment and tuning of practices found in most iterative software development methods (e.g. see [MAL2002])

"Find defects early" shows up in various forms in nearly every software development method:
- Waterfall and V-model's emphasis on inspections of specifications, designs, and code to find errors
- Agile methods' use of test-first programming
- Prototyping to determine whether you're on the right track.

Some combination of these two principles is the basic idea behind most "quality" methods.

Why Doesn't One Method Work For Everyone?

Once a project starts controlling its most common defects, productivity will almost always increase. That's because, the longer a defect stays in the system, the more costly it will be to find and fix.

Setting up the control mechanisms for the first time will cost time and money, and this cost can be significant. But once the controls are in place, the cost of producing the software goes down. This has been demonstrated in many studies – both in waterfall-style development [MCCO98] and in agile development.

However, if the project applies control methods that *don't* control its most common defects, the cost of the project can actually go up. New work has been introduced, and it isn't reducing the old work enough to compensate. I suspect this is the problem behind many failed process improvement projects. They aimed at the wrong problems. You need to go after the problems that cause *your* projects' defects, not the problems causing somebody else's defects.

So let's look at defects and where they come from.

Where Do Defects Come From?

A software *failure* is erroneous or undesired behavior. A failure is caused by one or more *faults* – incorrect code that will have to be changed. Behind most faults is a *mistake* – a task incorrectly performed by some individual. The term *defect* is somewhat ambiguous – it is used to refer to both faults and failures.

Almost every failure traces back to one or more mistakes made during some task. The mistake might have been made during the task of writing the code, but it could just as easily have been made during an earlier task in software development. It's also possible to have defects caused by unreliable tools, in which case the underlying mistake was in the writing of the tool or in making the choice to trust that tool. In this paper, we'll focus on mistakes made during software creation.

The tasks directly performed in software production fall into four main categories:

- Gather user requirements (determine what the customer wishes the product would do)
- Specify the product (decide what the product shall do)
- Design the software (figure out how the product will do those things)
- Implement the design (write the code)

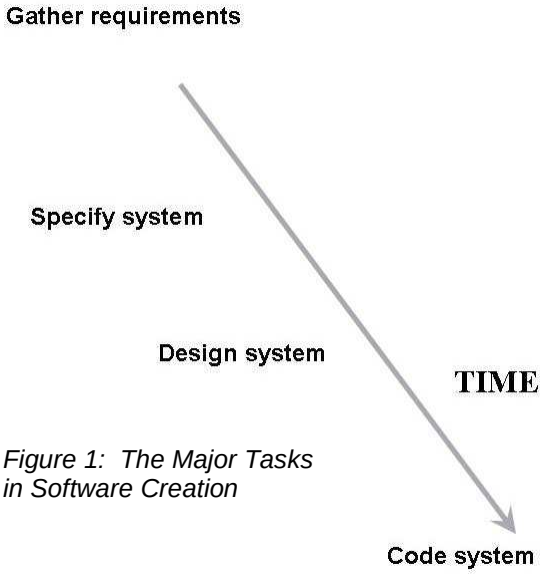


Figure 1: The Major Tasks in Software Creation

In all software lifecycles, these four tasks are performed in varying combinations. The classic waterfall lifecycle instructs the team to

- collect all the user requirements
- specify the entire system
- design the entire system
- code the entire system

in that order.

An iterative lifecycle recommends

- collecting some user requirements
- specifying the features to meet those requirements
- designing the code for that functionality
- building that functionality
- checking with the user to see if the functionality is what the user wanted

and then repeating the entire cycle.

Figure 1 shows the steps sequentially, for simplicity.

Each of these four tasks is prone to characteristic mistakes. For instance, failure to initialize variables, and running off the end of an array are two mistakes commonly made during coding.

Mistakes in design are often mistakes of omission – failing to handle foreseeable errors and exceptions, or failing to check for external conditions. The mistakes made in gathering requirements are usually either misunderstanding the user's needs or failing to discover some of the user's needs completely.

Figure 2 shows mistakes which are commonly made during each task.

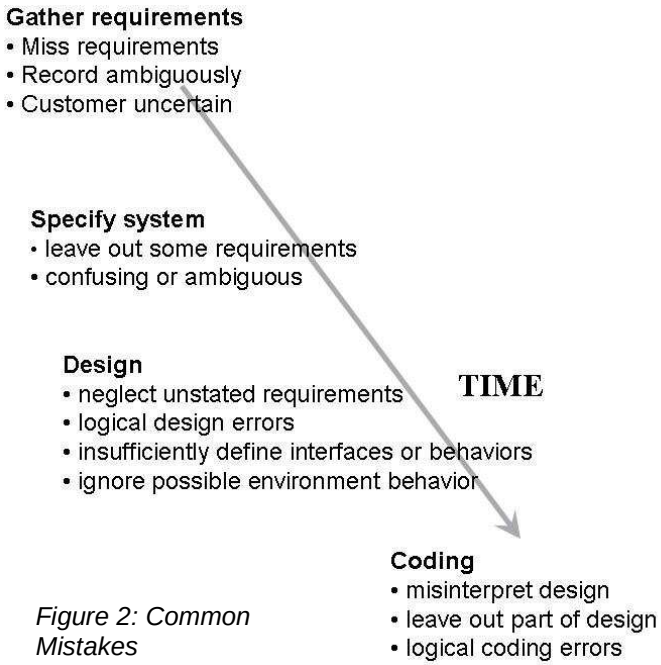


Figure 2: Common Mistakes

All organizations make most of these mistakes, but not in the same proportions. I have observed over the years the type of mistakes most commonly made varies widely between different organizations. I no longer assume that adoption of any particular practice will be equally effective in every organization.

I once attended a week-long class on inspections with half a dozen development teams from the same organization. The instructor assured us at the beginning, "Code inspections are the most useful type of inspection. My classes always discover that, but I'll teach you to inspect everything for the sake of completeness." We spent the next four days practicing inspections, using samples of our own specifications, design, and code from recent projects. After the class was over, some of the teams compared notes. Four out of five had discovered many more mistakes in their specifications than in their code. We didn't know if we were uncommonly good coders, or uncommonly bad specification-writers, but we could easily see that inspecting specifications would be more useful for us than inspecting code.

I've seen this diversity over and over in different organizations, both from personal experience and in the literature (for example, see [GRAD97a]. Every group of people has its own strengths and weaknesses, and every set of processes has its own strengths and weaknesses. This means that every organization has its own characteristic pattern of mistakes.

Later in this paper, we'll talk about how to identify your organization's characteristic pattern of mistakes. For the moment, let's assume we have determined that pattern, and go on to consider what to do about the mistakes.

How to Prevent or Detect a Particular Class of Mistakes

Each type of characteristic mistake can be prevented or detected by different means. One size does *not* fit all – a practice that's very good at detecting a particular type of mistake can be terrible at detecting a different type. For instance, white-box testing is good at finding coding errors but is blind to missing requirements. Prototyping is good at finding requirements problems but does nothing about coding errors. In order to craft your own customized quality control method, you not only need to know what types of mistakes your organization typically makes, but you also need to know which practices are good at dealing with those types of mistakes.

Prevention Methods

First, let's take a look at prevention. Prevention means just that – the mistake is never made at all. Prevention methods are generally under-utilized and are often ignored completely in favor of the more obvious detection methods. Yet the cheapest way to fix a mistake is never to make it in the first place.

The two most powerful methods to prevent mistakes generally are

- using an appropriate practice or method for a given activity
- adequate training in the activity and the chosen practice or method

For instance, most of the design methods in use today are intended to prevent design mistakes by using a systematic method and notation to help the designer think through the problem clearly. There are different design methods for different types of software problems. Choosing a method inappropriate to the problem (for instance, trying to design a database schema using structured programming) can introduce a lot of mistakes. Using no design method at all is a well-known way to introduce errors.

Another example is gathering requirements. Many software engineers have had little or no training in gathering requirements, and as a result tend to overlook the non-functional requirements of the system. The requirements are never gathered or are poorly understood, and the developers don't realize this. The omissions are built into the design, and then the code, and these mistakes are not noticed until user acceptance testing or beta testing. This is a very expensive set of mistakes, and yet they can often be prevented by investing less than a hundred dollars on a classic software requirements book and a day or two of developer time learning how to recognize basic omissions in non-functional requirements.

Detection Methods

Detection methods include both inspection or review of an intermediate document and exercise of the actual software. Some sources refer to these as *static* and *dynamic* testing respectively.

Intermediate documents such as requirements documents, design documents, and code listings are all outputs from one of the four main tasks. Inspecting output from a task generally finds mistakes made during that task. Mistakes made during previous tasks are not usually apparent.

Various types of dynamic testing also focus on finding particular types of mistakes. Some examples:

- Black-box system testing is good at finding omitted specifications, since it compares the software to the specifications.
- Black-box testing can be good at finding failures to consider the environment in which the software will run.
- White-box testing is good at finding logic errors – mistakes made during coding. Some logic errors can be found quickly by a black-box system test, but other logic errors will lurk unseen for weeks or months, and be very difficult to diagnose when they are exposed. White-box testing generally finds them all quite quickly.
- Automated functional testing (beloved of test-first programmers) does a great job of finding logic errors and design errors, but usually will not find failures to consider the environment nor omitted requirements.

*Table 1 (*next page) lists some of the most common methods to prevent or detect each type of mistake.

Table 1: Methods to Prevent or Detect Particular Types of Mistakes

Sample Prevention and Detection Methods			
Activity	**Mistake**	**Prevention**	**Detection**
Gather Requirements	Miss requirements	-- interviewing techniques -- use cases (as opposed to lists of features)	-- prototyping -- requirements inspections -- user acceptance testing -- beta testing
	Record requirements ambiguously	-- education in writing requirements [WIEG03]	-- requirements reviews
	Customers are uncertain - they don't know what they want		-- prototyping -- user acceptance testing
Specify System	Omit some requirements – fail to specify any feature to meet them	-- requirements traceability tools	-- user acceptance testing, beta test -- spec reviews
	Write confusing or ambiguous specifications	-- SMART requirements -- Planguage [GILB88] [WIEG03]	-- requirements inspection -- user acceptance testing
Design System	Omit some of the specifications – fail to design anything to implement them	-- requirements traceability tools	-- black-box functional testing
	Make logical design errors	-- design standards -- UML, ERD, structured programming	-- black-box functional testing -- design reviews
	Ignore possible environment behavior	-- education in technologies	-- black-box system testing
Implement System	Misinterpret the design	-- education	-- code reviews -- integration testing
	Leave out part of design – fail to code anything to implement it	-- design checklists -- CASE tools	-- gray-box test against design specs
	Make logical coding errors	-- Hungarian notation -- Coding standards -- Test-first development -- Mentoring, education -- see [MCCO04a] for more	code inspections -- unit testing (white-box) [MCC04b]
	Ignore possible environment behavior	-- defensive programming techniques	-- system testing

Designing Your Own Quality Control Method

There are a lot of different practices in the list above, and some of them actually conflict with each other. How do you go about choosing the right practices for your organization?

There are three factors which, once understood and recognized, allow an organization to identify the practices which will be most effective for that organization:

- Cost profile: The criticality of the software and the cost of making changes after release will make some types of defects more expensive than others. This varies a great deal from business to business. A given defect can be fatal in one business and merely annoying in another.
- Characteristic mistake profile: the tasks in which mistakes are typically made, and the types of mistakes most often made.
- Existing "lifecycle": The order in which the software creation tasks are performed, and the prevention and detection practices already in place.

Once you've identified your mistake profile and understand the cost of defects for your organization, you will know which mistakes you want to eliminate first, and you can then choose appropriate practices for your organization. We'll go over each of these steps in this section.

Cost Profile: Identifying the Mistakes to Target

It's not a good idea to attempt to reduce all types of mistakes simultaneously. If you start following someone else's laundry list of practices, you'll probably end up implementing at least one change which addresses a type of mistake that isn't prevalent in your organization. The change will cost time and money but nothing will get much better. This not only wastes resources, it makes people less willing to work on subsequent changes.

If a particular class of failures hasn't already been chosen as a target, you'll need to identify a class of failures to work on. Most organizations would prefer to reduce the failures which are costing them the most – in development time, in support costs, in reputation.

You may find that the organization already agrees on which failures to target, or you may find there is a lot of debate. I've run into mythical failures, where a type of failure was so problematic for so long that the management believes it's still occurring for a year or more after it was wrestled into submission.

If a particular class of defects hasn't been targeted, you might consider starting with failures that had an obvious cost, such as one of these groups:

- Failures which required maintenance releases or patches
- Failures which forced a project to miss its planned release date
- Failures which routinely eat up a lot of development time – failures which are not individually serious but are collectively numerous.

Characteristic Mistake Profile: understanding which types of mistakes are most commonly made

Once a particular class of failures has been identified as a target, it's necessary to understand what type of mistakes are being made to create these failures, and *why* these mistakes are being made.

One of the best ways to identify characteristic mistake patterns is through root cause analysis. [GRAD97b] *Root cause analysis* identifies the development task during which the fault behind a failure originated. (The tasks are listed in Figure 2). Once the task is known, it's easier to figure out what type of mistake is being made and what could be done to prevent it or detect it earlier. A root cause analysis study involves choosing a representative sample of defects and looking for patterns in their root causes.

In some organizations, a programmer is required to identify the root cause of a defect and record that when fixing the defect. In other organizations, this isn't done. Even when the original programmer is required to record the root cause, the root cause may frequently be incorrectly identified. Many developers will classify all defects as coding errors even when the mistake clearly originated in an earlier task.

Generally, identifying the root cause is most accurately done by the original developer or someone else familiar with the code and the defect. In organizations where root cause isn't typically recorded or is often inaccurate, root cause studies can still be done after the fact, by doing a root cause analysis on the entire set of defects or on a randomly chosen sample.

Sometimes the developers need some help learning to attribute root causes. The developer's initial response to the failure report often gives me a clue to where the mistake occurred.

- "It shouldn't be doing *that*." - the code or the design is wrong
- "It works as designed." - either there is no written specification, or unwritten requirements (as understood by the test team) are in conflict with the written specification
- "But users won't *do* that." – requirements and specification are probably both missing
- "I'm not sure if that's ok." – the specification may be ambiguous
- "I didn't know anyone wanted that." – a missing requirement
- "These don't match. Which one is right?" - conflict between two overlapping specifications.

The distinction between a problem introduced in design and a problem introduced in implementation is often fuzzy to the developers, particularly if there wasn't much of a written design in the first place. Some symptoms of design problems:

- Disagreement between two teams over interface definition or responsibility of their code
- Fixing the problem requires ripping out big chunks of code and redoing them
- Only occurs in one environment (characteristic of design problems caused by neglecting to account for differences between operating systems, locales, etc.)

Choosing the Appropriate Practice to Apply

Once you know which types of mistakes are being made, you can choose practices which will be the most likely to prevent or detect those mistakes early. Table 1 offered a number of choices for many types of mistakes. How do you choose the ones that will best work?

This usually depends on the lifecycle or software development method being used.

For instance, a requirements defect can be noticed either when the requirements are gathered (shown in bold), or noticed after the code is created by comparing the working system with the user's expectations (shown in italics). The intervening development tasks don't provide any additional opportunities to either notice or prevent requirements errors.

Mistake	Prevention	Detection
Miss requirements	**-- interviewing techniques**	**-- prototyping** **-- requirements inspections** *-- user acceptance testing* *-- beta testing*
Record requirements ambiguously	**-- education in writing requirements**	**-- requirements reviews**
Customers are uncertain - they don't know what they want		**-- prototyping** *-- user acceptance testing*

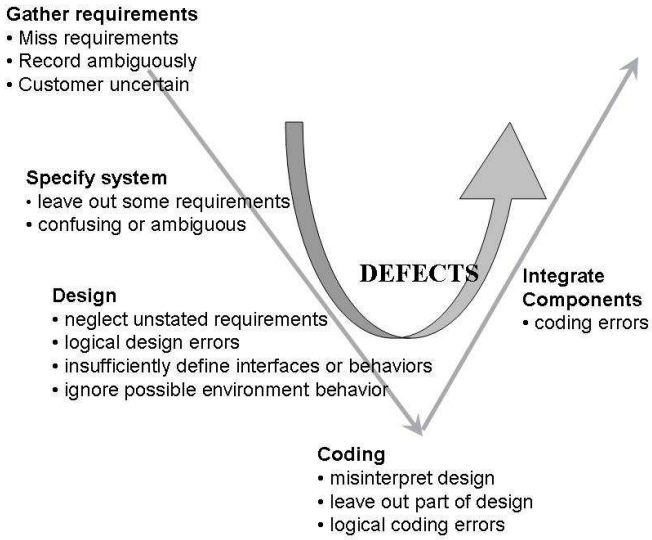
In a waterfall lifecycle, this means there are two chances to find requirements problems: at the very beginning of the project, and close to the end. In an iterative lifecycle, there are still two chances to find a requirements problem – but the second one is no longer close to the end of the project.

The effect of your lifecycle on the effectiveness of a practice is easier to see if you draw out your lifecycle and trace the path of a particular type of defect through the lifecycle.

Figure 3 shows the defect injection during a project using a waterfall lifecycle. If nothing is done to prevent or detect defects, each task introduces more defects and they accumulate until the end of the project.

However, all projects have at least some detection practices already in place. A lifecycle with such practices is shown in Figure 4.

Figure 3: Defect Injection in a Waterfall Lifecycle



Let's consider a defect caused by the customer not telling you what they want because they don't know what they want until they see the product.

Follow the gray arrow marked "time" from the original source of the defect (the requirements task).

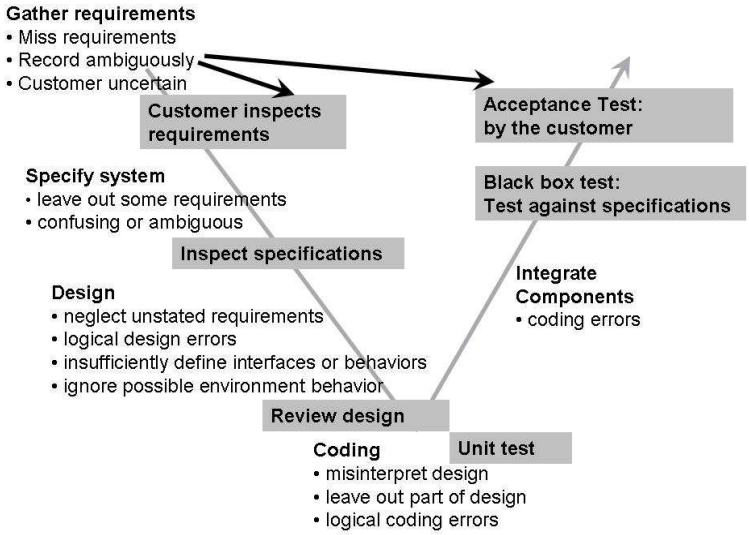
The first detection practice is a requirements inspection. Will a requirements inspection notice that the customer didn't tell you something? Probably not.

Next we come to design review, and then unit testing. Will design review or unit testing catch it? Definitely not.

Will the black-box testing catch it? Maybe – it depends on how well the testers understand what the customer wants.

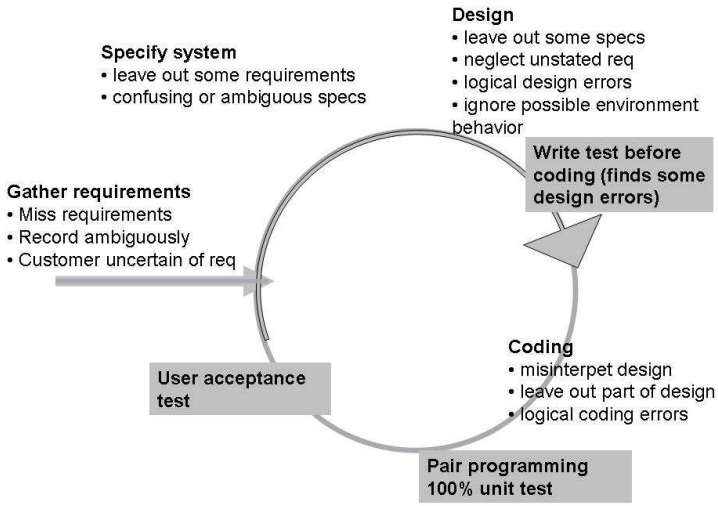
Will the user acceptance test catch it? Most likely yes.

Figure 4: Mistakes in gathering requirements are addressed by these practices



We've learned that we likely won't find this error until quite late in the project. If this class of mistakes is a significant cause of problems for your organization, consult Table 1 for other practices. Use the diagram to figure out if the practice will be effective within your particular lifecycle. In this case, the practice of prototyping might work.

The effective prevention or detection practices for a given type of mistake can be considerably different in different lifecycles. For instance, follow the same omission in requirements through the Extreme Programming lifecycle shown in Fig. 5.



The first detection practice is test-first development. This is not going to find missing requirements.

The next two practices are pair programming and unit testing. These won't find missing requirements either.

The fourth practice is user acceptance testing. When conducted by the user, this does find missing requirements.

Thus, the missing requirement is noticed quite early in the project, at the end of the first iteration.

Figure 5: An Extreme Programming project

Bringing It All Together

Optimizing your software development process to build in quality boils down to these steps:

0) Knowing which types of prevention and detection practices work best for each type of mistake. (This is "step 0" because it is background information – it doesn't need to be repeated for every process improvement project).
1) Identifying which mistakes your organization is most prone to making and which are the most costly
2) Understanding when in your lifecycle those mistakes are made
3) Knowing which practices are already in use in your department, and how effective those practices will be at preventing or detecting the problems
4) Based on all of the above, choosing appropriate practices to address those mistakes and applying them

Let's walk through a couple of examples. These are taken from real life, with some details changed to simplify the story and protect confidentiality.

Example 1: Project Giraffe 2.0: Backup Trauma

This project was a fairly straightforward data management application incorporating a commercial database management system. It was created by a single team who work in the same building.

Identifying the Mistakes to Target

The Giraffe team struggled with defects in the backup/restore function of Giraffe 1.0 and 1.1.
There were multiple failures on each project and each time, it took an agonizingly long time to find the fault behind the failure.

Understanding which type of mistake is occurring

While planning Giraffe 2.0, the team had a long talk about the backup/restore defects and why they had occurred. Giraffe 2.0 was planning to completely overhaul the backup/restore system to make use of newly released hardware with much larger storage capacity. The programmer who'd done most of the work, Steve, admitted that he didn't fully understand the backup/restore functionality provided by the database vendor and felt like he was working in the dark some of the time. The rest of the team was sympathetic, and assured him that no one else understood the backup/restore calls very well either.

Steve was making mistakes *during the design of the application.*..

The length of time it took to diagnose each defect was discussed too. Steve kept saying it would take him two or three days to diagnose each defect. Finally someone asked why it took that long, since the failed call could be read out of the error log. Steve informed the team that in fact, the error log didn't contain the return code from the failed call – he had to re-run the entire 2-hour backup with a debugger on to capture the return code. The error logging provided by the DBMS vendor wasn't working as advertised. The team agreed that this was a royal nuisance.

The cost of the design errors was compounded by the unusually high cost of replicating and diagnosing the failures.

Choosing appropriate practices for the existing lifecycle

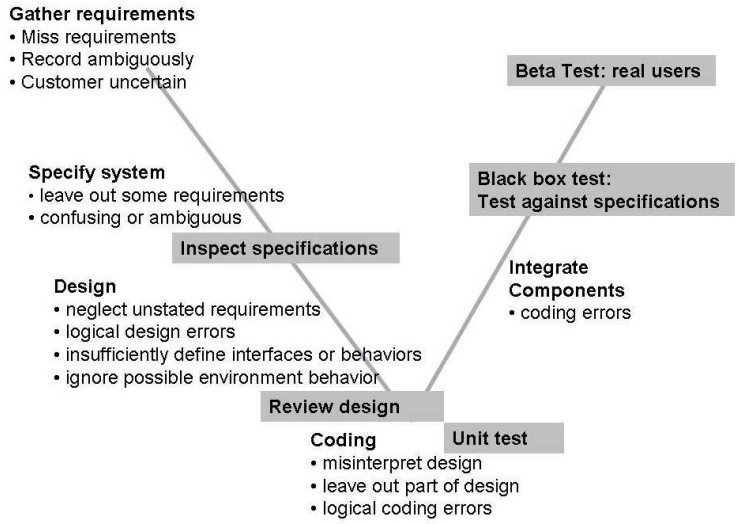


Figure 6: Lifecycle of Project Giraffe

The Giraffe team routinely mapped out their major detection activities in a "quality plan", so the entire team was already aware of the existing detection activities. This is shown in *Figure 6*.

After reviewing the existing prevention and detection activities, the team concluded that more design reviews wouldn't help, because none of them were familiar enough with the technology to notice mistakes. Instead, they discussed employing a prevention practice – education.

The team lead talked to Steve about the backup/restore functionality and what it would take for Steve to understand it better. Together, they identified a system administrator's class which spent a full day on the intricacies of backup/restore.

Another team member, Gina, had an idea which would speed up detection. She suggested writing a custom error handler to replace the off-the-shelf error handler, so the error codes would get logged whenever a failure occurred.

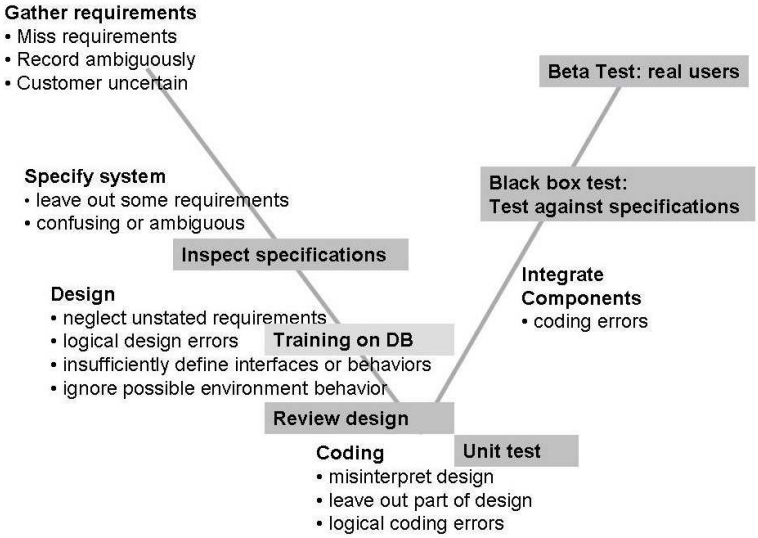


Figure 7: Project Giraffe with added quality practice

The team lead showed the manager the proposed changes.

Figure 7 shows Project Giraffe with the added prevention practice. This type of map is often helpful when discussing addition of practices with management, because it's fairly easy to show where defects are detected earlier and explain which types of defects will be addressed.

After some negotiation, the project manager agreed to the changes. Steve went to the training class, and Gina wrote the error handler in eight days (three more than her original estimate). The team was pleased when substantially fewer defects were found during the rewrite of the new backup/restore subsystem, and those were easier to find and fix than in the previous version.

Example 2: Project Zebra: Dialog Mismatches

The Zebra team was responsible for the installation software for a printer. The printer would be shipped with installation software and a hard-copy setup poster, both of which instructed the user on how to install the software and connect the printer. The printer software was created by a large set of teams who worked in various buildings.

Identifying the mistakes to target

The Zebra team was plagued with several episodes of problems spotted during beta tests. Various install dialogs didn't match the setup poster, resulting in confused users. Both the dialogs and the poster were expensive to change late in the project, so management was understandably concerned.

Understanding which type of mistake is occurring

As is often the case, the management initially said that the solution was "more testing". But was it? The Zebra team needed to know what mistake was occurring, and when. There wasn't agreement on whether the install dialogs were "wrong" or the setup poster was "wrong", much less what the root cause of the defect was.

The Zebra team set out to find the root cause of the problems – the task in which the mistake was occurring. This organization (unlike the Giraffe organization) didn't routinely map out its prevention and detection methods, and the sheer size of the organization sometimes obscured who was doing what. The Zebra team did some digging to find and put together the map shown in Fig. 8.

After the map was completed, the researcher traced the problem through each step in Fig. 8 to understand who was doing what. Since it wasn't clear where the problem was arising, the researchers started at the very beginning and followed both the install dialogs and the setup poster through each step in their lifecycle.

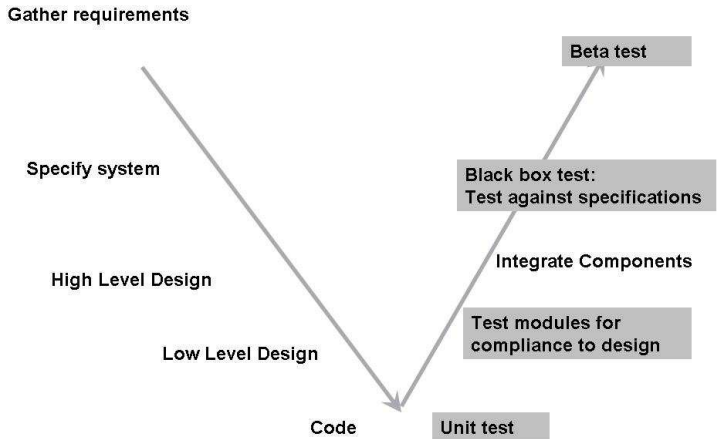


Figure 8: Development process for Project Zebra

- Requirements: the authors of the dialog and the setup poster both readily agreed that the users expect the install dialogs and the setup poster to match, and that they intended them to match.
  - The mistake was not made in this step.
- Specification of functional requirements: the researchers were told that the earliest "specification" of poster was a draft by the technical writers, and the earliest specification of dialogs was a sketch of install screens
  - Could the mistake have arisen here? Is dialog text stated in either specification?
- Design: there is no software interface between the dialogs and the setup poster
  - The mistake didn't arise during software design
- Coding / Implementation
  - Does the install dialog code match its specification?

The Zebra team needed more information. They sent someone to find samples of the install dialog specifications and a sample of the setup poster specification. They discovered that the early specifications of the poster and of the dialogs were both pretty vague concerning the actual dialog text. The authors of the dialogs continued to make changes throughout development, in response to feedback from various departments. The authors of the poster collected dialog screenshots at some point in development, but there was no obvious process for guaranteeing that the poster and the dialogs in synch with each other. The first time that the poster and dialogs were tested together was in beta testing.

Choosing appropriate practices for the existing lifecycle

The most obvious practice to apply was black-box testing: add earlier testing of the dialogs and setup poster together, rather than try to beef up the testing of either one alone. The Zebra team added a system test (prior to beta testing) which called for comparing the draft setup poster with the dialogs. This forced the poster writers and the test department to agree on a method for the test department to get access to the draft posters and be informed of the poster's deadlines. A distribution list was created.

If one wanted to prevent the mistake from occurring in the first place, then the setup poster and dialogs would have to be kept synchronized during their development. This can be a costly change to make in a large organization, because it involves three organizations (writers, developers, and test). It's frequently smart to make the less expensive change first and see how effective it is. In this case, when the developers started receiving defect reports on dialog/poster mismatches, they asked questions, discovered the new distribution list, and got themselves added, thereby allowing them to informally check the synchronization between the dialogs and the poster.

The Speed of Change

I've used the method described above in a number of different organizations over the years. It works reliably to identify practices which will be effective, but the speed of change depends a great deal on the organization in question.

The Giraffe team worked in an organization which had a history of analyzing problems and adapting its practices rather quickly. For instance, this organization responded to the unexpected learnings from the class on inspections by forming a temporary "Tiger Team" to revise the specification process for the entire organization. (The existing specification process had consisted of two paragraphs in the lifecycle handbook. After about three months, the team delivered a template, a three-page guideline on writing specifications, and a short training class. This organization typically did one or two such process updates per year, sometimes as many as four in one year.

One reason the organization could move so fast is that it was rather small – fewer than fifty software engineers total scattered across a dozen integrated product-development teams. Everyone knew the entire software development process rather intimately, and had easy access to all the people involved. Problems such as the Zebra team encountered were less likely to occur in the Giraffe team's organization simply because there were fewer people. The process of writing a setup poster wasn't a mystery to Giraffe's quality engineer, because she regularly ate lunch with one of the two learning-products engineers responsible for such things.

However, another significant reason is that the Giraffe team's organization invested in learning. The organization routinely required every project to create a list of the prevention and detection activities for its project in its "quality plan", and required every project to assess and document its quality plan's ability to prevent or detect the most dangerous errors. After a few years, the staff were all accustomed to this type of thinking and proficient at quickly choosing effective methods.

Once this sort of structure is in place, the organization is highly productive and can produce high-quality software for a very reasonable cost. However, there is an investment in learning how to do the analysis and then repeatedly doing the analysis, implementing the steps and observing whether or not the plan was effective. This is not an insignificant investment.

Conclusion

There are quite a number of software engineering or software process books which offer solutions to quality problems in the form of a process, such as the Personal Software Process [HUMP95] or eXtreme Programming [BECK00] or even the venerable V-model. The problem with this approach is that not one of these processes works on every single type of project. Processes which frequently produce good results for in-house IT applications may not make sense for embedded software, and processes that produce great embedded software may not result in excellent consumer applications.

By understanding the fundamental principles that make these processes work in their target situations, you'll be able to put together a process that will build in quality for any situation. Once you understand the principles, you can choose practices, testing methodologies, and so forth to fit whatever your environment may be.

In this paper, I explain two fundamental principles: understanding when and where mistakes are most frequently made, and choosing from a toolkit of prevention and detection methods to either prevent or detect the mistakes as early as possible.

We've seen:

- the types of mistakes commonly made during each of the four software development tasks
- which prevention and detection mechanisms typically work best at locating each type of mistake
- how to identify the problems that you most want to eliminate from your organization
- how to identify the types of mistakes behind those problems
- how to choose appropriate practices to eliminate those problems.

Good luck creating your own efficient and effective methodology to build in quality!

For further reading on this topic, I recommend Martyn Ould's *Managing Software Quality and Business Risk*, 1999.

References

[BECK00]: Beck, Kent; *Extreme Programming Explained*; Addison-Wesley; 2000.

[GILB88]: Gilb, Tom; Principles *of Software Engineering Management*, chapter 8; Addison-Wesley; 1988; also see www.gilb.com ;

[GRAD97a]: Grady, Robert B.; *Successful Software Process Improvement*; Prentice-Hall; 1997; p. 200

[GRAD97b]: Grady, Robert B.; *Successful Software Process Improvement*; Prentice-Hall; 1997; ch. 11 "Software Failure Analysis for High-Return Process Improvement Decisions." Root cause analysis appears in the literature under several keywords: *defect cause analysis*, *defect causal analysis*, *root cause analysis*, and *orthogonal defect classification*.

[HUMP95]: Humphrey, Watts; *A Discipline for Software Engineering*; Addison-Wesley; 1995. Describes the Personal Software Process

[MAL2002]: Malotaux, Neils; http://www.malotaux.nl/nrm/English/WrkSlds.htm

[MCCO96]: McConnell, Steve; *Rapid Development*; section 4.3 Quality Assurance Fundamentals; Microsoft Press; 1996; McConnell refers to a series of published papers from 1988 through 1994 which cite various figures on the cost of rework and the advantage of finding defects early. The most-often quoted statistics on this topic date back to the early 1980s and may not be entirely applicable to the projects of today, but the principle appears to stand firm.

[MCCO98]: McConnell, Steve; "The Power of Process"; IEEE Computer, May 1998; also see www.construx.com. This article quotes four studies demonstrating my point.

[MCCO04a]: McConnell, Steve; *Code Complete, 2nd ed.*; Microsoft Press; 2004;

[MCCO04b]: McConnell, Steve; *Code Complete, 2nd ed.*; Microsoft Press; 2004; ch. 22 "Developer Testing"

[OULD99]: Ould, Martyn; *Managing Software Quality and Business Risk*

[WIEG03]; Wiegers, Karl; *Software Requirements ed. 2*; pp. 228-229; Microsoft Press; 2003.