

# Brownfield Application Development in .NET

Kyle Baley  
Donald Belcham



Unedited Draft





**MEAP Edition  
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# *Contents*

Preface

## **Part 1 The ecosystem**

Chapter 1 Understanding Brownfield applications

Chapter 2 Adding version control

Chapter 3 Continuous integration

Chapter 4 Automated testing

Chapter 5 Software metrics and code analysis

Chapter 6 Defect tracking

## **Part 2 The code**

Chapter 7 Bringing better OO practices to the project

Chapter 8 Layering

Chapter 9 Loosen up: taming your dependencies

Chapter 10 Cleaning up the user interface

Chapter 11 Model view

Chapter 12 Risk and corruption: managing external system dependencies

Chapter 13 Keeping the momentum

Appendix Tools

# 1

## *Understanding Brownfield Applications*

“Welcome to Twilight Stars! Where fading stars can shine again!”

So claims the welcome package you receive on your first day. You've just started as a consultant for Twilight Stars, a company that provides bookings for bands of fading popularity. The web site that manages these bookings has had a storied past and you've been told that your job is to turn it around. This being your first gig as an independent consultant, your optimism runs high!

Then Michael walks in. He props himself up against the door and watches you unpack for a while. You exchange introductions and discover he is one of the more senior programmers on the team. As you explain some of your ideas, Michael's smirk becomes more and more prominent.

“How much do you really know?” he says. Without waiting for an answer, he launches into a diatribe that he has clearly recounted many times before...

“Did they tell you our delivery date is in six weeks? Didn't think so. That's a hard deadline, too; they'll miss their window of opportunity if we don't deliver by then. The mucky mucks still cling to the hope that we'll meet it but the rest of us know better. If we hold to that date, this app is going to have more bugs than an entomology convention. As it is, the client has zero confidence in us. He stopped coming to the status meetings months ago.

“Anyway, you'll be hard-pressed just to keep up after Huey and Lewis both left last week. They're only the latest ones, too. Despite what he may have told you about refactoring, the PM, Jackson, had to bring you in just to get this sucker out the door on time. Yeah, we know it's a mess and we need to clean it up and all but I've seen guys like you come through here on a regular basis. It'll be all you can do just to keep up with the defects. Which, by the way, we are mandated to resolve before delivery. Each and every one of them.”

With that, Michael excuses himself after suggesting (somewhat ominously) that you review the defect list. You locate the Excel file on the server that stores them and are taken

aback. There are some 700 open defects, some over two years old! Some are one-liners like "Can't log in" and many were reported by people who no longer work on the project.

Without knowing what else to do, you scan the list for a seemingly simple bug to fix: Display an appropriate page title in the title bar of the window. "That should be an easy one to pick off," you think.

Luckily, your welcome package contains instructions on getting the latest version of the code. Unluckily, the instructions are out of date and it's a good forty-five minutes before you realize you're looking at the wrong code.

After retrieving the actual latest version, you open the solution and do what comes naturally: compile. Errors abound. References to third-party controls that you haven't installed and code files that don't exist. As you investigate, you also discover references to local web services and databases that you have yet to install. In one case, it even appears unit tests are running against a production database. (Upon further investigation, you realize the tests are being ignored. "Phew!")

You spend the rest of the day tracking down the external dependencies and missing code off various team members' machines. As you drive home, you realize that the entire day was spent simply getting the application to compile. "No matter," you say to yourself, "now I'm in a position to be productive."

You arrive early on day two ready to do some coding. As you settle in to find that elusive line of code that sets the window title, you realize the code itself is going to be a significant obstacle to cramming the work required into the remaining time before release.

Eventually, you find the code to change a window title. It's buried deep in what appears to be a half-finished generic windowing framework. Tentatively, you "fix" the bug and test it the only way you can at the moment: by pressing F5. CRASH!

Fast forward to two o'clock. In the intervening hours, you've touched every major layer of the application after you discovered that the window title was actually being stored in the database (explained one team member: "at one point, we were going to offer the app in Spanish"). Near as you can tell, the new title you were trying to set was too big for the corresponding column.

Stop us if any of this starts to sound familiar

In the meantime, you still aren't able to even fix the bug because key files have been checked out by others (including the recently departed Huey) and the version control system doesn't allow files to be modified by more than one person. Also, though some areas appeared to be protected by automated tests, in fact, these tests are either ignored, not testing what they claim, or outright failing. Clearly the team doesn't run them often. One class containing over 10,000 lines of code is "covered" by a single test. Disturbingly, it passes.

Over the ensuing weeks, the longer you spend working with the code the more questions you have. The architecture looks as though it is in the process of moving to a new

UI pattern. The architect behind it steadfastly refuses requests for meetings to explain it claiming it should be self-explanatory. The designers of the original code are long gone, so you're left to cobble together solutions that make use of both the original and the new patterns. You know that it's not the best that you can do, but you're scared to even suggest to the PM that it needs to be revamped.

Eventually, you're told that you will be taking over the test-fix-test cycle for one area of the application. Or as some developers call it: fix-release-pray-fix. Changes to the code have indeterminate impacts. Verifying any change requires a full release and testing cycle that involves the end users. The result is uncontrollable code, sliding moral and no confidence among the parties involved.

In the end your team makes the deadline, but at a cost. More developers have left for greener fields and lower stress. Management begins a process of imposing crippling accountability practices on the team. As you sit down to start on Phase 2, you pause to reflect on your initial optimism in the project. An e-mail notification interrupts your reverie:

Issue#3,787 Created: Window title incorrect.

It's not easy inheriting another team's project. If you haven't been in this situation already, clearly you are new to the industry and we welcome you into our midst. Whether you join a project as a replacement or as an additional developer, it is rare for someone to go through their entire career working solely on so-called "greenfield" applications. Even if you do, as figure 1.1 shows, it doesn't take long before a greenfield application starts to show signs of contamination.

<<<IMAGE DEPICTING A GREENFIELD TRANSITIONING TO A BROWNFIELD>>>

**Figure 1.1 Applications need constant tending to keep from becoming brownfield**

When you do start work on a project that has been around for any amount of time you will discover that it carries with it a certain amount of baggage. This baggage can come in many forms and no two projects seem to carry the same combination of "baggage intensity". There are a number of factors that can appear in projects as baggage. Poor attention to both initial and ongoing architecture and code design can contribute. So can slow or non-responding practices throughout the project team.

#### **OTHER POSSIBLE SOURCES OF BAGGAGE**

- Poor team, client, or management morale
- Steady turnover of development team members
- Recurring or lingering bugs

- Inadequate defect tracking
- Changing requirements and the team's inability to deal with them efficiently

Over time, existing members of the project team become numb to this baggage and begin accepting it as the norm. It usually isn't until a new person (or team) comes on board, or until a concerted effort is made by an existing team to challenge the assumptions made on the project, that any real change can be effected.

Presumably, you are reading this book because you are that new person or you are partially responsible for the concerted effort. As a new team member, often you will see things with a fresh eye that more readily identifies the baggage brought on by the project's history. As a current member of the team, you will still need to look at the project as if for the first time. As M.C. Escher's famous staircase in Figure 1.2 shows, a new perspective can often discover some very obvious problems.

<<<ADD IMAGE>>>

**Figure 1.2 Often a new perspective is all it takes to make you realize you aren't getting anywhere**

It's at this point that your brownfield experience truly begins. In this introductory chapter, we will define what it means to be a brownfield application and explain the three criteria we use to characterize them. We will also discuss the unique challenges facing a brownfield application including how to convince stakeholders to undertake a reworking of the application. Finally, we'll plot our course through the rest of the book.

Let's begin by defining the term that will drive the book.

## ***1.1 What is a Brownfield Application?***

Although the term "Brownfield" may not be a familiar one, the idea is all too common. Many people have heard of a greenfield application. That is an application you are starting from scratch with no history or previous version to guide you. It is, essentially, a blank slate. You can write in the language, data storage mechanism, and architecture of your choice.

"Brownfield" is not a new expression. It is used to describe an industrial or commercial site that may be contaminated by hazardous waste but has the potential to be useful once cleaned up. The key points are:

- it is an existing site,
- it is contaminated,
- but it can be improved upon and possibly re-used

So with our knowledge of greenfield applications and brownfield industrial sites in place, we can now formally define a brownfield application.

**Brownfield Application**

A brownfield application is a project, or codebase, that was previously created and may be contaminated by poor practices, structure, and design but has the potential to be revived through comprehensive and directed refactoring

Again, like the industrial definition, the key points are:

- Existing codebase
- Contaminated
- Potential for re-use or improvement

Figure 1.3 shows the three components and as you can see, they are inter-related.

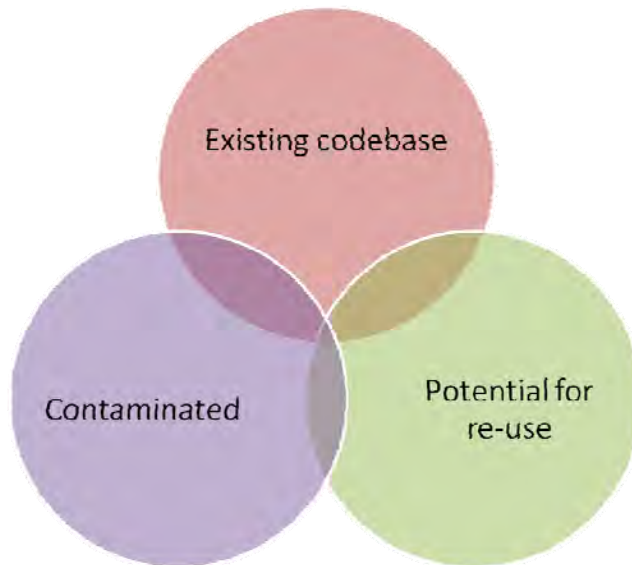


Figure 1.3 The components of a brownfield application

As a point of reference, brownfield applications fall between greenfield and legacy in almost all ways. (We'll come back to legacy codebases shortly.) Table 1.1 shows a comparison of the three types of projects.

Table 1.1 A comparison of the major project concerns and how they relate to greenfield, brownfield and legacy applications.

| <b>Concern</b>                   | <b>Greenfield</b>   | <b>Brownfield</b>  | <b>Legacy</b>   |
|----------------------------------|---|--|---|
| <b>Project State:</b>            | Early in the development lifecycle focusing on new features                                 | New feature development and testing and/or production environment maintenance occurring              | Primarily maintenance mode  |
| <b>Code Maturity</b>             | All code is actively being worked   | Some code is being worked for new development while all is actively maintained for defect resolution | Very little code, and only that required for defect resolution, is active |
| <b>Architectural Review</b>      | Reviewed and modified at all levels and times as the codebase grows                         | Only when significant changes (business or technical) are requested                                  | Rarely if ever reviewed or modified                                       |
| <b>Practices &amp; Processes</b> | Developed as work progresses  | Mostly in place, although not necessarily working for the team/project                               | Focused around maintaining the application and resolving critical defects |
| <b>Project Team</b>              | Newly formed group that is looking to identify the direction of its processes and practices | Mix of new and old bringing together fresh ideas and historical biases                               | Very small team which maintains the status quo                            |

Now that we've provided some criteria for a brownfield application, let's examine those criteria in more detail.

### ***1.1.1. Existing codebase***

One of the defining characteristics of a greenfield application is that there is little or no prior work. There is no previous architecture or structure to deal with. By contrast, brownfield projects are ones that have a significant codebase already in place.

This doesn't mean that you leave work after the first day and your greenfield application magically turns into a brownfield one overnight. Typically, it has been in development for some period of time (or has been left unattended) and now has some concrete and measurable work that needs to be done on it. Often, this work takes the form of a phase of a project or even a full-fledged project in and of itself.

Because this is an existing codebase, there is a chance it has already been released into production. Whether or not this is true will have some bearing on your project's direction but not necessarily on the techniques we'll talk about in this book. Whether you are responding

to user requests for new features, addressing bugs in the existing version, or simply trying to complete an existing project, the methods in this book still apply.

There are a couple of points worth mentioning here even if they are a little obvious. First, you must have access to the code being changed. If you are writing a wrapper around an existing third-party (or in-house) library for the sake of cleaning up the interface, that is different than what we are discussing here.



Secondly, you should be actively working on the code. An application that is sitting in your source code repository untouched does not a brownfield application make. Not that we want to discourage you from purchasing this book but you'll get more out of it when you are actually in a position to start checking out code.

These last two points disqualify certain types of applications from being brownfield ones. For example, an application that requires occasional maintenance to fix bugs or add the odd feature. Perhaps the company does not make enough money on it to warrant more than a few developer hours a week or maybe it's not a critical application for your organization. This application is not being actively developed. It is being maintained and falls more into the category of a "legacy" application.

Instead, a brownfield codebase is an "active" one. It has a team assigned to it and a substantial budget behind it. New features are considered and bugs tracked.

An existing codebase is not enough to define a brownfield project. One of the other characteristics is "contamination" which we'll discuss next.

### **1.1.2. Contaminated**

Besides being an existing, actively-developed codebase, brownfield applications are also defined by their level of contamination, or "smell". That is, how bad the code is. Of course, this is very subjective in the best of cases but in many cases, there is usually agreement that *something* is wrong, even if the team can't quite put their finger on it or agree on how to fix it.

There are different levels of contamination in any codebase. In fact, it is a rare application indeed that is completely free of bad code and/or infrastructure. Even if you follow good coding practices and have comprehensive testing and continuous integration, chances are you've accrued technical debt to some degree.

#### **Technical debt**

Technical debt is a term used to describe the, shall we say, "quainter" areas of your code. They are usually pieces that you know need some work but that you don't have the time, experience, or manpower to address at the moment. Perhaps you've got a data access

class with hand-coded SQL that you know is ripe for a SQL injection attack. Or each of your forms has the same boilerplate code cut and pasted into it to check the current user. Your team recognizes that it is bad code but are not able to fix it at the moment.

It is akin to the to-do list that is up on your refrigerator right now taunting you with tasks like “clean out the garage” and “find out what that smell is in the attic.” They are easily put off but need to be done eventually.

Along the same lines, “contamination” means different things to different people. You’ll need to fight the urge to call code contaminated simply because it doesn’t match your coding style.



The point is that you need to evaluate the degree to which the code is contaminated. If it already follows good coding practices and has a relatively comprehensive set of tests, but you don’t like the way it is accessing the database, the argument could be made that this isn’t a full-fledged brownfield application, merely one that needs a new data access strategy.

Bear in mind that every application can be improved upon. Often, all it takes is a new perspective. And like any home owner will tell you, you are never truly finished renovations until you leave (or the money runs out).

Our final criterion discusses how you plan to use the application.

### **1.1.3. Potential for re-use or improvement**

The final criterion is important. It means that your application is not only salvageable but that you will be making an active effort to improve it. Often, it is an application that you are still actively developing (perhaps, as mentioned earlier, you are even still working toward version 1). Other times, it is one that has been put on hold for some time and that you are resurrecting.

Compare this with a traditional “legacy” application. These applications are usually left alone and resurrected only when a critical bug needs to be fixed. No attempts are made to improve the code’s design or refactor existing functionality.

Projects that have aged significantly, or have been moved into maintenance mode, fall firmly into the “legacy” category, in our view. Such applications are not so much “maintained” as they are “dealt with”.

In his book, *Working Effectively With Legacy Code*, Michael Feathers uses a different definition of legacy code. There, he defines it simply as any code that does not have tests<sup>1</sup>. This is a good definition for the purpose of his book but just to clarify for anyone

---

<sup>1</sup> Page xvi of the Preface of *Working Effectively with Legacy Code*,



who is familiar with it, we are using a more traditional definition: really, really old code. This differentiation is important as we are big fans of Feathers' book and will refer to it occasionally throughout this book.

With this definition, it isn't hard to come up with an example of a brownfield application. Anytime you've worked on an application greater than version 1.0, you are two-thirds of the way to the core definition of a brownfield application. Often, even applications working toward version 1.0 fall into this category. As we've mentioned, it is rare that you will make it very far in a development career without encountering one.

Our next step is to crystallize some of the vague notions we feel when dealing with brownfield applications.

## ***1.2. Inheriting an Application***

In many cases, you may think that inheriting a brownfield application comes with a feeling of dread. The idea of working in a code base that has been thrown together haphazardly, patched up on the fly, and allowed to incur ongoing technical debt does not always inspire one to great heights.

So consider this section of the book your rallying cry. Let it not be said that the task ahead of you will be easy. Dealing with someone else's code (or even your own code, months or years later) rarely is, even if it does follow good design practices. And in many cases, you will be making changes that end-users, and even your own managers, do not notice. The person who will truly benefit from your work is either you or the developer who comes after you.

### **Reasons to be excited about brownfield applications**

- Business logic already exists in some form
- Can be productive from day one
- Easier to fix code than write it

But it is exactly the "contaminated" nature of the code that makes your task exciting. Software developers, as a group, love to solve problems. It's why we build software in the first place. And brownfield applications are fantastic problems to solve. The answers are already there in the existing code; you just need to filter out the noise and reorganize.

The nice thing about brownfield applications is that you see progress almost from day one. They are great for getting "quick wins". That is, expending a little effort for a big gain. This is especially true at the beginning of the project where a small, obvious refactoring can have a huge positive effect on the overall maintainability of the application. And right from

the start, we will be making things easier by encouraging good version control practices (Chapter 2) and implementing a continuous integration process (Chapter 3), two tasks that are easier to accomplish than you might think.

In addition, most developers find it much easier to “fix” code than to actually write it from scratch. How many times have you yourself looked at code you’ve done even six months ago and thought, “I could write this so much better now”? It’s human nature. Things are easier to modify and improve than they are to create. Because all the boring stuff (project set up, form layout) has already been done.

Compare this with greenfield applications where the initial stages are usually spent setting up the infrastructure for code not yet written. Or writing tests at a relatively granular level and agonizing over whether you should create a separate Services project to house that new class you are testing.

In short, brownfield applications should not be viewed as daunting. Quite the contrary. By their very nature, there is tremendous potential to improve them. And once you start improving an application, you’ll find it becomes very addictive.

But before we go any further, this is a good point to lay some groundwork for the task that lay ahead. To do that, we’ll talk about some concepts that will be recurring themes throughout the remainder of the book, starting with the concept of “pain points”.

### **1.2.1. Pain points**

A pain point is simply some process that is causing you grief. Typically, it is one that causes you to find a solution or alternative for it. For example, if every test in a certain area requires the same code to prime it, it may signal that the common code needs to be either moved to a Setup method or, at the very least, into a separate method.

Pain points come in all sizes. The easy ones are ones that can be codified and where solutions can be purchased. These include refactorings like Extract to Method<sup>2</sup> or Introduce Variable which can be done with a couple of keystrokes within Visual Studio.

The harder ones require more thought and might require drastic changes to the code. These should not be shied away from. After all, they have been identified as pain points. But always keep in mind the cost/benefit ratio before undertaking major changes.

#### **LIKE THE DOCTOR SAYS**

If it hurts when you move your arm that way, don’t move your arm that way.

One of the things that differentiates a brownfield application from a greenfield application is that pain points are more prevalent at the start. Ideally, pain points in

---

<sup>2</sup> page 110 of Refactoring: Improving the Design of Existing Code by Fowler, Beck, Brandt, Opdyke and Roberts

greenfield applications do not stick around long as they are removed as soon as developers encounter them. Or they are less noticeable because there isn't as much code. In brownfield applications, pain points have been allowed to linger and fester and grow.

Pain points are especially relevant to brownfield applications because they define areas in need of improvement. Without pain points, there is no need to make changes. It is only when you are typing the same boilerplate code repeatedly, or having to cut and paste more than once, or are unable to quickly respond to feature requests, etc, etc, and so on and so forth, that you need to take a look at finding a better way.

### **Be on the lookout for pain**

Very often, we are not even aware of pain points. Or we accept them as a normal part of the development process. For example, we might have a web application that restricts access to certain pages to administrators. In these pages, you've peppered your code with calls to `isAdmin()` to ensure regular users can't access them. But after you've made this call for the third time, your "hack alert" should trigger you to find out if this problem can be solved some other way (like, say, with a `<location>` tag in your `web.config`).

We will talk about pain points fairly often throughout the book. In fact, almost every piece of advice in it is predicated on solving some common pain point. Identifying those pain points is normally the result of some kind of productivity friction, which we'll discuss next.

## ***1.2.2. Identifying Friction***

Related to pain points is the idea of friction. Anything that gets in the way of a developer's normal process causes friction. Things that cause friction aren't necessarily painful but they are a noticeable burden and you should take steps to reduce it.

An example of friction is if the build process is excessively long. Perhaps this is due to integration tests dropping and re-creating a database (we'll cover automated builds and integration tests in chapters 3 and 4, respectively). Perhaps it is calling out to an unreliable web service. Whatever the reason, building the application is a regular event in a developer's day-to-day process and if left unchecked, this friction could add up to quite a bit of money over the length of the project.

Like pain points, friction may go unnoticed. You should constantly be on the alert when anything distracts you from your core task: developing software. If anything gets in your way, it must be evaluated to see if the process can be improved upon. Very often, we can get so wrapped up in our work that we forget computers can automate all but the most complex of tasks.

Noticing and acting on friction and pain points takes a clarity of thoughts that often subsides as your involvement in a project lengthens. As a result you have to explicitly expend effort thinking outside of your current box, or, as we'd like to say, challenging your assumptions.

### **1.2.3. Challenging your assumptions**

You are at a unique point in your application's development in that you are actively trying to improve it. So it behooves you to consider challenging the way you have thought about certain techniques and practices in the past in order to deliver on that promise.

#### **CHALLENGE YOUR ASSUMPTIONS**

Throughout the book, we will call attention to ideas that may be unintuitive or may not be commonly known. These will not be traditional pieces of advice that you should follow heedlessly. Rather, they should be taken as a new perspective. An idea worth considering that might fit in with your style or that might solve a particular problem you are having.

Examples of challenging your assumptions range from the mundane to the far-reaching. It could be as simple as raising your monitor to eye level to reduce strain. Or it could be re-examining your data access layer to see if it can be refactored to an object-relational mapper (see Chapter 11). Do you find yourself typing the same code over and over again? Maybe you can create a code snippet for it. Or maybe it's the symptom of some design flaw in your application. Whatever the case, developers, more than anyone, should recognize that doing the same thing over and over again is a sign that something can be improved.

Challenging assumptions is a good practice in any project but more so in brownfield applications. There is a good chance that some questionable assumptions led to the code being contaminated in the first place. Usually, bad assumptions are the root cause of pain points and friction.

But by the same token, there is a fine line between critical thinking and outright irreverence. You should be practical with your challenges. At all times, you should weigh the costs and benefits of any major changes you make. Although we developers would rather not consider it at most times, the expectations of clients and management need to play into our decision making process.

### **1.2.4 Balancing the client's expectations**

There is a common metaphor in the software industry: The Iron Triangle. It is depicted in Figure 1.4.



Figure 1.4 The Iron Triangle

One way the triangle has been summed up is: The project can be done on time, on budget, and feature complete. Which two do you want? The idea is that in order to increase the quality (i.e. the area of the triangle), one or more of the others will need to give. Conversely, if we reduce the resources, for example, without changing anything else, quality will suffer.

The Iron Triangle metaphor is flawed somewhat. One example is that if you reduce the scope of an application without reducing anything else, it implies the quality will go down.

Instead, we feel the relationship is more akin to a scale, as shown in Figure 1.5.



Figure 1.5 Generally speaking, your job will be to balance time and resources against code quality and features.

On the left, we have (estimated) time and resources, representing things we want to keep low. On the right exists quality and features, which we want to be high. The important thing to remember is that the scale must remain balanced at all times. So if you want to add a new feature, we need to add an appropriate amount of time and resources to keep the balance. If we lose some funding, then we have a decision to make: sacrifice quality or features in order to re-balance.

The development team has influence in all aspects of this equation except cost (without working for free of course). We have the ability to work in ways that will ensure that we provide the functionality that the client wants. We can also use different tools, techniques and practices that will enhance the quality of the application. Although more limited, the development team does have the ability to meet or miss deadlines. More effective though, is our ability to influence deadlines and set realistic and attainable expectations.

The relationship between Time & Resources vs. Quality & Scope is of particular interest in a brownfield application. This is because the scale needs constant adjustment as time goes on. As you progress, you may discover that your estimates were off and need adjustment. That is, you may not be able to physically deliver the application at the current scope and quality within the existing timeframe and budget. In effect, external factors have shifted the balance so that the left side of the scale is too light. You need to either increase the schedule or resources, or lower the quality or scope, in order to balance it out.

As developers, we are often focused on one aspect of the scale: quality. By contrast, project managers and clients typically need to balance all four. Brownfield applications, in particular, may require some radical decisions. Almost by definition, you will be affecting the quality of the code. As you outline the changes that need to be made, be sure you consider their impact on the scale, especially from the standpoint of your client.

By itself, balancing the scale is a complex task. Management courses and development methodologies exist by the dozens simply to try to address this task. With a brownfield there are added, yet subtle, complexities that come into play. Let's take a look at some of the challenges you'll face as you start your brownfield project.

## ***1.3. The Challenges of a Brownfield Application***

Your task will not be an easy one. There will be technical and not-so-technical challenges ahead.

Unlike a greenfield project, there's a good chance your application comes with some baggage attached. Perhaps the current version has been released to production and has been met with a less-than-enthusiastic response from the customers. Maybe there are developers who worked on the original codebase and they are not too keen on their work

being dissected. Or maybe management is pressuring you to finally get the application out the door so it doesn't look like such a blemish on the IT department.

These are but a few of the scenarios that will affect the outcome of your project. And unfortunately, they cannot be ignored. Sometimes they can be managed but at the very least, you need to be cognizant of the political and social aspects of the project so that you can try to stay focused on the task at hand.

We'll start the discussion off with the easy ones: technical factors.

### **1.3.1. Technical factors**

Almost by definition, brownfield applications will be rife with technical problems. The deployment process might be cumbersome. The web application may not work in all Internet browsers. Perhaps it doesn't print properly on the obscure printer on the third floor. You likely have a list of many of the problems in some form already. And if you are new to the project, you will certainly find more. Rare is the project that holds up well under scrutiny by a new pair of eyes.

In many ways, the technical challenges will be easiest to manage. This is, after all, what you were trained to do. And the good thing is that help is everywhere. Other developers, user groups, news groups, blogs, virtually any problem has already been solved in some format, save those in bleeding-edge technology. Usually, the issue isn't that a solution exists, the issue is finding one that works in your environment (and, it must be said, finding one that doesn't violate the more draconian corporate web filters; e.g. Planning Poker).

Although the range of technical factors that face you will vary from the simple to the exceedingly complex, you will be required to tackle them all. As stated earlier, the nature of our jobs is to solve problems. This is where you will come face-to-face with that reality.

One of the keys to successfully overcoming the technical factors that you inherit is to focus on one at a time and to prioritize them based on pain. Sometimes trying to solve all the technical issues on a project is impossible. More often, trying to solve many of them at one time is overwhelming. Instead of trying to take on two, three, or more technical refactorings at one time, focus on one and make sure to complete that task as best as you possibly can.

#### **STAY FOCUSED!**

When you start looking for problems in a brownfield application, you *will* find them. Chances are they will be everywhere. There will be a tendency to start down one path, then getting distracted by another problem. Call it the "I'll-just-refactor-this-one-piece-of-code-first" syndrome.

Fight this impulse if you can. Nothing adds technical debt to a project like several half-finished refactorings. Leaving a problem partially solved is more a hindrance than a

solution. Not only does the problem remain, but you have introduced inconsistency. This makes it that much harder for the next developer to figure out what is going on.

Working on some technical factors will be daunting. Looking to introduce something like Inversion of Control and Dependency Injection (see Chapter 9) into a tightly coupled application can be an overwhelming experience. This is why it is important to bite off pieces that you can chew completely. Occasionally, those pieces will be large and require a significant amount of mastication. That is when it's important to have the motivation and drive to see the task through to completion. For a developer, nothing is more rewarding than completing a task, knowing that it was worthwhile and, subsequently, seeing that the effort is paying off.

Unfortunately, technical problems are not the only ones you'll face. Unless you work for yourself, you will at least need to be aware of political issues. We'll discuss this next.

### ***1.3.2. Political factors***

Whether you like it or not, politics play a factor in all but the smallest companies. Brownfield applications, by their very nature, are sure to come with their own brand of political history. By the time an application has become brownfield, the politics can become pretty entrenched.

Political factors can take many forms depending on the nature of the application, its visibility within the organization, and the personality of the stakeholders. And it doesn't take much to create a politically-charged situation. The application could be high-profile and several months late. It could be in production already but so unstable that the customers refuse to use it. The company may be going through a merger with another company that makes similar software. All of these situations will have an effect on you as shown in Figure 1.6.

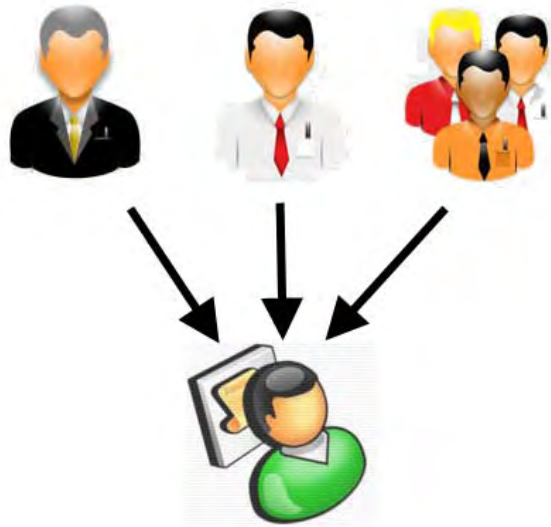


Figure 1.6 Many parties have a stake in the application. And each one will have different interests. It helps to know the political dynamics even if you can't affect them too much

Political factors, regardless of the category you assign a project to, exist mostly at a macro level. Rarely do politics dip into the daily domain of the individual programmer. Programmers will, however, usually feel the ramifications of politics indirectly.

One of the most direct manifestations is in the morale of the team, especially the ones that have been on the project longer. Perhaps you have a highly skilled senior programmer on the team who has long stopped voicing her opinions out of resignation or frustration. Alternatively, there is the not-so-skilled developer who has enough political clout to convince the project manager that the entire application's data storage mechanism should be XML files to make it easier to read them into typed datasets. Sure, you could make the case against this, and possibly win, but the root cause (i.e. the misinformed developer) is still running rampant through the project.

Another common example is when management, project sponsors, or some component of the organization has soured to your project and they have decided, rightly or wrongly, to point the blame at the technology being used. Being the implementers of the technology, the developers are often dragged into the fray. In these cases, we are almost always on the defensive, having to explain every single decision we make, from the choice of programming language to the number of assemblies we deploy.

Regardless of the forces causing them to influence a project, politics are the single most difficult thing to change when on the project. They are usually based on long-standing corporate or business relationships that have a strong emotional backing to them and this is difficult to strip away. It is nearly impossible to meet these situations head on and make changes.

Dealing with political situations is never an easy undertaking. Our natural tendency is to hide in the code and let others worry about the political landscape. Indeed, if you're able to safely ignore these problems, we highly recommend doing so. However, we have yet to encounter a situation in a brownfield application where we've been able to disregard them completely. And in our experience, burying your head in the sand tends to make things worse (or at the very least, it certainly doesn't make things better).

Instead, subtlety and patience are often the best ways to address politically charged situations. Tread lightly at first. Like working in code, learning the pain points is the key to success. In the case of negative politics, learning what triggers and fuels the situation sets the foundation for solving the problem. Once you know the reasoning for individual or group resistance, you can begin appealing to those points.

### **Get it done with fewer defects**

One of the projects that we worked on had a poor track record of releasing new features that were littered with defects and/or missed functionality. This had continued to the point where management and clients intervened with the mandate that the team must "...get it done with fewer defects." And they were right, we had to.

At the time, one of our first suggestions was to implement a policy of Test Driven Design (TDD) to force the development team to spend more time defining the requirements, designing their code around those requirements and generating a set of lasting artifacts that could be used to verify the expectations of the requirements. As is often the case when proposing TDD to management, the added coding effort is perceived as a drain on resources and a problem for meeting schedules or deadlines. The idea that doing more work ensuring successful implementation instead of more after the fact work fixing defects didn't compute well with them.

As we backed away from the (heated) conversations, it became quite apparent to us that there was a historically based set of prejudices surrounding TDD. Past experience and the ensuing politics surrounding a very visible failure were creating a great deal of resistance to our ideas.

Instead of selling a technique known as Test Driven Design, we proposed a combination of practices that would allow us to address the quality problems that the project was seeing. With the implementation of these practices by the development team, delivered quality improved and management's confidence in our abilities increased.

To this day we're not sure if that management team understood that we were doing TDD on their project, but that didn't matter. The political charge around the term was so high that it was never going to succeed when packaged in that manner. Research into the reasoning behind that level of negativity pointed to a past implementation and execution experience. By re-bundling the practices, we were able to circumvent the resistance and still deliver what was requested of us by management.

In the end, your goal in a project that has some political charge is not to meet the politics directly. Instead you should be looking to quell the emotion and stay focused on the business problems being solved by your application. Strike a happy balance between groups who have interests that they want addressed. You'll never make everyone fully happy all the time, but if you can get the involved parties to rationally discuss the project, or specific parts of the project, you are starting to succeed.

That was kind of a superficial, Dr. Phil analysis of politics but this is a technical book, after all. And we don't want to give the impression that we have all the answers with regard to office politics ourselves. For now, we can only point out the importance of recognizing and anticipating problems. In the meantime, we'll continue our talk on challenges with a discussion on morale.

### **1.3.3. Team morale**

When you first start on a brownfield project, you will usually be bright-eyed and bushy-tailed. But the existing members of that team may not share your enthusiasm. Because this is a brownfield application, they will have some history with the project. They've probably been through some battles and lost on occasion. In the worst situations, there may be feelings of pessimism, cynicism and possibly even anger among the team members. This type of morale problem brings with it project stresses such as degradation in quality, missed deadlines, and poor or caustic communication.

It is also a self-perpetuating problem. When quality is suffering, the testing team may suffer from bad morale. It is frustrating to feel that no matter how many defects you catch and send back for fixing, the returned code always has a new problem caused by "fixing" the original problem. The testing team's frustration reflects on the developers and the quality degrades even more.

Since every team reacts differently to project stresses, resolving the issues will vary. One thing is certain though. For team morale to improve, a better sense of "team" has to be built.

One of the most interesting problems that we've encountered is Hero Programmer Syndrome (HPS). On some teams, or in some organizations, there are a few developers who will work incredible numbers of hours to meet what seem like impossible deadlines. Management loves what these people do. They become the 'go-to guy' when times are tough, crunches are needed, or deadlines are looming.

It may seem counter-intuitive (and a little anti-capitalist), but this should be discouraged. Instead of rewarding individual acts of heroism, look for ways to reward the team as a whole. While rewarding individuals is nice, having the whole team on board will usually provide better results both in the short and long terms.

### **WE DON'T NEED ANOTHER HERO**

The project will run much smoother if the team feels a sense of collective ownership. That is, the team succeeds and faces challenges together. When a bug is encountered, it is the team's fault, not any one developer. When the project succeeds it is due to the efforts of everyone, not any one person.

One of the benefits you get from treating the team as a single unit is a sense of collective code ownership. No one person feels personally responsible for any one piece of the application. Rather, each team member is committed to the success of the application as a whole. Bugs are caused by the team, not a team member. Features are added by the team, not Paul who specializes in data access and Anne who is a whiz at UI.

When everyone feels as if they have a stake in the application as a whole, there is no "well, I'm pulling my weight but I feel like the rest of the team is dragging me down." You can get a very quick sense of how the project is going by talking to any member of the team. If they use the words "we" and "our" a lot, their attitude is likely reflected by the rest of the team members. If they say, "I" and "my", you have to talk to other members to get their side of the story.

HPS is of particular importance in brownfield applications. A pervasive individualistic attitude may be one of the factors contributing to the project's existing political history. And if you get the impression that it exists on your team, you have that much more work ahead of you moving toward collective code ownership.

And it is not always an easy feat to achieve. Team dynamics are a topic unto themselves. At best, we can only recognize that they are important here and relate them to brownfield applications in general.

Unfortunately, the development team is not the only ones you have to worry about when it comes to morale. If your team members are disenchanting, imagine how your client feels.

### **1.3.4. Client morale**

It's easy to think of morale only in terms of the team doing the construction of the software. As developers we are, after all, in touch with this group most. But we are also heavily influenced by the morale of our clients. Their feeling towards the project and working on it will ebb and flow just like it does for a developer and the team in general.

Because the client is usually outside of our sphere of direct influence, there is little that we can do to affect their overall morale. The good news is that the biggest way we can affect their mood is simply doing what we were trained to do: build and deliver software.

When it comes down to it, clients have fairly simple concerns when it comes to software development projects. They want to get software that works the way they need it to (without fail), when they need it, and for a reasonable cost. This doesn't change between projects or between clients.

The great thing for developers is that many of the things that will make working in the code easier will also address the concerns of the client. Introducing automated testing (see Chapter 4) will help to increase quality. Applying good OO principles to the code base (Chapter 7) will increase the speed that changes and new features can be implemented with confidence. Applying agile principles, such as increased stakeholder/end user involvement, will show the client first hand that the development team has their interests at heart.

Any combination of these things will increase the morale of the client at the same time as, and possibly correlated to, increasing their confidence in the development team. The increased confidence that they have will significantly reduce friction and increase communication within the team. Those are two things that will positively influence the project's success.

And to relate this again back to brownfield applications, remember that the client very likely is coming into the project with some pre-conceived notions of the team and the project based on very direct experience. If there is some "bad blood" between the client and the development team, you have your work cut out for you. In our experience, we've found very constant and open communication with the client can go a long way to re-building any broken relationships.

Whatever bad blood has happened in the past, you should make every attempt to repair damaged relationships. Don't be afraid to humble yourself. In the end, it is helpful to remember that you are there for your client's benefit, not the other way around.

So far, we've psychoanalyzed our team and our client. That just leaves the management team in the middle. Let's talk about how they feel.

### ***1.3.5. Management morale***

Thus far we've discussed the morale of the team and the client. In between those two usually sits a layer of management. These are the people that are tasked with herding the cats. Like everyone else, they can fall victim to poor morale. From the perspective of the developer, management morale problems manifest themselves in the same way that client morale problems do.

Management is, in effect, another client for the development team. They have deliverables that they expect from the development team. Like a traditional client, the development team has influence over the morale of the management team. And like a

traditional client, the best general advice we can give to help ensure management stays happy is to build quality software and keep the communication lines open.

That covers everybody. In the next section, we're going to meet some of the people you might encounter on your project.

### ***1.3.6 Social landscape***

It's time to dispel some myths about software developers. Gone are the days when a programmer can hole himself up in a cubicle/basement/garage getting by on pizza and carbonated beverages. Hollywood may not have received the memo yet but we need to be just as adept at navigating the social aspects of a project as the technical ones.

To that end, we're going to have some fun with this topic. Over the years, we've both encountered a number of "archetypes" on various projects. These are people in various roles who throw up roadblocks in one form or another. See how many you can recognize. Better yet, pick out the ones that apply to you.

#### ***The Ivory Tower Architect***

Sporadically swooping into meetings with the rest of the team from his namesake home, the Ivory Tower Architect loves to show his mastery of software design. Whether it works in practice or not matters not to him. The rest of the development team is left to implement the grandiose plans handed down from on high, but the Ivory Tower Architect will never work directly on the code himself. Often the designs are overly complex and time consuming to implement, but he will not change them. To do so undermines his very existence.

Dealing with the Ivory Tower Architect requires a strong will, the willingness to have long winded and abstract design debates, and an almost fanatical adherence to the KISS (Keep It Simple Stupid) principle. Lucky for you the Ivory Tower Architect will only rarely grace mortal developers with his presence.

#### ***The Disenfranchised Client***

Beaten down by months of missed deadlines, misunderstood requirements and repeated defects, the Disenfranchised Client has lost all faith in the team's ability to deliver and is merely going through the motions until the next budget cycle. Because of this and because she pays the bills, she is one of the most dangerous archetypes. To counter her, you'll need strong social skills to assure her that her needs are being met, though it may be weeks until you see any progress.

What is the best way to deal with a disenfranchised client? Give her working software, dammit!

### *The Absentee Client*

When the project starts the Absentee Client engages the team just long enough to build up some velocity. As soon as he perceives some progress being made, the project is quickly re-prioritized to a much lower level in his calendar. The team will wait days or weeks to hear back from The Absentee Client about the simplest of clarifications. Important meetings will be rescheduled numerous times at his request and when he does attend they will have to be cut short due something else important having come up.

One way to get the Absentee Client's attention is to outline how significantly the lack of engagement is increasing the timeline and the risk to the project. Your best bet is to put the effects of The Absentee Client's actions into terms of cost. Unfortunately, and this should not be undertaken lightly, sometimes the only way to do this is to address the Absentee Client's superior.

### *The Process-Heavy PM*

This person is one of the most feared by developers around the world. While the team is working to deliver software, he is asking them to write action reports and detailed impact analysis documents and any number of other reports he needs to grease his process and document pacifier. Though able to bend Outlook and Microsoft Project to his will, the simplest of changes, like re-scheduling a recurring meeting, will throw the Process-Heavy PM into a fetal position. It will take him days to recover from the disruption to his carefully scripted master plan for the project. Suggest that there is no need to type up the minutes for a meeting and watch the blood drain from his face as if you have just sounded the project's death knell.

While the Process-Heavy PM can be (in our twisted minds) the most fun person on a project to toy with, he will be the one that will, more often than not, have you writing summary documents after your call to order pizza for the team. To counter him, be sure you account for every single second you spend on his process when you fill out the intricately-detailed timesheet he has requested from you. Caution: Read Chapter 5 (Code metrics) with care when dealing with a Process-Heavy PM.

### *The Disinterested Developer*

Though you will usually see the Disinterested Developer working diligently whenever you walk by, you also notice she has a well-known social networking site constantly minimized on his desktop. Over time, you see a pattern: social networking tools when no one is thought to be around, development tools when there is. During meetings the Disinterested Developer is looking out the window or checking her mobile phone. She doesn't engage the rest of the team on a technical level. She answers only when required and delivers the minimal amount of work necessary to keep the project manager happy. Her heart either isn't in software development or she's bored with the project.

The Disinterested Developer isn't necessarily a bad person. Not everyone is as keen on software as you are (right?). But care should be taken to ensure she carries her weight. The last thing you need on a brownfield project is a team revolt because she is perceived as getting special treatment. Your best bet is a heart-to-heart with the developer to see if there is a reason behind her ennui. Give her every chance to turn things around. It may sound mercenary, but if there is no improvement, don't discount removing her from the project as an alternative.

### *The Skeptic*

Every time that the team attempts to implement a technique, process or technology that will address a project problem and better the team's ability to deliver to the client, the Skeptic will interject. If it's unknown to the Skeptic, then he must speak out about his doubt. Often the doubts are small and founded in his complete lack of knowledge. Regardless, he will stand by them vigorously until the project has discussed (at least three times) the situation.

The discussion will not, however, satisfy the Skeptic. If he does agree to move forward, it will be under very vocal protest. He will end his part of the discussion by stating that he "still isn't sure that [he's] fully convinced of any benefit". Regardless of how the project moves forward, the Skeptic will always be the first person to point out any small flaw in something that he opposed. He was, after all, trying to warn the team of the potential problems ahead of time. And now it's your fault for not listening back then.

If you're lucky, The Skeptic can be ignored. But if he has some political clout, you may need some allies in the form of other developers. Focus on convincing the other developers that what you are doing is correct.

IMPORTANT NOTE: Just like you are not always right, The Skeptic is not always wrong. Don't discount his arguments just because you always have before.

### *The "Experienced" Developer*

Every project needs experienced people to improve the odds of succeeding. Skilled developer resources are hard to come by so you're really excited to be joining a team that has some "Experienced" Developers. But it doesn't take long on any project to realize that experience is a relative term. When asking for help, the "Experienced" Developer's questions are often at the same level as some of the junior developers on the team. When he does propose a solution, and has it turned down, he immediately plays the "I have 12 years of experience in this industry. I think I know what I'm doing." As a result, he will be a ferocious, if not weakly armed, foe when arguing the merits of a situation.

On his own, the "Experienced" Developer is usually an easy person to handle. A simple "show me" is enough to counter him. Eventually, he will stop playing his "number of years in the industry" card and come around.

### *The Front of the Magazine Architect*

Once a month, every month, she visits the team and blurts out "We should use/do/implement *<insert technology of the month here>*." The monthly rhythm of the visits is tied directly to the delivery of the Front of the Magazine Architect's magazine subscriptions and RSS feed. Anything that is boldly emblazoned across the cover of an IT related periodical becomes that month's flavour.

The Front of the Magazine Architect is another easy one. While they are annoying distractions, her visits will be infrequent and short lived. As quickly as the idea enters her head, it will leave and you won't hear from her until the next month. If the same technology is mentioned two months in a row, it's probably time for you to contact the magazine editor and request that they do a better job covering new technologies.

### *The Hero*

Everyone loves a hero. The PM, the architects and the client relish the long hours he puts into delivering results. When the client is told we don't have the budget or manpower to add a feature, the hero's cubicle is his first stop after the meeting. "Old man Baley says we can't have this. But we NEED it." The Hero hums and haws and complains how badly the project is being managed, then with a sigh says, "I'll put it in. But this is the LAST TIME."

The Hero then proceeds to circumvent your entire development architecture wedging the feature in because he doesn't understand terms like "budget" and "resources". All he cares about is getting his ego stroked and being the martyr that saved the project. The long hours he puts in are heralded by the PM and the client who don't realize his effort is not directly correlated to the value he is delivering.

Project Managers and Clients will scoff at you when you make claims against their Hero. In their mind, he is a cornerstone in the project and whose absence will wreak havoc on the success of the project. Regardless of their actual ability, Heroes are often more trouble than they're worth.

### *The Ex-Tech PM*

Monday morning and the Ex-Tech PM appears on the edge of the status meeting as an observer to the team's daily ritual. One by one the developers tell of their current work and its state. As one developer mentions that there will be some design work in his day ahead, the Ex-Tech PM interjects with "We used to do *<something>* when I was an RPG developer. You should look into doing that too." The team continues through the standup and when completed the Ex-Tech PM corners the developer to provide more details about his idea.

Over the next few days the developers come up with a solution (different from the PM's, naturally) and it is mentioned at the morning status meeting. On hearing this, the Ex-Tech PM becomes adamant that the team didn't put enough consideration into his idea and the results of his project five years ago justify the same solution be used here and now.

Luckily, the Ex-Teach PM is easily appeased. A simple “yes, you’re right, dear” will keep him at bay until he has long forgotten the issue.

### *The Enhancing Tester/QA*

Usually found in the confines of an organization that has heavily silo’d roles and responsibilities, the Enhancing Tester will be assigned responsibility for ensuring the product quality. She believes it is her personal responsibility to question and alter any specifications that were used in creating the software. Since she wasn’t involved at the start of the development cycle, the Enhancing Tester will question the design and requirements only after the development team has passed them on for test. The proposed ‘enhancements’ are usually obscure and with far reaching architectural ramifications. For example, “this really should be an MDI application.”

Since she understands the original requirements, but not the overall business, the Enhancing Tester has no choice but to log these as software bugs so that they will get the attention of the team. After working with her for a few months you will wake up in a cold sweat yelling “It’s not a bug, it’s a feature change!”

Usually, you can counter this by assigning cost estimates to the “bug fixes”. It helps to do so in front of the client.

### *The ‘Oooo...Shiny!’ Developer*

An incestuous cousin to the Front of the Magazine Architect, this developer is easily distracted by any new technology. Not only will he want to talk about it endlessly, the ‘Oooo...Shiny!’ Developer will simply add the technology to the project without telling anyone. You will find, scattered through the code base, a number of different techniques, tools or frameworks that are used one time and then abandoned. While adding to your technical debt, the ‘Oooo...Shiny!’ Developer is working feverishly to keep adding new entries to the “Experience Using” section of his resume.

Sometimes it is easy enough to counter his predilection for new and shiny simply by placing a pretty glass bead on their keyboard every morning. When that fails, it’s time to up the priority of the code reviews for The “Oooo...Shiny!” Developer. And be merciless.

### *The Possessive Developer*

During your first week on the project you’re assigned to have a mentor who has written a large portion of the existing application. While working on your first serious defect in the system, you ask the Possessive Developer about an existing piece of code and suggest refactoring. She looks back at you and states “There’s no reason to change that code. It works just as I want it to.” After explaining its deficiencies, the Possessive Developer is sullen and in a less-than-cordial mood. At the end of the conversation she has neither agreed nor disagreed with the suggested changes, but there is tension in the air.

The following morning the Possessive Developer corners you at the water cooler and lashes into a list of reasons that there should be no changes to the code discussed the previous day. At the end of the one-way conversation she walks away with confidence in her stride. You now know. It is her code, her creation and her domain. Only the Possessive Developer can state when her code is to be changed.

Like the “Oooo...Shiny!” Developer, code reviews with peers can be useful. In a more neutral forum, it’s much harder to argue against the majority. But don’t ignore The Possessive Developer. It doesn’t take long to turn her into a Skeptic or a Hero.

### ***The Over-Protective DBA***

A good many application require access to a database. If you’re lucky, you’ll have free rein over the database to make whatever changes you deem necessary. If you’re unlucky, you’ll need to make those changes through an Over-Protective DBA.

The Over-Protective DBA protects his database with an iron fist. Requests for changes to a stored procedure go through several iterations to ensure they include the standard corporate header and naming conventions. He also challenges every single piece of code in the procedure to see whether you *really* need it. Only when satisfied that the application can’t be deployed without it will he grace the database with your changes. In the development environment, at least..

If you really want a battle on your hands, suggest to an Over-Protective DBA that you should switch to an object-relational mapper (see Chapter 11). Be prepared to launch into a prolonged debate on the performance of stored procedures vs. dynamic SQL, the dangers of SQL injection, and the “importance” of being able to deploy changes to business logic without re-compiling the application.

The Over-Protective DBA often has company policy on his side so he will be a challenge. Don’t spend a lot of time confronting him head-to-head. Your database is an important part of your application, it behooves you to get along with him. Instead, arm yourself with knowledge and talk to him in common terms. In our experience, DBAs can often be negotiated with for certain things, such as an automated database deployment (see Chapter 3).

We hope you didn’t recognize too many of these people, but the fact is that they all exist in some form out in the wild. Let’s close off this section with a discussion on change.

### ***1.3.7. Being an agent for change***

As a software developer, you likely don’t have a problem embracing change at a personal level. After all, change is one of the defining characteristics of our industry. The challenge will be effecting change on a broader level in your project.

This being a brownfield application, change *must be* inevitable. Clearly, the path that led to the code base being a brownfield application didn't work too well and the fact that you are reading this book probably indicates that the project is in need of some fresh ideas.

Not everyone has the stomach for change. Clients dread it and have grown accustomed to thinking that technical people are only concerned with the "cool new stuff". Management fears it because it introduces risk around areas of the application that they have become comfortable with assuming are now risk free. Testers loath it as it will add extra burden to their short term work load. Finally, some developers reject it because they're comfortable having "always done it this way." (See Challenging Your Assumptions earlier in this chapter.)

Even with all these barriers to change, people in each of those roles will be thankful that change has taken place after the fact. We do assume that the change was successful, and, just as important, that the team members can see the effects of the change. If change happens and there is nothing that can be used for comparison, or nothing that can be trotted out in a meeting that indicates the effect of the change, then it will be hard for team members to agree that the effort required was worth the gain received.

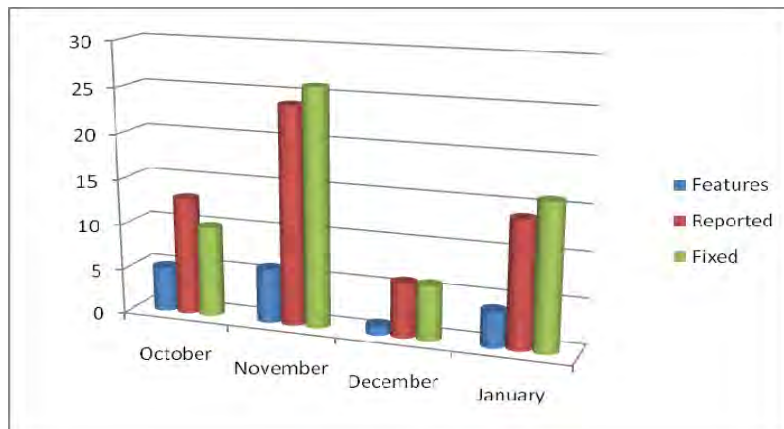


Figure 1.7 Before starting any change, be sure you can measure how successful it will be when you're done

As a person who thinks of development effort in terms of quality, maintainability, timeliness and completeness, you have all the information and tools to present the case for change, both before and after it has occurred. Remember, you will have to convince people to allow you to implement the change (whether it is in its entirety or in isolation) and to get the ball rolling. You will also be involved in providing the proof that the effect(s) of the change were worth the effort. You have to be the agent for change.

This will come with experience but there are a couple of tactics that you can try. The first is "it is better to beg forgiveness than ask permission." Now, we're not espousing

anarchy as a way of getting things done but if you can defend your actions and you feel the consequences are reasonable, simply going ahead and, say, adding a unit test project to your application can get things rolling. If your decision is easily reversible, that's all the better.

Also, be efficient. Talk only to the people you need to talk to and tell them only pertinent details. Users don't care if you want to implement an object-relational mapper (see Chapter 11). If you aren't talking about cash dollars, leave them and your VPs out of the conversation.

### **Challenge Your Assumptions: Making Change**

When talking about working on poorly performing projects, a developer we both highly respect said to us "Change your environment, or change your environment." That is, if you can't change the way things are done in your organization, consider moving on to a new environment.

Sometimes you will find environments where, no matter how well you prepare, present and follow through on a proposed change, you will not be able to implement it. If you can't make changes in your environment you're not going to make it any better, and that is frustrating. No matter how change-resistant your current organization may be, you can always make a change in your personal situation. That could be moving to a different team, department, or company. Regardless of what the change ends up being, make sure that you are in the situation that you can be most comfortable in.

Acting as an agent is similar to how a business analyst or client works for the business needs. You have to represent the needs of your project to your team. But change doesn't just happen; someone has to advocate it, to put their neck on the line (sometimes a little, sometimes a lot). Someone has to get the ball rolling. You can, and should, be that person.

If you're joining a brownfield project, you're in the perfect situation to advocate change. You are looking at the code, the process and the team with fresh eyes. You are seeing what others with more time on the project may have already become numb to. You may have fresh ideas about tools and practices that you bring from other projects. You, above all else, can bring enthusiasm and energy to a project that is simply moving forward in the monotony of the day-to-day.

As we've mentioned before, you will want to tread lightly at the start. First, get the lay of the land and find (or create) allies within the project. More than anything, you will want to proceed with making change.

During our time talking about this with different groups and implementing it in our own work we have come up with one clear thought: the process of change is about hearts and minds, not shock and awe. Running into a new project with your technical and process guns blazing will alienate and offend some of the existing team. They will balk at the concepts simply because you're the "new person". Negativity will be attached to your ideas because

the concerned parties will, without thought, think that they are nothing more than fads or the ideas of a rogue or cowboy developer. Once you've entered a project in this fashion it is extremely difficult to have the team think of you in any other way.

<<<IMAGE: Hearts and minds vs. Shock and awe>>>

**Figure 1.8 You'll find more success winning over hearts and minds, rather than using shock and awe.**

"Hearts and minds" is all about winning people over. Take one developer at a time. Try something new with a person who is fighting a pain point and, if you're successful at solving the problem area you will have a second advocate spreading ideas of change. People with historical relevance on the project usually carry more weight than a new person when proposing change. Don't be afraid to find and use these people.

Being an agent of change is about playing politics, suggesting ideas, formulating reasoning for adoption and following through with the proposals. In some environments, none of these things will be simple. The problem is, if someone doesn't promote ideas for change, they will never happen. That's where you (and, we hope, this book) come in. One of the common places that you'll have to sell these changes is to management. Let's take a look at what can happen there.

## ***1.4. Selling it to your boss***

There may be some reluctance and possibly outright resistance to making large-scale changes to an application's architecture and design. This is especially true when the application already "works" in some fashion. After all, users don't typically see any noticeable difference when you refactor. It's only when you are able to deliver features and fix defects that much quicker that the efforts get noticed.

Often, managers even recognize that an application isn't particularly well-written but are still hesitant. They will ease developers' concerns by claiming that "we'll refactor it in the next release but for the time being, let's just get it done."

### **The "Technical Debt" Phase**

As mentioned in Section 1.1.2 (Contaminated), technical debt is the laundry list of items that eventually should to get done but are easy to put off. Every project incurs technical debt and many projects deal with it through a "big push" phase (if they deal with it at all). This is some major undertaking where all work on the application stops until you have worked through at least a good chunk of your technical debt.

This is not the way to go. Instead, we promote the idea of a time-boxed task every, or every other, coding cycle (whether that be development iterations or releases to testing). The sole purpose of this task is to pay down some technical debt.

This doesn't have to be a team-wide task. It may take only one person working half a day a week. And it is limited by time, not by tasks. That is, the person working on technical debt works on it for the allotted time, then stops, regardless of how much is left on the technical debt list. (Of course, the person should finish what she's started and plan accordingly.)

This sentiment is understandable and may even be justified. Just as there are managers who ignore the realities of technical debt, many developers ignore the realities of basic cost/benefit equations. As a group, we have a tendency to suggest changes for some shaky reasons:

- You want to learn a new technology or technique
- You read a whitepaper/blog post suggesting this was the thing to do
- You don't want to take the time to understand the current codebase
- The belief that doing it another way will make all problems go away and not introduce a different set of issues
- You're bored with the tasks that have been assigned to you

So before we talk about how to convince your boss to undertake a massive rework in the next phase of the project, it's worth taking stock in the reasons why we are suggesting it. Are we doing it for the client's benefit? Or is it for ours?

The reason we ask is that there are many very valid business reasons why you shouldn't undertake this kind of work. Understanding them will help put your desires to try something new in perspective:

- Project's shelf life is short
- The application actually does follow good design patterns, just not ones you agree with
- The risk associated with the change is too large to mitigate before the next release
- The current team or the maintenance team won't be able to technically understand the changed code and need training first
- Significant UI changes may risk alienating the client or end user

### **WARNING!**

Be cautious when someone suggests a project's shelf life will be short. Applications have a tendency to outlive their predicted retirement date. It may be a good idea to get some

form of guarantee that, if the application does continue to exist after the specified end date, real money will be put aside to get it into a stable state. (If that eventuality does happen, try to hold your tongue.)

## NOTE

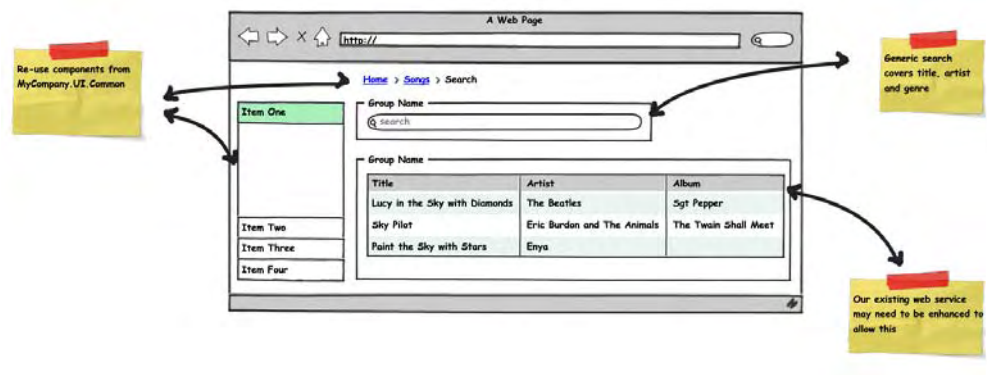
Note that lack of time and money are two very common reasons given for choosing not to perform a large-scale refactoring. Neither one is valid unless the person making the decision outright admits they are hoping for a short-term gain in exchange for long-term pain.

You will not have an easy time trying to justify the expense of large-scale changes. It's hard to find good solid information on this that is written in a way meant to convince management to adopt. Even if you do find a good piece of writing, it's common for management to dismiss the article on the grounds that the writer doesn't have a name known to them, or that the situation doesn't apply.

By far, the best way to alter management's opinion is to show concrete results with clear business value. Try to convince them to let you try a prototype in a small and controlled situation. This will give them the comfort that if the trial goes sour, they will have protected the rest of the project and not spent a lot of money. And if it goes well, they now have the data to support their decision to go ahead.

For these same reasons, you should not be looking at the proof-of-concept solely as a way to convince your management that you are right. You should always consider that your suggestion may not be the best thing for the project and approach the trial objectively. This allows you to detach yourself from any one technology and focus on the success of the project itself. If the experiment succeeds, you now have a way of helping the project succeed. If it fails, at least you didn't go too far down the wrong road.

In Figure 1.9, we show something more akin to a mock-up but with notes on how it can work within our existing infrastructure.



## Figure 1.9 Proofs-of-concept and mockups can help convince your boss of the value of a concept and can help you work out kinks before implementing it large-scale

Another advantage of proofs-of-concept is that because you're constrained to a small area, you can be protected from external factors. This will allow you to focus on the task at hand and increase your chances to succeed. This is your ideal goal.

When discussing change with decision-makers, it is a good idea to discuss benefits and costs in terms of cash dollars whenever possible. This may not be as difficult as it first sounds.

Previously, we discussed the concept of pain points and friction. This is the first step in quantifying the money involved: isolate the pain point down to specific steps that can be measured. From there, you have a starting point from which you can translate into business terms.

There are two kinds of cost savings you can focus on. The first is direct savings from being able to work faster. That is, if you spend two hours implementing a change and that change saves half a day for each of four developers on the project, the task can be justified.

Another way to look at this method of cost savings is as an opportunity cost. How much money is the company foregoing by not completing a particular cost-saving task?

The second method of cost savings is what keeps insurance companies in business: How much money will it cost if we don't implement a certain task and things go wrong? That is, can we spend a little money up front to reduce our risk exposure in the event something goes horribly wrong? This is one of the reasons for implementing version control on your application.

It's time to get into the technology. Chapter 2 starts at a seemingly innocuous part of the process: Version Control.

## **1.5 Summary**

Like so many in software development, you will spend more time on projects that are incomplete or in ongoing development than you will on projects that aren't. It's much more common for projects to need our services once they've been started and resource shortages or inadequacies have bubbled to the surface.

With these projects, we inherit all of the problems that come with them. Problems will include technical, social and managerial concerns, and you will, in some way, be interacting with all of these. The degree of influence that you have over changing them will vary depending on both the project environment and the position that you have on it.

Remember that you probably won't solve all of the problems all of the time. Don't get wrapped up in the magnitude of everything that is problematic. Instead focus on little pieces where the negative impact on the project, at whatever level, is high and where you are in a

position to make or direct the change. These pain points are the ones that will provide the biggest gain.

Positive change works wonders on the project, no matter what the change is. Remember to look at pain points that have cross-discipline benefits. Implementing a solution for a problem that is felt by developers, management, testers and the clients will have positive influence on each of those disciplines individually. More importantly, positive cross-discipline improvements will increase the communication and rapport between the groups that may exist within the project team.

No matter the effect that we, or you, have said that a change will have on a project team, there will still be people who resist. Convincing these people will become a large part of the work that you do when trying to implement a new technique or process onto the project. One of the most effective ways is to sell the change using terms and conditions that will appeal to the resistor. If part of their role, as in management, is to be concerned with money and timeline, then sell to those points. For technical people, appeal to the sense of learning new techniques and bettering their craft.

Even with the tactical use of this technique, you will often run into people and topics that take more selling. If you feel like you're at a dead end, ask for a time-boxed task where you can prove the technique or practice on a small scale and in an isolated environment. Often, results from these trial or temporary implementations will speak volumes to the person that is blocking their use.

As a developer the changes that you make to a project, whether permanently or on a trial basis, will fall into two different concerns: Ecosystem and Code. No matter how well schooled a project is in one of those two areas, deficiencies in the other can still drag it down. Don't concentrate on one over the other. Instead focus on the areas in the project where the most significant pain points are occurring. Address those first and then move onto the next largest pain point.

As we step through this book, you will see a number of different categories that pain points will occur in. While we have tried to address these in a specific order, and each builds on the previous, your project may need an order of its own. This is fine. Adapt what you see in the book and the sample project to provide the best results for you.

With that, let's move to solving pain points.