

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

Contents

Introduction

Part 1 The ecosystem

Chapter 1 Understanding Brownfield Development

Chapter 2 Source control systems

Chapter 3 Continuous integration

Chapter 4 Automated testing

Chapter 5 Metrics and static analysis

Chapter 6 Defect tracking

Part 2 The code

Chapter 7 Bringing better OO practices to the project

Chapter 8 Layering

Chapter 9 Inversion of control

Chapter 10 Object relational mappers

Chapter 11 Model view

Chapter 12 Working with other systems

Chapter 13 Keeping the momentum

Appendix Tools

1

Understanding Brownfield Development

In this chapter, we will:

- Define what it means to be a Brownfield application
- Discuss the challenges facing a Brownfield application
- Talk about how to convince your boss to take on a Brownfield application
- Plot our course through the rest of the book

It's not easy inheriting another team's project. If you haven't been in this situation already, you almost certainly will be at some point in your career. Whether you join a project as a replacement or as an additional resource, it is rare for a developer to go through their entire career working solely on so-called "greenfield" applications.

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. It 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.

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 at least 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. If in the latter category, you will still need to look at the project as if for the first time.

It's at this point that your Brownfield experience truly begins. In this introductory chapter, we'll define what a Brownfield application is and discuss the challenges you may face when working on one.

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.

From this definition (and likely from the blurb on the book cover), you can probably tell what a Brownfield application is. But before we define the term formally, let us first take a short look at its history.

“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

As a point of reference, Brownfield applications fall between Greenfield and Legacy in almost all ways.

Concern	Greenfield	Brownfield	Legacy
Project State:	Early in the development lifecycle focusing on new features	New feature development and testing and/or production environment maintenance occurring	Primarily maintenance mode
Code Maturity	All code is actively being worked	Some code is being worked for new	Very little code, and only that required for

		development while all is actively maintained for defect resolution	defect resolution, is active
Architectural Review	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
Practices & Processes	Developed as work progresses	Mostly in place, although not necessarily working for the team/project	Focused around maintaining the application and resolving critical defects
Project Team	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

Table 1.1 shows a comparison of some of the major project concerns and how they relate to Greenfield, Brownfield and Legacy applications.

1.1.1. Existing codebase

The main differentiating point between a Brownfield application and a Greenfield application is that the code already exists in some form. This doesn't mean that you leave work after the first day and your Greenfield application magically turns into a Brownfield one. Rather, it is an existing codebase that has been left for some period of time and now has some concrete and measurable work that needs to be done on it. Typically, 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 slightly different than what we are discussing here.

Secondly, you should be actively working on the code for a period, whether it be days, weeks, months, or years. A .NET application that is sitting in your source code repository untouched does not a Brownfield application make. Just because you dangle the carrot in front of your developers that you will "refactor that mess soon" doesn't mean the rest of us

believe you. 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 LOB application. This application is not being actively developed. It is being maintained and falls more into the category of a "legacy" application.

1.1.2. Contaminated

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.

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 infected. 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 Brownfield application. Or at least not a full-fledged one that requires a full-on project phase to improve it.

NOTE:

It is good to bear in mind that every application can be improved upon. Often, all it takes is a new perspective. And as with home ownership, you are never truly finished renovations until you leave (or the money runs out).

1.1.3. Potential for re-use or improvement

The final criteria is important. It means that your application is not only salvageable but that you will be making an active effort to improve it.

There is another implication with this point: your code is not necessarily "legacy" code in the traditional sense. That is, a Brownfield application is not one written in COBOL thirty years ago. Typically, these applications are left alone for the most part and resurrected only to fix critical bugs. 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. This definition of "legacy" differs significantly from Michael

Feather's definition which would include any code that does not have automated tests. Such applications are not so much "maintained" as they are "dealt with".

For this book you can, in general terms, consider a Brownfield application as one written in .NET (any version) that is hard to work with but that you have a vested interest in improving in the near term.

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.

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 codebase 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 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.

List 1.1 Some 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 and implementing a continuous integration process.

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, business logic – OK, we’re kidding about the last one) 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:

- Pain Points
- Friction
- Challenging Your Assumptions

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 fixture requires that you type in the same code to prime it, it may signal that the priming 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 Method¹ 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.

¹ page 110 of Refactoring: Improving the Design of Existing Code by Fowler, Beck, Brandt, Opdyke and Roberts

One of the things that differentiate a Brownfield application from a Greenfield application is that pain points are more prevalent at the start. Ideally, pain points in Greenfield applications do not stick around long as they are removed as soon as developers encounter them. 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.

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.

1.2.2. 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. 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.

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.

1.3. Challenges

Let it not be said that your task will 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 are not too keen on their work being dissected (and worse, some of those developers may be on your team). 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.

Let's take a look at some of the challenges you'll face in more detail.

1.3.1. Technical factors

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).

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 we 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. Sometimes trying to solve all the technical issues that you find 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. 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. Or at the very least, give it strong consideration to make sure it is absolutely necessary. Nothing adds technical debt to a project like several half-finished refactorings.

And finish what you start. Taking on the task of fixing a technical problem and leaving it partially completed is more a hindrance than a solution. A partially completed refactoring

adds to the technical debt of a project you have introduced inconsistency. The refactoring may be technically and theoretically perfect, but the inclusion of two methods to solving a problem (the original version and the half-finished refactored version) adds a significant point of questioning for other developers working in the codebase.

Working on some technical factors will be daunting. Looking to introduce something like Inversion of Control and Dependency Injection 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.

1.3.2. Political factors

Whether you like it or not, politics play a factor in all but the smallest of companies. And Brownfield applications, almost by their very nature, are sure to come with their own brand of political history.

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.

That's not to say politics don't roam the technical realm directly. One common example is when management, project sponsors, or another 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.

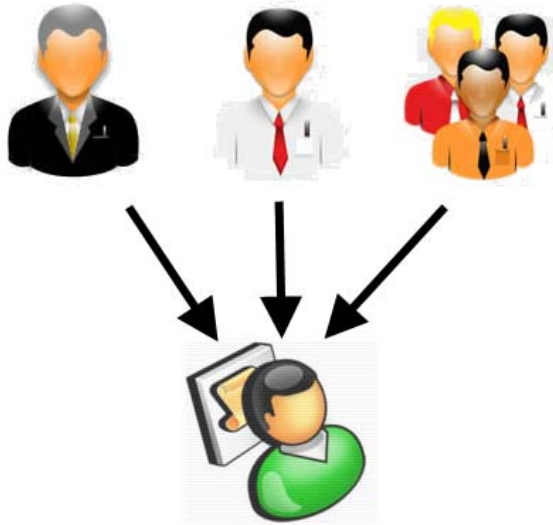


Figure 1.1 Many parties have a stake in the application. And each one will have different interests. It helps to know the political dynamics even if you can't affect them too much

Regardless of the forces causing them to influence a project, politics are the single most difficult thing to change when on the project. Corporate or business relationships that are negatively affecting a project usually have a significant emotional backing to them that is difficult to strip away. It is nearly impossible to meet these situations head on and make changes. Subtlety and patience are often the best ways to address them.

In any situation, it's best if you enter the project treading lightly in the flower beds. 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.

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. Like politicians, you are looking to strike a balance in happiness 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.

1.3.3. Team morale

When you first start on a project as a developer, you will usually be bright-eyed and bushy-tailed. But the existing members of that team may not share your enthusiasm. 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.

Likewise, constant pestering by management usually has the exact opposite effect on achieving unattainable deadlines.

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 code 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.

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 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 THREE ATTRIBUTES OF A SOFTWARE PROJECT

A common question asked of clients requesting software is: The project can be done on time, on budget, and with high quality. Which two do you want?

This is amusing mostly because it has traditionally been true, especially in Brownfield applications where any one (or two or three) of these attributes has probably failed. One of your goals will be to reverse this traditional notion of software projects.

The development team has significant 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 great thing for developers is that many of the things that will make working in the code easier will also help to better address the concerns of the client. Introducing automated testing (unit and/or integration) will help to increase quality. Applying good OO principles to the code base 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 a very constant and open communication with the client can go a long way to re-building any broken relationships.

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 involved in the software development process, 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 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.

1.3.6. Being an agent for change

As a software developer, you likely don't have a problem embracing change at a personal level. It is one of the defining characteristics of the industry.

This being a Brownfield application, change is inevitable. Clearly, the path that led to the codebase 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 else 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."

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 change has successfully altered the item that it was supposed to, and 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.

As a person who thinks of a 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) 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.

Challenge Your Assumptions: Making Change

When talking about working on poor performing projects, JP Boodhoo said to us "If you can't make changes you should make a change." 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 for something is similar to how a Business Analyst, client or client proxy works for the business needs. You have to represent the needs of your project to your team. The thing is change doesn't just happen. Someone has to advocate it. Someone has to put their neck on the line (sometimes a little, sometimes a lot). Someone has to get the ball rolling. You can 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. You'll want to get the lay of the land (so to speak). You'll want to 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.

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 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 come in.

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 that application already "works" in some fashion. After all, users don't typically see any noticeable difference when you refactor, unless it is specifically to address performance concerns or recurring bugs.

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."

NOTE: TECHNICAL DEBT

Technical debt is code and/or architecture that have been put together without adequate thought or concern. There are numerous reasons that this may have occurred, and every project incurs technical debt as it grows. What we commonly find is that technical debt is left in a project and is never addressed. Instead the team works around it and ignores the fact that it should be fixed. We promote the idea that there should be a time-boxed task every few coding cycled (development iterations or releases to testing) that has the sole purpose of paying down technical debt. This doesn't have to be a task worked on by the entire team. It may only take one person a few hours. Certainly the time required will be kept to a minimum by regularly doing paying down the technical debt.

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 more 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
- 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 due 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.

Let it not be said that you will have an easy time trying to justify the expense of large-scale changes. It's hard to find good solid information 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.

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 something 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.

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.

1.5. Planning your path

With our psychology discussion behind us, we can now start actively thinking about the task at hand. This section will talk about the path the rest of the book will take and the rationale behind it.

1.5.1. Setting up the ecosystem

In Part I, we will take a look at the project's ecosystem. That is, what can we do to set up a good infrastructure for when we eventually start diving into the code. There are three things we need to review and/or set up before we can start working with code:

- Chapter 2: Version control
- Chapter 3: Continuous integration
- Chapter 4: Automated tests
- Chapter 5: Metrics & Static Analysis
- Chapter 6: Defect Tracking

In the next chapter, we'll look at setting up your project in a Version Control System (VCS). Chances are, your project is already in a VCS, but even if it is there are still some things you will want to check, like a labeling strategy and how you can set up the folder structure to minimize friction for developers.

Chapter 3 discusses Continuous Integration (CI), whereby we create a process where the application is compiled and tested several times a day on a relatively clean machine. We'll see how easy it is to get a project up and running in a CI environment and how it is an integral part of maintaining a stable codebase.

The next chapter will be automated testing. That is, how can you set up a unit testing framework for your application and get into the habit of ensuring new code is adequately covered by tests. We'll also talk about the role integration testing can play in generating

additional confidence in our code and talk about how we can test those pesky UIs without involving an actual user.

NOTE

Although we discuss them separately, the first three topics are inter-related. For example, the continuous integration process will need to retrieve code from the VCS for compilation and the tests need to be automated so that they can be executed within the continuous integration process.

In Chapter 5, we'll get into some mildly dangerous territory. But that's what makes this chapter fun! We'll talk about different ways of measuring your code to check for things like duplication, class coupling, and test coverage. We'll get you hooked on the nearly infinite ways you can analyze your code followed by a stern warning on reading too much into the results.

The final chapter in Part I covers defect tracking. This may seem to be extraneous but consider that a major portion of many Brownfield applications deals with managing defects. Having an efficient way to track them is an important step in ensuring the team continues adding value to the project. If nothing else, read this chapter for the horror stories.

1.5.2. Working with the code

Part II will be all about the meat and potatoes of the application: the actual code. Throughout this section of the book, we will look at very pointed methods of refactoring code to make it easier to fix bugs and add features. Specifically, we'll discuss:

- Chapter 7: Pimping your OO skillz
- Chapter 8: Layering the application
- Chapter 9: Inversion of Control and Dependency Injection
- Chapter 10: Object/Relational Mapping
- Chapter 11: Model View Controller and Model View Presenter patterns
- Chapter 12: Interfacing with other systems
- Chapter 13: Keeping the momentum going

Chapters 7 through 11 cover various ways of modifying the code to make it easier to maintain in some manner. We'll go into specific design patterns and refactoring methods, borrowing heavily from our experiences as well as experts in this space such as Martin Fowler and the Gang of Four.

In Chapter 12, we'll talk about a common scenario, especially in Brownfield applications. That is, how can we effectively integrate with applications, web services, and other systems that we don't control?

Finally, after all our hard work setting up the ecosystem and modifying our code, we'll end with another rallying cry to encourage you to keep the fire burning.

Refactoring/rework vs. ongoing development

Working in isolation

Working in the fray

Stay focused on one refactoring at a time

1.6. The sample application

We haven't talked code too much in this chapter but don't be alarmed. We have an entire sample application waiting for us starting in the next chapter. Until then, you'll have to be content with a description of it.

The application was written for Bitter Sweet, an online restaurant delivery service. The company that built it has arrangements with several restaurants across the country and provides a menu with selections from each one. In addition, they have a fleet of delivery personnel available to collect the food and deliver it.

When a hungry user hits the website, the menu is presented and the food is ordered. Payment could be taken at the time of the order or when it is delivered. The relevant restaurant is notified of the order and a driver is dispatched to pick it up. The driver delivers the order and collects payment if necessary.

For the less-technological but equally hungry users, a toll-free number is provided. The customer service representative at the other end uses the same application to place the order but with some added features such as driver tracking.

The application was written in ASP.NET 2.0, using C#, by two junior developers guided by an architect. Like many Brownfield applications, it reflects the good intentions and inexperience of the developers. Thanks to the initial guidance of the architect, it does have some code separation between layers with the UI, business logic, and data access in their own assemblies. In addition, there is a Common project that holds helper methods used by one or more layers.

The code is available for download at XXXX and it is accessible via Subversion. Successive versions of it (one for each chapter) are stored as branches. Appendix XXX contains a brief tutorial of Subversion and TortoiseSVN (a GUI client for Subversion) and how to use them to access the code.

1.7 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.

We inherit these projects and 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 still will 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. Likewise, this can be done with technical people as well.

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're going to want to 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.