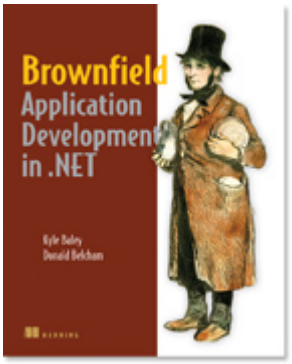
What is Brownfield Development?



By Kyle Baley and Donald Belcham, May 2008

This green paper is taken from the forthcoming book Brownfield Application Development in .NET by Kyle Baley and Donald Belcham (Manning, 2008). This 550-page manual shows you how to approach legacy applications with the state-of-the-art concepts, patterns, and tools you've learned to apply to new projects. Using an existing application as an example, this book guides you in applying the techniques and best practices you need to make it more maintainable and receptive to change.

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 baggage. There are a number of factors that can create baggage on a project, including

- Poor attention to both initial and ongoing architecture and code design
- 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 paper 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 paper, we'll define what a Brownfield application is and discuss the challenges you may face when working on one.

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, 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.
- It can be improved 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: *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, as shown in Table 1.

Table 1: A comparison of some major project concerns and how they relate to Greenfield, Brownfield and Legacy applications.

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	Some code is being	Very little code, and

	being worked	worked for new development while all is actively maintained for defect resolution	only that required for 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

Let's examine the three facets of the Brownfield definition, starting with the existing codebase.

Existing codebase

The main differentiating point between a Brownfield application and a Greenfield application isn't only that the code already exists in some form, but that it has been left for some period and now needs substantial concrete and measurable work. 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. 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 of Brownfield development 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 our book, but you'll get more out of it when you are actually in a position to start committing code changes to the source code repository.

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 Line of Business (LOB) application. This application is not being actively developed. It is being maintained and falls more into the category of a *legacy* application.

Regardless of the effort that is being expended on a code base at any given time, the code can have problems lurking in it. This brings us to the next aspect of a Brownfield application which is the idea of *contamination*.

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. 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).

Once we've recognized that there is a level of contamination in our code our development instincts kick in and we begin to formulate ways to remove the sore spots. It's this desire to continually improve code quality that leads us to the last component of a Brownfield application which is that the application is salvageable in some manner.

Potential for re-use or improvement

This final criteria is important. It means that your application is not only salvageable, but that you will be making an active effort to substantially 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 this legacy category. Such applications are not so much maintained as they are dealt with.

NOTE

Our version of legacy code differs significantly from Michael Feathers' definition who, in *Working Effectively with Legacy Code* (Prentice Hall, 2004), defines it as "code without tests." The actual nomenclature isn't important to our discussion. The salient point is that we differentiate between existing code that is still actively developed and existing code that is resurrected occasionally solely for maintenance.

For this paper 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. However, it should be noted that Brownfield applications are certainly not limited to .NET.

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.

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. 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.

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.

Now that we've got the easy challenge out of the way, we'll turn our attention to ones that, for many developers, can be much harder to manage.

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 have decided, rightly or wrongly, to point the blame at the technology being used (see figure 1). Being the implementers of the technology, the developers are often dragged into the fray.

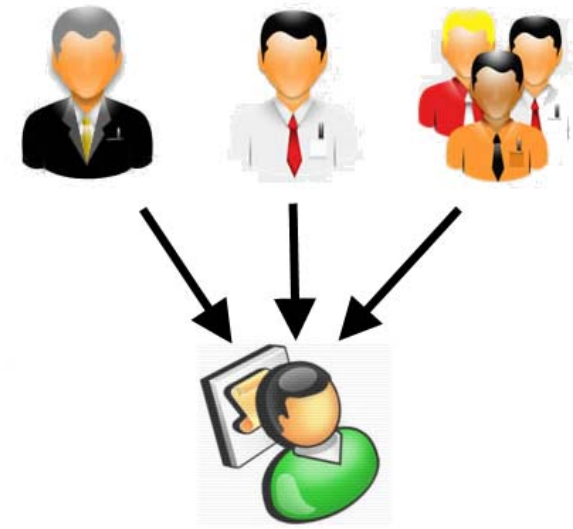


Figure 1 Many parties have a stake in the application. And each one will have different interests. It helps to know the political dynamics even if you can't affect them too much

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 analysis of politics but this is a technical paper, after all. And we don't want to give the impression that we have all the answers with regard to office politics ourselves. We'll forge on with team morale.

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. In the meantime, it's time we discussed our client.

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.

Developers and clients are not the only people involved in repairing project relationships. An integral group that is often perceived to have more influence on a project is management.

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.

That was a quick run-down of the social engineering aspect of a Brownfield project. Before we go into how to sell the changes to your boss, we still need to discuss one last piece in the puzzle: you.

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 paper 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 accustom 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 is the most comfortable for you.

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.

It's 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.

That's all aspects of the team covered: you, the team, the client, and your management. Now, we come to an important part of any Brownfield application: how can you convince the stakeholders of the benefits of change.

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. What they notice is new features, fixed defects, and introduced 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."

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 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 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:

- The 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.)

Lack of time and/or money is not an argument

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.

By sweeping problems under the rug, you are essentially arguing that you are choosing to sacrifice quality in favor of coming in on time and on budget. Depending on the corporate culture, this may be acceptable to the stakeholders, though it shouldn't be. If it is, your job will be much more difficult.

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 awry? This is one of the reasons for implementing version control on your application.

Having brought everyone on board to effect positive changes, we now turn our attenion to an important detail: what do we do now?

Getting Started

When you sit down at your desk on the first day of your project, you may have a sense of "Okay, I'm here. Now what?"

This is a healthy question to ask, even if you've worked on other Brownfield applications. No two projects will be the same, and although there are some general guidelines that will help, be mindful of possible tweaks due to differing team dynamics, client and management morale, and project history.

When creating your Brownfield to-do list, there are a number of options for Task 1. We'll discuss them now.

Analyze the build process

In our opinion, this is probably the best bet for your first step. In it, you examine exactly what needs to be done in order to build your application. This may seem an obvious thing. After all, how much is involved in pressing Ctrl+Shift+B in Visual Studio?

There is so much more to a build than just the simple compilation of the application. The build process will tell you about the project team's tolerance for mundane and repetitive tasks. Manual repetition of tasks is a ripe target for automation. As much as possible, your build process should be automated, to the point where a new developer can retrieve the latest version of the code from version control and double-click a batch file to build the application in its entirety.

Automating your build has benefits other than getting new developers up and running quickly. By examining how your application is built, you get a sense of its architecture as well as its dependencies:

- Does the build order make sense?
- Are you building your data access project before your domain project?
- Are third-party controls required in order to compile?

These are the types of questions you will encounter while reviewing how the application is built.

Finally you may be able to get insight into the deployment process. Similar to architectural insights, deployment can tell you why an application is layered (physically and/or logically) in the way it is. Information about layering is invaluable when you begin to refactor the code base.

Call a meeting with various stakeholders

A meeting can have a few benefits. First, it will help you get up to speed on the nature of the project, especially if you include stakeholders that have been on the project for some time. Second, it will help you assess the morale of the stakeholders and the team so that you have a better idea how to work with the different personalities. Finally, it can help the existing stakeholders collect their thoughts on what has worked well in the past and what could use some improvement.

Review existing documentation

As we know from experience, when we are working on projects the documentation of code usually plays second fiddle to delivering business functionality. While some people will stand up and proclaim that code documentation is the panacea that will ensure ease of ongoing maintenance, we rarely step onto projects where that is possible.

What usually happens when you ask to see a project's documentation (architectural, code, operational or otherwise) is that you're presented with a virtual stack of paper. Along with the documents you're sometimes told that "…these haven't been updated for a while

and we've made some changes since then". When you're not explicitly told that, we've found that it's best to assume that the statement is implied.

If documentation is available, don't dismiss it. Read it over starting at the highest level view of the system and working your way down to the code level. Don't take the contents of the documents for granted though, and always ask other people to clarify when you have any doubts.

Review existing bug reports and/or feature requests

Examining open issues on a project could help determine potential areas in need of special attention. For example, if there are a lot of bugs in one area of the application, this could be a candidate for early refactoring. If there are a lot of requests for similar features, this could indicate where the client would like to focus the development effort.

Be aware of how soon you begin to use this technique. While it is invaluable, it is most useful if you have a decent understanding of the business domain and the driving force behind the client's use of the system. Knowing what the client intends to do and how they would like to do it can explain and elaborate on the bugs that are currently logged in the system.

Dive into the code

As a software developer, this will probably be your natural instinct. But you should try to fight it early on. There will be plenty of time to code the longer you're on the project. Your initial concern should be at a higher level. For example, is the team using an adequate version control system? Is it easy for new developers to start working with the code quickly? Does the code "break" often?

When you do start looking at the code, a good place to start is with the unit and/or integration tests, assuming any exist and that they are functional. If there are a decent number of tests in the project, they will give you a sense of how the application is structured. If there aren't many tests, this is still helpful to know so that you have an idea of what needs to be done.

Pair with another developer

If the application is being actively developed, it might be a good idea to work with another developer who is working on it regularly. Not only will you get up to speed faster than working on your own, but few developers can resist the opportunity to "tell it like it is," so you get some insight on the political situation for the project.

When pairing, don't sit idly behind the experienced developer and listen. Instead, be an active participant in the development process. Ask questions about the techniques and implementations being used. Insight on previous decisions and current pain points can surface from the most innocuous question about a small snippet of code.

Pair programming is, essentially, a variation on diving into the code. Like diving straight into the code, be aware that this probably is not something that you will want to do too early

in your tenure on the project, though it is an invaluable practice once you actually do begin coding

What's Next?

After reviewing the code, documentation and build process you still aren't quite ready to dive into the code yet. You still have a number of things that you'll want to review in your development ecosystem before you start to effect change in the code itself. These include:

- Are you making the most out of your version control system? Can developers check in and check out painlessly?
- When and how often should you check in? By feature? By passing unit test? (Hint: the answer is once a day at a bare minimum.)
- Is everyone notified immediately if there is a problem in the code? Is there a continuous integration process in place?
- Do you have a plan in place to run tests against the code? Does it include automation?
- Are there plans in place to review code metrics at regular intervals and to act on them accordingly?
- Is there an effective defect management process in place?

Only after these questions are answered can you actually start getting on with what you were formally trained to do; that is, you can start writing code.

While this may seem a lot of infrastructure, it is all based on a very simple goal: Giving the developers as friction-free an environment as possible so that they can focus on the code. You want to keep them in a good rhythm and not distract them with environmental issues. If they are constantly fighting with the version control system, or if they are consistently introducing errors that aren't discovered for days, that is time spent not writing code. Worse, it is time spent encouraging workarounds and bad code.

This is the path that moves a project into the realm of Brownfield. You're already working on a project that has these problems. Part of the work to move the project out of trouble is to prevent bad habits from forming when they can easily be prevented.

Conclusion

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 your 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 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 into 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 the 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've witnessed on projects that we've worked on, even implementing some of these suggestions can significantly change the overall mood of a project. The key to getting the biggest return for any change that you try to implement is finding the largest pain point that the project is facing. If you can implement changes that minimize or, better yet, completely eliminate the pain points, the project will quickly shift for the better.