Ruby for the Nuby

Pete McBreen

E-mail: pete@mcbreen.ab.ca

May 03, 2002

(A very early rough Draft)

Preface

Since writing the Software Craftsmanship book [McBreen, 2001] I've had many people ask me how can they learn how to be a software developer. I'm very dubious about the effectiveness of "sheep dip" courses, and it is hard to find an appropriate introductory book that explains how to become a software developer. Overall I still think that Ted Nelson had the best idea when he wrote:

The best way to start programming is to have a terminal running an interactive language, and a friend sitting nearby who already knows the language. and has something else to do but can be interrupted with questions. And you just try stuff. Till more and more you get the feel of it. And you find yourself writing programs that work. [Ted Nelson, "Computer Lib/Dream Machines", 1974, p.???]

Having someone right there with you who can instantly answer your questions makes learning much, much easier. Indeed I'm often tempted to speculate that the "terminal rooms" of the 1970's and early 1980's made it much easier for developers of that era to learn from each other compared to the modern practice of isolating each developer in their own cubicle.

For most people it is difficult to re-create all of Ted Nelson's kind of learning environment. Luckily however, it is possible to create a much better technical environment than Nelson had. We can use the power of modern scripting languages to create a much better interpretive environment, and while it might be hard to find an experienced friend to sit with, learning alongside another beginner can help ease the frustrations that come with to write software.

Why Software Craftsmanship?

Becoming a good software developer involves a lot more than just learning to write programs. Software development is a craft, it blends science, engineering, mathematics, linguistics and art.

- Science is an important part of software development – you will spend a lot of your time thinking about cause and effect.
- Engineering tradeoffs happen all of the time – you want the application to be blazingly fast and to support hundreds of users over the Internet.
- Mathematics shows up in number crunching and conditional logic.
- Linguistics becomes important when you realize that other people have to be able to understand your application, both to be able to use it and to be able to improve on it.
- Art is the most important because great software is beautiful.

The craft of software development blends all of these in a unique and ever changing manner. Getting good at software development involves mastering the aesthetics of software.

Why an Introductory Book?

There are lots of books that have been written for experienced developers covering all of the different aspects of software development. Unfortunately most of these are not very accessible to beginners as they assume a lot of prior knowledge. They are great books for developers with some experience, it is just that they are not a good place to start.

The book is not encyclopedic. Although encyclopedic books make good doorstops, personally I do not find them very easy to read. Unlike most introductory books, this one is relatively slim. It is deliberately focused on the most important topics that a developers should know before they can effectively contribute as part of a small team. Many interesting topics have had to be omitted, but there are many suggestions for extra reading scattered throughout the book.

Object Oriented Development

Over the years there have been many different approaches to software development, some more successful than others. Object oriented development is easier than most because it conceptually groups information with the code that processes that information.

Classes and objects represent the concepts in your application. Over the course of this book we will create a small application for a Running Club. The Running Club is an object in the application, as are the Members of the Club and the Results they get in various Races. Chapter 3 explains Object Oriented concepts in detail.

Ruby as a First Language

Yukihiro Matsumoto, a.k.a Matz, the creator of Ruby, describes the many attractions of Ruby on the website.

> *Ruby is the interpreted scripting language for quick and easy object-oriented programming. It has many features to process text files and to do system management tasks. It is simple, straight-forward, extensible, and portable.*
>
> *Oh, I need to mention, it's totally free, which means not only free of charge, but also freedom to use, copy, modify, and distribute it.* [http://www.ruby-lang.org/en/whats.html]

Ruby has many advantages as a first language, most notably that it can run interactively so beginners do not have to worry about compiling and then running their code. Initially it can be as simple as typing programs statements at the command prompt and getting immediate feedback as each statement is processed.

Admittedly, Ruby will spoil you for other, less pleasing programming languages that you will learn later on, but starting out easy is always a good idea.

Software Development is Not Always Easy But it Can Be Fun

Software development is much more than simply programming. Developers who have really mastered the craft understand how to

- work with their users and customers to figure out what is really needed,
- document and plan the delivery of the needed application features,
- design an application so that it can be extended as new requirements are identified,
- create a pleasing, easy to learn, productive user interface,
- write well documented and tested application code,
- work with the users to ensure the application is thoroughly tested,
- support the users in their use of the application,
- maintain and enhance the application, and
- fix mistakes as they are discovered.

Some of this may come naturally to you, but I can more or less guarantee that you are going to have to work hard at some of these activities. At the same time though I can promise that if you are cut out to be a software developer, you will find many of these activities to be extremely if not addictively enjoyable.

Points To Remember

- Object Oriented Development is just one way to develop applications – you can easily learn other approaches once you are familiar with Objects.
- Ruby is just one programming language – you will learn lots of different ones as you learn more about software development.
- The first programming language you learn sets many of your overall habits of software development.

Chapter 0: Read Me First

The goal of this book is to help you learn the craft of software development. In order to learn you need to practice the craft. That means you have to actively read this book, play with the examples and experiment with the different concepts. The best way to read this book is to sit down at a computer and try variations on the examples that are given.

Even better, try making subtle typos when you play with the examples and see what happens. Recovering from mistakes is a normal part of software development, so you might as well get used to it from the start. Sometimes the messages you get back will be kind of cryptic;

```
irb(main):006:0> 1 = i
SyntaxError: compile error
(irb):6: parse error
1 = i
   ^
        from (irb):6
```

Learning what these strange messages mean is just part of the process. This is the reason why Ted Nelson suggested that you should have someone else on hand who can answer your questions – a partner can act as your proofreader, spotting your mistakes for you.

Becoming a Software Developer Takes Time

Although it is possible to create useful applications quite quickly, becoming a software developer takes a significant time. Despite all of the books and training courses that seem to promise that learning will be quick and easy, that is not actually the way it works.

This book is not about "making it look easy." The intention behind the book is to reveal the complexity and frustrations underlying the seemingly easy job of developing software. As such this book breaks away from the traditional beginner books in that it does not explain everything in detail. I think that anyone who is really interested in becoming a software developer has already played around with computers and software quite a lot. You don't really need me to explain what a command prompt is, how to change directories or how to edit a file.

What, a programming book without a CD?

Yes, I know it sounds strange, but there is a reason. Ruby is an evolving language with new and improved versions being released all of the time. Rather than restrict you, the reader to using an older version that is frozen in time on the CD, this book allows you to use the latest version from the Internet (http://www.ruby-lang.org/).

Now is the time to visit the main Ruby website http://www.ruby-lang.org/ and download a copy of Ruby. Later on you will probably want to download the source for Ruby, but for now just pull down an installable version. A windows installation is available from Dave Thomas and Andy Hunt's Pragmatic Programming website http://www.pragmaticprogrammer.com/ruby/downloads/ruby-install.html. Once you have installed it test your installation by getting Ruby to display it's version number.

```
D:\>ruby -v
ruby 1.6.6 (2001-12-26) [i586-mswin32]
```

I won't insult your intelligence by describing how to install Ruby. A character trait that all good software developers have is persistence and the ability to figure out how to get things done in the face of minimal instruction. If you are not able to install it successfully then you probably need to become more familiar with computers before attempting to become a software developer. All detailed hand-holding instructions would do is postpone the point at which you discover whether you have enough persistence to overcome the inevitable frustrations of software development.

An Introduction to Software Development

This book is an introduction to software development for people who have never written software and are new to the field of software development. It covers everything from when someone first dreams up an idea for an application, through designing, developing, testing and releasing the application as well as supporting, maintaining and enhancing applications.

As this book is intended to be a slim volume, many subjects have been omitted, glossed over or simplified. What this book does give you however is the necessary vocabulary for discussing all of the important software development activities and the opportunity to practice the essential ones on a small project of your own choosing.

Although this book has a sample case study that is used to demonstrate the various aspects of software development, you will find your learning is more motivated if you pick a project of your own. Every step of the way through this book you should make sure that your project is keeping pace with the sample project. That way you will have samples to follow and a reason for working through the samples. Indeed I would recommend that you put off working through the samples until you have a project of your own that motivates your learning. Although learning for it's own sake is possible, it is much more fun when that learning results in something that you, yourself, value.

Since this is an introductory book, it includes copious references to further reading that you can do to broaden your knowledge about different aspects of software development. Some of these references will be to websites and online material, others will be to books that you might want to add to your library.

Not Just For Beginners

Since this book introduces object oriented development, experienced developers will also find this book useful as they start to use objects. Obviously, if you are an experienced developer you will be able to skip over any material that you are already familiar with. You may find however that it pays to scan through the parts that seem familiar just in case object oriented development takes a different approach than you are used to.

Why Ruby?

With Java and C# dominating the mainstream software development press it might seem strange that this book focuses on Ruby. My reasoning is simple. Ruby is an ideal first programming language -- it is simple and easy to learn yet fully embodies the power of the object oriented paradigm. Once you have taken the time to learn Ruby, you will find it easy to learn any other object oriented language.

It might seem inefficient to learn Ruby first in order to learn a different programming language, but you might find that you don't actually need to learn another language. You will discover over the course of this book that most of the things you would ever need to do as a developer can easily be accomplished using Ruby. You will probably also find that using Ruby is fun!

The other answer to the question of "Why Ruby?" is that you have to start somewhere. Over the course of my career in software development I have been paid to write code in at least 15 different programming languages and have learnt at least 10 more (either for interest, for projects that never came off or just for fun.) [An incomplete list includes Ada, APL, BASIC, C, C++, COBOL, CORAL-66, Datatrieve, Delphi, Eiffel, FORTRAN, Java, Lisp, Macro-32, Modula-2, Modula-3, Oberon, Pascal, PL/1, PL/SQL, Powerbuilder, Python, Rally, Ruby, Smalltalk, SQLWindows and Visual Basic.] If you become a serious software developer you will probably learn as many if not more, all it takes is working on a variety of projects for several different clients.

Learning to Work as Part of a Team

This book covers the basics of what you need to know before you show up on your first real software development project. The specific software development techniques covered include;

- using Use Cases to document requirements,
- using Unified Modeling Language (UML) as a design notation,
- Test Driven Development,
- acceptance testing,
- maintenance,
- incremental development,
- the effective use of version control software to support collaborative development, and
- programming using Ruby

Many other aspects of the software development will be touched on including;

- Graphical User Interface (GUI) design,
- database design,
- application architectures and design patterns,
- how software development projects are organized, and
- how teams collaborate on projects.

This is not everything that you need to know in order to be productive as part of a project team, but it is a good starting point. Once you have worked through this book I recommend that you re-read chapter 13 in detail with a view to starting your journey towards mastering the craft of software development.

Mistakes, Frustration and Learning

When learning anything as complex as software development you are going to make lots of mistakes. These mistakes are a natural part of learning – if you don't make any mistakes you are probably not learning all that much. As part of learning you will also get to experience various amounts of frustration, either with this book, Ruby or the progress you are making. Again this is normal.

For me, frustration seems to peak just before I finally get a new concept. As such I now see frustration as a sign that I'm dealing with a new idea that I don't yet fully grasp. Although this doesn't change the fact that I don't like feeling frustrated, I've noticed that avoiding the frustration doesn't work because then I don't learn. I read through the book and it all looks really easy, no hassle whatsoever. A week later when I try to use what I read the frustration finally hits as I realize I didn't learn it the first time.

So irritating as it may be, this book doesn't set out to make software development look easy. Some parts of this book are going to be deliberately vague so that you are going to have to experiment and try different things out. Yes, this will mean that you will make some mistakes along the way, but you will also learn a lot.

Whenever I'm learning a new programming language I try to see how many different mistakes I can make and how many different error messages I can generate. I started doing this by accident when I first learned COBOL, the class I was in was getting so many compilation errors that we decided to see who could generate the most errors. Deliberately making mistakes early on helps me stay relaxed later on when I get a strange error message when developing an important application.

Points To Remember

- Ruby is a human oriented language.
- All good software developers know a lot of different programming languages – even if the bulk of their work is in just one or two languages.
- Mistakes are a natural part of learning, as is frustration.
- This book is just an introduction to software development, mastering the craft takes a long time.

Chapter 1: Software Development - The Basics

Software development has a very mixed reputation. On the one hand the uninitiated often think that it "shouldn't take long" for a developer "just to add this one extra feature", while other have first hand experience of software development taking forever. The problem is that there is no real connection between how easy it is to describe what you want a computer to do and how long it will take to develop the software. It might take a few minutes of one developer's time or it might need several hundred developers working for five years or more.

The problem is that software development is bedeviled by haste and ignorance. Software can be immensely valuable, so when asked "When do you need it?" most people say "Yesterday!" At the same time, it is practically impossible to estimate with any accuracy what is involved in creating an application until you have done a good portion of the development work, mainly because we are ignorant of what is really involved.

Whenever someone gets an idea for a new application, we nearly always think that it will be easy to implement. It is only after getting a closer look at what the ideas really involves do we begin to get a clue about the true complexity of the new application. No matter how much experience a developer has, the first guess as to how long the application will take to create is going to be completely wrong.

> To give a sense of this let us consider a simple application, say a membership system for a running club. Nothing too complicated you understand, just a means of recording their names and addresses, of knowing when their membership fees are due and a way of sending out emails to notify the members about upcoming events and races. It would also be nice if there was a way of recording race results so that the club could publish lists of age ranked results in the newsletter.

How hard could it be to produce a little application like that? It sounds quite trivial does it not? Take a guess now and then periodically as you read this book check back to see if you would want to revise your original estimate.

Dealing With Haste and Ignorance

Over the years many different ways have been tried to make software development more predictable. All have involved some way of formalizing the description of what exactly is wanted, the hope being that by spending time on investigating the requirements the development team would have a better idea of what they had to create.

Detailed requirements obviously help, but they are not the complete answer. The problem remains that even the most precise statement of requirements that you can think of writing is very imprecise compared to the level of detail that developers have to use when writing the code to tell the computer what to do. The leap from the requirements view of an application to the design as implemented in the source code is massive. Yes, various approaches and notations have been tried over the years, but the fact remains that this leap requires creativity and learning.

Overcoming ignorance is the rate limiting step in most software development projects. The speed at which you can type the code is never a bottleneck, what limits a project is how fast the developers can learn about the requirements and the implications of each of their design choices. Most approaches to software development accept that many of the implications of design choices only really show up when the code is written and tested, so most projects now use incremental development.

Interestingly though it is not just the developers who suffer from ignorance. Even the people who want the application learn a lot when they start using their new application – in effect they were ignorant of the full implications of what they asked for. As soon as version 1.0 is released the users learn what they really needed. Involving your users while you are developing the application helps alleviate this to some extent, but it is only by using an application every day that users discover the true implications of their choices.

The software development community is staring to discover that haste makes for lousy software. By taking the time to think through the implications of choices and to try out different options it is possible to create better software.

Why Software Development Is So Hard

If only programming computers was as easy as speaking. After all when you ask someone to "put the cat out" he or she can understand what you mean – please find the cat (wherever it is hiding), then pick it up and physically place it out the back door, making sure that it does not sneak back inside before the door is closed. Humans can understand instructions because they are intelligent and have learned to deal with the ambiguities of spoken language.

Computers are different, they are just fast, non-intelligent machines (despite all of the claims over the years for "machine intelligence"). You don't want a machine to be choosing between the different possible interpretations of the instructions you gave it. It is bad enough when people choose an interpretation different from the one you intended -- "You were supposed to put out it out the back door not the front door" – given the speed at which computers operate it would be a impossible to correct any misinterpretations, only to clean up the mess afterwards.

Software development is hard because we have to be extremely precise when we write software. The software has to tell the computer exactly what to do and how to do it in excruciating detail. There is absolutely no room for error, be it a five line program of a 50,000 line long application, even a single typo or spelling mistake will cause the computer to do something other than what you wanted it to do.

Today we are lucky that we have high level object oriented scripting languages that are very powerful, but in the old days the programming languages were very close to the machine. At the basic level all computers do is move around chunks of data and do some basic arithmetic. Every other capability that computers have has been programmed in by software developers. So in Ruby we have libraries that make it easy to interact with web servers using http.

```
require 'net/http'
http_connection = Net::HTTP.new('www.mcbreen.ab.ca')
resp, data = http_connection.get('/index.html')
puts resp.content_length  # print the size of page in bytes
puts data[0..100] # print first 100 bytes of the web page
```

But even with Ruby you can see hints of how it used to be in the last line of the above example where we have to tell the computer exactly how much of the web page it should print.

Overall these powerful libraries make programming easier, but the task of software developers is still hard because in order to be able to use these higher level facilities, you have to learn how to use these libraries. This is no small task because each of these libraries is quite large and feature rich.

How to Craft Great Software

Small teams of developers can create great software. Basically what the team has to do is fairly simple;

- Maintain a clear vision of what they intend to create,
- Involve the intended users in the creation of the application,
- Understand how the users want to achieve by using the application,

- Use classes to represent the design concepts in the application,
- Keep the code in every class coherent and understandable,
- Release a new, tested version of the application to the users every few weeks, and
- Respond immediately to the feedback the users give about every release.

Unfortunately, although this is fairly simple advice, it is not necessarily easy to follow. For many applications just getting everyone to agree on the important features that the application has to have is not easy. Keeping the code coherent and understandable is always a challenge, and it is very easy for developers to not listen to their users.

The best way to learn to do all of this is as a junior member of a team that is successfully maintaining some applications. Unfortunately, there is a slight Catch-22, most teams don't want a complete beginner. That is where this book comes in. It walks you through a small project from the initial idea "a membership system for a running club" through to enhancing the application when change requests come in. Along the way it will help you learn how to document requirements and implement your designs in Ruby. Yes, you will still be a beginner, but you will at least know what questions to ask in order to become a productive member of the team.

Scripting vs. Programming

Historically, developers have preferred compiled programming languages rather than interpreted scripting languages because the compiled versions ran much faster. This was very important in the old days when machine time was very expensive, but with modern machines the speed difference is not really noticeable. Yes, for some compute intensive tasks (say image processing) a compiled programming language might still be preferable, but for many purposes what matters is ease of development.

Scripting languages like Ruby are becoming popular because they make it possible to quickly develop an application, often by gluing together existing components. Yes, they might run a bit slower than a compiled application, but the money saved in the development activities could be used to buy faster hardware if absolute speed is important. If all else fails the Ruby version can be used to demonstrate the overall feasibility of the application, then the slow parts can get rewritten in a compiled language.

The big advantage that object oriented scripting languages have is that they are what Matz (Yukihiro Matsumoto) calls "human oriented." Rather than being designed to make it easy for a computer to execute the code quickly, they are designed to be easy for a person to use and understand. By making the computer do more of the work, humans win by having a more productive, fun environment to develop applications in.

Questions To Ponder

1. Why do you want to become a software developer?
2. How long will it take to create the running club membership system?
3. What application do you want to build to showcase your talents as a software developer?

Points To Remember

- Haste and ignorance are the main problems that any software development project faces.
- Learning is the rate limiting step in any software project.

Chapter 2: Ruby For The Nuby – WORK IN PROGRESS

The really nice thing about Ruby is that you can run it interactively without having to worry about compiling the code or ever saving the code in a file. All you have to do is start up Ruby specifying that you want to use the **irb** tool, at which point you can treat your computer just like a calculator. All of the usual arithmetic operators are available, plus a few you might not expect like `*`, `**` and `==`.

```
D:\Ruby\bin>ruby irb
irb(main):001:0> 2**3
8
irb(main):002:0> 2 == 3
false
```

Take a few minutes now to experiment with the arithmetic capabilities of Ruby, see if you can remember the difference between integers and real numbers when the division operator `/` surprises you.

The Vocabulary of Programming

As with any field of endeavor, programming has its own jargon. Once you have learned this vocabulary, it is very easy to talk about and critique the source code for any application. Learning the vocabulary allows you to make distinctions between different parts of the code. Learning these distinctions is crucial, so make sure you spend a lot of time playing with Ruby as you read this chapter.

Variables and Constants

Ruby is object oriented scripting language that garbage collects all unreferenced objects. What this means is that effectively all objects anonymous and easily forgotten unless we name them. Ruby supports two different types of names, variables and constants.

Variables are things like **http_connection** that was used in the last chapter. Whenever you need a variable make sure you give it a *Role Suggesting Name* – something that helps you recognize the purpose of the variable and how it is used.

Constants hold references to objects just the same as variables do, but constants are special in that they are not supposed to be changed. Ruby recognizes constants by the fact that they start with a capital letter, **Standard_http_port**. Although some languages stop you from ever changing a constant, Ruby is more forgiving, it just gives you a warning and does the assignment.

```
irb(main):001:0> Standard_http_port = 80
80
irb(main):002:0> Standard_http_port = 81
(irb):2: warning: already initialized constant Standard_http_port
81
```

Objects, Classes and Methods

In Ruby you do not have to worry about specifying the datatype for any constant or variable because every object remembers its own type. This makes programming much easier since you have less bookkeeping to do.

The other really nice thing is that beginners do not have to worry about declaring the types of variables, which drastically reduces the potential for making mistakes. The following example creates an array of four numbers and then invokes a few methods to check that it is indeed an Array that has been created.

Line five then prints out the first element of the array, the 'nil' output is the report from **irb** if the value returned from the print method. Comments start with a **#** and run to the end of the line.

```
irb(main):002:0> a = [1, 2, 3, 4]
[1, 2, 3, 4]
irb(main):003:0> a.class
Array
irb(main):004:0>
```

A really nice part about using Ruby in an interactive way is that as you type in each line, it checks the syntax and reports any errors as shown in this example of attempting to use an iterator to print each element of the array.

```
irb(main):006:0> a.each { |i| print_line i }
NameError: undefined method `print_line' for #<Object:0xa04a818>
(irb):6:in `irb_binding'
(irb):6:in `each'
(irb):6:in `irb_binding'
irb(main):007:0>
```

The iterators in Ruby are really powerful because they remove the potential for whole set of the typical "off by one" errors that are easy to make when explicitly looping over a collection. The block of code is executed once for each element of the array, with **elem** being set to the value of the element being processed.

```
irb(main):007:0> sum = 0
0
irb(main):008:0> a.each { |elem| sum = sum + elem }
[1, 2, 3, 4]
irb(main):009:0> sum
10
irb(main):010:0>
```

So Where Are The Objects?

The simple answer is everywhere. Even simple numbers are objects, and it is easy to ask anything what kind of object it is, what it is derived from and the messages it can respond to. (The list of methods has been truncated here for brevity.)

```
irb(main):010:0> a[1].class
Fixnum
irb(main):011:0> a[1].class.ancestors
[Fixnum, Integer, Precision, Numeric, Comparable, Object, Kernel]
irb(main):012:0> a[1].methods
["<=", "zero?", "times", "step", ......]
irb(main):013:0>
```

And just to prove that the numbers really are objects, here is a variation on the familiar "Hello World!" example of using the **times** method which executes the passed block of code the appropriate number of times.

```
irb(main):013:0> a[1].times { |n| p "Hello World! " + n.to_s }
"Hello World! 0"
"Hello World! 1"
2
irb(main):014:0>
```

Covers the real basics of writing programs through lots of examples and exercises. Will use diagrams to show the basic structures of sequence, iteration and conditionals.

The basic data types will also be introduced, as will methods and code blocks. Simple scripting tasks will be used as exercises in this section.

This chapter will close with a short section looking at code blocks in Ruby and the way that they are used in iterators.

Questions To Ponder

1.

Points To Remember

•

Chapter 3: Object Oriented Development

Covers the basic object concepts and illustrates them using UML diagrams. The intention is to provide a gentle introduction to how these concepts work in Ruby.

Focus is on how objects help manage complexity and improve maintainability of code. Will look at inheritance as the sharing of implementation and talk about polymorphism as conforming to an interface. Exercises will be used to emphasize the idea that polymorphism does not have to imply an inheritance relationship.

This chapter will close with a short section looking at static and dynamic typing.

Points To Remember

-

Chapter 4: Ruby for the OO Nuby

Shows how Ruby implements the basic OO concepts, with small code samples and the associated simple UML diagrams.

Inheritance will be covered in detail, with short exercises that illustrate how a group of objects can cooperate to deliver useful behavior. Exercises will also point the reader towards experimenting with some of the existing Ruby classes.

This chapter will close with a short section looking at how in Ruby even single object can be specialized by adding new methods to them.

Points To Remember

-

Chapter 5: Objects and the Software Development Process

This points out the fact that objects change software development. We are no longer creating standalone programs, but a set of interacting classes and objects. The impact this has on the overall development process is discussed.

Agile and other methods are mentioned in passing, but the book will use some of the terminology from the Unified Process. Will discuss the UP phases of Inception and Elaboration, then talk about Test Driven Development.

This chapter will close with a short section looking at software development for one - how the larger team processes change when there is only one developer.

All too often I hear people dismiss the complexity of software development by claiming that something is just a *simple matter of programming*. My normal response is that if the speaker thinks it is going to be simple then he can write it himself.

Even a seemingly trivial application can be amazingly complex when you have to address all the weird and wonderful things that can go wrong. Even a simple thing like sending out email event notifications can have hidden complexities as we saw in the last article, Part 5: Creating Acceptance Tests from Use Cases [please ref.]. If the requirements for even a simple running club membership system can be quite complex, imagine how much complexity is hidden in larger commercial applications.

All Software Is Complex

Once we get beyond the typical toy, teaching example that is less than 100 lines long, understanding the entire application takes time and effort. Individually, each line of code is nearly always simple and easy to understand. Yes, sometimes you have to *Read That Fine Manual* (RTFM), as was the case with the **`collect!`** [http://www.rubycentral.com/ref/ref_c_array.html#collect] idiom in the Ruby for the Nuby article [Please ref. article].

The complexity in an application lies more in the interactions between the effects of each line of code rather than in the individual lines themselves which are more or less easy to comprehend. The complexity arises because the data that one line of code operates on may have been changed by any number of lines of code that preceded the one you are looking at.

Object oriented programming languages such as Ruby [http://www.ruby-lang.org/en/index.html] reduce this complexity somewhat by encapsulating all data inside objects, so the volume of code that can directly affect any data is reduced. Whenever the data within an object is changed, you know that it had to have been changed by one of the object's methods. This drastically decreases the complexity of the task facing us as developers when we want to under what some code is doing.

The complexity is further reduced by the fact that with objects, most programmers try to give the methods intention revealing names while keeping the methods themselves relatively short. This means that it is relatively safe to infer from the name of a method what is happening. Ruby takes this even further by allowing trailing **`?`** and **`!`** in method names, query method names conventionally end in **`?`** (**`Array.empty?`**) and methods that can drastically affect the object conventionally end in a **`!`** (**`Array.sort!`**).

One hidden source of complexity in understanding software is that even with encapsulated data, objects can still be modified by simply invoking one of their methods. To get around this type of complexity, most developers adopt the convention that an object should only send messages to closely related objects [http://www.ccs.neu.edu/home/lieber/LoD.html] and hence frown on any object that is globally accessible (and hence could be accessed from anywhere). Ruby supports this convention by explicitly prefixing all globally accessible variables with a **$**, making it immediately obvious when this style rule is being broken.

Controlling Complexity Though Design

All of the things that help make existing software more understandable need to be considered when you have the task of designing new application. After all, once you have created it, sooner or later you are going to have to revisit the code to add in new features of change existing ones. When you have to do that you will find out whether you have been successful in creating understandable, maintainable code.

To create maintainable software you really need the help that objects provide. You need to make sure that you design the interactions between the objects in a way that makes the resulting code easy to read and understand. After all you never want your teammates to hear you cry "*Who wrote this piece of garbage, I cannot make sense of it at all ... oh! it was me ...* "

Classes Represent Concepts

The first thing to remember when designing your application is to use classes to represent important concepts. How do you know what the important concepts are for your application? Simple. Look at your requirements, use cases and test cases, the concepts that show up in those are in all probability going to become classes in your application.

For the Running Club Membership system discussed in the last article, some candidate classes are

- Member - represents a club member
- Club - represents the club the members belong to (this is a very easy concept to miss believe it or not)
- ClubEvent - The special events that the club secretary wants to tell the members about
- Race - a special kind of ClubEvent, one that members report RaceResults against
- RaceResult - represents a members time and placement in a particular Race

Assigning Responsibilities to Classes

Once you have some ideas for the candidate classes, your next design task is to work through one use case and associated acceptance test cases [please ref. Part 5 article] to assign responsibilities for delivering the various subfunction goals to classes within the application. When doing the initial assignment of responsibilities, try to group related responsibilities together on the same class. Don't worry if you identify a responsibility that doesn't fit any of your candidate classes, it just means that you have to invent a new class for that particular responsibility.

As you work through your design ideas you will inevitably learn about how your design works and in the process realign the responsibilities. Indeed you might find it beneficial to come up with three different initial designs to explore different options. If you do this you might even discover that the first idea you came up with is not actually the best option.

When looking through the use cases to identify the responsibilities it is easiest to just read through the main success scenario first. Defer looking through the extensions for responsibilities until after you feel you have correctly assigned the main responsibilities.

For the Running Club Membership system Club Secretary: Notify members about special events use case here is one possible assignment of responsibilities

Member - represents a club member

- Knows name, email address and phone number
- Matches what it knows against specified selection criteria
- Records delivery of event notification

Club - represents the club the members belong to

- Knows members and special events
- Sends event notification emails
- Prints event notifications

ClubEvent - The special events that the club secretary wants to tell the members about

- Knows name, date and description of special event

In this design option the ClubEvent is a data holder object without any really interesting behavioral responsibilities. Another design option would be to have the ClubEvent responsible for emailing itself.

The Craft of Programming

Becoming a good programmer requires that you develop a feel for the aesthetics of code. It is not enough that the code *sort of works*, we need code that is beautiful as well as functional. After all, as a developer you are going to spend a lot of every working day working with code in one way or another, and it is a drag having to work with ugly things all of the time. Much better to make the code pleasing to the eye, because it is amazing how long software lasts. Ten years from now you don't want to be looking at some of your code and cringing because of how ugly it looks.

Converting from design ideas to source code is not a mechanical process. Yes, initially you can just mechanically translate the design ideas, but one you have done that you have to read the code and adjust it until it feels right. This process of making it feel right is called Refactoring [http://www.informit.com/content/index.asp?product_id={B372FE30-1699-4C71-8C46-36DF503ED0A1}] and is a process of identifying and removing code smells (as in "that code stinks, we need to clean it up").

Responsibilities Are Implemented As Methods

Whatever an object is responsible for knowing is likely to be an attribute of the class, and as such will need a set of accessor methods. These are represented in Ruby as follows:

```
class Member
  attr_accessor :name, :phone_number, :email_address
end
```

Responsibilities that involve the object answering a question are usually implemented as query methods, and the method name will have a trailing **?** if the method returns a boolean true/false answer.

```
class Member
  def matches? ( search_criteria )
    # to be implemented using test driven development
    return false
  end
end
```

Normal behavioral responsibilities are usually implemented as methods that return whatever is appropriate. Only give a method name a trailing **!** if the method can have drastic consequences.

```
class Member
  def notified ( club_event )
    # to be implemented using test driven development
  end
  def already_notified? ( club_event )
    # to be implemented using test driven development
    return false
  end
end
```

For symmetry a query method – **`already_notified?`** -- has been added to make it easier to ensure that duplicates are never sent.

Allow Test Driven Development To Improve Your Design

While it is extremely tempting to jump from the design ideas into writing the methods, that is normally a mistake. It is much better to pause, think of how you are going to unit test the methods and then write the test first [please ref. Test Driven Development article]. In all probability when you write the test you will realize that the original name you were going to give the method just doesn't really look right when written down. Whenever that happens, pick a better name and use that one instead of your original design.

The next article in this series will show how the email notifications part of the Club Membership system can be implemented in Ruby using Test Driven Development. In doing so it will uncover yet more complexities in what initially looked like a really simple application.

Points To Remember

-

Chapter 6: Analysis, Design and Other Myths

Looks at the difference between phases and activities and the intertwined nature of all of the activities.

Will take a very close look at Dijkstra's take on testing not proving the absence of bugs and position this in relation to what is practically feasible. It will also look at incremental development and software architecture, and touch on reuse, rework and refactoring.

This chapter will close with a short section suggestions that all developers need to be skeptical about most claims and pronouncements made about software development.

Points To Remember

-

Chapter 7: Before You Start The Project

Looks at what is important when thinking of starting a new project, especially envisioning and scoping.

Will also look at how projects are planned and the way that effort is phased on projects. Will revisit the haste and ignorance discussion again to point out that in the early stages of a project it is impossible to predict and/or plan all aspects of a project.

This chapter will also introduce a club membership system case study that will run through the next 4 chapters. The reader is invited to invent their own project that they can run in parallel to reading about this example case study. The toy example in the book is merely an illustration, the motivation for learning will come from working on their own project.

This chapter closes with a short section looking at the ideas of diversity, fast failure and prototyping.

Points To Remember

-

Chapter 8: Starting Your Project

Looks at refining the vision for the project, using Use Cases and UML diagrams to record the requirements and the scope of the project.

Looks at haste and ignorance again by talking about estimating, prioritization and project planning. When looking at the case study project it will emphasize that it is not possible to have it all at once, that developers need to stay focused and deliver what is important.

Case study will include low precision use cases, some diagrams showing the scope of the project and some candidate design ideas. It will also show a plan for the elaboration phase. The reader is encouraged to develop the same artifacts for their own project.

This chapter closes with a short section looking at learning from surprises.

One very intriguing aspect of software development is that whenever someone gets an idea for a new application, we nearly always think that it will be easy to implement. It is only after getting a closer look at what the ideas really involves do we begin to get a clue about the true complexity of the new application. No matter how much experience a developer has, the first guess as to how long the application will take to create is going to be completely wrong.

To give a sense of this let us consider a simple application, say a membership system for a running club. Nothing too complicated you understand, just a means of recording their names and addresses, of knowing when their membership fees are due and a way of sending out emails to notify the members about upcoming events and races. It would also be nice if there was a way of recording race results so that the club could publish lists of age ranked results in the newsletter.

How hard could it be to produce a little application like that? It sounds quite trivial does it not? Take a guess now and then after you have finished reading this article see if you would want to revise your original estimate.

Understanding The Requirements

Requirements are very slippery things. Software developers often speak about "The Requirements," as if they are cast in concrete never to be changed or questioned in any way. The reality however is that requirements come from people, and you can bet that whoever dreamt them up in the first place is just as capable of making a mistake or forgetting things as you are.

So the first thing you want to do whenever you hear about an idea for a new application is explore the requirements space and attempt to understand the vision for the application.

Exploring The Requirements Space

What this fancy sounding phrase really means is that you need to meet with the person with the ideas and have a conversation about what they are really trying to achieve. You want to get an idea of the motivations underlying the request for the application. You also want to get a sense of the kinds of ways in what the application is intended to be used and any ideas that people have for ways in which the application could be used.

Understanding The Vision

In order to do a really good job in developing the application you have to understand the vision for the application. This means you have to work with the people who want the application to jointly envision how the application will be and could be used.

Defining The Project Scope

Developers have always used typical scenarios to try to understand what the requirements of a system are and how a system works. Unfortunately although developers have done this, it has rarely been documented in an effective manner. Use Cases are a technique for formalizing the capture of these scenarios.

Although Use Cases, as defined in the Objectory book (Object Oriented Software Engineering, Jacobson, Christerson, Jonsson & Overgaard. Addison Wesley 1992), are associated with Objects, the technique is actually independent of Object Orientation. Use Cases are an effective way of capturing both Business Processes and System Requirements, and the technique itself is very simple and easy to learn.

Making Requirements available for Review

The reason for formally capturing the scenarios is to make them available for review by both the users and the developers. There are 2 specific criteria that any functional requirements notations needs to meet:-

1) it should be easy for the sources and reviewers of the requirements to understand and

2) it should not involve any decisions about the form and content of the system.

Functional Requirements are external requirements that are to be used to evaluate the design and final implementation of the system. What these requirements have to do is capture in an implementation-independent manner what the needs and expectations of the stakeholders are.

Use Cases make requirements available for review

Use Cases are starting to become widely used. Compared to other requirements capture techniques, Use Cases have been successful because:-

- Use Cases treat the system as a black box and
- Use Cases make it easy to see implementation decisions in requirements

This last point arises out of the first. A Use Case should not nominate any internal structure to the system that the requirements relate to. So if the Use Case states “Commit changes to Order Database” or “Display results on Web Page”, the internal structure is easily seen and can be flagged as a potential design constraint.

The reason why the requirements should not nominate any internal structure, is that specifying the internals puts extra constraints on the designers. Without these constraints designers have more freedom to create a system that correctly implements the externally observable behavior, and the possibility exists that they may come up with a breakthrough solution.

What are Use Cases?

A Use Case itself is an interaction that a User or other System has with the system that is being designed, in order to achieve a goal. The term Actor is used to describe the person or system that has the goal, this term is used to emphasize the fact that any person or system could have the goal.

The goal itself is phrased with an active verb first; examples being "Customer: place order", "Clerk: reorder stock".

As part of the Use Case it is necessary to document what goal success and goal failure means to the Actor and the System. Within the context of placing an order, goal success probably includes the goods being delivered to the Actor and the Company receiving the appropriate payment for said goods. Careful definition of Goal Success and Failure are essential for defining the scope of the system, since for a limited Order Entry system, Goal Success would merely mean that the order has been validated and delivery scheduled.

The Role of Scenarios

Different scenarios within a Use Case show how the goal succeeds or fails; a success scenario is one in which the goal is achieved, a failure scenario is one where the goal is not achieved.

The nice part about this is that because the goals summarize the intention of the various uses of the system, the users can see how they are supposed to use the system. Users can also spot when the system does not support all of their goals, without having to wait for the first prototype, or even worse having to wait for the system to be developed.

How "Big" should a Use Case be?

An interesting topic is that of trying to determine how to "size" a Use Case. One way to address this is to relate the size to the purpose and scope of the Use Case. With a really large scope, a Use Case is not so much addressing a single system but all of the systems used by a business. Such a Business Use Case, would treat the entire company as a black box, and speak of the Actor's Goals with respect to the Company. The scenarios for these Business Use Cases would not be allowed to assume any internal structure to the company. A customer would place an order with the Company, not the Customer Service department.

For System Development however, the scope of the Use Case is restricted to a single system. These are the most common form of Use Cases, which we could call System Use Cases, and they treat the system as a black box. These Use Cases cannot specify any internal structure and should be restricted to using only words from the problem domain.

Another scope for Use Cases is the design of sub-systems or components within a system. These Implementation Use Cases treat a component as a black box and the actors are the components that interface to it. For example it would be possible to use Implementation Use Cases to specify the requirements for an email component to be used by an application.

Given this classification, the conversation about the size of a Use Case is easier. The scope of the item to be designed sets the overall size. To assist the system designers, each Use Case should describe only a single thread of activities without major branches. Violation of this restriction can usually be seen by imprecise or vague Goal Success criteria, which makes it hard to use the Use Case as a source for test specifications.

As an example of a System Use Case, "Query database for low stock" is too small as can be seen by the mixing of the implementation detail with the requirements. By contrast however, as a System Use Case, "Manage Warehouse" is too large, since it cannot be accomplished as a single thread of activities without major branches, and from a system viewpoint it would be hard to specify Goal Success. But it would make a good Business Use Case for a Parts Department. For a

Parts Department, it would be possible to define Goal Success for "Manage Warehouse",

(probably in terms of Inventory turns, parts availability, operating costs etc.)

The nice thing about these Business Use Cases is that they can be used to categorize the other Use

Cases. Thus "Manage Warehouse" could be used to group all of the Use Cases involved with the

actual management of the warehouse.

A Formal Definition of Use Cases

A Use Case a delivers a measurable result of value to a particular Actor.

As noted previously, Actors can be people or external systems that the system being designed has to interact with. The requirement for a Use Case to have a measurable result, is derived from the need for a single thread. As part of having a measurable result, the goal either succeeds or the goal fails. There is no room for middle ground.

Achieving the goal of the primary actor is defined as a success, all results that do not meet the goal of the primary actor are defined as goal failure. The different scenarios show all of the paths to success or failure.

Documenting Use Cases

The nice thing about Use Cases is that the scenarios can be documented with varying degrees of formality. Each scenario refers to a single path through the Use Case, for a particular are set of conditions.

Informal text descriptions can be used, but they are sometimes difficult to follow when there are multiple conditions and possible failures. The informal narrative style, however is very useful initially while trying to understand the requirements. Later on in the development of the Use Cases however, it is useful to use a more formal mechanism for documenting the Use Cases.

A **rough sketch** of the Customer: Place Order Use Case could be as follows:-"Identify the customer, check that requested goods are in stock and that their credit limit is not exceeded"

A structured narrative format has proved to be very effective. What this format does is specify a sequence of Actor:Goal pairs for each of the scenarios. The way these are written down is to specify the simple success scenario first, as a sequence of Actor:Goal statements, all of which assume success of all previous goals. This sequence, as shown in the example, is the simplest scenario that leads to success. The Use Cases consider the system that we are designing to be a single black box. No internal structure is recorded at all, and it can be considered to be a single Actor for the purposes of writing out the scenario. The Use Cases do not say anything about the internals of the system, only what goals the system will have and what goals it is responsible for handling.

1. Clerk: Identify Customer

2. System: Identify Item

3. System: Confirm Ship Quantity

4. System: Confirm Credit

5: Customer: Authorize Payment

6. System: Dispatch Order

Extensions

1a. Customer not found

1a1. Clerk: Add new Customer

3a. Partial Stock

3a1. Clerk: Negotiate quantity

Handling Goal Failure - Extensions

What needs to be identified next is that each of the steps above may fail. The conditions that may lead to failure need to be captured as extensions to the scenario. These extensions are dealt with by writing a partial scenario below the failure condition and following that partial scenario until it either rejoins the main track or fails.

This separation of the failure conditions makes the scenarios more readable. The primary success scenario is the simplest path through the Use Case, each step of the way, the actor's goals were successful. A separate listing of all of the failure conditions allows for better quality assurance.

Reviewers can easily check whether all conditions have been specified, or whether some potential conditions have been omitted. Failure scenarios can either be recoverable or non-recoverable. Recoverable failure scenarios eventually succeed, non-recoverable failure scenarios fail directly.

Failures within Failures

One extra complexity that needs to be highlighted at this point is that within a failure scenario, other failures can occur. This means that the extensions section can have further Failures identified with a slightly longer prefix number: 1a1b. Customer is a bad credit risk. This would be recovered through 1a1b1 .

Why use a structured narrative format?

The value of the structured narrative format is that it is a refutable description. The value of having a refutable description is that it is precise enough to be argued over and disagreed with.

"That is not the way it works"

"We do not check availability when taking orders"

"You missed a few steps"

In contrast, the rough sketch provided by informal narrative text descriptions is hard to refute, but it is useful for early understanding of a problem domain.

Another way of describing the value of a refutable description is that when you document Use Cases, expect to get feedback from users and developers about the quality of the Use Cases. This is very valuable since it means that corrections can be made very early in the development process.

The typical feedback from users highlights different sequencing, possible parallelism or missing steps. Typical feedback from developers is related to requests for clarification about what a particular failure condition means and how to detect it.

Requirements Capture and System Complexity

In review, the scenarios capture system complexity while the Actor:Goal pairs enable even large systems to be documented in a relatively condensed format. The value of this Use Case format is that instead of having a large functional specification that is rarely read, with Use Cases, users and developers can identify the actors and then confirm that the listed goals match (or do not match) the job responsibilities of that actor.

Only then, for the Use Cases that the user or developer is interested in, is it necessary to dive into the details of the scenarios.

But Systems have more than just Functional requirements

Use Cases however, do not capture all of the external requirements of a system. Use Cases merely capture the functional requirements of how the system will be used. There are many other aspects to the requirements that must be captured and addressed. Some of these non-functional requirements are however, use related, so they can be attached to an individual Use Case.

Examples of this include requirements such as throughput and performance. Other requirements however are not use related and need to be captured separately. Examples of these would be

- System scope
- the Goals that the Users have for the system
- user interface prototypes
- general rules
- constraints
- algorithms

Run Time vs. Build Time Requirements

An important factor to remember when capturing requirements is that the constituency for the system is much larger than just the User community. There are many different Stakeholders in the system, and Use Cases only capture the needs of some of these Stakeholders. In essence, the Use Cases only capture the run time requirements for the system and ignore a major Stakeholder, the system development organization. The Development Organization is has a strong interest in specifying the build time requirements for the system.

Run time requirements include: System scope, Goal and expectations of the product in the user organization, Use Cases, Other non-functional requirements.

The build time requirements include: Ease of development, Robustness in the face of change, Reuse of existing components.

The build time requirements can partially be handled by Use Cases. But the are many other aspects that the development organization needs to address;

- **Project scope and goals**: what this project must deliver (as distinct from the system scope which typically will be delivered over several projects)
- **Instances of growth and change**: these can be captured as "Change cases" in the normal Use Case format
- **Development sponsor constraints**: these include standards, practices, tools, quality measures and quality assurance principles and practices.

Applicability of Use Cases

Use cases are primarily for systems that need to respond to external events. They can be used in other environments provided that there is a clear actor who has a clear understandable goal.

Use Cases cannot be used when the outcome is undefined or unclear.

This means, that if goal success and goal failure cannot be defined, then Use Cases should not be used to capture the requirements.

Having said that however, most modern object methodologies are now using Use Cases. This is because Use Cases have proved to be a very effective mechanism for capturing requirements.

Summary

Use Cases capture requirements in a readable, refutable format. The Use Cases are a refutable description of the externally required functionally of the system. Refutable means that when you document Use Cases, expect to get feedback from users and developers about the quality of the Use Cases. Use Cases do not need to be precisely defined right from the start. The typical development sequence is as follows

1. Identify actors.

2. Identify the goals of the actors.

3. Identify what success and failure means for each of the Actor:Goal pairs.

4. Identify the primary success scenario for each of these Use Cases.

5. During elaboration, identify failure conditions and the Recoverable/non-Recoverable scenarios.

It is only necessary to get to step 4 before deciding which releases of a product a particular Use Case will be developed in.

In summary, Use Cases are an effective requirements capture technique that makes requirements available for review, avoiding any implementation bias in the requirements.

Points To Remember

•

Chapter 9: Elaborating Your Project Ideas

Looks at the elaboration phase in detail, formalizing the design sketches using UML diagrams (Class and Interaction, ?maybe a state chart?) and building a small executable architecture.

The case study will create the executable architecture in Ruby. Minimal error handling will be included at this stage, the focus is on demonstrating the design concepts. Rather than complicate this book with database and SQL concepts, the data for the application will be stored in simple files.

The reader is encouraged to bring their own project along to the same stage by posing a series of tasks and questions to ponder about what they are building.

This chapter closes with a short section looking at learning from technical experiments.

Points To Remember

-

Chapter 10: Iterative Development

Looks at Test First Development by introducing Test::Unit for unit testing in Ruby.

The case study will be extended over 3 iterations, each one implementing an extra use case. This chapter will also provide a very simple introduction to a GUI toolkit for Ruby. [Two sections may have to be written here, depending on whether there is a reasonable cross platform library for Ruby - Fox/FXRuby covers Windows and Linux, but does not support the Mac, so we might need a RubyCocoa section here].

The reader is encouraged to bring their own project along to the same stage by posing a series of tasks and questions to ponder about what they are building.

This chapter closes with a short section looking at the controversy surrounding design documentation.

Points To Remember

-

Chapter 11: Acceptance Testing and Delivery

Looks at acceptance testing and the roles that testers play in projects.

The case study will be extended by looking at the usability of the application and how developers interact with testers and users as a project nears completion.

This chapter closes with a short section looking at what it means to release software and what happens when people start to rely on your software.

Points To Remember

-

Chapter 12: Maintenance, Documentation and Evolutionary Development

Looks at how software is never finished and the need to support the applications that you write.

This chapter also looks at the nature of documentation and how it assists when maintaining and evolving applications over long periods of time.

This chapter closes with a short section looking at supporting multiple version of the same application.

Many organizations complain that their legacy systems are unmaintainable, but guess what? The organization let the code get that way by not paying attention to the design of the ***teams*** *that created and maintained the code. Refactoring the code may be a stopgap solution, but refactoring the organization is a better solution.*

The sorry state of software in many organizations is attested to by the way that people talk about "legacy systems." Nobody seems to be excited about working on a "legacy system," even those that are mission critical and are handling the bulk of an organizations revenue stream. Sometimes it seems as if nobody wants to work on "legacy systems," except maybe as a precursor to replacing those legacy systems.

The problem is that many organizations have let their mission critical systems fall into an abysmal state where nobody in the organization really understands these "legacy systems" any more. Even worse, the organizations have failed to train their developers in the technologies they need to know to look after these mission critical applications. No wonder it takes forever to get a simple change made on these mission critical systems – nobody in the organization knows how to write COBOL. It would be laughable if it wasn't so sad.

As Trygve Reenskaug once said "As time passes, only COBOL lasts." The reality is that in the 1970's and 1980's lots of mission critical applications were written in COBOL or similar vintage languages like Assembler, FORTRAN, PL/1 and RPG. Even now, in the age of the Internet, Java and Web Services, most companies are still dependent on applications written in these "legacy" languages. The Y2K fiasco did not kill all these mission critical applications off, they are just as important as they ever were.

Maintaining "Legacy Systems"

Although it is possible to "refactor" (rewrite) all of these "legacy systems" into modern OO, Internet aware applications, until organizations have learned how to maintain mission critical applications, this would be a waste of time and effort. The problem is that no matter how clean, tidy and up-to-date the rewritten application is made, it would soon fall into disrepair just as the original "legacy system" did.

One organization that was having trouble hiring COBOL developers decided to use Java instead. The crazy thing they found though was that they couldn't hire enough Java programmers for their needs either. So guess what happened? They ended up paying to retrain all their COBOL developers in Java rather than train a few new hires in COBOL.

All of this activity is a direct consequence of the way that organizations reward developers. All of the high status and pay goes to developers who work on new applications. The maintenance of the mission critical applications is a *low status position*! Absolutely crazy. No wonder these mission critical systems become "legacy systems" – they are being "looked after" by people who cannot wait to stop working in maintenance and start working on "real projects".

Maintenance of Software is a High Status Position

The way out of this problem is to take a leaf out of the Free Software/Open Source world. Make being the maintainer of an application a *high status position* and then reward longevity of service.

Think for a minute about all of the money that has been wasted over the years on rewriting "legacy applications". Surely a much better solution would have been to attract experienced, talented developers into maintaining these mission critical applications so that they didn't need to be rewritten in the first place?

Yes, that would mean that we would have to stop making all of the jokes about maintenance programmers, and we would still have to teach COBOL, but at least it would prevent us from getting sucked in by the hype surrounding new tools and technologies.

I've lost count of the number of organizations that have nearly crashed and burned on a Java project [http://www.informit.com/content/index.asp?product_id={C6771D0D-4325-4DD5-A787-A34CB30DB24D] in a failing attempt to replace a legacy system. The reality is that it is only recently that the Java tools have attained the stability and robustness that "legacy" mainframe developers used to be able to take for granted.

Length of service is a very important factor because the reality is that software does not wear out. Many urgent "legacy" replacements have only become necessary because of how badly the application has been maintained or because a vendor decided that a platform is no longer "strategic."

In recent years many of the old "mini-computer" vendors have decided to stop supporting their older machines and operating systems leaving many organizations with no option but to rewrite applications. No wonder the Free Software/Open Source model is becoming attractive – there is a lot less chance that a platform will be declared no longer "strategic."

Design For Maintenance

1. **Assess the planned economic life of your applications**. Yes, some applications might only be needed for a year or two, but most mission critical applications are going to last for a long time. After all an application that takes 18 calendar months and 40+ developer years to create had better last for a long time. Planned obsolescence might be a good idea for a manufacturer of consumer goods like cars, but for everyone else it is just a waste of time and effort to continually have to invest in new versions.
2. **Make your organization a great place for developers to work**. Losing an experienced developer from a team is very expensive. Large mission critical applications take years to learn so it pays to retain the people who know the applications. The side benefit for the rest of the organization is that great organizations to work for are a lot less stressful and a lot nicer place to be.
3. **Forget about the documentation myth – insist that the team that develops an application maintains it**. Despite what the software engineering world has been trying to claim, software development is an intellectual, learning process. The best people to maintain and enhance an

application are the people who developed it in the first place. Whenever you are thinking about starting a new project, make sure that the bulk of the team is willing to commit for the planned economic life of the application. After all it makes no sense at all to plan to pay for two teams to learn the application.

4. **Start paying maintenance programmers more than the people creating new applications**. The way to attract good people into maintenance is to pay them. In the same way that lots of people jumped on the Java bandwagon for the rewards, once maintenance is hot, good developers will be attracted to it. Then the bragging rights will go to the maintenance teams with the best track record rather than the DotComs that are burning money playing with bleeding edge technology.

5. **Hire developers based on their experience with maintaining systems**. A key hiring criteria that organizations have to get serious about is experience with maintenance. Rather than looking at how many applications a person has developed, look instead at how many applications a person has supported and for how long. Shipping the first version of an application is useful, but a developers learns many useful lessons by sticking around to ship the fourth and fifth versions of the same application.

6. **Train your developers in all of your technologies**. Developers who only know one technology are hazardous to the health of your applications. Your developers need to be familiar with all of the operating systems you use, mainframe, mini-computer and PC. Cross-training is essential so that the developers understand the real issues that face your projects. Similarly your developers need to be able to at least read and understand every programming language in use in your organization. After all, if you do ever have to replace an application the developers are going to have to read the source code for the old application to figure out what exactly it was doing.

7. **Ask vendors to commit to long term support contracts**. Make sure that all of your vendors are committed to your technologies for the planned economic life of your applications. Where possible make sure that the source code is available to you or at least held in escrow so that if all else fails you can fix any problems yourself. Above all try to make sure that a system that was planned to last for 20 years has to be replaced after 5 years because some software is no longer supported or a hardware component is no longer available.

8. **Discourage the use of proprietary technologies**. Proprietary technologies are a major source of "legacy systems" because sooner or later the vendor will decide that the product is no longer strategic or the market is not big enough. You might be really lucky and the technology will be bought by another company, but as the saying goes "Hope is not a strategy." If you have to use a proprietary technology, pick one from a small, fiercely independent small company – they have a much better track record at providing long term support than do the big companies. In the long run though your best bet is to use standardized or Free Software technologies as these both have good track records for both support and portability.

9. **Design interfaces to avoid vendor lock-in**. In the long run, even standards change, so design your applications with clean interfaces between the various components of the system. Getting locked into a particular vendor's technology is always a mistake. Sure, use whatever proprietary technology that makes sense, but make sure that if necessary it is possible to replace that technology with another. You never want to get to the state where it is cheaper to rewrite the application than it is to replace say the database or the user interface.

10. **Encourage your maintenance teams to do perfective maintenance**. Rather than make the mistake of saying "If it ain't broke, don't fix it," remember instead that your maintenance teams have to live in the code every day in order to keep their knowledge current. So encourage the teams to perfect their applications, let them adopt the motto -- "Maybe it ain't broke, but we can improve it." This will

encourage a farsightedness that will allow the maintenance teams to adjust to new technologies as they arise.

11. **Encourage your maintenance teams to experiment with new technologies**. Your center of technical excellence should be your maintenance team. Encourage them to experiment with new technologies to see how they will impact their existing applications. By casting their net far and wide they reduce the chance that they will be surprised by any new technology that shows up. After all even the latest and greatest thing like "Web Services" are firmly rooted in things like XML-RPC which has been around since 1998.
12. **Make robust, maintainable and extensible software an explicit goal for all projects**. Project teams deliver on what is important and downplay what is not important. It should not be a surprise therefore to discover that many applications that were developed in what used to be called "Internet time" are buggy, unstable and practically unmaintainable. OK, lesson learned, future projects need to have more appropriate goals.

Once appropriate team and organizational structures are in place you can start to look at design and coding techniques that improve application maintainability, but that as they say, is the topic for another article.

Points To Remember

•

Chapter 13: Becoming a Software Craftsman

"*languages, platforms, protocols, companies come and go - but crafting quality software that matters remains a wickedly hard thing to do well.*" – Grady Booch on an Extreme Programming mailing list

Looks at how far the reader has come and looks ahead to the rest of the journey towards mastering software development.

This chapter will also give hints about databases, web development and graphical user interfaces. The main focus though will be on how the skills learned so far can be translated into larger projects and how they will carry into other OO development environments.

This chapter closes with a short section looking at the future of software development.

Points To Remember

•

Annotated Bibliography

Lists all books referenced with a short paragraph explaining why the book is important and the ideas that can be gained from the book.

Glossary

This will define the key terms used in the book.

Epilogue

Follows on from the preface and basically invites the reader to continue on their journey.

Acknowledgments

Put at the back of the book to avoid clutter at the start.

Index

Needs to be relatively complete as far as the programming and software development concepts go.