# Working With Others


Michael Heron
(drakkos@discworld.atuin.net)

## Beta Draft

# Table of Contents

# Mojo The Monkey Says...



If you wish to use this material for non-personal use, please contact the authors of the texts for permission.

If you find these texts useful and want to give less niche programming languages a try, come check out http://www.monkeys-at-keyboards.com for more free instructional material.

My apologies for the unfriendly legal boilerplate, but I have had people attempt to steal ownership of my material before.

Please direct any comments about this material to drakkos@discworld.atuin.net.

That's mojo at the top right. He's very clever. He has a B.A in Nanas!

# Playing Nicely With Others

## Introduction

I know it sounds horribly touchy-feely – we're game developers, not teenagers on a camping trip. However, the most vital skill for life that you'll pick up from being a Discworld creator is how to work with other people. Absolutely everything we do is a collaborative exercise.

In terms of your immediate environment, you are part of a team in your own domain. However, before too long you start to collaborate within the larger context of creatordom as a whole. At that point, your ability to work with others in a large-scale development environment is perhaps your most valuable asset.

## Whole New Skills

It's very rare that developers work within as close quarters as we do on Discworld – thus, even those with considerable coding experience are going to find this a largely unique experience. There are things about multi-developer environments you just don't learn until you start working in one.

I teach software engineering at my local university. The students I teach are competent coders, who have even got a little bit of group-work under their belts. None of them appreciate the intricacies of environments such as Discworld because you simply have to be part of it. The things that software engineering courses teach in the abstract are things you are going to learn about first hand.

It's important to provide some caveats here. First of all, working well with others doesn't mean you have to like the people you work with. It's always better if you do, but perfectly satisfactory collaboration can occur even when the participants hate each other. Gilbert and Sullivan for example had a notoriously quarrelsome relationship, but it didn't stop them penning some enduring popular works. Actually liking people isn't necessary - successful collaborations can be born from affection, trust, or respect. Ideally you have all three, but one is enough to build a working relationship. The problem comes of course when none of these are present, but those circumstances are thankfully quite rare.

It doesn't matter if you like the people you work with, it only matters that you can work effectively with them. We can't force anyone to feel a different way about another person than they actually do, nor would we want to.

Part of the skill-set you need to build as a creator is the ability to successfully collaborate. That involves a whole lot of concepts that are new to most people – it involves understanding a complex and dynamic social context, as well as understanding the software development process. It involves becoming familiar with technologies that are often for rather alien purposes. In short, it's learning a whole lot of entirely new skills.

People tend to look down on MUDs as development environments because of the stigma attached. Most MUDs are vanity affairs in which a handful of coders put together an ad-hoc, unprofessional game based on some stock code-base. Such games rarely have more than a dozen or so players, and those players tend to be drawn from already formed social circles.

Discworld is not one of these MUDs… Discworld has existed since 1992, and had over a thousand developers working on it at one point or another. There is over a gigabyte of source code, spread over seventeen administrative domains. There are objects in the game that have been in constant use since the MUD was first opened. There is an extremely complex object hierarchy and system of handlers in which very subtle interrelationships of code cause the strangest and most bizarre observed behaviour. In short, it's far more complex than the vast majority of ‘real world' developments.

We are also a volunteer environment, and that introduces a whole range of new issues. We don't expect people to know how to code. We don't expect people to understand formal software development. We don't expect people to know about the complex etiquette that goes along with multi-developer environments. Over the years, that has led to an adoption of code written to dozens of different standards, in dozens of different styles. That causes many problems.

It's hoped that this material makes you understand the importance of some of the fairly abstract things we tend to insist on, and why they are not arbitrary exercises in nit-picking. There's a good reason why we ask you to do all of the things we ask you to do.

# Standard Standards

This is a rather grand heading for something we don't actually have...

Within Discworld, we all realise that those who come as developers do so as volunteers. That means we have fairly limited leverage in forcing a particular agenda. To be sure, domain leaders have the authority to hire and fire within their own domains, but we much prefer to have people working with us than not. As such, things like our style guidelines are only inconsistently followed.

It's my hope that, after reading through this material, you understand why we ask for these things, and that through knowing the intention you'll actually be motivated to follow the standards. It's too late for a lot of people, but if you're just starting out with us it's a fantastic opportunity to get into the habit of writing code that is structurally clean.

I would like to add a word of warning here – the things that I am going to talk about are surprisingly emotive issues. You will find creators trying to insist that their particular formatting style is the best, sometimes 'humorously', sometimes not. This is an extremely unhelpful situation, and I would ask you to ignore any of these comments.

It honestly doesn't matter what standard of code that is adopted, the only thing that matters is that everyone uses it. There is virtually no difference in code readability from one standard to another, but it dips dramatically when everyone is using their own standard. We compromise a little so that we all have a more pleasant development experience.

I am in no way saying that the style of coding that is outlined in these documents is the best way to layout code. I am making no value judgements at all – however, we need to decide on one standard and this is the one we're going to use. We'll talk about that in the next chapter.

# Professionalism

Volunteers we may be, but we do like on the whole to maintain at least a veneer of professionalism. Some of us are worse than others at that, but it's an ideal to which everyone should aspire.

That means, your personal issues with someone shouldn't get in the way of you fulfilling your obligations as a creator. If you and another person are working on a project, then you have to put aside your disagreements enough to allow a working relationship to emerge.

As far as is possible, you should keep personal issues off of the public channels. If you feel you need to tear into someone for their (in your opinion) gross incompetence then do it in tells. Otherwise it's just awkward for everyone. Failure to do this is only going to get you a reputation as someone who doesn't ‘play nicely' with others, and if that persists the only real option is for you to be removed as the obstacle you are. This is an unusual step taken in rare situations, but there are precedents of people who just could not get along with anyone who are no longer creators.

It's often harder with some people than it is with others… in life, there are just people who rub you up the wrong way no matter what they say or do. Your best bet in such circumstances is to simply try and maintain a degree of civility when interaction is required, and avoid them otherwise.

In situations in which you simply cannot resolve your differences, it's worth looking for a mediator – ideally someone of higher rank than both so as to allow for ‘binding agreements'. If your problem is with someone in your domain administration, you should arrange for a discussion between all members of the domain administration team to see how the situation can be resolved.

In all cases, you want to provide a framework for constructive engagement with your colleagues. Where that can't be achieved, you need to find a way to simply be around them. Nothing sows more disharmony into your fellow creators than a persistent and public slanging match. The creator channel, and the boards, are not the place for that kind of thing.

# Conclusion

We're going to cover a lot of ground in the chapters of this material, including ground that will be entirely new even for a lot of experienced developers. In all cases, I am going to ask you to engage with the material and not dismiss it as an irrelevance. As I have mentioned above, there is a reason why we ask you to code to a specific style. Although we can't really force it in the same way that can be done when people are being paid for their effort, it makes the MUD a much nicer place for us all to develop if we can rely on a little professional courtesy from our colleagues.

# Code Layout

## Introduction

There is nothing that will make your code more readable than having a clean layout. Inversely, there is nothing that will make your code less readable than having a bad layout. There is a set coding standard than we have on Discworld, and while it has been inconsistently applied over the years it is something you should try to get into the habit of before you are Too Set In Your Ways.

This is a surprisingly emotive issue for some people – for some reason, people seem to invest a lot of their own self-worth in the choices they make in terms of laying out code. The truth is, it doesn't matter in the least which standard you choose to use – they're all equally readable. The only thing that matters is that everyone uses the same standard. Don't be one of those creators who stubbornly refuse to compromise on this point – it's stupid, and actively unhelpful. Likewise, don't pay attention to those who try to force their own standard upon you. Just roll your eyes and move on.

## Code Formatting

This is going to be a rather dull section, but it's important that we talk about it. I've already said this but I'm going to say it again – having a clean style for code is the most important thing you can do to make your code readable. The standard that we apply is as follows:

- Indent two spaces per level of coding structure.
- Lines of code no longer than 79 columns.
- The opening brace of a structure is placed on the same line as the structure to which it belongs.
- All defines should be in UPPER CASE.
- Functions are all in lower case, with an underscore separating words. In java, a function might be thisIsAFunction. In LPC, that would be this_is_a_function.
- All for loops and if statement to have opening and closing braces, even if they are not syntactically required.
- Use spaces, never tabs!

That's all – it's not much to remember, and once you get into the habit of it you'll do it subconsciously. When I started coding on Discworld, my own personal standard was contrary to all of these. As time went by, I migrated towards the Discworld standard because it made things much easier for everyone involved, and it came as no real cost to me.

Let's look at two bits of code, one without formatting, and the other formatted to Discworld standards. First, without formatting:

```
void this_is_a_function() { for (int i = 0; i < 100; i++) {if (i % 2 == 0)
{tell_object(this_player(), i + " is an even number.\n");}else
{tell_object(this_player(), i + " is an odd number.\n");}}}
```

And then formatted to our internal standards:

```
void this_is_a_function() {
  for (int i = 0; i < 100; i++) {
    if (i % 2 == 0) {
      tell_object(this_player(), i + " is an even number.\n");
    }
    else {
      tell_object(this_player(), i + " is an odd number.\n");
    }
  }
}
```

Hopefully the latter example is obviously more readable. There are also coding clues given for you – indenting to a different level depending on the depth of the structure gives you an instant visual hint as to where opening and closing braces should go. It demonstrates ownership – you know that the if statement belongs to the for loop, because that's what the indentation shows.

Look at those two samples again, slightly altered:

```
void this_is_a_function()   {  for (int i = 0; i < 100; i++)   {  if (i % 2
== 0)   {  tell_object(this_player(), i + " is an even number.\n");  }  else
{  tell_object(this_player(), i + " is an odd number.\n");  }  }
```

This code won't work, and it's not immediately apparent why. On the other hand, if we reformat it:

```
void this_is_a_function() {
  for (int i = 0; i < 100; i++) {
    if (i % 2 == 0) {
      tell_object(this_player(), i + " is an even number.\n");
    }
    else {
      tell_object(this_player(), i + " is an odd number.\n");
    }
}
```

The eye is instantly drawn in the second example to the fact a closing brace doesn't exist where we would expect it to. The layout actually makes it easier to code.

This additional readability can be lost when multiple people with different coding styles work together on the same file:

```
void this_is_a_function() {
  for (int i = 0; i < 100; i++)
  {
    if (i % 2 == 0) {
      tell_object(this_player(), i + " is an even number.\n");
    }
    else
      tell_object(this_player(), i + " is an odd number.\n");
      tell_object (this_player(), "This is a line that appears on "
        "every number!\n");
  }
}
```

Rather than this code being easy to use because standards have been applied, it becomes harder to read because **inconsistent** standards have been applied.

Now that we've seen the difference the formatting makes, we'll talk about each of these rules in turn and why they are in place:

## Indent Two Spaces Per Level of Coding Structure

There's no magic formula as to why two spaces is best, other than it gives you slightly more screen real-estate to work with while still showing the relationship between coding structures. I will emphasise something I said before – people will try to convince you that the number of spaces they use is a better way of laying out code than the number of spaces this document tells you to use. Pay these people no heed, for they are deeply tedious. There is no real difference between two spaces, three spaces, or four spaces. The only thing that matters is that all agree to use the same level of indentation.

## Lines Of Code No Longer Than 79 Columns

This is a readability issue – many people are still working within the 79 columns of a standard telnet display. If you have code that goes over that line length, it is virtually unreadable.

For example:

```
set_day_long ("This is a lovely stretch of beachfront along the coast of the
"
"mysterious Pirates Cove.  To the south can be seen the buildings of the "
"pirate settlement that has grown on the island.  The masts of tall ships "
"pepper the horizon like the spears of an approaching army.  Pirates Cove i s
"
"a popular stop for the many rogues who journey the Circle Sea, and the who
le "
```

Sadly, that's from a piece of my own code...

It makes even room descriptions hard to read, so imagine what it does for complicated code structures.

## The Opening Brace of a Structure

Once again, there's no reason why this should magically be so – it's just that in order for there to be a standard, everyone has to do the same thing. As with the two space rule, it provides a little extra real estate on the screen when viewing things in the very restricted environment of the MUD.

## All Defines Are In Upper-Case

Having an ‘at a glance' way to tell which values have been defined and which are drawn elsewhere from the code aids tremendously in readability. Moreover, when defines aren't in all upper case, it dramatically detracts from readability because everyone expects the alternative. Violating that assumption has a measurable impact on code comprehension.

## Functions Are In Lower Case

This is usually a convention of the language rather than a convention of Discworld particularly. The stylistic conventions obeyed as part of the internals of the programming language define how our functions are to look. The MUD's efuns for example use *this_kind_of_standard* and so that's what we use for our own code. Otherwise we need to make a mental check each time we use a function – ‘is this an efun, an sfun, or an lfun?' and choose the formatting accordingly.

### All Structures To Have Opening and Closing Braces

LPC allows you to omit these on *for*, *while* and *if* structures if you have a single line of code to be executed:

```
for (int i = 0; i < 100; i++)
  tell_object (this_player(), "The number is " + i);
```

This is syntactically correct, and it will work as intended. However, when you come back to add more complex functionality you need to remember to put the braces in, and those who work with your code need to be observant enough to notice that you haven't already put them in place. You gain nothing from omitting them – it's like using an indicator in a car, you should use it even if you're alone on the road because it's easier overall when such an activity is an unconscious rather than conscious decision.

This is an example of an area in which competing formatting standards will actually cause a decrease in readability. Imagine person one, who indents to two spaces and doesn't use opening and closing braces. then person two, who indents to four spaces:

```
for (int i = 0; i < 100; i++)
  tell_object (this_player(), "The number is " + i);
  tell_object (this_player(), "Some stuff\n");
```

The visual clue here suggests that these two statements are part of the same for loop. In fact, only the first belongs. This kind of ambiguity can be reduced by simple layout and clarity of expression:

```
for (int i = 0; i < 100; i++)  {
  tell_object (this_player(), "The number is " + i);
}
tell_object (this_player(), "Some stuff\n");
```

Again, it costs you nothing to put in the braces, so you should get into the habit of it being a ‘muscle memory' thing rather than a conscious choice.

## No tabs

If you put a tab in your code, it creates a very ugly visual artifact when you read it on the MUD – it gets interpreted as <TAB>:

```
bookcase->set_long( "This bookcase is made from oak and "
  "varnished to bring out the glow.  It has 2 shelves, "
<TAB>"upon which you can see some books, and other objects.\n" );
```

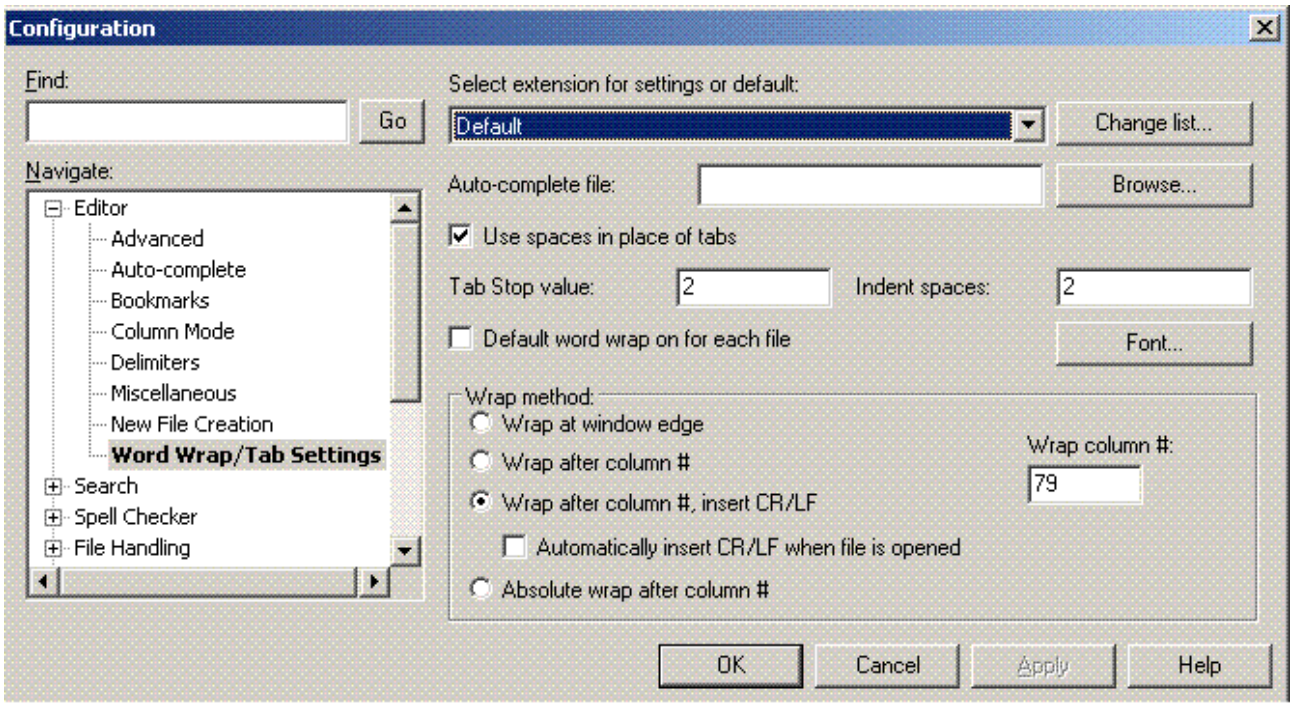
Bleuch. That's horrible!

Instead, use spaces, never use tabs. Luckily, if you are using a good text editor you can get the best of both worlds – you can tab as much as you like, and the editor will simply interpret it as a set number of spaces.

## Make It Easy On Yourself

Provided you have a good text editor, a lot of this can be handled for you. We'll use Ultraedit as our example of this. Other editors will undoubtedly have similar facilities, but these are left for you to discover.

First of all, we want to remove the tabs from our code. This is the most important first step to take. Go to Advanced->Configuration, and that will open up the configurations editor. Navigate to ‘Word Wrap/Tab Settings':




Three of our rules can be automated for you – make sure ‘use spaces in place of tabs' is selected. Notice here that we can set the Tab Stop value and the Indent Spaces value – set both of these to two. Finally, you can also automate adherence to line lengths by setting the wrap method. Ultraedit will thus do a big chunk of the work for you, without you needing to worry about it all.

# Conclusion

Standards are a good thing, and it would be tremendously helpful if you could get into the habit of writing your code to them. Over the years we have had an inconsistent approach to formatting, varying across domains, individuals and even time periods. While it may have been more convenient for single people, we are a team trying to achieve a collaborative goal. Everyone has to be willing to compromise on this to make the whole project work together better. In the case of code layout, you're not even being asked to compromise much – no matter how fond you are of your own particular style, it is not so much better than any other style that it justifies the lack of clarity that comes from inconsistency.

Additionally, please ignore those people who try to force you to deviate from these standards with mockery, or bizarrely strident advocacy. There is nothing Big and Clever about trying to undermine any effort to increase consistency of code across a massive developer-base. In real world coding environments, you code to the set standard or you lose your job – that's not an option we like to consider here on Discworld – the best we can do is appeal to your presumable desire to be a useful, valuable member of a highly integrated and collaborative team.

# Collaboration

## Introduction

We are big on collaboration on Discworld. At least in theory. In actuality, everyone has their own particular approach to how and when they collaborate. However, game development is an inherently collaborative endeavour, as the things that you make available in the game will have an impact on many other things. If you make a bank available, it alters the flow of money for all domains. If you add a vault, it affects game performance. The connections are incredibly complex.

We have several good tools in place for enhancing collaboration, but they are for naught if the will to collaborate is not present. In this chapter we'll talk a bit about the collaboration styles you will tend to encounter, both here and in 'real life' environments.

## The Social Context of Collaboration

The social context of an environment is one of the key elements in fostering an atmosphere that supports collaboration. Every context has its own particular features.

Discworld has a strong tradition of meritocracy in advancement, and this meritocracy is usually demonstrated through a system of emergent authorial leadership. In essence, you progress by showing yourself to be a 'safe pair of hands' on the basis of the projects you are involved with and the contribution you make. In addition to this is the value ascribed to seniority – combined with the authorial leadership, authority accrues to those who have been around sufficiently long to be considered 'tribal elders'. This pattern is also reflected in the playerbase, where a distinction is made between 'newbies', 'midbies' and 'oldbies'. The combination of these two social dynamics is common to many collaborative, volunteer endeavours.

Collaboration is enhanced by the further tradition of 'ownerless code'. Code on Discworld does not belong to any particular creator, although one creator may take a greater or lesser interest in its upkeep and maintenance. Code instead belongs to a domain in the first instance, and the MUD as a whole in the second. It is not only possible for another creator to modify code you have written, it is actually an active part of the development context. Everyone owns the code in their domain, and there are creators who have wider responsibilities that work across domains. It's important that you understand your code is Communal Property, otherwise you'll find it very difficult to cope – especially if your code is important enough for people to take an interest in.

A sense of shared responsibility over code, and a tradition of authorial leadership, are important traits in a successful collaborative environment. Although we predate it by a Good Long While, these features are apparent in one of the most successful of modern collaborations – Wikipedia.

However, the social context is modified by the people who are involved, and some people simply do not collaborate. Our environment facilitates collaboration but doesn't mandate it – you don't need to take anyone's views on board while doing your development with the exception of those of your domain administration. Some people work best like this, but it's not a mindset which we like to encourage.

There is often a generational gap that comes along with willingness to collaborate. Older developers may be less willing to engage in such a process, because they perceive development as a solitary effort. This is not something that is especially pronounced in Discworld, but it can be observed in other environments. Younger developers now grow up in an atmosphere of extreme collaboration, brought on by a culture of social networking and the prevalence of shared wiki tools. Older developers are less familiar with this as a mindset, and so are often somewhat resistant to broad and indiscriminate collaboration, preferring to collaborate instead with a few hand-selected and trusted colleagues.

Different people have different ideas about what collaboration actually means. Does it mean the intense collaboration of something like Wikipedia where changes are small but accumulate with the weight of an avalanche? Or does it mean that one person writes something, and another person writes a bit, and the first person writes a bit more – essentially serial development. Or does it mean that both individuals make their own attempt, and then the best of these two attempts are merged together? All of these describe different, but perfectly acceptable, models of collaboration.

You also tend to encounter one or two people who actively disapprove of collaboration. It's not just that they don't collaborate themselves, but they actively despair of collaboration in general. Authors such as Jaron Lanier have written of an encroaching 'digital Maoism' in which individual ability is swamped by the mediocrity of averages.

When building a social context, it is important to start with the people. For collaboration to occur, first and foremost there must be a will to collaborate. Some environments are set up in such a way that collaboration quite simply will not happen. In a now-famous paper, Wanda Orlikowski discusses an attempt to introduce a groupware product to a team of consultants; the implementation failed, due to endemic social issues stemming from competition and no tradition of mutual trust, as well as a deep lack of communication as to what the tool was for.

The mindset many developers adopt when introducing collaboration technology is what I like to refer to as the Field Of Dreams mindset – 'if we build it, they will come'. Experimental evidence however shows that this is hardly ever true – people have to see a need before they make use of the tool. Collaboration tools work only if they make existing social processes easier to mediate.

# Development in Volunteer Environments

Why do people choose to give their time and effort, for free, to a cause like Discworld? Everyone is going to have their own reasons for this, but there are certain commonalities within different projects.

Many people report 'altruism' as a reason for participation. Whether altruism actually exists or not is a philosophical quandary, but what can't be denied is that people often feel a pull towards a cause in which they believe. Presumably you enjoyed your time playing Discworld, and felt sufficient draw to the game that you wanted to devote your time to making it better. This is something reported often in open source communities.

However, there are many additional benefits that come from participating in a project like this. For one, you develop many skills that are genuinely marketable. A number of Discworld creators have profitably included their development experience on their CVs when applying for jobs, and attribute at least some measure of their success in those interviews to the skills they have developed here. Many professional developers despair of the lack of attention paid in university educations to 'operational skills' such as dealing with source control – Discworld gives developers exposure to the complexities of working within a codebase of quite staggering complexity and size. That's something that sets you apart right away from many other developers.

There is an interesting element to how people choose to join as creators in the first place – unlike most volunteer coding movements, Discworld does not incorporate simply any change from any interested developer. Instead, Discworld is a 'hybrid open source' environment in which the game files are secret, the driver is freely available, and public releases are made of the core mudlib. Self-selection of contributions is a big feature of environments such as Linux or Apache, but it is not reflected in our approach to development. Developers self-select in so far as they choose to apply, but the process is much more like applying for a job than it is developing for Linux as a movement. This has issues of scale that often manifest themselves – when a domain leader is absent, a domain can grind to a halt.

The administration of a domain is responsible for the ultimate vetting of quality. The playtesters domain is an opt-in service for those domain leaders who wish to make use of it, but the exact system for determining whether or not a development is to go into the game varies from domain to domain. Some domains make extensive use of peer review (as Forn did during its development of Genua), while others have more informal processes in place.

# What Are The Benefits of Collaboration?

There are several benefits that come from collaboration. Linus Torvalds, the man responsible for building the first version of the Linux kernel, is credited by Eric Raymond with formulating Linus' Law. This states, informally, that 'given enough eyeballs, all bugs are shallow'. Collaboration allows us to harness the different skills and abilities of many people – we end up with the whole being more than the sum of its parts.

It's often surprising the wealth and depth of experience that is available when you widen the parameters of your search. In any room of average people, you'll find individuals with the strangest combination of skills – some because of their occupation, and some because of their hobbies. The combination of skills that people develop over the course of their lives lends a unique perspective to their views and opinions – everyone views the world through the lens of their own experience.

Aggregating the views of people with multiple sets of skills and abilities can lead to results that are better than any single individual is capable of producing. The book 'Wisdom of the Crowds' by James Surioweicki is an extremely interesting discussion of this, and it is very relevant to the idea of intense collaboration in programming environments.

# Collaboration Tools on Discworld

We have numerous tools for persistent communication and collaboration within Discworld. At the simplest level is the internal mudmail and board system – these allow for communication, and at the core that's what collaboration is all about. However we also have two tools that fall into the more modern category of 'collaboration software'. The first of these is our extensive wiki system – we use the TWiki engine — with each of the main domains having its own wiki web for collaboration. Some of these are quite extensive, while others are used infrequently.

There is a point of critical mass that needs to be reached for such tools to get momentum. Simply using a tool, even if no-one else is using it, can generate interest, and that can in turn generate further contributions. All it takes is one person to get the necessary traction. Encourage people to read your contributions – direct them to your wiki page when you're asked questions. Get people to look, and you might just get people to join in.

The Wiki may be found at http://discworld.atuin.net/twiki/bin/view/Main/WebHome. Have a read through – you may be surprised at what you find.

The second tool we have is our bespoke knowledge management software – the Discworld Oracle. This is a system for collaboratively eliciting the considerable knowledge and expertise of our creator base. I would encourage anyone who has a question, no matter what the question may be, to check Oracle to see if the information is available, and then ask the question if it is not. Everyone who asks a question is engaged in the task of knowledge elicitation – you are helping to make information available for all those who follow.

The Discworld Oracle may be found at http://discworld.atuin.net/lpc/secure/creator/oracle/oracle.c. Please contribute anything you think might be of interest, and ask any questions that come to mind.

# A Suggested Collaboration Process

First of all, your task is gathering ideas. You'll undoubtedly have many of these yourself, so create a wiki page for your project and outline them. You'll find there are people in the creatorbase who read every change made to the wiki (I'm one of them, I'm a wikiholic) so even if no feedback is received it doesn't mean your contribution hasn't been read. Update the page as thoughts occur to you – it can be a useful project planning document for you and for your domain leader.

For a domain leader, keeping track of where each project is and how complete it is is a complex task. The best thing that you as a developer can do to ease this task is to keep your own developer page up to date - that can be tremendously helpful. If you are developing a specific area, you might want to consider making an abstract of the area available on your domain wiki, outlining features and quests. Have a look at http://discworld.atuin.net/twiki/bin/view/Forn/MainGate as an example of this; the entire city of Genua can be navigated in the abstract, and this was a very valuable tool for when we were making sure the city was feature complete. Small areas may not lend themselves well to this, but it's great when you can make use of it.

Once you've got your wiki page up and running, try making a post to your domain board outlining what your plan is and where the wiki page can be found. A simple request for 'any thoughts people may have' can elicit many useful suggestions (though sometimes, no suggestions at all). Occasionally you will find that a project has interest to someone beyond the people you would normally expect, and that can lead to profitable, albeit unexpected, collaboration. As an example of this, if you were coding an island full of pirates, it would certainly be of interest to me as part of the development of the piracy system.

If anyone expresses a particular interest, have a chat with them to see the level of their interest. You may find people who are interested in supporting particular parts of the development, or who are interested in hooking in code of their own. All of this is a great opportunity to make your area richer than it would be when developed from the perspective of an individual.

Check with your domain administration to see if you can canvass the playtesters for suggestions. The ptforum board is a great place to see what ideas they may have for things they would like to see, or things that they definitely wouldn't like to see. The more perspectives you can solicit, the more of a pool of good ideas you'll have to choose from.

It's important to note here that this doesn't mean you abdicate ownership of your project – you are soliciting feedback, but that doesn't mean you're obligated to use it all. It's just that getting a wider range of perspectives will give you a much better foundation from which to develop your thinking.

# Conclusion

Collaboration is an entirely social problem – you should not confuse the tools with the concept. Our tools exist to support existing social dynamics, not supplant them.

Collaboration is an important part of what we do on Discworld – everything impacts on everything else in very complex and complicated ways. Our unique cultural makeup has led to the emergence of certain organisational norms – a shared ownership of code, authorial leadership based on meritorious contribution, and a general respect for those who have contributed long enough to have become 'tribal elders'. All of these help support an environment in which collaboration can flourish, but it still requires critical mass for it to be effective. You can either be a barrier to collaboration by looking inwards, or you can be a spark for further collaboration by doing your best to solicit feedback.

# Social Capital

## Introduction

Social Capital is the glue that keeps a society together. As a term, it refers to the reserves of trust, respect, collegiality and norms of reciprocity that exist in social networks. It's a measure gaining considerable traction in sociological and economic debate – while it can't be quantified, it provides valuable qualitative analysis of the level of function and dysfunction in an organisational environment.

What we're going to talk about in this chapter is how social capital is built in a online technical context like Discworld. The creator-base has a rather extreme reputation for 'creator politics', but in the main this stems from a handful of isolated but extreme problems, rather than being a systemic feature of the environment.

## Creator Politics

The fact is that creator politics are nowhere near as endemic as the popular player perceptions would indicate. Player perceptions are distorted by the handful of disproportionately loud examples of ex-creators who were either fired, or resigned, because of their inability to integrate into our working environment. Those who complain the loudest about not being promoted because of 'politics' are those who, invariably, have not played their part in engaging in the collaborative process of building a lasting reserve of social capital.

That's not to say that there are no politics – as soon as you put more than one person in a room, politics suddenly happen. Politics is the word we give to the necessary friction and abrasion that comes from people having multiple, perfectly valid, viewpoints.

Many outlandish claims are made about creator politics. There are claims of institutionalised nepotism, lasting grudges, and projects that have been killed because the wrong person was involved with them. While there will be examples of each and every one of these, they are not widespread – they stick out precisely **because** they are not widespread. Follow the gingerbread trail of rumours to their sources, and you'll find that a small number of bitter ex-creators are responsible for their perpetration. These creators, without exception, found it difficult to work as a creator because of their own unwillingness to engage fully in the process.

So, put these complaints in perspective. The politics of Discworld creators are no more cut-throat than the politics you will encounter amongst any group of people. If you are mature enough to try and work with other people, you'll find they are willing to try and work with you.

# The Ten Commandments Of Egoless Programming

Way back in 1971, a guy called Jerry Weinberg wrote a tremendously influential book – the Psychology of Computer Programming. In this book, he outlined a mindset he termed 'egoless programming' as a way to deal with the often emotive issue of ensuring quality in software development projects. His system revolved around the ten commandments that tend to lead to a more positive, collegial relationship between software developers. As a 'consciousness-raising' exercise I would like to outline them here because they serve as a useful set of precepts for how trust and respect flow in development. The commandments are his, although the commentary is mine.

## Understand and Accept You Will Make Mistakes

We all make mistakes – some of us more than others. The consequences of these mistakes may be minor, or they may be a major inconvenience to the entire user-base of the MUD. I once made a particularly bone-headed error that locked the MUD up tight for a good hour, something that would have been impossible to do if I hadn't explicitly switched off the sanity checking built into the driver. You will make mistakes – learn from them, and move on.

## You Are Not Your Code

When people criticise your code, they are not criticising you – at least, they shouldn't be criticising you. If they are, then it's a problem with them personally. Constructive criticism is very valuable – it's how you learn from people with a little more experience. In order to accept constructive criticism, though, you need to divorce yourself from the code you have written – you need to be able to take a dispassionate view and say 'Ah, yes – it does indeed have defects I need to address'.

## No Matter How Much "Karate" You Know, Someone Else Will Always Know More.

It doesn't matter how good you are – there's always someone better. There's one person in the world for whom that isn't true, and that person is perpetually looking over their shoulder for the day that it is. Moreover, everyone has their own particular areas of expertise – even if you consider yourself the Top Guru in a particular area, someone else is going to know more in another. Creating on Discworld requires a very odd blend of skills, and some people have these skills to greater or lesser extents. Even if you know you're good, don't let it go to your head – the chances are you're not as good as you think you are.

## Don't Rewrite Code Without Consultation

We don't encourage creators to 'own' code on Discworld – code is a communal resource. However, that doesn't mean you can write code without regard for other people. This is especially true when you are working with lower level inherits and critical handlers – there is an etiquette that goes with rewriting code, and it is vital you adhere to it. If you are going to do some serious remodelling of important code, then make sure you consult with the people who are likely to be affected.

## Treat People Who Know Less Than You With Respect, Deference, And Patience

I have my doubts about 'deference', but the general rule is strong. Everyone was a beginner at one point, and if those to whom we had turned for help had mocked and dismissed our queries, the chances are none of us would be here at all. It's an act of considerable courage to ask for help, and it's in everyone's best interests for every creator on Discworld to be as good as they can be. Encouraging and constructively engaging with creators who have queries is the best way to foster an atmosphere in which self-evaluation and improvement is possible.

Additionally, the very act of asking a question can add value to the creator-base. A good question will tax the understanding of the teacher as well as the student; the teacher gains a little extra clarity, and the student gains the understanding they desire. If questions are asked through the Oracle system, then a good question is worth its weight in gold to the creators who come after you.

## The Only Constant In The World Is Change

On Discworld, we are not beholden to the whims of our clients. Our clients are the players of the game, and they play the game which we provide. In other organisations this is not the case – when a client asks you to make a change, you make it.

However, even though we are the ones setting the development agenda, we all still need to learn to deal with changes in fundamental coding tools and systems. Changes elsewhere in the game will impact on the code you are writing, and you must be willing and able to adapt your code to deal with emerging situations.

## The Only True Authority Stems From Knowledge, Not From Position

Those who have attained higher rank in the creatorbase have usually done so on the basis of their contributions to the game. While you should accord people the appropriate level of respect, you shouldn't confuse position with authority. The newest creator may have more knowledge of a particular area of the game than the most senior creator, and you shouldn't allow the position of the latter to override the expertise of the former.

## Fight For What You Believe In, But Accept Defeat Gracefully

Everyone has a different view on what's important in the game, and everyone has a different view on what will improve the game. It is important that you fight for what you feel to be right, but you also have to realise that the authority for making the ultimate decision usually does not reside with you. In such circumstances, it's important to let go of the debate and accept the conclusion, even if you personally disagree with it. We've all had to live with the consequences of decisions that we didn't like.

## Don't Be The "Guy In The Room"

The "Guy In The Room" is the one who doesn't engage with the rest of his or her team. The Guy has a project, and writes the code for that project without collaboration or input from others. The "Guy In The Room" doesn't know what's going on and isn't really a part of the team. Not knowing the broader context in which code is being developed and deployed is a major disadvantage in an environment like Discworld, and you'll end up doing yourself more harm than good.

## Critique Code Instead Of People – Be Kind To The Coder, Not To The Code

Too many people don't understand what 'constructive' means. Additionally, too many people deliver criticism without even a basic understanding of human psychology – there is a reason why such people very rarely manage to convince others of the worth of their remarks. Critiques should focus on the code, and not the coder. 'You write code that is full of problems' is an attack on a person, and that's never going to get someone on side. 'There are some problems that need to be resolved with this code' focuses the remarks where they belong. Of course, if Commandment One is not being observed, it's not going to make a lot of difference.

The number of people who don't understand the most productive way to deliver criticism is staggering. In the main, it's how you do it rather than what you say. To begin with, concentrate on everything that is right about the artifact in question – start with the positive, and then introduce a discussion of the negatives. If necessary, interleave positive and negative feedback to ensure you're not simply giving a laundry list of flaws. The important thing about positive feedback is to front-load it – it's counterproductive to start off with negative feedback since it just puts people on the defensive, and even if there are positive comments at the end, you've already lost the chance to win someone around to your way of thinking.

Inability to deliver criticism correctly is the leading reason why some people just can't get others to listen to their feedback.

# Trust and Common Ground

Perhaps one of the trickiest aspects of this system is that it requires you to be able to trust and respect your colleagues. Trust is the alchemical property that makes sure teams keep functioning even when there are breakdowns in communication, conflicts of interest, or simple personality clashes.

Invariably there will be people you trust more than others, and with whom you feel these commandments can usefully apply. Conversely, there will be people you *don't* trust, either in terms of their personality or their competence – the thought of following these commandments with these people would be laughable. However, once trust is built it can work effectively to bridge the day to day problems that collaborative development will introduce.

There are several ways in which trust can be built in any organisational environment. They are based on a genuine willingness of all participants to work towards a common understanding.

First of all, trust is built on common ground – this is the set of principles and issues on which you and your interlocutor can agree. This involves a certain amount of give and take, and a willingness for each to see things from the other's perspective. Unwillingness to build common ground on the part of one participant is a sign that they are either not interested in — or not capable of — participating in a constructive dialogue.

Common ground tends to increase on the basis of familiarity with individuals, and familiarity with a particular environment. It is particularly strengthened when it involves shared experiences or a shared background. Common ground requires a willingness for individuals to compromise – in organisations, it's based on the willingness of an individual to be integrated into a complex community, rather than being an outsider unwilling to consider more productive engagement. Most organisational culture is a manifestation of common ground within a particular operational context – the 'work songs' of IBM in the fifties were comical, but they served their purpose in creating (an arguably somewhat dangerous level of) common ground.

Common ground can be maintained by simply 'keeping people in the loop'. That doesn't mean that everyone has to be updated about every little detail of your project, but it helps if you touch base with the right people at the right time. This helps resolve ambiguities about what people are doing, and prevents small problems becoming larger conflicts. Conversely, conflict causes people to withdraw from the process of building common ground, creating a self-reinforcing cycle of disharmony.

It may sound trivial, but common ground requires constant calibration. It's not something that is achieved and then you move on, it's something you continually work towards.

Even when common ground disappears, trust is often enough to ensure the resilience of a team of people over the short term. Where there is no trust, our assumptions of motivation are always flavoured negatively – we assume people are doing things for the worst reasons, rather than for the best. When there is no trust, there is no will to collaborate, and this strikes at the heart of the way our environment works. When there is no trust, mistakes are hidden rather than brought out for everyone to help resolve. When there is no trust, people are less willing to say 'I don't know', and that is no good for anyone.

So, if we know how common ground is built, how do we build trust? Sadly, this can be much harder to do in an online rather than an offline environment. Trust is built as a consequence of informal social interaction – coffee breaks, having lunch together, and idle chit chat. We do have channels for discussion, but our environment poses numerous extra challenges.

For one thing, there is an inherent ambiguity in the medium. If you ask me a question, and I don't respond, why is that? Is it because I'm not actually at my keyboard? Am I actively ignoring you? Did it just slip past as I blinked during a wave of debug spam? It's hard to tell.

Likewise, there is none of the nuance in written text that comes with face to face communication. The words themselves are only a part of what we project when we talk to someone – tone of voice, facial expression, and body language are all vital in decoding the actual intent of the words. We have smilies in text, but they do not do nearly enough to bridge the gap. Where trust already exists, one can read the best intentions into communication. Where trust doesn't exist, we can read the worst.

One of the easiest ways to completely demolish any initiative to build trust is to talk behind someone's back rather than bring it up directly – people almost always hear the backroom gossip anyway, and when they do it lowers you in their perception. My own personal rule for this is 'Never say in private what you are unwilling to say in public'. While that may get me a reputation for being 'A Bit Of A Dick', at least people are sure that if I have a problem with them, I will bring it up directly rather than passive-aggressively.

Additionally, the simple nature of our distributed developer base is problematic – it's often not possible to get immediate feedback on such communication because it was written asynchronously – we were sleeping when it was written, and now the person who wrote it is asleep while we read.

Keeping the channels of communication open are the best way to build trust – just talk to people. It doesn't have to be about the game, although if you have a project that's exciting you can go a long way by communicating that excitement. In our particular environment, individual initiative is hugely important, because it's what allows you to work when there is no instantly available authority – while you need to touch base with your domain administration, you also don't need to wait for them to approve every little detail.

Of course, this is a two-way street – you don't build trust if you're the only person doing it. Everyone needs to be willing to work towards it.

# The Trust Triad

In the end, trust boils down to three key elements. If you trust someone, you have to:

- Have the capacity for trust. Some people are so damaged by their life experiences that they simply find it impossible to let themselves trust others.
- Have confidence in their competence. The person with whom you are building a trust relationship has to have demonstrated their capability within their role.

- Have confidence in their intentions. Where ambiguity rules, it's your confidence in someone's intentions that will carry you through. If you trust someone, you have to believe that they are actually doing what is best for the game rather than what is best for themselves.

As to the capacity for trust, that's an internal measure. The only one who can build that is you, and deeply seated trust issues are well outside the scope of this material.

The other two are professional measures, and they are driven partly by you, and partly by the other person. There's a great old saying, 'Never attribute to malice what can adequately be explained by incompetence' – it's an aphorism of dysfunctionality in team work. In a team that is functioning properly, you shouldn't have to rely on either extreme. You'll simply accept the fact that 'people make mistakes' (Commandment #1), and work to fix it.

Another word for this kind of institutional trust is 'respect'. Notice nowhere does it say you have to like someone; you just need to have sufficient respect in them and their abilities that you can assume the best. You can like people and not respect them, and you can respect people but not like them. The best state, of course, is when you both like and respect them. If you can only manage one though, try for respect.

I have often made the statement that 'respect is earned, not given', which invariably causes disagreement along the lines of 'you should start off respecting people'. This disagreement, I feel, is due to a lack of common ground as to our perception of what respect actually means. The absence of respect is not disrespect. You should, by all means, be civil — even friendly — to people who you have no cause to respect. Active respect, though, demands a little more from both parties. It requires adherence to the 'trust triad' outlined above and this can't be done with someone you've only just met. They need to have proven themselves, and shown that their intentions can be trusted. That takes time, and a willingness to actually build that trust, and it needs both parties to engage in that process.

# Conclusion

For all you hear about 'creator politics', the fact is that we do try, in the main, to get along. There are always politics – that's what happens in life. You put people together, and politics is one of the by-products of whatever activity was the intention. There are examples of systemic distrust between individuals, and examples of complete breakdowns in communication. If you look a little deeper though, you can often find that the reason for such situations is that one or more of the participants have simply withdrawn from the exercise of building trust and maintaining common ground.

If you keep this as an active priority in mind when developing, you'll find it easier to function in our somewhat overwhelming world. People who get on well with people have a 'superpower' all of their own – they keep the MUD running when communications have broken down between others.

# The Dark Art of Refactoring

## Introduction

A lot of what we do as creators is tidying up code that has already been written. Technically this is known as 'refactoring' code. It has something of a reputation as a 'dark art' amongst programmers, mainly because so few genuinely understand what the process of refactoring involves. It's actually a very simple principle, albeit with technical complications that go with it.

In this chapter we're going to talk about refactoring – how it's done, why it's done, and most importantly of all, how it shouldn't be done.

## Refactoring

Put in its simplest, most accessible terms – refactoring is the process of turning bad code into good code, while not impacting on any of that code's functionality. Refactoring is ideally an invisible process – if you do it right, no-one should know you did anything at all.

Refactoring is not about adding extra functionality, although it may be a precursor to this. It's also not about fixing bugs, although bugs may disappear as a consequence. Many of the oddest random bugs are a result of badly structured code, and cleaning up the internal architecture of a problematic object can result in real, observable improvements even though that was not the actual intention.

## Good Code

Part of the problem people have with refactoring as a process is that it is inherently subjective. It's usually pretty easy to identify bad code, but it's much more difficult to identify good code. Different coders will vary in their opinions as to what exactly a good piece of code looks like, and this subjectivity is at the heart of what makes refactoring somewhat non-intuitive. As you grow more experienced as a developer, it becomes easier to identify code that you personally class as good - that judgement is built on the basis of experience. 'Ah, yes – I've worked with code like that before, and it was easy to make my changes'.

Since refactoring is the process of turning bad code into good code, we need to have a pretty solid grasp of how the code should be improved. You need to be sure your change won't introduce new problems – there's no point after all rewriting bad code so it becomes different bad code. We want the quality of the code to actually increase from our efforts.

# Impact of Change

Before we get to the discussion refactoring properly, let's talk about a concept that doesn't get enough discussion on Discworld – the **impact of change**. In its simplest term, this relates to the number of objects that will need to be altered if you change core functionality in another object.

Discworld has, in the main, two kind of objects that carry with them a potentially high impact of change. The first are handlers, such as the armoury and the taskmaster. The impact of change that goes with each will vary with how widely they are used. To get a clear picture in your mind, think 'What would happen if I broke this object right now?'. If it's something like the state change handler, it may go unnoticed for a short while. If it's the taskmaster, everyone will be complaining in seconds. This gives you a rough measure of the impact of change.

On the other hand, if you break a single room in an area, then that one room becomes inaccessible. That's not a huge deal. Break the inherit that every room in that area uses, then the whole area becomes inaccessible. That's a bigger deal. Break /std/basic/room and the entire game world becomes inaccessible. It is this that defines the impact of change.

On the whole, you can get by with four categories of object:

| Object Impact | Examples |
|---|---|
| Critical | Core handlers, the lowest level of inherits (the ones that every single object inherits). |
| High | Other mud-wide handlers, and the rest of the inherits in /std/. |
| Medium | Local area handlers, area level inherits |
| Low | Single rooms, single NPCs, single items |

That's one measure of impact of change. The other measure is how many objects make use of the functionality of another object. There is usually a fairly close overlap between categories.

There are rules that go along with refactoring, and one of these rules is to be very careful when dealing with objects with high impact of change. Object orientation as a programming framework also has a number of tricks for making objects as amenable to change as possible. We'll talk about them in this chapter also.

# The Rules

Here are the rules that go with refactoring. Note, these are rules, **not guidelines**. A good software development process will obey these rules and have punishments for transgressing them.

1. Methods and variables may be made more visible. They may not be made less visible.
2. The functionality of public methods cannot change. If a public method does X, it should continue to do X (and nothing more or less) after it has been refactored.
3. The return type of a method cannot change unless that change is for it to be less restrictive (from a string to a mixed, for example)
4. The name of a method or public/protected variable cannot change.
5. The parameter list of a method must remain the same, or there must be a translation scheme in place for a change.

These are restrictive conditions, and necessarily so. In a massive code-base like Discworld, you have to assume that if something is accessible to other objects, then some other objects have made use of it. People will have looked at the object and said 'Wow, the do_groovy_stuff method does exactly what I need' and then made use of that method in an entirely unrelated piece of code.

By default, all methods and variables in an LPC object are publicly accessible. That means every object in the game has access to the methods, and also to the variables (although variables are at least a little protected by the comparatively primitive object model of LPC). Protected methods and variables are available only to the object in which they are defined, as well as all subclasses (all classes that, somewhere along the way, inherit the base class). Private methods and variables are accessible only to the object in which they are defined – no external objects, and no subclasses. These are known as **visibility modifiers**.These also map onto categories for impact of change:

| Visibility Modifier | Impact of Change |
|---|---|
| public | High |
| protected | Medium |
| private | Low |

Thus, an example of a forbidden refactoring would be to turn a public method into a private method. This breaks our first rule – it reduces the visibility of the method. Any object making use of that method will break as a result of our refactoring.

We could take a private method and make it public – this increases the visibility, but is almost never a good idea. There's usually a reason why a method has been given restricted access rights. This isn't the place for a discussion of proper object oriented design though – you'll find more of that in LPC For Dummies 2.

You can't change the return type of a publicly-accessible method without violating the rules. This, for example, would be a forbidden refactoring:

```
int add_nums (int num1, int num2) {
  return num1 + num2;
}
```

Into:

```
double add_nums (int num1, int num2) {
  return to_float (num1 + num2);
}
```

Likewise, you can't change the type, order, number or meaning of parameters in a method unless you provide some way for that change to be transparent to all the objects making use of it. This would be invalid:

```
int add_nums (int num1, int num2) {
  return num1 + num2;
}
```

Into

```
int add_nums (double num1, double num2) {
  return to_int (num1 + num2);
}
```

The rules of refactoring are in place to make it a more pleasant environment for everyone to work within. If everyone is obeying them, the chances are greatly reduced of you logging in one morning to find none of your code working the way it did the night before.

# Breaking The Rules

Sometimes it's okay to break the rules. Mostly this comes along with the access you have to fix the problems that come along. Imagine you are a young turk, looking to make your name with some great improvements to /std/basic/desc.c. Someone gives you access to that code, and you decide 'Ha, set_short is too limited. I'm going to make it take half a dozen parameters, all of which will be mandatory'.

That's not allowed, because it would break... well, almost everything. In this hypothetical situation, the only access you have is to that file directly, not every file that uses it. The rule is, 'if you break it, you fix it', and if you can't reasonably fix it, you don't get to break it.

On the other hand, if you have write access to /d/waterways and you want to break something in /d/waterways/handlers then, with care, you can break the rules because you're in a position to fix everything that might be using the code. Sometimes refactoring is a task involving many objects, not just one object or one method.

Additionally, you can use common sense to tell whether or not anything is likely to be using the method in question. If you have a method 'check_things' in an obscure room in an obscure area, then go ahead and change that method if you need to – you'll only need to fix that one room after all. It's extremely unlikely that anything else will break as a consequence, even if it is a public method.

Let the impact of change categories guide you – if it's critical or high, don't do it. You'll break more things that you can realistically fix. If it's medium or low, then proceed with caution. Just be prepared to fix anything you may break.

Sometimes, although the situation is rare, someone needs to change something critical even though it is almost guaranteed to break other objects. When add_action was removed from the driver, or when type-safe checking was implemented – this was changing driver code, so it impacted every object in the game. If something has to go, then it has to go.

There's a process that you go through when this is the case, and it's something like this:

- Announce the deprecation of a piece of functionality.
- Provide a list of objects that will need to be fixed.

- Post a deadline by which the code must be changed, along with guidance as to how changes should be implemented. Be on hand to help with this.
- When the deadline is reached, give a few days' notice before you make the changeover. Give a date at which time the change will be made. Don't say 'in the next few days', be specific.
- Make the change, even though things will probably break.
- Fix the things that broke.

This is a time consuming process, and one that we don't go through very often because it's a big hassle for everyone involved. It's only permitted in extreme situations, and if you have to ask yourself whether you have the authority to make such a change, the answer is you don't.

# Refactoring

So, what kind of things do we do in refactoring? There are several, but the most common things are:

- Removing dead code
- Making inefficient code more efficient
- Making code more readable
- Making code more maintainable

Ideally, refactoring is a proactive process – you do it as an ongoing part of development. In reality, we tend to refactor only when there is a problem with the code with which we are currently working. Refactoring is normally a first step towards adding new functionality. When we refactor, we are looking to make the code look the way it would have done if it were written properly the first time.

Refactoring can be as simple as changing the name of a variable to something more meaningful; however, if this is a publicly-accessible variable, even that trivial change can break code.

Sometimes, it's a case of improving the aesthetics of an object. Said object may work perfectly, but offend the sensibilities when the code is viewed. The aesthetics of code are important – they are usually a hint as to where refactoring can profitably be applied. Complex, unwieldy, and ungainly coding structures are usually there to handle complex logic operations that could potentially be either extracted or remodelled. However, you shouldn't automatically think 'complex code is bad code', especially if the cold is old. Some code isn't ugly, it's **battle-scarred** – it's been thumped to bits by countless rounds of testing, and then patched up and fixed and put back into the field. Recognising the difference between ugly code and battle-scarred code gets easier with practise.

# Some Common Refactoring Tasks

There are several common tasks that are done to refactor objects. Some of these are structural, relating to the way in which objects are connected to other objects:

- Generalising object functionality.
- Specialising object functionality.
- Improving encapsulation.
- Lowering impact of change

Some of these are related to the code inside objects:

- Simplifying internal structures.
- Improving variable names.
- Simplifying logical comparisons
- Substituting one algorithm for another
- Consolidating conditionals
- Extracting functionality into separate methods.
- Reducing inconsistency in naming and parameter ordering

Martin Fowler (http://www.refactoring.com/catalog/index.html) has a list of common refactoring tasks. Not all apply to Discworld, but you can get a taste of what refactoring is all about.

# Conclusion

This chapter isn't a technical resource about refactoring, but the outline of a philosophy for refactoring that can reduce tension amongst your colleagues. Refactoring is an important and on-going process, one that you will undoubtedly get involved with at one point or another. Its role in this material is to outline a set of criteria by which you should refactor – a 'manifesto of courtesy' for how to make sure you don't inconvenience everyone with your changes.

# Coding Etiquitte

## Introduction

Formal codes of etiquette exist to smooth relationships between people by establishing boundaries of acceptable behaviour. So it is in coding within a multi-developer environment – there is an etiquette that works to reduce friction. As far as I am aware, no-one has actually formalised this before, so this is my own clumsy attempt to do for coding what Emily Post did for the 1920s.

It's very easy to step on toes as a developer, and having a little consideration for your colleagues is the best way to show the necessary respect for their time and effort. I don't think there's a lot of value in having an exhaustive encyclopedia of such rules, so I have concentrated only on those of real importance to the process.

## Before You Write Any Code

The first steps you take to ensure you are being polite are taken before you write any code at all.

### Check for Duplication of Effort

To begin with, look to see if there has been any duplication of effort. One easy way to cause Conflict is to say 'Hey, I'm coding this cool thing here', when someone else is already coding that cool thing elsewhere. Additionally, check to see if there's any abandoned work that has been done in the past on similar developments. Sometimes projects stop simply because people don't have enough time, rather than because they weren't working out. A rescue and adaptation of old code can be as valuable and efficient as writing the code from scratch.

## Make Sure All Involved Parties Are Consulted

An easy way to rub people up the wrong way (and to run the risk of your project being mothballed) is to not consult the people who need to be consulted. You can't just say to yourself 'I'm going to write a guild-house for wizards in my development', because the placement of guild-houses has strategic importance for the Guilds domain. Similarly, if you're writing an island of some kind, it has importance for the Waterways domain; and if you're writing a complex and potentially generalisable subsystem for your area, then the Special domain may be interested.

Making sure that the relevant parties in those domains are consulted before you start coding is the easiest way to avoid potential future conflict – it's hardly ever the case that you're told 'No, you can't do anything like that', but there may be conditions, or modifications that are necessary, or perhaps a recasting to fit in line with future domain objectives.

## Ensure A Migration Strategy

If you're remodelling code that's already in the game, or code that's likely to impact on other developers (such as a change to inherits or handlers), then make sure you plan in advance to make the change with the smallest possible 'interruption of service'. Make sure everyone knows, ahead of time, what's going to happen and what that means for their own code. Do it first, because it'll already be too late if someone points out a problem with your migration strategy after you have made the changes.

## When You Are Writing Code

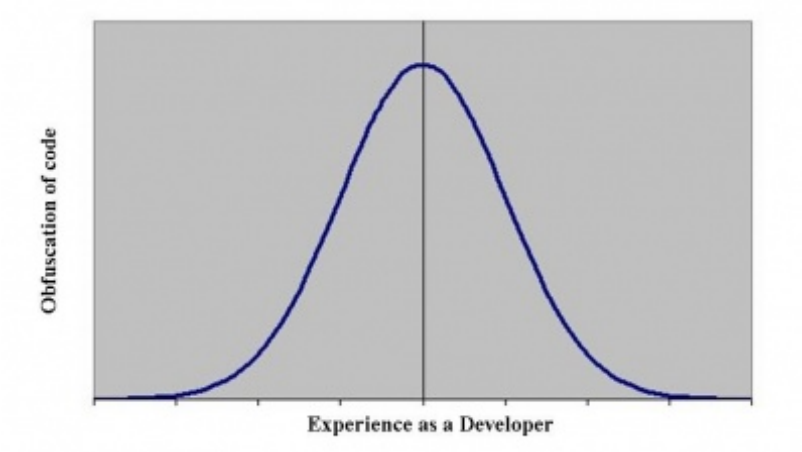
The biggest area for potential conflict is in the code you actually write – all code on Discworld is very tightly interconnected, and you can easily cause problems for other developers if you write code without care.

## Be Wary Of The Impact of Change

As per our discussions on the impact of change, you need to bear this in mind when you are changing code. If it's going to change the way the code functions, don't do it unless you've gotten people on board and given warning. The higher the impact of code, the greater your consultation with others should be. Seriously, don't be a dick – don't just break other people's code because it's convenient for you to do so.

## Write Your Code Cleanly

There exists, at least in my mind, a bell curve that describes the simplicity of the code written by developers. I call it the Obfuscation Curve:




Newbie developers write simple code because it's all they know how to do. However, as the years go by, they accumulate knowledge of new tools, techniques, and syntax. They then fall into the incredibly common trap of thinking that a good developer is defined by the number of tricks they know. These tricks then tend to make their way into every piece of development, just to show how 'clever' the coder is. Additionally, developers at the midpoint of experience tend to associate complexity with quality – if the code is clever and does what it is supposed to do in a highly efficient (albeit inscrutable) way, then it must, by definition, be good code.

This couldn't be further from the truth.

There is some merit to the idea that good code can be an intellectual exercise – doing things in new and unusual ways is personally satisfying after all. However, in a multi-developer environment you are actively retarding development by making every developer who follows you puzzle out the logic of your code. Your lapse into egotism is a burden to all. I don't care what you do with your solo development projects, but when you're working with others, don't do it.

As developers gain further experience, especially after working with other people, they start to return to the idea of clean, simple code. It's much more maintainable, much more readable, and the minor efficiency losses are more than compensated by the ease with which fellow developers can add features or address problems.

Really beautiful code, and there are some lovely examples of this, is elegant. Elegance demands simplicity of expression. The developers with the best Code Fu write elegant code, not complicated code.

As a counterpoint to this, I don't mean to say that you need to write code that even the newest creator can understand. You can make available to yourself the full vocabulary of the programming language – when I talk of readability, I mean readability to a fellow developer of reasonable literacy with the language.

For example, the following code is needlessly obfuscated:

```
if((check = ::move(dest, messin, messout)) != MOVE_OK)
  return check;
```

You can follow what it does, but it doesn't need to do it so awkwardly. There's nothing lost by being explicit in the code:

```
check = ::move (dest, messin, messout);

if (check != MOVE_OK) {
  return check;
}
```

You lose nothing, and you gain readability in exchange.

To show that I'm not simply having a go at other people here, I'll include an example from my own code:

```
tot = map (filter (property_list, (: member_array ($1->street_name,
  $(monopoly_sets)[$(m)]) != -1 :)), (: $1->houses :));
```

Sure, it's nice Code-Fu, but it was written eight years ago when I too confused complexity for cleverness. But I bet it would take you quite some time to work out exactly what this line of code is supposed to be doing. I'll give you a hint, though, it's doing something quite simple.

## Document Extensively

Good code is its own documentation. I am in no way a proponent of 'commenting metrics', whereby X lines of code must have Y lines of comments. As long as you pick meaningful variable names and don't over-complicate it, you'll find that your code is readable enough. Everyone has a different opinion about readability, though. I've occasionally found that someone's gone to the trouble of commenting code that I was too lazy to comment myself. They have my thanks for this!

However, whenever you're doing something a little bit exotic, you should outline what your intentions were within the code. Don't describe what the code does, describe why it does it. The following would be a bad comment:

```
// Does a map on the filtered property_list array.  The filter filters
// the array for all those streets that are part of the value $m in
/ the monopoly_sets mapping.
// It returns the houses member of the class in array format.

tot = map (filter (property_list, (: member_array ($1->street_name,
  $(monopoly_sets)[$(m)]) != -1 :)), (: $1->houses :));
```

That explains what the code does, not what the intention of the code is. You can't tell from that comment whether the end result is what the code would suggest. A better comment would be this:

```
// This piece of horrible code gives an array of all the houses a player
// has in the properties that belongs to a set.
```

That explains what the code was supposed to do, so that a developer can look at the end result and decide whether or not that's what happens. Comments should explain intention, not simply dissect the code.

## Attribute Contributions

Coding is a collaborative effort, and much successful coding is simply well-targeted plagiarism. That's absolutely fine – if someone has already solved the problem you are having with a bit of their own code, then use that code as a template for your own. However, when you do this, it's very nice if you can provide an attribution, such as:

```
// This code borrowed from Drakkos' Killer Weasels
```

Attributing the work of those who have gone before you is a respectful activity – it doesn't take away from you as a coder, it enhances your reputation in the eyes of other people. Moreover, it makes it easier for people to maintain the MUD - if I know that you based a piece of code on a function in /d/waterways/stupid_thing, then I know that if I fix your code I may also have to fix it there.

# When You Have Written Code

Once you have written code, there are a few more things that fall under the general criteria of politeness.

## Abdicate Ownership

Once the code is written, you should mentally hand it over to the domain. It's no longer your code – it's production code, belonging to everyone. It's a mental activity, so you don't need to actually do anything for this… but if you get wound up or upset by someone else changing something in your code, then you're doing it wrong. Complaining that someone else 'changed your code' is rather rude in an environment where everyone else is being a team player with the code they provide.

## Be Willing To Maintain

Even though you abdicate ownership of the code, you are still the person best qualified to maintain it. As far as possible, you should keep an eye on reported problems with your developments and be prepared to fix them, especially if they flummox other creators. Although you aren't the owner of the code, you are the expert on it.

## Make Sure All Parties Have Adequate Information

There's little worse from a 'creator satisfaction' perspective than those with a need to know not being told what they, well, need to know. Need to know parties will vary from project to project, but if it's an area going into the game then the liaison domain need to know as much as you can tell them about features that may potentially go wrong. If it's a cross-domain collaboration, then all the collaborating domain administration teams need to know the details.

A post on the liaison board is the usual mechanism for announcing a new feature for the game, but a mail to the relevant parties ahead of time also shows the appropriate amount of respect for your collaboration partners. Something like, 'Hey, we're going to make this live next week, so let us know if you have any last-minute comments' is a great way to make sure everyone gets a chance to have a last look over the development before everyone is committed.

# Conclusion

There aren't all that many rules you need to worry about – and most of them are fairly self-explanatory. They all stem from a single basic concept though – don't make life harder for people than you actually have to. Working with other people means being able to compromise and communicate clearly. It's much easier when everyone has a firm idea as to what is acceptable and what is not when making development decisions.

Some of us are better (or worse) than others at being polite and respectful developers. It is not unknown for a developer to wake up one morning to find that everything they have written has been broken because someone else 'fixed' a low level inherit. Such occasions are rare, but there are precedents. Simply bear this in mind – how would you feel if you logged on one morning and everything you had written had to be changed, without any warning or consultation? You'd be legitimately pissed off. If it would bother you, don't do it to other people.

# Source Control

## Introduction

One of the most useful systems we have in place on the MUD is our source control system. The MUD uses a system called RCS – the Revision Control System – to ensure that multi-developer collaboration proceeds as smoothly as it can. It is a system for restricting the access to make code changes so that only one developer modifies a file at a time. This neatly gets around the problem of one developer overwriting the changes of another developer – something that, with the best of all intentions, is quite common when no formal system exists to prevent it.

## Source Control In The Abstract

Imagine you are a Discworld developer. That should be quite easy, because presumably you are! You are working with a file, /d/forn/awesome.c. You download a copy of this to your local machine, and make some changes. While you are doing this, unaware of the fact you are working with the file, I come along and download it to my local machine and start making changes. You finish up, and upload the file to the MUD. I finish up, and then upload my file to the MUD. Despite the fact we both made changes, my version overwrites yours.

There are ways to minimise the chances of this – for example, like so:

```
(forn) Drakkos: Hey, I'm about to do something with /d/forn/awesome.c  (forn)
AnotherCreator: Can you hold off a bit, I'm currently working with it.
(forn) Drakkos: Sure!
```

This is imperfect though – it doesn't catch anyone who is offline and working through FTP, and it doesn't catch anyone who isn't reading the channels. Communication can go some way towards making sure problems like this don't occur, but it doesn't solve the root issue.

Source control steps in and provides the solution to this problem. All files on the MUD can be placed into source control (this has to be done manually, because sometimes it's inconvenient), and once they have been registered with the system it becomes impossible for anyone to make changes unless they first 'check out' the file.

Only one person at a time can check out a file, which means that if you are working with awesome.c, I will get an error message if I try to make any changes. When you're done, you 'check in' the file and provide a little description of what you did. At that point, your changes are said to have been 'committed'.

Part of the beauty of a system like this is it keeps track of the differences between the new version and all previous versions, and with a single command a developer can revert the file back to a previous version. If your changes to awesome.c introduced some horrible, game destroying error... well, we just 'revert' the file back to its previous version and no harm, and no foul.

It really is a wonderful system. In our example of you and I working with the same file, no-one has done anything wrong. It's not a sign of bad communication or a dysfunctional environment – it's just One Of Those Things. While it's almost never a good idea to apply a technological solution to a social problem, what we're describing here is a technological solution to a technological problem.

Like any system though, it's only as good as the people using it. We'll talk about some of the social problems with RCS later in this chapter.

# The Discworld RCS System

If you are the sole developer working on a project that is not in the game, you may find it easier to keep everything off of RCS while you work. It can be inconvenient to continually have to check files in and out of the system, and somewhat against the spirit of the thing to check them out and never check them back in until the development is completed. When the code goes live, or into playtesting, you should – without exception – add the files to the RCS system. Luckily, it's very simple to do – you use the **rcscreate** command:

```
rcscreate /d/learning/learnville/chapter_02/rooms/*.c
```

You'll be prompted to enter some text – as a matter of convention, this text is usually 'Initial Revision'. Once you've entered the text, that's it – your files are now on the system and you won't be able to make any changes until you first check them out. That's done using the **rcsout** command:

```
rcsout /d/learning/learnville/chapter_02/rooms/street_01.c
d/learning/learnville/chapter_02/rooms/RCS/street_01.c,v  -->
d/learning/learnville/chapter_02/rooms/street_01.c
revision 1.1 (locked)
```

Now you have access to change the file, and nobody else does. When you've made your changes, you use the **rcsin** command:

```
rcsin /d/learning/learnville/chapter_02/rooms/street_01.c
```

When you do this, you'll be prompted to enter some text. Please provide something useful and meaningful for this, because it's what people will see when they look at the log of changes that have been made. You shouldn't detail the code you changed – that's available already (more on this later). Instead, your comment should focus on intention and what the change was supposed to achieve. If you haven't actually made any changes to the file, it will automatically revert to the last version:

```
d/learning/learnville/chapter_02/rooms/RCS/street_01.c,v  <--
d/learning/learnville/chapter_02/rooms/street_01.c
file is unchanged; reverting to previous revision 1.1
```

If, after having checked out a file, you decide that you don't actually want to make any changes, or if you've made changes and they're not what you want, you can release your lock on the file using **rcsrelease**. This will release your claim to the file without committing any of your changes. It'll revert automatically to the most current version of the file:

```
> rcsrelease /d/learning/learnville/chapter_02/rooms/street_01.c
d/learning/learnville/chapter_02/rooms/RCS/street_01.c,v  -->
d/learning/learnville/chapter_02/rooms/street_01.c
revision 1.1 (unlocked)
```

If you want to see what files you have locked out, the **mylocks** command is your friend:

```
> mylocks
You have the following files locked:

/d/learning/master.c
/d/waterways/handlers/docks_handler.c
/d/waterways/handlers/ship_ownership.c
/d/waterways/inherits/mooring_area.c
/d/waterways/inherits/pier_inherit.c
/d/waterways/inherits/ship/helm.c
/d/waterways/inherits/ship/hold.c
/d/waterways/inherits/ship/outside_ship_room.c
/d/waterways/inherits/ship_object.c
/d/waterways/ships/inherits/ship_rooms/base_inherit.c
/d/waterways/ships/inherits/ship_rooms/inside_ship_room.c
/d/waterways/ships/inherits/ship_rooms/player_ship_room.c
/d/waterways/ships/types/sloop/rooms/bridge.c
/d/waterways/ships/types/sloop/rooms/cabin.c
/d/waterways/ships/types/sloop/rooms/nest.c
/d/waterways/ships/types/sloop/rooms/plank.c
/d/waterways/ships/types/sloop/rooms/weapons.c
/include/learning.h
/include/waterways.h
/obj/handlers/oracle.c
/w/drakkos/public_html/secure/project.c
/www/external/includes/discworld.js
/www/external/includes/discworldv.js
/www/header.c
/www/secure/creator/oracle/oracle.c
```

Whoops, I should probably check some of those back in! You can also check to see which files another creator has locked out with mylocks:

```
> mylocks gruper
Gruper has the following files locked:

/d/waterways/islands/pirates_cove/cove/beach.c
/d/waterways/ships/inherits/ship_functions/oars.c
```

Damn him, always making me look bad by virtue of his professionalism!

If you want to see who has a lock on a particular file, that's what the **rcslocks** command does:

```
> rcslocks /d/learning/master.c        File /d/learning/master.c locked by
drakkos.
```

Finally, if you want to see the changes that have been made to a file, the **rcslog** command gives you all that information:

```
> rcslog /d/learning/master.c
RCS file: d/learning/RCS/master.c,v
Working file: d/learning/master.c
head: 1.5
branch:
locks: strict
    drakkos: 1.5
access list:
symbolic names:
keyword substitution: kv
total revisions: 5;          selected revisions: 5
description:
----------------------------
revision 1.5         locked by: drakkos;
date: 2008/10/05 20:06:51;  author: drakkos;  state: Exp;  lines: +88 -2
Changed way the domain info is displayed, and the order in which domain
creators are shown.
----------------------------
revision 1.4
date: 2008/10/02 19:14:33;  author: taffyd;  state: Exp;  lines: +1 -1
Forcibly unlocked by drakkos
----------------------------
...
```

We'll spend a little bit of time talking about these entries, because there's a lot of information in there. Let's take the last of these as an example:

```
revision 1.5         locked by: drakkos;
date: 2008/10/05 20:06:51;  author: drakkos;  state: Exp;  lines: +88 -2
Changed way the domain info is displayed, and the order in which domain
creators are shown.
----------------------------
```

The first piece of information we are given is the revision number of the file. Each time a change is committed, the decimal part of the version increase by one. This follows a general convention of software updates... it's possible for a revision to increase the whole number part, but nobody ever does it... indeed, the only piece of code for which I know it has been done is the taskmaster – it was updated to version two when the degree of success (critical success and critical failure) code was introduced.

The date is when this change was committed, not when the change was made. As such, there can be wide disagreement with the 'official record'... sometimes files remain locked out for weeks or months, and so the date of a revision bears no relationship to when the changes were made.

The author is the person who committed the change. The state tag is not something we use much on Discworld, or indeed use at all. It relates to the state of the release – EXP stands for 'experimental'. The state can be anything, although the convention for other states is STAB (for stable) and REL for release. You'll be unlikely to encounter anything other than EXP though as you work your way through the Discworld codebase.

Lines indicates the net number of lines that were added (defined as any line that was changed) and lines that were removed (defined as any life that is no longer in the code).

The most important bit of all of this though is the text that accompanies the entry – this is the text that the creator entered as part of the rcsin. If good practise is being followed, this will be a meaningful description of the change that was made.

The final command you're likely to make use of is **rcsdiff** – this gives you the exact difference between two version of a file, showing exactly what was added and what was removed. If I wanted to know what changes were made between version 1.4 and version 1.5 of a file, it's this command I use:

```
> rcsdiff -r1.4 -r1.3 /d/learning/common.c
```

This will give the following output:

```
< int do_sit( string command, object *indir, string dir_match,
< string indir_match, mixed *args, string pattern );  32c30
< "sit", ({ (: do_sit :), "[in] <direct:object>" })  ---
> "sit", ({ this_object(), "do_sit", "[in] <direct:object>" })  37c35
< "sit", ({ (: do_sit :),  "[in] <direct:object>" })  ---
> "sit", ({ this_object(), "do_sit",  "[in] <direct:object>" })
```

It can be slightly difficult to read this output. Lines marked with a < are lines that have been added, and lines marked with a > are lines that have been removed. In essence, the following two lines:

```
> "sit", ({ this_object(), "do_sit", "[in] <direct:object>" })
> "sit", ({ this_object(), "do_sit",  "[in] <direct:object>" })
```

Were replaced with the following:

```
< "sit", ({ (: do_sit :), "[in] <direct:object>" })
< "sit", ({ (: do_sit :),  "[in] <direct:object>" })
```

There are more commands available as part of the RCS system, but these are the commands you'll be working with most often. The help-file for RCS will outline some more interesting and useful options.

# Problems

There are, as usual, social problems that come along with any system. While our use of RCS is on the whole very good, there are lapses – usually centred around specific individuals. Alas, I count myself amongst these – senility has grabbed hold of me in my old age, and I thus often forget I have files locked out. And then, when I check them back in, I forget what it is I have done.

Usually this is a result of carelessness rather than malice – files remain locked out for as long as it takes for someone to realise (for example, another person who needs access to the file). However, there is a more insidious problem of people pre-emptively checking out code so that other people can't change 'their code'. This, as we have discussed, is not a mindset we like to encourage.

The rcslog of a file is a historical record – it shows what changes were made, along with a short summary. However, this record can be constantly interrupted with 'noise' such as files being forced (this is when someone forcibly checks in a file for you – this is something available only to senior creators and above), or less than helpful rcs entry messages. For example, from /d/forn/master.c:

```
----------------------------
revision 1.5
date: 2003/06/07 18:33:53;  author: drakkos;  state: Exp;  lines: +599 -591
Forcibly released due to inactivity
----------------------------
revision 1.4
date: 2001/11/16 21:35:46;  author: drakkos;  state: Exp;  lines: +590 -596
Buggered if I know.
----------------------------
revision 1.3
date: 2001/05/26 16:57:53;  author: terano;  state: Exp;  lines: +596 -600
Fixed a thing.
----------------------------
revision 1.2
date: 2001/05/26 16:49:56;  author: drakkos;  state: Exp;  lines: +600 -476
Fixed up a few things.  Changed a few more.  Added some very crude metrics.
```

None of these are useful messages, especially considering how many lines of code have been marked as changed. This is a problem with the people using the system, not the system itself.

Source control is not a substitute for a good developer environment – it stops accidental collisions of code, but it won't help with genuine social problems between developers.

Note too that once a file has been put on RCS, it can't be removed except by a Trustee. That means no shuffling files around, or deleting them permanently. You need to be sure that you can commit to what's there unless you want to risk the wrath of waking a Trustee from their blissful slumbers.

# Conclusion

Source control is one of the most important systems we have for supporting our development work. It means we don't need to keep backups, or tediously roll back changes by hand. It means we can track changes made to objects, and identify people who were responsible for making changes. All of this in addition to its core function of making sure that we don't end up killing each other over code collisions. Learn to love it!

# Documentation

## Introduction

Writing documentation is one of the least enjoyable tasks that comes along with developing code. As such, it tends to be something that's left until the last minute, or done in infrequent, unreliable pushes of effort. It's a shame it is so tedious to produce because good documentation is worth its weight in gold for those who come after you.

When it comes to documentation, I don't necessarily mean commenting. I am not a proponent of the view that comments should form X% of your source code (although many people are) because I believe that good code is its own documentation. In addition to comments that describe what code is supposed to do, Discworld has a commenting format that allows for automatic extraction and indexing of object functions, their return values, and their parameter lists.

In this chapter we will also talk about the format used by the MUD's help-files, and how you can aid in our documentation effort by migrating user help into bespoke object help-files.

## Commenting

The usual argument is, 'It is good practise to comment your code'. In my experience, when this argument is followed to its logical conclusion it actually detracts from readability. Imagine the following code (and this is not an exaggeration, I have seen files like this by the dozen):

```
// This declares an integer variable called num.
int num;
// This declares a string variable called name.
string name;
// Create a for loop with a counter  variable called i.
// It will loop while i is less the num.
for (int i = 0; i < num; i++) {
  // Send the text that is in the variable name to
  // this_player(), using the tell_object() method.
  tell_object (this_player(), name + "\n");
}
```

The problem with these comments is that they are aimed at the wrong audience. You shouldn't write comments so that your grandmother, or your mother, or your best friend can understand what is going on. You should write comments so that a fellow literate programmer can know what is going on. These comments do not say anything that the code itself doesn't say. The code itself gets swamped in the comments, making it a little less readable. Nothing is gained from this, even though everything has been commented.

Good code is its own documentation. The following is bad code:

```
m=i+((r*i)-d);
```

While you can work out what this is doing, there is no hint as to why one variable is being modified by another in a particular way. On the other hand, simply choosing meaningful variable name turns that into self documenting code:

```
my_money = income + ((reserves * interest_rate) - debits);
```

In this code it is obvious what is happening – there's no need to comment this. You'd certainly need to comment the former.

Sometimes though, even with meaningful variable names, you're going to end up doing something a little bit esoteric. Whenever you feel that it is unlikely that a fellow literate programmer will be able to tell, at a glance, what you were trying to do – that's the time to add a comment. This gives a happy balance between enhancing readability and not restricting you from coding productively.

However, even assuming complete comprehension of what each individual line of code is doing, it is hard to tell, 'at a glance' what the big picture is. That's where the system documentation comes in – we provide documentation on each of our functions so people can tell what they put in, what comes out, and what the value that comes out will mean.

# Commenting Good Practice

One of the biggest problems with large bodies of commenting is the difficulty in keeping it up to date. What tends to happen is that the code gets changed and the accompanying comment doesn't get updated. Before too long, the comments bear little relation to the functionality and become actively unhelpful. That's why it's important to document the intention, not the actual steps taken.

You should also try to document 'why' along with 'what' – why did you decide to do something one way over another? Any time you had to spend a bit of time puzzling over alternatives, you can save those who follow you the effort by saying 'I decided to do it this way because it's more efficient/maintainable/readable than the other way'.

If you are making any assumptions at all in your code, then for the love of god document those assumptions. If the entirety of your function assumes that a particular parameter is within a certain range, make sure that information is documented somewhere other than in your head. Of course, if you are going to rely on such things you should have a guard condition in the code ensuring that the function won't be executed if the parameters are invalid. Still, document the assumptions you make.

Avoid being humorous in comments if it comes at the expense of clarity. Don't use code words, or in-jokes, or obscure references, no matter how widely understood you believe the reference to be:

```
// This fubars the string.
string do_fubar(string str) {
}
```

Finally, don't comment out obsolete functionality – delete it entirely. The revision control system means that the functionality is available should it be required (oh – make sure the file is on RCS first!), and removing it entirely from the code greatly increases clarity.

# Autodoc

Discworld has a commenting convention based on the Javadoc standard. It's called Autodoc, and it integrates documentation for functions into the standard help system. Imagine you wanted to know what query_area in /obj/handlers/armoury.c does – you can find out by typing 'help query_area', and you'll get a little help-file discussing it:

```
query_area         Discworld creator help                 query_area

Name
          query_area -  Returns the list of domain armoury items.

Syntax
          mapping query_area(string domain)

Parameters
          domain - the domain/area to get the items from.

Returns
          the area sub-mapping.

Defined in
          /obj/handlers/armoury.c

Description
          Returns the list of domain armoury items.
```

That help-file is generated automatically from the comments that have been put before the function in the file. As long as the comments adhere to a particular syntax, they can be parsed and made available to everyone. You are unlikely to ever need to do this for rooms, NPC and specific items – but if you're doing anything more substantial, it's very useful if you can provide autodoc commenting.

An autodoc comment starts with a special code: /**

Every line that follows begins with a star in line with the first of the asterisks, and it ends with the normal closing of a block comment: */.

```
/**
 *
 *
 */
```

The first line of text is what is used for the summary that follows the name of the function. The rest of the text is used for the description of the function. You can mark this up with normal HTML, so you can add in paragraphs and line-breaks as necessary.

The syntax of the function is extracted automatically by the autodoc handler, as is where it is defined. The rest of the information we need to provide, and we do this using autodoc tags. These begin with a @ symbol, and are interpreted by the autodoc handler according to the text that belongs to the tag.

For example, let's take a simple function from /d/learning/master.c and put it through the autodoc format. The function is this:

```
int set_project(string name, string pro) {
  if (geteuid(this_player(1)) != query_lord())
    return 0;
  return ::set_project(name, pro);
}
```

Its task is simple – it checks to see if the person making use of the function is the lord of the domain. If they aren't, it returns 0 and does nothing. If they are, it passes responsibility onto the object that this object inherits.

First, let's describe that in an autodoc:

```
/**
 * This function sets the project of a domain member.  It first checks to
 * see if the person making the call to the function is the lord of the
 * domain.  If they are not, it will return 0 indicating failure. The
 * method will make a call to the set_project of /std/dom/base_master.c
 * if this initial check is passed.
 *
 */
```

Next, we add in tags to provide an explanation of what the parameters and return value mean. We have to be careful with formatting here – there should be one space between the asterisk and the tag, or it won't be picked up by the handler. @param is used to give a meaningful description to a parameter, and @return is used to describe how the return value should be interpreted.

```
*
* @param name The name of the person for which we want to change the project.
* @param pro The project the person is to have in the domain.
* @return 1 if the project is successfully changed, 0 if it is not.
*
```

There are some other valuable tags we can provide:

| Tag | Description |
|---|---|
| @see | Adds a reference for other objects of interest. You can use this to direct attention towards related objects. |
| @example | You can use this to provide a code example of the object in use. |
| @ignore | Makes it so the autodoc handler ignores the function for the purpose of automatic generation. This is useful if it's a small, private function that no-one need worry their pretty little heads about |

We should definitely add in one of each of the first two;

```
* @see /std/dom/base_master.c
* @example
* ret = set_project ("drakkos", "Being Awesome");
```

This would give us our full autodoc comment:

```
/**
 * This function sets the project of a domain member.  It first checks to
 * see if the person making the call to the function is the lord of the
 * domain.  If they are not, it will return 0 indicating failure. The
 * method will make a call to the set_project of /std/dom/base_master.c
 * if this initial check is passed.
 *
 * @param name The name of the person for which we want to change the
 *             project.
 * @param pro The project the person is to have in the domain.
 * @return 1 if the project is successfully changed, 0 if it is not.
 * @see /std/dom/base_master.c
 * @example
 *  ret = set_project ("drakkos", "Being Awesome");
 */
```

This will generate the following help-file for help set_project:

```
set_project         Discworld creator help                    set_project

Name
          set_project -  This function sets the project of a domain member.

Syntax
          int set_project(string name, string pro)

Parameters
          name - The name of the person for which we want to change
                    the project.
          pro - The project the person is to have in the domain.

Returns

          1 if the project is successfully changed, 0 if it is not.

Defined in
          /d/learning/master.c

Description
          This function sets the project of a domain member.  It first checks
          to see if the person making the call to the function is the lord  of
          the domain.  If they are not, it will return 0 indicating
          failure.  The  method will make a call to the set_project of
          /std/dom/base_master.c  if this initial check is passed.

Example 1
          ret = set_project ("drakkos", "Being Awesome");

See also
          /std/dom/base_master.c
```

Cor, don't that look purty?

As a matter for convention, you should also add such a comment at the top of the file detailing, at the very least, who the author is and when it was started. If you're working with a legacy file, then that might actually be known:

```
/**
 * Learning Domain Master Object
 * @author Who Knows
 * @started A Long Time Ago
 */
```

Still, it's better than nothing. Marginally, anyway.

Autodoc works for function level documentation. For commenting within a function, standard commenting is all you need to use.

# The Autodoc Process

Once you've written a file that you want to add to the autodoc system, you have to add it using the **autodocadd** command:

```
autodocadd /d/learning/master.c
```

A second or so later, your file is in the system. However, it won't appear when you try to get the help file. Help-files are updated on a delay to reduce load on the system – it will be generated at some point in the not too distant future. However, you can kickstart the process by using the **autodoc** command – this will force the generation of the documentation:

```
autodoc /d/learning/master.c
```

This will create an overall view of the file in /doc/autodoc/. The filename will be the same as the file path, except all the backslashes will be replaced with dots. Thus, it's the file /doc/autodoc/d.learning.master.c.

Each of the functions that have been documented will be stored under /doc/creator/autodoc/- everything in here is organised in a familiar file hierarchy. The one for set_project will thus be found at /doc/creator/autodoc/d/learning/master/set_project.

We can force these files to be added into the help system using the **rehash** command on the directory:

```
rehash /doc/creator/autodoc/d/learning/master/
```

Your help-file will now be available in all its glory, to everyone who needs it.

# Other Help-Files

Player help-files are written in a different format called nroff. This is a little more esoteric, but fairly simple once you get the hang of it. It's good practise to provide help for players for anything that may involve a game of 'guess the syntax' – help-files can be attached to rooms, NPCs, or items using the add_help_file function in the setup of the appropriate object.  The following help-file is taken from a story cabinet in a Genua pub:

```
.DT
Story cabinet
Discworld player help
Story cabinet
.SH Name
.SI 5
Story cabinet - Morality tales in the making
.EI
.SH
Syntax
.SI 5
push <lever> on <object>
pull <lever> on <object>
.EI
.SH Description
.SP 5 5
This is an old fashioned story cabinet... you can pull the lever on it and
it will tell you a story!
.EP
.SH Example
> push lever on cabinet
.SH
See also
.SP 5 5
None
.EP
```

The first special code here is .DT, and it means 'Do Title'. It will put the next three lines into three columns in the normal standard of help files through the MUD.

.SH indicates a section heading – this will appear in bold when the file is viewed. Only the text that follows the code will be used for this.

.SI indicates the start of an indent – the number indicates the number of spaces to indent. The indent will be in effect until it is cancelled with .EI (end indent). .SP works similarly, except it indicates the start of a paragraph. The first number is how far to indent from the left, and the second is how far to indent from the right. Once again, this is in force until a corresponding .EP is encountered.

At the end, you get this file out of it:

```
Story cabinet          Discworld player help         Story cabinet

Name
          Story cabinet - Morality tales in the making

Syntax
          push <lever> on <object>
          pull <lever> on <object>

Description

          This is an old fashioned story cabinet... you can pull the lever on
          it and  it will tell you a story!

Example

> push lever on cabinet

See also

None
```

You can use this basic template for any help-file you need to create – it's a good habit to get into. It doesn't take long to add a help-file, and it adds dramatically to the sense of satisfaction players get – there's no frustration in trying to guess syntax, only playing with the cool new thing they have encountered.

# Why Document?

Sad as it is, you probably won't be here when the End of the Disc comes. That's true of almost everyone – in the space of a few short years, absolutely everything can change. People who seemed like part of the scenery become merely part of your memory. What will go on though is the contribution you made to the Disc. The only constant in life is change – the MUD is going to change around your code, and if your contribution is to remain in the game it's going to have to be written in such a way that it is possible for other to maintain it after your departure.

There is a great saying that helps get the idea in your head – 'write your code as if the person who maintains it after you is a homicidal maniac who knows where you live'. Knowing that guy is going to have to deal with your code, wouldn't it be nice if you could placate him with some calming, soothing, useful comments?

Moreover, good documentation can serve as an aide-memoire for yourself – when you write a lot of code, you're guaranteed to forget the older stuff. When you come back and look an incomprehensible mess a year or so later, you'll be in only a slightly better state than any creator coming to the code for the first time unless your code is well documented.

# Conclusion

Good code is its own documentation, I can't stress that enough. Commenting done improperly is worse than no commenting at all – comments can be unhelpful, misleading, or downright wrong. In the process they can drown out the source code amidst a sea of green. However, when done properly, they are immensely valuable to everyone who works with your code.

The Discworld Autodoc system is a powerful way of providing help for coding functions in a consistent way – for small objects such as rooms and simple NPCs, it's safe to ignore it. For anything that is going to be used more widely, you need to be considerate of your fellow creators and thoughtful of the future maintenance duties that go along with making a lasting, maintainable contribution to the game.

# Domain Integration

## Introduction

Effective development on Discworld is a complex problem to solve. It involves many different developers, with many different cultural backgrounds, with varying degrees of expertise in software development, spread across many time-zones. It's remarkable we ever get anything done, when you think about it. As the MUD has grown more complex, it has introduced a whole new set of issues that need to be resolved.

In this chapter we're going to talk about a process called Continuous Integration, but we're not going to use it in the same way most software developers mean. Most of what Continuous Integration involves are things we don't actually need, or tools that make no sense in the context of Discworld development. You can think of this then as a modified process for continuous integration. We almost always do this anyway, but it's worth discussing why this strategy is worth adopting when dealing with code files coming from multiple sources.

## Multiple Developers – the Traditional Approach

We've already spoken a bit about the cultural and technical barriers that come with working with multiple developers. Once those have been resolved, the problems don't go away – it just reveals the existence of new problems. The comments I am going to make here don't necessarily apply to single developer projects, but as soon as multiple developers start working on the same code files, there comes a problem in terms of integrating this code together.

The traditional approach in software development works something like this:

- Everyone writes their code in isolation, over a period of time.
- At some point, the project leader says 'Right, let's put everything we've written together to see it all work'
- Everyone pools their code, and links it all up.
- Hilarity Ensues
- Everyone spends the next week or so changing their code so that it all actually meshes together properly.

The length of time between code integration events directly influences how many errors will be experienced. Technically, these are known as **integration errors**, and they come from various sources. The key source though is assumption – everyone assumes everyone else is doing things a different way.

Often, the problems aren't as obvious as files not loading – the errors can be much more subtle. You probably remember the 1999 Mars Lander probe that went horribly wrong, costing NASA around $125 million. What you probably don't know is that the reason was because of an integration error. One team at NASA was writing their code using imperial measurements. Another was writing using metric measurements. All of the internal error checking that was done at NASA failed to pick it up because each part of the program was actually working correctly. The problem came when the two pieces of code were supposed to work together.

Partially this is a political problem – if everyone decides on a standard to begin with and everyone sticks to it, the problem can be greatly mitigated. However, in large part it's a simple consequence of multi-developer work. People will make assumptions.

One of the reasons why this is a problem is in the observed behaviour – a failing probe, for example. Another reason is the simple stress and hassle of getting a project to work properly – it can take weeks to resolve integration errors in a complex project (admittedly, the project for which this is true are usually a good deal more complicated than the typical Discworld project) at a time when tensions are already high (integration is not a relaxing process). It can actually be bad for your health! It's certainly not fun, and that's what we're all here for.

Discworld operates a 'reuse' mentality rather than a 'roll your own' mentality. That's what all of our many inherits and handlers are for – to make it so people don't need to reinvent the wheel. However, if someone in another project is making use of your code, and it suddenly breaks because of an integration error – well, the last place people tend to look for the problem is outwith their own code. This is especially true if integration is an infrequent event – the less frequently people integrate their code, the less likely people are going to assume that an integration error caused their new, baffling problems.

The problem breaks down to the length of time between attempts to integrate – the longer people go without bringing all their code together, the longer bugs and errors have to creep in.

# Examples of this on Discworld

This may sound like an abstract problem with little relevance to Discworld, but one particular domain development strategy shows it in clear focus. It used to be the case for some domains that project code was developed in your /w/ directory, and the /d/ directory was only for finished code. Imagine that extended to the development of, for example, a whole city... in order for anyone to actually walk around the city, every room has to connect to the right directory in the right /w/ drive.

The code can't make use of the armoury because the armoury doesn't pick up items in a /w/ drive. Everyone has to have workarounds and shoddy code just to make sure the areas work (like cloning objects from a /w/ directory rather than using the armoury). And then it's 'Integration Day', all the code gets moved into /d/, and absolutely nothing works properly. Everyone then has to spend the next week or so getting to the point where everyone thought they already were. People have inherits they have written that would be of use to others, but only their code is using it because nobody knew it was there. One person has their move zone called 'Blah Zone' and the other has 'blah_zone'. Another person thought the connection to Awesome Street from Fantastic Avenue was a north/south exit, whereas everyone else was working under the assumption it was east/west. Multiply these problems (and others) by the size of the project and the number of creators, and you have yourself one massive headache.

Solving the problems before they arise is always the best bet, but who can solve the problems across a dozen /w/ directories? Very few people have blanket write access to /w/, and while individual creators can grant permission to their /w/ directories, it's a lot to co-ordinate. If you're having a problem with your code and you need someone to help, they can only advise from the sidelines – sometimes it helps if someone can just pop a few lines of code in place to show a tricky concept in situ.

The problems of distributed and decentralized development get smaller with fewer creators, but they don't go away. Any length of time between integration events is going to cause integration issues.

# Continuous Integration

Continuous Integration is how we solve this on Discworld, and indeed our continuous integration is usually a good sight more continuous than 'real' programming. The philosophy is simple – if the delay between integration events causes problems then the solution is surely to simply remove the delay. This is why large projects tend to work using 'live ammo'. When you get a project, you're told which directory in /d/ your project resides, and often there's already a skeleton in place so that people can actually walk around.

The benefits of this are considerable. It gives context so people know where their code will fit into the larger arc of the domain. It ensures everyone is working with the right tools – everyone uses the same inherits. It means that the handlers we use for in-game code can be used for development code. It also means that your project leader or domain administration don't need to go hunting through your things to check something for an update, they just wander to where the code is supposed to be.

For large projects (the development of Genua, the development of Bes Pelargic, the redesign of AM and DJB), it is simply infeasible to do development any other way. The projects are too large, and there are too many creators working on them. Imagine the hassle if each creator had their street in their /w/ directory!

There are a few problems that come with incorporating development code into a /d/ directory though. For example:

- Bugs in development and PT areas tend to skew the error tracker figures.
- Unless there's something in place to stop it, the area is accessible to mis-flights and mis-portals.
- NPCs in the area can be scryed or long-sighted.

These are all substantial issues, and a scheme is in place to resolve them. Directories that are under development should contain, as part of their path, the string _dev. For example, if you are working in /d/waterways/awesome_project and you want it to be marked as a dev area, you would change it to /d/waterways/awesome_project_dev. When it comes time to put the area into playtesting, it becomes awesome_project_pt. When it goes into the game, it becomes simply awesome_project. The code that is likely to have to make a distinction between play and development areas all have filename checks built into them. If you set a filename to have _pt in it, this also helps regulate certain PT capabilities, such as when and where PT protection may be switched on, and for how long.

There is a second advantage that comes from this naming system – it tests your integration. Changing the name of a directory should be as simple as changing a define in a path.h file, and everything should work flawlessly from that. If it doesn't, then you found out earl. That's a good thing – it's important you find this out before the code is due to go live! There are few things likely to upset your domain leader more than them saying 'Area awesome is now live!' and then finding out everyone needs to fix up the directories so they don't break.

# A Framework for Area Integration

This is a problem that normally comes from areas since they have the largest number of discreet files and developers to go with them. As such, my comments in this regard will centre around a framework for area integration. When working with any project with a significant number of parts, it's important to have a number of 'utility files' in place so that area-wide behaviours can easily be implemented and changed. As a general rule – if you're doing anything even remotely clever, try to break the functionality out into an inherit. Even if nobody ends up using it but you, it'll make it easier to fix bugs and add new features. We'll see that when we start developing Betterville in LPC For Dummies 2.

A new area should come with an architecture that permits easy integration – everything should be using the same inherits for rooms and NPCs – even if you don't see a need for one, create an inherit for these and have everyone use it. You'll be grateful for this when someone says 'Let's hook up this crime handler I wrote' and you don't need to manually alter two or three hundred files to do it.

I tend to break out new developments into five directories:

- rooms
- chars
- items
- handlers
- inherits

Everyone should be storing things in the same directories – a common repository, rather than for the village of Awesome to have its own chars subdirectory. If necessary, sub-directories under the base chars directory can be added – it's just important everyone knows where to go and everyone is adhering to the same standard. Where the directory is in your domain doesn't really matter very much

The items directory however must be in the root of the domain – if your domain is waterways, the items should be in /d/waterways/items/. This isn't a knee-jerk mandate – the armoury looks specifically in that directory when building its lists, and if you have your items elsewhere they won't be made available. It's a functional necessity that it be a root domain directory. Again, sub-directories within here can help manage the mass of items that are likely to exist.

When they go live, things like clothes, weapons and armours are moved into the relevant /obj/ subdirectory. It's my recommendation that this is not done until the area enters the game. Comparatively few people have access to /obj/, and it makes the task of debugging more difficult if access is not easily forthcoming.

In your inherits directory, make a placeholder inherit for every type of room that will exist – make one for shops, pubs, inside rooms, outside rooms, the whole works. It allows you to control the entirety of your development with only a few lines of code. Do you want to switch off XP gains while the area is in PT? You just need to set that up in the base level NPC inherit, and it's true throughout the entire area. Do you want to make it so nobody can portal? It's a matter of seconds to add the property to every room across the entire development. The convenience of changes like that cannot be over-estimated, and that's without talking about how easily you can add complex base-level functionality.

Handlers can remain empty, but if and when they are written this is where they should live. Handlers have such wide-ranging impact that it's vital everyone knows what they are, where they are, and what they do. They shouldn't be hidden away in a subproject's sub-directories.

It's not so much the framework here that's important, it's the fact that everyone knows where things should be. It greatly reduces confusion and improves integration across an entire development, and that's extremely important.

# Conclusion

Developing in your /w/ drive may seem sensible, but it carries with it a penalty in terms of ease of integration. When working with multiple developers on a project, or even with multiple developers on a suite of related projects, integration issues can easily surface and cause disproportionate amounts of heartache.

It is my suggestion that you adopt a process of continuous integration by incorporating your development into a large skeleton in the /d/ directory. This gives improved integration, a sense of context, and makes it easier for people to lend a hand with coding problems. The technical problems that come with an area being in a 'live' directory can be resolved using the _dev and _pt naming conventions. This has the added advantage of regulating the use of playtester commands in a sensible way.

# Group Dynamics

## Introduction

You and the other members of your domain form a Team. A team with a capital T! However, teams that don't have the right kind of dynamics tend to be problematic and cause issues for other teams in the developer base. Your domain should work well as a group, with everyone complementing everyone else to produce a whole that is greater than any one individual is capable. That only works if the internal dynamics of the group are such that individual interacts enhance, rather than detract, the efforts of others.

At the same time, you are part of a larger Team – that of the creator-base as a whole. There is sometimes a tension in your domain team versus the larger context within which that team operates. It's easy for a team to be insular and inward-looking rather than part of a larger, collaborative effort. In this chapter we'll talk a little about the dynamics of domains, and where you should be wary of treading.

## What is a Domain?

Domains in Discworld have existed since the start of the MUD. Partially they are an artifact of our top-down administrative structure, and partially they are a way of decentralizing the responsibilities of development and focusing people within a particular well-defined (or at least, hopefully well-defined) remit. Perhaps the most important effect of a domain though is the impact it has on creator cohesion - it allows the creators to form a productive group. A group is formed when certain elements are in place:

- A shared sense of identity
- A shared purpose
- A conformity to some set standards or hierarchy
- Individuals have clearly defined goals.

All of these things are in place for a domain – the geographical (or abstract conceptual) remit gives a sense of purpose for the members of the domain, and the fact that everyone is working towards this gives a sense of shared identity. Each individual has (or should have) a clearly defined goal – for most creators, this will be their project. For others, the goal may be more nebulous, but we'll talk about that.

Each domain has its own standards and hierarchy, and partially it is this adherence to specific standards that defines a CWC creator from a Forn creator from a Ram creator. Every domain handles thing slightly differently, and those differences contribute to the sense of identify.

There are different kinds of groups that exist, some of which are healthy and some of which are troublesome. We'll talk about those a little later.

Domains help us deal with problems that are otherwise intractable, as well as distribute out effort to ensure that all parts of Mister Pratchett's world get creator attention. It could be argued that they have outlived their usefulness, but that view tends to focus on the problems that come along with domain atrophy rather than the usefulness of domains as a mechanism for fostering group interaction. When they work, they work very well.

Domains restrict the development focus to a subset of the Discworld. This greatly increases cohesiveness of development because people only need to be an expert in that subset. A ram creator doesn't need to be word-perfect on CWC, and a creator for Forn does not need to know anything about Klatch. There are several dozen Discworld books now, and nobody can be completely conversant with the details of all of them. It's perfectly possible though to be a 'subject matter expert' in your own narrow field. The benefit of this is that the development is richer – it can be full of deep detail born of knowledge rather than superficial familiarity.

In addition to the benefit of managing development, groups provide several highly useful social benefits. A good group provides a support network for its members – everyone is part of the team and so it is to everyone's benefit if particular individuals are supported. If you are having problems with development or meeting your obligations as a creator, you will often find the members of your own domain are your first stop for support.

One of the reasons for this comes from the previously mentioned shared sense of purpose – this incentivises collaboration because a win for the team is a win for everyone. If the domain has a significant success (such as a new project being put into the game), everyone gets a little bit of that reflected glory.

In the overall development direction of the MUD, it's sometimes hard to feel that your voice has any weight – within the confines of your domain, your voice counts for much more because it's one of a smaller chorus. The exact amount of influence your voice has will of course vary from domain to domain, leader to leader and person to person – but there are fewer people who have the authority to comment on the development of a single domain as opposed to mud-wide development. Having a sense of common ownership over a domain direction enhances the connections between group members.

None of this is to say that domains are All Good All The Time. The domain structure carries with it a number of significant drawbacks, but these are almost all related to domain atrophy – when a domain is no longer an active part of MUD development. This can occur for a number of reasons, but the most common of these is the disappearance of active leadership and a lack of development momentum. Most of the benefits of a domain come from the 'buzz' of working within a group of people – if that buzz is not there, the domain can be a millstone rather than an energy boost.

Individuals draw their sense of organizational norms from the people around them – that's what people mean when they talk about organizational culture. If the norm in your domain is an active culture of development, then that's the lesson you'll take away. If the norm is for apathy and a lack of interest in what's going on, then alas that's what you'll tend towards also. Breaking these kinds of negative patterns isn't easy, but it's possible. We'll talk a little about that later.

# When Is A Group Not A Group?

All domains are groups, but there are also ways in which groups can be subdivided. The simplest of these is to divide groups into either teams or cliques. You can think of teams as 'groups done right' and cliques as 'groups done wrong'. The difference really comes in terms of the inclusiveness of the group – teams are inclusive and outward facing, and cliques are exclusive and inward facing. Teams foster collaboration within a domain and also integration into the larger community of the MUD. Cliques are hostile to outside 'interference' and prone to organizational dysfunction as a consequence.

As a matter of collegial courtesy, our MUD is far more open than most development environments. Code is freely available, and the decisions taken by individuals are open for discussion. This is a healthy, albeit challenging, environment – you don't get the benefits of secrecy in how you deal with things.

Teams embrace this kind of environment, whereas cliques withdraw from it. The biggest differentiating factor of a clique is a sense of unity that members are 'better than' other groups. The 'cool kids' at school are a common example of the dysfunctional nature of a clique – individuals within the clique are accorded higher status than those who are not within the clique. In order for members of a clique to be 'better than', then everyone else must, by definition, be 'worse than'. This attitude tends to manifest itself in bullying, or dismissive behaviour towards outsiders.

In terms of integration within a larger MUD community, the inward nature of a clique tends to make it overtly hostile to outsider commentary. This is destructive to the fabric of collegiality that is at the core of Discworld creating. First and foremost, the loyalty of a clique is to the clique. This loyalty over-rides the wider responsibilities of membership within the creatorbase, with all the attendant problems that brings. Often this loyalty is based on the presence of an 'Alpha Member' attended by one or more lieutenants. This lends itself towards a cult of loyalty in which pressure can be brought to bear on clique members to ensure a consistent attitude amongst members towards outsiders.

In the past, it was common for domains to have clubs for their members. These were sensible and appropriate, and permitted easy interaction between members in a way that was not obtrusive to the wider creatorbase – it's not appropriate for one domain to fill up the creator channel with domain specific chatter, for example, and difficult to co-ordinate a domain wide discussion with tells. However, one side effect of this process was for domains to become exclusionary – you became a member of the club through being a member of the domain, and membership of this club was usually not permitted to others. This engendered an often unintentional secrecy in domain development that ultimately resulted in clique-like behaviour from even otherwise perfectly functional domains.

That's why we now have creator channels for each domain – the ease of communication between members is provided, but access to the channels is an assumed right for all creators of senior rank and above, permitting a greater degree of inclusiveness and transparency of decision making. Some domains go even further than this and make the domain channel available to all creators regardless of position (Learning being the obvious example of this).

The question then is, 'how can I tell if I am a member of a team or a clique?'

It's hard to tell, from the inside – it requires a willingness to be rigorously honest about your dealings with your fellow domain members. There are though some questions you can ask that help clarify the situation:

- Does the bulk of domain conversation occur in an inclusive forum?
- Are your decision-making processes open for comment to the wider creator-base?
- Are differences with other creators resolved constructively and in the open?
- Is outside criticism taken as constructive and assessed for validity?

If the answer to any of these is 'no', then it is a warning sign that you may be in a clique. These are the features of a team that enhance creator-wide collaboration and inclusiveness. There are also features of a clique specifically that highlight potential dysfunction. Ask yourself the following questions:

- Is there a tradition of loyalty to the group over loyalty to the MUD?
- Is there a tradition of covert sniping about professional disagreements with outsider individuals?
- Is there an obviously dominant alpha member who distorts collegiality with outsiders?
- Do significant portions of domain conversation occur in a private, exclusive forum?
- Is there a tendency within your group to react aggressively towards negative criticism?

Cliques share many of the features of constructive groups – they foster a sense of community, shared purpose, and an incentive for collaboration. However, for all the reasons above, they cause a distortion of the positive atmosphere of open inclusiveness we try to foster across the MUD and create an atmosphere in which group-think can thrive (more on group think later).

If you think your domain may be a clique rather than a team, the simplest thing to do is just voice your concerns – the response that this gets will soon tell you the truth of the matter! If you are part of a clique, my advice is that you speak to an uninvolved director or admin as soon as possible and ask for guidance and support. Feel free to come to me if you want to discuss your worries on this score – I am, after all, a Professional Outsider!

# Group Roles

Each individual within a group has a particular role. Sometimes this role is highly defined, such as project or position in the administrative hierarchy. Sometimes it's a more nebulous, social role. Sometimes people take on more than one such role within a group. If everything is working effectively, you'll find everyone has at least one positive role they are playing in terms of group dynamics.

Some of the roles that tend to be highlighted by group dynamic theorists are:

- Encourager
- Gate-Keeper
- Harmoniser

- Mediator.

A good group has people who exemplify all of these roles.

The encourager is a cheerleader for the group, actively listening to ideas and encouraging participation from all group members. An encourager can motivate people on the fringes of a group to integrate more completely into the team.

The gate-keepers are people with large stocks of bridging capital – they help connect distributed subgroups within an organization. It's impossible in any large group of people for everyone to form cohesive bonds with each other, and gate-keepers allow for the lines of communication to be kept between disparate and unrelated clusters.

Harmonisers work to reduce tensions in a domain. No matter how well a group works, tensions will be encountered as a result of the friction of day to day work. Harmonisers help resolve those tensions and smooth over problems so everyone remains committed to the group goals. This has a great deal of overlap with the role of the mediator, who can work to resolve tensions by encouraging compromise.

However, in dysfunctional groups there exist particular 'anti-roles' that actively disrupt the process of building good team dynamics. It's important that people know what these anti-roles are, and identify individuals who demonstrate those features. Knowing that a problem exists is the first step towards resolving it.

Well known anti-roles include:

- Dominator
- Aggressors
- Deserter
- Recognition Seeker

The dominator is the person who feels as if they must monopolise every discussion, to always be the one to make all the suggestions and to mould the domain development strategy in the way that they intend. Group collaboration works only when everyone's views can be heard.

Aggressors work to undermine alternative viewpoints by attacking them with unwarranted aggression. Aggressors discourage participation by adding an element of psychological trauma to putting forward an opinion. These two anti-roles work together to produce an environment in which group-think can thrive – coincidentally, these two anti-roles are most visible in groups that can be defined as cliques.

Deserters are perhaps the most harmful of the anti-group roles. Deserters simply withdraw from the process entirely, and coast on the effort of others. This kind of behaviour is often referred to as 'social loafing' or 'free-riding', in which the individuals rely on the relative anonymity that being party of a group provides. When one is an individual working on a project, all the work can be attributed to that individual. In a group, the work is distributed so it's not so easy to identify who is pulling their weight and who is not. A social loafer is one who takes advantage of this to gain the benefits of domain membership without actually meeting their obligations to produce content.

Finally, there is the recognition-seeker. Groups work best over the long-term if successes are shared amongst all members. The recognition-seeker subverts this by insisting on individual recognition for their actions. Everything on Discworld is a group effort, but the recognition seeker seeks to put their name ahead of other contributions.

You will probably recognize examples of all of these from people you have known and worked within the past. It's important in a successful group that anti-roles are discouraged and positive-roles enhanced. Talk with your domain administration whenever you feel someone is falling into a persistent anti-role within your team.

# Group-think

Group-think is a feature in many team-settings, and is a consequence of individuals not being willing (or indeed being able) to put forward viewpoints contrary to those of the group as a whole. Such behaviour is endemic in cliques (because of the Top Dog mechanism) and in groups with large numbers of aggressors and dominators. As a result of alternative viewpoints not being put forward, a group reaches a premature consensus without actually analyzing the ideas that have been put forward in a suitably critical manner. The exclusionary nature of a clique also insulates it from external viewpoints in a way that can create a sense of 'false invulnerability' because competing viewpoints are simply not available.

It is wrong to think though that this is a feature only in groups with systemic problems – sometimes it can come from a group simply having too many people with the same kind of background – good collaboration comes from having many people with different view points, not having many people with the same viewpoint.

Groups that suffer from this tend to have an implicitly defined 'comfort zone' in which discussions can take place – topics outwith the parameters of this comfort zone are usually not expressed or considered in the decision making process. Individuals do not wish to incur the wrath of an Alpha Member, or risk looking foolish as they are attacked by an Aggressor – it's not the case that the conclusion is foregone, but there is an element of self-censorship applied by all members to the way in which they set the limits of their expressed opinions. As a result of this, even though individual agreement with a topic under consideration may be marginal, the overall impact of the social processes of the group is to suggest an overwhelming mandate because nobody has raised an alternative viewpoint.

There is some fascinating research by a social psychologist by the name of Solomon Asch looking into conformity in group situations. The simplest experiment he did was known as the 'line experiment, and worked like this:

The individual being observed was put into a group of between five and seven 'confederates' of the researcher. One by one, they were shown a card with a line on it, and then another card with three lines marked A, B and C. The participants were then asked to select which of these lines matched the length of the first line they had seen. For the first few trials of this, everyone would select the right card. In the next trial, the confederates of the researcher would then pick the wrong card. Astonishingly 75% of the subjects in the experiment conformed to what the group had selected at least once, even though their choice was obviously wrong. The social pressures of group are so intense that even in a group of strangers, there is conformity out of a a fear of looking foolish.

However, one of the more encouraging things to come from the study was the counterpoint that if at least one other person goes against the prevailing group opinion, conformity rates plummet. The answer to group-think is to stand your group and speak your mind – your alternative viewpoint could be the thing that brings out all the alternative viewpoints in everyone else.

# Conclusion

Groups foster a sense of cohesion and domains are a powerful way to harness that cohesion for positive ends. However, the dynamics of groups are complicated and prone to dysfunction unless everyone is aware of how easily inclusive and effective teams can become insular and exclusive cliques. The most important thing to do when you are part of a clique is to recognize it for what it is and to make an attempt to break the dysfunctional elements. This can be hard, because by their very nature cliques can exhibit tremendous pressure on members. If you feel that you are not able to do so without support, then talk to someone outside of the clique for guidance.

Some of this has a horrible air of 'if you're being bullied, tell the teacher', which is not how it is supposed to be interpreted. Social forces have an unbelievable amount of power in a closed society – if you want some proof of this, I recommend checking out the book 'The Lucifer Effect' by Phillip Zimbardo. The power of bad group dynamics to influence individuals for the worst is such that these dynamics must be broken when they are observed. The first step in that is recognizing the problem, the second is getting the support you need to change the system.

# Project Management

## Introduction

Project management is one of the skills that any successful Discworld creator is going to have to acquire. While there is a domain administration team who are responsible for managing the larger scope of a development, it is you and you alone who have the responsibility over managing your time to ensure that your code is developed in a timely and effective way. Project management may be a somewhat grandiose way to refer to this - it brings up thoughts of Gantt charts and costing models. I'm not going to talk about any of these systems in this chapter, because they are very dull and I'd rather chew off my own feet.

What I will talk about is the self-regimen needed to deal with the complexity of day to day development. This is a far more interesting topic, and one that's actually possible to condense down to a single chapter. Those (few) readers who are interested in the formalised aspects of project management should consult... well... a therapist, I guess.

## Project Management 101

Most projects are sufficiently large that they dwarf a novice creator. When you look at the number of things you need to code, it can be overwhelming. Unless it's a project of a handful of rooms, you'll find your ability to deal with the scope of the development is limited. A plan is called for!

In the material for Being A Better Creator, we looked at creating a feature density chart in which we outlined each of the things our development would contain. A development plan is the same thing, except simpler and more granular. How granular you choose to make it is entirely up to you. Imagine a development plan for a simple room - a room has a number of sub-parts that can be done independently of each other:

- A room has long descriptions for day and for night
- A room has chats for day and for night
- A room has items for day and for night

Each of these is a 'task' for your project. When all of the tasks have been completed, then your project is done. As a principle, this is known as **incremental development** - each bit of the project is small by itself, but and you incrementally build your project by completing these small tasks.

Defining what is a task and what is not is something that comes with experience of your own capabilities. My recommendation is that a task should be something that takes perhaps an hour or two of development time - that way you can easily slot it into whatever time you can spare for the MUD. It gives a sense of progress if you can tick off things that have been completed as you go along. They should be big enough to give a sense of satisfaction as they are ticked off, but not so large that their sheer size is discouraging.

Choosing tasks is a matter of individual preference -it's what makes logical sense to you as a subdivision of labour. Some people like to have individual rooms and objects as tasks, some people like to have things conceptually linked such as 'write room skeletons, write add_items for rooms, write chats for rooms). It'll be influenced mainly by your own way of approaching your development. There's no right or wrong way.

Once you have decided on the tasks that go with your project, you'll need a way of keeping track of how far you are through each them. At the simplest end this can be done with a few columns on a spreadsheet, or in a notebook. However, if you want to make the results of your planning easily available to other interested parties (such as your domain administration team), you may want to consider making use of our project tracker software. There will be more on that later in this chapter.

# Frameworks

Good projects have a framework that shows how everything links together, and the development is then attached to this framework as it is completed. The skeleton area we put together for Learnville is an example of a framework - it allows for a system of continuous integration amongst multiple developers. Any project you develop should have a framework, even if you are the only person developing it. Incremental development is the easiest way to ensure that this framework exists and that each development effort moves you closer towards your eventual goal.

Consider something outside the normal examples of this material the Discworld Oracle. It's partially web-based (the HTML front-end) and partially mud-based (the handler that works it, and the MUD-based front-end). The framework for this was to setup the simplest possible connection between the back-end and the front-end - a handler that had no functionality, and a front-end that had no interface. The first step in providing incremental development in this system is to add an option to the interface - for example, to list all questions. This then forces a further development in the handler (representing questions and then providing a list of them when queried). When that's done, there's the Discworld Oracle. From that point onwards, you're just adding features.

Imagine doing that the other way - writing the handler and then trying to put a front-end on it. Such a development would be ten times more difficult because of the scope of development. Before you can even see any kind of progress, you need to write all of the functionality that belongs to the handler. You then need to worry about how the interface is going to look and how it is going to communicate with the handler. Only when you have completed both of these tasks do you actually get to see the system function. Then when you have the both side of the system developed, you have to test the whole thing - if something doesn't work, where is it not working? Is it in the interface? The handler? The connection between the two?

Bite-sized development in a framework gives you instant cues as to whether your development is progressing along the right lines. This simplifies your development (you can correct faulty assumptions right away), eases your testing burden (you know where problems are to be found because they are in some small subset of functionality you just implemented), and motivates development (you can see this Mighty Organ grow under your gentle caresses).

# Communication and Team Roles

Project management is all about communication - indeed, you've probably noticed that's a pretty major theme throughout this material. Much as within a domain, a project has a number of roles that must be fulfilled. Sometimes all of the roles will be fulfilled by one person - that's absolutely fine. It's also fine for one role to be filled by multiple people. However! If this is the case, communication is vital to ensuring that there is no overlap of authority.

## Domain Administration

The domain administration team, unless they are an active part of the development, are usually 'hands-off'. Responsibility for ensuring thematic correctness and feature density belongs with the project leader, who is appointed by the domain administration. It is the domain administration who give the go-ahead for a project and also who will provide approval for the final set of features to be developed. It is also the domain administration team who will approve a project for entry into play-testing or the game according to domain standards.

## Project Leader

The project leader in the development is the one who is responsible for setting the 'vision' of the project. They decide on the number and relative complexity of features, and how the project integrates with the larger context of the domain. That's not to say they are the only ones responsible for developing the plan - they're just the ones that get the final say on what is to be put forward and what is not. The domain administration is the source of approval of the plan, but the project leader is the one who develops the plan for approval.

For projects involving more than one developer, the project leader is usually the one with the most experience coupled with the best people skills. Project leading is more of a social activity than a development one, although it is unusual on Discworld that a project leader will not also pitch in with the coding.

The project leader is also responsible for keeping everyone motivated, and for making sure everyone is happy with what they are developing and knows how they are contributing to the overall effort. Nothing sows dissatisfaction easier than people not being sure why they are doing something. The project leader identifies developmental problems and puts in place solutions to deal with them.

## Documenter

The documenter in a team is the one who makes sure that the result of discussions is made available in a suitable format - for example, on the wiki. The documenter is usually the one who keeps the project tracking documentation up to date to ensure it reflects the developmental reality. While this is unlikely to be a 'full-time' role within a project team, it should be an on-going role. Part of the requirements of a successful project is providing your domain administration with updates on progress - it is almost impossible to co-ordinate a domain unless you know what everyone is doing, and how much of it they have done. A documenter can ease this burden on their domain leadership by making sure the information is available in some pre-agreed store.

## Developers

Almost everyone on a project team will be a developer - we don't have such a fine specialisation of creator jobs on Discworld as such that a creator will be a 'professional manager'. Everyone is expected to dual-role to a degree. Developers are responsible for actually implementing the plans that have been agreed upon, and making sure that the project leader for the project is aware of problems in a timely manner. Developer problems can revolve around many different areas, such as real-life distractions, technical issues, and social issues within the project. As long as these are being communicated to the project leader, then these blocks can be removed with appropriate redress. Real-life issues can be dealt with by reassigning some duties to a developer with more free time, technical issues can be resolved by pairing a developer with someone who has the necessary coding skills, and social issues can be resolved through mediation and conflict management. All of this can only happen if the project leader is made aware of the problems.

# Subdivision of Effort and Ownership

The only way in which a project works is if everyone has their own little part of it they are developing. Part of the job of a project leader is to subdivide effort so that everyone has something to do, that they are the person who is responsible for it, and that they are happy with the work they are to do. Every task should have a named creator who is responsible for its development - if things are left as 'we're all responsible for this' it generally means that no-one is responsible. This is a well established principle in social psychological research where it goes under the name of the 'bystander effect'. Nobody works towards the task because everyone thinks someone else will do it.

Responsibility in project management is another word for 'ownership' - this may seem to strike at the heart of the 'communal code' system we operate on Discworld, so I will talk about this a little to explain why ownership is important in project development and why it doesn't violate our deeply held principles of shared source.

Ownership can be expressed in a level of autonomy over design choices. A bad project leader enforces a development from the top - 'This is the development you will code, and it will be coded exactly like this'. Projects that operate under this principle hardly ever succeed. In volunteer developments especially, coder motivation is a finite resource that has to be grown and nurtured, and that doesn't happen through management by fiat.

Half the fun in developing is to come up with ideas and bring them to life. It's nowhere near as much fun to implement someone else's ideas. This is something that our players often don't realise - an idea report such as 'Do something cool with otters' is usually a far better prospect than a five page idea report on a deeply-complex 'otter management' subsystem - creators need to be able to take ownership for the ideas they are developing. The code that is being developed is MUD code, but delegating the responsibility for deciding what that code is going to be is a valuable motivational tool. When people have ownership over something, they have an incentive for doing well with it, and they can take a pride in its success that is not available when all you did was set out the code. Half the fun is to plan the plan.

Subdividing effort goes hand in hand with delegating ownership of the project. Subdivision should impact on the 'fun stuff' as much as the 'boring stuff'. When you are subdividing effort you should make sure that some of that effort involves a degree of freedom to operate. 'You're responsible for designing and coding the quest in the castle' is a better task than 'I have designed the quest on the wiki, and it's your responsibility to code it'. The latter means that the coder can later shrug and say 'I only coded it, someone else designed it - it's nothing to do with me that it sucks'.

The ownership then is not over the project or the code, but is instead over the way a part of the project is to be designed and evolved. Once the project is completed, then that ownership is lost and it transfers back to the domain (and the MUD in general). It is a short-term ownership, rather than something that is gained for the long-term.

# The Discworld Project Tracker

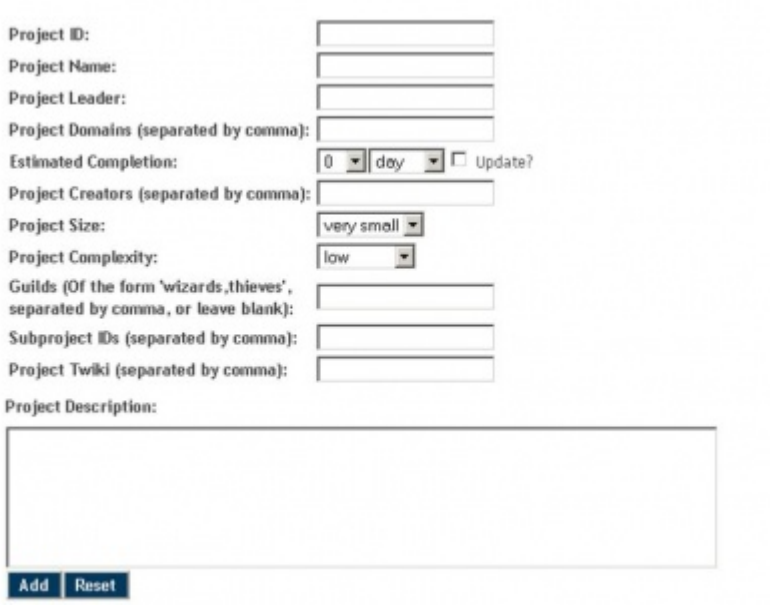
The MUD actually has a project management system that is available - or rather, it's a project tracker system with some management facility. You can find it http://discworld.atuin.net/lpc/secure/creator/project.c here. It's important to know how the project tracker works, because all projects that enter the playtesting stage must be registered with the tracker. How you choose to use it beyond that is entirely up to you.

When you open up the tracker, you'll be greeted with the following menu:


There will also be a list of projects that have been 'touched' (modified) in the past week. Feel free to browse around these, it's good to see what's happening elsewhere in the MUD.

Each domain has an entry along the top, and clicking on the link will bring up the projects that belong to this domain. Note though that this won't necessarily reflect the full range of development since projects often do not get registered with the tracker until they are in playtesting, and projects that have been abandoned often still linger because no-one has removed them.

It's very simple to add a project to the tracker - you click 'add project' at the top, and you'll be given the following page:




The Project ID is the unique identifier for the project in the system. It's the special code used to distinguish one project from another. This can be anything you like, but it will be made into one word when your project is added. The name is a 'human readable' short description of the project.

We've spoken about what a project leader does in a project - you should record who the leader is so there is a clear chain of authority. Notice you can only have one of these - a team can't function effectively with two chains of command. Some projects belong to several domains, and a list of these can be provided. Projects belonging to multiple domains will show up in the individual domain queries.

Estimated completion is how long the project is expected to take. Note, this isn't a promise, but an estimate. You should be realistic with this - overly ambitious estimates are of no use to anyone. Next, all the creators involved with the project are listed - the links associated with a creator's name in the project tracker will bring up all of the projects with which they are involved.

Project size and complexity are two important values as they directly impact on how much playtesting attention the project receives. Don't fret it too much to begin with - you can easily change any of the data you enter at a later date. The size of the project influences how many playtesters are assigned in a playtesting rotation, and how long a rotation period lasts. The complexity influences how many rotations a project receives. This is all handled automatically by the playtesting system, but setting these values correctly is important in ensuring your project gets tested effectively. The figures relating to each of the values are found at http://discworld.atuin.net/twiki/pt/bin/view/Playtesters/PlaytestingRotations.

Guilds allows you to indicate which guilds should be included as part of the rotations - there's no point in thieves testing a wizard only area, for example. Leave this blank if there are no specific guild restrictions.

A project may have several substantial subprojects, each with its own entry in the handler. Take a look at the entry for Genua City to see this in action. You can list each of these subprojects as being part of this project.

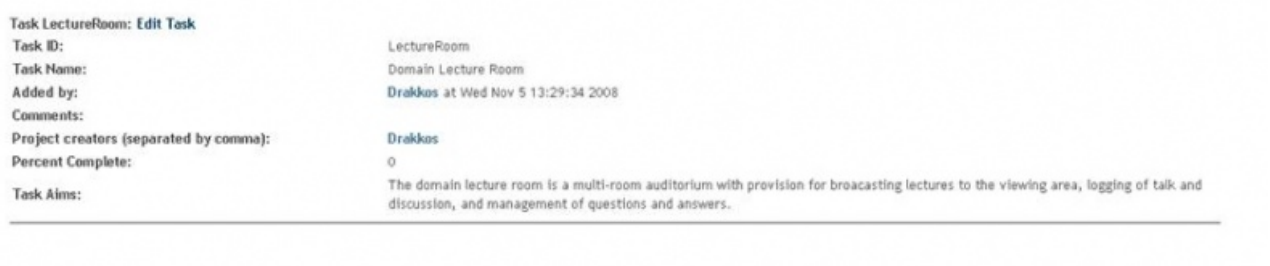
If you have a wiki for your project, then make sure you add its details. One of the benefits of the project tracker is it makes it easy to integrate all these disparate bits of information into one consistent location. Finally, you can give a short description of the project - you don't need to be too detailed about this, certainly if you have a wiki page set up. It's useful for those who want to know, at a glance, what the project involves.

All projects initially start off on the 'Heap' - this is where they're officially a domain project, but no active development is ongoing. By clicking 'edit project', you can set the project to be one of a range of other states:

| State | Description |
|---|---|
| Heap | Heap projects are on the 'todo' list of a domain. No active development is occurring. |
| Development | The project is active and undergoing developer attention. |
| Playtesting | The project has entered the playtesting stage, and is recorded with the playtesting handlers. Don't set a project to this unless you are actually releasing it, because it triggers assignment of PTs and automated mailings. |
| Play | The project has been completed and is now in the game for players to enjoy (or not). |
| Limbo | The project's status is undecided, usually because some domain development has been invested but the project development team is no longer active, or because a project has left playtesting and its future state is currently under review. |

When an area enters playtesting, you should also fill in the 'notes for playtesters' section indicating how the development may be tested, how it can be reached, and any features you would especially like to receive playtester attention.

You can also add tasks to the project:




These tasks carry with them details on how complete they are, and this will automatically update the 'overall' completeness of the project - as you complete tasks, you'll see the project to which they belong become more complete over time. You can override this behaviour, but if your tasks are comprehensive then you won't have to.

# Conclusion

Project management on Discworld is about communication, not processes and diagrams. However, a basic understanding of roles, responsibilities and subdivision of effort can make an otherwise problematic project function smoothly. At the very least, having a list of your tasks and their status is hugely useful in giving you a sense of the scale of your development and progress on an incremental basis. This insight is invaluable for communicating your status to those who are responsible for co-ordinating your project across an entire domain.

While the project tracker software on Discworld is mandatory only for projects entering playtesting, it can also be used for providing easy access to your development progress to anyone who may be interested - while only your domain administration are likely to have a specific 'need to know', this visibility of effort is something that enhances the collegiality that leads to a successful development team.

# Maintenance

## Introduction

Maintenance is an on-going task in a continually evolving environment like Discworld. I think it's fair to say that, with the exception of a few individuals, we don't do as much of it as we really should. The sheer complexity of our codebase virtually guarantees a never ending struggle to fix defects and cope with changes elsewhere in the game. It's an important process though and deserves proper consideration when we talk about how individuals in a domain should pull together to improve the quality of the code in the game and in development.

Reading and understanding code requires a different set of skills to writing code – they are related, but not identical, skill-sets. Maintenance involves being able to look at a piece of code and dissect its inner workings. You shouldn't turn down an opportunity to do maintenance work if it presents itself (and it always does) because you'll get a good test of skills that you may or may not yet possess, and it's a way to provide a very real positive impact on your domain.

## Maintenance In The Software Development Process

We don't follow a formal software development process on Discworld, but if we did maintenance would be at the very end of it. Maintenance is traditionally the phase of software development that consumes the largest number of resources – while development is temporary, maintenance is eternal. Changes always need to be made to code, and while the problems can be multiplied by a bad software development process, they are simply a natural artifact of the complexity of software. You can't get rid of them, no matter how carefully you code – and even if you could, maintenance centred on fixing problems is only part of the process.

Maintenance tends to fall into one of four categories:

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance
- Preventative maintenance

Corrective maintenance is what people tend to automatically associate with the process – fixing bugs. Bugs don't necessarily mean the code was badly written – many problems are noticed only when enough eyeballs have been passed over the code. No matter how carefully you plan, bugs you could never have dreamed of will be reported when a development goes live. Players will always attempt things you could never have anticipated:

```
'Yeah, so I tried to eat the table and I got a runtime error...'
```

Adaptive maintenance is updating code to reflect changes in the underlying software systems, or the provision of additional features in existing code. In all cases, this has to be done with compatibility with existing code in mind, and with provision in place for ensuring previously stored data persists over modifications.

Perfective maintenance concentrates on improving the maintainability and efficiency of existing code systems. The driver on Discworld is single-threaded, and our highly complex code-base means we eat up an awful lot of CPU and memory resources. Perfective maintenance is about improving the scalability and performance of code. Optimization is a secondary concern for sensible software developers – it's only when code is operational that you can really see what code needs attention in this department.

Preventative maintenance seeks to fix small problems before they become bigger problems – there are always a few of those in any development. I like to refer to this as the 'we'll fix it in editing' problem – every so often there is a persistent bug you can't track down, and rather than fix the root cause you apply a band-aid fix.

One example of this would be if your code was persistently getting an 'off by one' error in a loop and you were unable to find where the extra iteration was coming from. You can either track it down and solve the problem (which could take hours, if not days), or say 'Oh, I'll just make it loop one less time'. That's a band-aid solution – the problem isn't that the loop is being executed one extra time, the problem is that the number of loops isn't what you're expecting it to be. That's almost guaranteed to cause problems elsewhere in the code, given time. Preventative maintenance takes these band-aid problems and resolves them.

The first thing that is needed for a maintenance strategy in a development environment is a formalized system for recording bugs when they are encountered. Luckily, we have a very powerful and entirely bespoke system on Discworld for doing this. We'll get to that in a few moments.

# Domain Maintenance

Few domains have a formal approach to maintenance. It tends to come and go in cycles – the bug-count for a domain will reach a point whereby the domain leader says 'Fix these or I'll cut you!', everyone in the domain spends a few days reducing the bug-count to something a little less painful to look at. Then everyone goes back to their Regularly Scheduled Programming and the whole cycle repeats. Every now and again a domain appoints a 'maintenance czar', but it is often thankless and grinding work with few observable gains. No matter how hard you try, the bugs keep coming.

The best approach is to have everyone pitching in – that way, nobody gets overwhelmed and everyone has to take responsibility for bug-counts in their assigned directories. You'll have probably noticed that regular reports on the domain's bug-count get posted to your domain board. These are valuable updates on the relationship between bugs reported and bugs fixed. There are also useful graphs of activity to be found here showing the relationship between bugs opened, and bugs fixed.

Everyone has to take ownership for this process – bug fixing is a domain task, not a task for individuals. There are bug-fixing stats that can be accessed using the fixed command, such as:

```
fixed all
```

Or:

```
fixed drakkos
```

The figures for this are somewhat distorted because of several large-scale crashes of our bug-fixing databases – those bugs that were fixed prior to the last crash do not show up on the tables. However, the tables can be a good motivational tool and a way to make a game of maintenance – friendly competition within a domain can be healthy, provided it remains friendly.

It's a simple rule for a domain to aim for to fix more bugs in a week than are reported – that way the count heads in the right direction. It is possible, through hard-work and commitment, to get a domain bug count to single digits and keep it there. Bugs will always be reported, but if everyone takes ownership of maintenance they can be dealt with quickly and effectively – that's even something players tend to remark on!

Microsoft have a system of maintenance called Zero Defects (don't laugh) – this doesn't mean that their code has no problems, it means that their development strategy is that no new features are added until all known issues are resolved. A variation of such a strategy would be a worthwhile policy for a domain – 'no new development while the bug-count is above 50!'. This focuses everyone on the collaborative effort to manage the domain's responsibilities towards maintenance, and communicates the importance of the effort to all developers.

# Where Do Bugs Come From?

Bugs come from all sources, and sometimes they are not simple to fix. Sometimes indeed they are unfixable. Sometimes they are introduced as a consequence of fixing another bug – a well known informal metric is that for every two bugs you fix you run a good chance of introducing a new one as a side-effect. Another metric derives from experimental data suggests that, as far as commercial software goes, there is an average of between twenty and thirty bugs for every thousand lines of code. That's commercial software mind, written (one assumes) by people with formal training as software engineers.

Sometimes bugs stem from temporary issues – someone broke an important object and everyone runtimed. Those runtimes led to dozens of bug reports unrelated to the domain in which the bugs were reported, and as such don't really reflect the issues with domain code. A quick pass through the domain bug list can remove these from the statistics.

Sometimes they're not actually bugs at all, but ideas that have been misclassified. The more opinionated players may report their ideas as bugs on the grounds that 'My not being able to do X is a clear bug'. Our error system allows for these to be easily reclassified, so again a pass over the domain bug lists can remove these from consideration.

Sometimes the bugs are legacy issues – because of a mismatch between the old way of doing things and the new way of doing things, code stops working properly and nobody has re-factored the code to fit in with the new regime. Such bugs can last for a long time because the amount of effort that need be invested to fix them far outweighs any potential benefit.

Sometimes the bugs are unfixable – we have several of these and they are a consequence of the underlying structures not permitting certain kinds of functionality. It's hard to deny them because they are actually bugs, and you can't mark them as fixed because fixing the problem is outwith your powers.

Sometimes bugs simply linger, because nobody found them interesting enough to fix. Some bugs are more interesting to deal with than others, and so these tend to be dealt with preferentially leaving the more uninteresting bugs undealt with. Every domain has a good few of these moldering in the archives.

# Bug Triage

When dealing with a massive backlog of bugs, it's worth adopting some form of triage system. Triage is a system of prioritizing based on severity so that the greatest gain is obtained from limited resources. Your development time is not infinite, and time you spend bug-fixing is time that is not spent on your assigned projects. Triage can be based on impact, and also on severity.

First of all, there are the critical bugs. They have to be fixed as soon as is humanly possible. Usually these are the bugs that have an impact on the functioning of the MUD as a whole – infinite XP bugs, infinite cash bugs, or bugs that seriously impact on a player's ability to function (destroyed inventory, corrupted playerfiles, all that kind of stuff). All domain development should halt until these are resolved because their effect extends beyond the domain.

Next come the high priority bugs that seriously impact on the functioning of code within the domain – a fault in a domain-wide crime handler, or a fault in a domain-wide room inherit, would be examples of these. A bug that allows someone to completely clear their criminal record would be an example of a high priority bug – it doesn't impact on the MUD as a whole but seriously infringes on the correct functioning of the domain. Moreover, the scope of the bug is across the entire domain.

After that are the medium priority bugs – things that impact on an area of a domain but do not extend further. An issue with a key area feature would fall into this category – a problem with a quest, a broken shop, or a malfunctioning area inherit would all fall into this category.

Finally come the low priority bugs – things that affect a single room, single item, or single NPC.

The priority of the bug can be cross-referenced with the severity to give you a more granular view of the issue:

| Issue | Severity |
|---|---|
| The code is completely broken and does not load. | Highest |
| The code gives disproportionately large advantage or disadvantage to users | Highest |
| The code does not function with regards to its key features | High |
| The code malfunctions with regards to its key features | High |
| The code does not function with regards to ancillary features | Medium |
| The code malfunctions with regards to ancillary features | Medium |

| | |
|---|---|
| The code has cosmetic issues | Low |
| The code has typographical errors | Low |

A malfunctioning domain level inherit that deleted player-files would thus be of high priority and highest severity – the only bugs you'd look to fix before that would be critical bugs of similar severity.

Having put together a triage for domain bugs, a concentrated effort can be made to resolve all the most important ones immediately, and the ones of lower priority can be addressed on a more incremental basis.

# The Error Handler

Our error handler is remarkably powerful, giving considerable control over the bug-fixing process. It comes in both web and mud flavours, although the functionality is the same for both. We'll concentrate on the in-mud system here as the web system should be intuitive for anyone who understands how the process works.

First of all, pick a directory that you know has bugs in it. You'll find plenty of these in the weekly domain status report that gets posted to the board. Navigate to that directory in the mud and use the command to invoke the error handler:

```
errors
```

You'll get an interface that lets you navigate through all bugs reported on that directory. The bugs will look something like this:

```
Bug #99682 OPEN BUG ROOM
Date Reported  : Tue Sep 23 16:43:54 2008
Assigned To    : ptoink
Reporter       : gruper
File name      : /d/waterways/islands/pirates_cove/cove/beach
Directory      : /d/waterways/islands/pirates_cove/cove
The Jolly Farmer and The Sea Pig show up as being docked here, but do not
exist according to the ship_handler.
Environment: /d/waterways/islands/pirates_cove/cove/beach (beachfront)
[1 of 1] STFCOLHA-+PNGQ :
```

Each bug in the system has a unique ID. If you want to see that bug specifically, you can view it using the pterrors command:

```
pterrors 99682
```

Note that there is a lot of information provided with this bug report. Along the top is the status of the bug, which can be any one of the following:

| Status | Meaning |
|---|---|
| Open | The bug has not been dealt with, and remains an issue to be resolved. |
| Fixing | We accept this is a bug, and we are actively working to fix it. |
| Considering | We haven't quite decided whether this is a bug or not, but we'll mark it as considering to show someone has looked at it. |
| Fixed | This was a bug and it has been fixed. |
| Denied | This is not a bug, you are on crack. |

Changing the status of a bug triggers an inform for the person who submitted the report – it's not purely for our own benefit.

Next is displayed the category of report – it can be a bug, a typo, or an idea.

Our example above is a bug of type 'room', meaning the report was submitted on the room in which the error was encountered. Bugs get categorized by which object they were reported on – they can be room bugs, object bugs, help bugs, ritual bugs, spell bugs, command bugs, or general bugs. This categorization allows for you to easily access the specific bugs in which you are most interested.

The rest of the bug report should be fairly self explanatory, except for the gobbledygook at the bottom – those are your menu options for what you can do with the error. Instructions to the handler are issued as a command code, and any associated data. Let's say for example that this bug doesn't really belong to this directory and we want to send it somewhere else. We can use 'f' (forward) along with the directory where we want it to go:

```
f /w/drakkos/
```

If we want to change the status, we can use 's' (status) along with the status we wish the report to have:

```
s fixed
```

If I want to provide a custom message to go along with this, I can add a 'custom' tag:

```
s fixed custom
```

Or if I don't want to send a message at at all:

```
s fixed none
```

I can use 't' to change the type of the bug:

```
t typo
```

Or 'o' to change the category:

```
o object
```

You can navigate forward and backwards with + and -.

The web-based front-end works exactly the same way, except with a nicer interface and easier access to functionality:


Selecting a directory in the web handler gives you a much easier way to navigate through all the reports.

A combination of the two often serves best – 'errors' to get an at-a-glance look at the errors in your current working directory, and the web interface for more involved work. Your mileage will vary though – some people exclusively choose to use one over the other.

# Conclusion

A solid maintenance strategy will help marshal individual efforts into a larger collaborative approach to domain bug-fixing. The downside of everyone owning the code is that everyone is responsible for bug-fixing – quality control is a joint effort within a domain.

Discworld has some very powerful error management software, and you should make an effort to become familiar with it as it is something you'll find invaluable as you do your day to day work as a creator.

While maintenance is not glamorous and does not give the same sense of player satisfaction that a new and fun piece of code will, it is vital to the well-being of the MUD. We have a massive bug count, and only by everyone pitching in will we be able to arrest the increases and eventually turn them into decreases.

# The Experience Divide

## Introduction

One of the persistent areas of conflict that exist between developers on Discworld is the divide between 'professional' software developers, and 'amateur' programmers. I don't intend for that to be pejorative, but it's a situation that must be resolved for effective collaboration to proceed. As a matter of full disclosure, I will make mention of the fact that my degree is in software engineering, and I have been involved in teaching software development and programming at all levels of higher and further education for the past seven years. I have also been called upon for external professional consultancy and software development, as well as research in the fields of artificial intelligence, knowledge management, and accessibility support. Whether that makes me a professional or one of those who 'can't do, so teach' I will leave as a judgement for individual readers to reach.

The fact that we require no software development experience for creators on Discworld has been one of our strengths in the past – creators require a fairly unusual set of skills in order to fulfill their day to day duties, and finding professional programmers with those skills is a daunting task. Instead, we find people with some of those skills (or who we believe can develop those skills) and work to provide an understanding of programming. In this chapter, I'm going to talk about some of the issues that come along with this, and how a constructive attitude on the part of professionals and amateurs can work to everyone's advantage.

## Professional and Amateur Programmers

While many of our creators have advanced training in technical topics, comparatively few have a background in formal software development. This is a very specific topic embracing a number of esoteric disciplines across a number of broad fields – requirements gathering, analysis, design and implementation. A software engineer is not the same thing as a programmer, it's a broader discipline than that. These are the professional programmers – by definition, programming is their profession.

On the other hand, many of our creators do have prior experience with programming in one form or another – introduction to basic programming concepts is a core part of many school and university curricula. However, no matter how much experience someone may have in writing programs for themselves or as an ancillary part of their job, this development is strictly amateur. By this I don't mean that their coding is bad (their code-fu may actually be very good), I mean that the mindset that accompanies the development is not that of a professional.

Partially, the difference comes down to an intended audience. Professionals, while they often write programs for their own purposes, are usually writing for an audience. The intention is that, at some point in the future, real people who are not the professional will make use of the software that has been written. Amateurs tend to write for their own use – to pass assessments, or to do some task that is unique to their own requirements. Amateur code may eventually make its way into the hands of others, but that's not why it was usually developed.

Writing software for other people engenders a certain way of thinking about software development. An amateur programmer can compensate for a lack of formality in development when the program is being used, and can discard inconvenient requirements at will. If an amateur has written a program to check their lottery numbers, for example, they don't necessarily need to have a user interface – it could be set to read the numbers in from a file. If checking for the bonus ball is too much work, then it can be left out of the functionality. The only person using the software is the person writing it. There don't need to be meaningful error messages or input validation – if there's a horrible error that occurs when using the program, it's fairly easy to work out what the problem was and correct for it. Amateur development gives an opportunity for some of the programming planning stage to be shuffled to the user interaction stage where they can be ignored or compensated for.

Professionals tend to fall into this mindset when writing code for themselves. I have a piece of code that I wrote for generating electronic versions of books I had written from XML documents. It has no user interface and is configured entirely with config files – to anyone but me, it's unusable, and it requires the files to adhere to a fairly lax, ad hoc standard. I could never give this software to anyone else because it makes so many assumptions about what I am going to need. It's a piece of software written with an amateur mindset.

Amateurs thus tend to focus on solving the problem, and the extra insight into programming that comes along with this is an extra benefit but not integral to the process. Professional development doesn't permit that luxury because you don't know who is going to be using your code (although you may have some shrewd ideas that can inform your development). There needs to be data validation, input handling, and ways to resolve ambiguity. Requirements can't simply be chopped away from a project without getting multiple people to sign off on changes. You can't simply expect people to work around bugs, or interpret misleading output. Most of all, you have to accept you are going to be rated and judged on the basis of the work you do – people are actually going to see the software you put in place! Additionally, because a professional knows this is going to be the case, code is usually written in such a way to minimize the problems of future development. All the while, the professional is looking for ways in which to improve on the process so as to make future development smoother or more effective. Professionals are thus focused on the process, and the problem itself is only a stepping stone to further clarity of understanding.

This is where the unique problem of Discworld comes into play – without the necessary experience in professional software development, everyone writes code the same way that an amateur does. The problem is, the code should be developed according to the mindset of a professional – we're hardly ever writing code for ourselves, the ultimate destination of our code is for it to be operational and experienced by players.

Growing into the mindset of professional development is something all creators do eventually, if they hang around for long enough. However, the time spent in that process of developing that mindset can be a source of tension between professionals and amateurs.

# Deep Smarts

More than anything else, the thing that separates an amateur developer from a professional developer is a thing known as 'deep smarts'. That doesn't mean that a professional is smarter than an amateur, it means that a lot of their insight comes from experience rather than pure natural ability, and that experience can only come with time. It takes time to build mastery in a subject – usually around ten years or so before the skilled practitioner really becomes a master practitioner. Professionals are distinguished by how far they are along this process.

An amateur may know more obscure coding trivia than a professional – they may have more up to date skills because they are conversant with a hip technology the professional hasn't had time to become familiar with. That's not what makes someone an expert in a topic.

In many ways, the building of deep smarts can be likened to a gradual internalizing of knowledge, where explicit knowledge (that which can be expressed) becomes internalized into tacit knowledge (that which cannot be expressed). People with deep smarts in their subject area can tell at a glance things that others may need to puzzle over. They understand the complexities and interactions of internal and external influences. They understand the possible alternatives and which is more appropriate for the task at hand. More than anything else, Deep Smarts is the ability to see patterns in jumbles of information.

The thing about Deep Smarts is that, although some of them are transferable (social skills in one arena tend to translate quite neatly into social skills in another), on the whole mastery of one subject does not confer mastery in another. There is no short-cut to building deep smarts, you just have to wait for time and experience to develop them, and in the meantime practise the skills whenever you get an opportunity.

Experts behave differently from novices – they can see problems before they occur, and can make decisions swiftly, even instantly, in a way that is far beyond the capability of a novice. They can instantly identify the context of a problem or a solution because of the bank of experiences they have built up, and can make fine distinctions between one situation and another that are unknowable to a novice. They know when usual rules don't apply. Most importantly, they know what they don't know, and what they need to know in order to provide a solution to a problem.

Deep smarts is the reason why the questions that a professional developer may ask you about problems you are having are entirely contrary to what you may have asked in the same situation. It's why a professional can identify that the real problem with your code is on line 355 of a different object, while the MUD is telling you the problem can be found on line 48 of your own.

I will reiterate that I am not saying all professional programmers have deep smarts (some do, some don't). All I am saying is that professionals, by virtue of earning a living on the basis of their software development, are further along in the process of developing deep smarts than any amateur can hope to be.

# The Tension

From the perspective of professionals, a lot of the tension comes from over-confidence on the part of an amateur. There's a world of difference from writing a complicated handler to writing a simple NPC – it's not just a matter of invested effort in code or understanding of the problem, it comes down to experience and ability to architect a complex solution to a complex problem. However, that's something that's obviously true only from the perspective of a professional because the amateur, by definition, doesn't have the breadth and depth of experience in building code to appreciate the gulf.

Sometimes people just don't appreciate how much of a framework the MUD provides for them in terms of supporting very complex functionality with comparatively little effort. Most of these frameworks of functionality can't be relied on for lower-level MUD code. This engenders a sense of 'Well, if I could pick it up without any previous experience at all, how hard can it really be?' attitude that can spark off conflict. Accusations of 'sloppy' or 'lazy' coding can especially rankle since they almost always stem from a position of relative inexperience. Of course, such accusations are rarely constructive even when they come from a position of relative experience either, but never mind.

Amateurs on the other hand have cause to resent professionals – I'm sure a lot of what I have been writing comes across as patronizing (although it isn't intentional), and professionals can be unfairly dismissive of the work of others. There can also be a rather unhelpful divide between 'coders' and 'builders', with the former tending to look down on the latter. This isn't an attitude that is especially pronounced on Discworld, but is endemic in some other MUDs. The professional who derides the work of more amateur coders is guilty of a sin far less justifiable than the lack of understanding demonstrated by an amateur.

Moreover, because of the entirely different mindsets (one focused on problem solving, one focused on process improvement), collaboration between the two groups can be an exercise in frustration. The professional is well placed to give guidance, but the amateur just wants an answer to their question – they don't want a lecture on process or to be told to re-engineer their approach.

A lot of the advice given by professionals tends to be abstract and of dubious immediate worth – it often centres on good practice, and ways of limiting problems in the future. Without the benefit of a comprehensive bank of experience, it's hard to see why this advice even matters – the only way you really understand the value of maintainability is when you have to completely re-engineer a piece of code because someone didn't care enough about your time to write it cleanly.

Part of the benefit that comes with a more professional mindset is to be able to work at a level that is appropriate for the problem – this is why I mentioned in an earlier chapter that premature optimization was a bad idea... it's easily possible to over-engineer a solution when a more modest approach is warranted. Shaving half a second off of the loading time of an object that gets created when the MUD starts and stays loaded until it shuts down is not a good investment in effort. Shaving a tenth of a second off of the time taken to execute a loop in the combat handler would be a valuable efficiency improvement. Experience helps identify which situation is which.

Amateur programmers especially have a tendency towards self-selection in programming tasks – essentially, people pick the tasks that are most interesting to code. These may not be the things that are most valuable to code. This kind of cherry-picking of duties is good in that it ensures a developer is motivated by the work they are doing, but more problematic in that it means the less fun stuff might not end up getting the same attention. This can frustrate professionals who then feel duty bound to pick up the slack.

It should also be pointed out that two professionals of different backgrounds are often a source of tension too... undoubtedly anyone familiar with the concept of 'dominance' in herds of animals can work out why!

All that is required to resolve these problems is for both sides to be willing to view things from the perspective of the other, and to be a little less critical on the efforts that others have invested. That's true for everyone actually, not just those on either side of this particular divide.

# Strategies for Success

There are actually great opportunities to ensure that people on both sides of this divide can work together in a way that increases the effectiveness of both, providing both sides are willing to yield a little. Pairing people together can achieve this, even if it's only an informal, ad-hoc pairing.

Professionals can provide invaluable aid to amateur coders by advising on architecture of code and adherence to quality standards. A professional can also greatly increase the capabilities of a novice by delegating sections of more complicated work and requesting it be written to some set criteria (usually the parameters, return type and functionality). Getting someone to participate in a more formal process like this can really underline the importance of the exercise when they see how easily their code can slot into a larger development. The professional gets a little extra developmental effort in exchange for a little of their expertise, and the amateur gets a little more knowledge in exchange for a little effort.

In order for a process like this to be effective, the professional has to be willing to actually be constructive – getting a piece of code back, sighing and saying 'Well, this needs rewritten' isn't going to give extra satisfaction for anyone. The professional also has to be prepared for the extra effort to be delayed gratification – adding a novice to a complicated project will slow things down to begin with before it speeds them up.

This kind of coaching effort can be hugely beneficial, but it has to be done in the right way. For one thing, both participants need to agree on the direction of the partnership – friction on something fundamental like why something is being done will frustrate future efforts to transfer knowledge.

Tacit knowledge, by its very nature, cannot be expressed. It's important then that a professional gives opportunities for an amateur to observe how something is done, as well as what is done. They should talk through their decision-making process with the amateur – they won't actually be able to identify every thought that went through their head, but they might be able to signpost some of them.

Knowledge building can be improved by finding some common ground between the two participants – expert knowledge can be transferred effectively with small vignettes or war stories. Stories convey a lot more information than instructions do, because by their very nature they are complex and highly nuanced.

A professional relating their experience through the use of relevant stories can be highly effective on two levels – on one, it gives the amateur a chance to parse their own meaning from the tale, and on two it helps induct the amateur into a wider community. Simply knowing certain stories is a badge of membership in many respects – there are stories about Discworld (or more correctly, about people who are a part of Discworld) that you simply won't know unless someone else considers you to be an insider rather than an outsider. Over fifteen years, Discworld has had its fair share of drama and gossip, and being 'in' on that gossip reflects your inclusion in a wider social context.

There is no need for a coaching process to be mandated from on high – the best such collaborations form spontaneously between people who genuinely like each other. However, within a domain it is worthwhile for a more experienced developer to keep an eye on younger creators to make sure that they are well supported and producing code that everyone can be proud of.

# Conclusion

I do apologise if reading this chapter made me sound like a pompous dick. My only excuse for that is that I actually am a pompous dick and it's sometimes hard to hide it. The fact that experience plays a big part in group dynamics should be uncontroversial though – the important thing is making sure that the friction caused by varied levels of experience is harnessed to a worthwhile aim.

Those with a more professional background in software development should do their best to respect the efforts of more amateur developers. Amateur developers should appreciate that there is a gulf of understanding between themselves and professionals. If both parties do this, relative harmony can be achieved.

Amateurs should take heart in the fact that they are in a tremendously good environment for learning more about software development as a process. We may not follow any sensible or formal strategies for developing our game (largely because it's hard to get volunteer developers to do anything they don't really want to do), but you'll at least understand why they are something to shoot for.

# Wrapping Up

## Introduction

So, that's our discussion of working with others, and the tools we have in place to support the process. The social aspect of Discworld is the thing that keeps most of us logging in, day after day, week after week, and year after year. There is a genuine satisfaction that comes from working together with other people to produce something of which everyone can be proud. However, it's also possible for people to clash, fight, cause problems, and generally degrade the atmosphere for everyone. Everyone is guilty of this at one time or another, and it can be invaluable in letting people blow off a little steam in tense times. We all have to work together to make sure that the problems are not systemic.

## Collegiality

You will hear a lot about the viciousness of creator politics, usually amongst misinformed players. I want to emphasise something I have said a number of times through this material – the creatorbase is, on the whole, a collegial body. We have disagreements, sometimes very strong disagreements, but the traditions that are at the heart of being a creator on Discworld are about openness and constructive engagement. What people misinterpret as politics are usually rooted in one or more parties not being able to integrate effectively into that environment.

Collegiality implies a common respect for the commitment that everyone has to the purpose of the organization. I would hope that we all agree on that central point – every creator is here to improve the game. We may disagree in how improvements are to come about, but we have to have faith that we have at least common desires in common.

We are also a very informal body – there is a hierarchy, to be sure, but membership of the higher ranks is based on fulfillment of a role. They are job titles, not aristocratic endowments (even in the past when directors were lords and trustees were high-lords, that was still the case). Domains are an administrative structure to ensure cohesion of purpose, not as a way to add restrictions to a body of volunteer developers.

Collegiality as a founding principle of our development system requires everyone adhere to it. Fostering that is a process in which everyone has a stake.

# Further Reading

While this material will give you the necessary grounding in our tools and philosophies, there is so much more to learn. Most of it applies to development in general – the fact that we are game developers does not invalidate what people have to say about constructive engagement with colleagues. The following are recommended reading for anyone who wants to delve a little more into the topic.

| Name | Author | Topic |
|---|---|---|
| The Psychology of Computer Programming | Gerald Weinberg | A fascinating look at the psychological aspects of computer programming. This is the source of the information on egoless programming. |
| Deep Smarts | Dorothy Leonard and Walter Swap | This book covers the development of Deep Smarts, which is what characterises novices from experts |
| Bowling Alone | Robert Putman | Not directly related to the topics under consideration, but a full and interesting discussion of the importance of social capital. |
| The Lucifer Effect | Phillip Zimbardo | A fascinating look into the power of social processes in closed systems. |

If you have any other suggestions for recommended reading, let me know so they can be included where appropriate!

# Conclusion

That's it for now, we're done here. Could you switch off the lights on your way out? Thank you, I appreciate your kindness in these trying times.

As a final note, you could really sum up the entirety of this material into the aphorism proposed by our friends at Penny Arcade in their range of reverential garments (available here if you think you could get away with wearing it in your daily life).