

amethodology

© Charles Weir, James Noble 2005

*I am the very model of a modern methodologist
Of OML and UML I am a trained apologist
Catalysis and Synthesis, analysis paralysis,
I am the very model of a modern methodologist*

*Every method every object and component I encapsulate
I think that architectural symmetry is really great
Everything I recommend has been endorsed by Billy Gates
My clients have a heart attack when'er I quote my daily rate.*

*I jump on every bandwagon as soon as it is fash-na-ble
I've written several books that are in no way understand-a-ble
I think of aggregation and of other things intang-i-ble
And all my best ideas are stolen by the folks at Rational.*

[Attributed to Charles Weir & James Noble, TOOLS Europe 2000]

Abstract

A pattern is a solution to a problem in a context. This paper uses the pattern form to address a basic problem in software development today: which methodology shall we pick to rule our project?

Introduction

You're sitting there, contemplating the start of a new project. You have (perhaps) a potential team of programmers, project managers, office cleaners, specification gurus, architects, testers, telephone sanitizers, configuration management specialists etc.

How are you going to coordinate them? How are they going to coordinate themselves? Even imagining the strictly personnel management side has been taken care of (or not) according to the whims and working of your company, the technical management is more difficult. You'll need an agreed way of working, of agreeing in advance what we're going to do, and of cementing technical contracts between developers – in other words, you need **a methodology**.

There are many Audi-driving 'consultants' who will gather around to help you help them meet their sales quotas. But how do you make the right decision?

Almost always, this decision gets ignored. Perhaps one particular methodology is all-pervasive in the developers' world. Perhaps they're all fired up and enthusiastic about the latest CV-enhancing opportunity a new approach will give them. Perhaps nobody's ever given the time to thinking about that at all, and therefore the approach has defaulted to the last common denominator [8]. And perhaps, just perhaps, whichever approach is used is in fact the best one for the job. But don't bet on it!

In this paper we're going to look at some various methodologies in common use in the current decade. This paper won't teach you how to use them, or even very much about what they do, but rather it will highlight features and strengths and weaknesses of each for different kinds of projects.

Scope of this Paper

Some days, it seems that methodologies are to software managers what meths is to alcoholics: poisonous, causing blindness, insanity, and death.

Some days, it seems that websites, newsgroups, wiki-webs, and mailing lists are dedicated to the art of not having a methodology - or rather, to having a methodology that pretends not to be a methodology.

Some days, its hard to tell where software development ends, and kindergarten (or, for that matter, drinking meths at the side of the road) begins: no big design up front, no analyses, bedtime stories instead of use cases, standing in a circle and hugging every morning, nerf guns, bouncy castles, free carbonated soft drinks...

On other days, you find yourself wondering if you are doing hard time in a maximum security prison: pages of documentation in triplicate before you make a change, methodology fascists controlling your every move, serried ranks of reviewers editing and annotating the lines ...

The intended reader of this paper is, well, ourselves the authors. Because, perhaps, we've been drinking meths on the side of the software development hypeway for quite some time now. We asked ourselves a question: how do you decide which methodology to pick? How do you choose? Does it matter? Is there really any ontological difference between extreme programming, extreme skateboarding, extreme ironing, and extreme irony? Is it better to have a metaclass or a metamodel? A user or a user story? How can you decide between using a brief case and briefing a use case? Between editing a diagram and diagramming an editor?

This paper, then, is a methylated intoxicated faltering stumbling attempt to try and make some kind of twisted sense of methodologies: in particular, when you should chose any particular methodology for your project. We don't try to give recipes for any particular kind of meths in this paper – instead, we'll just tell you enough so you know what we're talking about, and hopefully point you towards many “reassuringly expensive” descriptions of these methodologies.

Similarly, we haven't tried to be comprehensive in any way: rather we aim to cover the most popular, most reliable, most fun, most stupid, or most hyped methodologies. And we don't claim to be in any way objective. In order to talk about this at all, we need to talk frivolously. If people took this paper too seriously, we'd be in the middle of a holy war. People have literally bet their careers and their companies on there being one true methodology.

Pattern Form

Inasmuch as we have a pattern form, we ignore most of what you'd expect in the interests of humour. All these patterns express the same problem: how the ***** should I develop this program? Of course the forces are much the same for all of them: how long will it take, how many people we have, how much meths have they drunk recently... The patterns differ in how they resolve these – the things particular methodologies make better, or make worse. We also made up some examples, and pasted in the cheesiest clip art we could find.

We frankly admit that it is probably heresy to treat methodologies as patterns. But if pattern is “a solution to a problem in a context”, a methodology is at heart nothing but

a solution to the problem of how should you organise a software development project, in the context of the requirements, resources, and responsibilities with which the project must deal. Choosing which methodology to use can certainly be a pattern.

So, when you're stuck by the roadside, and pantechinological juggernauts are rolling by, and you're trying to work out if it really matters if you're doing extreme canoeing down a waterfall, or hacking together a Big Ball of Mud [8], we hope that these patterns may help you decide which circus mega-transporter to flag down, holding out a sign, and get another helping of methodology to help you on your merry way.

Forces

What concerns should you consider when choosing a methodology? Here's a checklist to get you started on the methodological road to oblivion. Let's start with the project content – what the project might, possibly, achieve:

Correctness: How important is it that the software you build will be fit for purpose?

Classically, correct software has implied that the software produced the correct outputs for all inputs; this also involves producing outputs within any timing constraints, and, more prosaically, not crashing. Software to run atomic power plants usually needs to be correct. Correctness is less important for software that will animate Furby soft toys or Microsoft's Barney.

Size: Size matters. Some projects are just bigger than others. Bigger does not necessarily mean harder or more complex or how scalable your solution must be. Just that there's more of it. Bigger projects often take more coordination between programmers.

Long-term existence: How long is the software you're building expected to last?

Some software (say a major operating system) can have an expected service life of twenty, thirty, or forty years. Other software is written one day to be used the next and thrown away the day after. So also, are you contractually required to document your software? And if so, must your documentation really describe the software you've built?

Degree of difficulty of domain: Some domains, such as financial market modelling and astrophysics, are intrinsically complex. There may be good reference materials or expertise available to your team that describe these domains: that that doesn't alter the fact that these things are intrinsically complicated.

Next there are the constraints on the actual project work itself. How are you ever going to deliver anything if you've got all the following to think about...?

Time to Market: How soon must you market the software you've produced? This is generally related to the speed of development, or at least to the amount of time that is available.

Cost: How much can you spend on developing the software? A major driver of software cost is the number of developers you require: although numbers of testers, managers, or tea-ladies can also dramatically increase the cost of a project.

Programmers' Temperament: Even though most programmers score INTJ on Myers-Briggs' personality tests [1], you may have the misfortune to have an ESFP

consultant-to-be Charlie on your programming team. More seriously, different programmers have different temperaments, different preferred working habits and different attitudes to comments, documentation, testing, and Britney Spears. And a major consequence of that is their attitude to discipline and working with others - some may be shoehorned into RUP; others will rebel against anything but Mission Impossible on a solo hot keyboard.

And finally – and often the most important in meths choice – is the nature of your interaction with the customer and the outside world driving the project...

Opacity of requirements: How difficult it is to find out what the requirements or the domain actually are? Many organisation support systems for large companies (universities, hospitals, enterprise purchasing etc) face this problem: Nobody knows the rules that the software must enforce. The resulting system may be technically simple (a relational database with a web front end, all on an intranet with no hard transaction time requirements). But even if the domain modelled by the system is simple, finding out what that underlying model should be may be extremely difficult.

Fluidity of requirements: Some domains entail requirements that change daily. Some other domains may have requirements that change yearly. Other projects have effectively one attempt at building a system and no time for the requirements to change.

Technical Constraints: Many projects are done as changes to existing code. Such project can have very tight technical constraints: the new work must fit seamlessly with the old, or, at a minimum fit into a pre-existing framework. There are still “green field” projects, with no constraints on the solution technology. This force interrogates the degree of freedom implementing a solution. What you can change? What are other systems you’re working with?

Commercial Constraints: Projects never exist in isolation: every project will be shaped by the commercial and organisational environment that sustains it. These environments – especially contracts where more than one organisation is involved – can place very strong constraints on the methodology chosen. Some contracts, for example can mandate a particular methodology (“thou shalt use RUP!”). Other contract or business planning disciplines can impose a methodology by default – requiring a fixed-price bid or large amounts of documentation, for example.

Project Cancellation: There is only one thing certain in software development: sooner or later your project will be cancelled. Perhaps it is cancelled when you expected it to finish. Perhaps beforehand, because you finished before the deadline and under budget – yeah right! Perhaps it is cancelled long after you hoped it would be, after staggering around like a zombie for years. All good things come to an end. So do all software projects. This force considers how likely and how soon your project will be cancelled, and what kind of mess will be left afterwards.

The Patterns

The patterns in this paper are as shown below. We've shown how they might be positioned on a variety of scales – the level of programmer discipline involved, and the level of planning discipline required.

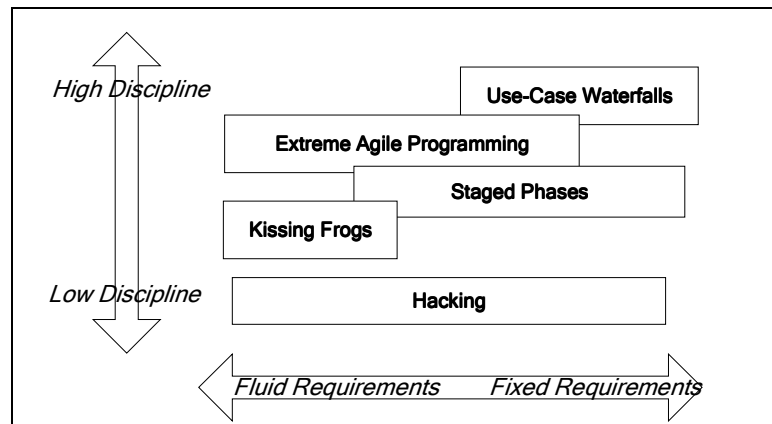


Figure 1: Programmer temperament vs. Requirement Fluidity

Or, we can look at how the meths handle different kinds of projects:

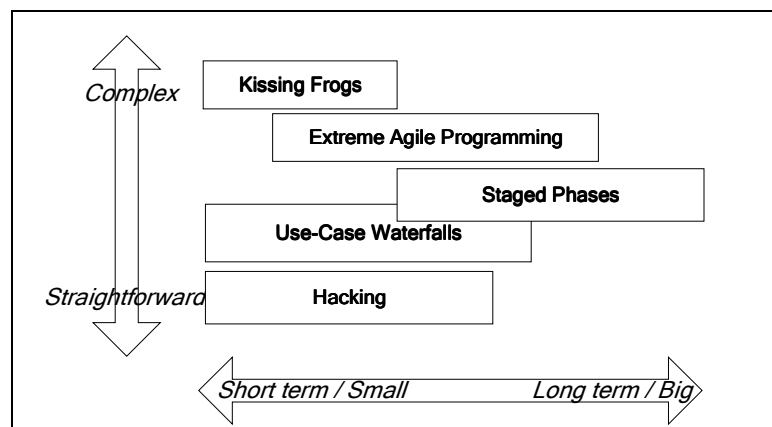
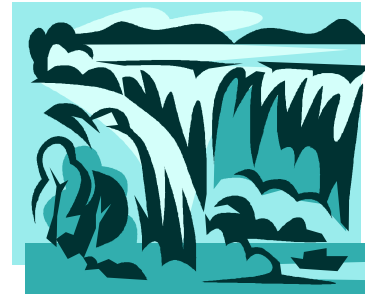


Figure 2: Different kinds of Project

You won't find the forces itemised by name in the patterns, but we think you'll catch on to which ones are involved. Where we don't mention a force in a pattern, it's probably because the meths in question doesn't do much for or against it.

Pattern: Use-Case Waterfalls



Context: You have to develop some piece of software.

Example: *ConsultUs Corporation is bidding on a government contract to provide a new system for supporting the Department of Administrative Affairs. As per the government process, after the Department had prepared a business case, they issued a Request for Proposals (RFP) to which ConsultUs replied; then the Department issued an initial contract worth approximately 5% of the overall cost to the lowest-bidding contractor (in this case, ConsultUs). After successfully completing the initial contract, ConsultUs was awarded a contract to complete the whole system.*

Problem: How the ***** should you develop the program?

- You have a boring system to develop (i.e. a system in a well-understood domain). There's a lot of detail, and you'll need to be disciplined to keep track of it all.
- Your client insists on a fixed price contract, or at least rigorous contracting, or governance processes between themselves and the development team.
- Your project will take more than one month to do – perhaps it is quite large and will take years...
- ... Or somebody will get seriously sued if it goes wrong and they can't prove they've taken reasonable care.
- You just hired a methodology manager with an expensive suit

Therefore: Use a Use-Case-Waterfall

In spite of twenty or thirty years of pontification in the software engineering community about the wonders of agile iterative increments, most projects are still done in one-pass waterfalls.

The life cycle of a typical Use-Case Waterfall project is:

- Provide an initial quotation based on a Request for Proposal (RFP).
- First catch your use cases, and get customer sign off on them and on a revised estimate.
- Produce a nicely bound requirements document.
- Do design – possibly ignoring the requirements – writing UML class diagrams, and sequence diagrams for every important use case.
- Produce a nicely bound design document.
- Write the code – often ignoring the design document.
- Document a few test cases based on the use-cases.
- Produce a nicely bound test plan document.
- Sell the software.
- Rinse and repeat.

RUP™ is often done this way, but, technically, as a meta-methodology it can in fact be instantiated to cover **any** of the patterns in this paper [3], [4]. That's what being a customisable grand unified methylated framework means. ☺

More to the point, most object-oriented development methodologies usually end up being implemented in this waterfall style.

Consequences:

- Managers are happy because this is a clear process.
- Clients are happy because this is a clear process.
- Boring programmers are happy because this is a clear process.
- Even non-boring programmers will be familiar with this process...
- ... Perhaps because it's well documented in textbooks, and easy to teach. It's much easier to describe than one of those fancy Agile things.
- It's easy to plan, though the plan usually turns out to be wrong.
- It's relatively easy to organise larger teams of programmers – typically by subdividing the work required based on the design documentation.
- It's easy to quote and organise payment (typically do an initial quote to the RFP, and a firm quote after the use-cases are done).
- And you probably won't get fired if it fails!

However:

- All the other programmers are unhappy because they spend most of the time writing documentation they will ignore
- It's hard to change or add requirements in mid-stream, and you often end up with complex change-control mechanisms, because changes either mean the developers work for free, or mean the client has to agree to pay more.
- Maintenance isn't as easy as expected, because the software doesn't match the documentation.
- Traditionally, your project is not cancelled, but the customer refuses to pay the last 20% because "it's just what we asked for, but it isn't what we wanted!"

Example Resolved: *ConsultUs uses RUP exclusively to manage all their development projects. A group of 5 ConsultUs consultants used the inception stage of RUP to respond to the RFP, then used the initial 5% contract to build a first pass at a use case model and complete a detailed project plan and baseline budget. Once the main contract was let, ConsultUs completed the analysis model, revised the plan, and then hired in about fifteen extra staff — ten developers, four testers and a tech writer. Eight months after the contract was let ConsultUs had completed the core of the project, however change requests from the DAA took another three months elapsed time to iron out all the difficulties. The project was formally handed over to the Department's operations staff about fifteen months after the project began — four months late and 7% over the baseline budget (but within the contingency of 10% included in the*

project plan). Both the Department and ConsultUs considered the project a success: and ConsultUS has been invited to submit a sole tender for a new project that is about one and a half times the size of this project.

Pattern: Staged Phases

Context: You have to develop some piece of software.

Example: *The Government of a small pacific island nation has hired the Worldwide Commercial Computer Contracting Corporation (WC4) to implement a system to support a new Universal Benefit provision recently passed into law. The Universal Benefit will replace many separate benefits (unemployment, health, disability, stupidity, even the pension) with a single centralised benefit. This project covers hardware selection, software design and construction, training, installation, and operation for a least a year before handover. It is expected to take between three and five years to complete, at a cost many millions of dollars.*



Problem: How the ***** should you develop the program?

- You have a very large system to develop.
- The project is big and boring: the system is a well-understood domain, and there do not appear to be too many technical risks, and
- You have experience in building similar systems before, and
- Your client insists on a rigorous contracting and governance process between themselves and the development team... but you can't really face the risk of the lawsuits if the whole thing goes wrong at the end.
- ... Or the project requirements seem to get into the realms of fantasy towards the end, and you don't want to risk losing the whole project because of later insanities.

Therefore: Use Staged Phased Process.

Rather than attempting to construct the whole system in one go, split up the system in to stages (or phases) that can be worked on one after another.

- Depending on the project, each phase might last for a year or more, a few months (probably best), or even a few weeks. You'll have to figure it out for yourself.
- Based on your previous experience in working projects, you divide up the requirements for the whole project into a set of phases. The client should do it; probably you'll end up doing it for them and presenting it as their idea.
- Then implement each phase in turn; each leads to a complete release.
- Of course phase 2 always changes completely before you finish phase 1, and as for phases 3, and 4... but at least something gets delivered, and
- You get feedback, pay, and the client takes ownership of the results of each phase.

You can even use different meths for the different phases. Perhaps you'll be bored of being Extremely Agile after a year of it, and resort to Hacking or Waterfalling for the next phase. Perhaps the new meths might even be more suitable. Perhaps.

Consequences:

- You have a strategy for attacking such large projects
- Clients are happy because they get something delivered within the typical two years of a single employment – so the people who initiated the project get credit for it!
- It's easy to plan, and who cares that the plan usually turns out to be wrong?
- It helps with organising larger teams of programmers – the phases force everyone to get things to work together at least once a year.
- It's easy to quote and organise payment (typically do an initial quote to the RFP, and a quote for each phase).
- And you probably won't get fired if it fails!

However:

- If requirements change a lot during the course of the project, in particular those requirements that are implemented in early passes, then it gets costly and embarrassing (unless you're *Agile...*).
- Phases give the client an excuse for bottling out without giving you the full project work to do. Or give the work to someone else.
- The testing for the end of each phase can be expensive – and boring.

Example Resolved: *W4C decides to take a phased approach to this problem. First they performed an in-depth analysis of the existing systems and of the new legislative requirement, To complete even this stage, Worldwide Commercial had to establish an office of thirty senior analysts and ten partners (and approximately fifty locally recruited support staff) in the capital city of the Small Pacific Island Nation. Setting up the office and finding staff took six months, getting to a basic plan took another year.*

Luckily, the conversion of unemployment beneficiaries to the universal benefit was identified early in this process, as the rules for the two benefits were quite close and, politically, few people who voted for the incumbent government would mind if the unemployment benefit payments were interrupted. In conjunction with the government, Worldwide therefore implemented and deployed a test version of the unemployment portion of the system a little over two and a half years after beginning the project.

Flushed with this success (and with an eye to its most important support bloc) the government pushed Worldwide into implementing the pensions system next, although, being much more complicated, it was originally planned for the last phase of the project. Unfortunately Worldwide was not so lucky this time: after concentrating many of its resources to meet the deadline, when the resulting system went live two years later (six weeks before the next election) it spectacularly failed to pay anyone anything. Worldwide staff worked around the clock to reinstate the previous system, but the damage was done: based on this failure, and the revelation that to meet the deadline, Worldwide had outsourced much of the development to a workforce of

Alabama prison labourers, indentured orphans from the Bosnian civil war, and sweatshops for new immigrants in Darwin — the government lost the election.

In the ensuing inquiry, Worldwide lost the contract. The next government wrote off the pensions subsystem, then hired Worldwide's greatest competitor, Global Electric Data Solutions, to restart the project from the second stage of implementing health benefits.

Pattern: Extreme Agile Programming

Context: You have to develop some piece of software

Example: *HappiTeam Incorporated have been engaged to develop an internal expense tracking system for a large Chicago corporate. The project is not well scoped — the contract is open ended, e.g. time and materials rather than a fixed price bid — because the large corporate was unable to finalise project requirements*



A senior accountant at the corporate, Jill Moneybags, is a “Sunday afternoon programmer” (she started on the ZX81 as a precocious teenager) is quite knowledgeable about the requirements for expense tracking: indeed, auditing expenses is a major part of her job, and that is why she advocated hiring HappiTeam to work on the project.

Problem: How the ***** should you develop the program?

- You expect the project’s requirements to be fluid.
- On the other hand, it’ll be straightforward at any given time to find the current requirements.
- You (or your client) has a good candidate for the customer role: someone who understands both the domain, the business requirements (and reality) of the project, has the respect of the programmers and doesn’t mind working eighty hour weeks [7].
- Your commercial contracting environment permits agile or time-and-materials contracts.
- Your project is smallish — probably requiring a single team of less than ten people working for less than a year – but might be complex in concept.
- Your client is willing to take the risk on a nonstandard project (or doesn’t know or care).

Therefore: Use an eXtreme Lean Agile SCRUM

Wave the flag! Drink the Cool-Aid! Join the party and enjoy the fun! Stick to the principles [2], [5], [6]:

- Make frequent releases, of products that claim to be of ‘release quality’. (Don’t worry too much about release testing – that’s the customer’s job).
- Keep refactoring – tomorrow’s code should look nothing like today’s.
- Listen carefully to the customer, see what she wants, and tell her up front how long each bit will take. Expect the customer to change her mind and probably the entire project at least twice a month.
- Up-front documentation – who needs it? Why make design decisions at all? Paste in the documentation afterwards if it’s wanted.

- Share the code – if everyone owns it then nobody can trace my bugs back to me. Maybe even work in pairs, groups, scrums etc.
- ... Then rinse and repeat, until the money runs out or the customer gets fed up and cancels it all.

Consequences:

- You get to feel really cool, on the edge, eXtreme, agile. HOOAH!!
- ... Even though you are still overweight from sitting in a dark hole with desks made out of gaffer tape [9]
- Your programmers feel cool too: doing agile development is good for their CV, so they can bail out of the project when it fails and get another job.
- Your customer ends up with something rather different at the end from what they anticipated at the beginning. It's probably better too.
- And Charles liked this process so much that he bet the company on it [12].

However:

- Your customer gets to do 80-hour weeks, since they must direct the project on a day-to-day basis instead of just handing over a specification and critiquing the result. The programmers get to work 40-hours.
- Programmers can kid themselves they have lots of “American freedom”™, though in practice they must subsume their ego into the groupthink.
- It's difficult to scale this up to more than one team, or to more than ten or so programmers.
- Unless you are very disciplined, this can easily devolve into *Hacking*
- Your project generally gets cancelled earlier than you were expecting, because the customer's got what they want and doesn't want to pay for more.

Example Resolved: *HappiTeam chose to use Extreme Programming on this project, with Jill Moneybags as the customer.*

The project started well, with a metaphor session deciding that expense accounting was like giving blood donations, however after while things got bogged down as Jill needed to spend more time on her actual job rather than managing the project.

Because Jill had some influence over the corporate's CEO, the first release of the project has now been placed into production and HappiTeam are working their 37.5 hour weeks on the second release of the software.

Pattern: Hacking



Context: You have to develop some piece of software

Example: *When Gazza's Dinkum Tyres (a Australian nationwide tyre repair franchise) upgraded their CRM system to run on Windows XP, their automatic customer allocation system (which had been running fine under Windows 3.1) failed catastrophically. They hired InfoDevelop, a local software development shop, to rebuild the functionality for the new environment.*

Problem: How the ***** should you develop the program?

- You know exactly what to do. Either your problem is trivially small, or your team is very experienced. There's not too much complexity to keep track of.
- You have a small team, just one or two people
- Alternatively, your team hates one another, are extremely introverted, or do not speak the same language
- ... Or half your team is in Bosnia, the other half in Serbia,
- ... Or half your team is in Israel, the other half in Palestine
- ... Or all your team is in Belgium.
- Your client doesn't understand software development

Therefore: Hack it!

"Hacking: the art of making furniture with an axe" - ES Raymond [10].

Rather than thinking, planning, organizing, discussing, prototyping or gaming your development, hacking dispenses with everything except writing the code. As the saying goes: "just hack it!" Write the code, get it to me yesterday, that's fine, there's the cash, under the table, Bob's your uncle, Britney's down the rubitty-dub, knoworahrramean [20]?

Hacking seems most effective in one of two situations. The first is where the problem to be solved is very simple, well within the capacity of one or two people working for a day or less to solve. Why bother with documentation, stand-up meetings, testing, stories, design, acceptance, planning, analysis, customers or requirements when the overhead imposed by any *kind* of process approaches the cost of programming the whole thing up from scratch. Why pay the costs of documentation or commenting or carefully eliminating waste when you could have thrown the current version away and rebuilt it before you'd scheduled your lean process improvement value chain scrum?

The second situation — paradoxically — is where the problem to be solved is highly technically idiosyncratic. Perhaps you have to extend a legacy system where there are no manuals, no source code, and all the programmers who once worked out the system are now retired or have died. Perhaps you have to work with a modern system whose API is as orthogonal as a porcupine. From this perspective, hacking is highly optimised process for learning about technology: the code you write are experiments

with the system, which may be falsified at any time so there is no point in investing to heavily in your experimental apparatus. Once you're done, however, you just leave the code as it is, with none of that refactoring stuff you have to do if you're being Agile, and none of the documentation you'd have to do if you were RUPping.

Consequences:

- Your programmers are relaxed, because they can do whatever they like.
- Your programmers like you because they can turn up and get paid without having to produce much
- You spend all your effort on development; and don't waste any time on peripheral activities that don't develop customer value (like analysis, design, testing, or documentation)
- This works really well if your programmers hate each other: each of them can work on their own part of the problem with minimal discussion and interaction, and somehow integrate their disparate chunks at the end...
- ... Or perhaps not, but being such superhero programmers, they'll easily find another gig.
- Methodologies and disciplines are for people who are afraid of programming. We're so tough we can break Tonka toys.
- Even hacking is a discipline — in this case, the discipline is to write code. Some people find writing documentation, attending stand up meetings, pairing with people who can program, or choosing website colours better than writing code: hacking is not for them

However:

- You need programmers who are “Happy Hackers” and who don't end up spending all their time writing UML, documentation, comments, tests, and so on, or at least whinging about low code quality.
- It can all easily go horribly wrong, with nobody knowing how all the bits go together, poor quality deliveries, and total uncertainty about delivery timescales.
- The end result is often difficult to maintain, and you have to throw it all away and do it again.
- Or you have to throw it away and do it again because the result is a festering pile of half-working *****.
- Or you can never decide whether to throw it away because it is 90% complete, needs just another week to finish (and has been that way for the last five years...)

Example Resolved: *InfoDevelop allocated one of its most experienced developers, Mark Griswold, to the job, under a contract to analyse and design a new customer allocation system for Windows XP. While visiting a client site, Mark managed to start a debugger on one of their machines, and then determine that the reason the old system would not run was that it relied on direct BIOS calls to protected mode kernel routines that Windows XP no longer supported. Mark took a copy of the system home, and over the weekend, he managed to patch the binary (the source code was long since gone) to call a couple of stub routines he had handcrafted in assembler. InfoDevelop and Gazza's Dinkum Tyres renegotiated what had been a six-month analysis/design/build contract so that Gazza's Types kept using the hacked system (delivered within a week) while paying out half the contract, making a nice profit for InfoDevelop (and Mark) but saving Gazza a bundle too.*

Pattern: Kissing Frogs



Context: You have to develop some piece of software

Example: *Weird-O-Tron, a Mobile Paradigm Entrepreneurship Innovator based in Santa Monica, California, have been hired by a large telephone company to “Imagineer” the next generation conceptual category killer applications for emerging cell phone operating systems ring tone management systems.*

Since Weird-O-Tron have the programming skills of a pickled herring, they’ve hired DieGeekHaus, a specialised mobile development company to actually build the “emerging paradigm concept”.

Problem: How the ***** should you develop the program?

- You no idea what to do.
- Perhaps you have completely opaque requirements. Perhaps there are no requirements to speak of. Perhaps (implicitly) part of the aim of the project is to find the requirements – or even to find out if the project itself is worthwhile
- Or perhaps you have clear requirements (e.g. a web site) but a very wide implementation space — either on apparently inessential aspects (what colour should the web page background be) or, quite possibly, significant technical decisions (should you use VB, C#, Perl or play-dough?) and you don’t really know where to start.
- You have a client wearing black t-shirts, black jeans and glasses with chunky frames.
- You have happy-go-lucky programmers.
- You are trying to change the world.

Therefore;

Kiss Frogs™. Means you develop lots of prototypes, and keep hacking. Dean Kamen is the inventor of (amongst other things) the stair-climbing wheelchair and the Segway Happy Horizontal People Transporter [11]. He develops a huge sequence of prototypes, starting from rough hacks but working up to designs that are used to analyze the manufacturing process.

Similarly, many web shops and other design-based disciplines work in the same way: develop a number of nonfunctional or partly functional prototypes (in HTML, Photoshop, Flash, Director). Eventually one of the prototypes can be chosen and that is then re-implemented often using quite a different technology. Multiple prototypes may be developed in parallel or in sequence, but prototypes never turn into final products.

This differs from Agile XP in that it is iterative, but not incremental; it develops (barely functional) prototypes for significant parts of the system, and then throws them all away. But not before picking the one you liked best. Then you go into a “product development phase” (perhaps doing a ‘*Use-Case Waterfall*’, more likely just rebuilding somehow more seriously) to build the final product. Traditionally, the project is cancelled halfway through this second phase.

Consequences:

- It’s good for exploring lots of weird stuff.
- The client can just decided on what’s cool, what’s very “today”, without giving any rationalizations for it or even having to think too hard.
- You projects can get cancelled early, when its obvious none of the projects are anywhere near what’s wanted – or more likely, when its realized that what is wanted is actually useless rubbish.

However:

- Doing the same thing ten times in different shades of pink can get really frustrating for programmers really fast
- Why do all that coding and throw it all away? Why not just pick the first one? (If you like that idea, you should be doing *Hacking*).

Example Resolved: *Weird-O-Tron had no idea what they wanted or needed, so they arranged to have GeekHaus build a series of prototypes. Each month, the imaginers and programmers would get together, the programmers would show their designs, Weird-O-Tron would say how great they were — and then ask GeekHaus to build something totally different.*

After a while, Weird-O-Tron began to appreciate the limitations of the laws of physics — or the amount of storage available on the phones, while GeekHaus learnt the importance of coordinated user interface colour design. Although the ring-tone management system (iRings) was not accepted into the operating system, the automatic dog whistler application GeekHaus eventually developed to Weird-O-Tron’s specifications was one of the year’s top selling applications in the Hammer-Schlepper SkyMall catalogue, available in the cheap seats of all major airlines worldwide.

Conclusion

We've considered five of the more common methodologies:

Use-case Waterfalls – grinds through the process tracking the detail to nail down a solid development.

Staged Phases — makes large projects possible through a series of large increments.

Extreme Agile Programming – leaps nimbly and iteratively, through a series of disciplined small increments..

Kissing Frogs – makes fast and lightweight trials to validate ideas.

Hacking – is fast and allows independent programming, but tends to be uncoordinated.

We've used all of them; maybe you have too. And they all work in their place.

They can be used together, of course. Here's a table of the possible combinations:

	Waterfall	Phases	Agile	Kissing Frogs
Phases	Works well			
Agile	No!	Works well		
Kissing Frogs	K.F works first	No!	K.F. works first	
Hacking	No!	Works well	No!	Works well.

The combinations that fail are those that require opposing disciplines – so, for example, Agile and Waterfall have strong and incompatible disciplines.

What next?

You can try out any combination of these – or none at all. For no methodology will work for you if you don't actually do it. Even *Hacking* won't work if your lead programmers are all born-again UML-diagram documenters. Somewhere you'll need discipline, and team buy-in to get it all to work.

Good luck!

Acknowledgements

Our thanks to our EuroPLoP shepherd, Joe Bergin, to the group who workshopped this paper at EuroPLoP 2005, and to all the people who invented the 'ologies.

References

1. Keirsey and Bates, Please Understand Me, Prometheus Nemesis 1984
2. Alistair Cockburn, Agile Software Development, Addison-Wesley, 2001.
3. Philippe Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley 2003.
4. Per Kroll, et al, The Rational Unified Process Made Easy, Addison Wesley 2003
5. Kent Beck, Extreme Programming Explained: Embracing Change, 2004
6. Mike Beedle, Ken Schwaber, Agile Software Development with Scrum, Prentice Hall 2002
7. Weir, Noble, Martin, Biddle, My Friend the Customer, EuroPLoP proceedings 2004?
8. Brian Foote and Joseph Yoder. Big Ball of Mud. Pattern Languages of Program Design 4. Addison-Wesley, 2000.
9. Dick Gabriel. Patterns of Software: Tales from the Software Community. OUP 1998
10. Eric S Raymond. The New Hacker's Dictionary. MIT Press 1996.
11. Dean Kamen, quoted in Bruce Mau, Massive Change, Phaidon 2004.
12. Penrillian Website <http://www.penrillian.com>.
13. Cockney Rhyming Slang,
http://en.wikipedia.org/wiki/Cockney_rhyming_slang