

DPI

De Programmatica *Ipsium*

DE PROGRAMMATICA IPSUM

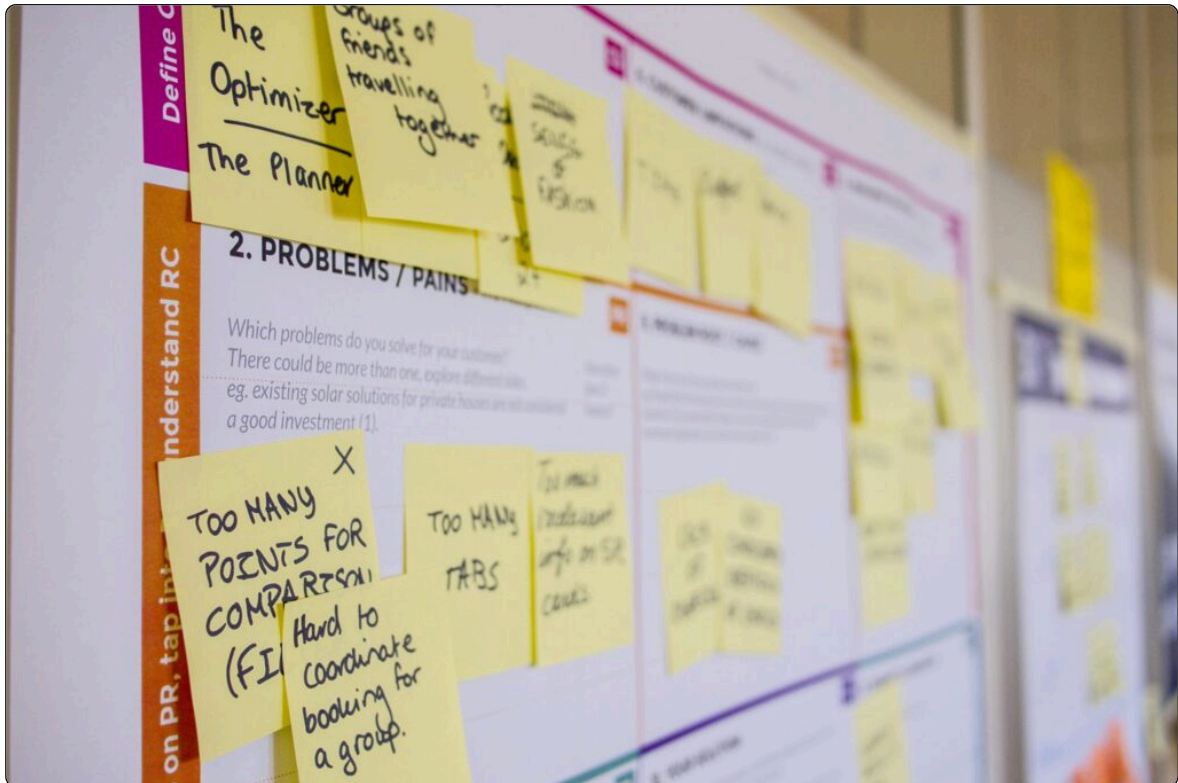
Issue 039:
Methodology

December 6th, 2021

Table of Contents

Issue 039: Methodology	5
Methodologies: The Next Two Decades	9
You Are Doing It Wrong	17
The Three Amigos, Among Others	23

Issue 039: Methodology



December 6th, 2021

Welcome to the thirty-ninth issue of *De Programmatica Ipsum*, dedicated to the subject of *Methodology*. In this edition:

- Graham forecasts what the next two decades might bring¹ in terms of methodologies.
- Adrian warns of the danger of methodologies becoming all-or-nothing dogmas².
- In the Library section³, Graham reviews the collective work of “The Three Amigos”⁴: Grady Booch, Jim Rumbaugh, and Ivar Jacobson.

ISSUE 039: METHODOLOGY

Enjoy this issue! Please subscribe to our free newsletter⁵ to stay updated about new releases, share the articles on social media, or contribute⁶ if you would like to support our work.

Cover photo by Daria Nepriakhina⁷ on Unsplash⁸.

REFERENCES

- ¹ <https://deprogrammaticaipsum.com/methodologies-the-next-two-decades/>
- ² <https://deprogrammaticaipsum.com/you-are-doing-it-wrong/>
- ³ <https://deprogrammaticaipsum.com/category/library/>
- ⁴ <https://deprogrammaticaipsum.com/the-three-amigos-among-others/>
- ⁵ <https://deprogrammaticaipsum.com/newsletter/>
- ⁶ <https://deprogrammaticaipsum.com/contribute/>
- ⁷ https://unsplash.com/@epicantus?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
- ⁸ https://unsplash.com/s/photos/method?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Methodologies: The Next Two Decades



By Graham Lee

Let me start by getting something off my chest that has been annoying me this whole millennium. Methodology is the study of method. The process you use to develop software, and the practices embedded in that process, constitute a method. The understanding of how that process results in working software, in customer value, in arguments between business analysts and QAs, and in the modification of that process to yield different outcomes, that is a methodology.

The people who came together at the Snowbird conference to discuss their lightweight methods were, by and large, methodologists, in that they had developed those methods and explored the commonalities, differences, and impacts of those methods. The people who subsequently read a book called “succeeding with Agile” and bought Jira licenses were not methodologists. They were applicers of methods.

Scrum actually embodies both method and methodology, in that it is a self-referencing method that includes process improvement as part of the process. This has caused a problem for many organisations who implement it, because they employ process-followers rather than process-definers, but try to follow a process that has them defining the process. If they are lucky, they ignore the process-improvement aspects of Scrum and follow the baseline process forever. If they are unlucky, they try to improve on the baseline.

Anyway. That is out of my system now, let us move on. There have not been any advances, either in software development method from the methodologists, or in software development methodology, for the last couple of decades. In the beginning, there was “let us run this like we run our hardware projects and our manufacturing procurement projects”. That was very quickly (surprisingly quickly, if you only read the headlines) seen not to work. By the 1980s people were recommending shifting risk left (i.e. exposing it earlier), by constructing working prototypes or “walking skeletons” early in the development process and sharing those with customers. By iteratively and incrementally delivering the project to customers. By periodically reflecting on whether following the plan was still the best outcome for the project, or whether the plan needed to change.

Like bankruptcy, Agile happened at first slowly, then all at once. This shifting left of risk and responding to change became so important to the outcomes of projects that people started to realise that they should basically reorganise their methods around correcting for what needed to change. Big, detailed design documents were deferred, or replaced entirely with lightweight sketches or even metaphors relating the planned system with some other system. Lengthy analyses of what users might do with the hypothetical systems got replaced with summaries of conversations about what users might want from the system, or at least reminders to have those conversations.

The publication of the Agile manifesto came neither at the beginning nor the end of this process, but in the middle. It reflected two things: one that methodologists across the industry realised they were all pointing in roughly the same direction by promoting these lightweight methods, and wanted to increase their impact by collectively sharing the commonalities. The other that they did not want to be called “lightweight methodologists” because that might make them sound trivial.

The process that is going on now is the wholesale adoption of Agile and lightweight methods across the industry, accelerated by the sharing of the manifesto but already in progress before it and still in progress two decades later. There has been no methodological novelty since then. You might point to DevOps, DevSecOps, DevSecTestQABAProjProdThreeBagsFullOps, or whatever the latest name is, but that is just the same story as Agile told again for people who did not hear it in 2001. Shift risks left. Work out what your goals are, and let the professionals collaborate on achieving those goals. Stop planning for getting things wrong, and slavishly following that plan.

We do not have methodology any more, we have method. We have people kind of following agile, and people kind of helping other people to kind of understand how to kind of follow agile. Technological changes (we can now write and run proofs of software in real time; we can create and deploy software without needing to know how it works using ML; we can update software multiple times a minute) have not changed the processes or the process of constructing processes.

In fact, if anything, dark scrum shows that there has been a regression in methodology. We have gone back to focusing on the processes and tools, with managers looking at Jira dashboards to understand “sprint velocity” rather than how the individuals are interacting to produce the software. They look at the comprehensive (project) documentation (in Jira), enforce contract negotiation (via sprint commitments), and follow a plan (via infinite backlogs).

This shows two things. One that the methodological advance that was Agile is not yet evenly distributed, even among those who are of the opinion that they are on board. The second that there is still scope for further methodological advance, because the explaining power of Agile is not sufficiently compelling to work for a

significant majority of software organisations or sufficiently effective to point them toward successful methods.

The benefits of Agile are significantly clearer to someone who lived through the turmoil of its ascendance (one of the benefits of being a developer after 40) than they are to someone who has grown up in the code mines of dark scrum. Software is *way* less riskier now: most agile software projects never fail. They may well get “sunset” when they do not achieve revenue or growth goals, but they do so much more quickly and they deliver more value to more people than the typical failed software project did in the 1990s. The business and the technical folks are much more aligned on goals, and are invited to collaborate on ensuring that the next thing that happens is the most beneficial of the options available.

To some extent, anyway. Scrum itself is surprisingly Leninist in its promotion of democratic centralism¹. Everybody negotiates on what the sprint goals are until they get “locked in” and then everybody agrees to follow the plan instead of responding to change. In theory, this stops factional squabbles, except that the biggest drawback of Agile is the baked-in factionalism.

That drawback is the central contradiction of Agile: it promotes unity of purpose and self-organisation while also dividing participants into two camps. On the one hand, you have the Bizshevik party, who own the money and the purpose of the organisation. On the other hand, the Techshevik minority, who merely produce the value.

You will notice that there are way more Certified Scrum Masters than Certified Scrum Developers, because the Bizsheviks ensure that the power is concentrated in their party. Business people and developers must work together daily throughout the project, but that does not mean the Techsheviks get to have a say in how it is run. The sponsors, developers, and users should be able to maintain a constant pace indefinitely: as long as the sponsors are making money from the users, the developers get to work on making more.

Where is the next step in methodology? We need a methodology that creates methods focusing on what *the people using (or otherwise exposed to) the software want*, distinct from *what the customer wants*. Agile focuses on the interaction between

“the business” and “the technical people”, and that is overly limiting. Of course, the business are putting up the resources to make this software happen, so they get some say in the software that gets made. But they should not get a say in *how* it is made, and they should not be the final arbiters in how it is used either.

We need to get past the product owner as the proxy for what “the user” wants, when they actually represent what the shareholder wants. We need to divorce the question of whether the software is successful from the question of whether the software engineers look busy. We need to get out of the situation where the people accepting the software get to decide how much refactoring or testing we do.

In other words, we need to continue the focus on professional autonomy that agile started: we need to proceed towards anarchic software development.

This clearly cannot be done within the confines of in-house software teams. No, the conflict of interest between doing good software, and demonstrating shareholder value, gets in the way. Good software is sacrificed for feature factories, perpetual busy-working, and building what some in-house “customer representative” thinks people want.

To do good software, the software doers have to be solely responsible for the software that is done. That means acting as agents, and telling the client this is what we do, and this is how it is done, take it or leave it. It means doing work so exemplary, and so consistently exemplary, that the obvious choice is to take it.

And this means that we have to take ownership of the relationship with the people who so far have been called the “user” or “customer”, but should really be thought of as “victims”. We need to take control of that relationship until we can turn the victims into beneficiaries.

The next generation on methodologies will define methods that do not make “technical people” the servants of “the business”. The primary measure of progress will not be working software that satisfies the needs of “the customer”. It will be the mutual satisfaction of the paymasters, the beneficiaries, and the engineers: such satisfaction achieved through the instrument of working software where necessary. The technical people and the business will be working together to provide something that works for them and for the software’s beneficiaries.

Of course, to succeed, such methods need to be so productive that those who are doing it the old way are clearly doing it the old way. Actually, no. This idea that everybody needs to adopt the new thing or it is a failure, that every new innovation needs to succeed at global scale, is a fallacy. All that is needed for the new methods to succeed is that some people successfully use them to make themselves and other people happy. The rest can happen slowly, and then all at once.

Cover photo by [engin akyurt²](#) on [Unsplash³](#).

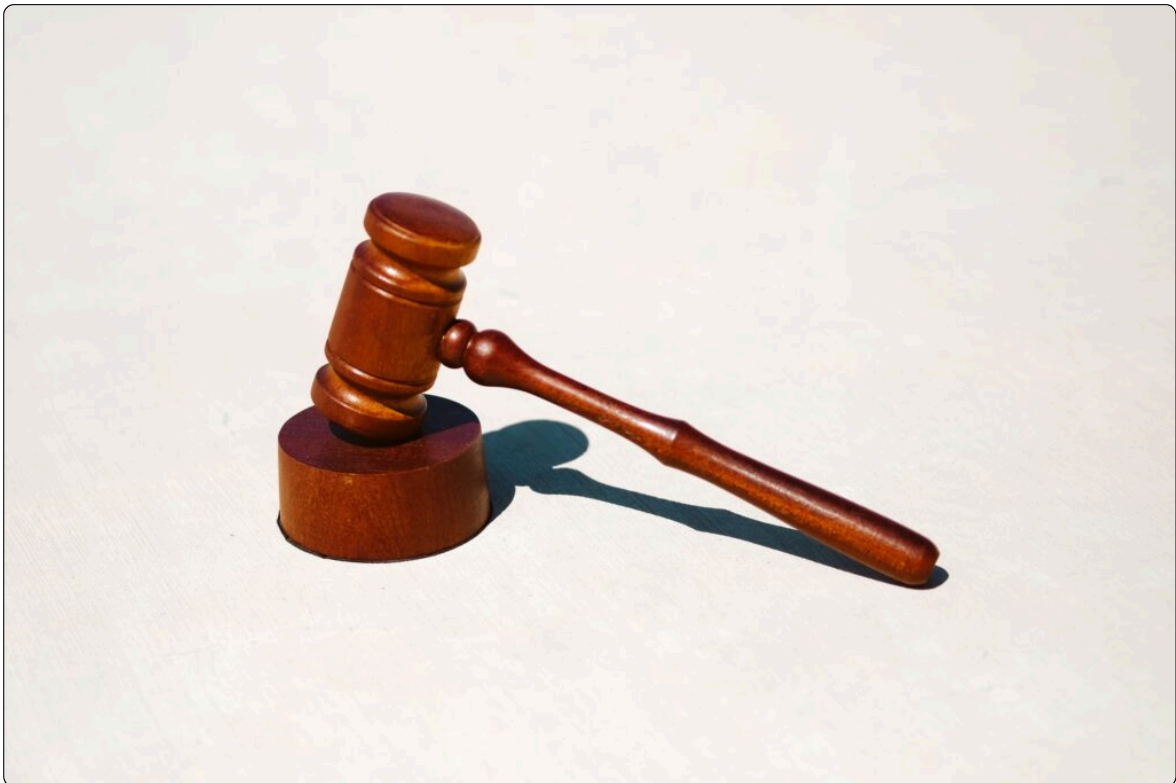
REFERENCES

¹ <https://www.marxists.org/archive/lenin/works/1921/10thcong/ch04.htm>

² https://unsplash.com/@enginakyurt?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

³ https://unsplash.com/s/photos/black-flag?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

You Are Doing It Wrong



By Adrian Kosmaczewski

On March 13, 1995 in Paris, during a press gathering held at a conference called *Le cinéma vers son deuxième siècle*, Danish filmmakers Lars von Trier¹ and Thomas Vinterberg² introduced a new artistic movement called Dogme 95³. In the world of arts, this declaration is comparable to the Agile Manifesto⁴ in many ways; not only are they both contemporary, they are also a reaction to the current state of things in each of their respective industries.

In the case of Dogme 95, it was a reaction against the “Hollywoodisation” of movies; ignominiously big budgets, “make it or break it” in the first weekend, the dictatorship of the box-office, the star system, and the consequent loss of depth and relevance of the movies produced.

The Dogme 95 manifesto contained a set of rules for movies to follow:

1. Movies should be shot in location, not in studios, and without extra props.
2. No music nor sound postproduction is allowed.
3. Only handheld cameras must be used.
4. No black and white movies; only color, and no extra lightning is allowed on set.
5. No optical effects allowed during postproduction.
6. No gratuitous action allowed: no murders, no firearms, etc.
7. Films happen here and now; not in the past, not in the future.
8. No genre movies allowed (western, science fiction, etc).
9. Films must be shot using 35 mm film.
10. The name of the director must not appear in the final credits.

The “Dogme 95 Collective”, also known as “Dogme Brethren”, that gathered around these ideas, produced 31 films⁵ following these guidelines between 1998 and 2005.

Interestingly, most of the films in the list⁶ violated one or many of the points above. Each filmmaker, following their own artistic license, deviated from those guidelines (or were they rules?) as they saw fit. Some Dogme 95 directors went as far as publishing a “confession” of the deviations they took from the manifesto, in order to finish their films, like in the case of “Mifunes sidste sang” (1999)⁷ by Søren Kragh-Jacobsen.

Were they doing Dogme 95 wrong? Hardly.

Art does not serve any purpose per se. Hence, it exists for its own sake, and thus, it is never wrong. It can be shocking, it can generate emotions, or it can be simply *beautiful*, without any other objective. Hence, any deviation from the Dogme 95 rules simply meant... not being included into the Dogme 95 movie list. Nothing else. Those movies could still receive awards, be shown on festivals, or even released as special BluRay boxed editions for Christmas.

But coding is very often considered an art⁸. Why is it then, that an Agile software-making organization has to face that eternal Damocles Sword engraved with the Latin words *Nefas facis*?

This is one of the most disturbing phrases one team could ever hear: “You Are Doing Agile Wrong.” “This is not how you hold retrospectives.” “This is not a proper Kanban board.” “What would Kent Beck⁹ say of those tests.” And so on, and so forth.

In doing so, consultants and disgruntled developers join the ranks of Dogma. Not “Dogme” (see the “e” at the end of the word) but Dogma. A God-like contraption that holds unlimited power of life and death over the minds and actions of anyone involved in a programming team. A way to instill guilt, not helping in actually figuring out the deeper structural and communication problems that a team might be suffering from.

The idea of Dogma is that, by following it, there is a guarantee of success, order, and outcome. Of course, early successes makes followers become fanatical; and on top of that, rigid hierarchical social structures in corporations block the dynamic alignment of skills towards the resolution of problems, thereby preventing all deviation from Dogma.

Yet, in a mostly ironical turn of events, the Dogma of Agile advocates precisely for that: *the dynamic alignment of skills towards the resolution of problems*. That is, in essence, Agile.

Which means that the Dogma of Agile is, in essence, to avoid Dogmas. Yet, instead of remembering that, we have The Scaled Agile Framework® (SAFe®). Yes, there is a real trademark sign at the end of the name.

The root problem of SAFe is hypocrisy¹⁰: SAFe works as a way to hide hierarchy behind the pretense of Agile. In essence, Agile was a cry against hypocrisy. So, instead of SAFe, if companies want to implement hierarchies for their software processes, they should just do it, and enjoy all of its caveats and advantages. Software produced under such conditions might be very rigid as per Conway’s Law¹¹, but it is totally possible for it to gain wide market acceptance, and to generate an interesting ROI; as the French say, *personne n’est à l’abri du succès*.

But please, do not even mention the word Agile; doing so risks turning the Dogma of Agile into nothing else and nothing more than a cargo cult¹².

History shows that hierarchical structures can be wildly successful when they embrace their hierarchy. So, who knows? Maybe a more rigid structure can enforce documentation production, stricter Q&A processes, better programming language choice guidelines, more architectural or design meetings, enforced code reviews, and more. Drop the crazy standup meeting¹³, YAGNI. Maybe I am an older developer, but I could totally understand why companies would like to work like this.

Of course, due responsibility for failures should be acknowledged at the right levels, which in itself would be politically dangerous for career-minded managers, and this could be the demise of the structure itself. It is hard to cope with the hubris of a few well-placed individuals in a structure.

When a methodology (any methodology) becomes an objective per se, instead of a way to organize the production of working software, all hope is lost, and Agile is no more. Instead, it is the time of the Dogma of Agile.

Quoting Edsger Dijkstra:

The first effect of teaching a methodology-rather than disseminating knowledge-is that of enhancing the capacities of the already capable, thus magnifying the difference in intelligence. In a society in which the educational system is used as an instrument for the establishment of a homogenized culture, in which the cream is prevented from rising to the top, the education of competent programmers could be politically unpalatable.

(Dijkstra, Edsger W. 1972. “The Humble Programmer.” Communications of the ACM 15 (10): 859–66. <https://doi.org/10.1145/355604.361591>¹⁴.)

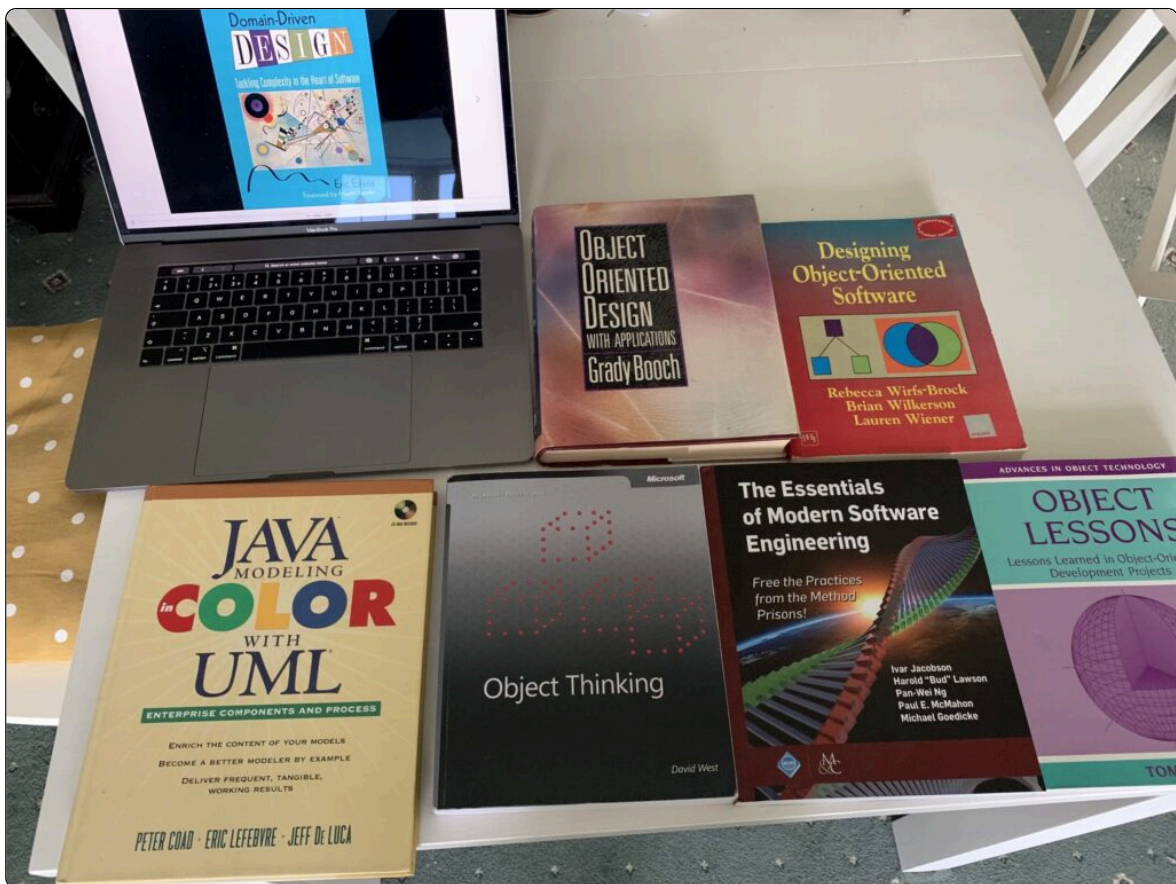
Dijkstra’s words apply not only to country-sized political systems, but also to most social structures, from corporations to the smallest of teams therein.

Cover photo by Tingey Injury Law Firm¹⁵ on Unsplash¹⁶.

REFERENCES

- ¹ https://en.wikipedia.org/wiki/Lars_von_Trier
- ² https://en.wikipedia.org/wiki/Thomas_Vinterberg
- ³ https://en.wikipedia.org/wiki/Dogme_95
- ⁴ <https://agilemanifesto.org/>
- ⁵ <http://www.dogme95.dk/dogme-films/>
- ⁶ <http://www.dogme95.dk/dogme-films/>
- ⁷ [https://en.wikipedia.org/wiki/Mifune_\(film\)#Confession](https://en.wikipedia.org/wiki/Mifune_(film)#Confession)
- ⁸ <https://deprogrammaticaipsum.com/a-brief-history-of-programming-artists/>
- ⁹ <https://deprogrammaticaipsum.com/kent-beck/>
- ¹⁰ <https://seandexter1.medium.com/beware-safe-the-scaled-agile-framework-for-enterprise-an-unholy-incarnation-of-darkness-bf6819f6943f>
- ¹¹ https://en.wikipedia.org/wiki/Conway's_law
- ¹² https://en.wikipedia.org/wiki/Cargo_cult
- ¹³ <https://akos.ma/blog/the-various-styles-of-standup-meetings/>
- ¹⁴ <https://doi.org/10.1145/355604.361591>
- ¹⁵ https://unsplash.com/@tingeyinjurylawfirm?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
- ¹⁶ https://unsplash.com/s/photos/wrong?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The Three Amigos, Among Others



By Graham Lee

This month, the methodology issue, is a good opportunity to take a look across a whole swathe of my bookshelf and deal with it all at once. The important point to bear in mind is that methodology is about the approach you take to building software. That means how you decide what to build, plan it out, design it, implement it, test it, deploy it, document it, and manage (and even pay for) all of that.

The 1990s was the decade when object-oriented techniques hit the mainstream: it was also the time of the methodology wars.

Agile programming is more than user stories on index cards, in the same way that Object-Oriented Programming is more than a language that lets you write `self.foo()` instead of `foo()`. In fact, the two are very closely related: Extreme Programming rose out of experiences running projects that used Smalltalk, and many of the Agile Alliance are OOP experts. Much was written about OOP as a methodology, but now when people critique it they are mostly talking about their experiences with a programming tool. Let us recover that understanding of how to build software by thinking in objects.

In 1991, Grady Booch published *Object Oriented Design: with Applications*. “With applications” because five chapters describe the design and implementation of software using his method, in five different object-oriented languages. By applying object-oriented design, Booch said, “we achieve a greater level of confidence in the correctness of our software through an intelligent separation of its state space. Ultimately, we reduce the risks of developing complex software systems.” Sounds like he is saying it makes it easier to reason about code¹.

Booch’s method is an iterative, incremental approach to designing software that fits best into a similarly iterative, incremental methodology. It involves identifying the classes and objects at a given level of abstraction, identifying their semantics, identifying the relationships between them, and implementing them. It is iterative in that the design informs the analysis and implementation, the implementation informs the analysis and design, and the analysis informs the design and implementation. It is also iterative in that the analysis, design, and implementation at one level of abstraction influences the work performed at the levels above and below. It is incremental in that there are many ways in which you can deliver the designed and implemented parts of a system without having to deliver the whole system.

A chapter on “pragmatics” links object-oriented design to the methodology in deeper ways. Object-oriented design benefits from object-oriented programming, though you can of course do it in any language. Object-oriented programming benefits from object-oriented tools, though you can of course do it with Notepad

and a machine monitor. It also benefits from object-oriented staffing, an object-oriented approach to release management, testing, and all the other things.

And it has drawbacks. Booch is clear about the risks of adopting OOD, particularly in 1991. There were performance problems associated with dynamic method resolution that could have significant drawbacks in deployment (but on the other hand, dynamic method resolution made development much quicker, by letting you modify your program live and only rebuild the modified parts). That is no longer true: the same advances in processing power that enable fancy type systems also make it fairly quick to run message dispatch systems. And those themselves have got faster, through theoretical advances in virtual machines and garbage collection.

But one of the biggest drawbacks was that programmers needed to change mindset. Booch recommended a three-step training scheme:

- *Provide formal training to both developers and managers in the elements of the object model.*
- *Use object-oriented design in a low-risk project first, and allow the team to make mistakes; use these team members to seed other projects and act as mentors for the object-oriented approach.*
- *Expose the developers and managers to examples of well-structured object-oriented systems.*

Has this even happened now? I have had a total of two weeks of formal training in the object model (involving UML, which we will come to later): the first by Sun Microsystems, in 2008, and the second by the University of Oxford, in 2016. I have an impression that this is two weeks of formal training more than many developers receive in *any* approach to software design (though let us not pretend that useful software does not get written despite this).

Booch's method is very similar to many other contemporary methods of object-oriented design. In *Designing Object-Oriented Software*, Rebecca Wirfs-Brock and coauthors describe a comparable design method with comparable benefits and drawbacks. "The object-oriented approach attempts to manage the complexity inherent in real-world problems by abstracting out knowledge, and encapsulating

it within *objects*. Finding or creating these objects is a problem of structuring knowledge and activities.”

Some of the details are different from the Booch method, but the broad strokes are the same. When Booch identified classes, they got entered into a collection of diagrams expressing static and dynamic characteristics of the classes and objects. When Wirfs-Brock identified classes, they got written up on index cards based on Ward Cunningham and Kent Beck’s Class-Responsibility-Collaborator cards². Slightly different diagrams identified relationships between classes.

Alongside all of this, there was a growing realisation that significant benefits in object-oriented processes and their adoption could be realised by combining these “competing” methods, integrating the nuanced differences in each and providing a standard representation that benefited from network effects. This process had started in 1989, when multiple vendors formed the Object Management Group consortium to define a vendor-neutral approach to distributed objects.

When it comes to object-oriented design, people realised that as the field matured, engineers needed a core set of principles, not a specific prescription of activities³. The OMG adopted a technique that combined the methods of the Three Amigos: Grady Booch, Jim Rumbaugh (Object Modeling Technique⁴), and Ivar Jacobson (Object-Oriented Software Engineering⁵). OMT, OOSE, and the Booch method came together to form the Unified Modeling Language, UML. The UML tells you *what you need to know* to design object-oriented software, but does not tell you the exact process to acquire that knowledge. Methodologists can still promote their methodologies, IDE vendors can still sell their IDEs, and now everybody understands what everybody else is talking about.

Of course, the story does not stop there. People have extended the ideas listed above, either to address missing gaps or to stake a claim on virgin territory in the object-oriented frontier. David West’s Object Thinking⁶—which reveals that the object-oriented approach is a philosophy not a design tool—expands the CRC card to a six-sided object cube. Side one is the CRC card. Side two describes the nature of the objects. Side three lists the method contracts (particularly visibility: public, protected, private). Side four describes the information needed by the objects, where they get that knowledge, and where they keep it. Side five lists all of the

messages and method signatures. Side six describes events that the object will need to listen out for.

Another attempt to stick extra dimensions into the object-oriented design language came from the authors of *Java Modeling in Color with UML*⁷. This adds four colours to the UML, conveniently (and tellingly, when we think about the longevity of the diagrams created) the four pastel colours typically found in a block of sticky notes.

The colours represent *archetypes*, top-level abstractions that imply broad characteristics of behaviour. These archetypes are:

Moment-interval (pink)

Instances or durations in time that are important in the problem, and thus in the software.

Role (yellow)

A way in which an actors participates in the system.

Description (blue)

A collection of attributes and values that are consistently used in relation to a given object or context.

Party, Place or Thing (green)

A real-world person, location, or object (sorry) represented as a software object.

In addition to providing modeling tools beyond the basic UML, the *Java Modeling in Color* book additionally gifts us a whole new methodology called Feature-Driven Development. You would probably recognise many of the aspects of FDD, even if you would not bundle them under that name. Two-week iterations, focused on client-valued features. Design and build by feature. Replace detailed use case documentation with single-sentence feature and feature set descriptions: though these take a different form from user stories, focusing on domain model concepts rather than stakeholder intents and actions. A feature set may be called “Making a product sale to a customer”, within which one feature is “Calculate the total of a sale”.

But you may not recognise other characteristics of FDD. Develop an overall model and features list before you start (while your engineering department may not know this, your marketing, sales, operations, and executives have actually already done this, and they call it the roadmap). Organise a feature iteration around a chief programmer⁸, but organise the implementation expertise around class owners—reinforcing the point that object-oriented programming is a staffing issue not just a programming language trick. No long-running teams; constitute new feature teams every iteration.

Rebecca Wirfs-Brock suggests that while the object design methodologies of the early 1990s were helpful, they were overly simplistic. By overly relying on identifying concepts in the problem domain to reify in the software, they missed out on important techniques of abstraction⁹ that come from looking for concepts that are *implied* or *absent* in the problem domain. She points to Domain-Driven Design¹⁰ by Eric Evans as a better, more flexible way to incorporate these ideas and drive a software design. It represents a constructivist philosophy, in which the problem's structure arises from the ways people communicate about it, rather than the previous positivist philosophy in which the problem is taken as “real” and its properties measurable by experiment.

Ivar Jacobson, one of the Three Amigos of Object-Oriented methodology, went as far as to abandon methodology entirely, as in his book *The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!*¹¹ Jacobson and co-authors say that methodologies are fundamentally flawed, as they are rarely empirically justified and replacing one methodology with another typically jettisons much that is good in addition to much that is bad. They prefer a systems thinking approach, where what we currently do creates software with certain characteristics (maybe late, buggy, and well-documented) and we consider how changes to practices affect the process and characteristics (maybe cutting back on documentation improves lateness, makes bugginess worse, and reduces documentation quality but within acceptable limits).

So, what do we learn from the methodological explosion and continuing evolution of object-oriented design? I will turn the last word to Tom Love, co-founder of object-oriented tools company Stepstone (who guided Objective-C through

the early years of its development). In his 1993 book *Object Lessons: Lessons Learned in Object-Oriented Software Development Projects*¹², Love looks forward to *Software Development in 2002* (a decade after writing, not publication!). He argues that groupware and computer-supported collaborative working would be commonplace by then, “we should find real-time, full-motion video-conferencing facilities in every sizable development facility in five years”. Turns out this actually needed a pandemic to happen, and took way longer.

He predicts that NeXT will need to find a new way to sell their software to succeed at driving object adoption—and indeed they did. He predicts that “Microsoft NT++”, an evolution of their workgroup software, would remain important—and indeed it has. He predicts that there will be greater recognition that software development is a team activity, and of course this has come to pass too. But he also predicts that the microcomputer of the future would be running a variant of OS/2, and that we would have recognised the power of the spreadsheet: the way to solve the software crisis is to make everybody a programmer. Today’s computers have fewer, not more, facilities for user empowerment.

Cover photo by the author.

REFERENCES

- ¹ <https://deprogrammaticaipsum.com/how-to-reason-about-mutable-state/>
- ² <http://worrydream.com/refs/Beck%20-%20%20A%20Laboratory%20For%20Teaching%20Object-Oriented%20Thinking.pdf>
- ³ <https://dl.acm.org/doi/10.1145/260028.260114>
- ⁴ <https://archive.org/details/objectorientedmo00rumb>
- ⁵ https://openlibrary.org/books/OL1718405M/Object-oriented_software_engineering
- ⁶ <http://davewest.us/product/object-thinking/>
- ⁷ https://en.wikipedia.org/wiki/Object_Modeling_in_Color
- ⁸ <https://www.sicpers.info/2014/08/the-wealth-of-applications/>
- ⁹ <http://wirfs-brock.com/blog/2021/09/13/design-and-reality/>
- ¹⁰ https://www.dddcommunity.org/book/evans_2003/
- ¹¹ <https://dl.acm.org/doi/book/10.1145/3277669>
- ¹² https://openlibrary.org/books/OL22372248M/Object_lessons