INFO-TECH WHITE PAPER

Software Methodologies: Battle of the Gurus

No software methodology appeared out of thin air -- each has a long pedigree. Read this white paper to find out how the top development methodologies evolved and learn their best practices.

info-tech research group

401 Clarence Street, 2nd Floor, London, Ontario, Canada N6A 3M6
Tel: 519-432-3550 Toll Free: 1-888-670-8889 Fax: 519-432-2506 www.technologynews.net

Table of Contents

1. Introduction

It is fitting that this white paper is being written a few weeks after Professor Edsger Dijkstra passed on. In March, 1968 Dr. Dijkstra published an influential letter to the editor of the *Communications of the Association for Computing Machinery* (CACM) called *Go To Statement Considered Harmful*. The impact of this letter was analogous in some ways to Martin Luther's posting of his *95 Theses* on the church door in Wittenberg. Like the latter, Dijkstra's letter inaugurated a religious war which continues to this day.

I don't mean to be either sacrilegious or flippant by this statement. While software development may not be as important an activity as looking after one's soul, it does have enormous impact on everyone's material life — but not just. Every time I board a plane, which is quite often, I think to myself how much my safety depends on some programmer getting it right. This is not a very comforting thought to anyone who is even slightly involved in this industry.

How we develop software matters: doing it properly reduces the risk of economic failure, of personal failure in one's career (if you are in the field), and, most importantly, for some systems it reduces the risk of lives being lost.

This paper is about methods of developing software. The first section provides an historical overview. It is included since the emotional distance that time provides allows us to look at current issues in a more detached way. Moreover, knowing the past is also a defense against the hype of the present. It amazes me how often I read articles in the trade press which promise salvation for software developers, while in reality all they deliver is a repackaging of old ideas that have long been discredited. Knowing the history of the field will help you avoid the pitfall of buying into this hype. Finally, there are several key ideas developed over the past 35 years that are still relevant.

The second section provides a general framework — useful approaches and tools that form a universal software development methodology. We focus on the *4M* method of software development. We also discuss the *Unified Modeling Language* (UML) which provides a standard communication tool — a notation for specifying software systems at many levels.

The third section provides information about software development methods that are currently popular. The emphasis is on *agile* software development methods — more on what that means later.

Finally, we conclude with some useful and practical tips for IT managers. I'm the kind of person who always skips to the end of a book before I read it, so I'll give the ending away up front. Software development is not an industrial process. Rather it is a human-centric activity. Success is determined almost exclusively by how well you manage human resources. The software methods that work are mostly about managing people and ***not*** about sophisticated coding techniques.

2. Historical Background

2.1 A Discipline of Programming

2.1.1 Unstructured Programming

We need to go back a bit in time in order to understand what was bothering Dijkstra and why his paper had such a huge impact. Most younger people in the industry forget (or never knew) that software "programming" began by flipping switches on the computer's console. The use of "high-level" languages only began in the fifties (Fortran and Cobol being the most popular of these). The mere fact that these existed was considered a great achievement. Programmers didn't think very much about programming "style." Given the limited size and speed of early computers, the big issue facing programmers was how to write code that was small and efficient. Compilers often weren't very good, so programmers prided themselves on knowing tricks on how to fool the compiler into generating the best code possible.

By 1968, however, mini-computers were already popular. In fact, that year Data General Corp., started by a group of engineers that had left Digital Equipment Corp., introduced the Nova, with a whopping 32 kilobytes of memory, for the incredibly low price of $8,000. Software developers began to understand that Moore's Law — "The density of chips doubles every year" — which he had first stated in 1964, was really true. This meant that as time went on and computers got bigger and faster, a program's size and speed were not the main criteria for measuring its effectiveness. IBM's popular System/360 and the spread of high-level languages meant that programs were durable and portable over time. The ever lower cost of hardware, which Moore's law guaranteed, meant that for the first time the cost of software development could exceed the cost of the hardware on which it ran.

Hence, a new set of criteria for measuring the success of software development came to the fore. These criteria are still relevant today. A project is deemed successful if the code produced:

- Has a relatively low cost of initial development.
- Is easily maintained.
- Is portable to new hardware.
- Does the job the customer wants.

While high-level languages were very popular by 1968, there were no set of rules to guide programmers on how to write code that met these criteria. In fact, programmers still focused on writing code that was fast and small mainly because it seemed like a whole lot more fun to do (having fun still is the main criteria used by programmers to judge the value of their work). And then along came Dijkstra.

2.1.2 Structured Programming

Dijkstra was an academician who saw computing science as a branch of applied mathematics. Computer programmers, like other engineers, must apply formal mathematical methods to create effective programs. Dijkstra maintained that the "go to" construct in high-level programming languages was an abomination. It led to incorrect programs. Dijkstra's ideas formed the basis of "Structured Programming," which was really the first software development methodology. Basically, structured programming advocated:

- Developing programs top-down (as opposed to bottom-up).
- Using a set of specific formal programming constructs (the "go to" was to be banished).
- Following some formal steps to decompose the larger problem.

Following this methodology, proponents argued, would ensure programmers met the criteria of successful software enumerated above.

In 1971, Professor Nicklaus Wirth released the programming language Pascal that did not provide a "go to" statement and had control structures to support the Dijkstra's structured programming paradigm. Subsequently, almost all programming languages have been influenced by Dijkstra's and Wirth's ideas of creating well structured and readable code.

By now you are asking yourself, "but what about the religion bit? " Ah, those were heady times. Every programmer who had any sense of pride in his or her craft (including me) ran out to buy Dijkstra's book. Few (and I was not among them) actually read it. Nonetheless, with the fervent belief of new converts, we all believed: "Pascal Good, Fortran Bad." The pages of the *CACM* were covered with religious debates about the "go to". Sects and counter-sects were formed. A milestone in this debate came in 1974 when Donald Knuth, who pointedly called his *magnum opus, The **Art** of Computer Programming*, wrote *Structured programming with go to statements* in the ACM's *Computing Surveys.* (In researching this paper, I even found citations as late as 1996 still debating the issue!). Looking back, it is hard to believe how much anger and heat the debate generated. It truly was a religious war. Fortunately programmers tended to attack each other with words not guns.

2.1.3 Structured Design and Analysis

But Dijkstra's influence did not stop with great "go to" debate. Structured programming led to the idea of structured design and structured analysis. In fact, a whole new discipline was born: software engineering. Software development was seen as an industrial process. Methodologies which gave detailed rules and regulations for the process were developed. Dijkstra retreated back into the world of academia, but the mantle of religious war was taken up by people like Edward Yourdon, Peter Coad, James Martin, and Tom DeMarco (remember these names because these guys are still around today). Each of them – sometimes alone, sometimes in pairs – set up denominations (also known as consulting groups). Bibles were published, sermons were written and declaimed (at great expense to the listener). They all promised that if only those lazy, good-for-nothing programmers would follow ***their*** discipline (as opposed to that of their charlatan competitors'), then software development would be far less expensive, code would be easy to maintain and port, and customers would be thrilled and delighted at the results.

In hindsight it is hard to believe that we bought into this. (Well maybe not. As noted earlier, the same ideas are being repackaged and sold today. But I am getting ahead of the story.) However, the plunging cost of hardware placed software in the spotlight. Managers were desperate to bring the unwieldy process of software development under control. Customers were despairing over the huge sums spent and the meager results of software development projects. *Something* had to be done. The snake oil of software development methodologies seemed mighty tempting.

Formal Methods

Dijkstra's book, *A Discipline of Programming*, is not about the "lay" concept of structured programming. Dijkstra argues that programs are mathematical constructs and should be subject to proof. He proudly noted that none of the programs in the book were actually run on a machine (a contention many books at the time noted to show that they were error free). Constructing programs as proofs (as formal methodologists demand) is not likely to happen in our lifetime. A brief anecdote: Many years ago I was a manager on the development of the early Statemate(TM) system. Statemate implemented a graphical mathematical language called Statecharts. The idea is that users specify a reactive system using Statecharts. Statemate then executes the specification — essentially executing a formal mathematical proof of the specification. I shared a room with the programmer who developed the system executions. I know, first hand, that the programmer did not apply formal methods in developing his software. So obviously the results of the Statemate simulations were only as good as his (non-formal) code. To contend that these results are formal proofs is disingenuous.

That is not to say formal methods aren't useful. And this is not a criticism of Statecharts or Statemate. Statemate is an excellent tool for prototyping reactive systems. It just illustrates the "chicken and egg" flaw of formal methodologies as an approach to software development. Please note that computer science is not detached from the real world of software development. It does yield quite useful and practical methods for implementing sub-classes of programming problems. Dijkstra's main contribution to the field was in the area of distributed systems. Algorithms he developed are widely used in classes of problems he clearly defined. Most of the really useful methods computer scientists developed are found in Knuth's *Art of Computer Programming* (if he ever finishes it).

2.2. The Mythical Man-Month

In 1975, when the first Software Engineering methodology war was at its height, Fred Brooks published his famous book. Unlike the academic Dijkstra and his consultant followers, Brooks learned his lessons about developing software when he was the manager of probably the first large-scale software development project ever undertaken — the development of IBM's Operating System/360 (OS/360) in the early 1960's. The core of Brooks' message is that software development is a human-centric process, not an engineering discipline. His book lays out the first useful software development methodology – useful because it stresses methods for managing the people process, and not the engineering process. In this section, we lay out Brooks' analysis of the problems of software development. We discuss his methodology in the concept section below since it forms the core of any useful software development methodology. Unlike Dijkstra's, this is a book I have read and referred to many times over the years I managed software projects. Every software developer, and certainly any software manager, *must* read this book.

2.2.1 Key Issues of Software Development Projects

Brooks, in a humorous and insightful style, identifies the many problems software projects face. Here are just some of the points Brooks raises. Anyone who has ever been involved in software development will find them equally relevant today.

1. **The Tar Pit:** "Software projects are perhaps the most intricate and complex (in terms of the number of distinct kind of parts) of the things humanity makes." This is more true today, by several orders of magnitude. "The tar pit of software engineering will continue to be sticky for some time to come."

2. **The Mythical Man-Month:** "More projects have gone awry for lack of calendar time than for all other reasons combined." This opening sentence of Brooks' book is also still true. No one has yet developed a good way to estimate how long programming projects take. Programming is a creative process, like art and music. Programming projects are commercial endeavors. Therein lies the core of the problem: how do we successfully manage a human, creative process (as opposed to a mechanical, productive process).

3. **Brooks' Law (which he now claims has been substantiated by subsequent research):** "Adding manpower to a late software project makes it later." People and time are not interchangeable. There are certain processes that can't be hurried along. Adding more people increases inter-communication and training overhead, as well as disrupts progress.

4. **The Second System Effect:** "The second is the most dangerous system a person ever designs; the general tendency is to over-design it." In modern parlance we would call this "featuritis".

5. **Why Did the Tower of Babel Fail?** "Schedule disaster, functional misfit, and system bugs all arise because the left hand does not know what the right hand is doing. Teams drift apart in assumptions." Communication is the most difficult part of any human endeavor. This, of course, is the core of the mythical man-month.

6. **One Step Forward and One Step Back**
"System entropy rises over a lifetime." This is also known as "bit rot." Repairs tend to destroy structure and increase disorder.

2.2.2 The Impact of M M-M

The book sold well and consistently, and is always widely quoted. Besides raising the problems, the book offers some important solutions that, when taken together, offer a universal method for software development (we discuss this more in depth in section 3). However, *The Mythical Man-Month*, unlike structured programming, did not form the basis of any guru-promoted methodology in the 70s or 80s. Perhaps this was because Brooks moved on to a career in academia, and did not set himself up as guru with a consultancy. While the book was of incredible importance, it was not until the late '90s that Brooks' ideas were taken up by the proponents of agile programming methods (see section 4 below). In 1995, a twentieth anniversary edition of the book came out. Brooks tried to update it, but as he himself admitted, being in academia he was out of touch with the real world issues, problems, and solutions that had developed in the interim. Nonetheless, the core of the book is still relevant today, 27 years after it was first published.

2.3 1975 to the Present

2.3.1 IDE and OO

In the ’70s and early ’80s, structured analysis and design dominated the mind share of software developers. As noted earlier, various sects and sub-sects coalesced around well-known gurus. A standard development “life-cycle” model for software systems, called the “waterfall model”, was first publicly documented in 1970. It immediately became the dominant paradigm (the source of the name and the specifics of the model are irrelevant, since no serious software developer should use it). Unfortunately, structured analysis and design failed to deliver the promised costs savings and increase in reliability. Soon heretics attacked the waterfall model and developed all kinds of new models whose names have thankfully been long forgotten. Despite the wars, two interesting ideas in software development methods did emerge in the ’80s.

The first is computer-aided software tools. Each guru had specific and often complex notations and processes which designers had to use in order to model their software systems. Many of the gurus contended that their methods would begin to pay off only when computer-aided software engineering (CASE) tools would be integrated with the methods. The early CASE tools were primitive graph drawing programs tied to a specific method’s notation. Remember these were the pre-GUI days. But the person who really helped CASE take off was Philippe Kahn, the founder of Borland International. In 1983, Kahn released a revolutionary product: an integrated development environment (IDE) called Turbo Pascal. The idea of an IDE was not new — Emacs had been around for a number of years. But Turbo Pascal came along just when the personal computer was taking off and becoming a widespread software development platform. Turbo Pascal handled many of the tedious and repetitive tasks of program development, and allowed the developer to concentrate on high-level design issues. In fact, Turbo Pascal was the conceptual forerunner of Visual Basic and other visual and integrated development environments. While none of these tools are associated with the traditional idea of a software methodology, they do form an important part of the toolkit for software developers. They are the closest thing to a “silver bullet” for improving the software development process. Unlike IDEs, most of the gurus’ CASE tools have long been forgotten.

The second interesting and useful idea is object oriented (OO) software development. One of the fathers of OO is Alan Kay, the man who said “The best way to predict the future is to invent it.” In 1971, he began developing the ideas behind the programming language Smalltalk. It was further developed at Xerox Palo Alto Research Center (PARC) during the ’70s and ’80s. An in-depth treatment of OO is beyond the scope of this paper. In brief, Kay’s key goal was to make the world of software much closer to the “real” world. In the real world, objects of various sorts communicate by sending messages back and forth. When one object interacts with another, it doesn’t have a clue about the internal workings of the other object. Each object knows the protocols of interactions and communications. Very complex systems can be built by combining objects and letting them interact naturally. The Smalltalk language provided a sophisticated implementation of these ideas.

Proponents of OO argued that its widespread adoption would allow for greater flexibility in software development than earlier structured techniques. It would encourage re-using software since once a well-behaved object was created it could be used over and over in different systems. The ideas behind OO began percolat-

Steve Jobs and Object Orientation

Most everyone is familiar with the famous story of Steve Jobs visiting Xerox PARC, seeing the WIMP GUI of PARC workstations, and falling in love with the idea. This encounter eventually led to the creation of the Macintosh, which in turn spawned Microsoft Windows. What most people miss in this story is that Jobs didn’t have a clue of what he was really seeing. The WIMP was literally the window dressing for the incredibly powerful object-oriented infrastructure that lay beneath. Ironically, Jobs did eventually understand the power of OO. After leaving Apple, he founded Next, which developed a workstation and OS built around OO ideas and and an OO language called Objective C. This was a true successor of Smalltalk. It never succeeded in the marketplace.

ing in the software development community. In the early '80s, the US department of Defense decided that it could save millions of dollars and ensure software reliability by mandating that all software development be done in an OO-like language. Around 1983, the DOD spent millions developing a next generation Pascal with OO aspects. It was named Ada. Around the same time, Bjarne Stroustrup of Bell Labs, created a next generation C with OO aspects, called C++. Neither of these languages fully implemented the power of Smalltalk. Perhaps for this reason (although not only), for ten years OO had to fight for mainstream acceptance. It was the creation of the Web that led to the explosion in the adoption of OO languages and techniques. While Smalltalk itself never took off, its successor languages Java and Python have been extremely successful.

2.3.2 Unified Modeling Language (UML)

The gurus quickly picked up on OO. Perhaps they understood that the conceptual shift from sequential programming to the OO approach would be a difficult and painful one, and training people to make that shift would be a useful (and lucrative) endeavor. The ins and outs of the OO method wars are mostly irrelevant, although they raged for nearly 15 years. In 1997, the war seemed to have finally ended. The Object Management Group (OMG) released the first UML standard. The industry had finally reached agreement on a standardized *notation* for modeling software systems.

Despite my criticism, it is important to note that the gurus also had a human-centric message of great importance that got lost in the heat of the wars. One main argument for structured programming is that software development is mainly about communications between people, and not so much the communication of people to machines. Programs need to be well written and well organized so that software developers can more easily communicate amongst themselves and with their customers. This idea as well is still relevant. The UML is precisely about improving communications.

Of course the methods war did not end with the creation of the UML. Even with a unified notation, there remain many alternative methods for modeling software systems. Nonetheless, the UML is a major achievement. UML is a notation, not a method. Since it is a standard, and not a bad one, it serves as a useful and universal language that can be used to communicate designs and ideas about software systems, no matter what method one chooses to use.

3. General Framework

3.1 Introduction

The history of software development methodologies is filled with a lot of snake oil salespeople. Some of the discredited ideas of the past continue to be recycled and re-packaged. However, there are four main ideas that have emerged that continue to be of great value:

Brooks' people-oriented method.

The UML as a communication aid.

Object-oriented design methods.

Computer-aided software tools.

In the sub-sections below, we provide more information about the first two – these create a general framework for any software development method one might choose to use. Section 4 below provides information about more specific OO software development methods. Computer-aided software tools support the use of the first three, and will be discussed in context and where relevant.

3.2 The Mythical Man-Month Methodology (4M)

In his book, Brooks doesn't just raise issues; he proposes solutions as well. While he does not present these as a methodology as such, they do form the core of a practical and workable software development methodology. In fact, the Mythical Man-Month Methodology (4M) can be seen as the archetype of the various agile methods we will encounter in section 4. These are all heavily influenced by 4M. We will update Brooks slightly to take into consideration the current state of the art. The essence of his methodology is one simple idea: less is more. The methodology itself has five main components.

3.2.1 Project Design — Conceptual Integrity

Brooks contends that integrity of design is the most important consideration in designing and building of anything. "A conceptually integrated system is faster to build and to test." To ensure conceptual integrity, Brooks argues that:

1. The design must proceed from one mind or a small group of agreeing minds. Having a committee specify and design a system is a recipe for disaster.

2. The architect of a system and its implementers must be different people.

For Brooks, architecture refers to the functional specification of the system. The technical implementation requires its own design. One major source of problems in software development is that the conceptual integrity of the functional architecture can be violated or destroyed in the mapping from function to implementation, especially in the world of procedural programming. Brooks didn't address this issue in detail. Object-oriented (OO) system development goes a long way to solving the mapping problem. It also calls for a slight modification of Brooks' original method.

Firstly, we divide the world of software projects into two parts — user projects and programmer projects. The former are those projects where the intended audience are non-programmers. In this case, the system architect needs to be of two minds: a representative of the user community and a software guru. Together they will ensure maximum usability and minimum friction between functional and technical architecture. In programming projects *for* programmers, ideally the architect should be one person, or at most two like-minded people, since presumably the chief architect is also going to be the chief user. One sees many successful examples of the latter approach in free and open-source project development. In fact, in these projects the chief architect and the implementer are often one and the same, so Brooks' second requirement does not seem to apply.

3.2.2 Project Structure — The Surgical Team

Brooks rightfully notes that programming teams don't scale. More programmers mean less, not more, productive work. Brooks proposed a two-part solution to this problem:

1. Break large systems into smaller components. Brooks wrote his book before OO approaches had taken root. This part of the methodology is not easy, but easier today when ideas such as objects and components are taken for granted.

2. For each object or component of the system, assign a small surgical team to design and implement it. Once again, this is easier today, since the mapping between components and their implementation is closer.

In Brooks' world of large projects there are two teams: the system team and the component team:

Producer	The person who gets things done: builds the teams, divides the work, manages the schedule.
Director	The system architect, the person who creates the overall design.
Of course, all kinds of administrative help should be part of this team. Here are the main members of the **component team**:	
Chief programmer	Does most of the heavy lifting: designs, codes, tests, and documents. Research Brooks cites contends that good professional programmers are ten times more productive then poor ones, so it is quite realistic to have one main programmer do most of the programming work.
The programming assistant	Serves as both a sounding board and built-in redundancy. Can also assume some of the less arduous programming tasks. The chief programmer's apprentice.
Program clerk	Handles all the clerical chores of checking code in and out, creating system builds, and whatever mundane technical administrative tasks are involved in developing a lot of code.
Toolsmith	Constructs, updates, and maintains special tools needed by the team.
Tester	Develops system tests and test data.
Editor	Helps put the documentation into good shape.
Language lawyer	Develops and finds the neat programming tricks to get things done. The hacker's hacker.
Administrator	Handles money, people, space, and machines, and interfaces with the rest of the organization's administration.

Some of these team members can be on multiple teams (e.g. the editor or the language lawyer). In addition, there may be administrative support assigned to the team.

3.2.3 Project Implementation — Plan to Throw One Away

Brooks argued that the only way you can know how to build a software system that meets the design specifications is to actually build one. And, of course, the first time you build it, it won't be very good at all. Moreover, as you are building it, the system requirements will change in many ways as a result of users gaining deeper understanding of their needs and the programmers gaining deeper understanding of the problem set. So don't fight the inevitable: plan to build the system more than once, to discard, and to redesign.

This is another part of the method that is far easier today than it was when Brooks first wrote the book. Using tools such as:

- IDEs,
- application servers,
- a very high level language such as Python, and
- reusable objects and components,

developers can rapidly build and deliver small parts of the system for user feedback. In fact, as Brooks notes in his updated edition, with such tools developers can *incrementally* and rapidly build several working versions of the system, getting user feedback, learning about the problem set and system from the very beginning of the process. This allows the developers to modify the system as it is being built, to more closely approach both what the users need and what is the most efficient solution.

Brooks, in his updated book, talks a bit about system simulators and prototyping. Real-time, reactive systems are particularly complex, since they usually involve combinations of both hardware and software. They also tend to be precisely the kind of systems mentioned in the introduction — systems that people's lives depend on. There are many general-purpose and specialized tools for modeling and simulating such systems. Doing so is part of the learning curve and redundancy, which "plan to throw one away" is all about. It behooves developers of such systems to first prototype, then implement.

Also, in the updated edition, Brooks talks about what he calls "Microsoft's Build Every Night" approach. This is an extension of the incremental build approach – check in software modules every night and then build a working version of the system for user testing. Brooks mistakenly attributes this very common implementation approach to Microsoft since he first heard about it from a manager at Microsoft. It actually first came out of the Unix world, and is extremely popular in free software and open source projects. Early versions of the system that barely function are put out to the user community for feedback on usability. In later stages the "nightly builds" are a good way for the user community to be involved in the testing process. Using this approach requires good tools for managing software versions. Two powerful and flexible tools for managing and building large and complex software systems are the open-source Concurrent Versions System (CVS), along with the Unix "Make" utility. IBM, Microsoft and others offer version control and creation that integrates with their visual IDEs. CVS and Make, while text based, are more flexible and, of course, offer cross-platform compatibility.

3.2.4 Project Communications — The Documentary Hypothesis

Brooks says: "Amid a wash of paper, a small number of documents become the critical pivots around which every project's management revolves. These are the manager's chief personal tools." Brooks identifies six such critical documents that answer Who, What, When and Where, and, of course, How Much:

What:	**objectives**	Defines the needs to be met, goals, objectives, and priorities of the system.
What:	**product specification**	Begins as a proposal and ends up as documentation.
When:	**schedule**	
How much:	**budget**	
Where:	**space allocation**	
Who:	**organization chart**	The surgical teams.

Communications is the key to the success of any software development project, and Brooks stresses the need to communicate frequently and in every direction. Communication must flow between the users and the developers, between the managers and the workers, and across and between all teams. Brooks talks about the need for a "project workbook" to facilitate such communication and create historical documentation of the system. Lotus Notes is a well-known commercial system to facilitate such project communication. However, the growth of the Internet and the Web has introduced useful new tools. Wikis and Web-based content management systems are two of these. Plone is an excellent open-source tool that provides these facilities and more. It can be quite useful in building a project workbook.

We will talk more about the use of the UML in developing documents in the next section.

3.2.5 Project Organization — Plan the Organization for Change

Brooks brings a pithy quote from Jonathan Swift: "There is nothing in this world constant but inconstancy." Brooks notes that it is easier to design a system for change than it is to structure an organization for change. Whole books have been written on this subject, and in current management thinking these are issues not just related to software development methods but to the success of any enterprise. Nonetheless, any method focusing on software development must have a huge amount of flexibility built in. OO development and the surgical team approach are key elements that allow for design and organizational flexibility.

3.3 UML Redux

If 4M is the universal method, then UML is the universal language. In this section, we briefly describe the main types of documents found in the UML. The UML is an extremely elaborate set of notations, since it contains 35 years of notation development by dozens of gurus, which was then distilled by an OMG committee over several years. UML is entering into version 2.0. Go to the computer section of any mega-bookstore and you will find shelves upon shelves of books about the UML. Do not despair: only a fraction of the UML is needed in the 4M method. Save your money. Instead of buying elaborate and expensive UML CASE tools, use a simple diagram editor to draw them. Don't worry about being 100% accurate, or syntactically or semantically perfect. Ignore the more detailed notations. In short, use them without buying into the religion behind them. We will note where these documents are used in the communication phase of the 4M.

3.3.1 Use Cases

How do we create and document the product objectives and specification? Story telling is a fundamental tool of human communication — probably the oldest tool we have. Use cases are essentially little stories about the system. They describe a set of scenarios, that is interactions between the user and a system, related to a specific user goal. For example, one use case for an online bookstore is: "Buy a Book." To this goal we add a set of scenarios, and voila, we have a great way to communicate a system requirement. Not only does it clearly delineate what must be done, but it also forms a useful basis of discussion amongst all those who need to define, design, and build the system. The UML includes a use case diagram language. Those who know say skip the diagrams and stick to text. Keep in mind that the operative word is "little." If use cases start getting long, wordy, and overly detailed, they will remain on the shelf, just like any boring book. In general, each use case can fit on one index card.

3.3.2 Class Diagrams

These describe the type of objects in the system and the static relationships between them: associations (customer buys a book) and subtypes (books are a type of product). Fowler rightly points out that there are three levels of class diagrams:

1. **Conceptual** — the objects in the "real world" within which the software system operates. This type of class diagram forms part of the product specification, and is what Fowler calls a domain model.

2. **Specification** — the objects of the system software architecture. At the highest level of a large system, classes can be grouped into "packages". A package diagram is a very high level class diagram, and can serve as the map used for assigning components to the various surgical teams in the 4M.

3. **Implementation** — the lowest level. May be used as software module documentation, although with programming languages like Python and Java this is probably unnecessary.

The UML has quite elaborate, detailed, and advanced notations for describing classes. Since we advocate the 4M, it makes far more sense that the team spends its time coding classes rather than creating elaborate and detailed class diagrams. Simple class diagrams are, however, useful for communication between developers and users, and between different development teams.

3.3.3 Interaction Diagrams

These describe how groups of objects collaborate in some behavior, usually a specific use case. There are several ways of laying out interaction diagrams. Again, use them as a simple modeling tool for the product specification, not for elaborate documentation of the system. An alternative way of modeling collaboration and interaction of objects is using CRC (Class-Responsibility-Collaboration) cards. Remember how in grade school your teacher taught you the index card method of writing papers? When doing research, put each thought on one index card. Then, when you are ready to build the paper, take the index cards and shuffle them around till you have the right order. CRC is very simple. Put an object on a card, and shuffle the cards around to see how they should interact.

3.3.4 State Diagrams

State diagrams show the behavior of a particular object as it reacts to events that reach the object. They are used to show the lifetime behavior of a single object or how an object behaves across many use cases. State diagrams are most useful for describing reactive, event-driven systems. Their main use is in the product specification.

3.3.5 Activity Diagrams

Activity diagrams are relics from the pre-OO days, when the focus was on procedure and process. They describe the sequence of activities in a system. Unlike old-fashioned flowcharts, they allow for showing parallel processing. Activity diagrams are useful for modeling workflow within the organization, providing a deeper understanding of business processes. As such, they serve as an adjunct to use cases.

3.3.6 Physical Diagrams

Deployment diagrams show the physical relationships among software and hardware components in the system. The component diagram further shows the components and their dependency. These are useful for the “where” documents of the 4M.

4. OO Methodology Overview

4.1 Introduction

Information technology in the 21st Century is about sharing information — both within the organization and across organizational boundaries. Developing applications in this environment presents challenges never before faced by IT departments. First, there is the complexity resulting from the heterogeneous nature of IT infrastructures. Then there is the inherent complexity of the applications themselves.

- Internet applications are distributed — they involve multiple computers, both client and server.
- They are event-driven — some event occurs in one location which effects what must be done in another location.
- They are asynchronous — the reaction to the event may occur at a later time.

Hence, complexity of software development has increased by several orders of magnitude since the days of Brooks' OS/360 project. However, despite 35 years of concerted effort, there still is no consensus among computer professionals on how best to approach this complexity. The only thing everyone agrees on is that OO development is the only possible approach for developing distributed networked applications.

There are two major schools of OO methods. The first school is the intellectual descendant of the structured analysis top-down approach. This school continues to believe that a formal, top-down methodology is the best way to develop complex OO software systems. The second school consists of the "agile" methodologists. In fact, they are intellectual descendants of Brooks, although, for some reason, they don't stress enough their huge intellectual debt to him. In fact, agile methods, for the most part, are variations on the 4M theme.

We can only briefly cover OO methods in this paper. If one puts aside the hyped up promises these methodologists offer, there is much to be learned from both schools. Ultimately, however, OO methods can be encapsulated in two simple Roman maxims:

1. *divide et impera* — divide and rule: reduce complexity by dividing the problem up.
2. *veni, vidi, vici* — I came, I saw, I conquered: achieve victory through seeing (understanding) the core problem of software development: human behavior.

4.2 The Three *Amigos*

The UML, and its associated Rational Unified Process (RUP), is the brainchild of three gurus, Grady Booch, Ivar Jacobson, and Jim Rumbaugh, who are referred to as the three *amigos*. This friendly designation softens the monolithic juggernaut that is their company: Rational Software. Someone recently called Rational the Microsoft of the CASE industry. Not a very complimentary comparison! The RUP is the engine through which the three *amigos* and Rational sell their books, training, consulting, and CASE tools. Unfortunately, there is no FDA for the software world, or else the publisher of one RUP tome would never be able to get away with making the following statement: "With this book as your guide, you will be able to more easily produce, within a predictable schedule and more reasonable budget, the highest-quality software possible." Of course Brooks has already taught us that using "predictable schedule" and software development in the same sentence is an oxymoron, no matter what method you use.

That is not to say RUP is not useful. Telling someone to go out and develop a system can be intimidating to a novice. In learning a martial art, the guru stresses mechanical and rote techniques with the beginner. The emphasis is on form, not content. Only with time and experience, does the novice learn that these forms are shells to be discarded once the practitioner has mastered the art. When starting out in OO development, the techniques of the RUP may give the software developer confidence to approach the problem by providing a clear recipe or road map. However, there is no need to invest huge sums of money in RUP books, training, and CASE tools. Moreover, RUP methods can be stripped of heavy "ceremony" and made more lightweight (and thereby more useful).

4.3 Agile Ecosystems

4.3.1 Introduction

The software methods war has come full circle. While Dijkstra's manifesto was a call for *more* discipline in software development, the dissidents to the three *amigos* monolith have issued a manifesto that calls for *less*! Herewith the *Manifesto for Agile Software Development*:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and Interactions** over processes and tools.
- **Working Software** over comprehensive documentation.
- **Customer Collaboration** over contract negotiation.
- **Responding to Change** over following a plan.

That is, while there is value in the items on the right, we value the items on the left more.

There are several parallels between the two schools. The self-styled agilists have begun issuing books in the *Agile Software Development Series* similar to the three amigos' *Object Technology Series* (Addison-Wesley has achieved a real coup because it publishes both series!) The agilist gurus all have their own consultancy and training companies. Agile methods rode the dot-com hype wave since organizational "agility" was seen as the key advantage of "new economy" companies. Despite this, the agile methods books *are* worth a read.

The key contribution of the agilists is that they are spreading the 4M human-centric approach far and wide. Moreover, they are adding additional depth and understanding to the human issues behind software development in general, and object oriented development in particular. Highsmith uses the word "ecosystem" instead of "method" to stress that software development is about people and their interactions and adaptations to a wider environment.

Unlike the monolithic RUP, the agilist stress one size does not fit all, and they provide a variety of methods that fit under the agile umbrella. Here is a very brief overview of just some of them:

4.3.2 Extreme Programming (XP)

XP certainly has had the greatest press and mind-share of all the agile methods. A clever choice of name is one reason, of course. XP is quite close to 4M in many ways, and has done quite a bit to spread its use. Some highlights of XP:

- Use only 3-10 programmers.
- Arrange for one or several customers to be on sight providing ongoing expertise.
- Programmers work in pairs, two per workstation, following agreed upon coding standards.
- Development is done in three-week iterations, with delivered tested code at the end of each iteration, and a working system build delivered to customers at the end of every two to five iterations.
- The unit of specification is the "user story." For each iteration, programmers estimate how much time each user story will take, and the customer then sets priorities.

The team is led by a coach.

- The programmers need to constantly simplify the code. Programmers rotate pair assignments every couple days so that everyone is familiar with all the code.

XP, despite its emphasis on flexibility and human-centeredness, is high ceremony with a disciplined set of rules and procedures.

4.3.3 Crystal

Alistair Cockburn uses Crystal as a metaphor for a family of methodologies. The key idea behind Crystal is to tune the method to the type of project. There are two dimensions to a crystal — color and number. Darker color crystals symbolize larger projects with more people, which require "heavier" methods. Higher numbers in crystals symbolizes harder, higher risk projects, which require more rigor and ceremony. In Cockburn's words:

...the core Crystal philosophy is that software development is usefully viewed as a co-operative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game.

Two consequences of that philosophy are that different projects need to be run differently, and the amount of modeling and communication that people need to do is just the amount they need to jointly move the game forward.

The flexibility of the Crystal approach is also what makes it difficult — without hard and fast rules, managers are left to their own instincts of what is the needed amount of method ceremony. Cockburn builds a self-correcting mechanism into Crystal by requiring projects be divided into short increments of one to three months, and that pre- and post-increment "reflection" workshops be held by the team.

4.3.4 Open-Source Methods

Unlike most other agile method, free software and open-source development doesn't have methodology gurus (they have gurus of other sorts). Eric S. Raymond (referred to as ESR) did create a manifesto called *The Cathedral and the Bazaar*, which tried to explain why open source development was so successful. It boils down to open-source being human-centric, and in fact sharing many characteristics of 4M. Cockburn rightly points out that while open-source development conforms to the Crystal philosophy, it differs from other projects in that it is not a resource-limited game — there is no budget, and programmers play the game as long as they like. Nonetheless, the enormous success of free and open source projects like GNU, Linux, Apache, Mozilla, and so on confirms the value and validity of 4M and agility in large, complex software projects that are mission critical.

5. The Seven Habits of Highly Effective Software Developers and Managers

We conclude this paper with some key lessons to be learned about software development methods:

5.1 Read the books

This paper just provides the tip of the iceberg. Software development is hard, and managing software projects is harder. Despite my ribbing of the gurus, many of them are highly intelligent and some of them have actually successfully ran software development projects. So read the books, but filter them through your own experiences. These are not the words of gods, but of men (mostly).

5.2 Focus on peoples, not programs

This is so counter intuitive that most software developers continue to ignore it. That's half okay when programmers do it — they, after all, do have to ultimately communicate with the machine. But for software managers not to adopt this habit is unconscionable.

5.3 Code more, document less

Another counter intuitive principle. How will the next generation know what we have done if we don't document? But if you have to choose between having a working system or a set of documents, which would you choose? Unfortunately, most of the time, for most projects, that's exactly the choice you face. In any case, the real documentation is contained in the brains of the development team and is best transmitted through storytelling to new team members (or the next team).

5.4 Keep teams small

After 35 years the temptation to succumb to the mythical man-month is still extremely strong. Don't give in! You will live to regret it.

5.5 Have programmers and customers talk

Communicate, communicate, communicate, and communicate.

5.6 Have fun

Cockburn's choice of a game metaphor is not arbitrary. Ultimately, successful projects succeed because the team members enjoy what they are doing. No other motivation works as well as the pleasure of doing a good job.

5.7 Trust your own judgment

If software development is a craft, not a science, then ultimately its success depends on your own skill as a craftsperson. Listen to what the gurus say (including me), but in the end you have to go with your own gut.

6. Annotated Bibliography / Resource Guide

1. For further insight into Dijkstra's thinking, see his article: *How do we tell truths that might hurt?*

http: //www.cs.virginia.edu/~evans/cs655/readings/ewd498.html

2. More on Dijkstra and formal computer science (mainly why its useless on a large scale) at:

http: //laurel.actlab.utexas.edu/~cynbe/muq/muf3_17.html

3. *A Discipline of Programming* by Edsger Wybe Dijkstra. Prentice Hall PTR/Sun Microsystems Press; ISBN: 013215871X.

A quote from a review on Amazon, which reflects my own sentiments: " I really wanted to get my hands on this book and now that I have (via interlibrary loan) I want to warn folks that this is not light reading. I found a majority of this book very boring and all but impenetrable."

4. *The Mythical Man-Month* by Frederick P. Brooks, Jr. Addison Wesley Longman, 1995; ISBN 0-201-83595-9

One interesting addition to the original book is Brooks' essay, written in 1986, entitled *No Silver Bullet*. Brooks predicted that no one programming technique would by itself bring an order of magnitude improvement in software development over the subsequent decade. Sixteen years later, that contention still stands true. \

5. Biography of Alan Kay:

http: //ei.cs.vt.edu/~history/GASCH.KAY.HTML

6. To find out more about the UML, visit the OMG's official page:

http: //www.omg.org/uml/

7. An excellent application server for rapid development of Web-based applications is Zope. Check it out:

http: //www.zope.org/

8. For more about Python, check out:

http: //www.python.org/ and http: //www.python-in-business.org/

9. For more about CVS, see the CVS home page:

http: //www.cvshome.org/

http: //www.cvshome.org/cyclic/cvs/windows.html Also available for Windows!

10. For an introduction to Make, see:

http: //www.mtsu.edu/~csdept/UNIX_HELP/make.html

11. For more about Plone, visit:

http: //www.plone.org/

12. The two best management books I have ever read about structuring organizations for change are:

What They Don't Teach You at Harvard Business School by Mark H. McCormick. Bantam Doubleday Dell, 1988; ISBN: 0-553-34583-4 and

What They Still Don't Teach You at Harvard Business School by Mark H. McCormick. Bantam Doubleday Dell, 1990; ISBN: 0-553-34961-9

13. Unless you are a masochist, or your boss is a sadist, probably the only UML book you will ever need is:

UML Distilled: A Brief Guide to the Standard Object Modeling Language by Martin Fowler with Kendall Scott. 2nd Ed. Addison Wesley Longman, 2000; ISBN 0-201-65783-X

Fowler provides extremely useful tips on how and when to use the various UML documents, without getting too hung up on the formal methods pushed by Rational.

14. A good open-source tool for drawing diagrams that includes UML symbols is Dia, available for Linux, Windows and Mac OS X (via Fink: http: //fink.sourceforge.net/):

http: //www.lysator.liu.se/~alla/dia/

http: //dia-installer.sourceforge.net/

15. A good commercial product for drawing diagrams that includes UML symbols is Smartdraw (and which is a less expensive alternative to Visio):

http: //www.smartdraw.com/

16. Of course, if you are in charge of the specification of a system, you might want another UML-related book all about use cases:

Writing Effective Use Cases by Alistair Cockburn, Addison-Wesley, 2001; ISBN 0-201-70225-8

http: //www.usecases.org/ is the associated website

17. The original paper on CRC cards can be found here:

http: //c2.com/doc/oopsla89/paper.html

18. For more about State Diagrams, visit the white papers section of:

http: //www.ilogix.com/

i-Logix' tool, Statemate(TM), is an excellent simulation tool for prototyping reactive systems, as noted in the 4M method. Full disclosure: in 1989 I was given options in the company (valued at the time of 1/4 of 1% of the shares). The company never IPOed, and subsequent investment substantially diluted the value of those options. Nonetheless, if you all go out and buy i-Logix products and they do IPO, I may be able to buy my son that digital video camera he wants!

19. If you want to read a book about RUP, then read:

The Rational Unified Process: An Introduction by Philippe Kruchten, Addison Wesley Longman, 1995; ISBN 0-201-70710-1

It's virtue is in its relative brevity. The Fowler book on UML is worth reading just for his low-ceremony version of the RUP.

20. An excellent introduction to the new school is:

Agile Software Development by Alistair Cockburn, Pearson Education, 2002; ISBN 0-201-69969-9

Despite the new age-y style, Cockburn has some extremely useful and interesting things to say about the human issues around software development. The more in-depth text is:

Agile Software Development Ecosystems by Jim Highsmith, Pearson Education, 2002; ISBN 0-201-76043-6

21. The Agile Manifesto has its own Web site with links to the various associated gurus and methods:

http: //www.agilemanifesto.org/

22. XP has its own series, published by (surprise!) Addison-Wesley. Start with:

eXtreme Programming Explained, Embrace Change by Kent Beck, Addison-Wesley, 2000; ISBN 0-201-61641-6

http: //www.xprogramming.com/index.htm is the mother of the XP websites.

23. Cockburn has a Web site that contains his writings and more about Crystal. Don't be put off by the poetry and touchy-feely aspects. Cockburn is one smart guy:

http: //alistair.cockburn.us/

24. Eric Raymond continues to elaborate his manifesto at:

http: //www.tuxedo.org/~esr/writings/cathedral-bazaar/

About the Author

Aron Trauring has been managing software projects for small and large companies around the world for over 25 years. He is currently the CEO of Zoteca. Zoteca offers a software workbench with support and consulting services for the rapid development of efficient, safe, robust, and scalable applications used in distributed, networked environments. Zoteca combines open source technologies with unique extensions, offering powerful frameworks for information technology in the Internet age. For more information about Trauring and Zoteca, visit http: //www.zoteca.com/.

info-tech research group