1 Introduction

It is possible to make software by jumping directly on a computer and start hacking away at it. A lot of great software has been written this way. Unless we are software geniouses, however, it pays off to use a more cautious approach. One of Software Engineering’s primary concerns is to establish methods and processes that should be followed in order to develop quality products in the best possible way in a given setting.

A prescribed way to do something is called a method. A method that is enacted in a business or organisational setting is a process. Research on methods is called methodology. A lot of people use the words “method” and “methodology” interchangeably, but it makes no harm to be somewhat more precise—besides, the word “method” is shorter, and a lot of ink would have been saved had it been correctly employed where “methodology” is carelessly thrown in.

From around the 1970s, Software Engineering has had more than its fair share of methods; methodology was thriving. In technical bookstores the shelves are chock-a-block with books on how to develop supposedly great software the right way. As fashions come and go, that means that some approach and a group of associated methods is fashionable for around seven years. Then a new approach arrives that exposes such deficiencies in the until then fashionable approach that people start wondering how they could have been blind all along. The new approach gains adherents. In about five years, it has entered the mainstream, even being taught in universities.

The cycle has been compared to what happens in science. When reading a textbook, one gets the impression that science has been advancing in an orderly pace. That is nonsense. Science advances by leaps and bounds. For some amount of time most scientists work within a given conceptual framework, a so-called paradigm. Then another paradigm appears that supplants the previous one in a process called scientific revolution. In the beginning of the renaissance, this is what happened in astronomy when the Ptolemaic system, with earth at its centre, was supplanted by another world view developed by Copernicus and Galileo.

It is usually thought that a paradigm supersedes another paradigm simply because it is better. That is also nonsense. In the beginning, the new paradigm is not better than the old one; it is untested and imprecise, while the old paradigm
is mature and fully worked out. Indeed, the first attempts to describe the world with the sun at the centre were worse than the Ptolemaic system. It takes time to flesh out the details, and the devil often lies therein. One paradigm wins over a previous one by dint of non-scientific approaches like shameless propaganda, or simply the impetuosity of youth, when young scientists are ready to question their elders, no matter what.

Something similar happens in Software Engineering. Many people mistakenly think that we are witnessing an inexorable move towards better and better methods. The truth is that there is no best way to write software; in the same way that there is no silver bullet to slaughter werewolves, there is no silver bullet that will cure all of Software Engineering’s ailments. Moreover, although we may have literally some dozens of methods at our disposal, on close inspection it turns out that these are not that different from one another.

2 Software Development Processes

The bad news is that a bibliography on software methods would easily fill a tome. The good news is that readers, if they take the trouble to go through them, will discover that all of them comprise a limited set of building blocks. These are:

• Requirements Analysis (often called Requirements Engineering)
• Design and Implementation
• Verification and Validation
• Maintainance

This is about all Software Engineering methodology is about. The reader may now throw away these pages, having conquered the pinnacle of software wisdom.

Before readers do that, let us see a similar situation in an everyday field like arithmetic. Arithmetic is based on the following five axioms, given by Peano:

1. Zero is a number.
2. If \(a\) is a number, the successor of \(a\) is a number.
3. Zero is not a successor of a number.
4. Two numbers of which the successors are equal are themselves equal.
5. If a set \(S\) of numbers contains zero and also the successor of every number in \(S\), then every number is in \(S\).

If you know that, you know arithmetic. Now try adding two numbers using these axioms.

So, although all software development processes are composed from the same set of building blocks, we still need effort both to understand exactly what these components are about and how each method combines them to make software in the real world. Here is an outline.
Requirements Analysis  Before we build something, it is essential to know what we are going to build. Requirements Analysis, or Engineering, is concerned with the “what”, but not with the how. Requirements are captured with special techniques and are written down following special guidelines. All stakeholders must agree to them. Requirements are the guide to design and implementation. It makes no sense to make something that is not required. And a project that does not meet its requirements is incomplete or wrong.

Design and Implementation  Before we build something, and once we do have an idea of what to build, we need to do some planning and exploration. A house is not built once the contracts are signed; architects and civil engineers must make plans and cooperate before a single shovel is lifted. It is much the same with software. First we have to design it, using special design approaches for software; there are many. Then we can implement it, usually by writing code.

Verification and Validation  When we make something, we must be sure that it is what we want. We must also make sure that it is correct. Verification is what we need to do to answer the question “are we building the right system (or application, package, or whatever)?” Validation is what we need to do to answer the question “are we building the system (or application, package, or whatever) right?” We try to answer these questions throughout the whole project effort. Although people often associate verification and validation with program code, requirements, and designs must also be verified and validated—what sense does it make to code the wrong design?

Maintenance  Software always changes, following the changes in its environment, or, alas, the discovery of bugs. A program that does not change is simply defunct. Every software process somehow accepts the fact and tried to fit it in its own way of doing things.

Caveat Emptor  What distinguishes software processes from one another is the emphasis they place on each of these activities and what they prescribe regarding them. A more disciplined development approach will stress the need to perform them in order; a more flexible approach will accept that this is often impossible, and will propose to perform them as the need arises, while trying to avoid chaos. It is also important to keep in mind that there is no perfect method, and no method can be copied verbatim from a book and applied in a project. Those who cook know very well that the simplest of recipes requires improvisation and initiative. The same goes for software methods. We know what they have to do: analysis, design and implementation, verification and validation, and maintenance. They tell us one way to do it. They assume that we are intelligent individuals that will use their judgment to make them work in our own unique project.

No two projects are ever alike. Slavishly following a method and expecting to reap its publicised benefits is a sure way to sink a project. It certainly feels nice to be able to say that our organisation develops software using the X-method, tried and tested in thousands of projects around the world. Discipline gives a feeling of security, but it does not necessarily bring success. We may easily
fall in the trap of spending most of our resources trying to follow as close as possible a method’s prescriptions, instead of spending them in trying to make the product that got us started in the first place. Blind adherence to procedures is not a sign of quality or progress, but of bureaucracy. A method stickler is on a par with a martinet. All methods are but blueprints. A lot of initiative and creativity is required to make them work.

3 Process Maturity

We follow a process so that it may help us develop better software, in a better way. In essence, it is a tool to improve the quality of our efforts. Interestingly, processes themselves are the subject of improvement.

Carnegie Mellon University’s Software Engineering Institute (SEI) has developed a model for appraising the quality of an organisation’s processes. The model is called the Capability Maturity Model (CMM) and has been very influential for the last 10 years. Lately, it has been superseded by a newer version of the model that is applicable to any processes, not necessarily concerning software. The original CMM is still useful as a starting point, as the principles remain the same and it is easier to understand.

The aim is to define a process’s maturity. Five levels are defined, each one of them subsuming the level below it. From the lowest to the highest, they are as follows.

3.1 Level 1—The Initial Level

At the Initial Level there is simply nothing. An organisation has no processes for making software. The people working in a software project devise how to do it, finding solutions to problems as they arise. There is no continuity; each project is handled in an ad hoc way. The next project starts from scratch, and may only benefit from the accumulated experience of the individuals participating in it.

An organisation operating at the Initial Level is not doomed to failure. If it possesses a pool of extraordinarily talented employees and an exceptional manager, it is possible to create amazing products. Indeed, most software start-ups begin like that. Typically, however, as a company grows it is more and more difficult to produce good products like that.

When something untoward happens, project teams react instinctively. They may get it right once, but this is no guarantee that they will get it right the next time. There is now way to predict organisational performance other than the hints we may have from the individuals’ capabilities.

3.2 Level 2—The Repeatable Level

The Repeatable Level is the first, and crucial, step towards process improvement. The organisation has established policies for managing a software project, and procedures to implement these policies. Experience from previous projects is used to help making plans and estimates for new projects. Typically, the organisation has a set of processes that project teams are supposed to follow. It sets targets and guidelines, and the results are compared to expectations.
The adopted processes reflect the experience gained in the past. The level is called repeatable exactly because the organisation tries to set down and repeat what has already been used with success, while avoiding the pitfalls that have led to failures. A rudimentary project management system has been put to practice.

At the Repeatable Level, an organisation has realised that it had better make software in an organised way. It then uses its experience to find that way. Software development is organised, even though the means by which the organisation manages to define its processes are not defined.

3.3 Level 3—The Defined Level

The Defined Level subsumes the repeatable level: the organisation has documented processes for handling software processes. The difference with the level below is that these processes are the results of a specific group that is responsible for the organisation’s process activities—this is called a “software engineering process group” or something similar. That is, not only are processes defined, the way these processes are defined is also defined.

The software engineering process group is responsible for taking the documented processes and tailoring them to the needs of each specific project. Not only are software engineering activities stable and repeatable, management activities are stable and repeatable as well.

An organisation that has reached the defined level has not only realised that it had better make software in an organised way, thus making it to Level 2. It has also realised that it must organise the means by which the organisation defines its software processes, and has created the appropriate structures and procedures to do so.

3.4 Level 4—The Managed Level

At the Managed Level, the organisation has already reached Level 2 and Level 3, so it already has documented processes and also prescribes how these processes are developed. In addition to that, it now sets quantitative quality goals for both its software and its processes. It also collects quality measurements for both of them, and keeps them in some sort of organisational database.

The existence of quantitative targets means that the organisation tries to move within specific limits in a project. Moreover, it keeps measuring its own performance, so that if significant variations are noticed, corrective action is taken.

At this level, the organisation is able to predict its performance, based on knowledge gained from the past. It is even able to manage its own performance based from feedback as the project unfolds. This is a very high maturity process level that requires considerable effort and focus to achieve.

3.5 Level 5—The Optimising Level

While organisations at Level 4 manage their processes based on the feedback they receive, organisations at Level 5 do not solely wait for feedback but also take preemptive steps to optimise their performance. They identify new technologies
and software engineering practices and transfer them in the organisation in order to improve an already carefully managed software process.

Organisations at the Optimising Level continuously analyse their performance and seek ways to optimise it. They try actively to identify which innovations would bring the greatest benefit, without waiting for variations from the targets they have set.

Level 5 organisations are continuously improving. They are not only able to predict and correct their performance, but also to further improve it based on opportunities and deficiencies that they are at the position to identify themselves. Very few companies around the world manage to reach this level.

3.6 The Importance of the CMM

The CMM is very important for the SEI, being a lucrative business in terms of publishing, consulting and certification. There are now many books on the subject, corporations may sign up for courses on how to improve their processes, and consultants are ready to inform companies how mediocre their process maturity is and how they can improve it. For a company, getting the certification that it has reached a certain level is expensive and important. For instance, there are project tenders where bidders are required to have reached a maturity level.

The CMM is important for the rest of the world as well. Even if companies do not have the resources and the financial muscle to go through an institutionalised process improvement model, they still stand to benefit from knowing what they should try to do to improve themselves within their constraints and opportunities. It is naïve to follow software methods blindly; it is no less naïve to follow process improvement methods blindly. The CMM may be most important as a roadmap, but not a recipe, of what a company should achieve. Companies should look at the recommended practices outlined above, and adopt them in their unique environment in any way that seems convenient or practical to them, selecting and adapting one of the methods outlined below.

4 The Build and Fix Development Model

This is the simplest software development method. It consists of the following steps:

1. Build the system and deliver it to the client.
2. If the client is happy go to 4.
3. If the client is not happy, fix it and go back to 1.

The Build and Fix Development Model is probably the most widely used software process in the world. That is thanks to its inherent advantages:

- It needs no training. Everybody can implement it with no prior reading, coaching, or consulting.
• It needs no preparation. You do not need to waste your time; you just jump on a computer and start coding.

• It is easy to understand by all those who take part in the development process—as there is not much to understand about it, anyway.

Despite (or perhaps because) of those advantages, it could be argued that the whole field of Software Engineering is a concerted effort to go past this development model. That is thanks to its inherent disadvantages:

• Unless one can count on an unnaturally talented development team and a management with unique oracle capabilities, the process leads to low quality products. To all probabilities, the client is not likely to accept the first shipment. To all probabilities, the client is not likely to accept the second shipment, or even the third, for that matter. As we never properly designed the software, incorporating changes is fiendishly hard. The honest solution is to actually rewrite the application between each shipment; the commonly used solution is to patch it in any way possible, making future patches even more necessary and difficult.

• There is now way to control the project’s progress. The only thing that a manager or a client ever gets to see is a suite of defective products. There is no indication as to how close we are at delivering a finally accepted version; we may feel we are getting closer, but we may not be, and the client is certainly losing patience.

• It is very unlikely that the client actually agrees with this development model. In other words: should they know that we develop software this way, they would never have hired us in the first place.

Perhaps the best way to get a feeling for the methods’ problems is to try and make software that way. This will be a sobering experience, leading to a lot of stress and sleepless nights, and should lead to a study of the methods that follow.

5 The Waterfall Model

The Waterfall Model takes Software Engineering’s building blocks and puts them in a sequential order that looks like Figure 1.

We start by eliciting and analysing the systems requirements. When analysing the requirements, we are interested in what the system is supposed to do, not how it is supposed to do it. The output of this activity is a requirements specification detailing how. The requirements specification is the input to the next activity, the system and software design.

At the second step we are interested in how the system is supposed to do what is supposed to do. We typically design the interaction of the system with other systems and its users, and create a blueprint of the programs that will have to be coded. The output of this activity is a design specification that is the input to the next activity, the implementation and unit testing.

We now follow the design specification in order to implement the units that comprise the system. It is assumed that in the previous step the system has
been decomposed to parts that are as independent from one another as possible, so that we can implement each one separately from the others. Once each unit is implemented, we test it separately. Once we get them right, we proceed to the next step.

There we take a set of system units and integrate them, i.e., make them work together. This requires some careful plumbing, and once this is in place we must make sure that all of them work together correctly to make the system do what is expected from it; this requires an additional testing step at the system level. Once we are satisfied, the product is delivered to the client.

When the product is put to operation, it is inevitable that deficiencies will be discovered; also, if the users are happy with it, they are likely to ask for some additional functionality. The system’s environment is also subject to change, and we may need to adapt the system accordingly. All this requires that the system must be properly maintained to keep the users and the customers happy.

In the real world, however, things rarely happen in that orderly sequence. At the second step we may discover that some requirements are wrong or missing, so we may need to trace back to the first step and correct the requirements specification before we proceed with design. The same need for a move backwards may arise between any step and any step that comes before it, so we may need to go back from step 4 to step 2, or from step 3 to step 1. That means that Figure 1 looks more like a mesh (or is it a mess?) than like a waterfall.

The Waterfall Model is a true classic:

- It is the oldest and more mature software development process. The first published reference to a stagewise development model was in 1956; the model itself is usually thought to have been first published in 1970. This longevity is amazing in a field where fashionable development methods come and go every few years. A lot of experience has been accumulated on how to use it, and we know potential problems fully well so as to be able to foresee them.

- The model is easy to understand by all those who are implicated in the development process. You do not need to be a Software Engineer to be
able to follow it; users, customers, even management can grasp it.

• It is often compatible with the customer's specifications. This being the oldest model, it is also the most familiar. If the customer knows of one development method and requires that development should proceed along its lines, then this is it.

• The model's building blocks are the fundamental building blocks of all Software Engineering methods. Requirements analysis, design, implementation and unit testing, integration and system testing, operation and maintenance, are always performed in one way or another, no matter the specific method.

• The model makes it easy to divide work between programmers, analysts, salespeople, and management. For instance, salespeople and analysts may work on the requirements; analysts and programmers may work in design; programmers work in implementation and testing. Not everybody is required to work on the same project at the same time.

Of course nothing is perfect; the Waterfall Model might well be a classic, but it is not the be-all and end-all of Software Engineering. It has its own set of shortcomings:

• The model presupposes a linearity that rarely happens in practice. It is true that it accommodates feedback by allowing going returning to earlier stages, but it is difficult and takes a lot of effort. For instance, the requirements specification is a fully-blown document; making changes to it is cumbersome; the same with the design specification. In short, although loops are a fact of life, the Waterfall Model works as if they did not exist, and if they do exist, it makes developers wish they did not exist.

• Requirements analysis can rarely be completed at the start of the project. To get a true feeling of a system's requirements, one frequently needs to do some design, even some implementation first. That leads to the situation described above.

• There is a considerable lapse of time between the start of the project and the first system delivered to the client. The system is only delivered at the very end; until then, a client only gets documentation (requirements specifications, design specifications, test results).

• Following the above, clients and users are late in getting a functional idea of the system under development. They only see it in the end, by which time the changes that have to be made require going back to step 1, with all the costs that this entails.

• There is a long delay before the development team arrives at a working system version. It is not very pleasant to work for months on specifications or program units, without being able to see the result until some time in the distant future.

• The separation of concerns between salespersons, analysts, programmers, and management can have a negative impact. It is easy to imagine salespersons that sell the impossible, or analysts that demand the impossible
from a system, or a management team that, having no working system at hand, dreams the impossible. There is no reality check built in the model—until the end, that is, because sooner or later reality always gets the better of us.

6 Prototyping

One of the problems with the Waterfall Model is the long time it takes before a first version of the system emerges. To correct this, one may go to the other end, and make a prototype of the system immediately upon the start of a project; then, go on making prototypes until the customer’s needs are met. This prototyping proceeds as follows:

1. Communicate with the customer to get an understanding of their needs.
2. Make a prototype of the system.
3. Deliver the prototype to the customer.
4. If the customer is happy, go to 6.
5. Listen to the customer’s comments and go to 2.
6. Celebrate.

This may look a lot like the Build and Fix model, but appearances are deceptive. In that model we develop what we believe will be the final system. In this model we develop what we know to be only prototypes, used to communicate with the customer and then possibly just throw them away. To be able to do that, we must be able to produce prototypes quickly. There are a lot of tools and language environments at our disposal that help us build prototypes fast; the methods requires the use of such prototyping tools, as we cannot afford to spend a lot of time in each loop iteration.

If we truly throw away prototypes in each iteration, then we are talking about throw-away prototyping. If instead we use the previous prototype and we build on it the next one, we are talking about evolutionary prototyping. There are various reasons to use a prototyping approach:

• More often than not, the first version of a system must be thrown away; it is a mistake that we are so hesitant to do it. The reason is that we start with so many unknowns that, when we do have some version in our hands, we would wish we could start afresh. This is the “had I known all that at the start, I would have done it correctly” feeling. Well, we can actually do something about that feeling and actually start again from the beginning. This can be relatively painless if it is easy to start again, and if we have not invested too much effort in building the version we have in our hands—two conditions that prototyping tries to meet.

• Prototyping allows us to discover along with the customers the systems requirements. We start with the barest of requirements, we show the result to the customer, the customer uses it and fleshes out the requirements, and so on and so forth.
Both users and developers have something in their hands, early on. Users like the fact that they do not have to wait too long. Developers like the fact that they can actually see the results of their efforts soon.

At the same time, one has to be cautious:

- It is easy for the customer to take a prototype for the final product. That is very dangerous. You go on and develop something that you know it is only a prototype and should never go into production. Alas, with all the modern development tools in your hands, this looks so professional that nobody things that you have not finished; they rather think you try to eat up some more budget, and they typically ask you to “finish it up by the end of the week”. Then you try to convince people that appearances are not what the seem.

- In order for prototypes to be produced quickly, it is inevitable that shortcuts are taken, and compromises are made. It is easy for them to make their way to the final product. What seemed only a temporary solution might become something that will haunt us for years to come.

- The final product is the result of prototypes evolution. That means that it is not the result of a careful design phase, and requirements have simply been accumulated on it. It may be difficult to maintain such a product.

7 Incremental Refinement

In prototyping, we try to arrive at the final project by means of successive implementations of it. We might, alternatively, follow an evolutionary approach where, instead of producing a series of prototypes, we produce a series of functional components. That is, we start with giving to the customer a working system that performs only some well defined part of the original requirements. If the customer is happy with it, we develop a new component adding some missing functionality and add it to the delivered system, and so on:

1. We partition the system in a number of functional components that can be developed independently.

2. We choose the most critical of the undeveloped components, design, implement and deliver it to the customer.

3. If there are no undeveloped components, we go to 5.

4. We go to 2.

5. Celebrate.

Suppose we have to develop a word processor. Clearly, the most critical component is the ability to create, save, and open text documents. Three other components are the ability to change text (cutting, copying, pasting), the ability to format text, and spell checking. We would start implementing the component with the creation, saving, and opening functionality. Then we would develop the component implementing the text modification functions, followed by the component to allow document formatting, and finally by spell checking.

Incremental development has a lot going for it:
• It can be used when we are short of human resources. As we concentrate on some part of the system each time, we need only a fraction of the people we would need to develop the full product at once.

• The clients can start using the system productively very soon. Whereas in prototyping they get a full view of the system, but which they cannot use, in incremental refinement they get a partial system of the view, but which is functional.

• As in prototyping, the experience gained from previous iterations can be used during development, by both developers and users.

• As we have something functional at our hands, we reduce the risk of a general failure where we may not be able to deliver anything at all.

• As the most critical parts of the system are the ones that are delivered first, in every iteration they are tested and used by the users so that they are more and more unlikely to harbour critical bugs.

Unfortunately, nothing is perfect:

• For the method to work, we must be able to partition the system in small functional components that can be developed in sequence. The meaning of “small” is relative to our resources, of course, but it is not always possible to partition a system that way.

• Any compromises we may make early on in the life of the system are difficult to change later on, as parts are built on the previously developed parts. That means that the system must be very carefully designed upfront.

8 The Spiral Model

In the Spiral Model, extra care is taken for an evaluation of the risks inherent in each development phase. A development phase might be the building of an early prototype, requirements analysis, high level design, detailed design, implementation, testing, and so on. There is no preordained list of allowed phases, and in each project there might different ones. At the start of each phase we do the following:

1. We set our objectives on the phase.
2. We assess the risks for the phase, and we try to find ways to reduce them.
3. We do the work associated with the phase, and we validate it.
4. Based on the results of our validation, we plan the next phase.

To take an example, we may start a new project by analysing the project objectives, alternatives and constraints. We assess the risks, with the information we have at our hands, and to make our evaluation more robust we build an early prototype. Based on the results of evaluating the prototype, we arrive at an early concept of the general operation of the future system. We then plan the
next development phases, which would be analysing the system requirements and creating a first plan for the system’s life-cycle. We review our progress and commit to our decisions.

We then proceed to a second iteration, where we determine objectives, alternatives, and constraints for requirements analysis. We assess the risks in requirements analysis, and we build a second prototype to help in our assessment. Based on the results of evaluating that prototype, we analyse and validate the requirements. We then plan the next development phase, which would be systems development.

We go on like that until in the final iteration we have an operational prototype, from which we go on to code, test, and deliver the system to the customer.

As each iteration continuous from the previous one, the whole process can be pictured as a spiral, shown in Figure 2. In that figure we start from the innermost top left spiral and we expand clockwise and outwards. Figure 2 depicts three dimensions. While going round in the $x - y$ plane, we go up in the $z$ axis, which represents the accumulated development cost.

Although in our example we used prototyping in each phase, this need not be the case. The spiral model can accommodate different development processes in each iteration. For instance, we may use prototyping in the first iteration, some adaptation of the waterfall model in the second iteration, return to prototyping in the third iteration, do incremental development in the fourth iteration, and we go on iterating as long as is required. As there is no preordained set of
development phases, there is no predetermined number of iterations, and what we do in each one of them is up to us. This means that:

- The Spiral Model is adaptive to many different settings and needs. In fact, it can be viewed as a meta-model, a model for creating other development models; we use the basic ideas of setting objectives setting, risk assessment and reduction, development and validation, and planning, to orchestrate the use of other methods.

- It makes risk management explicit. Risks are usually handled in an off-hand way in software, which may have detrimental results. The Spiral Model makes an explicit commitment in handling risks.

At the same time, however:

- The Spiral Model is complex. It has a long history, and has been influential in the Software Engineering literature, but Software Engineers and clients alike find it hard to understand.

- Proper risk management, which requires special metrics and appraisal tools, may be overkill for some projects.

9 Formal Methods

One of the biggest problems in Software Engineering is the gap between specifications and actual product. When we capture and write specifications, it is unavoidable that we are being vague in some parts. Natural languages are never precise. We may not say something because that something goes without saying; or something that is crystal clear to us may be all too obscure to others; and the implications of what we say may be open to interpretation. These are not really deficiencies; were they otherwise, natural languages would not be a viable communication medium between humans: imagine humans communicating with each other using some formal language, unambiguous and precise like a computer language.

On the other hand, mathematics and formal languages are clearly very effective at describing certain things. An algorithm is often best represented in a computer language, instead of free-flowing speech; might it not be that system requirements could be written in a formal manner?

This has been the subject of formal methods, where mathematical formalisms are used for recording system requirements. Such requirements are unambiguous—they only mean what the mathematics allows them to mean; and they are complete—a system should only do what prescribed by the mathematics, and nothing else. What kind of formalisms are these?

Figure 3 shows a formal specification in Z, one of the most popular formal specification languages. The specification is of a birthday book application. The application records birthdays of people. The specification states that we may add birthdays, find an already stored birthday, and remind us via birthday cards, in a given day, the people celebrating their birthday.

In case this specification looks otherworldly, formal languages have been and are used in real projects, like, for example, the Mars mission. There are several reasons for that:
\[ [\text{NAME}, \text{DATE}] . \]

\[
\text{BirthdayBook} \\
\text{known : } \mathbb{P} \text{NAME} \\
\text{birthday : NAME } \mapsto \text{DATE} \\
\text{known} = \text{dom} \text{birthday}
\]

\[
\text{AddBirthday} \\
\Delta \text{BirthdayBook} \\
\text{name? : NAME} \\
\text{date? : DATE} \\
\text{name?} \not\in \text{known} \\
\text{birthday}' = \text{birthday} \cup \{ \text{name?} \mapsto \text{date?} \}
\]

\[
\text{FindBirthday} \\
\Xi \text{BirthdayBook} \\
\text{name? : NAME} \\
\text{date! : DATE} \\
\text{name?} \in \text{known} \\
\text{date!} = \text{birthday}(\text{name?})
\]

\[
\text{Remind} \\
\Xi \text{BirthdayBook} \\
\text{today? : DATE} \\
\text{cards! : } \mathbb{P} \text{NAME} \\
\text{cards!} = \{ n : \text{known} \mid \text{birthday}(n) = \text{today?} \}
\]

\[
\text{InitBirthdayBook} \\
\text{BirthdayBook} \\
\text{known} = \emptyset
\]

Figure 3: An Example Z Specification
A formal specification gives a mathematically complete specification of the system. If we wish to ask whether the system should do something, we only need to check with the specification and use some mathematical techniques to see if the desired behaviour can be exhibited by the system.

A formal specification is unambiguous. Being a mathematical construct, there is no room for misinterpretation—if you know how to read a formal specification, that is. Experience shows that this is not exceedingly difficult; people need only some familiarity with mathematics to be able to read them.

A formal specification only specifies what a system must do, so there is a clear separation between the requirements and the system design. There is in general a tendency for design aspects to creep into requirements, and formal methods help against it.

The transition from a formal specification to a computer language is actually a transformation from one mathematical formalism to another. It is possible to develop systems by proceeding in steps, refining an initial formal specification at each step with the goal of arriving at the final program, while trying to make sure that each transformation step is correct. This may be an arduous task, but it increases our confidence that we are building the right system.

Still, only a small part of software worldwide is developed using formal methods nowadays, and this is usually safety-critical software where the margin of error is small and the cost of error enormous:

- Although not exceedingly difficult to learn, formal methods need commitment. Many people find it more natural to express and record their requirements in some form of natural language.

- A formal specification takes time to develop. It is more detailed than requirements in natural language, but that detail comes with its cost.

- A formal specification can only model some part of the world. That part it can faithfully represent, but the final system must be implemented in an environment where many factors that are not taken into account by the formalism have their impact. Programming language implementation details, the behaviour of the operating system and the underlying hardware, combine against the perfect mathematical model represented by a formal specification. A formal method cannot guarantee that a program is correct, because a working program runs in the real world, not in the realm of abstract mathematics.

**10 Exercises**

1. For each one of the software development methods presented in the text, describe a software system for which you would choose it and a software system for which you would not judge it as appropriate.
2. Describe in detail a use scenario of the spiral model, from the start of the project until delivery to the customer, during which at least three different development models are employed.

3. Which Software Engineering activities do you think can be automated? Which ones do you think cannot be automated, and why?

11 For Further Details

Scientific revolutions are the subject of the seminal book by Kuhn (1970). This book is probably the most influential book in philosophy of science. It has come to mean a lot of different things to different people. The book is eminently readable, short, and highly recommended. Readers who doubt that science advances by propaganda and wile should read the excellent book by Feyerabend (1993), where it is made clear that in science anything goes. The Peano axioms are copied verbatim from Mathworld (Weisstein, 1999).

The importance and ramifications of software processes are analysed in detail by Humphrey (1989). There are reams of paper published on the CMM; a short overview is given by Paulk et al. (1993).

The original publication of the Waterfall Model was by Royce (1970), although the first publication of a stepwise model was much earlier (Benington, 1956). Incremental development was proposed by Mills et al. (1980), and the Spiral Model was introduced by Boehm (1988).

References


