BIT 314-CSC 311

Chapter 2. Process and Model

Objectives

At the end of this chapter you should be able to:

- Define software engineering.

Describe generic framework activities of the software engineering process.

- Describe various process models, such as the waterfall and prototyping models, in depth.
- Explain the difference between prescriptive and agile process models.
- Describe the main components of CASE tools, and how they can address system development problems.

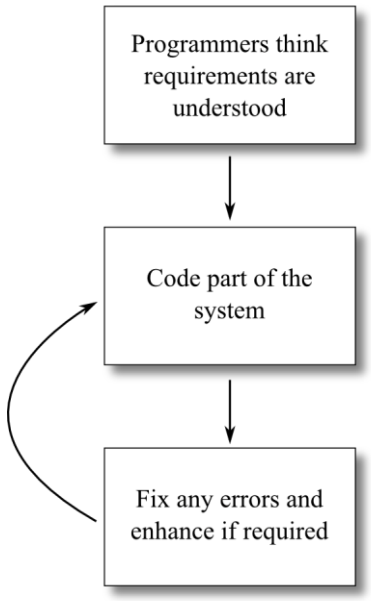
The software crisis

There were many difficulties in the development of large software systems during the 1960s and 1970s. The term “software crisis” dates from that time. The problems stemmed from an inability to apply the techniques used to build small software systems to the development of larger and more complex systems. The typical way to develop small systems can be described as “code-and-fix”.

The code-and-fix approach to software development

The “code-and-fix” approach to software development is not a proper life cycle (see later this unit). Code-and-fix development occurs when software engineers come together with a vague set of requirements and start producing software, fixing it, and changing it until the correct product appears.

Figure 2.1. The code-and-fix approach



This is the simplest way to produce software and is invariably how every programmer learns to program. But for anything other than small software projects, code-and-fix is a disaster for a number of reasons:

- There is no way to estimate time-scales or budgets.
- There is no assessment of possible risks and design flaws: you may come close to a finished product only to find an insurmountable technical problem which sets the whole project back.

We only mention the code-and-fix approach in the context of life cycle models since it is a base-line model which we should avoid. From a software engineer's point of view, code-and-fix is a worst case scenario.

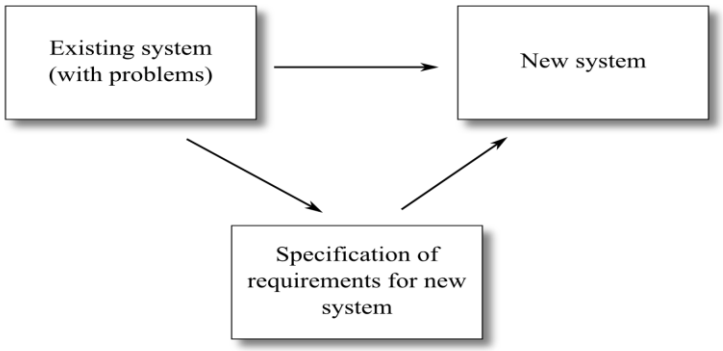
Many project failures resulted from the inability to scale the techniques employed when developing small software systems to handle larger, more complex systems. This failure leads to:

- never completed systems
- missed deadlines
- exceeded budgets
- a system that does not do all that is required of it
- a system that works but is difficult to use
- a system difficult to modify to meet changes in organisational needs and practices
- a loss of trust from users, who may experience many problems with using the software.

These problems were largely due to the lack of any framework for the planning and organisation of software development projects. Although some software projects were organised, and these were often the more successful ones, it was the luck of the draw whether a project manager had good intuitions for software development, and whether or not problems arose due to misunderstandings between the customers and the developers of the system. Likewise, there were no clear methods to monitor whether a system was soon to go over budget or miss deadlines.

From some of these problems we can see that at some stage the system developers attempted (not always successfully) to understand the requirements for the new system. We can now include in our diagram of the process these specified requirements for the new system:

Figure 2.2. The process with requirements



Software engineering and the software process

Recognising these problems, work was carried out to understand the process of software development and to transform it into a **reliable and rigorous discipline**, like architecture or engineering. An improved **process** should produce software that is **correct**, **reliable**, **usable** and **maintainable**. By understanding the process, it should be possible to **plan projects** with more accurate predictions of **cost** and **time**, and provide ways of **monitoring** intermediate stages of project progress, to be able to react and re-plan if a project begins to go off budget or timescale.

Software engineering is exactly the discipline of producing such software. Fritz Bauer defined software engineering to be: "*the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.* "

Much research has been put into the study of past systems that were both successful and unsuccessful. This can be summarised as:

- Some software development activities appear to be common for all successful projects.
- Some activities need to occur before others.
- There is a need to both understand that requirements change, and to manage this change.
- Any existing systems need to be understood before working on the design of a new one.
- It is wise to delay decisions that will constrain the final system — this can be achieved by initially designing an implementation-independent logical design (see the next chapter), before committing to a detailed design for a particular physical set of hardware and software.

Analysis of such findings led to a model of what is called the **software process**, or **system life cycle**. The *software process* is the process of engineering and developing software; a **process model**, or **life cycle model** is a *descriptive* model giving the best practices for carrying out software development (i.e., for carrying out the software process). However, a process model is often treated as a *prescriptive* model that needs to be followed precisely, without any deviation. This should not be the case. The specific model of the software process used should be tailored to meet the specific needs of the project and the developers working on the project.

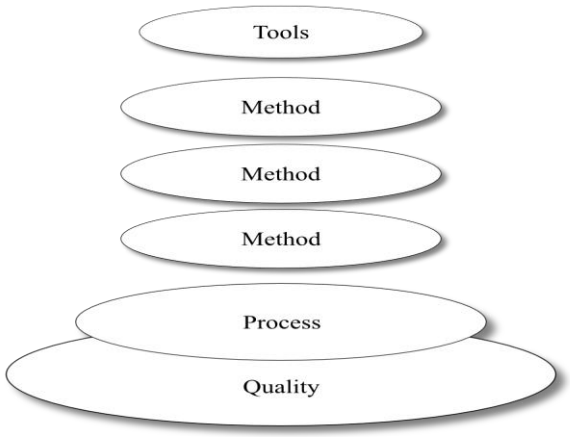
Note

The phrases “Software process”, “Software Life Cycle”, “System Development Life Cycle”, “System Life Cycle”, “Development Life Cycle” are all used to describe the same concept.

The layers of software engineering

Software engineering is a discipline that can be pictured as being built up of layers (Figure 2.3, “The layers of software engineering”).

Figure 2.3. The layers of software engineering



Software engineering demands *a focus on quality*. This should permeate throughout the rest of the engineering discipline.

On top of this comes the foundation of software engineering: *the software process*. The process is the framework on which the rest of software engineering is built. The process defines how management occurs, what the required input and output products are, what milestones should be reached, and so on. The process also describes how quality should be ensured.

On top of process, software engineering consists of *methods*. These describe how the various portions that make up the software process should be carried out. For instance, how to communicate with clients, how to test the software, to gather requirements, and so on. This makes up the process model.

And above all of this, and in support of the whole discipline, are the *tools*. The tools support the software process. Such tools are called *computer-aided software engineering* tools.

A generic framework of the software process

A software process consists of the activities that are carried out during the development of every software system. There are specific activities which are carried out at specific times, as well as activities carried out throughout the project's lifetime. Such life-long activities are called *umbrella* activities.

A generic framework defining these activities for the software process can be given. It identifies activities common to most of the models of the software process, although each model adapts the activities to its own ends.

The activities are as follows:

- Communication - This activity involves the gathering of software requirements from the customer, and related sub-activities.
- Planning - This is the activity of planning the work required to develop the software. This includes risk management, listing the associated outputs, and producing a schedule for the work.
- Modeling - This activity is involved with modelling both the requirements and the software design, so that both the developers and the customers can better understand the work being carried out.
- Construction - This is the development of the software. This activity also includes sub-activities for testing the software.
- Deployment - The software is delivered, and the customer provides feedback on the software.

Software models

The framework just presented provides a list of generic activities common to most models of the software process. However, each model treats the activities differently, and each model is suitable for different projects and for different teams.

It is important to realise that the activities outlined in the process models given below *should* be modified, based on:

- The problem having to be solved.
- The characteristics of the project.
- The nature of the development team.
- The organisational culture.

Prescriptive and agile models

Prescriptive software models are those which *prescribe* the components which make up a software model, including the activities, the inputs and outputs of the activities, how quality assurance is performed, how change is managed, and so on. A prescriptive model also describes how each of these elements are related to one another (note that in this sense, “prescriptive” is not meant to indicate that these methods admit no modification to them, as we previously used the word).

On the other hand, *agile software models* have a heavy focus on *change* in the software engineering process. Agile methods note that not only do the software requirements change, but so do team members, the technology being used, and so on. We will discuss agile methods later in this chapter.

Prescriptive software models

The waterfall life cycle model

The waterfall model was the first, and for a time, the only process model. This model is also known as the “traditional” or “typical” software life cycle.

Note

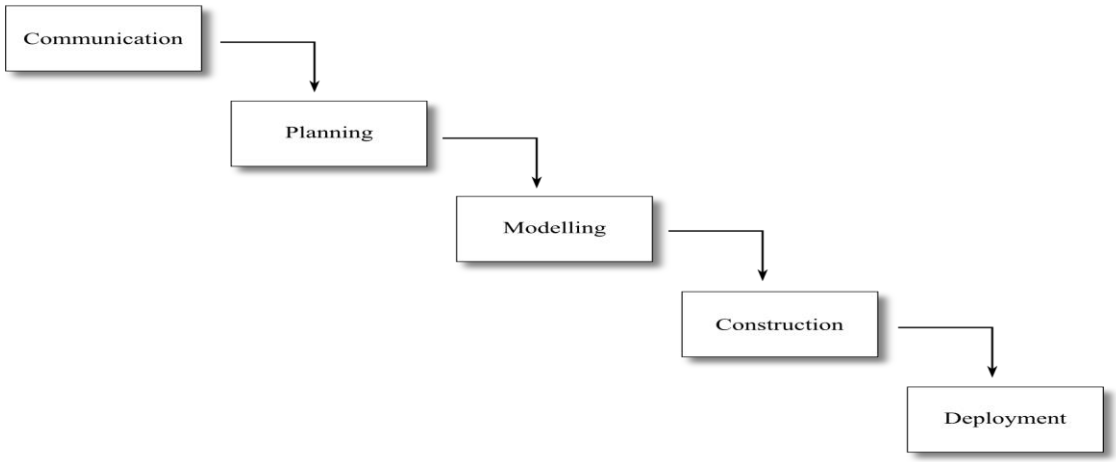
Some writers use the acronym TLC, standing for “Traditional Life Cycle”. In this module we will refer to this software model as the “waterfall” model. This model is sometimes also called the “linear sequential” model.

The features of the waterfall model are as follows:

- The system development process is broken into distinct stages.
- Each stage involves a particular project activity — such as “communication” or “construction”.
- Each stage, when completed, results in a deliverable (also called a product).
- The input to any particular stage are the deliverables from the previous stage.
- The model presents the stages in a strict, one-way sequence — a project cannot go back to repeat a stage once that stage has been completed. Any required re-work (as a result, for example, of changing software requirements) is very limited.

The name “waterfall” comes from likening this method to a river cascading over a series of waterfalls: the river is the output from each stage; the output from one stage is the input to another, in strict-sequence, and at no point do the stages reverse; a project cannot go back to a stage that has previously been completed.

Figure 2.4. The waterfall method



Each stage of the waterfall method flows into another.
Stages do not flow backwards through the model.

There are some slight differences in the way the waterfall model is presented between different books, however these differences are usually related to the number and names of the stages. Any presentation of the waterfall model will present a very similar sequence of stages to the following:

- Communication
- Planning
- Modelling
- Constructio

- Deployment

There are a number of advantages to the waterfall model:

- The stages consist of well-defined tasks which promotes good scheduling and cost estimation (if all stages occur in the expected sequence once only).
- The deliverables provide targets or milestones to see how far a team has reached in the development process.
- The life cycle is broken into well defined stages — so staff expertise can be used efficiently (e.g., a data modeller only needs to work on certain stages, a programmer only on other stages, and so on).
- At any one time the project team knows what should be happening and the deliverable(s) they are to produce.

However, there are also a number of major limitations of the waterfall model, which occur frequently in software development:

- It is rare that a software development project will follow the sequential process that the waterfall model uses.
- Although the requirements are specified early on, user understanding and feedback of the software will not occur until after the system is implemented, which is possibly too late (or very costly) to change.
- The user may not be able to describe the requirements of the desired system in any detail early on.
- The model does not easily allow for the anticipation of change — some systems take years to develop, but once the early stages have been completed the model commits the project to a fixed specification of the system.
- Many projects based on the waterfall model stress the importance of certain products (documents) being delivered at certain times — it is possible for a project to become managed in a bureaucratic way, with documents being delivered on schedule, but the focus drifting away from developing a usable, effective system for the users.
- If a problem is identified at a later stage, the model does not make it easy (or cheap) to return to an earlier stage to rectify the mistake (since all intermediate steps will need to be repeated, resulting in significant, unplanned, time and resource costs).

For many development projects, the limitations of the waterfall model are usually considered to far outweigh its advantages.

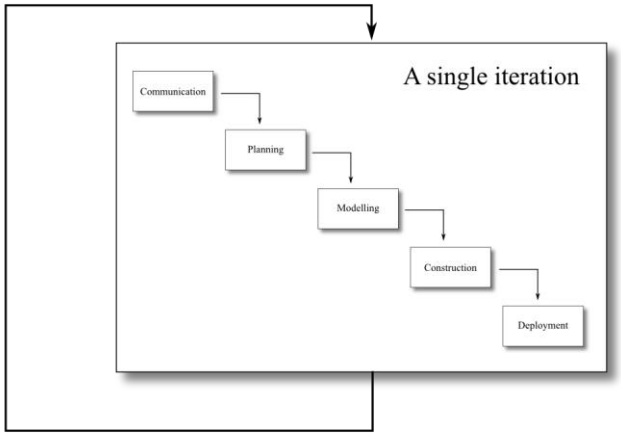
Incremental process models

Incremental process models provide limited functionality early in the software's lifecycle. This functionality is then expanded on in later releases. We will examine two such processes.

Incremental development software model

Incremental approaches attempt to maintain some of the advantages of the pure waterfall model, but attempt to allow for greater change management and overall flexibility in the software process.

Figure 2.5. The incremental development software model



Software development occurs in small increments, allowing this model to handle change far better than the waterfall method.

The incremental model allows the developers to quickly release a version of the software with limited functionality, and then at each development iteration to add additional, incremental functionality. The development in each iteration occurs in a linear method, as with the waterfall model. Ideally, the most important functions are implemented first and successive stages add new functionality in order of priority.

By doing this, the full development task is broken down into smaller, more manageable portions, allowing implementation problems to be highlighted before the full system is completed.

Incremental delivery is this process of releasing the product to the client at the end of each iteration. This allows the client to use regular, updated versions of the software, giving them the capability to judge the progress of the software development.

Although early, incremental delivery of the software is an option for project managers, it is not necessarily the case that each sub-system is delivered to the user as soon as it is completed. Reasons for delaying delivery may include the complexities associated with integrating the customer's existing software system with the limited functionality of the new system — it might make better sense to wait until a more functional implementation of the new system is completed.

An incremental development approach has the following advantages:

- The process is more responsive to changing user requirements than a waterfall approach — later sub-systems can be re-specified. Also a modular approach can mean maintenance changes are simpler and less expensive.
- There is an opportunity for incremental delivery to users, so the users can benefit from parts of the system development without having to wait for the entire life cycle to run its course.
- Incremental delivery means that users have a portion of the software to examine in order to see how well the software meets their needs, and whether the software requirements have to be modified.
- Complete project failure is less likely, since users will have some working sub-systems even if time and money run out before the complete system is delivered.
- The project can begin with fewer workers, as only a subset of the final product is being worked on.
- The risk associated with the development of the software can be better managed.
- The time taken to develop previous iterations can be used as an estimate for the time needed to develop the remaining iterations, and hence improve project planning.

There are some costs, and dangers associated with an incremental development approach though:

- This development model relies on close interaction with the users — if they are not easily available or slow in evaluating each iteration, the whole process can slow down.
- The reliance on user involvement can exacerbate the already difficult task of estimating the amount of time and budget required.
- High user involvement means that resources are drawn away from the customer's normal operation during system development.

Rapid Application Development (RAD) process model

Rapid Application Development is an incremental process model that has a focus on short development cycles (hence the term "rapid"). This speed is obtained by using off-the-shelf components, and a component-based design and implementation approach.

It has the following advantages:

- Development cycles are rapid, typically between 60 to 90 days. It has the following disadvantages:
- For large projects, RAD may require a large number of people to split the project into a sufficient number of teams.
- The developers and the customers must be committed to the necessary activities in order for the process to succeed.
- The project must be suitably modularised in order for RAD to be successful.
- RAD my not be appropriate where high-performance is necessary.
- RAD may also not be appropriate when technical risks are high.

Evolutionary process models

Product requirements may change with time, even while the software is under development. Worse, the initial specifications may not be detailed, and tight deadlines may result in a need to have software quickly ready.

All of this points to a product that evolves over time, and evolutionary process models are designed to satisfy the engineering requirements of these products. Evolutionary process models are, as we shall see, iterative; they allow for the software engineer to deliver a product, and then iteratively move towards a final product as the understanding of the product improves.

We will discuss two such process models below. One disadvantage to keep in mind is that it can be difficult to plan the number of iterations, and hence the length of the project, in advance.

Prototyping life cycle model

A prototype system is a smaller version of part(s) of the final system that gives the user a sense of the finished system's functionality. It has some of the core features of the final system and, where features and functions are omitted, it pretends to behave like the final system. Prototypes are typically developed quickly, may lack unnecessary features, may be buggy, and have poor usability. However, prototypes can fill an important role in understanding software which does not have clear requirements.

Where the system to be developed is a truly new system, there may be no clear requirements defining the software's behaviour. By building a prototype, both the developers and users have some real, visible

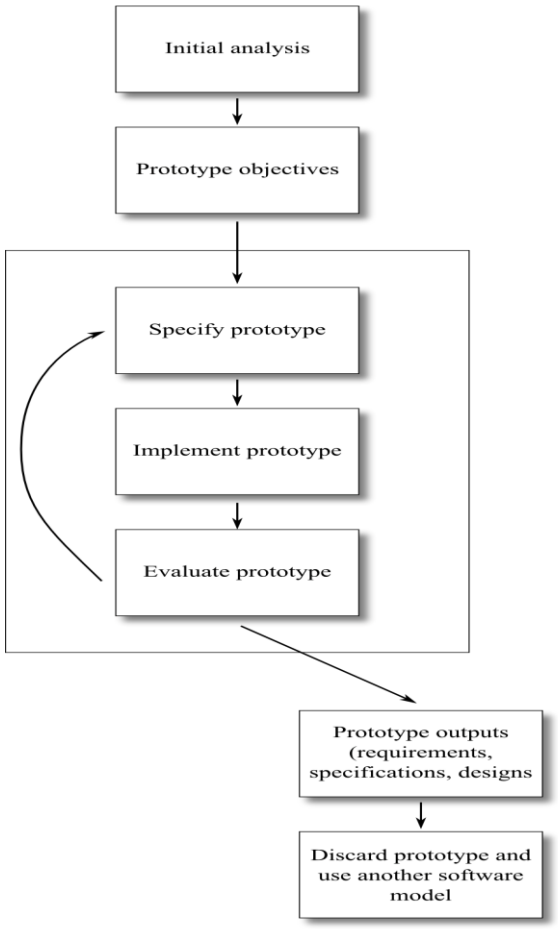
working system model on which to focus their ideas. An analysis of this prototype forms the basis for the requirements specification, and perhaps even some of the design. If there is still uncertainty of the new system and questions still remain, further prototypes can be developed (or an existing prototype extended). In this way, prototyping allows developers and customers to better understand incomplete and fuzzy software requirements.

Once the developers and users have a clear idea of the software's requirements, the project can move into a another development life-cycle, and the prototypes are *thrown away*. This is important, since as we previously mentioned, the prototypes are generated quickly and are not designed to be robust or complete.

To prototype quickly and effectively, fourth generation languages (4GLs), graphical user-interface (GUI) tools (like those that come with Visual Studio, QT and GTK), and off-the-shelf components are commonly used. The quality of the prototype is only of concern where it would hinder the prototype's use in understanding the final software being developed. If the prototype is usable enough to meet the objectives put forward for its development, the prototype has been successful.

A diagram of the disposable prototyping life cycle stages is presented as follows:

Figure 2.6. Disposable prototyping



As can be seen, after some initial analysis a set of objectives is developed for the prototype. These objectives may differ between projects — perhaps detailed requirements need to be elicited, perhaps alternative user interactions are to be evaluated and so on. Each version of the prototype should be specified so that the software can be correctly designed and implemented — a prototype that does not fully test the objectives is a waste of resources, and may be misleading. Once a prototype has been completed it should be evaluated against its objectives. The evaluation decides whether the prototype should be extended, a new prototype developed, or — if the specified objectives are met — if the project can move on to develop the software using another process model.

Advantages of prototyping include:

- Users get an early idea of the final system features.
- The prototype provides an opportunity to identify problems early and to change the requirements appropriately.
- The prototype is a model that all users and customers should be able to understand and provide feedback on, thus the prototype can be an important tool to improve communication between users and developers.
- It may be possible to use aspects of the prototype specification and design in the final system specification and design, thus some of the prototype development resources can be recouped.

A major problem with developing "disposable" prototypes is that the customer may believe it to be the final product. Customers may not understand the need to re-engineer the software and restart development, and may ask that the prototype be "cleaned up" and released to them.

Boehm's spiral model

The *spiral model* was published by Barry Boehm in 1986. It provides an iterative, evolutionary approach to software development combined with the step-by-step aspects of the waterfall process model and the requirements analysis abilities of prototyping. It is intended for development of large, complicated software projects.

This process model provides for the rapid development of progressively more complete versions of the software. Each iteration of the evolutionary development will have a release, which may merely be a paper model of the software, a prototype, or an early iteration of the software.

Each iteration of the spiral model contains all of the activities from the generic process framework outlined above: communication, planning, modelling, construction and deployment. One can consider an iteration to be an arc in a spiral: each arc contains the same breakdown of how the development is approached, but each arc will focus on something new.

Each iteration also requires a certain amount of risk assessment, in order to lay out the plans and determine how the project should proceed. Risk assessment will adjust the expected number of iterations, and also affect what milestones are expected. The development of prototypes (as with the prototyping life cycle model) is an ideal way to mitigate the risks involved with poorly understood or vague software requirements.

The advantages of this model are:

- The spiral model considers the entire software life-cycle.
- Because of its iterative approach, it is adaptable, and appropriate for large-scale projects. However, the model does have disadvantages:
- It requires expertise at assessing and managing risk.
- It may be difficult to convince customers that such an evolutionary approach is necessary.

Component-based development

In this process model, software is developed by integrating pre-developed software components and packages. This may be commercial, off-the-shelf components, or they may be components previously developed by the software engineers themselves.

Each component needs to present a well-defined interface to allow for easy integration.

The component-based model proceeds through the following steps:

- Determine what components are available and evaluate them for their suitability.

- Consider how the component will be integrated with the software.
- Design the software architecture so that the components may be easily employed.
- Integrate the components into the architecture.
- Test the software to ensure that all of the components are functioning appropriately together.

This approach may lead to a strong culture of component reuse. It has been shown that this model also leads to a 70% reduction in development time, an 84% reduction in project cost, and increased developer productivity.

This model is similar to RAD, which we discussed earlier. Note that RAD differs in that it is focused on *rapid* development, rather than specifically on component reuse.

The formal methods model

The *formal methods* model focuses producing formal, mathematical specifications of the software product. When the software is built to the given specification its behaviour will already have been verified to strictly meet the software's specific requirements.

The importance of this model comes from its ability to discover ambiguity, incompleteness, and inconsistency in the software requirements. This stems from the formal, rigorous, mathematical approach employed for software specification.

Formal methods are important to the development of safety-critical software, such as that used in aircraft avionics and medical devices. Formal methods have also been employed in business-critical software, where, for instance, severe economic problems may occur if the software contains errors.

Even though this model can produce extremely reliable software, it has many disadvantages:

- The development of the software's formal model is both time consuming and expensive.
- Very few developers have any training in formal methods, and so require extensive training.
- Formal methods cannot easily be used as a means of communicating with the customer and with non-technical team members.

Remember that expert training in formal methods is needed to employ this process model. Formal methods also do not replace traditional methods: they should be used in conjunction with each other. Importantly, employing formal methods does not mean that the software developer need not adequately test the software.

The unified process

This *unified process* is also known as the *Rational Unified Process* (RUP), after the Rational Corporation who helped in the model's development. The Rational Corporation also develops CASE tools to support the use of the model.

The unified process is a unification of the various early object-oriented analysis and design models proposed in the 80s and 90s. It is an attempt to combine the best features of these various models which initially resulted in the *unified modelling language* (UML). The UML has become the standard diagrammatic language for modelling object-oriented software.

While the UML provides a modelling framework for developing object-oriented software, it does not provide any process model. This led to the development of the unified process, which is a process model for developing object-oriented software, and uses the UML as its modelling language.

The unified process is an incremental software process that is architecture driven, focuses on mitigating risk, and drives development through using use cases. Being architecture-driven, early iterations focus on building the portions of the software that will define the software's overall architecture. Focusing on risk, early iterations also focus on developing the high-risk portions of the software. Software

development iterations moves through five phases: inception, elaboration, construction, transition and production. These phases cannot be directly mapped on to the generic process framework activities: rather, each iteration contains some of the framework activities.

The *inception* phase is concerned with project feasibility: what should the software do, in broad terms rather than specifics, and what are the high risk areas? Should the development go ahead? Inception is usually a short phase, often having no more than one iteration. Little development usually occurs during the inception phase, but the software requirements are discovered using use cases (communication), and a small subset of these requirements (those with high risk, and which focus on the software architecture) are fleshed out (communication and planning).

Programming begins during the iterations of the *Elaboration* phase. Each iteration develops the requirements fleshed out in the previous iterations (modelling and construction), and chooses more requirements to flesh out (communication and planning) for development in the next iteration. The elaboration phase completes once all of the requirements have been fleshed out. However, this does not mean that communication and planning activities stop and do not occur in later phases: there is always constant communication with the customer and an understanding that requirements may change.

Much of the construction activity occurs in the iterations of the *construction* phase. While the iterations of the elaboration phase each had at least one meeting in which some use cases are fleshed-out and selected for development in the next iteration, all the use cases have already been fleshed out when the construction phase begins.

The *transition* phase contains the initial portions of the deployment activity: the software is given to the customer for evaluation (called *beta testing*, which we will discuss in ???). The customer's feedback will cause the software to be modified as required, and thus the transition phase includes some communication and construction activities. The *production* phase includes the final portion of the deployment activity: the software is now being used by the customer, and is monitored and supported by the software engineer.

The transition phase employs a technique called beta testing. Beta testing occurs when the software is given to the user to allow them to use the software and uncover any defects and deficiencies. There should be a formal communications framework for the customer to report their findings to the developers, and for the developers to assess these reports and to determine how to proceed.

Agile process models

Many of the process models we have just discussed have a perceived weakness: a lack of acknowledgement in the importance of managing change in the software life-cycle, and an over-emphasis on the process, tools, and documentation associated with them.

Agile process models were developed as a way to avoid these weaknesses. The *Manifesto for Agile Software Development* states that the core values of agile process models are:

- Individuals and interactions over process and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The motivation for this manifesto is that it is difficult to predict how software systems, the teams that develop them, and the context in which the software is used, evolve. The market conditions in which the customer wished to use the software could change, and the customer's needs will evolve to meet these new conditions, changing the software requirements. We may not even be able to decide on the requirements when the development work commences. Software engineers and their development methods must be *agile* enough to respond to all of these changes.

The customer is important in agile development. There must be effective communication between the customer and the developers in order to properly understand what it is that the customer needs.

The customer usually also works closely with the development team, allowing the developers to more fully understand their requirements, and allowing the customer to more fully understand the state of the software.

Apart from stressing closer communication between customer and developer, agile models also stress better communication between the members of the team creating the software. The most efficient form of communication is considered to be face-to-face communication, rather than documentation.

While agile development is strongly driven by the customer, it also recognises that any plans laid out to meet their requirements may change. This generally means that agile process models use an incremental / evolutionary approach to development, delivering multiple increments of the software to the customer. This allows the customer to have working software, to evaluate the software, and to ultimately allow the developers to more effectively respond to the customer's requirements.

Apart from not adequately dealing with change, prescriptive models also do not necessarily deal well with the differences between people.

For instance, people differ in the skills they have, and the levels in which they have these skills; they differ in how well they communicate, and in which mediums they communicate the best in (verbal or written, for example).

A process model may deal with the differences between people either with discipline (i.e., there are no options other than to follow the process activities as outlined by the process model) or with tolerance of these changes. While this is clearly a continuum, prescriptive process models can be characterised as choosing discipline over tolerance, and agile models as choosing tolerance over discipline. There is a trade off involved in shifting from a tolerant to a disciplined process model, and *vice versa*: while tolerant models are easier for developers to adapt to, and hence more easily sustainable, disciplined models are apt to be more productive.

The features of agile process models

The key features of an agile process model can be summarised as follows:

- The software itself is the important measure of the team's progress, rather than documentation.
- The development team has autonomy to determine how to structure itself, handle the development work, and apply the process model.
- Adaptability to change comes in large part through delivering software incrementally.
- Adaptability also comes from frequent delivery, so that customers can more easily examine the software and provide feedback.
- The process is tolerant: it is adapted to the development team's needs.
- Software is important, documentation less so: this means that design and construction are often heavily interleaved.

Extreme programming

We will discuss the most widely used agile process model, *extreme programming*. Extreme programming is an object-oriented development approach and provides four framework activities: planning, design, coding and testing.

Planning

Planning begins by creating *user stories*, which are similar to *use cases*, which we will cover in depth in ???. User stories relate how the software will be used, and what functionality it will provide. The customer then prioritises these stories. The development team, in turn, determines the amount of development time required to develop the story.

The stories are grouped to form deliverables. These are the deliverables that will be given to the customer at each increment. Each deliverable is given a delivery date.

Importantly, the *project velocity* is determined at the end of the first increment: this is essentially the time take to develop the number of stories that were delivered in the first increment. This can be used to better estimate the delivery times for the remaining increments.

> ## Note
>
> New stories can be added at any time. The new stories are prioritised, and then added to an appropriate deliverable based on this priority.

Design

The design activity in the extreme programming process focuses around *class-responsibility-collaborator* (CRC) cards (see ???), which the developers use to organise the classes that need to be implemented in the software. These cards are the only design documentation produced using this process model.

When the appropriate design is difficult to decide upon, a prototyping method is employed. The design is quickly prototyped to evaluate the risks associated with it, and to estimate the required time needed to implement the story. This prototyping is called a *spike-solution*.

Extreme programming encourages developers to reorganise the internal structure of the code (without altering the program's behaviour) in order to make developing the software easier, and to make it less likely that bugs will be introduced in future work. This process of reorganisation is called *refactoring*.

> ## Note
>
> Because of the refactoring, the CRC cards often need to be modified. The design should not be viewed as something set in stone, but as something flexible, that changes as the software does.

Coding

Extreme programming has a number of distinctive coding practices. First, before coding begins, the process model recommends developing unit test cases to test each of the stories being developed in that increment's release. The developers then code towards satisfying those unit tests. This also allows the programmer to better understand how the code will be used and how the code should be implemented.

The second distinctive feature is *pair programming*: all code is written by pairs of programmers, with one developer programming and the other ensuring that the code follows an appropriate coding standard.

Testing

As just mentioned, unit tests are written before coding begins. It is also recommended that a framework is in place to run all the unit tests, allowing them to be easily and repeatedly run. This also allows for software testing to occur on a daily basis.

Apart from unit tests, *acceptance tests* are tests defined by the customer. They focus, by their nature, on the functionality visible to the customer, and will ultimately derive themselves from the stories used to develop the software.

Advantages and disadvantages

Extreme programming has the following advantages:

- Extreme programming is an incremental process model, and so the customer will have working software very early.

- The customer works closely with the developers, so the developers have a better understanding of the software requirements.
- Pair programming allows for quality checks of code as programming happens.
- Extreme programming has a strong focus on accepting changing project requirements. Extreme programming has the following disadvantages:
- Much time might be spend re-coding the software, rather than focusing initially on a better design.

Differences in life cycle models and inconsistent use of terminology

There are no "officially" agreed set of terms used in descriptions of software development. Likewise, there is no standard set of stages in the waterfall-model (and many other models). When reading texts from difference sources, and when speaking with information systems professionals, it is wise to bear in mind the meanings of the terms being used, and if necessary clarify how the terms are being used.

Such differences are not a bad thing — they are a result of both the relatively small time that information systems and software development has been studied for, and reflect the dynamic viewpoints and interpretations of what we have learnt about software development to date.

Note

Do not take for granted that a particular book your are reading or person you are speaking to has the exact same understanding of the waterfall model, or of a term such as, say, "system design", that you have in mind.

Avoid having a single "correct" version of a life cycle model, stage or deliverable, especially since the models and deliverables should be adapted to fit the needs of the development team and project.

A useful learning activity to apply to each different reference source you use is to create a summary of the terms used and the model stages. As you work with different sources note the differences between the sources. Your own understanding will improve as you attempt to build an inclusive and general picture of the stages and deliverables of different views of the software life cycle.

Computer Aided Software Engineering: CASE

The role of tools in engineering systems development

Each activity in a process model has tasks to be performed and deliverables to be produced. Various software tools have been developed to support software developers using a particular process model. These tools support project management and monitoring and the use of any special techniques required by a particular process model, such as the UML. CASE (Computer-Aided Software Engineering) tools can also support the integration of several different deliverables to provide a consistent, overall model of the system being developed. Such a tool facility is usually said to be performing consistency or integrity maintenance and checking. Although there are simple, single-technique tools (such as for the UML), sophisticated tools exist to provide integrated support for many parts of a process model.

CASE tools

There are a wide range of software tools being used to support information system development. Some examples include:

- programming language compilers and debuggers
- project planning and costing applications

- help-file author applications
- version control support systems
- program code generators and “intelligent” program editors
- fully integrated suites supporting construction and consistency checking between multiple modelling techniques

The most sophisticated of CASE tools is the last in the list — the fully integrated tools supporting multiple modelling techniques. Some authors argue that any software application that helps in the system development process is a CASE tool, including:

- word processors
- spreadsheets
- general purpose drawing and painting applications.

We shall define a CASE tool as follows: a **CASE tool** is a software application (or integrated suite of application) whose sole purpose is to support one or more aspects of the software development process. The inputs and outputs of a CASE tool are either final system components, or artifacts whose purpose in some way supports system development and management.

The above definition excludes general purpose applications such as word processors and drawing packages.

It used to be common to make a distinction between three classes of CASE tool:

- Upper-CASE
- Lower-CASE
- I-CASE

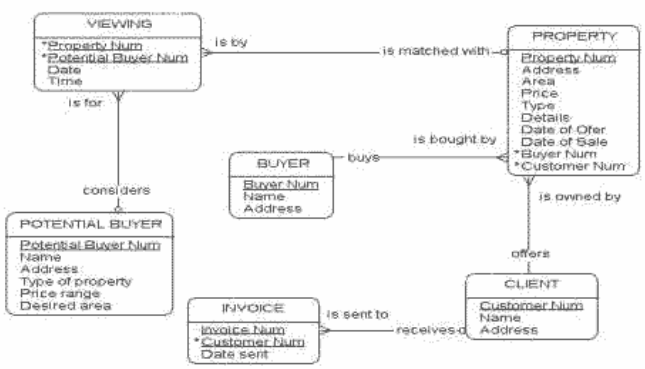
Upper-CASE

“Upper-CASE” refers to CASE tools that support the early, abstract, “higher-level” activities of the system development process activities such as requirements elicitation, system analysis, creating logical design models, and so on. Activities which actually involve detailed “lower-level” design and implementation issues such as coding and testing are covered by “Lower-CASE” tools.

Example of Upper-CASE tools are the SELECT-Enterprise CASE tool, and ArgoUML. They are sophisticated tools, supporting activities such as logical data modelling and data flow process modelling techniques.

The following screen shows the Entity-Relationship Diagram window of SELECT-Enterprise:

Figure 2.7. SELECT-Enterprise screenshot



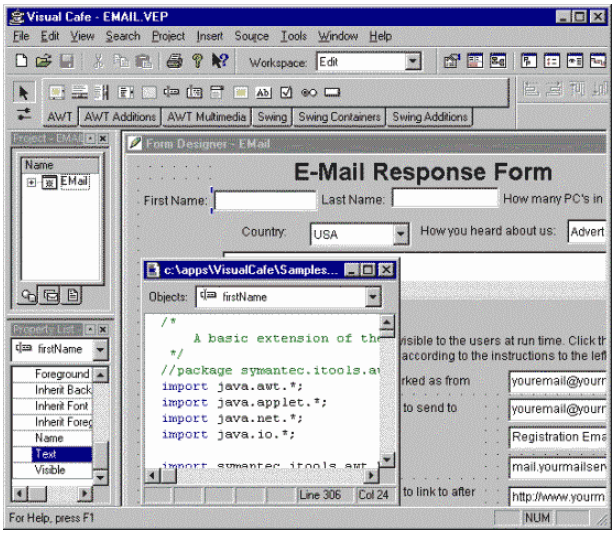
These notes will, however, mostly be making use of UML CASE tools, such as ArgoUML and Umbrello.

Lower-CASE

Activities which involve “lower-level” design and implementation issues are known as “Lower-CASE”. Examples of Lower-CASE tools are program code compilers and debuggers, interface building programs and so on.

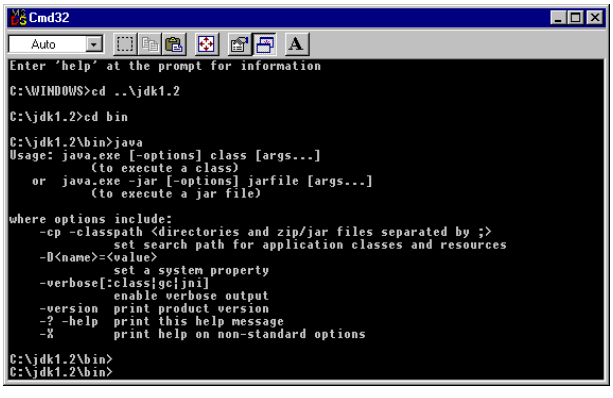
The following is a screen shot from a Lower-CASE tool — Symantec's VisualCafe environment for Java software development. In this screen we can see some Java program code and the corresponding interface objects on screen:

Figure 2.8. Java code



The following screenshot is another example of Lower-CASE — the command-line based java interpreter provided by the *Oracle Corporation* (previously owned by *Sun Microsystems*):

Figure 2.9. A command-line Java interpreter



I-CASE

The “I” in I-CASE stands for “Integrated”. The most powerful CASE tools offer functions to integrate a range of different kinds of models for a system, to manage a more complete picture of a system. Such integrated tools can check multiple models for consistency, in addition to assisting system modellers to handle the “cascade” of changes that occur when a change in one model requires a related change in another.

Each I-CASE tool tends to be developed to support a particular process model. Those CASE tools that do not provide integration between multiple models or products may be used for many process models — for example, regardless of the model, at some point a code compiler (or interpreter) will be needed.

IPSE — integrated project support environment

Integrated Project Support Environments are integrated applications to support project management. These usually provide some combination of project planning/scheduling, costing, version control of models and software components, and support for project management documentation and reporting.

Critique of CASE

Advantages of CASE

CASE offers a number of advantages for the development of systems:

- CASE tools allow for the faster development of models, and provide support for creating diagrams.
- Some CASE tools offer simulation of the system being modelled (e.g., given an event, possible system responses can be traced).
- CASE tools can be used to maintain a central, system dictionary, from which all models draw their components.
- CASE tools can help the team be more positive about improving models when errors or inconsistencies are identified.
- CASE tools provide consistency checking between different types of model.
- CASE tools can provide notation and documentation standards in and between projects.
- CASE tools provide navigation support between related parts of models and diagrams.
- They provide automated documentation and report generation from the system models and main dictionary.
- IPSE CASE tools increase the rigour of project planning and management, and support straightforward re-planning and response to unexpected events.

Disadvantages of CASE

Although there are clearly many advantages, there are a number of potential problems with CASE:

- They can be very expensive.
- Project staff require expertise (or training) on the software.
- A particular CASE tool based around a single methodology will force a project to commit more strongly to the methodology than they might wish to have done.
- There is a danger that high quality models and diagrams can lead to poor software development — the models may look impressive, but it is the quality of the system and not the diagrams that is fundamentally important.

Remember that having consistent models of the software does not guarantee that the software will meet the user requirements.

DO IT YOURSELF

What is a life cycle model?
What are the activities of the generic software process?
What features of the waterfall life cycle model separate it from other models? A discussion of this question can be found at the end of this chapter.
Describe the disadvantages of the waterfall model.
What are the main features of the prototyping life cycle model?
What are the points of the Agile Manifesto?
What is a CASE tool? Provide some examples of CASE tools. Try to also provide examples of tools that you may have used before.
Describe some of the advantages of CASE?

Answers

Discussion of Review Question 1

A life cycle model, or *software process model*, is a description of the best practices for engineering and developing software. It is usually broken down into stages, describing the deliverables produced during each stage, and describing the order or iterative cycles of the stages, and when each stage would be appropriate to perform.

Discussion of Review Question 2

There are five generic activities:
- Communication
- Planning
- Modeling
- Construction
- Deployment

Discussion of Review Question 3

The waterfall model breaks the system development process into a linear sequence of stages, each involving a specific activity. Each stage produces deliverables that become the inputs to the following stage. Once a stage has been completed it cannot be revisited — so decisions made in early stage are committed to and determine what happens at later stages.

Discussion of Review Question 4

The Waterfall model assumes that there is a full understanding and specification of the requirements at the beginning of the project, and that these requirements will not change during development. Since the deliverables at each stage is usually not software, it is easy for none-software related deliverables — and the bureaucracy surrounding these deliverables — to become the focus of the software development. Because of the linear nature of the model, problems identified in earlier stages become progressively more difficult and expensive to fix in later stages.

Discussion of Review Question 5

The prototyping model attempts to produce a small version of the final system that has only *some* of the final software's functionality. Prototype life cycle models allow for the developers and users to explore the uncertain requirements of the software system, and an analysis of the prototype can form the basis of a requirements specification for further development. Because prototypes are generated quickly and are incomplete, once the requirements are better understand the prototype is discarded and development restarted using another process model.

Discussion of Review Question 6

There are four points:
- **Individuals and interactions** over process and tools.

- **Working software** over comprehensive documentation.

- **Customer collaboration** over contract negotiation.
- **Responding to change** over following a plan.

Discussion of Review Question 7

CASE tools are software whose sole purpose is to support one or more aspects of software development.

Common examples include compilers and debuggers — if you are using Ubuntu or OS X, that will often be *GCC* for the compiler, and *GDB* for the debugger. There are also many CASE tools for planning and software costing, such as *Microsoft Project*. Other case tools include those for authoring help files, version control systems (such as *Subversion*, *Bazaar*, *Mercurial* and *Git*), code generators and program editors (such as *Emacs*, *Visual Studio* and *XCode*), and tools providing access to modelling techniques such as the UML (for example, *ArgoUML*).

Discussion of Review Question 8

Case tools allow for the faster development of various software models, and can sometimes even simulate portions of the software from these models. Some tools allow a central dictionary to be maintained where each model can take their components from. They can automatically check for consistency between various models, can provide a standard for notation and documentation, and can provide easy navigation between different portions of the models. They can support planning the activities of the software development life cycle, and can support the programmers directly with their development (by providing access to documentation, report generation, by keeping track of changes made to the source code, and so on).