Software Processes

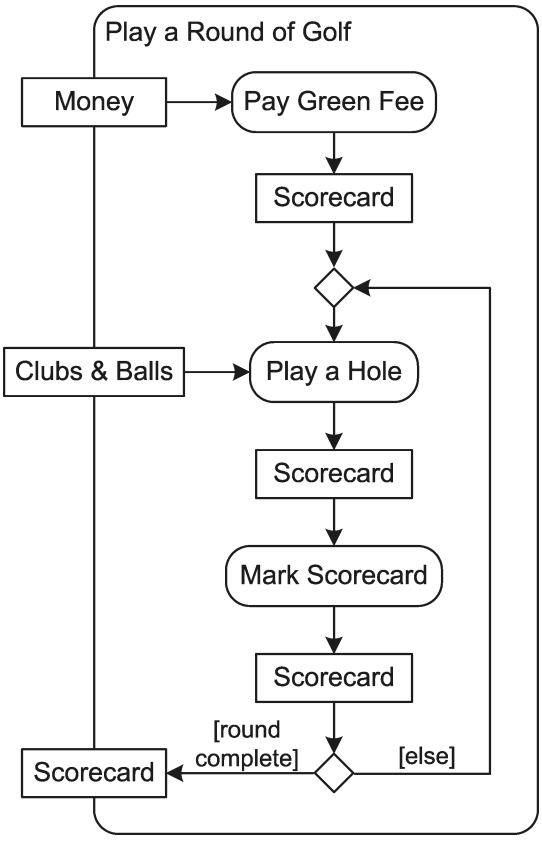
Processes and Models

A **process** is a collection of related actions that transforms a set of inputs into a set of outputs. For example, the process of making a sandwich transforms various ingredients, like bread, meat, vegetables, and condiments, into a single edible item. Steps in this process include spreading condiments on the bread, putting the meat on the bread, slicing vegetables and putting them on the bread, and so forth.

To describe a process we need to

- Specify the inputs to and outputs from the process as a whole;
- Specify that actions that constitute the process;
- Specify the inputs to and outputs from each action within the process (object flow);
- Specify the conditions under which actions occur and the order in which they occur (control flow).

There are various notations for describing processes, including UML activity diagrams [1]. For example, the UML activity diagram below shows a process for playing a round of golf.



Playing a Round of Golf

In UML activity diagrams, processes and actions are represented by rounded rectangles. Data and materials (*objects*) are represented by rectangles. An object icon on the boundary of a process icon means that the object is a process input or output. In this example, Money and Clubs & Balls are

process inputs, and Scorecard is a process output. Object icons connected by arrows to actions are action inputs or outputs. For example, Scorecard is shown as an output of the Pay Green Fee action, where it originates. It is subsequently shown as inputs and outputs of other actions, indicating that the arrows connecting these actions are object flows rather than control flows. The diamonds show where flows diverge (at *decision nodes* with boolean *guards* on their outgoing arrows) or converge (at *merge nodes* with several incoming and only one outgoing flow arrow).

Often, the actions that constitute the steps of a processes are themselves complicated enough to be regarded as processes. Hence actions in a process may themselves be shown in other activity diagrams, forming a hierarchy of diagrams that decompose a very complicated process into much simpler actions.

A **software process** is a process used in the creation or support of a software product. A **software lifecycle process** is a process depicting the steps that occur from software product inception through retirement from service—that is, a process that shows all the steps in the life of a software product. Lifecycle processes are good examples of very complicated processes with actions that are themselves processes. Variation in lifecycle processes comes not only from differences in the way their component actions are related to one another, but from the way that component actions are themselves decomposed as processes.

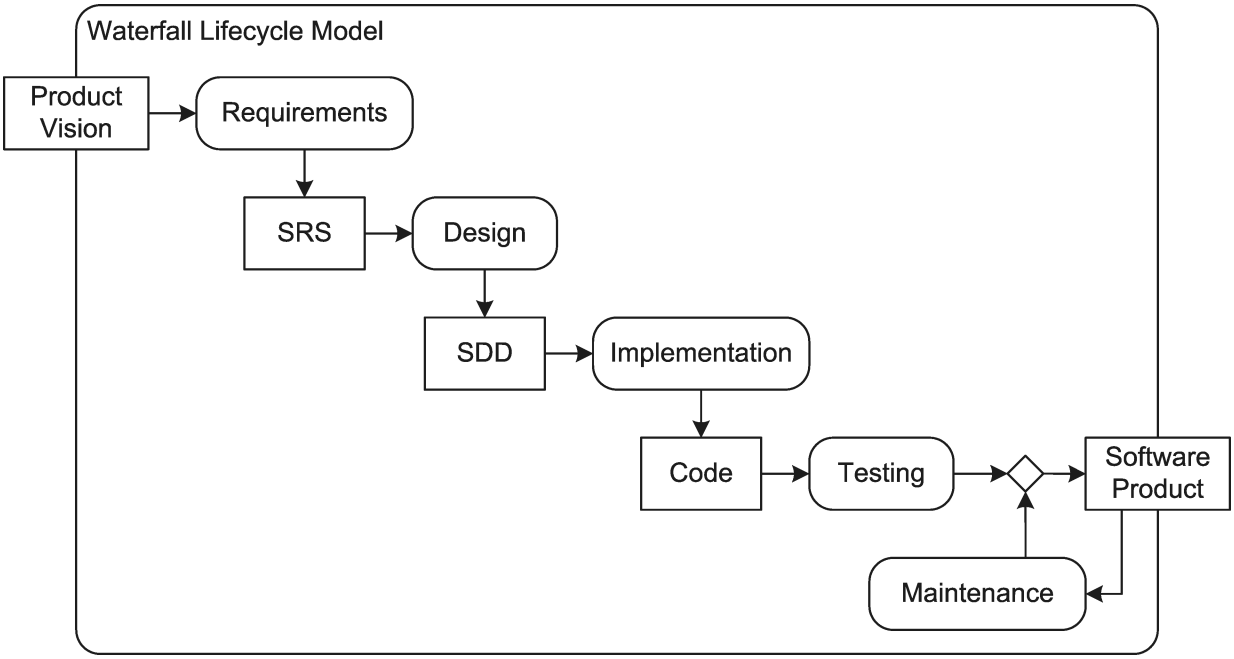
A **model** is an entity used to represent another entity, called the *target*. For example, a model railroad represents a real, full-sized railroad. There are many different sorts of models. For example, a parts diagram is a static 2D drawing that models how the parts of a real 3D thing look and how they go together. A model railroad is a 3D dynamic physical model of a (much larger) 3D physical thing. A graph of the equation $y = x^2$ is a 2D model of a function, which does not exist in space at all.

A **software process model** is a model of a software process. Usually software process models are static 2D diagrams (like UML activity diagrams) of the tasks of many people in the real world. They are usually quite abstract because they leave out many details. Nevertheless, software process models are very important because they either ***describe*** what actually happens in software development or (more usually) because they ***prescribe*** what is supposed to happen.

In the remainder of this chapter we present standard software lifecycle process models that prescribe what is supposed to be done when a software product is created and maintained. These models have a long history in software engineering and inform how we think about software development.

Waterfall Model

The waterfall lifecycle process model is the oldest prescription for the main tasks in the development of a software product. It was proposed by Winston Royce in 1970 based on processes in traditional engineering disciplines and it quickly became a staple of software engineering management. The following activity diagram shows a typical version of the waterfall model.



Waterfall Model

The process begins when a Product Vision document is provided to the developers. This document outlines the business goals for the product, its major features, constraints on its development, and so forth. The developers perform a Requirements action during which they create a **software requirements specification** (**SRS**) document that states in complete detail the behavior, appearance, and other properties (such as reliability and speed) that the product must have. The SRS is the input for the Design action during which the developers formulate a complete software design specification and record it in a **software design document** (**SDD**). The software design document is the input to the Implementation action during which the product's software is written. The Code is then passed on to a Testing action during which it is tested. Once Testing is complete, the finished Software Product is delivered to customers. Inevitably, defects are found and improvement suggested, so the product enters a Maintenance action that results in a new release of the product. Maintenance cycles continue indefinitely, though most products ultimately are retired from service (not shown in the diagram).

The waterfall process model is so named because work flows from one action to the next without returning to earlier actions, much as water goes over a series of waterfalls. Even projects that follow this model never wholly refrain from returning to earlier actions because it is impossible to always do every action completely and correctly the first time. However, the goal of the waterfall model is to complete each phase as thoroughly as possible so that return to earlier phases is minimized.

The waterfall model has several advantages.

- The software product is wholly specified and the project to create it is entirely planned early, so everything is predictable. This is especially valuable for large and complicated projects because managers need to know a big project's scope, delivery date, and budget up-front to make a good decision about whether to go ahead with it.
- It is easy to tell whether a project is on-time and on-budget by monitoring activity and comparing it to the plan. Managers can then take appropriate action to fix problems, adjust resource allocations, modify schedules, change scope, and adjust plans.

- Problems can (in principle) be found and corrected cheaply. For example, if a product requirement is incorrect, but it is not detected until the incorrect product specification has already been designed into the system, coded, and tested, it will be very expensive to fix because of all the effort wasted in building the mistake into the product, and all the effort required to fix it. On the other hand, if the requirements specification is done completely and correctly before any other work is done, and all incorrect specifications are found and fixed, then no time or effort will be lost.
- The waterfall model emphasizes production of complete and correct documentation at every stage. If people leave the project, or if new people must be brought in to augment the team, documentation is available to bring them up to speed.
- The waterfall model divides development work into distinct and relatively independent phases that can be performed by independent teams. This allows organizations to form teams of specialists to carry out tasks efficiently. Furthermore, because projects have detailed plans, these teams can be scheduled into different projects at the right times, maximizing resource utilization.

The waterfall model also has several disadvantages.

- The waterfall model relies heavily on being able to produce complete and correct product specifications that do not change appreciably during the project. But this is usually impossible, for many reasons that we will discuss when we consider requirements later on. Without stable requirements, all the advantages of predictability vanish.
- Even when requirements are stable, it is almost impossible to make them complete and correct. People are simply unable to envision complicated products and anticipate how all their features will work and interact, what problems will arise, and so forth. Making complete and correct designs is similarly difficult. This plays havoc with plans and leads to many defects in product specifications and designs that are not detected until much later in the process when they are very expensive to fix.
- Producing and maintaining all the documentation needed for the waterfall model is enormously expensive.
- Passing a product from team to team of specialists during development means that every team must study all the project documentation to understand what needs to be done. In contrast, a single team that carries the product through its life cycle is relieved of this heavy burden. Hence the advantages of using teams of specialists is reduced because of the communications overhead.
- The waterfall model typically uses many people in large teams who must coordinate their activities using a lot of documentation over a long period of time. This requires a lot of management. The upshot of this approach is high costs. Processes with a lot of documentation and management overhead (and attendant costs) are called **heavyweight** processes.
- The waterfall model does not deliver a product until completion of the development project, which may take several years. Often it is only then that bad product specifications, design problems, inadequate testing, and so forth, come to light. Furthermore, clients have not been able to use the product in all that time, and they may decide once they get it that they no longer need or want it.

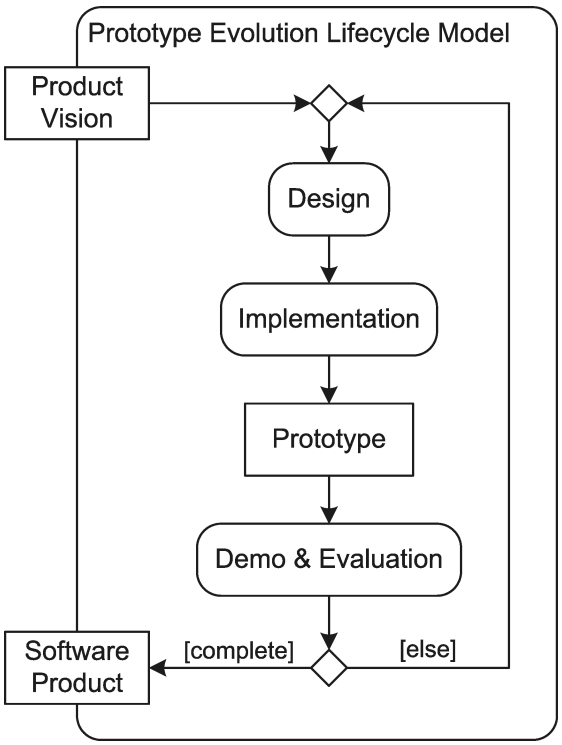
Is the waterfall software process model a good one or not? For several decades after the birth of software engineering, the waterfall model was the only defined life cycle process. During that time, waterfall model projects succeeded about a quarter of the time, failed completely about a quarter of the time, and had middling success the rest of the time. Success rates gradually increased and failure rates gradually decreased, but even today, waterfall projects fail quite often.

As noted, one key to the success of waterfall projects is the stability of requirements: this process only works with very well-defined and unchanging product specifications. Another is the size and importance of the product: the overhead costs of the waterfall process are not justified for small and simple products or for products that do not have stringent safety, security, or reliability requirements. The consensus is that a waterfall lifecycle process should only be used for fairly large projects with stable requirements or stringent safety, security, or reliability requirements.

Prototyping

One way to help overcome some of the problems of the waterfall life cycle model is to use prototypes to explore product and design specifications. A **prototype** is a working model of some or all of a finished product. For example, product specifications might be explored by mocking up a user interface for part of a product and showing it to users. Users can then get a much better idea of what they will and will not be able to do with the product and how it will work. This helps them decide what is good and bad about the envisioned product. Also, prototypes are easy to change, so several can be put together or changed rapidly to explore product ideas. Developers can also use prototypes to try out program design alternatives to help them make good design choices.

Prototypes may be **throwaway prototypes**, which are made just to help nail down specifications and then disposed of, or **evolutionary prototypes**, which are modified into a final product. Either kind of prototype can be incorporated into any software life cycle process. Prototyping can also be used as the basis for a life cycle process, such as the one pictured below.



A Prototype Evolution Process

This process begins when the team receives a Product Vision. The team makes an initial, very simple product design for a first prototype, then implements it. The team presents the Prototype to users and collects reactions and ideas in the Demo & Evaluation step. Based on this feedback, a new cycle of Design, Implementation, and Evaluation occurs. Cycles repeat until the Software Product is complete. As we will see, this process closely resembles incremental and agile processes, which we will consider more thoroughly later.

Prototype evolution has several advantages.

- Changes to product specifications are easy to handle, and customers are more likely to get what they want.
- Customers can get useful software very quickly.
- There is typically not a lot of documentation or management oversight required (such processes are called **lightweight** processes).

The disadvantages of the prototype evolution process are as follows.

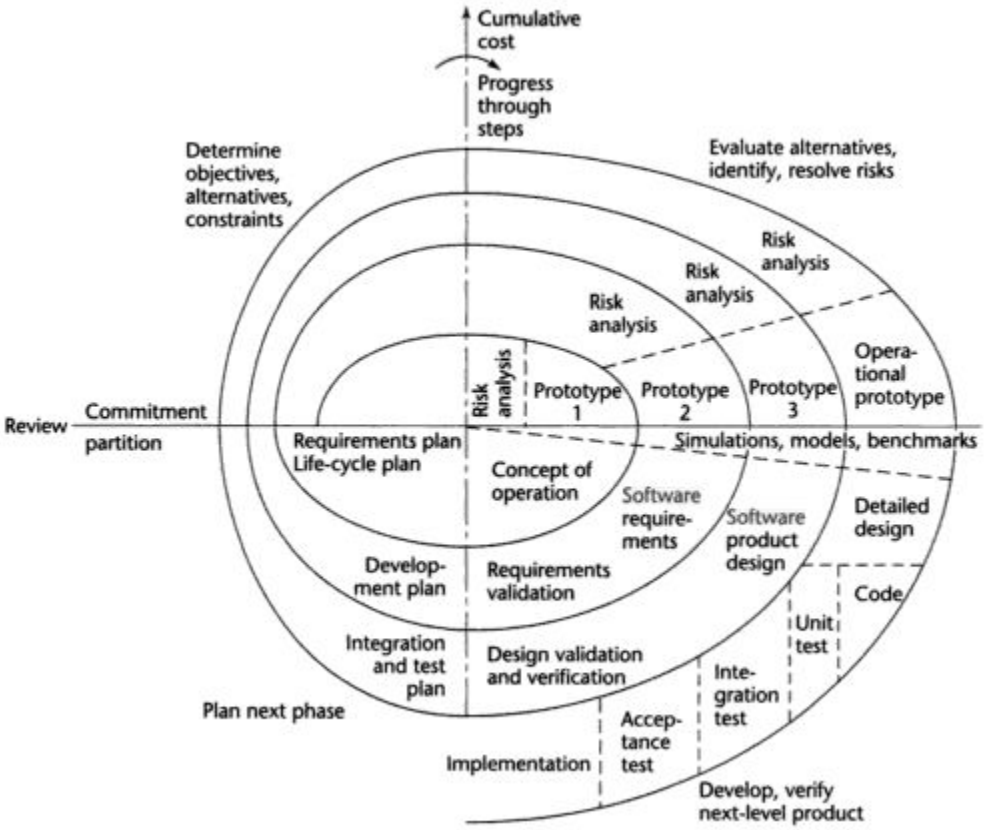
- It is very hard to predict when an adequate product will be finished and how much it will cost.
- The design of the finished product may be very bad because it may have evolved chaotically, so the product may be unmaintainable.
- There is little discipline in this process, so quality control may be lax, resulting in an unreliable or very buggy product.

Although prototype evolution is initially very attractive to customers because it is so responsive to their needs and desires and delivers products so quickly, lack of predictability, quality, and maintainability often cause products developed in this way to fail.

Risk Management

In software engineering, a **risk** is an occurrence with negative consequences. For example, losing the latest copy of a piece of software, losing a team member, finding a serious design flaw that causes a major program restructuring, or mis-estimating how long it will take to write some code, are all risks. **Risk management** is the practice of identifying, analyzing, and controlling or mitigating risks. Risk management can (and should) be incorporated into any life cycle process.

One well known life cycle process model incorporates risk as a fundamental activity. The spiral life cycle process model was proposed in 1988 by Barry Boehm [2]. The **spiral model** proposes a cycle driven by risk management. Each cycle begins with consideration of objectives of the next period of work in the project, alternative solutions (such as alternative product specifications or product designs), and constraints imposed on the project. The next step is to evaluate the alternatives in light of the objective and constraints, which leads to identification of risks. These risks are analyzed and steps are taken to remove or lessen them. These steps might involve prototyping, simulation, various requirements elicitation techniques (discussed in a later chapter), etc. Commitments are then made about the next steps in light of risks: developers might decide to create only a part of a product in the cycle, pursue a phase of the waterfall model, and so forth. The committed activities are completed to end the cycle, and another cycle of the spiral begins. The spiral model is intended to be adaptable to any other life cycle process. The figure below shows how it can be used in a waterfall life cycle process (note that not every item in the diagram must be created; for example, not every prototype in the diagram must be created).



The Spiral Model

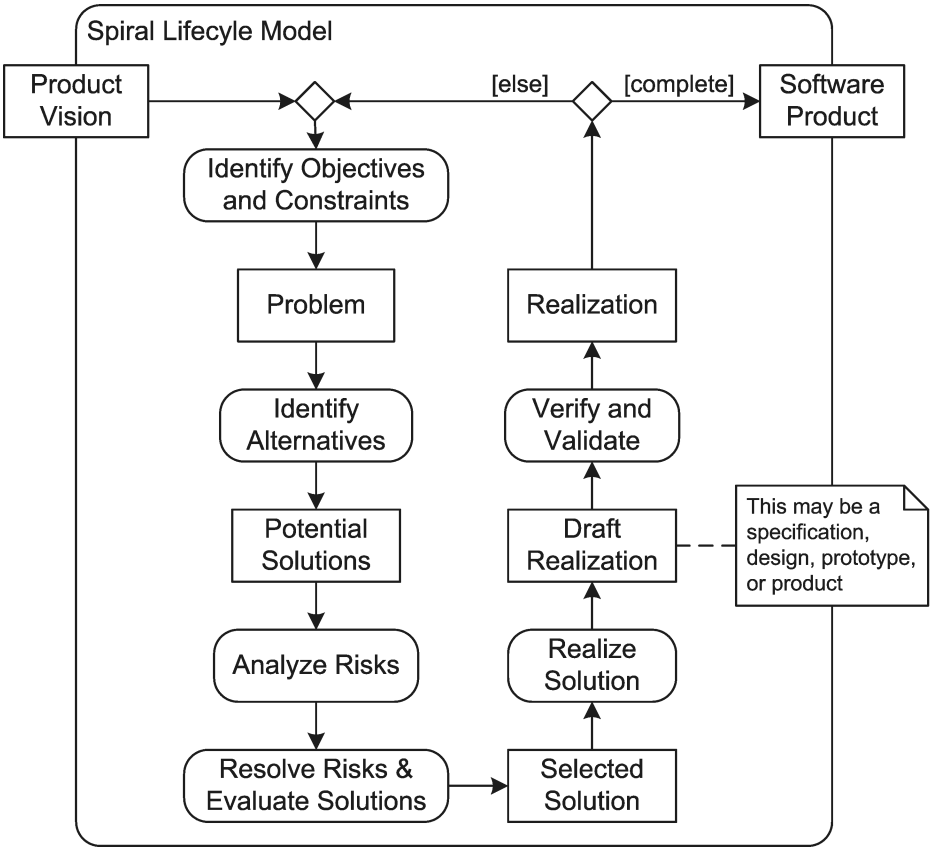
The spiral model is important because it demonstrated the central place of risk management in software development, and it showed how to incorporate risk management into several life cycle process models. It also emphasizes the importance of tailoring development practices to the project at hand, and how to decide what should be done and what should not be done. Specifically, if a risk is likely, then work should be done to remove or reduce the risk, but if a risk is unlikely, then it should not. For example, if a particular user interface is important for the success of a product, then considerable effort should be put into its design, but if it is not very important, then little design effort should be expended.

The drawbacks of the spiral model are the following.

- It is centered on risk management, but not very many people are trained in and good at risk management;
- It is very general and adaptable, and so it demands expertise in tailoring software processes that is not very common.

Iterative and Incremental Processes

The spiral model is also important because it is an early example of an iterative process. To see this, consider the following activity diagram representation of the overall structure of the spiral model.



Activity Diagram for the Spiral Model

As this representation makes clear, the spiral model contains many tasks that are repeated. This leads to the following definition.

> An **iterative process** is one that contains one or more repeated tasks.

For example, when debugging code one typically runs some tests, makes changes to the code, reruns the tests, makes more changes, and so forth. This process is iterative.

Iterative processes should not be confused with incremental processes.

> An **incremental process** is one that produces its output in parts or phases. An incremental process produces its output a bit at a time.

For example, when a customer orders the breakfast special, the chef can first cook the eggs, then cook the bacon, and then cook the hash browns. Alternatively, the chef can cook the eggs, bacon, and hash browns at the same time. The first process is incremental while the second is not.

Any software processes can be made iterative, and most are "naturally" iterative because it is difficult to get correct result after only one try (any process that works by trial and error is iterative). The waterfall process model is ideally non-iterative, but in practice, a certain amount of iteration is inevitable. The prototype evolution process is iterative, and the spiral process is iterative in its overall structure, particularly with respect to risk management.

The advantage of iteration is that it is the primary means of achieving work products of sufficient quality. As noted, it is usually very difficult to produce adequate specifications, design, code, documentation, and so forth on the first try, so iteration is necessary. The drawback of iteration is

that it so often involves **rework**, which is discarding or redoing previous work products. Rework is a major source of expense and delay in any process, and eliminating it is always desirable.

Software processes are sharply distinguished by whether they are incremental. The waterfall process model is not incremental while the prototype evolution process is incremental. The spiral process model, since it may be applied to any base process, is not inherently incremental, but may be used with an incremental process.

To understand why the waterfall process is not incremental, it is important to understand the difference between what economists call *intermediate* and *final goods*. Each action in the waterfall process produces an output, but the product specification and the software design are intermediate goods that are used in the production of the final good, the software product. So, while there are multiple actions each of which produces an output, one could not deliver the product specification and claim to have delivered a version of the software product. The cooking example might help clarify this distinction. While a restaurant using an incremental process to deliver breakfast to its customers might not make them happy, it can claim to be producing the breakfast in phases. In other words, the eggs alone can be considered a version of the breakfast.

It is important to note that the distinction between incremental and non-incremental processes is not always clear and depends on what constitutes the final product. Returning to the cooking example, can a sandwich without bread be considered a version of the final product?

An important iterative and incremental process is the unified process from Rational Software (now part of IBM) [3]. The **Rational Unified Process (RUP)** produces software products in parts called *releases* created during a *cycle*—each release is a product increment. The RUP is incremental because each release can be considered a final good (as opposed to an intermediate good). Each cycle has four phases called *inception*, *elaboration*, *construction*, and *transition*. Each phase is divided into iterations, and work in each iteration is divided into five workflows called *requirements*, *analysis*, *design*, *implementation*, and *test*. The work in each phase is summarized as follows.

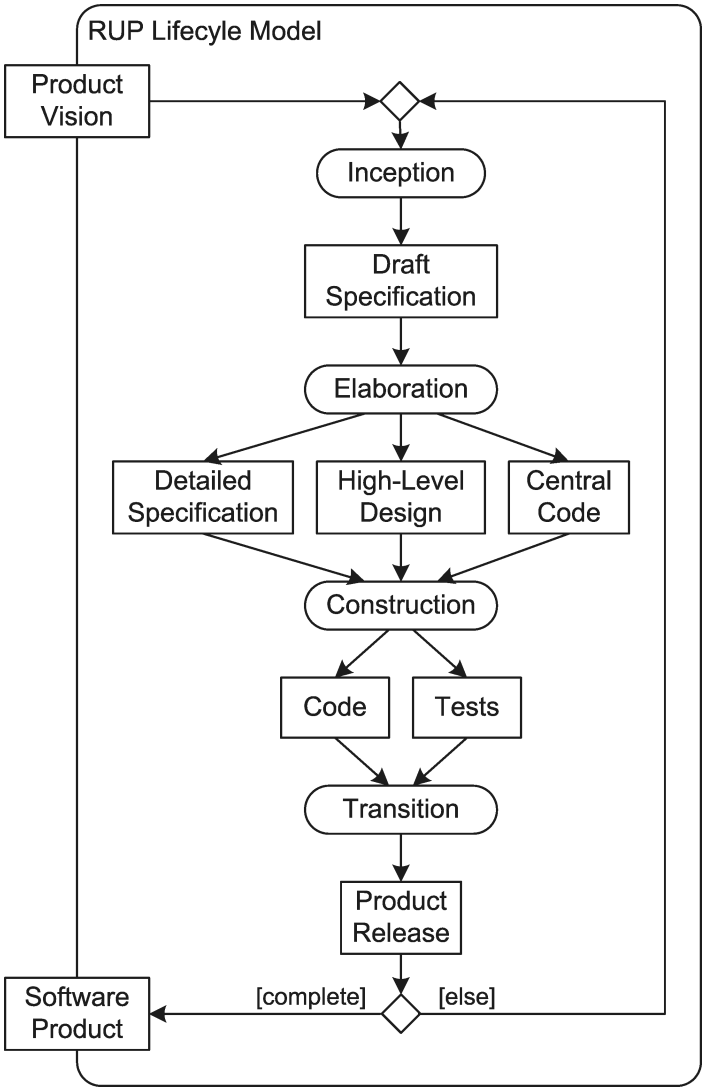
Inception —The product vision and business case are developed. Preliminary work is done on the product specifications and design.

Elaboration —The product specifications are set out in detail along with the high-level design of the system. Code is written and tested for core functions. Plans for the rest of the cycle are made.

Construction —The bulk of the code is written and tested. Designs are elaborated as necessary and corrections are made where needed.

Transition —The product is tested with customers and made ready for release.

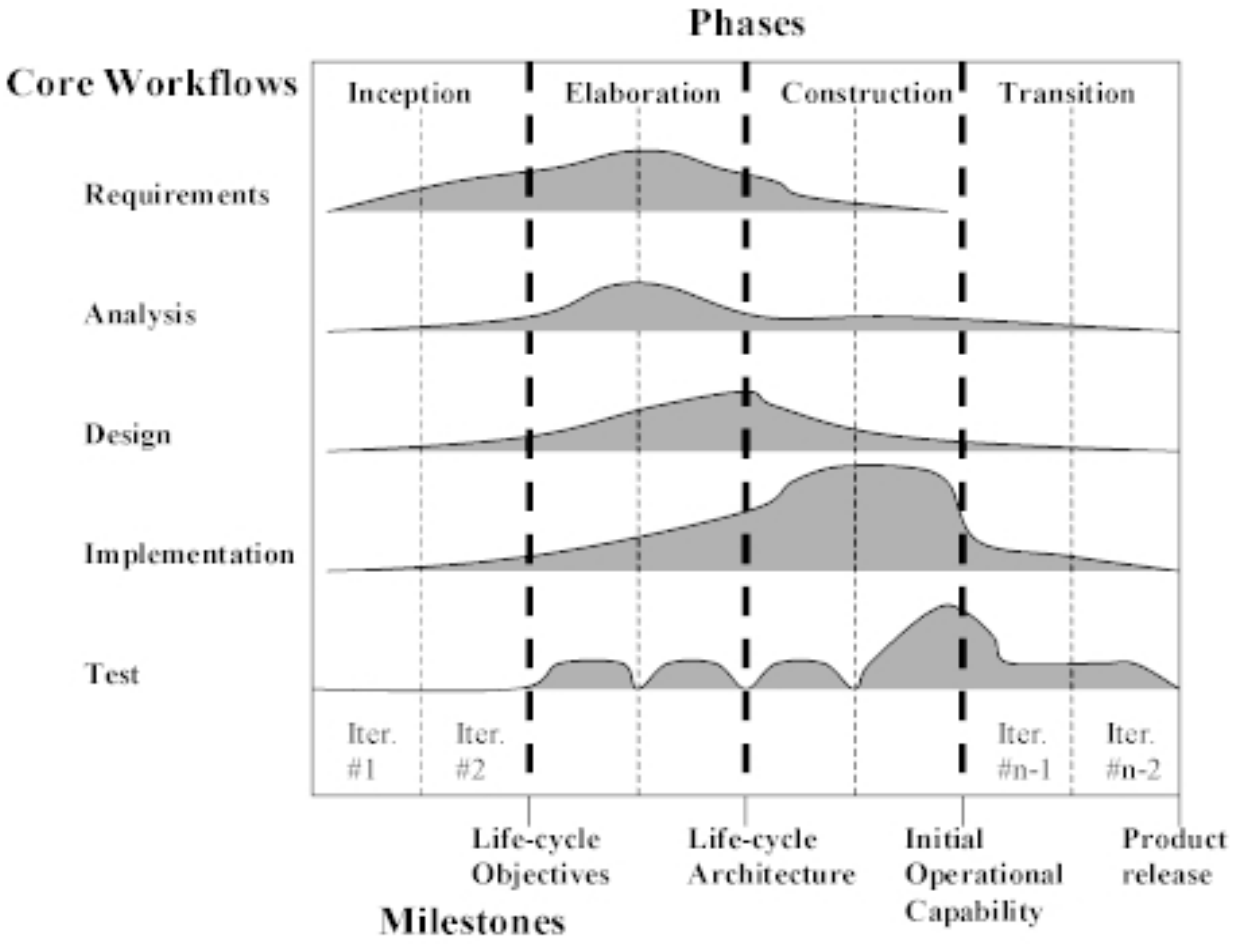
The phases can be represented in the following activity diagram.



Actions in the Rational Unified Process

Note that this diagram has a loop showing the cycles (increments) during which releases are produced. In addition, there are iterations during each phase that are not shown in this diagram. The RUP workflows are also not shown in this activity diagram.

The unified process is depicted below in a diagram showing phases, cycles, workflows, and iterations.



Workflows in the Rational Unified Process

The humps in the diagram show roughly how much effort is devoted to the various workflows during each phase.

The unified process has the following advantages.

- It is fully specified and comes with templates, checklists, and other process aids, and it incorporates best practices, such as risk management, so it is a complete package that developers can use to manage their life cycle process.
- It is adaptable to projects of different sizes with different needs.
- Because it is incremental, it lessens the risk of project failure, and it delivers usable product versions to customers fairly quickly.
- Because it is iterative, work is easier to schedule and monitor, work products can be modified to make them better, and risk management is supported.

The disadvantages of the unified process are the following.

- The unified process is complex and requires knowledgeable and experienced developers to use it properly.

- Although it is adaptable to different kinds of projects, it tends to impose a lot of documentation and management effort (it is a heavyweight process).
- Requirements change is not well-supported because development cycles tend to be fairly long, and requirements are assumed to be stable during a cycle.

The unified process has been adopted by many organizations successfully. It has also been the basis for many other life cycle process models that modify it in various ways.

Agile Processes

The waterfall life cycle process model, and various elaborations and refinements of it, dominated practice in software development from the 1970s through the 1990s. By the end of that period, it was clear that although there had been steady improvement in project outcomes, project failures rates were still too high and customers (and many developers) were very dissatisfied with the state of the software development enterprise. A group of influential developers decided to change this state of affairs and proposed a radical departure from traditional practice. Their philosophy was proclaimed in the *Agile Manifesto* [4]. The manifesto stated that its authors "had come to value

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

In other words, the writers of the *Agile Manifesto* wanted to change software engineering practice so that products could be created quickly and cheaply by streamlining development processes and practices and working more closely with customers. They also wanted to emphasize the "human" aspects of software development, taking into account teamwork, interesting work, and work-life balance.

Several agile methods have been proposed, the best known being Extreme Programming (XP), The Chrystal Methods, Dynamic System Development Method (DSDM), and Scrum. All these methods have similar life cycle process models with the following characteristics.

Incremental—All use an incremental process with short increments, ranging from a week to a few months. Short increments allow rapid response to changes in customer needs and preferences.

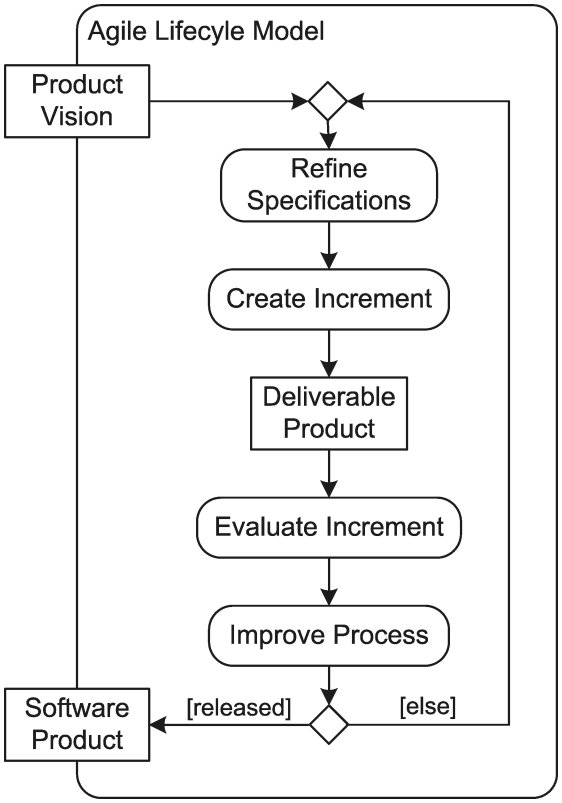
Customer involvement—All build-in close and continuous involvement with customers as a way to track customer preferences and promote informal communication.

Lightweight—All try to avoid documentation and project management tasks as much as possible to speed up work and avoid rework.

Test Driven—All try to avoid the problems that might arise as a result of frequent changes by using test suites that are created early in the process and maintained throughout.

In addition to their distinctive life-cycle processes, agile methods utilize many practices that accommodate change, impose discipline on work, and promote quality. We will discuss these practices later on when we discuss one agile method, Scrum, in detail.

The following diagram illustrates a generic agile process.



A Generic Agile Process Model

The process begins by formulating initial product specifications based on the Product Vision in the Refine Specifications activity. These specifications are the basis for designing, implementing, and testing a version of a Deliverable Product. This version is then evaluated by customers and other stakeholders in the Evaluate Increment activity. The work done to create the increment is also considered and used to make changes to the process in the Improve Process activity. If the Deliverable Product is deemed ready to release, then it is delivered as the Software Product. Otherwise, the Deliverable Product and the results of its evaluation are used to Refine Specifications and another cycle to produce a new product increment is begun.

Agile processes have many advantages, including the following.

- Product specifications can be changed frequently without disrupting the process.
- A version of the product is delivered to customers soon after development begins, and new versions with gradually increasing capabilities can be delivered frequently, if desired.
- Bad projects can be recognized and cancelled early.
- The process is lightweight, so much time and effort is saved.
- Waste and duplication of effort are usually greatly reduced.

Of course, agile processes have disadvantages as well.

- Customers and users must be involved throughout development but it is often difficult to get customers to commit so much time and effort.

- Although continuously refining the product design is a common agile practice, designs developed incrementally may not be very good, degrading product quality and increasing development effort.
- Agile processes are difficult to use on large projects because it is hard to coordinate the activities of many teams that are evolving different parts of the product in unpredictable directions, especially since minimal documentation is used.
- Although agile methods include planning activities, it is harder to predict the outcomes of agile projects.

Agile processes have grown in popularity since 2000, and they are on the verge of replacing the waterfall process as the dominant process used by developers. We will discuss Scrum, currently the most popular agile method, as our paradigm agile method in a later chapter.

References

1. Pilone, Dan. *UML 2.0 in a Nutshell, 2nd Edition*. O'Reilly, 2005.
2. Boehm, Barry. "A Spiral Model of Software Development and Enhancement," *IEEE Computer* 21(5), May 1988, pp. 61-72.
3. Jacobson, Ivar, Grady Booch and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
4. *The Agile Manifesto*. http://www.agilealliance.org/the-alliance/the-agile-manifesto/, accessed July 17, 2014.