CHAPTER 4

The Process

Design and programming are human activities; forget that and all is lost.
Bjarne Stroustrup, 1991

We use processes because we are afraid. We are afraid that

- The project will produce the wrong product.
- The project will produce a product of inferior quality.
- The project will be late.
- We'll have to work 80 hour weeks.
- We'll have to break committments.
- We won't be having fun.

Our fears motivate us to describe a process which constrains our activities and demands certain outputs. We draw these constraints and outputs from past experience, choosing things that appeared to work well in previous projects. Our hope is that they will work again, and take away our fears.

Over the years, we have come to recognize that many successful processes, though they differ in their particulars, seem to have a common shape. This chapter describes that shape. It provides a framework within which to build a process that may put our fears to rest. That framework is called the Rational Unified Process (RUP).

Rights

You, gentle reader, are most probably a software engineer. You are a software engineer because you like to write computer programs. You derive enjoyment and fulfillment from writing high quality software that serves your customers and employers well.

However, in order to do your job well, you have certain rights and needs. Ron Jeffries and Kent Beck outlined these rights as follows:

The Developer Bill of Rights

- *You have the right to know what is needed, via clear requirements, with clear declarations of priority.*
- *You have the right to say how long each requirement will take you to implement, and to revise estimates given experience.*
- *You have the right to accept your responsibilities instead of having them assigned to you.*
- *You have the right to produce quality work at all times.*
- *You have the right to peace, fun, and productive and enjoyable work.*

Some of you who are reading this book are customers of the developers. Most likely you are project or product managers who need the product that the developers are producing. You too have certain rights and needs. Kent Beck outlined these rights too:

The Customer Bill of Rights

- *You have the right to an overall plan, to know what can be accomplished, when, and at what cost.*
- *You have the right to see progress in a running system, proven to work by passing repeatable tests that you specify.*
- *You have the right to change your mind, to substitute functionality, and to change priorities.*
- *You have the right to be informed of schedule changes, in time to choose how to reduce scope to restore the original date. You can even cancel at any time and be left with a useful working system reflecting investment to date.*

These bills of rights are profound documents. If we could find a way to guarantee these rights, our fears would be greatly diminished. It is the job of a software process

to provide that guarantee. Any process that violates or ignores one or more of these rights, is doomed to fail.

The Goal

The goal of a software process is the production of software. Software that works, software that is on time, software that is within budget, software that can be maintained, software that can be reused. If this goal is met while preserving the rights of the developers and customers, then the process is a success.

Minimizing Intermediate artifacts.

It may be necessary to produce something other than software in order to eventually produce the software that is our goal. It may also be necessary to produce some artifact that acts to support the software that is our goal. However, such intermediate artifacts are not the goal of the process. They are, at best, a means to an end. They also represent a cost. A good process will decrease the demand for such intermediate artifacts to the barest possible minimum.

When constructing a process from the RUP framework, always remember to keep your eye on the goal. It is far too easy to become focused upon the intermediate artifacts and to forget that our goal is the production of software.

The Value System

A process achieves its ends by promoting certain values. Practices that support those values are compatible with the process. Practices that violate those values are rejected from the process. The values that we hold to are ages old. Kent Beck names them: Communication, Simplicity, Feedback, and Courage.

Communication.

Most of the bad things that can happen to a project are the result of miscommunication. If a project is late, it is because there was a miscommunication between those who managed the schedule, and those who executed the schedule. If the quality of the product is low, it is because of a miscommunication between those who needed the quality, and those who ensured it was present.

A good process *facilitates* communications. It provides the channels between the parties that need to communicate, and indicates the form, purpose and goal of that communication. Intermediate artifacts may be needed to achieve this facilitation, however a good process does not reduce communication to the blind production of intermediate artifacts. Communication takes place between *people*, documents are secondary.

Simplicity.

A process that is too complex will fail. Simplicity is a value to be intensely defended, both in our software, and in our process. We will not add activities or artifacts to our processes unless the need for them is critical. We will regularly sweep through our processes and remove accumulated complexity. Anything that cannot be completely justified, is eliminated. A process description should always look too small.

Feedback.

Dijskstra said it best: "*...as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than to try to ingore them, for the latter vain effort will be punished by failure.*"[1]. We do indeed have very small brains. If we try to do too much without checking our results, we will fail. Thus, we value small steps, where each step is tested for accuracy before taking the next.

A good process uses a method similar to that of the scientific method. Every step is at first nothing more than a hypothesis. Every hypothesis is tested by physical experiment. When enough experiments have been performed to establish the hypothesis as correct, the next step can be taken. This is the foundational motivation for all iterative methods.

Courage.

Daryl R. Conner says: "Its the People, stupid."[2] Alistair Cockburn says that a process can have only second order effects. First order effects are due to the people.

Of all the values, this is the most important. In many ways a process is put in place to make up for human failings. Yet, if we are to be successful, we must still have the courage to put our faith in people as opposed to a process. The moment we come to

1. [EWD72], p3.
2. [DRC98], p 6.

believe that a process is more important to success than the people who execute it, is the moment we consign our projects to failure.

Thus, when we define a process, we must take great care not to dehumanize it. It should protect us from the most critical of failures. It should remind us of the most important activities. It should establish the fundemental rules. But it should not and, and indeed cannot, protect us from every possible human failing. To attempt such coverage lacks courage.

How much human error should a process attempt to cover? That depends upon the criticality of the project. If human lives are at stake, then the process will have to have a very high coverage indeed. I would expect that the process for developing the control systems for a nuclear power plant would check and recheck every possible failing that could endanger lives. On the other hand, if we are developing a program to manage the football pool at work, we would not expect the process to specify much coverage at all.

Alistair Cockburn has identified four different levels of criticality[1], Life, Essential Money, Discretionary Money, and Comfort. Each requires a different level of coverage by a process, and defines the boundary between foolishness and courage.

When building a process, depend upon people in every instance where the risk is acceptable. The cost of a few uncorrected non-critical human errors is less then the cost imposed by a process that tries to prevent them.

The Rational Unified Process

In its simplest form, RUP consists of some fundemental workflows:

1. Business Engineering. Understanding the needs of the business.
2. Requirements. Translating business need into the behaviors of an automated system.
3. Analysis and Design. Translating requirements into a software architecture.
4. Implementation. Creating software that fits within the architecture and has the required behaviors.
5. Test. Ensuring that the required behaviors are correct, and that all required behaviors are present.

1. [ARC97]

6. Configuration and change management. Keeping track of all the different versions of all the work products.
7. Project Management. Managing schedules and resources.
8. Environment. Setting up and maintaining the development environment.
9. Deployment. Everything needed to roll out the project.

These activities are not seperated in time. Rather, they are executed concurrently thoughout the lifetime of the project. As we see in Figure 4-1, not much code gets written early in the project lifecycle; but the amount is not zero. Late in the project, most of the requirements are known, but some new ones are still identified.

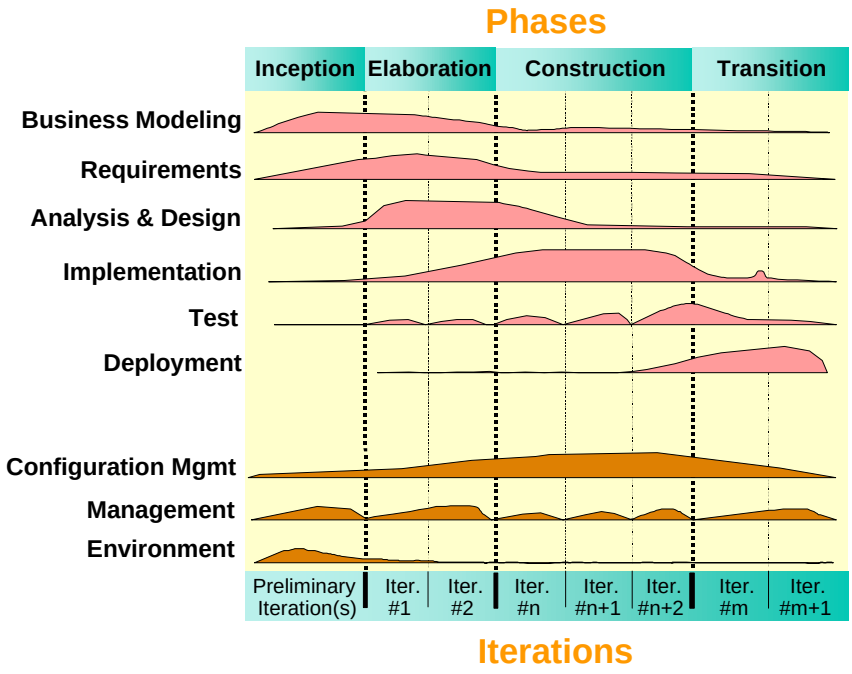

Figure 4-1
The Phases and Iterations of RUP.

Thus, as the project matures, the emphasis on certain activities increases or decreases, but activities are allowed to execute, and indeed do execute, at any time throughout the lifetime of the project.

Iteration

A project goverened by a RUP process moves forward in increments called iterations. The goal of each iteration is to develop some working software that can be demonstrated to all the stakeholders, and that the stakeholders will find meaninful.

The software developed by an iteration should cut through all or most of the major subsystems of the project. It should not be concentrated into a single subsystem. Each iteration represents an effort by each member of the team to build a small part of their part of the project and integrate those parts together.

The length of an iteration depends upon the kind of project we are working with. However, short iterations are to be desired over long ones. The shorter the iteration, the less time passes before the team gets feedback. Iteration lengths of one or two weeks are not too short for most projects.

In an iteration, all of the RUP activities are executed. There will be some business modeling in order to understand the needs of the business that will be fulfilled by that iteration. There will be some requirements analysis, to ensure we understand what behaviors the iteration must have. There will be some analysis, design, implementation, testing, etc. No activity is ever precluded from an iteration.

Estimation and Schedules

Iterations are not schedule milestones, and it is a mistake to manage a software project as if they were. Rather, the time taken to finish an iteration is measured, and then applied to the project as a whole. If, on average, it takes two weeks to finish an iteration, and if we know of 30 more iterations, then we know the project will run for another 60 weeks.

Initially, this kind of estimation is very inaccurate. But as iteration after iteration completes, it becomes ever more precise. Early in the project enough iterations will have been completed to give the project managers a very good indication of how to manage the project schedule.

The content and schedule of each iteration is negotiated by project management and development prior to the start of the iteration. If project management wants the iteration done by a certain date, then the developers must be able say how much work they can do within that time. Or, if project management specifies a certain amount of work that must be done within an iteration, the developers must be able to specify the end date of the iteration. Project management cannot specify both.

Even after such a negotiation, it may be that the iteration will finish later than scheduled, or with less work done than anticipated. In either case, this data is recorded and used to determine the next esitmates and schedule.

Phases

There are four phases to a RUP project: Inception, Elaboration, Construction, and Transition. These phases represent a certain emphasis to the activities within an iteration.

Inception. In this phase, the goal of the iterations is to help the project team decide what the true objectives of the project will be. The iterations will expore differnt possible solutions, and different possible architectures. They will be used to measure how quickly iterations can be done, so that the schedule can be calibrated.

It may be that all the physical work done in this phase is discarded. If the only thing that survives the Inception phase is the increased knowledge of the team, then the phase is a success.

Typically, however, several physical artifacts do indeed survive. Amongh these artifacts may be:

- A simple statement of major requirements, possibly in the form of use cases.
- A rough picture of the software architecture.
- A description of the project objectives.
- A very preliminary project plan.
- A business case for the project.

Life Cycle Objective Milestone. The Inception phase ends at the Life Cycle Objective Milestone. This milestone is crossed when the project team and stakeholders agree upon:

1. what the business need is, and what set of behaviors will satify that need.
2. a preliminary schedule of iterations.
3. a preliminary architecture

Though it may be common to wait till the end of an iteration to make these agreements, this is not strictly necessary. Once the agreements can be made, the project enters the Elaboration phase.

This event has no more significance other than that the project team and stakeholders have agreed what the project objectives are. Crossing the milestone is not necessarily

accompanied by any change in activities. Indeed, for the average developer, the day after the milestone ought to be very similar to the day before.

Some project managers may wish to schedule the end of the Inception phase. It should be understood however, that crossing the milestone is not a matter of certainty. It's simply a statement by the project team and stakeholders that the risk is low enough to make a resonably confident decisions about what is to be built and how long it might take. It shoudl also be understood that the time spent in the Inception phase is not a good predictor of the time that will be spent in other phases. Indeed, it is concievable that the Inception phase can be accomplished with no iterations at all, taking virtually zero time.

Elaboration. The iterations in the Elaboration phase will:

- Establish a firm understanding of the problem to be solved.
- Establish the architectural foundation of the software
- Calibrate and support a detailed plan of subsequent iterations.
- Refine the process and gel the team.
- Eliminate high risks.

The iterations produced in this phase are, on average, significantly less disposable than those produced in the inception phase. Each iteration should be adding new features to the growing body of software. Each iteratoin will also be adding new tests to the growing body of verification software.

During this phase, stakholders will see real progress against the project plan, and they will see the project plan becoming more and more stable and reliable. From iteration to iteration, confidence in the project and the plan will increase.

High risk development items are tackled early in this phase. The goal is simply to address these risks up front so that they don't catch the team later. Addressing the risky items up front also calibrates the esimates in a conservative manner.

Four artifacts are sure to be produced by this phase:

1. the growing body of software in the form of an architectural prototype,
2. the test fixtures and verify the operation of the software,
3. the use cases that describe the majority of system behavior,
4. a detailed project plan describing subsequent iterations.

Other artifacts that may also be produced are:

5. a preliminary user manual,

6. a software architecture description.

Life Cycle Architecture Milestone. This milestone marks the end of the Elaboration phase, and the beginning of the Construction phase. It is delineated by the ability of the project team and stakeholders to agree that:

1. the use cases describe the detailed behavior that will address the business need,
2. the chosen architecture will scale to support the full software development,
3. the major risks have been addressed,
4. the project plan is achievable, and will achieve the project objectives.

Construction. The iterations in the construction phase are not much different from the iterations of the Elaboration phase. Each iteration adds features to the software; features that the stakeholders care about, and give feeback about.

During this phase, one would expect the use case descriptions to stabilize to a certain extent; though in many project domains they will continue to change throughout the lifetime of the project.

Use cases are added, iteration by iteration, to the software. This continues until the stakeholders can reasonably make use of the system. The system may be far from complete at this point. Indeed, the earlier the customer can start using the product in anger, the better.

The artifacts that are produced during this phase are:

1. The software system.
2. The test fixtures.
3. The user manual(s).

Initial Operational Capability Milestone. Often called a *beta* release, this milestone marks the end of the Construction phase and the beginning of the Transition phase. This is *not* the end of the project, nor even close. Indeed, the project may be much closer to its beginning than its end. This milestone is crossed when the project team and stakeholders agree that:

1. the product is stable enough to be used,
2. the product provides at least some useful value,
3. all parties are otherwise ready to begin the transition.

Getting to this point as early as possible is important for a number of reasons. Transition of a small system is a lot less risky than transition of a big system. This is especially true if the big system remains operating except for the small part that the

nascent system can do. Developers will learn much more by watching the system in production use.

Transition. The iterations in this phase continue to add features to the software. However, in this case, those features a being added to a system that users are actively using. Clearly, the stakeholders and the developers will have to negotiate how often the production system can be upgraded. Short upgrade cycles are better than long upgrade cycles however.

The artifacts produced in this phase are the same as those produced in the Construction phase. The team is simply improving and enhancing the system towards the objectives that were set at the end of the Inception phase.

Product Release Milestone. This milestone marks the end of the Transition phase, and possibly the beginning of the next Inception phase. It is crossed when the project team and the stakeholders agree that:

1. The objectives set during the Inception phase (and modified throughout the other phases) have been met.
2. The user is satisfied (which may or may not be synonymous with point 1)

Summary

And so we see the RUP as defining processes that:

- Drive projects through many small iterations, each of which involves more or less the same kinds of activities.
- Have four major milestones which mark the boundaries between the four major phases.
- Produce software, and possibly other artifacts.
- Are estimated and planned based upon real measured progress.
- Adapt to change.

UML and RUP

Notice that the above description of RUP did not include anything about UML. This was intentional. It may be that many, or even most, processes derived from RUP will name specific UML artifacts as part of the deliverables of their iterations. However, this is not an essential part of RUP. The only essential outcome of RUP is software that satisfies the user.

dX: A minimal RUP process:

In the previous section we stated that the best process for a project is the smallest process that project could afford. In this section we shall present a very small RUP dreived process. The small size is consistent with both the scope of this chapter, and with a large number of projects. Because it is very small, I shall refer to it as *dX*.

The *dX* process has been used on several successful projects. The projects that have employed it have been highly productive, and disciplined; and the resulting software has been of very high quality. *dX* may make a good starting place for a company to begin to define their process. Many companies may find that *dX* is all they need.

***dX* Inception.** The workflows in *dX* during the inception phase are as follows:

1. Major use cases are written on index cards. Their descriptions are kept simple. They are written by the customer representative, with feedback from the developers. A customer representative *must* be part of the team at all times[1].
2. Simple prototypes of those use cases are created as potential throw-aways.
3. The prototypes are used to a) measure the development velocity of the team, and b) determine if the use cases have the right size and detail. We use this information to begin a project schedule, and to refine the use case process.
4. The prototypes are also used to explore several potential system architectures.

At the Lifecycle Objectives Milestone, which marks the end of inception, we have a good idea of the major use cases, the project schedule, and the beginnings of a system architecture.

***dX* Elaboration.** Most of the code produced in the previous phase is discarded. Some may survive, but only if the team agrees. It is in this phase that design and programming begin in earnest. The workflows in this phase are:

1. Use case cards continue to be written by the customer team-member.
2. The developers estimate the amount of work for each use case card, and write the estimate on the card.
3. The customer proiritizes each use case card, and write the priority on the card. High risk use cases are given priority in this phase.

1. Typically this representative will continue to carry out duites unrelated to the project; but will consider project related duties to be of highest priority. This person is there to serve the project team.

4. Iterations are planned by selecting an iteration duration, usually no longer than a week or so. The customer selects use case cards to develop in that iteration. The estimates on the cards must not total to more than the duration of the iteration.

   After an iteration is complete, it may be that the number of use cases implemented is different from the number estimated. A ratio called the load factor is calculated as: *actual / expected*. This factor is applied to the estimates of the next iteration.

5. Each iteration is analyzed and designed to fit into the system architecture indentified during inception. Analysis and design are carried out by the developers in design sessions. The developers may use UML diagrams, or CRC cards, or any other means of building analysis and design models that seems appropriate to them.

6. The design models are committed to code. In *dX* the rule is that two pairs of eyes must examine every line of code that is produced. Developers often accomplish this by working together in pairs, two to a workstation. Once committed to code, the analysis and design models fall outside the domain of *dX*. They may be discarded, or kept for posterity.

7. Testing is highly valued by *dX*; so much so that unit test code is written before the code it must test. Use cases are broken down into testable units; then developers alternate writing test cases, and production code. The tests and the code grow together at the same time.

8. Simplicity is also highly valued by *dX*. Designs and code begin as simple as possible; and no complexity is added unless mandated by a use case.

9. In *dX* a premium is placed upon the quality of the code, and how well it fits into the system architecture. Developers ruthlessly change any code that does not meet these standards. In *dX* code is considered extremely malleable, and the developers have no fear of changing it. The tests ensure that any problem introduced because of a change is caught.

10. Code is joinly owned by all teammembers. Any teammember may modify any module, regardless of who authored it.

11. Integration takes place at least daily.

***dX* Construction.** The Lifecycle Architecture Milestone is almost a non-event in *dX*. In most *dX* projects, the architecture and the project plan evolve together, and their quality improves together. Thus, the construction and elaborations phases of *dX* are nearly indistiguishable. The difference between them is simply in the stability of the architecture and the stability of the project plan. As the project moves into the Construction phase, the team works out a release schedule. That schedule is created by using the load factor, and summing up the use case cards needed for each release.

***dX* Transition.** In most *dX* projects, the Transition phase starts after the very first release. This usually takes place very early in the project. From now on the system will be live. It is also likely to be severely under-functioned. Thus, the old system will be kept running in parallel if at all possible.

The team will continue to plan out iterations and releases. Iterations accumulate until a new release can be made live. The shorter the release cycle, the quicker the team will recieve feedback from the running system.

***dX* Extras.** There are several other factors and rules imposed by *dX*. The process depends heavily upon the ability of the teammembers to communicate. Thus, an open workspace is required; allowing the teammembers, and the customer team-member, to be in close contact with each other.

In *dX* overtime is considered to be a failure of the process. Morevoer, two consecutive weeks of overtime are not allowed. The project plan is changed instead.

dX does not make explicit use of UML, or any other notation. The only intermediate artifact *dX* uses are the use case cards. This is not to say that engineers using *dX* cannot make use of UML when they need to create a model, or test some assumptions. It just means that *dX* does not tell engineers when such models ought to be created.

One might draw the conclusion that without UML artifacts to support analysis and design, that no analysis or design takes place in *dX* . That is however not the case. In *dX* analysis and design are accomplished in frequent design sessions. The decisions coming from those sessions are typically written down on index cards which are discarded once they have been committed to code.

While this may sound too informal for some projects; it has proven to be quite effective. Remember, a process does not have to be complex to be effective. Indeed, we want our process to be as simple as possible.

Do you need more than *dX*? Each project has its own particular needs. It may be that you need more formality, than *dX* supplies. If so, you can extend *dX* by carefully and sparingly adding practices and artifacts. However, it would be wise to do so only once you have the feedback loops in place to determine whether what you are adding to the process is beneficial or harmful.

Summary

RUP is a project framework that describes a class of processes that are iterative and incremental. RUP compliant process deliver functionality in small increments, each building on the previous, and each being driven by use cases rather than being the construction of a subsystem. RUP processes estimate tasks and plan schedules by measuring the speed of iterations relative to their original estimates. Early iterations of RUP driven projects are strongly focussed upon software architecture; rapid implementation of features is delayed until a firm architecture has been identified and tested.

We discussed a minimal implementation of RUP which we called *dX* . The principles and practices of *dX* were identified several years ago by Ward Cunningham, Kent Beck, Ron Jeffries, and a host of other developers and methodologists. They have used this process on several projects with significant success. Because of that success, they have gathered quite a following. They call the process Extreme Programming; or for short: *XP*[1].

1. [BECK99]

Bibliography

[EWD72]: *Structured Programming*, Dijkstra, Dahl, Hoare, Academic Press, 1972

[DRC98]: *Leading at the Edge of Chaos*, Daryl R. Conner, Wiley, 1998

[ARC97]: *The Methodolgy Space*, Alistair Cockburn, Humans and Technology technical report HaT TR.97.03 (dated 97.10.03), http://members.aol.com/acockburn/papers/methyspace/methyspace.htm.

[BECK99]: *Extreme Programming Explained: Embracing Change*, Kent Beck, Addison Wesley, 1999.

[KRUCHTEN99]: *The Rational Unified Process*, Philippe Kruchten, Addison Wesley, 1999