Implementing a Software Development Process

In our first article, The Problem with Software Development, we proposed the Rational Unified Process or RUP as a suitable process for software development projects. In that article we said that a process must be:

- Based on best practises
- Self-improving
- Easy to implement
- Acceptable by experienced software professionals
- Adaptable to any size of project

In this article, we look at RUP in more detail and justify it in the terms stated above. The following sections tackle each property in turn.

1. Process must be: Based of Best Practises

In the context, Rational define the following 6 best practises of software engineering:

1. Develop Iteratively
2. Manage Requirements
3. Use Component Architectures
4. Model Visually
5. Continuously Verify Quality
6. Manage Chan

One important feature of having a development process based on best practise is that if any organisation already has a set of defined best practises, they may readily be substitutes for the equivalents in RUP. This can sometime make speedy work of customising the process for a given organisation. This should not be done lightly since Rational have achieved considerably clarity of thinking in the manner in which they have already documented these best practises.

Let's spend a little time expanding on these best practises so that we are clear on what they mean.

1.1 Develop Iteratively

In RUP terms, an iteration in a self-contained, mini-waterfall development cycle with elements of:

- business modelling
- requirements gathering, documentation and prioritising
- object-oriented analysis and design (OOAD)
- implementation or coding
- testing and
- deployment

This is also supported by project management, configuration management and setting up the development environment.

The subject of each iteration is a sub-set of the total functionality intended to be developed during the project. The order in which functionality is developed is generally based on risk or architectural significance with the highest risk features being developed first. Naturally, in order to determine risk, some initial assessment has to be made of the total requirements for the project. This is done by defining phases in the project.

RUP projects are divided into 4 phases:

Name of Phase	**Purpose**
Inception	During Inception, the requirements of the whole project are reviewed, but only to the extent that we gain an overview of the whole project – an inch deep and a mile wide, if you like. It is during this phase we focus on discovering risks and we propose a suitable software architecture that will support all the functional and nonfunctional requirements.
Elaboration	This is the phase where we start our first development iteration. We develop the highest risk requirement and also attempt to prove the software architecture. Ideally, at the end of this phase we had a stable architecture and have reduced the total project risk by

	up to 75%.
Construction	This is where all remaining functionality is built. At the end of this phase, we have a completed beta-version of the software waiting for final installation in the clients environment.
Transition	This is where the software is installed and undergoes end-user acceptance testing. At the end of this phase, the project is complete (NB: RUP has no defined Maintenance phase).

We therefore have a number of iteration superimposed onto the 4 typical phases of a RUP project. The total number of iteration per project is sometimes given as 6 +/- 3. That is, approximately 9 iterations for large, complex or risky project and as little as 3 iterations for simple, low-risk projects.

Since each iteration is just like a mini-waterfall lifecycle, most experienced project teams have few problems adapting to the activities required of them. This is covered further in 4.1 later.

The biggest challenge to project managers managing iterative development is keeping the whole team busy. This too is covered in a subsequent article entitled "Managing the Iterative Development Life cycle".

1.2 Manage Requirements

The single biggest reason for project failing is, according to *The Standish Group* (standishgroup.com) is lack of end user involvement. Requirements management is all about addresses this issue and provides a systematic approach to:

- Finding, eliciting, organizing, documenting and managing requirements by…
- Establishing and maintaining agreement between customer/user and the project team on the changing requirements

Project success, on the other hand, is defined as meeting all the Customers requirements. It stands to reason, therefore, that getting the requirements correct is a pre-requisite to all successful projects. But do software teams really start projects without a clear understanding of requirements? Yes, most of the time!

1.3 Use Component Architectures

Over the last 20 years or so, software engineers have tried many approaches to designing software. This is the latest of them. A component is defined in RUP as:

> *A non-trivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A*

component conforms to and provides the physical realization of a set of interfaces

Or…

A physical, replaceable part of a system that packages implementation and conforms to and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files.

In lay terms, this means that a component is a bit of software that does a specific job and has defined inputs and outputs. The reason why they are part of our list of best practises is that they provide a way of dividing up the software into manageable, testable, maintainable, sub-assemblies.

1.4 Model Visually

Some years ago, if you were an object oriented analyst or designer, you had a choice of several different modelling notations to use. Nowadays, the most popular of these have been combined (by the original authors Rambaugh, Jacobson and Booch) into the Unified Modelling Language or UML. UML is now the industry standard OOAD notation and is applicable to practically every software developme nt project. Therefore, best practises now tell us that rather than describe our analysis and design decisions in words we should instead use UML. Why? Because UML allows communication between requirements gathers, analysts and designers in a far less ambiguous and far more condensed form. Having said that, we must always bear in mind that UML is a language and as such there is always room for more than one way to model a particular piece of software functionality.

There is another benefit to modelling visually. UML tool vendors generally provide the facility to generate code directly from UML Class Diagrams and some from Sequence diagrams. This ensures that effort put into OOAD is productive since it can significantly reduce the amount of coding in a project. Furthermore, most UML tool vendors provide "reverse engineering" that allows code to be turned back into models. This allows the project team to keep the efforts of the OOAD and coding in-sync. This is something that is rarely done when documenting analysis and design decisions in words. Typically, when errors in original requirements, analysis or design are discovered at the coding stage, rarely are the original documents retrospectively updated since this activity is often regarded as a waste of time. Unfortunately, of course, this also means that by the time the project is finished, we end up with a set of documents that are totally out of sync. with the finished code. This means that when it comes to doing maintenance on this application, we can no longer use the original documentation to see the impact of changes. Worse still, because of this situation, little if any analysis and design decision get documented, and so on and so forth.

1.5 Continuously Verify Quality

In the classic waterfall development lifecycle, testing is something that is done at the end of the project. Since the results of testing are our only way to prove that functionality has been delivered, this often means that we don't know how complete our application is until the last few weeks of the project. Potentially, a whole string of non-functional tests (like performance) could fail after coding has been completed leading to a significant amount of re-work or, in the worse instance, re-writing.

By developing iteratively, we can design, code and test and we go along. This *proves* that functionality works well before the end of the project. This is the mechanism we use to reduce risk in the project.

The design of tests begins the moment requirements gathering and documentation stops. In other words, all tests are designed against requirements and are devised in parallel with the OOAD activities. This also means that by the time coding starts, the developers have documented tests (and test data) with which to prove their code. This approach is specifically designed to drive risk out of projects and build quality into it.

1.6 Manage Change

We often hear developers moaning about Customers moving the goal posts. The reality is, this is part of the nature of software development. Therefore, our approach has to bear this in mind. To a large extent, iterative development lifecycles help to address this because we can choose to leave unstable requirements to nearer the end of the project. Hopefully, by that time decisions will have been made or the delivery of that functionality can be postponed until version 2.

But this isn't all there is to managing change. Something that often unsettles those new to iterative development is that we rarely finish any one job at a single point in time. For example, in the Inception phase of a project, we gather and document requirements in the form of use cases (where one focussed on the *use* of the software). However, we only document what we need to get a very shallow understanding of the requirement – the details get filled in during a subsequent iteration immediately prior to OOAD and coding. This means that we have always got numerous partially completed documents and models. These must be subjected to version control and configuration management. As much as anything, this in what we mean by Manage Change.

2. Process must be: Self-improving

In Dunstan Thomas, we have a saying that it is ok to make new mistakes on a project but it is a cardinal sins to make old mistakes! In any well managed project, the conclusion of the project should be punctuated with some sort of an assessment or project wrap-up. This is where we discuss:

- What we did badly? and
- What we did well?
- What we should change in the next project?

Trouble is, if this is only done at the end of the project, we don't get the chance to implement all the lessons learned until we start a new project. Unfortunately, it is highly likely that the next project will be staffed by different people with different skills and the project is likely to involve new challenges and often new technology.

By developing iteratively, we get to review lessons learned at the end of every iteration. This means that improvements can be implemented in the very next iteration. Most RUP project involve 6+/-3 iterations: This gives the team every opportunity to practise and improve their skills as the project progresses. Better still, since every project-related activity is documented in the process, the process itself can be updated and improved as the project progresses.

The lack of opportunity to practise and improve skills is a major problem with waterfall project lifecycles. Imagine a waterfall project that lasts one year. The project begins with an exhaustive period of requirements gathering and documentation. At the end of this, the requirements are signed off (in blood!, never to change!) and passed on to the analysts and designers. The problem is that any ingrained or stylistic errors only get discovered after the requirements have been signed off and passed-on. By then, it's too late to address these issues. In other words, what ever the requirements people hand-over to the analysts and designers is what they have to work with. When we develop iteratively, these issues are discovered and corrected at the end of every iteration, hence we incrementally improve our working practises too.

3. Process must be: Easy to Implement

We have very rarely seen RUP used "straight out of the box". In most instances, organisations tend to want to customise it to better fit their company goals and culture. However, the degree of customisation necessary to achieve this is often not that great. By the time the whole development organisation have been trained, they usually understand enough about the process to avoid the necessity of having to follow it verbatim in every instance.

The first step is to simply install RUP on a LAN or better still an intranet. From that moment, it can be used. However, part of the RUP installation is what they call a Project Web. This is a partial copy of RUP with links to the full original version (for help, etc.) and links to customisable templates, guidelines, etc. This means that in practise, the original version of RUP is very rarely customised but instead one customised the Project Web version. This in turn ensures that when Rational release updated versions of RUP, no significant customer- or projectspecific files are overwritten.

Almost every RUP artefact is provided as a MS Word or Project template. There are also HTML versions of most Word templates. Customising RUP templates is a simple task involving changing the American spelling and adding your organisations logos – not much else is generally required, at least to start with. As the organisatio n begins to use the templates, they may decide that further customisation is required. For example, if the organisation also has ISO 9000 controlled documents, they may consider merging some or replacing others.

Another part of RUP that is often customised are the descriptions of the Roles. Each and every RUP role has defined activities that result in defined artefacts. Hence roles own artefacts. Most organisations like to map job descriptions to RUP roles, for example, Principle Software Engineer might map to the RUP Software Architect. This too is a simple task involving modification of the Development Case artefact and the associated activities and guidelines documents.

Since RUP is delivered as a web site, the nice thing about RUP implementation and customisation is that all that is entailed is amending the appropriate HTML pages and/or the Word/Project templates. This is not technically difficult and is far from onerous. This helps to ensure that amendments to the process are carried out quickly and with the minimum of effort. The only thing that remains to be done is to ensure that it is being used! This is a very human problem that sometime requires some policing. This is addresses in more detail in the next section.

4. Process must be: Acceptable by experienced software professionals

Implementing a new development process, almost by necessity, is likely to change the way people work. This is something that many people react badly against. Let's take a quick look at the "head of families" or principle roles in most software projects and see how a new SDP affects their working lives.

4.1 Project Managers

A new software development process affects Project Managers more than most other project roles. Comments like "I've been managing projects for 17 years and I'm not about to be told now that I've been doing it wrong!" tend to be heard not infrequently when a new SDP is proposed. Now for the good news: Any project manager that has successfully managed waterfall-life cycle projects will be able to manage RUP projects – provided that they get the hang of managing iterations and focussing on risk. In the third and final article in this series, we focus on just this, managing iterative project life cycles.

The Project Manager is generally responsible for the day-to-day running of the project, making sure that people of doing what they are responsible for and, ultimately, the quality of the final deliver. Project Managers can be a little like chefs: They run their kitchens the way they want to, resist change, shout a lot and demand the same enthusiasm from their staff as they have! They can also sometimes be creative tending to vary the way they do things to suit different situations or pressures and rarely is this new approach written down anywhere. This can mean that if your "chef" goes sick or takes a holiday, nobody else is able to run the kitchen in the same way. This usually also means that one chef's successful recipes are not passed on to any of your other chefs!

On the other hand, most successful project succeed due to the heroics of the Project Manager. But wouldn't it be nice if the lessons learned from one project were always passed on to the next project or other Project Managers. Here's

were the SDP comes into it's own: Because every RUP activity is already documented in the form of Activity Guidelines, all we need to do to get a Project Manager to adopt the new SDP is for him/her to adapt the Activity Guidelines to suit they way they want to work. In broad terms, this doesn't take more than a few days to achieve. In fact, it generally takes place while the project is progressing and can be part of the Project Managers daily to-do list. Once the Activity Guidelines have been amended, it's now possible for another Project Manager to join the project and see immediately how the original Project Manager wants things done.

4.2 Requirement Specifiers

As above, the key to getting buy-in from your Business Analysts is to get them to customise their own Activity Guidelines. In most RUP projects functional requirements are documented using Use Case analysis whilst non-functional requirements (usability, reliability, performance and supportability) are documented separately in an artefact call the Supplementary Specification. In this example, customising the Activity Guidelines is a case of stating what should be included in the use case documentation, what styles are permitted and how/when requirements should be reviewed.

Generalising wildly, getting Business Analysts to adopt a new SDP is not as big a deal as it is for most Project Managers.

4.3 Systems Analysts

Getting Systems Analysts to adopt RUP is likely to be very easy. If they practise OOAD, they will change little if anything about the way they work. Nevertheless, involving them in customising their own Activity Guidelines should always be the first step. For most, this will require very little work at all.

4.4 Developers

In our experience, getting Developers to adopt a SDP is easier than one might think. Most developers like to learn new things but hate change and instability. Therefore, providing a new process and relatively rigid set of working practises is just what most developers thrive on. The biggest problem tends to come from their need for being creative. Whilst creativity in design is an asset, in development it tends to lead to broken deadlines and re-work. If your developers insist on exercising their creativity muscles, encourage them to become Systems Analysts.

Developers also tend to need a little shepherding to get them to focus on testing earlier than most are used to. To encourage this, shift your project incentive schemes (you don't have one?) to focus on delivering high quality software first time. Reward those that deliver the fewest defects and get those that produce more defects to document the sources of them. Their conclusions should then form the basis to the Developers customised Activity

4.5 Testers

By the nature, Testers tend to like process more than most other project members. Perhaps because of this, customising their Activity Guidelines is likely to involve more work than most other disciplines. Nevertheless, this too can be organised to coincide with the conclusion of each iteration assessment. By bottom line is that most Tester are unlikely to resist the introduction of a new SDP.

5. Process must be: Adaptable to any size of project

Typical excuses most often trotted out for not adopting a new SDP are:

> *"Our company has unique demands for software development and one process will never address them all"*

or…

> *"The nature and size of our projects are so different that one process is likely to be too big for some projects and too small for others"*

We say, hog-wash!

If your Project Managers behave more like chefs, this is always going to be the case. However, software development should be more like an engineering discipline than a culinary one. So how can RUP be adapted to any type or size of project?
The key to using RUP for any project is to understand the purpose of the project Phases (inception, elaboration, construction and transition). Section ???? contains a description of the purpose of each phase; take a quick look back to remind yourself. Let's take 2 examples of typical software projects:

- A brand new project
- A simple change request

Can a single process be adaptable enough to address both of these?

5.1 A brand new project

Conceptually, this is the easiest type of project to manage using a full SDP since we will need to apply every part of the process to it. Therefore, because RUP has been designed to cope with any size of project, it should be possible to use the whole of RUP on any new project. Indeed, that's exactly what we do.

5.2 A simple change request

This is where the problems often start. Suppose the change request is to add a new report to an existing application. Do we *need* a SDP to manage this? Maybe not. But let's see how RUP could be used to manage it anyway.

Firstly, all projects start at the **Inception** phase. During this phase "*we focus on discovering risks and we propose a suitable software architecture that will support*

all the functional and non-functional requirements". Since the change request is to produce a report, it is likely that there will be little if any risk (except perhaps schedule) and certainly no changes in architecture. Therefore, we can dismiss the Inception phase (NB we always minute our decision to do so just in case it turns out to be a bad decision).

Next is the **Elaboration** phase. During this phase "*we develop the highest risk requirement and also attempt to prove the software architecture*". But there aren't any risks and no architectural changes either! Therefore we dismiss this phase too.

Next, **Construction**. Our first step is to decide how many iterations we need. Generally, the more risk, the more iterations we plan (we can always reduce the number of iterations later if we feel that we don't need them). We have aleady acknowledged that our change request has no real risk associated with it, therefore one iteration should suffice. Guess what? We're just turned RUP into a waterfall life cycle! So we conduct our single iteration we elements of requirements gathering, analysis, design, coding, testing and deployment. Which takes us on to the Transition phase.

In **Transition**, "*this is where the software is installed and undergoes end-user acceptance testing*". Naturally, this still takes place – so we keep this RUP phase. As you have seen, by understanding the purpose of the RUP phases, we can adapt the process to even a simple change request.

6. Conclusion

We have shown here that RUP represents a software development process that is flexible, adaptable and may be used for any type of software project. It doesn't purport to be a silver bullet and it doesn't tell experience professionals that they will have to change the way they work significantly or that the way they've worked in the past was wrong. It enshrines Best Practises and is fully customisable with little more than an HTML editor. All artefacts are based on MS Word or Project templates and of course they too may be customised. There are also HTML versions of all the Word templates.

Dunstan Thomas have been using, teaching and mentoring RUP since 1999. If you have any further questions about it, feel free to contact Graham Stone (gstone@dthomas.co.uk).