The simple artifacts of Analysis and Design

Anthony Crain

From The Rational Edge: Anthony Crain proposes using three packages to group the Analysis and Design artifacts in a RUP-based project. Along with the analysis model and the design model, already part of RUP, this article describes a third artifact: the architectural model.

When I consult and train my clients, I try to show them the simplest possible view of IBM® Rational Unified Process®, or RUP®. Once they thoroughly understand the basics, they can start to adopt some of the more interesting parts and build on their solid foundation. This is a tried-and-true teaching practice: keep things simple, then let the complexity grow as familiarity and experience grows. In this article I will take a similar approach in describing the artifacts for the Analysis and Design discipline. A simplified view of this discipline can make its adoption easier for teams becoming familiar with RUP.

Take a look at the artifacts of the Analysis and Design discipline of RUP (Figure 1). You will notice a total of twenty-one artifacts, all representing different levels of abstraction. For example, one artifact is the design model, and three others are a Signal, an Interface class, and an Event. Clearly the latter three artifacts exist at a lower level of abstraction.
While there’s nothing wrong with mixing levels of abstraction, it can be confusing to those who are first learning RUP’s best practices. In this article, I propose using three packages to group these individual artifacts. Two of them already exist in RUP: the analysis model and the design model. The third artifact is not specifically defined in RUP, but its essence has always been there; I call it the architectural model. This new model includes the Software Architecture Document as a report, as well as the Reference Architecture artifact.

In later articles I will define the contents of these three models in more detail.

Artifact overview

As I mentioned above, we can simplify the twenty-one Analysis and Design artifacts by reducing this discipline to three essential artifacts: The analysis model, the architectural model, and the design model. The rest of the artifacts will fold into one of these three.

Again, please note that the architectural model is not currently in RUP, but it offers a useful model to aggregate the architectural decisions we make. Adding this model will greatly aid performing architecturally significant reuse in future projects, because you won’t have to hunt through a design model to find the reusable assets. Note that RUP does have an artifact called a Reference Architecture. In current RUP practice, the architectural model could serve to group one or more Reference Architectures.

These three models and their relationship to their predecessor and successor disciplines are shown in Figure 2.
Figure 2: Model overview: The analysis and architectural models must combine in a non-trivial fashion to create the design model.

Please note that this is not a standard UML diagram; if it were, the dependency relationship would most likely show in an upward direction (each arrow reversed). In this diagram, however, I want to show that the analysis and architectural models must be combined in a non-trivial fashion to create the design model. The small plus sign in the black circle represents the activities of joining those two models, which is a process that can be partly automated, but still requires effort to perform.

Before we examine in more detail the three models in my simplified view of the Analysis and Design discipline, let’s review the models created in the preceding Requirements Discipline and the following Implementation Discipline.

The requirement model feeds into the analysis and design models and contains use case diagrams, outlines, and detailed use cases. It also contains glossary terms, supplemental/quality requirements, design constraints, interface requirements, and so on. It even contains features and stakeholder needs. We typically create two documents that are views of the requirement model: the vision document and the software requirement specification (SRS). The vision document provides a high-level view of the requirements with use case diagrams, use case outlines, user and stakeholder descriptions, and more. The SRS is the detailed requirements contract often signed by a customer, which drives design, test, and other team activities. It contains use case diagrams (the same ones as the vision), detailed use case specifications, supplemental requirements, glossary terms, and rules.

The implementation model is the actual code (for RUP-guided software projects) or other implementation items (for business engineering, COTS integration projects, etc). In other words, the implementation model does not necessarily contain UML diagrams. Rather, it contains the actual code, the directories containing the code, the make files, the WAR or JAR files, whatever your implementation technologies require. Many tools, including Rational Rose and XDE, have
created automations to ensure the design model and the implementation model are synchronized automatically. Changes in either model automatically appear in the other.

Some products, such as IBM Rational XDE, are already working on automating higher models (for example, automating the combination of analysis model and architectural model, using pattern engines). Many exciting automation possibilities exist to generate portions of the analysis and architectural models from the requirement models as well, but these have yet to be exploited by a tool.

The RUP Analysis and Design workflow

The Analysis and Design workflow, which you see when you click on the Analysis and Design discipline in RUP, is shown in Figure 3.

Figure 3: The RUP workflow for Analysis and Design

This diagram depicts the workflow details, or activities, that the Analysis and Design team will need to consider when performing each iteration. This may not look like a standard UML diagram, but it is. As shown in Figure 4, IBM Rational has stereotyped a standard RUP activity that normally looks like a capsule and created a special icon for it, which is a valid use of UML.
RUP workflows always represent a single iteration of a project. As each workflow detail is performed during a single iteration, RUP will show you all the artifacts you will need for capturing the results of your work. For example, when you look inside the Analyze Behavior workflow detail, you will see eighteen artifacts being consumed or produced, as shown in Figure 5.
With the simplification I am suggesting, there would only be two artifacts to target here: the requirement model and the analysis model. As I describe the various models below, I will show you in which workflow details each model is used.

The essential models of Analysis and Design

Now, let's consider the three models in my simplified view of RUP's Analysis and Design discipline. It may be helpful to refer to Figure 2 occasionally for orientation.

The analysis model

The **analysis model** is the primary artifact of the workflow detail called Analyze Behavior (see Figure 6). It is a platform independent model (PIM), which means that it does not contain technology-based decisions. For example, it would be an error to place a JDBC class in this model. Because of this platform independence, people who lack skills in a specific technology can still create the models within the analysis model. This means that even our requirement specifiers can be trained to create this model if they are so inclined, and we wish to utilize our resources in multiple roles.
Also remember that, although this is the only workflow detail in which the analysis model is the primary artifact, in each iteration that includes this workflow detail, the team will update the analysis model again. This will lead to a maintenance issue between the analysis model and the design model in future iterations, but to date the maintenance issue has not outweighed the value of having the model. There are a few tool automations to assist here, but more features are needed to keep these models in synch automatically.

**Figure 6: The analysis model contains all the technology-free, domain-specific design for our project and is the primary artifact of the workflow details circled in red.**

As shown earlier in Figure 2, the primary input to the analysis model are the use cases, rules, and glossary terms of the requirement model. In other words, the primary inputs are the functional or behavioral requirements of our system. This means that if our use cases change (if our system's desired behavior changes), the changes are encapsulated within the analysis model. The mapping from use cases to the analysis model can be very tight, which means that when a single line in a use case changes, we will very quickly see what part of the design has to change to accommodate this. (IBM Rational's Mastering Object Oriented Analysis and Design class trains students to accomplish this.)
One common question asked on various OO forums I monitor is "If I am going to code in Java, should I avoid using multiple inheritance in my modeling?" For the analysis model, the answer is no, you do not have to avoid it. If you believe that multiple inheritance is the best way to model a solution idea and you are working on the analysis model, go for it. "Technology independent" also means "language independent." Of course, if you simply don't prefer multiple inheritance, that's a different matter. The point is, inheritance techniques and the preferences you will map to the analysis model are independent of the programming language you use.

The major content of the analysis model includes UML collaborations, which group class and sequence diagrams (or collaboration diagrams if that is your preference). The collaborations are traced back to the use cases from which they are realized, using a UML realization relationship. The analysis model also contains analysis classes, which are organized according to the logical architectural pattern defined by the project's software architect.

**The architectural model**

The architectural model contains all of the technology-specific decisions for our project but none of the behavior-specific decisions. For this reason, it is considered a Platform-Specific Model, or PSM, although we will add some platform-independent pieces here as well, provided they span multiple current or future use cases.

Earlier I mentioned it would be an error to place JDBC classes in the analysis model; those sorts of classes belong, in the architectural model (Figure 7), since they are technology-specific and problem-domain-free.
Figure 7: The architectural model contains all the technology-specific decisions for our project and is the primary artifact of the workflow details circled in red.

The technology-specific decisions in the architectural model include our design guidelines, our patterns, etc. One of these technical decision types is called a mechanism. In RUP, a mechanism is an area of technical difficulty for our project that we hope to solve once, and then reuse within other parts of our own solution or in other projects. As shown in Figure 7, the architectural model is the primary artifact of the "Perform Architectural Synthesis," "Define a Candidate Architecture," and "Refine the Architecture" workflow details of RUP.

Recall from Figure 2 that the primary input to the architectural model is the supplemental specification, which contains our URPS requirements (usability, reliability, performance, supportability) as well as our design constraints and our interface requirements. What this means is that while changes to our behavior are encapsulated in our analysis model, changes to our technologies are encapsulated in our architectural model.
Since the architectural model is a newer idea with respect to the current artifact set in RUP, I'd like to summarize the value of using this architectural model to separate technology and quality issues from behavior issues.

1. This separation between the analysis model and the architectural model will allow us to be very resilient to change. Behavior changes in our use cases or rules will be encapsulated within the analysis model, while technology changes or changes to our supplemental requirements will be encapsulated in the architectural model.

2. Technological reuse and supplemental requirements reuse will be easier to perform. Both technological choices and supplemental requirements (usability, reliability, performance, and supportability) tend to be very reusable. Regardless of the problem domain, we may have similar usability requirements, for example. The architectural model tends to tie to the supplemental requirements.

3. By creating a single architectural model, it will be easier to find the reusable assets of a previous program that had either similar supplemental requirements or similar technological choices and constraints.

**The design model**

The design model has a mixture of behavior and technology, and is considered a Platform-Specific Model or (PSM) like the architectural model. Basically the idea is that you combine the PIM analysis model with the PSM architectural model to create the PSM design model, as shown in Figure 2.

With this approach, if your use cases change, you can update your analysis model, then regenerate your design model. On the other hand, if your technologies or supplemental requirements change, you update your architectural model and regenerate your design model.
Figure 8: The design model combines of the business design of the analysis model and the technological constraints in the architectural model; it is the primary artifact of the workflow details circled in red.

Currently, most development organizations focus exclusively on this model. Many take advantage of the many tools that have been built to automate the binding between the design model and the implementation model (a.k.a. the code). In products like XDE, TogetherSoft, and Rose, changes to the design model cause automatic changes to the code; and changes to the code cause automatic changes to the design model. In fact, the design model simply becomes a view of the code.

To date, no vendor offers an automated tool that fully combines the analysis model with the architectural model to create a design model. However, XDE has a pattern engine, and Rose has a series of scripts that address a portion of this need. Ideally, we would be able to focus on just the analysis model and the architectural model, and generate our design model entirely. While technology to accomplish this is a way off, we can generate part of the design model today in this fashion.

For example, in the analysis model we might have a set of boundary, control, and entity classes. In the architectural model, we can create a pattern that shows how a boundary and a few specific entity classes should be changed to become JSPs and Helper EJBs. Now we bind our specific classes to those generic patterns, and generate a design model with a combination of technology-specific and behavior-specific elements.
Again, most people just create the design model. If you look at their visual models, there is no distinction between technology-specific design and behavioral design. This makes reuse much harder and more time consuming. With the separation into analysis, architecture, and design models, if we wish to reuse some technological best practices from another project we need only look in that project’s architectural model and copy the parts that we need. If we wish to reuse behavior, which happens much less frequently, we can target their analysis model. Today we instead have to look at their design model, since that is the only model we have, and weed through it to remove the behavior before we can reuse the technological ideas.

The architecture model and different project types

Consider this: There are two types of software projects you can work on. A green field project or a maintenance project, and the architectural model has a different level of impact on the different types.

A green field project is a brand new project, while a maintenance project is a project where we already have an existing system that is somehow being modified. Overall, I have seen far more maintenance projects than green field projects in my career.

Additionally, there are three basic types of maintenance projects, and in increasing order of difficulty they are correction, enhancement, and adaptation maintenance projects.

Figure 9: Four types of projects, in yellow rectangles

RUP can guide each of these four project types, but their development cases would each be a little different from each another. It is a good idea for an organization to create a roadmap for each of the four project types over time.

A correction project involves fixing one or more problems affecting quality, or related to missing functionality. In general, the requirements work in a correction project is less intense than in other project types, because much of the requirements work was done for the original project. Specifically, there will typically be less problem analysis, stakeholder interaction, and high-level
requirement work than in the original project. Some detailed requirement work might occur. Often, correction projects can simply be a Transition phase iteration added to an existing project, rather than a full RUP project with all four phases involved.

The architecture model will still constrain the design model for a correction project, but will probably not need to be modified. In fact, if it does need to be modified, then the correction project is more architecturally significant and should probably be treated as a new generation project that uses all four RUP phases.

An enhancement project usually involves a software product that is valued by the client, who now would like the software to do even more for them. An enhancement project may be regarded as a sign of quality, because it means the client is not only satisfied with the current project, but they also want it to take on more responsibilities.

These projects require more from the requirements discipline than a correction project does, because they are about new functionality that we need to understand. If the enhancements are simple, or more specifically, if they are not architecturally significant, or if they do not fundamentally change the original vision document or the concrete use cases, they can often be handled as a Transition phase iteration, much like a correction project. On the other hand, if the enhancements affect the architecture or the fundamental vision, then a new RUP project should be started for it with all four phases: Inception, Elaboration, Construction, and Transition.

The use of the architecture model can be more significant to the enhancement project. If the enhancement requests represent behavioral changes alone, they will be isolated in the analysis model, making it easier to make the changes without having to sort through a mixture of technology and business design. If the enhancements are architecturally significant, those changes will be absorbed in the architecture model. Even if the changes are a mix, the requirement model will be partitioned to separate the two types of requirements that lead nicely to the analysis model and architectural model changes.

Finally we have adaptation maintenance projects. Adaptation projects involve systems that the client now wants to operate in a new environment. For example, a system works on MS Windows, but the client now wants it to work on Unix; or a system uses an MS Access database, and now the client wants it to function with an Oracle database; or a mainframe COBOL system now needs to work on .NET or J2EE / Eclipse.

As a methodology consultant, I have encountered more adaptation projects than any of the other types, including green field projects. I'm not sure if this is because there really are more adaptation projects out there, or if these projects are simply more complex and thus more likely to require methodology consultants for help. But if I were to guess, I'd say that most of you reading this article are probably involved in an adaptation maintenance project right now.

Adaptation projects tend to be the most complex of the three types of maintenance projects, yet when I ask for the requirements for these adaptation projects I am usually told "We don't need requirements. The new system just needs to do what the current system does now." Or they simply point at the existing system and state "those are our requirements."
My answer to this is, “If it supposed to do exactly what the current system does now, then why are we adapting it?” They usually answer with statements like “the current system technologies are no longer supported, or soon they will not be supported,” or “the current system is too slow, fails too often, is hard to use, or is hard to maintain.” All these statements represent URPS requirements that must be captured; otherwise, we will be unable to confirm that the adaptation project succeeded, or know when to stop!

Also, what if we only deliver half the functionality in the new system that was available in the old one, but we solve all of those supplemental requirements? It is possible that the loss of functionality doesn’t matter, since the goal is switching, for example, to a new platform. Obviously, these projects need use case models and scope management just like any green field project does.

The architectural model really shines in adaptation projects. Most of the requirements work will be captured via supplemental requirements, design constraints, and interface requirements, all of which are part of the supplemental specification in the requirement model. All these requirements tend to lead us straight to the architectural model. In other words, if the behavioral requirements stay the same, we will have little to do in the analysis model and can focus on the architectural model.

In projects where all we have is the design model (which is the most typical case), we must try to separate the technology from the business design, or we must start designing both all over again. This can have a drastic effect on schedules and can increase the cost of development significantly.

**Future automation**

The race is on to automate more and more of this approach. Yesterday we were focused on automating the tie between the design model and our code. There are still plenty of improvements that we desperately need in this arena, because our current tools barely scratch the surface of what might be automatically generated. But the real game has moved to automating the creation of the design model itself, which ties directly to the code.

This means that our future job will be to focus on:

- Separating the analysis model from an agreed definition of the architectural model
- Modeling system behaviors
- Selecting the right technologies for our systems

But the automation game doesn’t end there. Anyone who has taken the "Mastering Object Oriented Analysis and Design" course with me has already seen that there are a lot of automation possibilities between the requirement model and the analysis model. This means that as we create the requirement model, we could be auto-generating portions of the analysis model. Taking this idea a step further, software projects of the future may be generating code from a requirement model and an architectural model.

We might also expect various automations from the requirement model itself. I teach a method for detailing use cases with a set of standard error messages that can be manually applied when
reviewing use cases, and this method is precise enough to enable automations into the analysis model as well.\textsuperscript{4}

**Summary**

If you want to stop re-inventing solutions to difficult problems, if you want to ensure reuse and increase the productivity of your design team, then separate your Analysis and Design efforts into the three artifacts: analysis model, architectural model, and design model. When building systems with any technologies used previously on another project, the software development organizations that are quickest in adopting this approach will help their companies outpace the competition.

**Notes**

\textsuperscript{1} Commercial off-the-shelf software

\textsuperscript{2} A development case is the RUP artifact that customizes the RUP for a specific project.

\textsuperscript{3} A roadmap is a RUP artifact that is a generic development case created by studying *successfully used development cases* and distilling their commonalities.

\textsuperscript{4} Called "CRUMB," this is a requirement detailing approach in use since 1997 that has never been formally documented.