Pitfalls using UML in RUP (2)

Hans Admiraal

Summary

In part 1 of this paper, I discussed the UML models for the business modeling and requirements disciplines of RUP. Now, I will take you to the complex discipline called "Analysis & Design". I will summarize the view of RUP on UML modeling, and the diagram types that can be used in the various models. Apart from that, I will give my personal opinion and suggestions about making practical decisions and dealing with the weak spots of RUP.

ANALYSIS & DESIGN

The Analysis & Design discipline adds five models to the set of models produced by the business modeling and requirements disciplines: the Navigation Map, the Analysis Model, the Design Model, the Data Model and the Deployment Model. The other models gave us insight into the business and the requirements, but the five newcomers are models of the actual software to be built. I will discuss these models one by one.

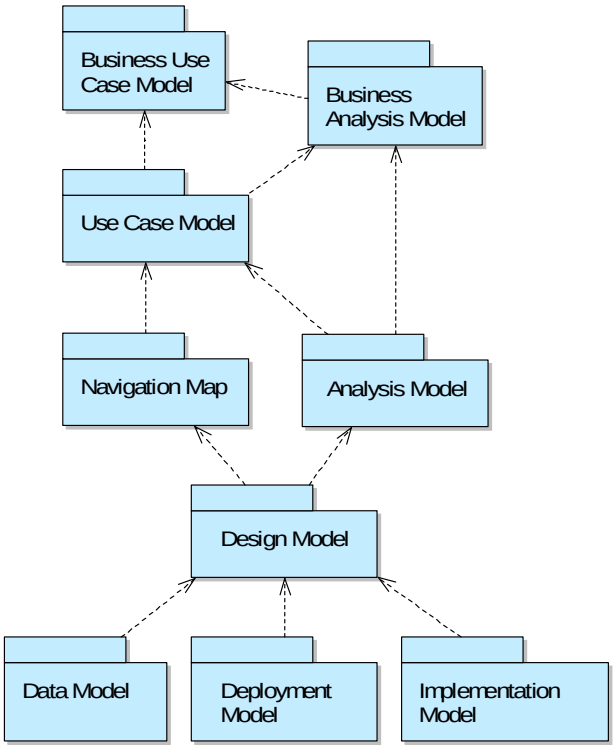

Figure 1. The complete suite of RUP models.

There is yet one other model: the Implementation Model, which belongs to the implementation discipline. It is not worth a separate paper, so I have included some words about this model near the end of this paper. Then, I will discuss the role of the Software Architecture Document before I conclude this paper.

Navigation Map

RUP defines the activity *Design the User Interface* resulting in the *Navigation Map*. This map is based on the use cases and shows the most important navigation paths. A navigation path is a sequence of screens (windows, web pages) traversed by the user. How does the map look like? In RUP, there are no rules. UML is considered not applicable.

I don't agree with that: UML's state machine diagrams are very well suited for navigation maps. The active screen is considered to be the state of the user interface and the transition arrows show the possible navigation paths. In Figure 2, I have

drawn a very sober and incomplete state machine. It is missing the triggers that cause the transitions and the actions taken by the system and it is missing exceptions. You may wonder if the user is ever allowed to go back to the main screen. The answer is yes and I explicitly mention in my user interface design document that the user can always traverse backwards, although it's not modeled in the map. The reason is that the map is not a formal, machine-readable model, but an overview, meant to convey the user interface structure to humans.

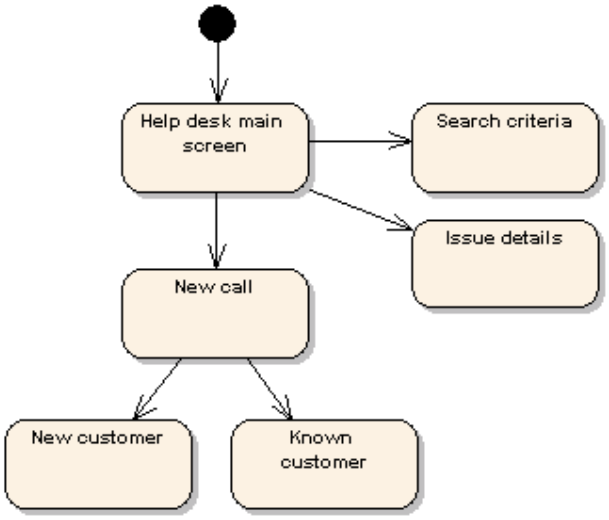

Figure 2. Navigation map.

Some people may like to use the full power of UML to model navigation details. If I try that for only two screens, I get something like Figure 3. I promise you very complex state machines if you continue that way for the complete application. I usually write more formal and detailed specifications too, screen by screen, but not using UML. These are important for implementers and testers, but there is no RUP artifact for them. RUP does not even mention the navigation map as input to any design, test or implementation activity! That's a pity, because the designer, tester and implementer have to take all user interface design decisions into account.

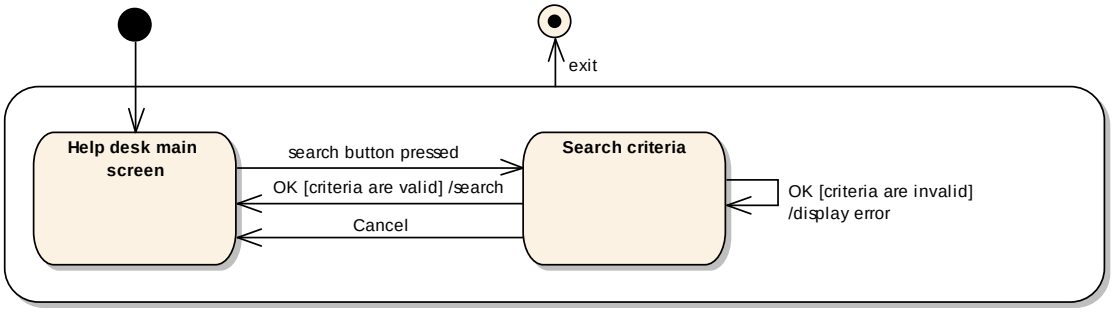

Figure 3. Detailed map of some navigation paths.

The navigation map is optional in RUP. If you have a complete user interface prototype, you may omit this model.

For large systems, the map can be very complex. According to RUP, you should put everything in one diagram, but I would create at least one navigation map for each use case package.

Analysis Model

The Analysis Model and the Design Model together reveal the system's internals that realize the use cases. The Analysis Model does this at a higher level of abstraction than the Design Model. The analysis objects are still "logical" in their nature, while the elements of the Design Model are directly recognizable in the source code. RUP allows you to go from use cases directly to the Design Model, but if this step is too large, you can set up an Analysis Model first. It should be decided per project whether the Analysis Model is replaced by the Design Model, or kept as a conceptual overview of the Design Model. Both the Analysis Model and the Design Model contain class diagrams to lay down the static structure and interaction diagrams that show the realization of use cases in terms of co-operating objects.

Nowadays, I don't make Analysis Models anymore. My Business Analysis Model and Use Case Model together provide enough information to make a first draft component architecture in the Design Model and to start making use case realizations in terms of interacting components.

The main problem with the classical RUP Analysis Model is, that it is not component-based. It consists of a lot of analysis classes that send messages directly to one another, without going through component interfaces. In my opinion, you should primarily design the component architecture and their interaction. At this level, the components are black boxes. At a lower level of detail, you design each component's internal classes and the realization of the component operations.

The second problem is, that analysis objects are logical and may not map very easily to the design objects, despite RUP's statement that the design objects are just a more detailed version of the analysis objects.

So my advice is: Put all the relevant business entities and business processes in your Business Analysis Model, then you don't need an Analysis Model anymore.

Design Model

One of the six best practices of RUP is to use a component architecture. I emphasize that even more than RUP itself, by making a strict division of the Design Model in two levels:

– the architectural level, where components are considered to be black boxes, and

– the component detail level, where the internals of each component are designed.

At both levels, the static structure is the basis on top of which the dynamic behavior is modeled.

Design Model: Architectural Level

The static structure on the architectural level is represented mainly by two diagrams: a package diagram (Figure 4) that depicts the layered approach and a component diagram (Figure 5) showing which components use which interfaces of other components. The software architect will highlight these diagrams in the Software Architecture Document.

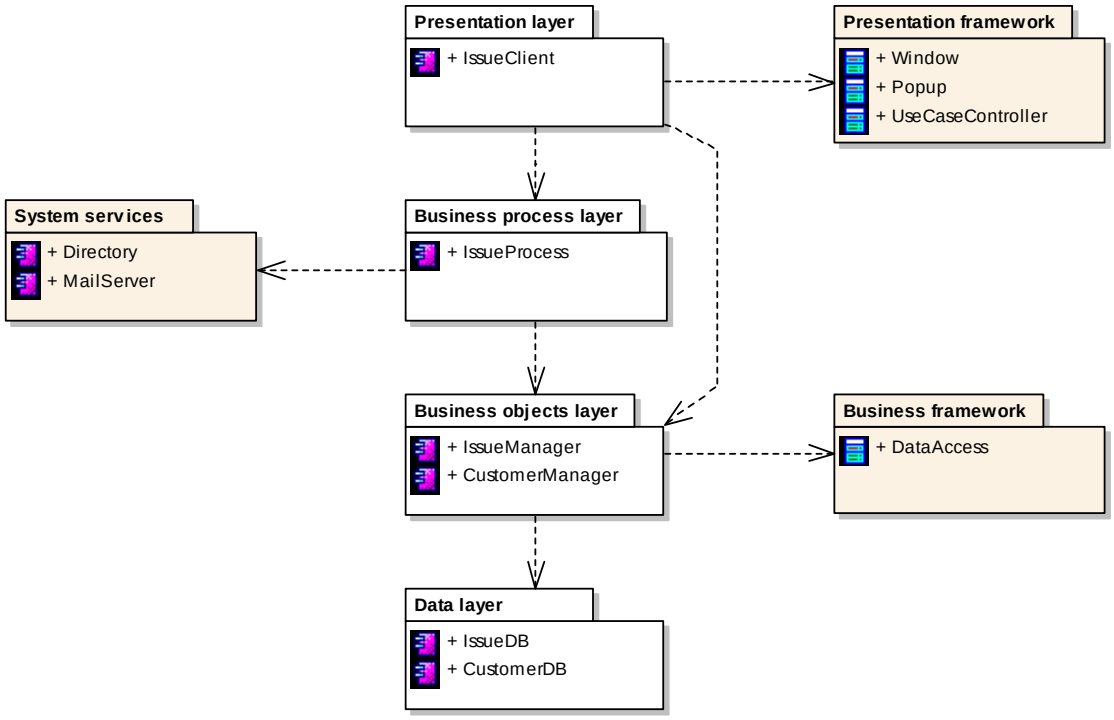

Figure 4. Package diagram showing the layered architecture

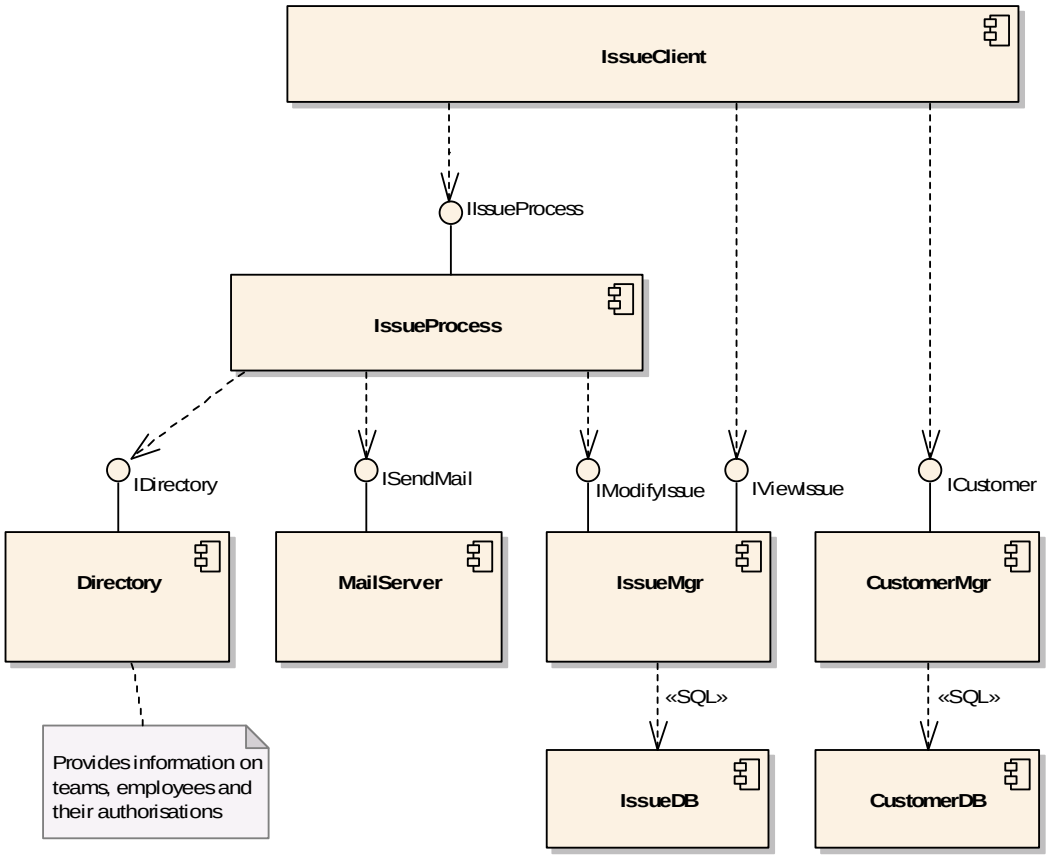

Figure 5. Component diagram

The dynamic part of the architecture consists of use case realizations (UCR). It is usually sufficient to have one interaction diagram for each use case. (Well, there may be use cases that follow very similar paths of interaction; in that case, I would only elaborate on one of them and state that the others are similar.) The interaction diagrams stay on the component level. Figure 6 is an example of that.

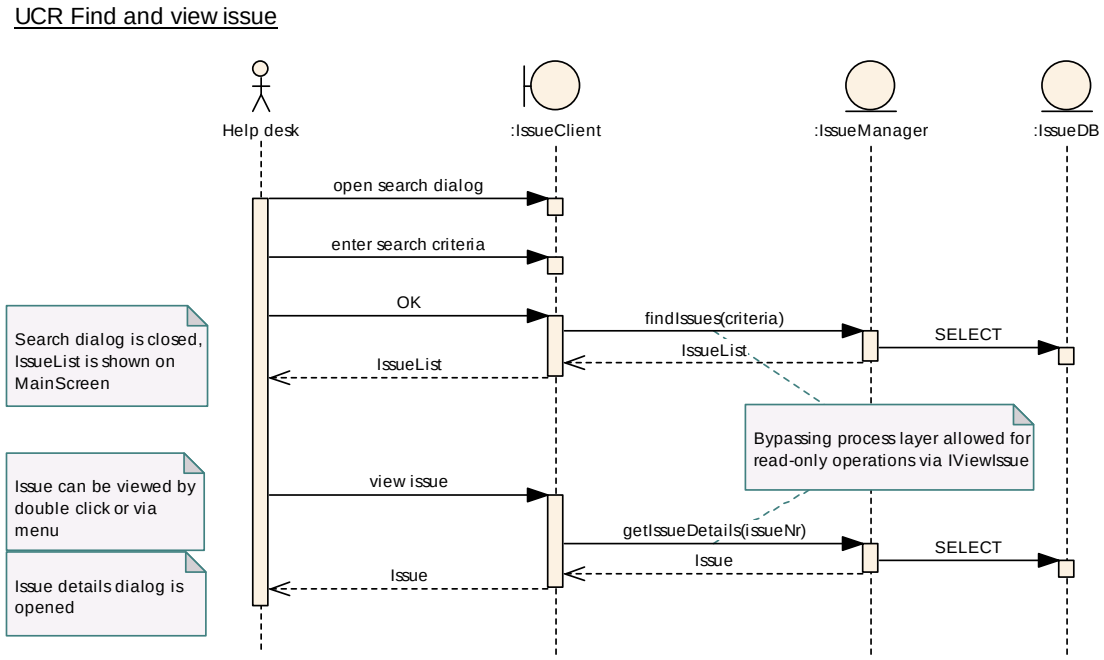

Figure 6. Interaction diagram: The realization of use case 'Find and view issue'

A well-designed component architecture and set of component interfaces is crucial for all applications. The internals of the components are far less important.

I haven't mentioned one aspect of the architecture that is required as well. The component operations may have parameters that are instances of non-primitive classes. Two of these classes are visible in Figure 6: Issue and IssueList. In order to provide a complete specification of the component interfaces, I draw them in a class diagram.

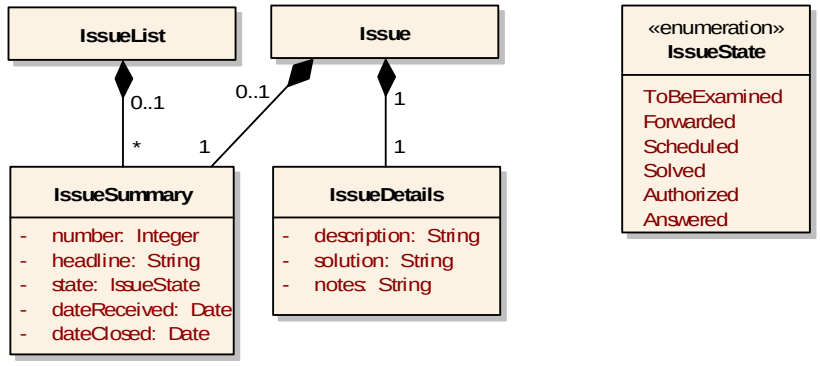

Figure 7. Complex parameter types defined by a class diagram

Design Model: Component Detail Level

At the component detail level, I model each component individually, since the interfaces and interactions with the other components are already defined on the architectural level.

The static part consists of class diagrams. These are the classes that will be programmed to implement the component.

The dynamic part is a collection of component operation realizations. For each operation with a non-trivial implementation, an interaction diagram (sequence diagram or communication diagram) is created. Depending on the agility of the project and the skills of the implementers, you determine what "non-trivial" means.

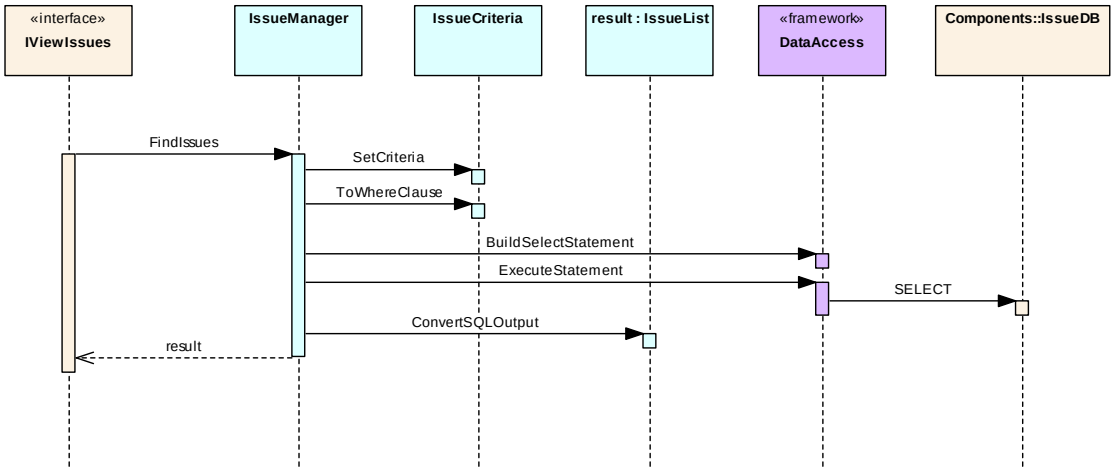

Figure 8. Sequence diagram in the Design Model: The realization of operation 'FindIssues'.

The user interface components are not triggered by operations, but by user-initiated events, like a button that is being pressed. For those components, the dynamic part consists of interaction diagrams that represent the reaction of the system to those events.

Figure 9 is a summary of the diagrams that are most valuable in the Design Model. For more light-weight designs, I would recommend at least one component diagram and for each component a class diagram (the blue ones). State machine diagrams and activity diagrams can be useful to enhance the dynamic parts of the design.

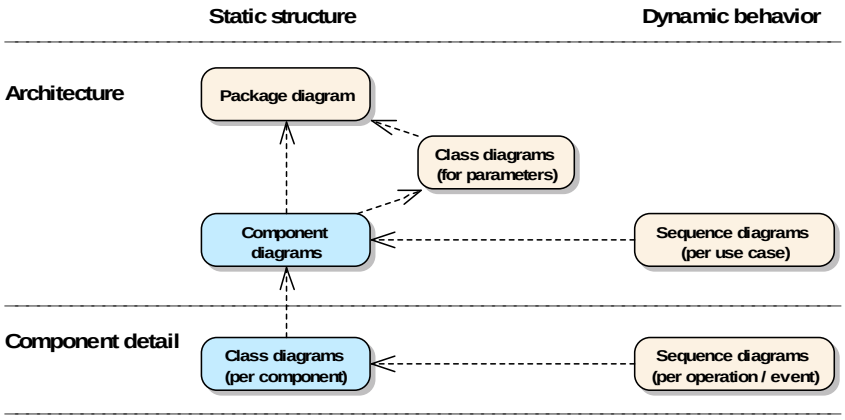

Figure 9. Diagram types in the Design Model.

Data Model

The last couple of models lay in a relatively safe corner of the field of pitfalls. We can switch from careful steps to a final sprint, if you don't mind.

The Data Model is a model of the database. If an RDBMS is part of the application, then the Data Model will specify the tables, columns and foreign key relations, and usually also stored procedures and triggers. These elements all fit in class diagrams, using special stereotypes like «table» and «column». When using multiple databases, each database should be shown as a component in the Design Model.

Sometimes, the Data Model is divided in packages, for example one for each database schema. A package diagram shows the dependencies.

Deployment Model

The Deployment Model specifies the required hardware and network connections. Within that framework, you allocate the software components to the machines on which they should be installed. UML's deployment diagram is meant to display this model.

Implementation Model

The Implementation Model is only needed if the organization of the physical source code differs from the package and component structure defined in the Design Model. In that case, the Implementation Model specifies the source code structure (the directories, for example) and the compilation order. If you like to visualize this, you can use a package diagram. The relationships between these packages and the packages or components in the Design Model should be clear, either by using a uniform naming convention or by explicit mapping.

Software Architecture Document

The Software Architecture Document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It may contain material from all models, except from the business models. Unfortunately, the set of architectural views does not precisely match the set of models, but the mapping is as follows:

- The Use Case View is a subset of the Use Case Model, containing all use cases that are significant to make architectural decisions.
- The Logical View and the Process View together constitute the Design Model's architectural level.
- The Deployment View is equal to the Deployment Model, or a less detailed version of that model.
- The Implementation View is equal to the Implementation Model, or a less detailed version of that model.

Good luck using UML in RUP!

It is often difficult to arrange the development process such, that a clear set of models is produced, taking the preferences and skills of the various team members into account. The RUP documentation is too fragmented. In this paper, I tried to bring the fragments together, mixed up with my own experiences. Did it help you? You and your team will have to sit together and find your own way. Remember, RUP is always too big. Create only those models that add value. I'm very interested to hear your comments and questions!

Hans Admiraal,
IT architect at Ordina, an IT company based in The Netherlands.
hans.admiraal@ordina.nl

The diagrams shown in this paper are created using Enterprise Architect® by Sparx Systems®.