Rose/Architect: a tool to visualize architecture

Alexander Egyed
University of Southern California
Center for Software Engineering
Los Angeles, CA 90089-0781, USA
aegyed@sunset.usc.edu

Philippe B. Kruchten
Rational Software
638-650 West 41st Avenue
Vancouver, BC V5Z 2M9, Canada
pkruchten@rational.com

Abstract

Rational Rose is a graphical software modeling tool, using the Unified Modeling Language (UML) as its primary notation. It offers an open API that allows the development of additional functionality ("add-ins"). In this paper, we describe Rose/Architect, a Rose™ "add-in" used to visualize architecturally-significant elements in a system's design, developed jointly by University of Southern California (USC) and Rational Software. Rose/Architect can be used in forward engineering, marking architecturally significant elements as they are designed and extracting architectural views as necessary. But it can be even more valuable in reverse engineering, i.e., extracting missing key architectural information from a complex model. This model may have been reverse-engineered from source code using the Rose reverse engineering capability.

1. Introduction

Mastering complexity through abstractions is an old engineering technique that worked its way into software engineering practices. Graphical representations, formalisms, and other techniques were found to be of great value, and software engineers soon identified a myriad of development techniques which provide some level of abstraction, each technique having unique features and often tailored to a particular viewpoint or domain.

It was only natural that people started to combine these techniques into development methodologies, which worked well, and seemed to cover the most important and interesting viewpoints of the development process [4]. Over time, more standardized development models emerged, providing more general models, which in turn were applicable to a larger domain of software-intensive systems. The Unified Modeling Language (UML) [3] and the 4+1 view model [6] are a result of the endeavor to unify object-oriented analysis and design techniques and their associated diagrams into a common model.

The abstractions provided through these modeling languages and their various diagrams have proven to be of great value in dealing with the complexity in software systems [5]. However, software systems have grown even more complex and the view abstractions - now starting to contain too many model elements - are in need of further abstractions, or architectural views. For example, Boehm et al. [2] show the results from an Architecture Workshop at USC where representatives from industry (both defense and commercial) identified the three most important challenges in architecture research: better formalisms, more scalability, and view needs.

2. Rose/Architect

Rose/Architect is one attempt to deal with complexity by using patterns and heuristics to extract relevant information from a system's model. Figure 1 shows the conceptual model of Rose/Architect. The system model,

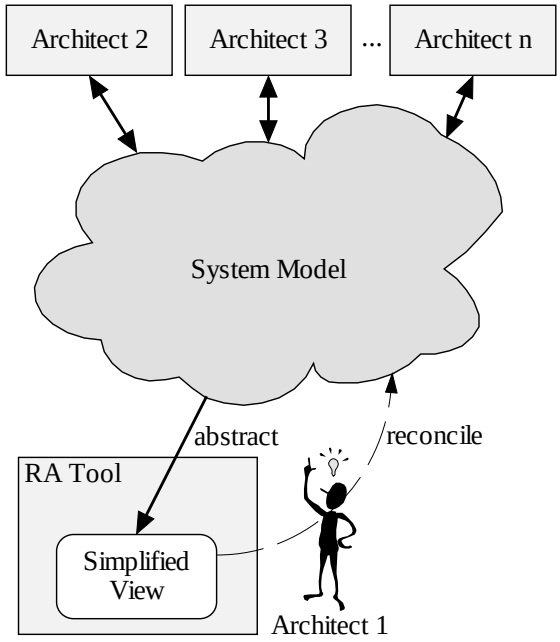

Figure 1: Rose/Architect (RA) concept

created using Rational Rose and Rose/Architect as an add-on to Rose, can abstract information from that Rose model. Even though the architects use and revise some of the same model elements while working on the same project, they may use them in a different context. Therefore, Rose/Architect creates a working environment in which a number of needs are supported:

- Abstraction: Use a subset of the model elements of the system model, which is sufficient for the developers' purpose. This step has the advantage that the resulting smaller model is less complex and therefore easier to comprehend and to modify.

- 'What happens if…' questions: A change of a model element in the system model could immediately affect any number of other developers if they use the same model element(s). By extracting a subset of the system model, developers can experiment with it independently. After completing the task, add-ons and changes are then reconciled with the original model.

The abstraction lets the developer (architect, etc.) focus on those model elements which are important for a particular task. The architect can then solve the 'simplified' problem (based on a subset of model elements) and reconcile these changes with the original system model after the task has been completed.

3. Abstraction

As mentioned earlier, Rose/Architect uses patterns and heuristics to deal with complexity. So far, we have only analyzed class diagrams (object diagrams). However, the technique described in this paper is applicable to other diagrammatic representations, called views. The technique utilizes the fact that some structures in views (e.g. collections of classes and their relationships in class diagrams) exhibit some recurring characteristics or patterns. This observation can be used to our advantage in many ways.

In Rose/Architect, we use patterns to define transitive relationships between classes. In class diagrams, a transitive relationship describes the relationship between classes which are not directly connected. A relationship, however, may exist through other classes (e.g. helper classes) which form a bridge between them. Thus, if some formula is discovered which could, with sufficient accuracy, derive a transitive relationship from the existing model, then some automatic support in simplifying and abstracting class diagrams could be provided in tool form. This would allow architects to abstract important classes from an existing model by eliminating the 'helper classes' and it would enable them to portrait and analyze the interrelationships between classes even if the classes were scattered throughout different locations (e.g. in different diagrams, or in different packages and name spaces). This paper will present a method automatically to derive transitive relationships.

4. UML Class Diagrams

We will briefly describe the notation of class diagrams in UML to explain the abstraction mechanism for class diagrams [3]. The two basic elements of UML class diagrams are components (e.g. *class* and *package*) and connectors (e.g. *generalization*, *aggregation*, and *dependency*). Components (also called model elements in UML) are categorized into a number of types, each having unique properties. Connectors depict relationships, or links, between components and may be constrained to a subset of components. Both components and connectors are first-class citizens in UML. This means that information can be 'attached' to them, which helps to further specify or characterize them (e.g. stereotypes, constraint, attributes, and operations.). This fact is very useful because components and connectors further refine the nature of the model elements and their relationship, which makes it possible to improve the accuracy of the abstraction mechanism.

4.1. Components

Components in class diagrams are, for the most part, classes and packages. Packages allow collections of classes and class hierarchies to be formed as a means of abstraction. There are other components such as instantiated classes but we will not use them in this paper since they are not relevant given the limited level of detail presented here. Furthermore, classes and packages are the most commonly used components and it is sufficient to initially concentrate on them.

4.2. Connectors

UML class diagrams support a number of connectors, most of which are unidirectional except for association-like connectors (see Table 1). Connectors, like components, may have additional attributes associated with them, such as stereotypes or constraints.

Table 1: Types of connectors

	Connectors	Instance of	Direction of Flow
1	Aggregation	Association	Unidirectional
2	Association		Unidirectional Bidirectional
3	Dependency		Unidirectional
4	Generalization		Unidirectional

As with components, we will only look at a subset of connectors. All other connectors are based on one of the three basic connectors—Association, Dependency, and Generalization. These three, together with Aggregation, are also the most commonly used connectors.

Some components may only be used with some connectors. For instance, only dependency connectors are allowed between packages. This issue becomes more complex if all components and connectors -- and their transitive relationships -- are analyzed.

5. Transitive Relationships

Transitive Relationships are the core of Rose/Architect because they provide the means of abstraction. The main challenge during abstraction is to exclude less important model elements (classes and packages in class diagrams) and to only show the relationships of the remaining model elements. The problem is that the relationships of the remaining model elements are often not explicitly stated as that is what the other model elements were used for in the beginning.

Transitive relationships may be used to replace these less important classes and thus reduce the overhead of having too many classes in one view. Since these less important classes were introduced later on in the life-cycle you may assume that the transitive relationships could be derived from the higher level abstractions represented in the logical diagrams. This is possible if the 'trace' from the logical stage to the physical stage was created and maintained properly but that is not always the case. Even if it is the case, the higher level views (e.g. logical diagrams) may already be too complex and may be in need of abstraction. Another problem is that different developers have different needs when it comes to abstractions. To support all of the developer's needs during the entire lifecycle, a large quantity of class diagrams would have to be created which would require more effort, and increase the risk of inconsistencies in the diagrams.

Therefore, it should be the developer's goal to keep the number of classes, class diagrams, and their various abstractions to a minimum. Relationships that can be derived automatically don't need to be created and maintained manually. This reduces the development effort and the risk of inconsistencies. This is where transitive relationships become important.

5.1. Patterns

Transitive relationships are usually based on patterns that can be replaced by simpler patterns, which can be further simplified if necessary. Since connectors in UML generally flow in one direction, it is necessary to differentiate between patterns made up of flows in different directions.

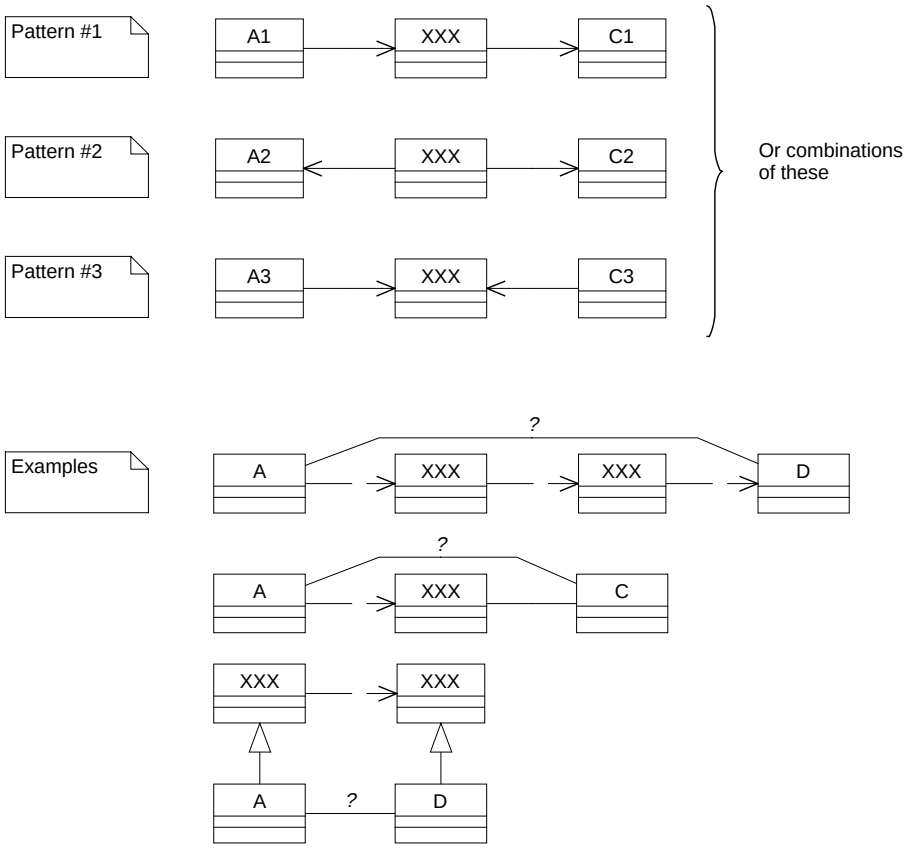

Figure 2: Simple patterns and examples of their use

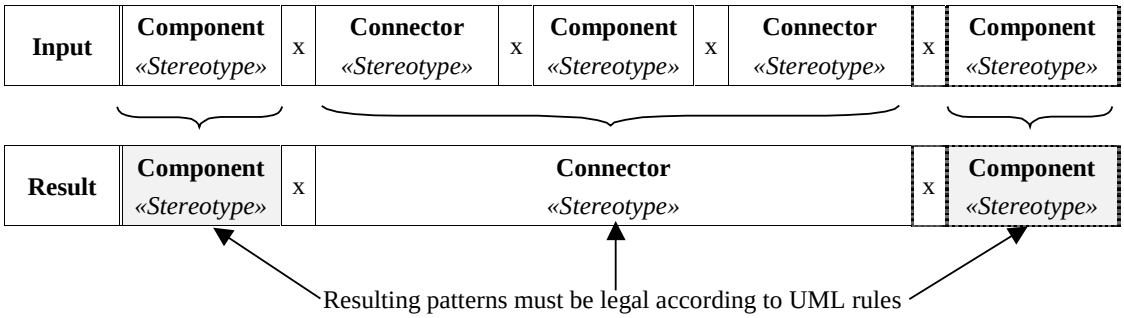

Figure 3: Simple rule input and result pattern

Figure 2 shows the three possible combinations in a simple three component setting (replacing three components with two is the simplest setting – the technique works just as well with more complex patterns). If more than three components are involved then a combination of the three basic patterns are possible (see examples). The three examples in Figure 2 show collections of classes where those classes marked with 'XXX' will be eliminated. The goal is to find the transitive relationships between the remaining ones. The first example shows pattern #1 twice in a series. Similarly, the third example may be seen as a combination of pattern #1 and pattern #3. The second example has a bi-directional flow and there pattern #1 and pattern #3 would also apply.

In cases where simple patterns are combined to make more complex ones, the order in which patterns should be applied first becomes an issue. Another challenge is when structures of classes may be resolved in different ways which leads to different outcomes. The question then becomes which pattern to apply and when. Therefore, patterns — and the rules on how to apply them — must be defined.

5.2. Rules

Figure 3 shows a simple setting in which the basic rule defines a mapping between three components (containing two connectors) and between two components (containing one connector). This rule represents the simplest structure, however, as noted earlier, the concept would also work for more complex input and output patterns.

The first row indicates the input pattern, which must be mapped onto a result pattern in the second row. In this mapping, the first component and the last component in the input and the result pattern are always the same. Both patterns must be based on the UML notation (it was explained previously in this article that some connector-component patterns are illegal).

Since the patterns and rules are based on heuristics, the rules may not always be valid. A form of priority setting can be used to distinguish more reliable rules from less valid ones. This priority setting can also be used when deciding which rules to use when. Basically, more predictable rules should be applied first.

Unfortunately, all possible combinations of components and connectors would total more than 4,000 combinations; and this does not include stereotypes or other distinguishing attributes supported in UML. If they were included, then probably millions of rules would have to be defined. Fortunately, some combinations are not possible (for example, are illegal in UML) and many others can be merged together. Table 2 shows some of the rules which can be defined using the previous pattern structure. In the table, each row shows one rule with the pattern described in Figure 3. The arrow next to the connectors indicates the direction of the flow. Results denoted with an 'xxx' indicate that there is no useable result (the pattern cannot be simplified). Sometimes, a weak result is given which may have a higher risk of failure when used.

6. Simple Example

Figure 4 gives a simple example of how the rules are applied to generate a simpler, more abstract class diagram from a collection of three diagrams. Class diagram 1 shows the relationships between people in a simplified Air Traffic Control system. It depicts a parent class *Person* for the main actors *Pilot* and *Passenger* (generalization connectors are used). Diagram 2 describes the *Flight* which has a *Location* at any given time and which uses an *Aircraft* as a vehicle (aggregation connectors are used). Diagram 3 shows the relationship of the people and the *Aircraft* (dependency connectors are used).

Using these three diagrams as input to our Rose/Architect model, we can generate a simpler model which only shows the relationships between *Person*, *Pilot*, and *Flight*. Since these components are not connected directly to each other in the input class diagrams, Rose/Architect derives the relationships using the rules defined in the previous section. The connector names in Figure 4 are preceded by 'RAGen' which indicates that they are names generated by Rose/Architect. The

Table 2 : Some rules

Rule	Component	Connector →	Component	Connector →	Component
1	Class	Generalization →	Class	Generalization →	Class
		Generalization →			
2	Class	Generalization →	Class	Dependency →	Class
		Dependency →			
3	Class	Generalization →	Class	Association →	Class
		Association →			
4	Class	Generalization →	Class	Aggregation →	Class
		Aggregation →			
5	Class	Generalization →	Class	Composition →	Class
		Composition →			
6	Class	Dependency →	Class	Generalization →	Class
		or weak Dependency →			
7	Class	Dependency →	Class	Dependency →	Class
		Dependency →			
8	Class	Dependency →	Class	Association →	Class
		×××			

[…]

38	Class	Association →	Class	← Association	Class
		×××			
39	Class	Association →	Class	← Aggregation	Class
		weak Association →			
40	Class	Association →	Class	← Composition	Class
		weak Association →			

[…]

54	Class	← Generalization	Class	Aggregation →	Class
		weak Aggregation →			
55	Class	← Generalization	Class	Composition →	Class
		weak Composition →			
56	Class	← Dependency	Class	Generalization →	Class
		×××			
57	Class	← Dependency	Class	Dependency →	Class
		×××			
58	Class	← Dependency	Class	Association →	Class
		×××			
59	Class	← Dependency	Class	Aggregation →	Class
		← Dependency			
60	Class	← Dependency	Class	Composition →	Class
		← Dependency			

information after that describes how this relationship was found.

For instance, between *Flight* and *Person*, there is a transitive relationship from *Flight* to *Aircraft* to *Passenger* and finally to *Person*. Figure 5 shows that process. *Aircraft* can be eliminated by applying Rule 59 and *Passenger* can be eliminated by applying Rule 6. Note that the process looks a little different if *Passenger* is eliminated first (Rule 6 would be applied first, followed by Rule 59). In this example, the result is the same but this may not always be the case.

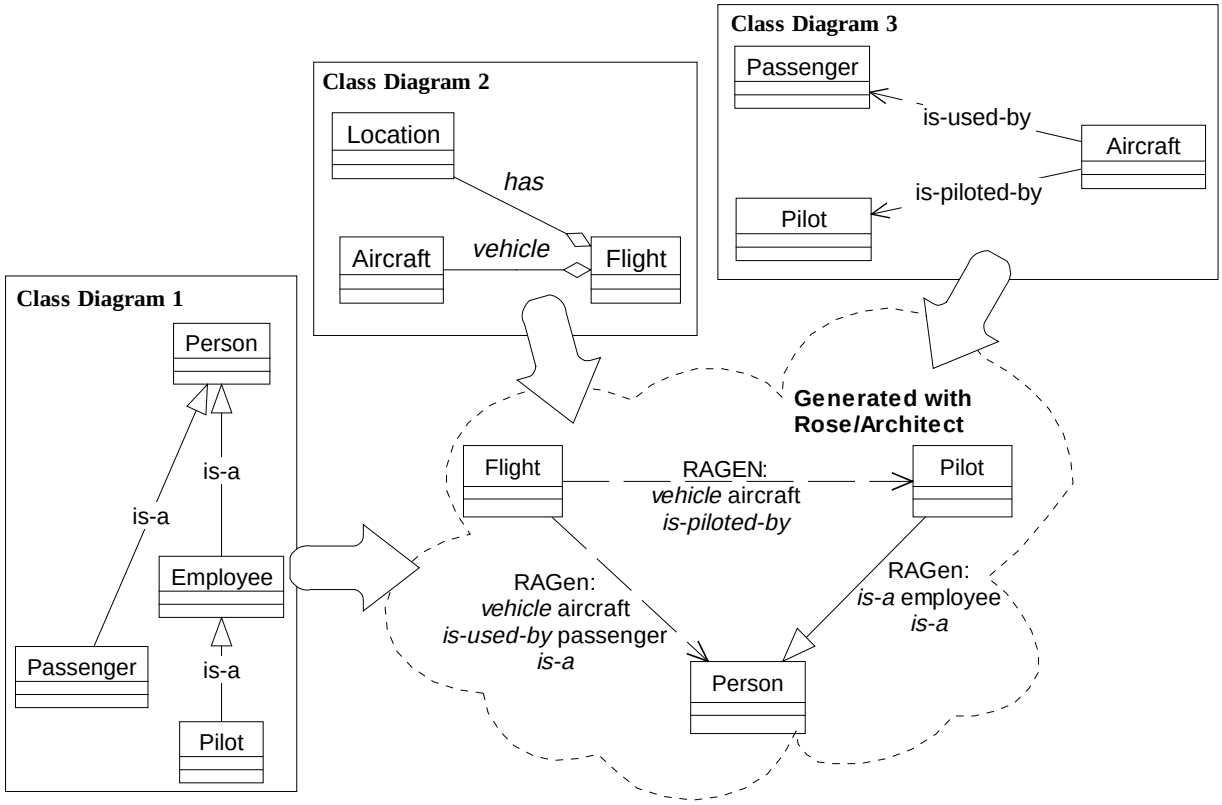

Figure 4: Simple example of RA generated abstraction from three input diagrams

7. Using Planes to Add Structure

So far we have introduced how rules and patterns can be found and how they are used by the Rose/Architect tool. However, we have not talked about how an architect uses the tool. For the tool to be useful, the architect must have a way of organizing the classes. For instance, the tool cannot decide which classes are important and which ones are merely helper classes, as this may depend on the viewpoint of the analysis. For example, a helper class in one view may be an important class in another one.

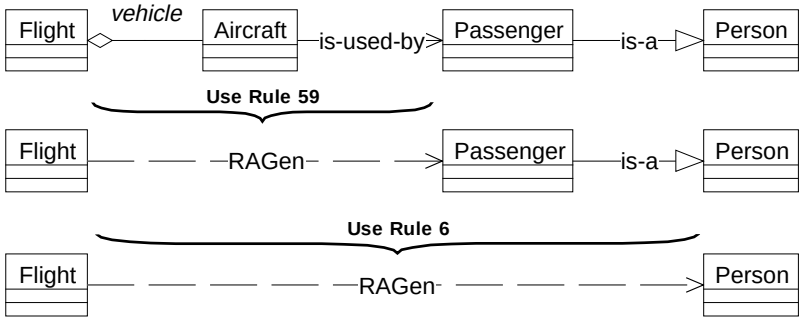

Figure 5: Generating transitive relationship from Flight to Person

Table 3: Different views in an Air Traffic Control System

Air Traffic Control System (ATCS)				
Life-Cycle	**Layers**	**Diagrams**	**Stakeholder**	**Domain**
Logical Physical Implementation	ATCS UI ATCS Components ATCS Framework Distributed Virtual Machine Basic Elements	Class Diagram Use Case Diagram Collaboration Diagram Sequence Diagram Component Diagram State Transition Diagram	Developer1 Developer 2 Tester	Specific Independent

For that purpose, we introduce the concept of planes.

Table 3 shows an example of a number of possible planes for our simplified Air Traffic Control System where planes support the logical grouping of classes (not connectors). Depending on the viewpoint of the architect, classes may belong to one or more planes. For instance, a class may be a part of the logical (high-level) design; it may be a part of the user interface; it may have been created by a particular developer; and it may be domain specific.

If the architect wishes to analyze the most important classes of the logical view, the results will be different if he or she were to look at

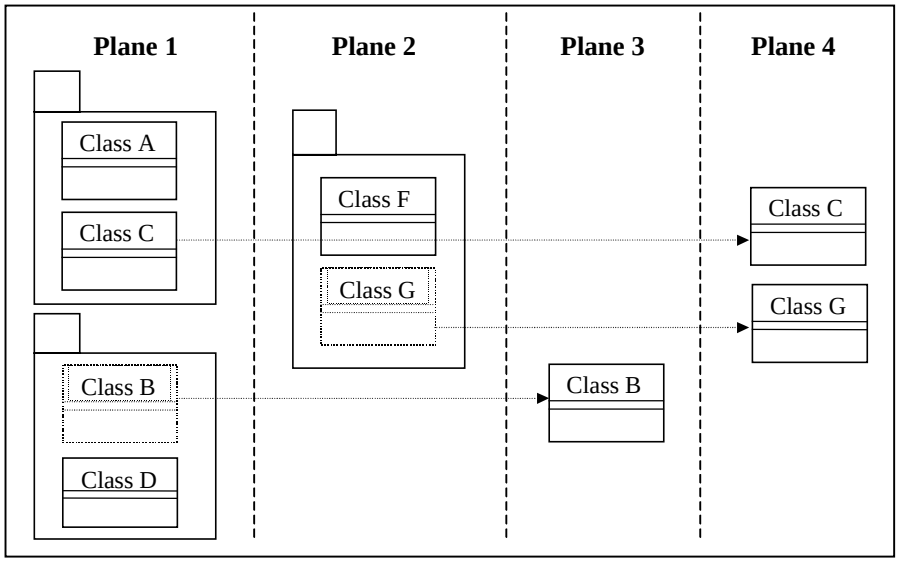

Figure 7: Classes and Planes

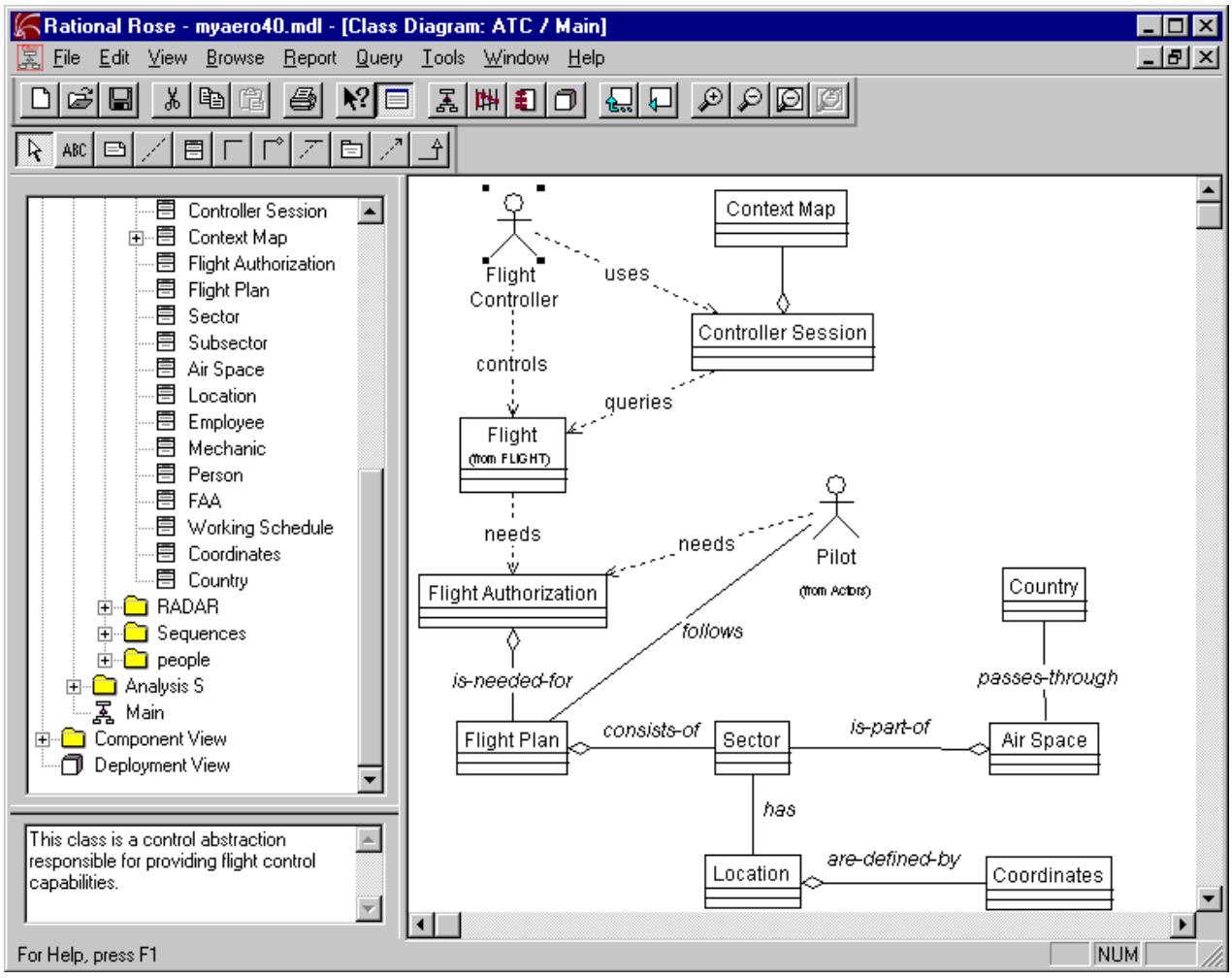

Figure 6: Simple Air Traffic Control System model in Rational Rose

the most important user interface classes (though they may overlap). For that purpose, architects may group classes (or other model elements) into planes (see Figure 6) for further analysis.

Rose/Architect can then take a plane (or a combination of planes) and create an abstract view - containing only those classes and their perceived relationships -by applying the mechanism explained in the previous section.

8. Example of Rose/Architect

To provide some understanding of what the Rose/Architect tool currently looks like and how it is used, we will show you a simple example of the look and feel of the tool using our simplified Air Traffic Control System example. The reader doesn't need to understand the details of this example, which contains approximately 40 classes. The classes are grouped in 6 packages with about the same number of diagrams depicting their relationships. Figure 7 shows one diagram which depicts the interaction of the Air Traffic Controller and the Pilot (note that this example shows a more complex environment than the previous one).

With the help of Rose/Architect, the developer can now assign each model element (class and package) to a set of planes (Figure 8). For instance, we may want to abstract the most important classes from the model to see their relationships. Therefore, we would assign all important classes to a plane and apply the rules defined above to filter the original Rose model.

The result of that process would be another Rose model containing only the important classes that we defined in the plane, including their real and hypothesized relationships (see Figure 8). The window in the upper right is used to associate classes and packages to planes and it shows the current rules (input and output pattern). Rules can be modified or added. Selecting a plane and filtering the model (not shown here) leads the architect back to the Rose tool which displays the simplified model containing only those classes which were a part of the selected plane(s). Automatically generated relationships are labeled RAGen (see lower half of picture).

8. Future work

The Rose/Architect tool and model are still in a prototype stage. The tool does not implement the entire model, e.g., the concept of priority of rules is not implemented and the names of automatically generated relations (currently called 'RAGen') are not very meaningful. However, the model can also be improved in many ways. Some of the major issues we have not dealt with are:

- **Supporting other views (diagrams):** Currently the model uses class diagrams only. Since class diagrams are not the only views which exhibit patterns, the model can be applied to other types of diagrams.
- **Incorporating 'remembering':** Since the model is based on heuristics, the results it produces will not always be correct. We would like to extend the model so that the tool remembers the errors it made, and how to avoid them the next time the same (subset of) model elements are used.
- **Reconciling views:** This is an aspect we briefly discussed but which deserves more attention. The purpose of Rose/Architect is to provide a simplified working environment where model elements which are not that important for a particular task are excluded from the developer's vision. However, changes made to the simplified model must be reconciled with the original model. This can become difficult if those changes affect other, previously excluded, helper classes.
- **Improving the accuracy of the model:** The accuracy of the model can be improved by examining the additional attributes that model elements provide. Both components and connectors have stereotypes and other attributes which describe them in more detail. Therefore, more and better rules can be designed using that information. However, a drawback to this approach is the potential explosion in the number of rules and how to efficiently deal with that. A possible way to minimize this drawback is by examining more complex input and result patterns instead of just the simple three to two component setting.
- **Integrating the model and the tool with other software architecture models and tools:** This may also be seen as an additional step towards improving the accuracy of Rose/Architect because if the RA model is integrated with other models, then the additional information provided by those models can be used to refine the existing rules and patterns thereby achieving a more accurate model.

In addition to the necessary improvements listed in the previous section, we are faced with the challenge of validating the Rose/Architect model. We can only speculate about its accuracy until it is tested in the real-world using real product models. A possible solution, which we are currently investigating, is a collection of real-world projects conducted at the University of Southern California (USC). There, 17 student teams developed products for their customer, the USC Library (see [2] for a more detailed description). Each team used a number of engineering techniques and tools, such as Rational Rose/UML, the 4+1 view model [6], the Rational Unified Process [7, 8], and many more, to develop

multimedia-related library products. We hope that the architectural designs of those models can be used to perform a first iteration of the validity of the Rose/Architect model.

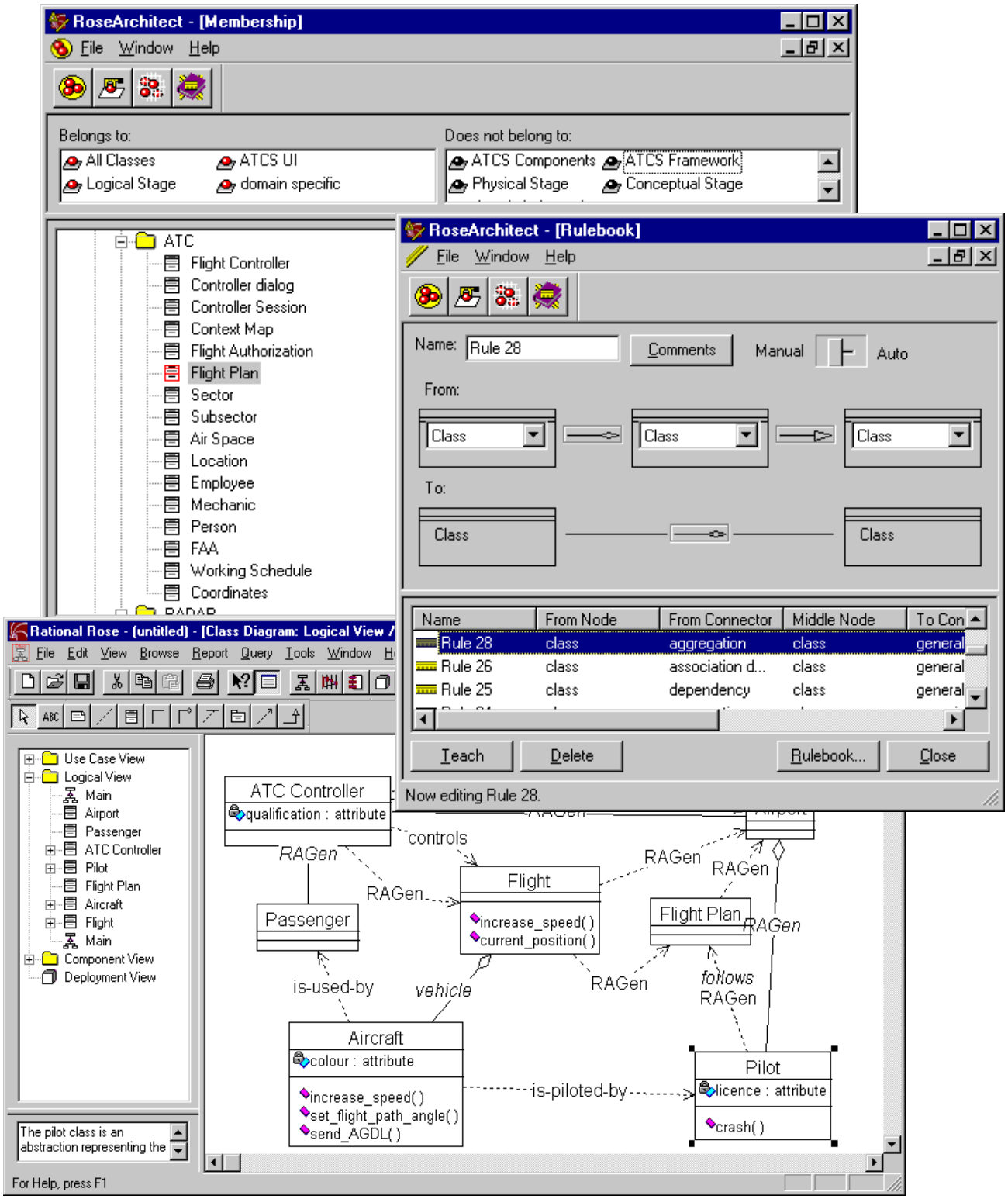

Figure 8: Planes and rules in Rose/Architect (above); abstracted Rose model (below)

9. Conclusions

Software models can be very complex and end up containing several thousand modeling elements, and that makes discovering and visualizing the essential structures of a system difficult, more so than the simple hierarchical packaging of these elements. Rose/Architect offers its users a large number of planes, that can be named and organized in groups. Modeling elements, such as classes, objects, and packages, can be associated with specific planes by the architect. The tool then visualizes, on demand, one plane or a set of planes, reconstructing missing relationships between elements using user-definable heuristics. The proposed changes are then captured in the form of Rose scripts that can be later played against the original model.

The heuristic-based approach of Rose/Architect is useful in forward engineering and backward engineering. In forward engineering, architecturally significant elements are marked and extracted into individual architectural views. In backward engineering, it can be used to extract missing key architectural information from a more complex model.

Transitive relationships can represent model elements from a different viewpoint, and can be used to verify the conceptual integrity and consistency of a model. For instance, a lower-level class model can be simplified to higher-level class diagrams and then cross-referenced with the existing higher-level class diagrams. Discrepancies between them may indicate inconsistencies within the model.

Acknowledgements

We would like to thank Mircea Bacinski from Ensemble Systems, Richmond, BC, Canada for the initial implementation of the Rose/Architect prototype, and Pamela Clarke for editing our manuscript.

References

[1] Boehm, B. W., Egyed, A., and Gacek, C., editors. Knowledge Summary: USC-CSE Focused Workshop on Software Architectures II, Center for Software Engineering, University of Southern California, Los Angeles, CA, 90089-0781, 12-14 November 1997.

[2] Boehm, B.W., Egyed, A. F., Kwan, J., Madachy, R, Port, D., and Shah, A., "Using the WinWin Spiral Model: A Case Study," *IEEE Computer* 31(7), July 1998, pp.33-44.

[3] Booch, G., Jacobson, I., and Rumbaugh, J., "The Unified Modeling Language for Object-Oriented Development," Documentation set, version 1.3, Rational Software Corporation, 1998.

[4] Finkelstein, A, Kramer, J., Nusibeh, B., Finkelstein, L., and Goedicke, M., "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," *International Journal on Software Engineering and Knowledge Engineering*, March, pp. 31-58, 1991.

[5] IEEE, "Recommended Practice for Architectural Description," Draft Std. P1471, IEEE, 1998.

[6] Kruchten, P., "The 4+1 view model of architecture," *IEEE Software* 12 (6), November 1995. pp.42-50

[7] Kruchten, P., *The Rational Unified Process–An Introduction*, Addison-Wesley-Longman, Reading, Ma, 1998.

[8] Rational Software, *The Rational Unified Process*, version 5.0, Cupertino, CA, 1998.