Role Composition in Requirements Engineering: the Method and Prototyping Tool

Pavel Balabko[2], Gleb Skobeltsyn[1], Alain Wegmann[2]

[1]Distributed Information Systems Laboratory (LSIR), IC-EPFL, Lausanne, Switzerland
Gleb.Skobeltsyn@epfl.ch
[2]Systemic Modeling Laboratory (LAMS), IC-EPFL, Lausanne, Switzerland
{Pavel.Balabko, Alain.Wegmann}@epfl.ch;

Abstract. Using multiple contexts improves model understandability and contributes to solving a scalability problem. In our work, we introduce a modeling method called Systemic Enterprise Architecture Methodology (SEAM) that considers systems to be designed in different contexts. For each context a designer specifies base roles and then composes them into the whole model of the system. The analysis of models that integrate multiple roles, however, is difficult. In many cases models can be evaluated only after their implementation. Building a rapid prototype for the system, composed of multiple roles, can be helpful for evaluating its key features before its implementation. We understand rapid prototyping to be an early development phase for building small-scale implementations (prototypes). In this work, we present a tool that supports role composition. It is based on two programming techniques: Subject Oriented Programming (SOP) and Aspect Oriented Programming (AOP).

1 Introduction

In the early stages of software development, it is important to reach an agreement between system developers and customers (or other system stakeholders) about system requirements. This agreement can be reached by means of specifying system requirements in the form of a model that can be discussed with all stakeholders.

Modeling complex systems poses a problem: models are very large and hence difficult to understand. The solution to this problem is to make models by composing small roles, where each role is small enough to reason about: each role can be discussed between stakeholders and then the composition of roles needs to be performed.

The composition, however, raises a new problem: the analysis of models that integrate multiple roles is difficult. In many cases such models can be evaluated only after their implementation. Building a rapid prototype for a system that integrates multiple contexts can be helpful in evaluating key features of a system before its final implementation. “At very early stages of planning, a small-scale prototype is built that exhibits key features of the intended system. This prototype is explored and tested in an effort to get a better handle on the requirements of the larger system. This process is called Rapid Prototyping” [22]. Rapid prototyping allows system stakeholders to rea-

son about and test the functionality of the future system. A prototype guarantees that system's requirements correspond to the common understanding of all stakeholders.

The goal of our work is to improve the early requirements specification process by means of a context-centric design method and corresponding rapid prototyping tool. Our method, called Systemic Enterprise Architecture Methodology (SEAM), is based on the SEAM visual language (VL). Using the VL gives us an opportunity to overview the structure of a system and allows us "…to discuss and validate process models with both users and owners of the process, many of whom are not prepared to invest their time in understanding more complex representations" [13]. The main advantage of the SEAM VL is human orientation. It was developed in order to improve the process of understanding models by humans. It is based on the philosophical and psychological principles that support human reasoning.

One of these principles is that every role of a system is specified in its own *context*. The context is modeled as a set of collaborating roles. To reason about a system as a whole, one should analyze relationships between different roles of the system. The notion of *compositional constraints* is used to show how different roles of a system are composed into bigger roles. In the composition process we understand *role composition* as (1) linking the *identical* model elements, representing the same entities in the Universe of Discourse, and (2) putting *constraints* between model elements found in these roles. Besides this, we also consider placing the *multiplicity* on roles as an important operation when composing a larger role out of several small ones.

To be more practical, we developed an archetype of the Rapid Prototyping (RaP) tool that supports our method. The RaP tool is based on the same idea: prototyping small (base) roles and composing them into a prototype of a system. We used Java to make prototypes of base roles and the subject oriented programming (SOP) together with aspect-oriented programming (AOP) to implement the composition of base roles.

In Section 2 we give an overview of the SEAM method. In Section 3 we describe how it can be supported by RaP. In Section 4 we illustrate RaP with an example. In Section 5 we present the state of the art. Section 6 is our conclusion.

2 Overview of the SEAM method and visual language

In this section we consider SEAM [21]: its method and its visual language (SEAM VL). The SEAM method is based on a set of systemic and philosophical principles [4] that explain how SEAM VL is used for modeling.

One of the ideas of SEAM is that any role of a system is modeled in a context. We use the definition of a context taken from [4]: *Context is the set of collaborating roles along with their state and behavior.* To represent a context in SEAM VL, we use a notation inspired by UML (see an example in Figure 1). We represent the context by a rectangle that includes some collaborations (dashed ovals), roles (stick men) and role names (below stick men). The name of the context is given in the upper part of the rectangle. We represent objects with nodes (cubes) with their names below the nodes.

For each role in a collaboration we can show a detailed specification (Figure 1.c): a box with three panes. This notation is similar to the representation of a class in UML. Instead of the attribute compartment in UML (middle pane in each box) we use a

graphical notation based on a UML class diagram. It contains attributes, relations between them and actions. We put together system's actions and attributes in the middle pane to provide a holistic view that shows the relation between the state structure and the behavior. We do this as our goal is to make explicit what the behavior does (in term of state changes) and what the state structure is used for (in term of participating to the behavior). For example, in Figure 1.c we can see that the attribute A was created in the context of the "Create A" role and then it will be used in the context of the "Create, Do, Delete A" role.

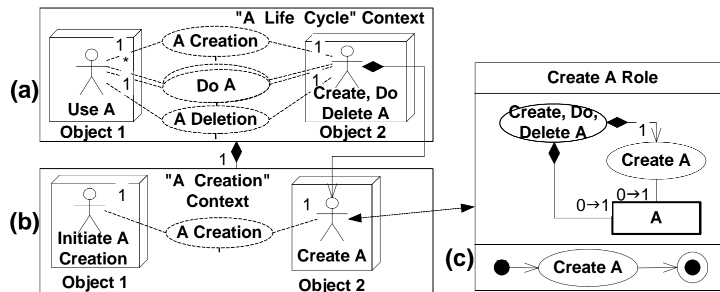

Fig. 1. Modeling using SEAM

In the SEAM method we do not prescribe how a modeler should build a model. We only indicate the SEAM modeling constraints that a modeler should follow:

1st constraint: A system of interest can be considered in a number of meaningful situations. Each situation should be modeled as a group of collaborating objects. We recommend using the pattern: "Creation X", "Do X", "Delete X". This pattern makes explicit the life cycle of relations between roles. Thus we can specify how a relation is created, how it is used and how it is deleted.

2nd constraint: The hierarchy of contexts should be specified using the whole-part relation. It specifies a context containment and the multiplicities of the context containment. For example, in figure 1 we specify that the "A Life Cycle" context contains one "A Creation", multiple "Do A" and one "A Deletion" contexts.

3rd constraint: For each collaboration (or a certain context), it is important to define the role of the system in this collaboration.

4th constraint: Based on the hierarchy of contexts, a composition of base roles[1] from lower level contexts should be made. Roles are composed basically by finding *identical* model elements and defining *composition constraints*. Identical model elements and composition constraints should be specified explicitly.

In our work we study how the composition of roles in a SEAM model can be implemented in a prototype. From this point of view the fourth SEAM modeling constraint is the most important for us. It defines two concepts that we use in the composition of roles:

Identity: Two model elements are identical if they represent the same entity in the Universe of Discourse[2] (UoD). To identify identical model elements, a clear relation-

[1] We call roles before composition *base roles* and after composition - *composed roles*.

[2] The UoD corresponds to what is perceived as being reality by an observer and *entity* is any thing of interest [8] in the UoD. Identified entities are modeled as *model elements* in a *model*.

ship between model elements and entities in the UoD should be specified. We refer to the work of Michael Jackson [9] that explains how this relation should be specified unambiguously. We use two identity constraints this paper: Attribute Equality and Action Equality. *Attribute Equality* ($attr_1$ ⊷ $attr_2$) specifies that values of $attr_1$ and $attr_2$ should be equal at any time. *Action Equality* ($action_1$ ⊷ $action_2$) specifies that two actions happen at the same time and preconditions should be satisfied for both actions before these actions can occur. We consider in Section 3.1 how the identity of model elements can be implemented in a tool.

Composition constraints are the constraints implied on the behavior of base roles. An example of a constraint often used in a composition of base roles is a *Sequential Constraint* ($Role_1.action_1 \rightarrow Role_2.action_2$). Detailed information about composition constraints in the SEAM method can be found in [5]. We consider, in Section 3.2, how composition constraints can be implemented in a prototype of a system.

The identity of model elements and composition constraints allow for the composition of roles. However, for important practical specifications, we have to take into account the multiplicities of roles to be composed. For example, a bank can be specified as an organization that manages multiple bank accounts. Therefore, if we specify the "Managing one bank account" role, than a bank would be a composition of these multiple roles. We consider a possible implementation of multiplicities in Section 3.3.

3 Technology Leveraged by the Rapid Prototyping (RaP) Tool

As we discussed in the introduction, the use of prototypes can improve the quality of requirements and makes the agreement between system's stakeholders easier. Furthermore, a prototype can be used as a basis for a final implementation. In our research experiments we have learned that the SEAM method can be used for a semi-automatic generation of system prototypes. In this section we present the result of our research that explains how a Rapid Prototyping (RaP) tool can be built to support the generation of prototypes for SEAM models.

Prototypes can be built similarly to the way SEAM models are built: system designers have to start with a specification of the base roles and by writing a corresponding code; and then they compose them into larger specifications and a corresponding composed code.

There are several case tools that support the generation of code from models (like Rational XDE, Telelogic TAU, Borland Together etc.). Therefore in our work we do not explain a code generation process for base roles. We suppose that each role at the lowest level of context hierarchy is simple enough. Hence it is simple to generate a code from it. In our work we explain how the composition of code for base roles can be performed when guided by SEAM identity and compositional constraints.

There are several technologies that allow for the composition of code fragments (such as AOP [11], SOP [7], role components [20] or coordination contracts [3], etc). In our work we decided to use only two of them: SOP and AOP. We found the SOP idea to be very close to the SEAM role composition: they both specify the identity of model elements. The AOP aspects seem suitable for implementing composition con-

straints. Furthermore, there are software tools for both SOP and AOP approaches based on Java. Therefore we use Java as a basis for prototype implementations.

The overall procedure of building system prototypes based on the SEAM design process is the following:

1. Specify base roles at the lowest level of the context hierarchy using SEAM VL,
2. Generate a code for each base role in a corresponding Java class,
3. Compose base roles by putting synthesis constraints and adding multiplicities,
4. *Automatically* produce a corresponding composition of the Java classes at the implementation level. This means: generate Java code that reflects the SEAM role composition to provide a prototype implementation,
5. Use the generated prototype implementation to test a model,
6. Improve the model by changing the rules of a composition, adding constraints or modifying base roles. Repeat steps 3-6.

Based on this process, we describe a prototype of the RaP tool and explain how SOP and AOP are used to support our approach.

3.1 Identity implementation with SOP

Subject Oriented Programming was proposed by Harrison and Ossher (from IBM) as an extension of the object-oriented paradigm to address a problem of handling different subjective perspectives on objects to be modeled. According to [Harrison03] the term *subject* means a collection of state and behavior specifications reflecting the perception of the world. SOP uses them to represent a subjective view on objects. Any object can be seen as a composition of several subjects, where each subject can be managed separately.

The description of roles in the SEAM method is closely related to SOP subjects. In turn, the SOP composition procedure is similar to the SEAM visual composition, i.e. it allows for implementing model elements identity. Thus we have chosen the SOP approach for a role composition. For testing purposes we use Hyper/J tool [2]. Hyper/J [19] supports a *multidimensional separation of concerns*[3], an extension of SOP. Hyper/J tool is a standard Java application that allows for the composition of conventional Java classes according to composition rules. The input of Hyper/J is compiled Java class files with a special options file and produces Java class files. The options file indicates which files participate in a composition, how equal named parameters or actions should be treated, and other complimentary information [19].

Figure 2 shows a composition of roles with a possible Hyper/J implementation. We assume that *CRole1* and *CRole2* classes are implemented in Java and correspond to the graphical notation The *CRole12* class is a result of composition of CRole1 and CRole2 classes based on a Hyper/J options file. If we compare figures 2.a and 2.b, we find out that composition constraints (links between equal entities) correspond to the Hyper/J options file.

With the limits of this paper we cannot give a detailed overview of the options file. In Table 1 we show only the part that is responsible for the compositional constrains.

[3] The multidimensional separation of concerns introduces a new unit of modularization, other than a class. Each concern encapsulates a particular area of interest [19].

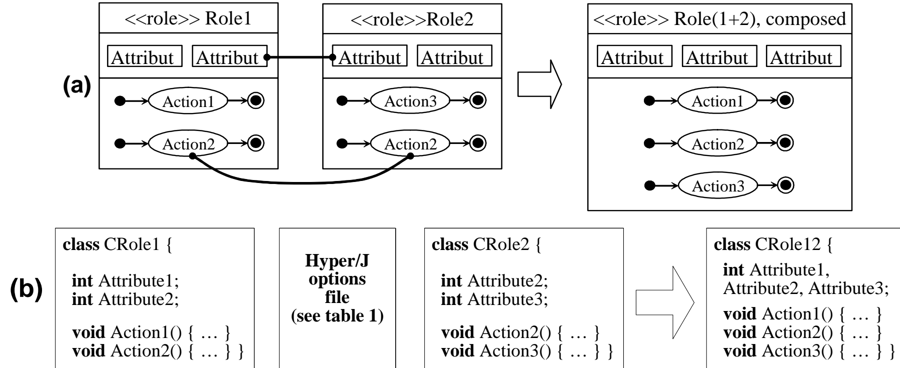

Fig. 2. Composition of roles: (a) identity modeling; (b) Hyper/J implementation

In order to support our approach in the RaP tool we need to generate a Hyper/J options file automatically from the visual composition links.

Table 1. Hypermodule part of the Hyper/J options file

```
Hyperslices: Role.Role1, Role.Role2
 ; define two roles as hyperslices
Relationships OverrideByName;
 ; It specifies what to do if two ore more actions with equal names
 ; are encountered: if OverrideByName - first action substitutes
 ; others; MergeByName ensure consequent invocation of all actions.
Equate class Role.Role1.CRole1, Role.Role2.CRole2
 ; Class CRole1 from Role1 role is equal to class CRole2
Rename class Composed.CRole1 to CRole12
 ; Rename composed class to CRole12
End hypermodule;
```

3.2 Constraint Implementation with AOP

The Aspect Oriented Programming (AOP) paradigm introduces a new concept called *Aspect* for encapsulating a crosscutting code. According to [6] a model is an aspect of another model if it crosscuts its structure. Compared to OOP, an aspect in AOP can be considered as a modular unit like a class. It wraps a supplementary code and also stores information about *join points* and *pointcuts*[4]. Such join points indicate when the supplementary code should be executed. In order to compose original classes with aspects, a waving procedure has to be made.

We use AspectJ 1.1 tool in our experiments. In contrast to Hyper/J, which proposes a composition of pure Java classes according to the options file, AspectJ supports only an augmentation of Java classes with aspects. We found this property convenient for *constraints implementation*. Since AspectJ permits the substitution of a certain method, in the simplest case it can be used to block execution of a certain operation

[4] *Join points* are well known points in the dynamic execution of a program. And *pointcuts* are sets of join points.

accordingly to a given conditions. Therefore we define constraints as "*execute this operation only if <Boolean expression>*". We believe that such simplified constraint definition covers the majority of the practical cases.

To support role composition, we provide a library of different constraints. Some of them (such as the sequential constraints) can be defined using templates (see Table 2).

Table 2. Constraint implementation pattern for sequential constraints

```
Aspect SomeConstraint {
 Public int flag=0;
 pointcut P1():call(* CRole12.Action1(*))
 after():P1(){flag=1;} //executed when CRole1.Action1 is called
 void around() execution(* CRole12.Action2(*)) { // Check flag value
  proceed(); //execute an original action if a condition is fulfilled
      ...
 }//This code is executed instead of the execution of CRole12.Action2
}
```

In this aspect we declare the *flag* variable to control an execution of the *Action2* according to the flag value, changed after the *Action1* call.

3.3 Multiplicity implementation

In the composition process, a modeler often needs to put a certain base role into a collection, i.e. the multiplicity for this role should be specified (Figure 3). How can we implement the role multiplicities? We do not go into details about multiplicity implementations - this can be a subject for another paper. We simply give some suggestions and in Section 4 we explain how we implemented the multiplicity in our example.

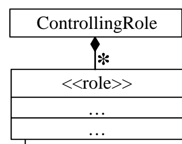

Fig. 3. The multiplicity example

We suggest using an additional Java class (ControllingRole) that implements a *list* of base role objects. This class will typically have the same methods as a base role. The purpose of this class is to redirect a method invocation to a certain member (role) in the role list. The following (non-exhaustive) choices are possible:

- Create a new member and invoke a corresponding method from this member;
- Invoke a corresponding method from every member of the list;
- Find a certain member by some condition and invoke a corresponding method from that member. It is also possible to modify a signature of a certain method to pass a needed information as a parameter(s);
- Delete the member from the list.

We emphasize one more important property of role multiplicities: it is possible to put a constraint on the number of roles in the role list. This task is very practical and allows for reasoning about the capacity of systems.

4 Presentation of the RaP tool

In this section we give an example that shows how the SEAM modeling method (based heavily on role composition) can be supported with the RaP tool. For this example we take a very simple case that shows how a prototype of a Reseller Shop can be built and tested based on the SEAM modeling method. The idea is to specify a set of very simple roles with the corresponding Java code (that can be generated using Together or Rational case tools). Based on these roles (and the corresponding code), a prototype of a model can be built by means of composing these roles. The SEAM method will navigate a system's developer in the development process by advising roles that should be composed and the way the composition has to be done. In order to see how the SEAM process can be supported with the RaP tool, we briefly discuss a model of the Reseller Shop and explain how a prototype of this model is built.

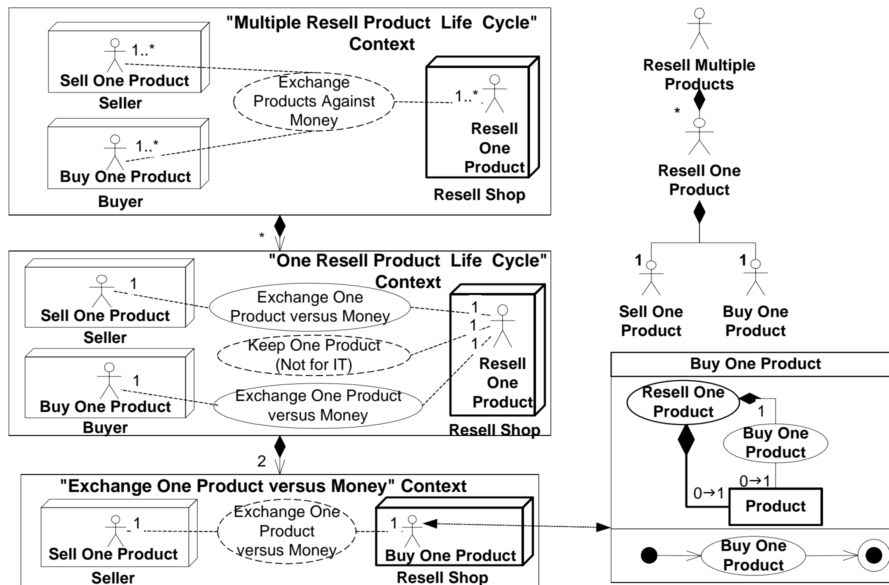

Fig. 4. The Resell Shop model

Figure 4 illustrates the main idea of the Reseller Shop model. At the lowest level we have two base roles: "Sell One Product" and "Buy One Product". Both these roles are defined in the "Exchange One Product versus Money" context. One level higher, these roles are composed into the "Resell One Product" role from the "One Resell Product Life Cycle" context. On the next level the "Resell Multiple Products" role is a composition of multiple "Resell One Product" roles. This role is defined in the "Multiple Resell Products Life Cycle" context.

Now we describe a scenario that shows how the RaP tool can be used to generate a prototype for the *Resell* Shop example (more detailed demonstration see in [16]). The scenario starts with the input of the base roles *Buy One Product* and *Sell One Product* into the tool. In this work, we do not consider how this can be done. For the simplicity, in the RaP tool we call these two roles *Buy* and *Sell* (Figure 5).

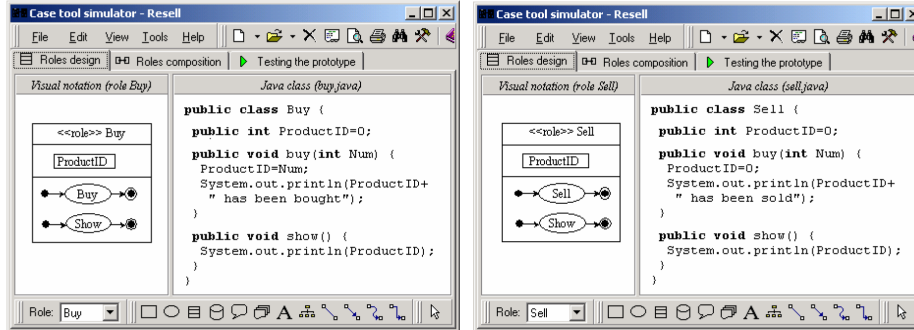

Fig. 5. RaP tool: Buy and Sell roles

1. Roles composition. The first step in our scenario is to perform the composition of the *Buy* and *Sell* roles (Figure 6). It is done by selecting identical attributes (actions) and linking them. Both, the visual representation of the composed *Resell One Product (*or simply *Resell)* role (Figure 7) and Hyper/J options file are generated automatically. A Java class corresponding to the *Resell* role is produced by means of Hyper/J tool (*buy* and *sell* classes along with synthesized *options file* are compiled by Hyper/J compiler). The result of this composition can be consulted by selecting the "Composition result" tab. It is similar with the Resell role in Figure 7b.

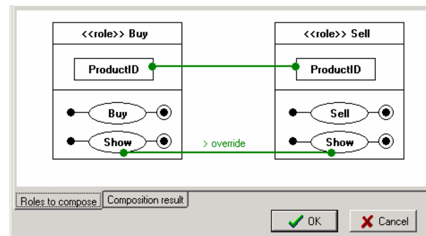

Fig. 6. RaP tool: roles composition

2. Constraint placing. The RaP tool offers the following functionality: the user selects an action that should be affected by the constraint. Then the list of available constraints with their properties will drop out (Figure 7a). In our example we require the *sell* action (from the *Resell* class) to be executed after the *buy* action. We use a *SequentialConstraint* to specify this. The aspect code, stored in the library of constraints, is added to the project by means of AspectJ compiler. A user only selects the names of actions that should be sequentially ordered (Figure 7a). Figure 7b shows the result of the constraint placing and Table 3 shows the corresponding code.

Table 3. SequentialConstraint aspect code automatically added to the project

```
aspect SequentialConstraint {
 public int FirstWasExecuted=0;
 pointcut First():call(* Resell.buy(*));
 after():First() { FirstWasExecuted=1; }
 void around():execution(* Resell.sell(*)) {
```

```
  if(FirstWasExecuted==0){System.out.println("CONSTRAINT OCCURED");
                                return; }
  proceed(); //Execute sell() action
  FirstWasExecuted=0; } }
```

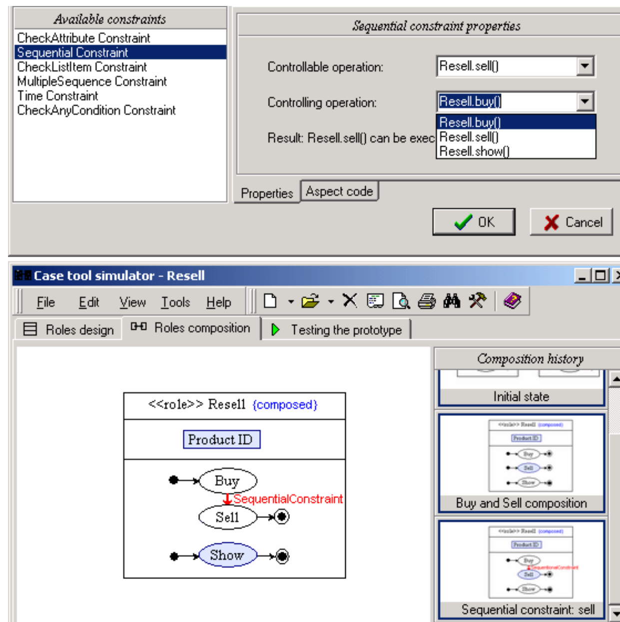

Fig. 7. Constraint applying: (a) selecting a constraint; (b) *SequentialConstraint* applying result

3. Multiplicity. The next step is to implement the *multiplicity* of roles. The multiplicity is based on the role composition that includes roles of the same type. There are different ways of composing the behavior of these roles. For example, we can compose the behavior of multiple Resell roles sequentially. This means that a reseller of multiple products will be able to resell products sequentially. Another way to compose multiple roles of the same type is to compose them in "parallel". In our example, this means that a reseller of multiple products will be able to resell multiple products simultaneously. The "parallel" composition can be implemented also in different ways (see [16] for parameters that allows for different "parallel" compositions) and can be a subject of a separate paper. In our paper we do not discuss different implementations of *multiplicity.* We only explain how the "Resell Multiple Products" role was implemented in our particular case.

To implement the composition of multiple Reseller roles we provided an additional *ControlWarehouse* class (Figure 8) that implements the Resell role. It stores a list of objects of the *Resell* type. *ControlWarehouse* has the same three methods as the Reseller role: *sell*, *buy* and *show*. The method *buy* of the *ControlWarehouse* class creates a new member (*Resell* role) and then invokes the buy method of this new member. The *sell* method does the following: finds a necessary member (*Resell* role) in the list of roles, invokes the *sell* method from this role and then deletes this member. The *show* method invokes the corresponding method from each member of the list.

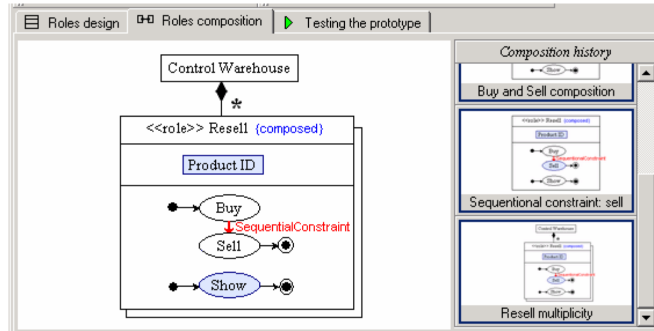

Fig. 8. RaP tool: setting up the multiplicity properties

5 State of the Art

We see the following two research directions in this area of concern-based modeling:

The first direction considers how to model concerns at the code level. AspectJ is the most known concern-based technology and UML is the most known modeling language. Therefore, main research topics here are about modeling the AspectJ code in UML (see [1], [10], [17], [18] and etc.). Dominik Stein in [17], for example, "… provides representations for all language constructs in AspectJ and specifies an UML implementation of AspectJ's weaving mechanism". These approaches are useful for programmers or software architects. They are not very useful, however, for requirements specifications because they focus mostly on the structure of the AOP code.

The second direction in modeling concerns is to model system requirements without direct link to the code structure ([12], [14], [15]). Approaches in this direction make abstractions of implementation details and deal mainly with roles and constraints. The analysis of models in these approaches, however, is difficult. In many cases models can be evaluated only after their implementations.

In our work we try to make a "bridge" between the two upper-mentioned directions. We propose a concern-based modeling method for requirements engineering that can be supported with Java-based rapid prototypes. Even if these prototypes are limited in their functionality and not efficiently implemented, they are useful for system stakeholders to reason about the functionality of the future system.

6 Conclusions

In this paper we have introduced a context-based modeling method called *SEAM* and an accompanying Rapid Prototyping tool for this method. A system in SEAM is considered as a composition of base roles, where each role is small enough to be reasoned about. Base roles are composed into larger roles by documenting explicitly the design decisions in a form of *identity* and *composition constraints* and kept in the history of the design process. This allows system designers to understand how a system is composed from base roles.

To facilitate the analysis of systems composed from multiple roles, we have proposed the RaP tool. This tool is based on the same idea as the SEAM process: prototyping base roles and composing them into a prototype of a system. The SEAM method with prototyping capabilities allows a modeler to understand systems and their goals instead of the invasive writing of a supporting code to test models. It aids in the comprehension of models functionality: even people without a deep knowledge of modeling techniques are able to reason about SEAM models.

References

1. O. Aldawud, T. Elrad, and A. Bader, "A UML Profile for Aspect Oriented Modeling," presented at Workshop on AOP at OOPSLA 2001, Tampa Bay, FL, USA, (2001)
2. http://www.alphaworks.ibm.com/tech/hyperj, accessed on November 24, (2003)
3. L. Andrade and J. Fiadeiro, "Interconnecting Objects via Contracts," UML'99, (1999).
4. P. Balabko and A. Wegmann, "Context Based Reasoning in Business Process Models," Proc. of IEEE Information Reuse and Integration (IRI 2003), Las Vegas, USA, (2003)
5. Balabko, P., Wegmann, A., "A Synthesis of Business Role Models", ICEIS, (2003)
6. Czarnecki K., Eisenecker U. "Generative Programming Methods, Tools, and Applications", Addison-Wesley Pub Co; 1st edition, June 6, (2000)
7. Harrison W., Ossher H. "Subject-Oriented Programming (A Critique of Pure Objects)", Proc. of OOPSLA'93 (1993)
8. ISO/IEC, 10746-1, 3,4 | ITU-T Recommendation X.902, Open Distributed Processing - Basic Reference Model - Part 2: Foundations. ISO, (1996)
9. Jackson M., P. Zave, "Domain Descriptions", Proc. RE'93, 1st Intl. IEEE Symposium on Requirements Engineering, (1993), 56-64
10. M. M. Kande, et al., "From AOP to UML - A Bottom-Up Approach", in proc. of the Aspect-Oriented Modeling with UML workshop, Enschede, The Netherlands (2002).
11. Kiczales G. et al., "Aspect Oriented Programming", in proc. of ECOOP'97, Finland (1997)
12. Motschnig-Pitrik, R., "Contexts and Views in Object-Oriented Languages", in proc. of IEEE CONTEXT'99, Lecture Notes in Computer Science, vol 1688, Trento, Italy (1999)
13. Phalp K.T. (1997). "Using Counts as Heuristics for the Analysis of Static Models", Workshop on Process Modeling and Empirical Studies of Software Engineering (1997)
14. Reenskaug, T., et al., Working With Objects: The OOram Software Engineering Method. ed: Manning Publication Co (1995)
15. Riehle, D. and T. Gross. "Role Model Based Framework Design and Integration", in proc. of OOPSLA'98, ACM Press (1998)
16. Skobeltsyn G., Balabko P., "RaP tool: one possible scenario" accessed from http://lamspeople.epfl.ch/balabko/RaPTool/ on November 24, (2003)
17. D. Stein, S. Hanenberg, and R. Unland, "A UML-based aspect-oriented design notation for AspectJ," presented at First Conference on AOSD, Enschede, The Netherlands (2002)
18. J. Suzuki and Y. Yamamotto, "Extending UML with Aspects:Aspect Support in the design phase," presented at 3rd Aspect-Oriented Programming Workshop at ECOOP (1999)
19. Tarr P., Ossher H., "Hyper/J™ User and Installation Manual" (2000)
20. M. VanHilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs," presented at OOPSLA'96, (1996).
21. Wegmann, A. "On the Systemic Enterprise Architecture Methodology (SEAM)", in proc. of ICEIS'03, Angers, France (2003)
22. Wilson, B. G., Jonassen, D. H., Cole, P., "Cognitive approaches to instructional design", in The ASTD handbook of instructional technology, New York: McGraw-Hill (1993)