Innovations Syst Softw Eng (2009) 5:65–78
DOI 10.1007/s11334-009-0078-3

ORIGINAL PAPER

A model-driven approach for the derivation of architectural requirements of software product lines

Alexandre Bragança · Ricardo J. Machado

Received: 1 September 2008 / Accepted: 12 January 2009 / Published online: 12 February 2009

Abstract The alignment of the software architecture and the functional requirements of a system is a demanding task because of the difficulty in tracing design elements to requirements. The four-step rule set (4SRS) is a unified modeling language (UML)-based model-driven method for single system development which provides support to the software architect in this task. This paper presents an evolution of the 4SRS method aimed at software product lines. In particular, we describe how to address the transformation of functional requirements (use cases) into component-based requirements for the product line architecture. The result is a UML-based model-driven method that can be applied in combination with metamodeling tools such as the eclipse modeling framework (EMF) to derive the architecture of software product lines. We present our approach in a practical way and illustrate it with an example. We also discuss how our proposals are related to the work of other authors.

Keywords Model-driven engineering · UML · Software product lines · Logical architecture

1 Introduction

The alignment of the software architecture and the functional requirements of a system is a demanding task. The conceptual gap that exists between the problem domain and the solution domain is very significant for the majority of the software projects. When this happens, it becomes difficult to co-relate requirements specifications and design decisions. The software architecture can rapidly become unsynchronized with the specified requirements of the system. To address this problem it is necessary to keep links between elements of the different levels of development. Achieving this without proper methodological and tool support is a daunting task. Model-driven development is a promising approach since models are treated as first-class software artifacts, as traditional development does with code.

Model-driven methods are still a research topic. Much of the actual effort is on supporting techniques for transformation of models. One example is query–view–transformation (QVT) [1], an Object Management Group (OMG) initiative to standardize model transformations in model-driven architecture (MDA). Also, reports of model-driven approaches tend to focus on transformations and are usually applied to design and implementation models, and usually do not include requirement and analysis models. Nonetheless, requirements specification and analysis are crucial activities of all software development processes. They drive the design of the system's architecture. As such, they should be integrated into model-driven methods.

Product lines are also another concept that is gaining popularity in the software industry [2]. In this paper we will explore how a model-driven approach can be applied to derive the architectural functional requirements of a product line from its requirements. We illustrate our approach by showing how a model-driven method called the four-step rule set (4SRS) [3] is extended to support product lines and the integration of requirements models. The 4SRS method is based on unified modeling language (UML) 2.0 [4]. The method applies transformational rules in order to derive a high-level

A. Bragança (✉)
Department of Informatics Engineering,
School of Engineering of Polytechnic Institute of Porto,
Porto, Portugal
e-mail: alex@dei.isep.ipp.pt

R. J. Machado
Information Systems Department,
School of Engineering of University of Minho,
Guimarães, Portugal
e-mail: rmac@dsi.uminho.pt

architecture from use case requirements. For modeling variability within 4SRS we propose the adoption of the UML-F profile [5]. Since the UML-F original focus was on class diagrams this paper presents some extensions to address variability modeling in analysis and requirements models. This paper also describes rules that can be used to transform analysis and requirements models into architectural models in a way that preserves variability and can be used without previous extensive knowledge of the problem domain.

To exemplify our approach, we will use a library product line. Suppose there is a software company that is planning to provide software management applications for libraries. Such a company could adopt a product line approach. It could develop products with different features to address the specific needs of its customers. It could also plan on releasing versions of its products with different features at different moments.

The remainder of this paper is structured as follows. Section 2 briefly describes the 4SRS method. Section 3 is concerned with issues regarding modeling variability with UML use cases. To illustrate our approach we present two use cases of a hypothetical library product line. Section 4 describes how functional requirements of a product line can be modeled with use cases and activity diagrams. In Sect. 5, we describe how use case realizations are used to derive the architectural requirements of a product line. We address the global logical architecture of the system in Sect. 6. In Sect. 7 we conclude the paper by discussing our approach and related work.

2 The 4SRS model-driven method

The four-step rule set is a method aimed at supporting the transition from system requirements to software architectures and design elements. The method is essentially based on mapping UML use case diagrams into UML component diagrams. It uses an approximation by which a use case is realized by a collaboration of components of three kinds: interface, control, and data; as similarly suggested in reuse-driven software engineering business (RSEB) [6]. After this initial transformation, a series of steps with transformation rules are proposed to transform the initial component model into a coherent component model that is compliant with the requirements. Basically, at each step, a set of refactoring rules are applied that modify the initial component model by grouping, splitting or discarding components. Some of these rules can be automated; others depend on human intervention.

Figure 1 presents an overview of possible activities of model-driven methods aimed at the development of product lines. These are also the activities that compose the 4SRS method. In this paper we will only cover the activities that are marked in gray (b, c, e, f, g, and h). For clarity reasons, some details are left out of the diagram of Fig. 1. For instance,

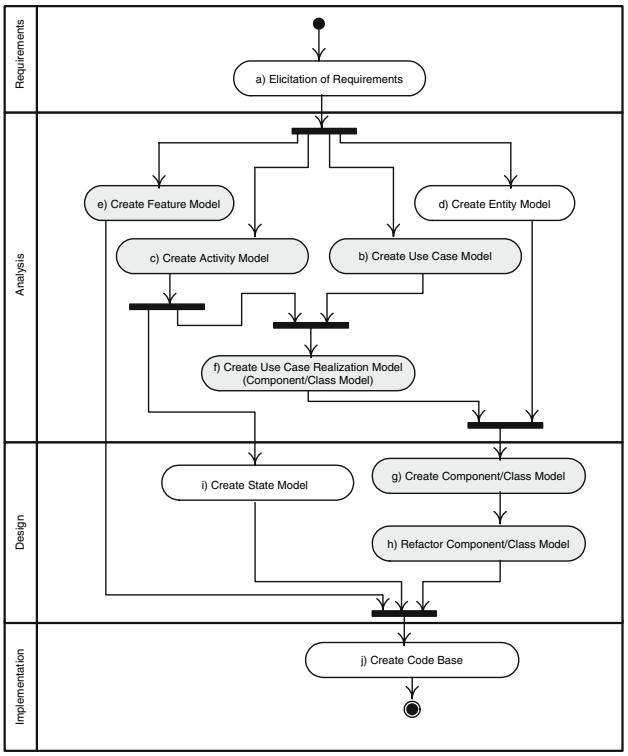

Fig. 1 Simplified view of the major activities of the 4SRS method

activities b, c, d, and e can be done in parallel but are not independent; they require coordination between them.

In 4SRS, as transformation rules are applied and elements are created, their names are prefixed with a special codification enclosed in brackets that is used to guarantee uniqueness and facilitate visual identification. The 4SRS transformation rules are applied through several of the activities presented in Fig. 1. Briefly, the steps of the method are: (1) component creation; (2) component elimination: (2i) use case classification, (2ii) local elimination, (2iii) component naming, (2iv) component description, (2v) component representation, (2vi) global elimination, (2vii) component renaming; (3) component packaging and aggregation; and (4) component association.

The method has been applied (and adapted) in several cases, from e-government [3] to protocol processing applications [7]. Experimental work on adapting the method to handle variability in the context of product lines has also been presented in the past [8]. From these previous works we have identified two fundamental issues regarding UML that require further clarification in order to fully integrate use case models into the 4SRS method. These issues are the semantics of the use case relationships (*include*, *extend*, and *generalization*) and the formalization of use case behaviors.

These two issues are addressed by an extension to the UML metamodel proposed in a previous work of the authors [9]. The major adaptation proposed was that *extend* relationships may require rejoin points that are different

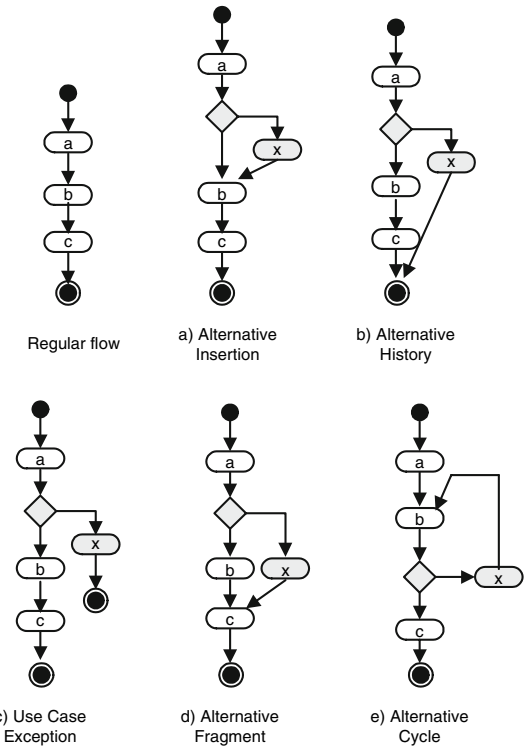

Fig. 2 Types of alternative sequences of actions in use cases

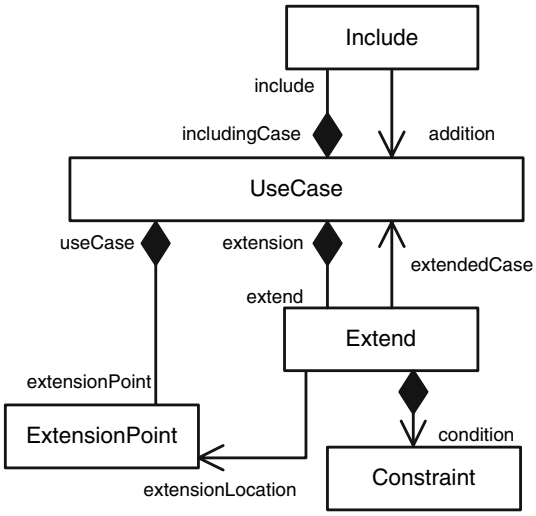

Fig. 3 Excerpt UML 2.0 use case metamodel

from the original extension point in order to support all kinds of alternative sequence of actions presented in Fig. 2. In the next section, we elaborate on our proposal to handle variability in use cases.

3 Handling variability in use cases

To fully integrate requirements into model-driven approaches the requirement model has to be formalized. In the case of UML this means the formalization of use cases. In software product lines, a vital concern is the specification of variability. Figure 2 presents types of alternatives (variability in action flows) that are common in the textual description of use cases. The UML metamodel does not support all these types of alternatives. In this section we address this limitation and propose an extension to the UML metamodel to support model-driven methods with such requirements for variability modeling. For the formalization of the behavior of use cases we propose the adoption of activity diagrams.

3.1 Use case relationships

According to the UML metamodel, use cases can be associated by two major relationships: *Include* and *Extend*. The *Include* relationship is used when there are parts (behavior) of use cases that are common. When this happens, the common part can be extracted to a new use case. The new use case is related to the original use cases by an *Include* relationship. According to the UML specification, an *Include* relationship acts like a procedure call, since the location of the inclusion of flow is coincident with the location of the rejoin of the flow. The *Extend* relationship is used to model behavior that can extend the behavior of a base use case. There is always a condition associated with an *Extend* relationship. This condition is used to specify when the extension will be active. Figure 3 presents an excerpt of UML metamodel regarding use cases. From the presented semantics and the metamodel, it is clear that in UML the *Extend* relationship is more suited to model variability because a condition can be associated with the extension. Nevertheless, other approaches have been proposed. For instance, Gomaa [10] proposes that the *Include* relationship can be used to model optional use cases in product lines.

When developing software product lines, features and feature diagrams are also commonly used to model variability. Features represent user-visible aspects or characteristics of a domain [11]. When they represent functional characteristics of a product line they can be related to use cases. Usually a feature can be modeled by one or more use cases [10,12]. However, in the case of fine-grain features, it is possible that a use case encompass several features. Features can be mandatory, optional or alternative. There can also be dependency relations between features. Optional and alternative features represent variability in a product line. These could be modeled with the *Extend* relationship.

Usually, use cases are described in natural language. In fact, there is a pattern for the textual description of use cases that is generally accepted by practitioners [13]. In this pattern, use cases are composed by sequences of steps, or actions. There is usually one main sequence and many alternative

Use case **Renew Loan**:
- Main flow:
1. The Librarian enters the renew loan data (user ID and Item ID)
2. The system retrieves loan info
3. The loan info is displayed to the librarian
4. The system retrieves item info «extension point»
5. The system renews the loan «extension point»
Use case ends

- Alternative flows:
2a. Loan does no exist (after step 2)
 2a1. The system displays a message to librarian
 Use case ends
3a. A fine is due (after step 3)
 3a1. The librarian collects the fine «extension point»
 Use case rejoins (before step 4)
 Alternative flows:
  3a1a The fine is not totally paid (after step 3a1)
   3a1a1. The system displays a message to the librarian
   Use case ends
4a. The item is reserved (after step 4)
 4a1. The system displays a message to the librarian
 Use case ends

Fig. 4 Excerpt of use case *Renew Loan*

Use case **Handle Gold Member**:
- Main flow: <empty>

- Alternative flows (Extension flows):
1. Handle Renew Loan
 Condition: MemberType=GoldMember
 1a. Handle Collect Fine
  (before *Librarian collects the fine*):
  1a1. If fine<member fee Rejoin base use case
   (before *Retrieve item info*).
  Rejoin base use case (before *Librarian collects the fine*).
 1b. Handle Borrow Rule
  (after *Retrieve item info*):
  1b1. If Item Reserved by non-gold member Rejoin base use case (before *Renew loan*)
  1b2. Display a message to the librarian
  Base use case ends

Referenced *Extension Points*:
-Librarian collects the fine:
 Renew Loan.The librarian collects the fine
-Retrieve item info:
 Renew Loan.The system retrives item info
-Renew loan: *Renew Loan.The system renews the loan*

Fig. 5 Excerpt of use case *Handle Gold Member*

sequences. There are five types of alternative sequences: conditional insertion, use case exception, alternative history, alternative part, and alternative cycle [14]. Figure 2 presents a graphical representation of the possible sequence alternatives.

An alternative insertion (Fig. 2a) is used to represent conditional behavior that is inserted into a precise point (extension point) of a flow. In this case the insertion point is coincident with the rejoin point, i.e., at the end of the alternative behavior the flow rejoins the main flow at the initial extension point. This is very similar to an *Include* relationship with a condition of insertion. Alternative insertions can be easily modeled by *Extend* relationships because the extension point and the rejoin point are coincident. In contrast, the other types of alternatives (alternative history, use case exception, alternative fragment, and alternative cycle) are not directly supported by the UML use case metamodel (Fig. 2). This is an important limitation since, in practice, it is not so unusual for extensions to have flows that are diverse from that of an alternative insertion.

To illustrate our approach we will consider the example of a library system and two use cases of that system, as presented in Figs. 4 and 5.

In Fig. 4, where the behavior of the use case *Renew Loan* is described, there are two types of alternatives: exceptions (2a, 4a, and 3a1a) and alternative flow (3a). These are internal alternatives of the use case. One of the reasons to use the *Extend* relationship is that it provides a way to extend already developed use cases with optional or alternative external behavior without interfering with them (it only requires the proper identification of the extension points in the extended use cases). This is a common process used to develop use cases: first identify and model usual and common behavior, and only then reason about alternative and optional behavior. Alternative and optional behavior is usually related to *entity* instances and values of their attributes that imply different behavior from one of the base use cases. The *member* type is an example of an entity type of the library domain that implies variability of behavior in use cases.

Figure 5 presents an excerpt of the description of the extending use case *Handle Gold Member*. This excerpt contains only the extending behavior that regards use case *Renew Loan*. *Handle Gold Member* extends *Renew Loan* and uses the three extension points defined in *Renew Loan*.

We adopt a similar structure to specify regular and extending use cases, even if it is not common to have extending use cases with main flow. Also, we describe alternative flows and extension flows in the same section because their structure is basically the same, the major difference being the fact that in an extending use case the alternative flows usually relate to flows of other use cases.

Figure 5 presents common situations that reflect two types of alternatives that are not adequately handled by the *Extend* relationship of UML:

- The extending use case adds conditional behavior that can result in an alternative flow (*1a. Handle Collect Fine*

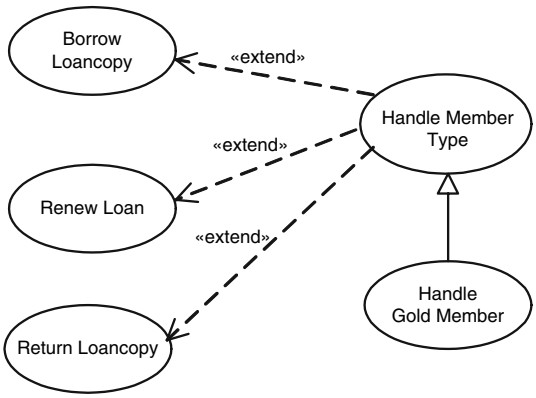

Fig. 6 Modeling *MemberType* as a dimension of variability in use cases

and *1b. Handle Borrow Rule*), i.e., there are rejoin points that do not match the original extension point;
- The extending use case adds conditional behavior that can result in an alternative history (*1b. Handle Borrow Rule*), i.e., the new behavior can lead to an alternative ending in the base use case.

These situations could be handled by the incorporation of the alternatives into the base use case. Nonetheless, this option would lead to base use cases that would be difficult to read and understand. It would also become more difficult to handle variability in a product line, because important features, such as *MemberType*, would be disperse into several use cases. Our approach (Sect. 3.2) enables an effective way to model alternative flows that facilitates the process of discovering new dimensions of variable features and easily integrating them into the use case model.

For instance, Fig. 6 presents how the alternative behaviors related to *MemberType* can be modeled in such a way. In this example, *MemberType* was identified as a dimension of variability because alternative behaviors that depend on the type of the member of the library need to be incorporated into several base use cases. In order to address future new types of members we can model the extensions regarding member type in a single abstract use case. This type of abstract use case represents a dimension of variability. Specific instances of that variability dimension are modeled as sub-use cases. As such, in our example, *Handle Gold Member* could become a specialization of the abstract use case *Handle Member Type*.

The use of the generalization/specialization relationship between the extending use cases (in our example, *Handle Member Type* and *Handle Gold Member*) enables the proper handling of dimensions of variability. The abstract extending use case acts as a template for the concrete extending use cases. For instance, the conditions of the *Extend* relationships are only specified in the concrete use cases.

As Fig. 3 shows, the UML metamodel only supports extensions that are basically conditional *Include* relationships. This represents a major limitation to the modeling of the diverse variability types that are commonly specified by textual use cases. In fact, it only supports alternative type *a* (Fig. 2a). In the next section we present and discuss a proposal of an extension to the use case metamodel that addresses the modeling requirements identified in this section.

3.2 Extending the UML 2.0 metamodel

In the past, several proposals have been made to formalize use cases [15–18]. Some more recent works also proposed approaches to manage variability in use cases in the context of product lines [19,20]. The main concern of their authors has been the lack of formalism of the usual use case text descriptions. Most well-known proposals regard nonvisual languages. In our specific case we aim at integrating requirements into a model-driven method. In the context of UML, the modeling of behavior can be addressed by activity diagrams, so we have adopted activity diagrams for modeling use case behavior. Figure 7 presents an excerpt of the UML metamodel adapted (extended) to support our proposal for formalization of use cases.

Figure 7 presents in gray those elements that correspond to extensions to the UML metamodel. Since, according to the UML specification, a use case is a specialization of a *BehavioredClassifier*, we use the *classifierBehavior* and *ownedBehavior* associations to model, respectively, the use case main flow and the alternative flows.

We propose a new *ExtensionFragment* metaclass to support the issues identified in the previous section. In our proposed metamodel it is clear that an *Extend* relationship can have a condition and make several extensions (via *ExtensionFragment*) to a base use case. Each extension has one extension location but can have several rejoin locations. An extension also specifies which behavior of the extending use case will extend the base use case in the extension location. Since use case behaviors are formalized through activity diagrams, extension locations and rejoin locations refer to elements of type *Action* of the corresponding behavior. To clarify if the extension or the rejoin points are made before or after the corresponding *Action*, we propose the attribute *moment*.

Regarding the *Include* relationship, we propose the new *InclusionPoint* element so that we have a similar approach to the one used in the *Extend* relationship. An *InclusionPoint* refers to the location where the behavior is to be included. This location has to refer to an element of type *CallBehaviorAction* of the same use case as the *Include* relationship. It is not necessary to specify what behavior is to be included because the semantic of the *Include* is to include the main behavior (*classifierBehavior*) of the included use case. The stereotypes *extension_point*, *inclusion_point*, and *rejoin_point* are used as a visual aid to more easily identify the semantics of the actions nodes of the activity diagrams.

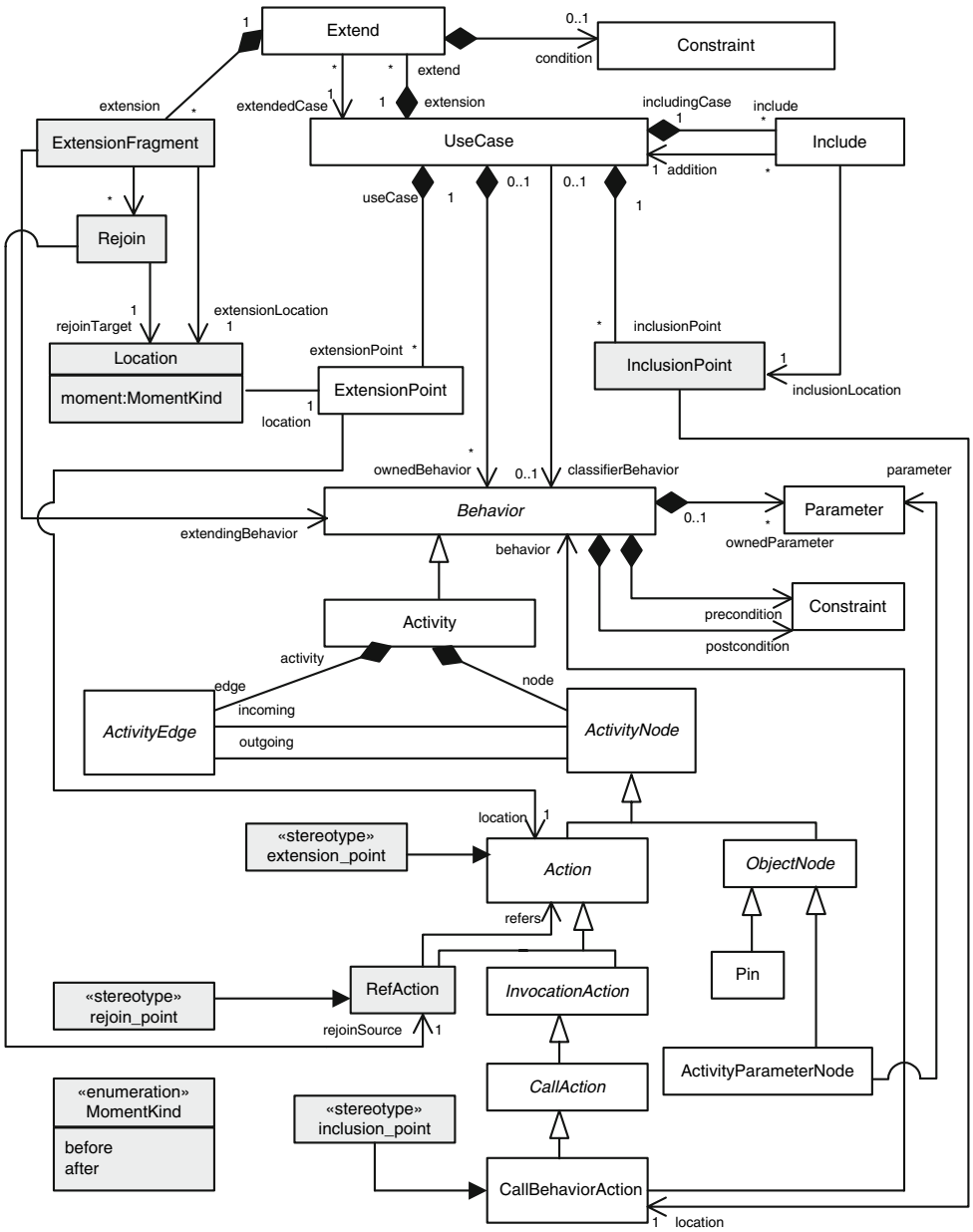

Fig. 7 Excerpt of proposed metamodel

The formalization of use case behavior by means of activities is also depicted in Fig. 7. For each use case behavior there is an activity. Extend and rejoin points of use cases trace directly to nodes of activities. In Sect. 4 we elaborate more on how activity diagrams relate to use cases and how they can be constructed.

3.3 Relating use cases and features

Although feature diagrams [11] are not part of the UML metamodel, they are a crucial artifact of product lines. They model the characteristics of a product line and how they relate to each other. A selection of the features represents a particular application of the product line. As such, features should relate to the requirements of the product line.

The issue of relating use cases and features is not new. Notably, there is the much referenced work of Griss et al. [12]. In their work they propose an approach by which functional features are extracted from the *domain* use case model. They also propose that the structure of the feature model can be created according to the structure of the use case model (by using the *include* and *extend* relationships). As the authors suggest, further types of features can be discovered and added along the development process, such as features resulting from architectural or design modeling tasks. More recent works in this field are also aligned with this approach

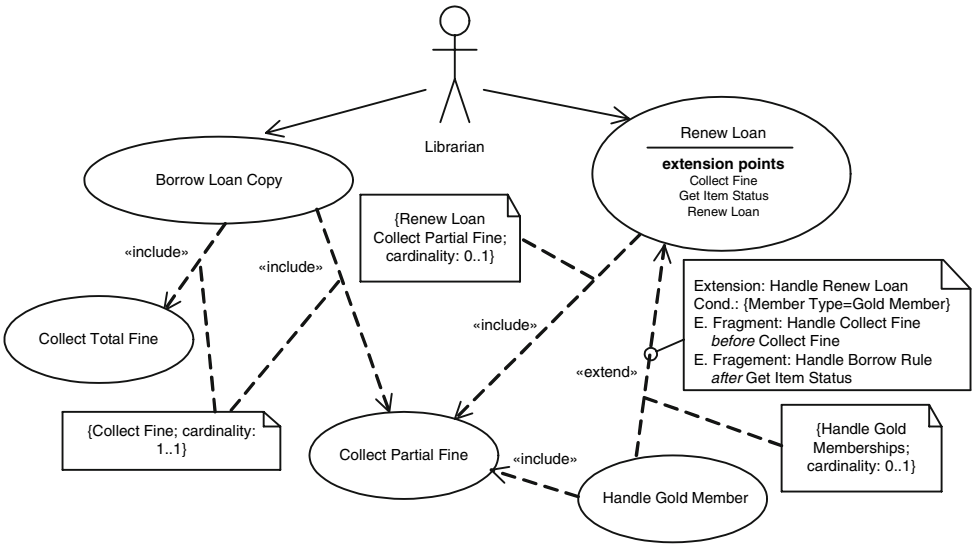

Fig. 8 Example of a use case diagram for a library product line

[10,20,21]. We also follow this approach, since feature modeling requires an extensive knowledge of the domain, which is only possible after the effective modeling of such a domain. This is particularly true for the functional features of the domain. So, the initial feature model can be built from the domain use case model as described [22].

Figure 8 is an example of a use case diagram that contains visual annotations that are used to model how variability in use cases can be combined to represent the functional features of a domain or of a software product line. We call these annotations variability annotations and the annotated use case models *use case domain models*. In Fig. 8, variability annotations are represented as notes linked to the *Include* and *Extend* relationships. They represent variability points with a name, a minimum, and a maximum cardinality and the respective options.

For instance, the variability annotation *Collect Fine*, has a cardinality `1..1` that says that one and only one of the options must be selected. The two options are the includes that related the use case *Borrow Loan Copy* to the included use cases *Collect Total Fine* and *Collect Partial Fine*. By annotating the use case model with this approach it is possible to automate the creation of a feature model such as the one presented in Fig. 9 [22]. Since it is the modeler of the use case domain model who is editing these use cases and relationships, he/she is also capable of making these annotations.

3.4 UML-F

UML-F was proposed as a UML profile for frameworks [5], and later support was added to product lines [23]. With this profile it is possible to annotate UML elements with stereotypes that properly model variability. Unfortunately, the original UML-F profile only covers design elements.

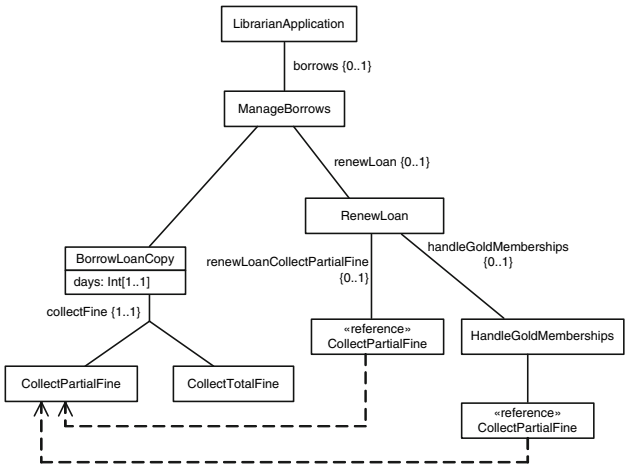

Fig. 9 Excerpt of a library feature model

The 4SRS method adopts the UML stereotypes proposed in UML-F and extends the UML-F profile with new stereotypes to include support for requirements and analysis models. Table 1 summarizes these stereotypes and informally defines their semantics.

4 Modeling requirements with use cases and activity diagrams

UML use cases are very useful in capturing requirements because of their simplicity but, as discussed in the previous section, they also have some informal characteristics that hinder their adoption in model-driven methods. To address these issues, 4SRS extends the UML use case metamodel and adopts *Activities* to formally specify use case behavior. In 4SRS, for each use case behavior there is an activity diagram.

Table 1 Summary of UML-F stereotypes and their meanings

Stereotype	Applies to element	Description
variant	UseCase	Indicates that the behavior of the use case can vary
mandatory	UseCase	Classifies use cases according to their *inclusion* in the product line
optional		
alternative		
inclusion_point	Action	Indicates that the *Action* is an inclusion point for the classifierBehavior of the included use case
extension_point	Action	Indicates that the behavior of the use case can be extended at the *Action*
vp	ExtensionPoint	Indicates that the *ExtensionPoint* is a variation point of the product line
template	«analysis»Component	Indicates an element whose behavior is affected by variants that relate to a hook (based on [23])
	«analysis»Class	
	DesignElement	
hook	«analysis»Interface	An element that represents (or contains) a location where variations occur, i.e., a variation point (based on [23])
	DesignElement	
rejoin_point	RefAction	Indicates that this *RefAction* rejoins the flow at the referenced *Action* of the base behavior
		Attributes: *Moment* (*before* or *after*)
application	DesignElement	Indicates that the *DesignElement* relates to a specific application of the product line (based on [23])
framework	DesignElement	Indicates that the *DesignElement* is global to all applications of the product line (based on [23]).
variable	Method	Indicates that the behavior of the method varies (based on [5])
		Attributes: *Instantiation* (*dynamic* or *static*)
extensible	Class	Indicates that new methods can be added to the class (based on [5])
		Attributes: *Instantiation* (*dynamic* or *static*)
incomplete	Generalization	Indicates that the generalization set can be incomplete, i.e., it is possible to add new classifiers to the set (based on [5])
		Attributes: *Instantiation* (*dynamic* or *static*)

When use cases of the domain are identified, their behavior is modeled by activity diagrams. This is not so different from the traditional way of describing use case behavior by natural language, such as in [13]. Basically, each step in a text description of a use case is modeled as an *Action* node in the activity diagram.

Sequence diagrams can be a helpful tool in modeling diverse use case scenarios that together describe the global behavior of a use case. Therefore, they also can help when building the activity diagrams for the behavior of the use cases.

Use case relationships are discovered during use case modeling. The informal *«include»* and *«extend»* relationships become formal as they are modeled in 4SRS, since they relate *Action* nodes and use case elements in a precise way (Fig. 7). Common use cases for diverse applications become mandatory use cases of the product line. Optional and alternative use cases will participate in *«extend»* relationships or, less commonly, in *«include»* relationships.[1] As such, during this process of modeling use cases, we are also identifying features of the product line. However, feature diagrams should not become direct mappings of use cases. For instance, all mandatory use cases could relate to a single root feature. In 4SRS, establishing relationships between features and use cases is a human task since it requires human decision. Nonetheless, it is possible for a tool to automate a significant part of this process, as described in [22].

The activity of the creation of the use case models is essential in any UML-based model-driven method, since use cases drive the creation of a very significant number of design elements, i.e., the design elements that are derived from functional requirements.

To illustrate this process we will describe how use case textual descriptions with specifications similar to the ones described in [13] can be modeled with our approach. Figures 4 and 5 are cases of such textual specification.

Use case specifications usually contain main flow descriptions and alternative flows. For the construction of the use case models it is usual to address first regular use cases (i.e., nonextending use cases) and *Include* relationships. As the use case model is constructed, it is possible to start also developing the activity diagrams. For each regular use case, usually a single activity diagram is sufficient. According to the

[1] Since an *including* use case is aware of the existence of the *included* use case some precautions are necessary when modeling variability with the *include* relationship. This is not the case for *extending* use cases, since the *extended* use case is not aware of being extended.

metamodel of Fig. 7, there should be an activity diagram for each *Behavior* (*ownedBehavior* or *classifierBehavior*) of the use case. Since the *classifierBehavior* specifies the main flow of a use case, and main flow alternatives can also be specified in the *classifierBehavior*, a single activity diagram is sufficient for the majority of the use cases.

4.1 Activity nodes

The construction of the activity diagrams is relatively straightforward. The base idea is that each step of the use case textual description becomes an *ActivityNode* in the activity diagram. Each *ActivityNode* refers to, or is performed by, the system or an actor. As such, we adopt UML 2.0 *ActivityPartitions* associated with *ActivityNodes* to identify «who» is related to the *ActivityNode*. For instance, step 1 of Fig. 4, "The Librarian enters the renew loan data (user ID and Item ID)" becomes the *Action* (*ActivityNode*) *Enter Renew Loan Data* associated with the *ActivityPartition Librarian*. Figure 10 presents the activity diagrams correspondent to the flows of use cases *Renew Loan* (Fig. 4) and *Handle Gold Member* (Fig. 5).

4.2 Decision nodes

An alternative flow implies a *DecisionNode* in the activity diagram. The alternative flow "2a. Loan does not exist (after step 2)" of Fig. 4 is transformed into the *DecisionNode Check if Loan Exists*. This kind of *DecisionNode* usually has two outgoing edges. One corresponds to the main flow and is traversed when the condition for the alternative flow is false. The other corresponds to the alternative flow. Usually, decision nodes in UML 2.0 are depicted with a diamond-shaped symbol. We represent all activity nodes in a uniform way. To identify control nodes we represent their symbols as small icons within the right side of the node's visual symbol. This makes it possible to attach more information to control nodes (such as stereotypes and partition names), making their visual representation more meaningful.

4.3 Object nodes

A very important aspect of using activity diagrams to model use case behavior is that it is possible to represent object nodes and their flow. The process of identifying the objects that are used as parameters of actions and behaviors can provide significant input to the *entity model* of the domain (see activity *Create Entity Model* of Fig. 1). Another aspect of object nodes and parameters is that they provide an effective way to validate *Include* and *Extend* relationships, since the parameters of the sources and targets of these relationships must be *compatible*. Also, when we reason about conditions for alternatives and *Extend* relationships, object nodes makes it possible to do so in a formal way, because they provide a way to constraint the modeler to only refer to objects that are *accessible* from the specific location of the condition in the activity diagram. These are all validations that are possible in our proposed metamodel.

4.4 *Include* relationship

An *Include* relationship in which use case A includes use case B, means that there is a *CallBehaviorAction* node is use case A that calls the *classifierBehavior* of use case B. This means that the main flow of use case B is included by the *CallBehaviorAction* node of use case A. The parameters of the *CallBehaviorAction* of use case A must be *compatible* with the parameters of the *classifierBehavior* of use case B.

4.5 *Extend* relationship

In order to support all possible alternative flows, our *Extend* relationship becomes significantly more complex than the original UML *Extend* relationship. Regarding the extended use case, there are no significant changes. Basically, we only have to specify the extension points. In the case of the activity diagrams, this is done by marking the respective nodes with the stereotype *extension_point*. These become locations that can be used by extending use cases as extension or rejoin points. We maintain the term "extension point" in the base use case to expose nodes that can be used either as outgoing flows or as incoming flows of extending behavior. Maybe a more appropriate term would be "public point." Figure 8 presents three examples of *extension_points* for the use case *Renew Loan*: *Collect Fine*, *Get Item Status*, and *Renew Loan*.

Similarly to this simple case, it is possible to relate activities, use cases, and features as the analysis models are built. In fact, as use case behavior is modeled by activity diagrams, a trace can be made from elements of the activity diagrams to features, through use case elements. In Fig. 10, the activity nodes marked with the stereotype *«extension_point»* relate to the correspondent use case extension points with the same name. And, as we saw, these extension points relate to the *handleGoldMemberships* subfeature relationship (Fig. 9). Similarly, the alternative behaviors of use case *Handle Gold Member* that extend the use case *Renew Loan* at the previous mentioned extension points must be related to the feature *HandleGoldMemberships*.

Nodes of the 4SRS activity diagrams are marked with additional information that is used to support the 4SRS transformation rules. The stereotypes *«interface»* , *«control»* , and *«data»* are used to classify each node regarding its semantic role in the system. For instance, the node *Find Loan*

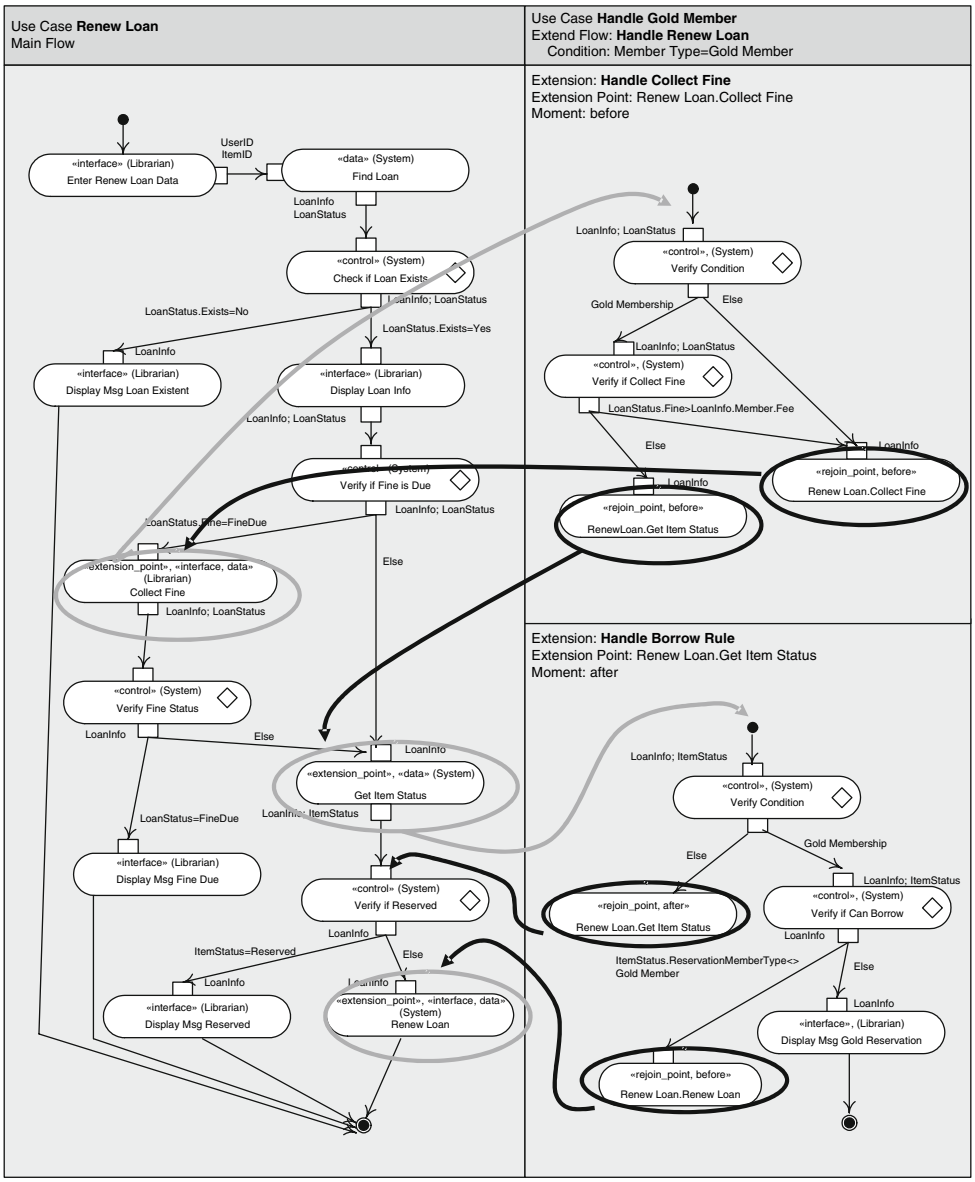

Fig. 10 Activity diagrams for *Renew Loan* and *Handle Gold Member*

represents a data operation in the system. We also use the concept of partitions to model "*who*" has the responsibility for the operation of the node or is the major "*actor*" of the node. For instance, the *System* is responsible for the node *Find Loan*.

5 Capturing functional architectural requirements with use case realizations

The adoption of activity diagrams for modeling use case behavior results in a precise specification of the functional requirements of the product line. A use case realization model acts like a *link* between the problem domain and the solution domain. It has the responsibility of guaranteeing that all functional requirements of the problem domain are addressed in the solution domain, on a use case by use case basis.

A use case realization is usually modeled by a group of *analysis* components/classes that collaborate to perform the use case behavior. They represent the first step in the transition from the problem space to the solution space. As such, they should be the primary (eventually the only) input for the software engineer as he/she designs the product line.

The first task of the design is the specification of the architecture of the product line, i.e., the collection of computational components of the product line and the interactions between these components. These elements can be essentially derived based on the input of use case realizations. Other requirements may also influence the architecture. For instance, the architectural style of a product line can be

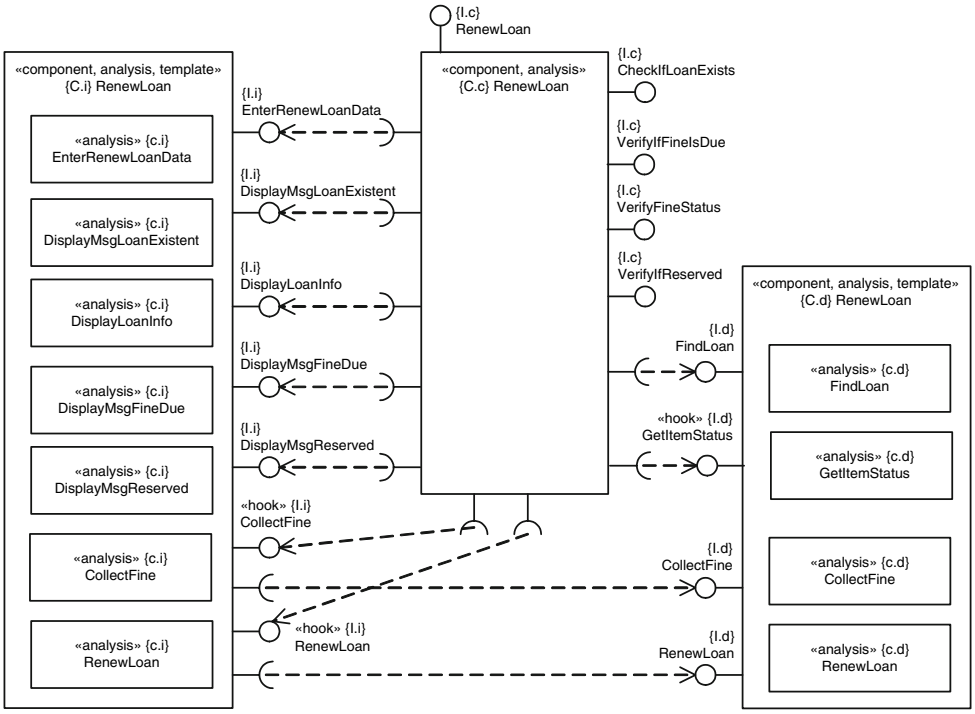

Fig. 11 Use case realization diagram for *Renew Loan* (filtered view)

influenced by the specific runtime platform, the topology of the hardware, etc. In this paper we will only address the functional requirements for the architecture.

Traditionally, the specification of use case realizations is a very creative task. It requires a lot of experience from the software engineer, as he/she identifies the classes that realize the use case from the use case textual description. However, even a very experienced software engineer can misinterpret the requirements or forget some specification. What 4SRS proposes is the automation of this task.

The automatic creation of use case realizations in 4SRS is possible since use case behavior is totally specified by activity diagrams. The annotations made on the activity diagrams (as described in the previous section) also support this automatic transformation. Basically, the method follows a well-known applied practice introduced by Jacobson et al. [24] and creates three components for each use case: *interface*, *control*, and *data* components. This corresponds to step 1 of 4SRS (Sect. 2). Since, in our approach, each use case is complemented with activity diagrams, it is possible to use these activity diagrams to populate each of the use case components with classes and interfaces that are responsible for realizing the behavior associated with the nodes of the activity diagrams. To guide this transformation each node in the activity diagrams must be marked with the following stereotypes: *interface*, *control*, and *data*. This facilitates the allocation of the classes and interfaces to the three components that realize the use case.

For a model-driven approach to be feasible there must be simple and direct mappings between the models. In this case, the mapping is done between *Actions* of activity models and *Methods* of component/class models. For each *Action* in the activity model we create an *Interface* with a *Method* in the use case realization model and also a *Class* that implements the interface. In the use case realization model these interfaces act like *roles* that are needed in the final system to address the behavior required by the specific use case. Extension, inclusion, and rejoin points are realized through required/provided interfaces. Since the components are populated with classes and interfaces, steps 2i and 2ii of 4SRS (*component elimination*) can be automated, i.e., components with no allocated classes and interfaces can be eliminated. Steps 2iii, 2iv, and 2v are also addressed at activity *create use case realization model* (Fig. 1), but they require the human intervention.

Figures 11 and 12 presents the use case realization of, respectively, use case *Renew Loan* and use case *Handle Gold Member*. Since we are addressing design at an architectural level we find that component diagrams are more adequate than class diagrams to model use case realizations. As such, in 4SRS use case realizations are component diagrams. As we saw, each use case gives origin to up to three analysis components. Each one is composed by the classes and interfaces that resulted from the transformation of the activity nodes. For instance, for the use case realization of *Renew Loan*, the *Collect Fine* node gives origin to two analysis classes: *{c.i}CollectFine* and *{c.d}CollectFine* because the correspondent node was annotated with stereotypes *interface* and *data*. Based on the *extend* relationship that exists between the two use cases, it is also possible to automatically annotate the

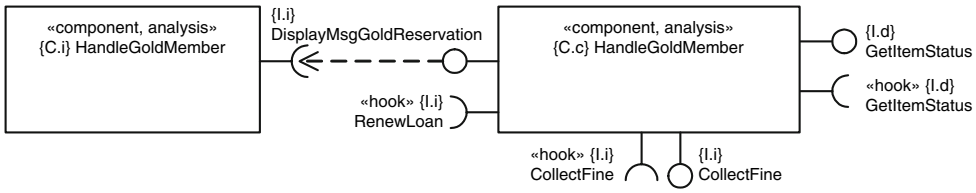

Fig. 12 Use case realization diagram for *Handle Gold Member* (filtered view)

use case realization elements with the *hook* and *template* stereotypes of UML-F.

Annotating these design elements with the stereotypes from UML-F enables the adoption of the design transformations proposed in UML-F to detail the logical architecture of the product line, as discussed in the next section.

The creation of use case realizations in 4SRS is a highly totally automated task. As such, all the generated elements are linked to their origins. This makes it possible to trace, for instance, a feature to its realization elements. Another advantage of this approach is that use case realizations can be easily transformed into executable code for a specific language or platform. Thus, use case realizations can also provide simple prototypes of the product line that can be of great help for the user validation of use cases.

One could argue that, with the proposed approach, a complex use case could give origin to a complex use case realization. This is true, but eventually this would also probably happen if the creation of use case realizations were not automated.

Besides the previously discussed characteristics of 4SRS use case realizations, the fundamental value of the method is that, as a result of a simple approach, the product line engineer is able to reason with precise functional architecture requirements.

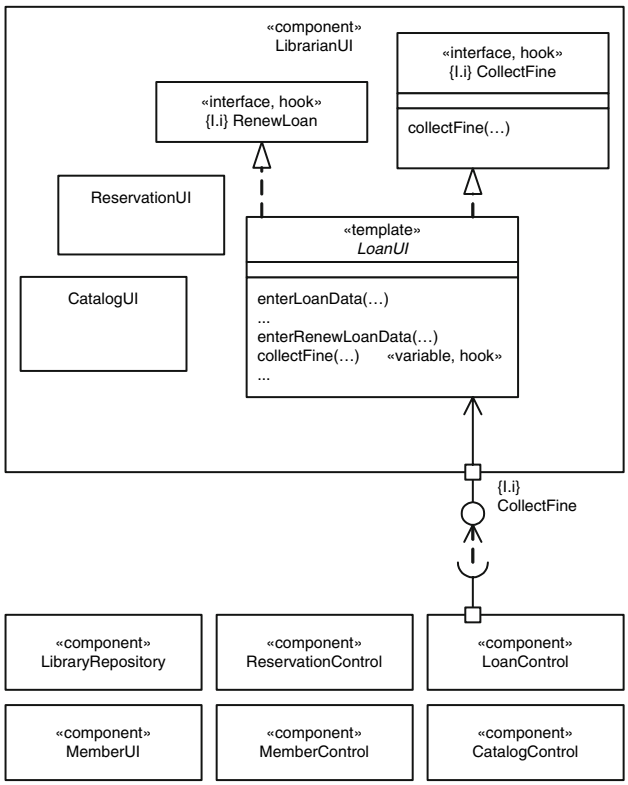

Fig. 13 Architectural logical view showing *{I.i}CollectFine* connecting the *LibrarianUI* and *LoanControl* components (filtered view)

6 Deriving the logical architecture

Each use case realization provides a partial view of the product line. It is necessary to create a global model of the architecture of the product line. In 4SRS, this global architecture is based on the use case realizations and represents an integration of them. In contrast to the creation of the use case realizations, the creation of the architecture is a human-based task.

In this task, the product line engineer has to transform the analysis elements that resulted from the use case realizations into design elements. Each analysis element gives origin to a new design element or is incorporated into an existing one. For instance, all *«interface»* analysis components that are related to the librarian role can be incorporated into the *LibrarianUI* design component.

Similarly, analysis classes give origin to new design classes or are incorporated into existing design classes. In Fig. 13, we can see that the analysis class *{c.i}CollectFine* was incorporated into the design class *LoanUI*. The only method of the class *{c.i}CollectFine* becomes a part of the *LoanUI* class.

To facilitate tracing to the original elements of the use case realizations, analysis interfaces are not transformed, i.e., analysis interfaces also exist in design. As the described transformations are performed the global architecture of the product line takes form. Associations between components become visible as matching required and provided interfaces are transformed from analysis to design. Variability points identified in the requirements and analysis phases are traceable to variability points in the architecture of the product line. For instance, the *hook* method *collectFine* of the *LoanUI* class originates from the *Collect Fine* extension point of use case *Renew Loan*.

As Fig. 1 shows, the design model is not based only on use case realizations. The entity model is also an input for the design model. Its classes also populate the design model.

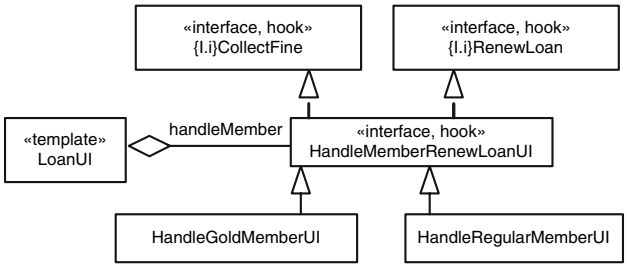

Fig. 14 Applying the *abstract factory* design pattern to realize the variability point of the *collectFine* method of the *LoanUI* class

The tasks described in this paper enable the creation of an initial version of the architecture of a product line that is traceable to requirements and incorporates all functional requirements (as they were modeled). Obviously, the design of a product line does not end with its architecture. More detailed design tasks are required to achieve the goal of activity *j* (Fig. 1): the creation of executable code. One example of such design tasks is the use of design patterns, such as the ones proposed initially by Gamma et al. [25].

Figure 14 is an example of the result of applying the *abstract factory* design pattern to realize the variability point of the *collectFine* method of the *LoanUI* class. Originally (Fig. 13), the *hook* (variability point), and the *template* were at the same class. The *abstract factory* design pattern separates the *template* from the *hook* [23]. This realization of the variability point supports changing variants at runtime. Such a design decision should be made in accordance with the requirements. In this case, for instance, the *bindingTime* attribute of the feature *HandleGoldMemberships* should have the value *runtime* (Fig. 9). If only static binding time were required, the creation of subclasses of *LoanUI* would be sufficient to support the variability point. More details about how feature characteristics can influence the design can be found in [23].

7 Conclusion

Use case realizations are a technique used to help the transition from the problem domain to the solution domain. The 4SRS method also adopts this technique. The particularity of 4SRS is that its use case realizations can be totally automated. This is possible because use case behaviors are formally modeled with activity diagrams and also because of the adaptations made to the UML metamodel. These adaptations support the proper modeling of variability in all the activities of the method. As a result of the 4SRS approach, it is possible to maintain traces between elements at different conceptual levels. In the case of product lines, use case requirements are traced to architectural requirements, use case variability is traceable into architectural elements, and features are related to architectural elements.

Transforming requirements to analysis models is a very difficult task since the semantic gap between the problem and solution domain is usually very significant. Usually, methods take for granted that the analyst is a domain expert and will discover the architectural elements without difficulty or that the architectural elements can be easily discovered by applying simple heuristics, such as transforming nouns into objects and verbs into operations. We can find this kind of approach in methods such as RSEB [6,12], PLUS [10], and Catalysis [26]. In real projects the transformation process is much more difficult. Our approach (as well as the one of the original 4SRS) has more similarities with methods which make the initial transformation based on objects and not classes (e.g., Fujaba [27]). With this kind of approach, use cases are first realized by object collaboration diagrams. Only after realizing all use cases with object collaborations do we proceed to create class models. Our approach goes even further since the use case realizations are based on performing the actions of the activity diagrams essentially by fragments of analysis classes. These fragments must exist to realize the system but are only transformed into design elements after the initial architecture of the system is completed. With this approach we aim to facilitate the work of the analyst since prior domain knowledge or architectural experience is not required to build the first complete architecture of the system. One relative drawback of our approach is that it requires metamodeling tools for the adaptation of the UML metamodel. Tool support for 4SRS is being developed experimentally with EMF [28] and GMF [29]. Details on how the approach presented in this paper can be applied using publicly available tools can be found in [9,22,30–32].

References

1. OMG (2005) MOF QVT final adopted specification. OMG, Available at http://www.omg.org/
2. Clements P, Northrop L (2002) Software product lines—practices and patterns. Addison Wesley, Reading
3. Machado RJ, Fernandes JM, Monteiro P, Rodrigues H (2005) On the transformation of UML models for service-oriented software. In: ECBS international conference and workshop on the engineering of computer based systems. Greenbelt, MD
4. OMG (2006) Formal/05-07-04 unified modeling language version 2.0: superstructure. OMG, Available at http://www.omg.org/
5. Fontoura M, Pree W, Rumpe B (2000) UML-F: a modeling language for object-oriented frameworks. ECOOP 2000-object-oriented programming conference
6. Jacobson I, Griss M, Jonsson P (1997) Software reuse: architecture, process and organization for business success. Addison Wesley Longman, New York
7. Marcus Alanen, Lilius J, Porres I, Truscan D (2005) On modeling techniques for supporting model driven development of protocol processing applications. In: Beydeda S, Book M, Gruhn V (eds) Model driven software development—vol II of research and practice in software engineering, vol 2. Springer-Verlag, New York, pp 305–328

8. Braganca A, Machado RJ (2005) Deriving software product line's architectural requirements from use cases: an experimental approach. In: Second international workshop on model-based methodologies for pervasive and embedded software. Rennes, France
9. Braganca A, Machado RJ (2006) Extending UML 2.0 metamodel for complementary usages of the «extend» relationship within use case variability specification. SPLC 2006. Baltimore, Maryland
10. Gomaa H (2005) Designing software product lines with UML. Addison Wesley, Reading
11. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study technical report, CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University
12. Griss ML, Favaro J, d'Alessandro M (1998) Integrating feature modeling with the RSEB. In: Fifth international conference on software reuse. Victoria, Canada
13. Cockburn A (2001) Writing effective use cases. Addison-Wesley, Reading
14. Metz P, O'Brian J, Weber W (2004) Specifying use case interaction: clarifying extension points and points of rejoin. J Object Technol
15. Hurlbut R (1998) Managing domain architecture evolution through adaptive use case and business rule models. Ph.D. at Graduate College, Illinois Institute of Technology, Chicago
16. Overgaard G, Palmkvist K (1998) A formal approach to use cases and their relationships. In: «UML» 98: beyond the notation, ecole superieure des sciences appliques pour l'Ingenieur, Mulhouse. Universite de Haut-Alsace, France
17. Porres I (2001) Modeling and analysing software behavior in UML. Ph.D. at Department of Computer Science, Abo Akademi University, Turku, Finland
18. Stevens P (2001) On use cases and their relationships in the unified modelling language. FASE'01
19. Fantechi A, Gnesi S, Lami G, Nesti E (2004) A methodology for the derivation and verification of use cases for product lines. SPLC2004, Boston
20. Eriksson M, Borstler J, Borg K (2005) The PLUSS approach—domain modeling with features, use cases and use case realizations. SPLC2005, Rennes, France
21. Jacobson I, Ng P-W (2005) Aspect-oriented software development with use cases. Addison Wesley, Reading
22. Braganca A, Machado RJ (2007) Automating mappings between use case diagrams and feature models for software product lines. SPLC 2007, Kyoto, Japan
23. Pree W, Fontoura M, Rumpe B (2002) Product line annotations with UML-F. In: Software product lines—second international conference. SPLC 2, San Diego
24. Jacobson I, Christerson M, Jonsson P, Overgaard G (1992) Object-oriented software engineering: a use case driven approach. Addison-Wesley, Reading
25. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns—elements of reusable object-oriented software. Addison-Wesley, Reading
26. D'Souza DF, Wills AC (1998) Objects, components, and frameworks with uml: the catalysis(SM) approach. Addison-Wesley Professional, Boston
27. University of Paderborn Software Engineering Group, 2006, Fujaba "Fujaba", vol 2007. http://wwwcs.uni-paderborn.de/cs/fujaba/index.html
28. Eclipse (2006) "Eclipse Modeling Framework," Eclipse Foundation. Available at http://www.eclipse.org/emf/
29. Eclipse (2006) "Graphical Modeling Framework," Eclipse Foundation. Available at http://www.eclipse.org/gmf/
30. Braganca A, Machado RJ (2007) Adopting computational independent models for derivation of architectural requirements of software product lines. Mompes, Braga, Portugal
31. Braganca A, Machado RJ (2008) Transformation patterns for multi-staged model-driven software development. SPLC 2008. Limerick, Ireland
32. Bragança A (2008) Methodological approaches and techniques for model driven development of software product lines. Ph.D. at Information Systems Department, School of Engineering, University of Minho, Braga