Modeling Dynamic Software Components in UML

Axel Wienberg, Florian Matthes[1], and Marko Boger[2]

[1] Software Systems Institute (AB 4.02)
Technical University Hamburg-Harburg, Germany
http://www.sts.tu-harburg.de
[2] Distributed Systems Group, University of Hamburg, Germany
http://vsys-www.informatik.uni-hamburg.de

Abstract. UML provides modeling support for static software components through hierarchical packages. We describe a small extension of UML for modeling *dynamic* software components which can be instantiated at runtime, customized, made persistent, migrated and be aggregated to larger components. For example, this extension can be used to describe systems built with JavaBeans, ActiveX-Controls, Voyager Agents or CORBA Objects by Value. With our extension, the lifecycle of a dynamic software component can be expressed in terms of UML. We can not only describe a system at design time, but also monitor its runtime behaviour. A re-engineering tool is presented that exploits our UML extension for a high-level visualization of the interaction between dynamic components in an object-oriented system.

1 Introduction and Motivation

In academia and industry, the concept of a *software component* [Szy98,Gri98] as a self-contained, persistent, customizable and large-grain building block for (possibly distributed) application systems has attracted a lot of interest.

In our research work on orthogonally persistent and mobile object systems [MS94,MMS96,BWL99], we encountered the need to model and to visualize the state and the behavior of a (possibly distributed) system which consists of a large number of objects, some of which are aggregated to *dynamic software components*. These software components are dynamic, because they can be instantiated at runtime, customized, made persistent and be migrated within a dynamic component hierarchy. We successfully applied UML for this modeling task and we were also able to develop a visualization tool (a debugger extension) to monitor the state and the behavior of such systems by means of UML diagrams.

However, it turned out that the existing notations of UML for static components (e.g., packages, deployment diagrams) are not suited for this task and that we had to extend UML slightly by notations for component links, component boundaries (see Sec. 2.1 and Sec. 2.2) and for component aggregation (see Sec. 3.4) to smoothly integrate components into UML object, class, collaboration and sequence diagrams.

Our minimal UML extensions have been chosen deliberately to capture the common semantics of the growing number of industrial component models (e.g., JavaBeans, ActiveX-Controls, Voyager Agents, CORBA with Objects by Value) while leaving room for their differences: Components may or may not consist of further components internally (defining a hierarchic structure as in Java beans or a flat structure as in CORBA); they may be able to migrate as in Voyager or have a fixed location as in CORBA; components may or may not be first class objects, allowing parameter passing of components and substitution of objects for components; and components may be active (running their own thread of control) or passive (waiting for incoming messages), depending on the model used.

The common characteristics of these components are their ability to communicate by sending and receiving messages, the possibility of having multiple components of the same type (class) in a system, and the requirement that a component is the unit of co-location, i.e. that all objects of a component reside completely on one node. Finally, the state of a component consists of the attributes and objects aggregated by the component.

In Section 2, we first introduce the notion of component links and component boundaries to identify components and component hierarchies in object diagrams. The gain in modeling power through this extension in collaboration diagrams, sequence diagrams and class diagrams is discussed in Section 3. As a practical application, in Section 4 we present a re-engineering tool that visualizes component behaviour by monitoring component boundaries at runtime. Section 5 briefly discusses ways to identify component links in existing component systems. After relating our work to that of others, we conclude with a summary.

2 Modeling Component Configurations with UML

In this section, we introduce a notation for dynamic components. A dynamic component is a runtime entity based on objects, and is therefore shown in an object diagram. The configuration of a dynamic component includes its relation to other components, as well as the set and structure of objects internal to the component. Using our notation, these aspects can be expressed in UML.

2.1 Component Links

The fact that a component is made up of certain objects which implement its functionality implies an aggregation of the objects to a component. We shall model this specific form of aggregation using *component links*.

Definition *component link* A form of aggregation link between two objects, a *component* and a *subcomponent*, with the additional semantics that each object is an immediate subcomponent of at most one component. Therefore, the graph of objects and component links forms a forest. A component link may change over time and implies no dependence of lifetimes.

Notation In an object diagram, a component link is represented using the stereotype ≪component≫, displayed graphically as a link with a half-filled diamond on the side of the component. Fig. 1 gives an example.

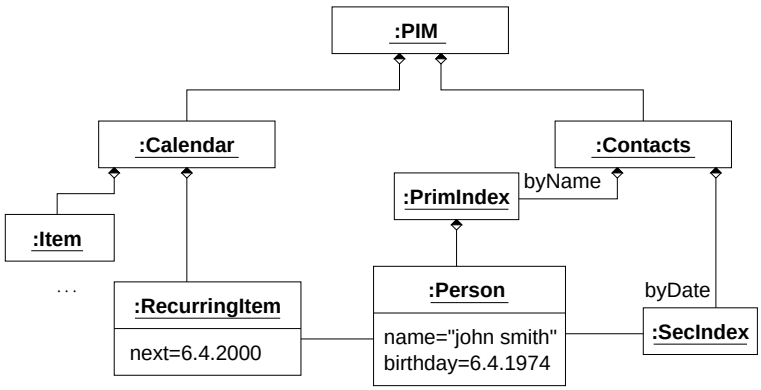

Fig. 1. A UML object diagram with component links

Fig. 1 shows the example of a personal information manager component (PIM) consisting of two independent subcomponents, a contacts database and a calendar, possibly implemented by different vendors. Internally, the contacts database stores its person records in a primary index by name, so that the person object is a subcomponent of the primary index. The secondary index only holds an association link to the person object.

2.2 Component Boundaries

Using component links emphasizes the relation between a component and its immediate subcomponent. However, we also want to show which subcomponents belong together to implement a given component.

This grouping of peer objects cannot be denoted easily through links. Instead, we introduce a slight notational extension, analogous to sytem boundaries in use case diagrams.

Definition *component boundary* A graphical element enclosing exactly the transitive subcomponents of a component.

Notation A dotted boundary with rounded corners is drawn around the set of subcomponents. The object representing the component itself (the *primary object*) is positioned on the boundary, to show its role as the primary interface of the component. Fig. 2 augments the personal information manager example from Fig. 1 with component boundaries.

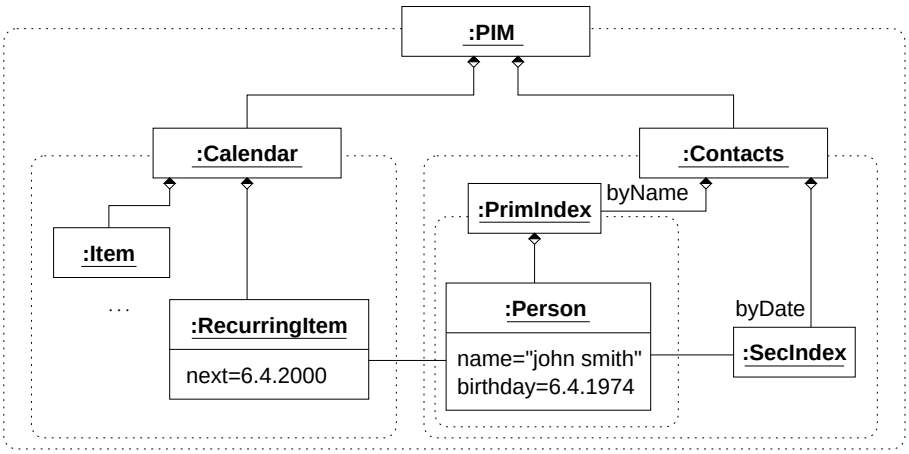

Fig. 2. A UML object diagram with component boundaries

Since the subcomponents of a component may in turn rely on internal subcomponents to implement their functionality, we naturally gain a hierarchical decomposition. This nesting of component boundaries can be used to depict logical as well as physical object spaces. It offers a unified notation for objects within arbitrary levels of components within nodes [Sto97], which in turn might be aggregated to local networks and to administrative domains [CG98].

The hierarchical structure can also be exploited to reduce the level of detail in an object diagram. By collapsing a component bubble and keeping the primary object, the overall structure is preserved, but irrelevant detail (internal to the component) is omitted. Fig. 3 gives a coarser view on our example, which was generated systematically from the detailed view.

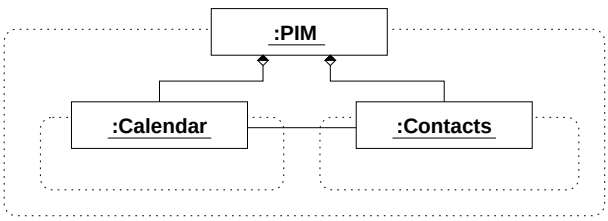

Fig. 3. A coarser view on the PIM.

Component links and component boundaries are essentially two views on the same concept, namely the grouping of objects to components. When a component link is given, the subcomponent will be inside the component's boundaries;

when component boundaries are given, an object's supercomponent is obvious as the immediately enclosing component. In contrast to component links, component boundaries offer no opportunity for attaching additional information like navigability or roles, and therefore model a less specific component concept.

3 Modeling Component Behaviour with UML

A component presents a defined behavioural interface to the outside. All interaction takes place in the form of incoming or outgoing messages crossing the component boundary. In this section, we show how the information about which objects belong to a component can be used to decide what is internal and what is external communication. Further, component membership of subcomponents may change over time. This is expressed by a notational extension in the component's interface.

UML offers two kinds of diagrams for interaction: collaboration and interaction diagrams. We will begin by investigating the influence of components on the former.

3.1 Components in Collaboration Diagrams

So far, we have described component boundaries in object diagrams. Collaboration diagrams are basically object diagrams overlaid with message flow information, so component boundaries can be drawn in a collaboration diagram as well.

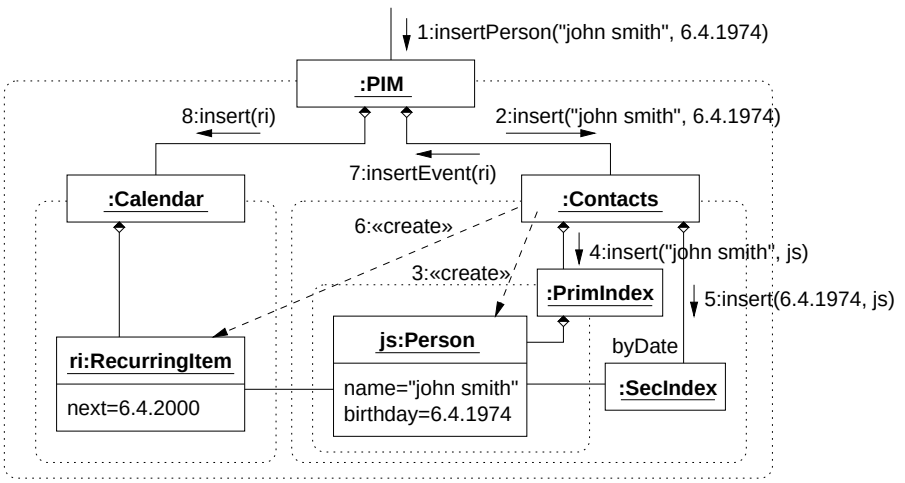

Fig. 4. A collaboration diagram including component boundaries.

When a new person is registered in our personal information manager example, some interactions take place in order to update the indices and to register the person's birthday as a recurring item in the calendar. Fig. 4 gives the whole detailed sequence, resulting in the state already shown in Fig. 2.

The coarsening that has taken place in the object diagram from Fig. 2 to Fig. 3 can be applied to this collaboration diagram as well. When the sender and the receiver of a message are members of the same component, and this component has been collapsed, the message will be abstraced from, and will not be shown. Fig. 5 gives the resulting diagram.

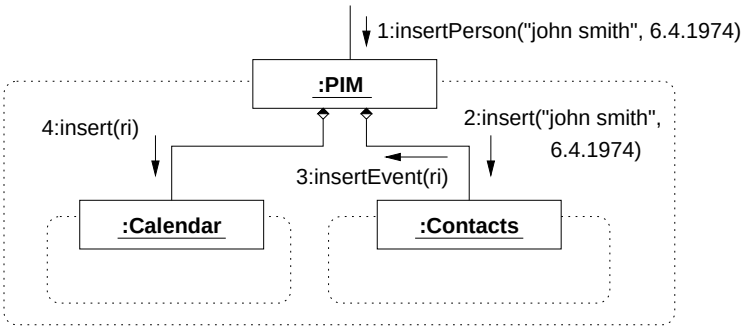

Fig. 5. A high-level collaboration diagram created from Fig 4.

3.2 Components in Sequence Diagrams

In this section, we shall briefly describe sequence diagrams with an emphasis on the concept of activation (which we will need in the next section), and will then consider the influence of components on sequence diagrams.

Sequence diagrams show the interaction of a number of objects, distributed horizontally, over time. Time flows from top to bottom. We will only consider the instance form here, which describes one actual sequence without branches, conditions or repetition. Fig. 6 gives an example.

At each point in time (represented by a horizontal cut), a number of living objects are represented by lifelines. Interactions, such as message send and return message, are depicted as horizontal arrows. Each kind of message can carry arguments, i.e. values or links. In the UML notation, arguments have to be represented in textual form. Each living object can have a number of activations, each representing an ongoing computation on the object. An activation starts when the object receives a message. The activation may then itself send a number of messages before it terminates by sending a return message to its caller.

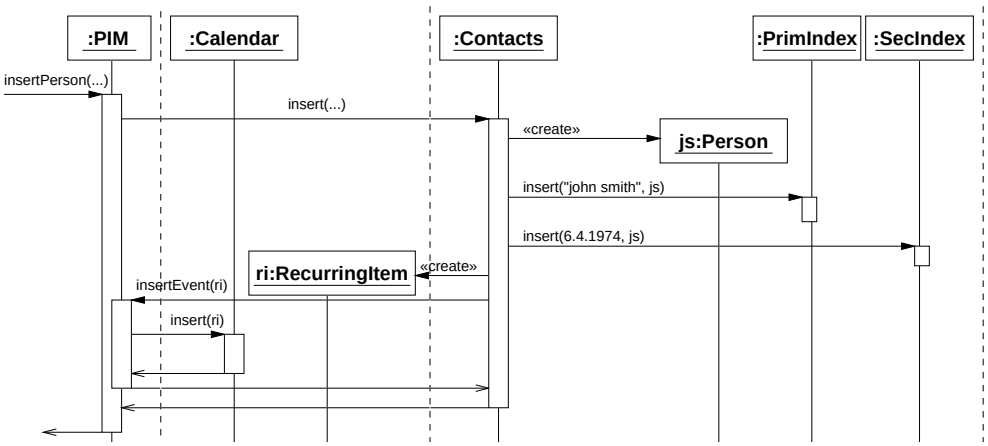

Fig. 6. The sequence diagram corresponding to Fig. 4.

Obviously, activations are part of the system state.[1] The system state at some point in time includes all activations, and for each activation, its respective progress in executing its method, and the local data it has gathered so far, including information taken from the invoking message.

In a collaboration diagram or in an object diagram, local links can be depicted using the ≪local≫ stereotype [RJB98], but the activation itself cannot be seen: the link originates at the active object to which the activation belongs. In a sequence diagram, activations are visible, but links are not.

We introduce a component boundary notation for sequence diagrams, too: A vertical dotted line, similar to the swimlanes used in UML activity diagrams, separates objects of different components. Messages crossing the boundary are easily recognized. However, since objects only have a one-dimensional (horizontal) position, drawing nested components is awkward.

A high-level sequence diagram is created by subsuming the lifelines of all objects inside a component under that of the primary object, as shown in Fig. 7. Internal messages are abstracted from, as well as internal activations. Note that the resulting interaction is the same as that depicted in Fig. 5.

3.3 The Lifecycle of a Component

In the example we have shown, a subcomponent (the recurring birthday item) is created in one component (the contacts database), and then migrated to another component, where it is made persistent by storing it in a database (the calendar). At a later point in time, e.g. when the person is removed from the contacts

[1] The designers of the programming language BETA even went so far as to unify the concepts of object and activation. We do not follow that trail here.

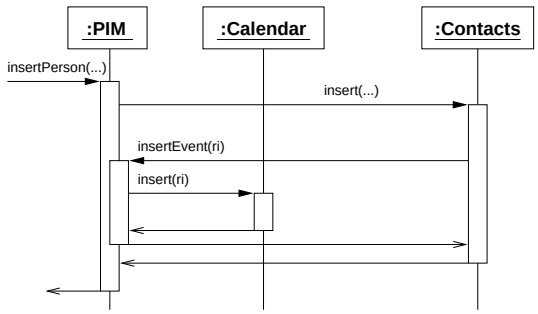

Fig. 7. A high-level sequence diagram created from figure 6.

database, the birthday item will be deleted from the calendar, and the lifetime of this subcomponent ends. This is what we call the lifecycle of a component.

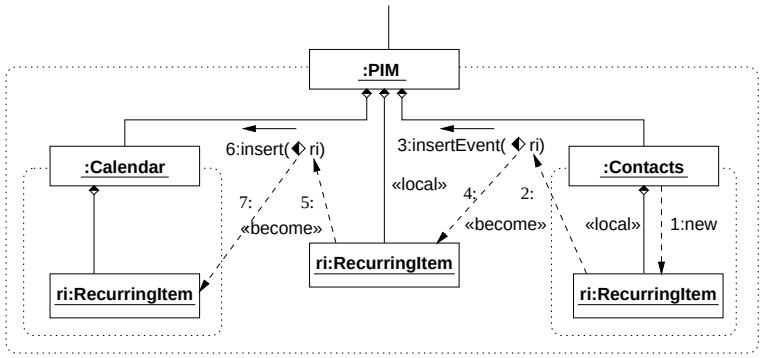

Fig. 8. Migration and persistence of a subcomponent.

Fig. 8 shows this process in greater detail. After creation of the birthday object, the creator (the contacts database) holds a local component link to the created object. The created object is then sent off in a message to the PIM component. As we have mentioned, links in UML messages have to be represented in textual form. In order to indicate that the message carries a component instead of an object reference, we have added a half-filled diamond before the object name. The symbol can be transcribed as the keyword `component`.

Upon receipt, the message starts a new activation in the PIM component (not visible in a collaboration diagram). The transmitted component link becomes a local component link of this activation. The PIM component passes the object on to the calendar component, where the object first becomes locally bound (not shown) and then becomes a persistent subcomponent.

Because component links determine the assignment of objects to components, the birthday object is actually a subcomponent of the message while in transit. This means that we can model the passing of objects by value as e.g. in RMI or CORBA. However, passing objects by value usually has copy semantics, which is not implied by passing a component link. By using the ≪become≫ stereotype [RJB98], we state that the object's identity is maintained across the migration. The ≪become≫ arrows indicate that the different birthday objects shown in the diagram are actually versions of the same object at different points in time.

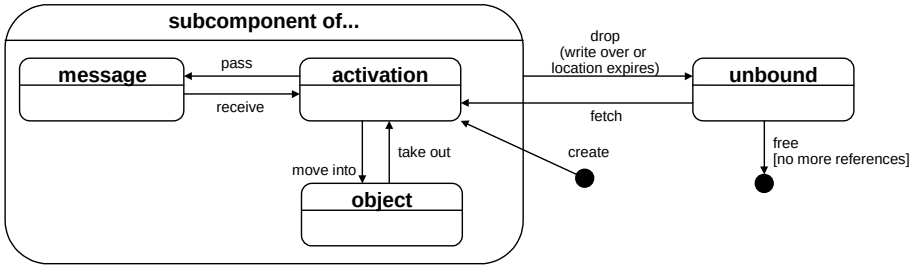

Fig. 9. Component lifecycle with respect to component aggregation.

The state diagram in Fig. 9 shows the lifecycle of an object with respect to component aggregation. In addition to the states already described – the component link may originate in an object, an activation or a message – the *unbound* state signifies that there are no component links to the object in question. This state can be reached when either the only component link to the object is explicitly removed (e.g. when it is changed to point to another object) or when the source of the link is destroyed. The latter takes place when an activation ends or when an object is destroyed. An object's subcomponents are not necessarily destroyed along with the component; the subcomponents may continue an individual existence even after the bubble has burst.

3.4 Components in Class Diagrams

A component as a group of collaborating objects is a concept at the object level, not at the class level. Therefore, dynamic components are not visible directly in a class diagram; especially, it makes no sense to draw component boundaries in a class diagram. For grouping classes, UML provides the concept of packages.

However, there are two impacts of components on class diagrams, as can be seen in Fig. 10. Firstly, an aggregation between two classes can use the ≪component≫ stereotype, turning all instances of this component aggregation into component links. As in the general case of aggregation, the component aggregation in the class diagram may include recursion in order to describe hierarchical structures. An example are the Java AWT user interface beans, where a

Container is a kind of Component that may include further instances of the class Component, so there is a cycle between Component and Container. Of course, every concrete user interface hierarchy only has finite depth, so there are no cycles in the object diagram.

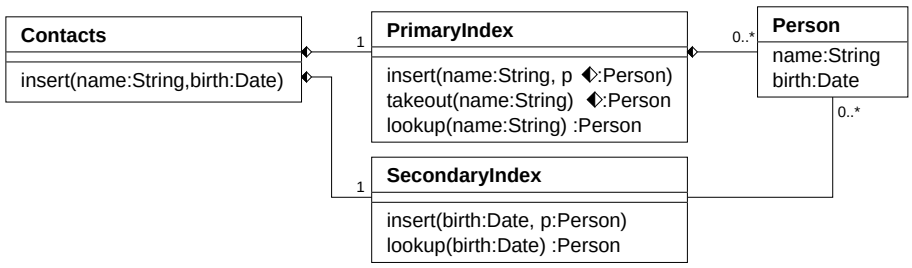

Fig. 10. A class diagram describing components.

Secondly, method parameters and results can be labeled with a half-filled diamond (transcribed as the keyword component) to indicate that component links are expected or returned. When a method takes a component parameter, the object supplied in the actual message is migrated to the message's receiver. Using this notation, the interface of a database can state whether the database will store the object itself or only a link to the object. In Fig. 10, the primary index aggregates the person objects, while the secondary index is only associated with them. So the insert operation of the primary index requires a component link to the person, while the secondary index takes a non-component link.[2]

As a second example, an object factory can state whether it passes the responsibility for its created objects to the client, or whether it manages the set of created objects itself and only passes out association links.

When component boundaries are interpreted as denoting physical location (similar to a deployment diagram), the interface specifies which arguments are to be migrated to the receiver's node, and the component links in those arguments specify which other objects are to be moved along with the primary objects. If a migrated component contains activations, this models a variant of the migrating threads described in [MMS96].

When the component boundary is interpreted as the border between volatile and persistent storage, the annotations state which objects are to be made persistent together. Again, the model covers persistence of active components, corresponding to persistent threads [MS94].

[2] The question of interface compatibility, e.g. whether there may be a common superclass for both primary and secondary index, and the question of parameterization, i.e. whether both classes may be instantiations of the same template, is discussed further in [Wie99] in the context of a strongly typed programming language.

4 Visualizing Component Behaviour at Runtime

As we have shown, the dynamic component structure can be employed to automatically coarsen a detailed interaction sequence to a high-level one, only showing the interaction between selected components and abstracting from communication internal to one component.

One application for this transformation is the presentation of interaction information observed in a running system in a re-engineering tool. This interaction data has a high volume and needs to be organized and filtered before it can be presented to the user in a meaningful form.

Existing approaches (e.g. [DKV94,SSC96,KM96]) distinguish between relevant and irrelevant interaction based on the static program structure, such as the class of sender and receiver. However, we believe that when observing a dynamic phenomenon like interaction, the dynamic system configuration has to be taken into account. For example, using static information only, it becomes impossible to observe the interaction between different complex instances of the same static component, unless the observer reverts to individual objects.

By exploiting the component structure, the visualizations produced by the re-engineering tool come closer to the abstract design-phase model, which facilitates understanding as well as validation [LN95]. The semantic gap between design and implementation is narrowed, and architectural properties can be observed in the running system.

For example, due to the hierarchic component structure, it becomes possible to distinguish between up-calls and down-calls. In the personal information manager example in Fig. 5, the PIM performs a down-call towards the contacts databases, which calls back up via an insert event in order install the birthday item in the calendar. The PIM then does a down-call to the calendar on behalf of the contacts database. Clearly, the PIM functions as a mediator.

In our approach, when examining a running program, the user has to specify the dynamic components whose boundaries [s]he wishes to monitor. This is achieved by *marking* their primary objects. Monitored components need not be disjoint, i.e. a marked object may be a transitive subcomponent of another

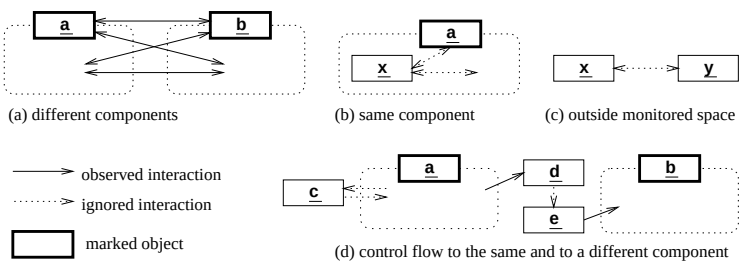

Fig. 11. Types of interaction relative to component boundaries.

marked object. The objects to be marked could be specified by an arbitrary predicate, e.g. "all instances of class A", or can be selected individually.

Our tracing tool examines all interaction between different objects. For the source as well as the destination of an interaction, the innermost enclosing monitored component is determined. If the objects belong to different monitored components, then we have an interaction crossing both component boundaries (Fig. 11a). This interaction will be presented to the user.

If source and destination belong to the same monitored component (Fig. 11b), or if both are outside of monitored space (Fig. 11c), the interaction is completely ignored.

If only the source object belongs to a monitored component, this means that the action leaves monitored object space. Control may flow through several unmonitored objects before it re-enters a monitored component. The leaving and entering messages are only reported if control flows between different monitored components, not if it re-enters the same component (Fig. 11d).

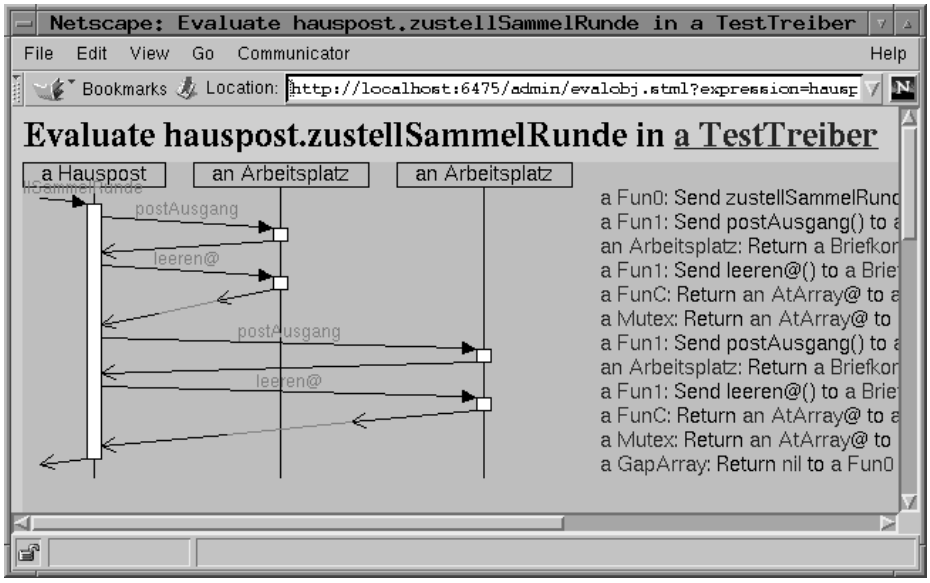

Fig. 12. An automatically generated high-level sequence diagram. (note to the reviewer: to be translated in final version)

We have implemented this filtering strategy in a research prototype described in [Wie99]. Fig. 12 shows an automatically generated sequence diagram, created from running an annotated program. Individual objects sending and receiving messages are displayed in the textual listing on the right hand side of the window; the sequence diagram itself only includes lifelines for the monitored components.

5 Identifying Component Links in an Implementation

In our research prototype, the observed program is prepared by annotating which variables (instance variables, local variables, parameters and method results) hold component-links and which variables hold non-component links. The component structure is deduced from this information.

If a modification of the examined program is not feasible, an existing distinction between different kinds of links can be mapped onto component and non-component links.

For example, distributed programming languages like Emerald [Jul89] and DOWL [Ach93] offer primitives for specifying a *migration group*, i.e. a set of objects to be migrated together. A variable attribute is used to determine which references are to be followed when computing the migration group. Such an object group can be interpreted as one component; the variable attribute then designates component links. In a similar manner, Java uses the `volatile` attribute to specify the boundaries of a group of objects to be made persistent together.

In Voyager, remote references can be interpreted as non-component links, and local Java references as component links. The disadvantage of this approach is that local references allow sharing between different components, precluding a clear assignment of objects to components for purposes of visualization, migration, persistence etc. This weakness is fixed in the Dejay system [BWL99] by grouping objects in *virtual processors*.

Many programming languages include the UML concept of composition (e.g. as member objects in C++, expanded objects in Eiffel [Mey97] or static links in BETA [MMPN93]). Due to the inflexibility of composition however, the object groups are usually quite small, and component membership cannot change over time.

If component links are not distinguished in the program source, an object can still be assigned to the component that created it. This creates a relation with the required formal properties. However, migration is not expressible.

As a last possibility, component membership can be assigned manually inside the re-engineering tool. When manipulating a sequence diagram, a command like "merge the lifelines of these objects" could be used. Such a command could easily be integrated in existing visualization tools, and would complement operations like navigation along the call graph [KM96]. The obvious disadvantage is that the grouping has to be specified each time the tool is used, instead of specifying it once and for all during design and implementation.

6 Related Work

Our concept of dynamic components is a generalization of the concepts of component instances and of nodes in UML. We have chosen the term *dynamic* component to stress their twofold dynamic nature: Firstly, they reside on the level of object diagrams as opposed to class diagrams; and secondly, they can migrate, so that the component structure itself is dynamic. Dynamic components also

differ from component instances in that their primary object is treated like any other object, i.e. it is treated as first-class. Primary objects could be differentiated from other objects using a stereotype, but only if required. The same is true for nodes, which become another stereotype of a dynamic component, with the additional semantics of a fixed, distinct location.

Civello [Civ93] talks about different kinds of aggregation. He lists several orthogonal properties used to specify an aggregation more exactly. In his terms, our concept of component aggregation is only restricted in that it excludes sharing; apart from that, any aggregation may be labeled as a component aggregation. Especially, Civello says that "it must be possible to model the migration of objects from one composite to another", which is possible in our model.

The importance of object configurations as opposed to static relations is stressed in [GL96], where the concept of environmental acquisition is presented. In the model by Gil and Lorenz, objects acquire properties from their ancestors in a hierarchical aggregation structure similar to the component structure described here. Gil and Lorenz also propose programming language mechanisms for distinguishing between component and non-component links (there called aggregation and nonaggregation links).

The hierarchical grouping of runtime objects to larger units has also been investigated in a number of papers dealing with aliassing in object-oriented programming languages, the latest of which is [NVP98]. The aim of that paper is to enhance encapsulation, which has not been considered in our work. However, the linguistic mechanisms developed in [NVP98] are also applicable for designating component links.

7 Summary

In this paper, we have introduced a small UML extension for denoting nested component boundaries in object and interaction diagrams, and have demonstrated its usefulness for systematically creating high-level diagrams of a component system. The component boundaries are defined based on component links, a form of aggregation between a component and its internal objects.

Secondly, we have shown how migration of components across component boundaries can be specified through an extension of the UML method signature notation.

Thirdly, a re-engineering tool has been presented that automatically creates meaningful sequence diagrams from a program, based on the dynamic component structure of the object graph.

References

[Ach93] Bruno Achauer. The DOWL distributed object-oriented language. *Communications of the ACM*, 36(9):48–55, 1993.

[BWL99] Marko Boger, Frank Wienberg, and W. Lamersdorf. Dejay: Unifying concurrency and distribution to achieve a distributed Java. In *Proceedings of TOOLS Europe '99*, Nancy, France, June 1999. Prentice Hall.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proceedings of FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, March 1998.

[Civ93] F. Civello. Roles for composite objects in object-oriented analysis and design. In *Proceedings of OOPSLA '93*, pages 376–393, San Jose, California, October 1993.

[DKV94] Wim DePauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In *Proceedings of ECOOP '94*, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[GL96] Joseph Gil and David H. Lorenz. Environmental acquisition – a new inheritance-like abstraction mechanism. In OOPSLA [OOP96], pages 214–231.

[Gri98] Frank Griffel. *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt-Verlag, Heidelberg, 1998.

[Jul89] E. Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, Department of Computer Science, University of Washington, 1989.

[KM96] Kai Koskimies and Hanspeter Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *International Conference on Software Engineering (ICSE '96), Berlin*, 1996.

[LN95] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA '95*, pages 342–357, Austin, Texas, USA, October 1995. ACM.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction, 2nd edition*. Prentice Hall, 1997.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[MMS96] B. Mathiske, F. Matthes, and J.W. Schmidt. On migrating threads. *Journal of Intelligent Information Systems*, 8(2), 1996.

[MS94] F. Matthes and J.W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of ECOOP '98*, number 1445 in Lecture Notes in Computer Science, pages 158–185, Brussels, Belgium, July 1998. Springer-Verlag.

[OOP96] ACM. *Proceedings of OOPSLA '96*, San Jose, California, October 1996.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley object technology series. Addison Wesley Longman, December 1998.

[SSC96] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Architecture-oriented visualization. In OOPSLA [OOP96], pages 389–405.

[Sto97] David Petrie Stoutamire. *Portable, Modular Expression of Locality*. PhD thesis, University of California at Berkeley, December 1997.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Wie99] Axel Wienberg. Dynamic components in an object-oriented programming language - model, language implementation and visualization. Diploma thesis, computer science department, University of Hamburg, Germany, March 1999. *in German*.