Software—Concepts and Tools (1995) 16: 94–103

Software—Concepts and Tools
© Springer-Verlag 1995

Active Guidance of Framework Development

Wolfgang Pree[1], Gustav Pomberger[1], Albert Schappert[2], Peter Sommerlad[2]

1) C. Doppler Laboratory for Software Engineering, Johannes Kepler University Linz, A-4040 Linz, Austria
e-mail: {pree, pomberger}@swe.uni-linz.ac.at

2) Siemens AG, ZFE T SE 42, Otto-Hahn-Ring 6, D-81739 Munich, FRG
e-mail: {albert.schappert, peter.sommerlad}@zfe.siemens.de

Abstract. An appropriate combination of object-oriented programming concepts allows the development not only of single reusable components but also of semifinished architectures (= frameworks).

The paper discusses an adequate way to represent the know-how of software engineers in the realm of developing and adapting frameworks. Active cookbooks rely on a particular knowledge base. These electronic books form an integration basis of various tools adjusted to a domain-specific framework. Active cookbooks guide programmers and end users through typical framework adaptation steps. Examples taken from a prototype implementation of such a cookbook illustrate the concepts that allow active support of framework-centered software development.

Keywords: knowledge-based software engineering, automated software development, application frameworks, rule-based systems, software architecture, visual object-oriented programming, design patterns, reuse

1. Motivation

Despite the obvious advantages of software reuse, almost all software systems continue to be developed from scratch. Even when object-oriented programming languages are used, their real reuse potential is seldom tapped. The concepts of inheritance and dynamic binding are sufficient to construct frameworks, that is, reusable semifinished architectures for various application domains. Such frameworks mean a real breakthrough in software reusability: not only single building blocks but whole software (sub)systems including their design can be reused. So frameworks enable a degree of software reusability that can significantly improve software quality.

Unfortunately, the complexity of framework development and adaptation constitutes a major hurdle for a widespread use of this advanced object-oriented technology. In order to cope with a framework's complexity, its design has to be captured and communicated, and guidelines for its adaptation have to be provided. The project carried out at Siemens started with these goals in mind. This paper presents the concepts and ideas behind frameworks and the concepts of the rule-based system underlying active cookbooks. A discussion of an active cookbook prototype illustrates how an adequate tool can significantly automate the framework-centered software development process.

2. Framework Concepts

Frameworks rely on *abstract classes*, whose general idea is clear and straightforward:

- Properties (that is, instance variables and methods) of similar classes are defined in a common superclass.
- Classes that define common behavior usually do not represent instantiable classes but abstractions of them. Thus they are called abstract classes.
- Some methods of the resulting abstract class might be implemented, while only dummy or preliminary implementations are be provided for others. Though some methods cannot be implemented, their names and parameters are specified since descendants cannot change the method interface. So an abstract class creates a *standard class interface* for all descendants. Instances of all descendants of an abstract class will understand at least all messages that are defined in the abstract class.

  Sometimes the term *contract* is used for this standardization property: instances of descendants of a class A support the same contract as supported by instances of A.
- The implication of abstract classes is that other software components based on them can be

implemented. These components rely on the contract supported by the abstract classes. In the implementation of these components, reference variables are used that have the static type of the abstract classes they rely on. Nevertheless, such components work with instances of descendants of the abstract classes by means of polymorphism. Due to dynamic binding, such instances can bring in their own specific behavior.

The key problem is to find useful domain abstractions so that software components can be implemented without knowing the specific details of concrete objects.

2.1 Frameworks as Sets of Abstract and Concrete Classes

Wirfs-Brock and Johnson [18] describe the relationship between abstract classes and frameworks in a general way: "Although abstract classes provide a way to express the design of a class, classes are too fine-grained. A framework is a collection of abstract and concrete classes and the interface between them, and is the design for a subsystem."

The term *application framework* is used if this set of abstract and concrete classes comprises a generic software system for an application domain. Applications based on such an application framework are built by customizing its abstract and concrete classes. It is often hard to decide whether a framework migrates to this category. So we use the terms framework and application framework as synonymous terms.

GUI application frameworks such as MacApp [17] and ET++ [14, 15, 2, 13] provide a reusable, blank *application* that implements much of a given user interface look-and-feel standard. GUI application frameworks can be viewed as the first test bed for the development of reusable architectures by means of object-oriented programming concepts. They have become one of the main reasons why object-oriented programming enjoys such a good reputation for promoting extensibility and reuse.

ET++ provides an impressive example of the degree of reusability that can be achieved in well-designed frameworks. (Well-designed means that a framework offers the required domain-specific flexibility for adaptations.) Besides allowing reuse of the architecture of the framework, Weinand et al. [15] state that writing an application with a complex GUI by adapting ET++ can result in a significant reduction in source code size (that is, the source code that has to be written by the programmer who adapts the framework) compared to software written with the support of a conventional graphic toolbox.

Of course, application frameworks are not limited to the construction of direct-manipulation, graphic user interfaces. Domain-specific application frameworks represent a newly emerging field in the realm of object-oriented technology that allows exploitation of the reuse potential of object-orientation. Examples are Taligent's frameworks [8], frameworks for visual language systems [1] and ProcessTalk [9], a framework for distributed process control systems.

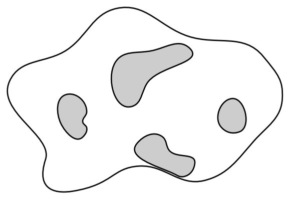


Figure 1. Framework as frozen cloud with flexible hot spots.

2.2 Framework-Centered Software Development

In general, a given framework already anticipates much of a software system's design. This design is reused by all software systems built with the framework. So not only source code but also architecture design—which we consider as the most important characteristic of frameworks—is reused in applications built on top of a framework. So frameworks are well suited for domains where numerous similar applications are built from scratch again and again.

A framework defines a *high-level language* with which applications within a domain are created through *spezialization* (= adaptation). Specialization takes place at points of predefined refinement that we call *hot spots*. Figure 1 illustrates this property of frameworks with the flexible hot spots in gray color. The overall framework is represented as a white cloud representing the standardized, i.e., frozen, domain aspects.

The specialization of hot spots requires the definition of additional classes in order to override methods and/or object configurations based on components already provided by the framework.

Experience has proven that both the development of a framework and its specialization are difficult. The pain of designing a framework from scratch is described by Wirfs-Brock and Johnson [18]: "Good frameworks are usually the result of many design iterations and a lot of hard work." Design patterns as discussed by Gamma [3, 4] and Pree [10, 11] help to develop new frameworks by applying design approaches that have already matured in other frameworks. Thus design patterns can help to reduce the number of iterations.

We consider a framework to be well-designed if it offers the required hot spots. As a framework evolves, software engineers will encounter missing or inadequate hots spots. It is often impossible to fix such design flaws without having to change a framework's source

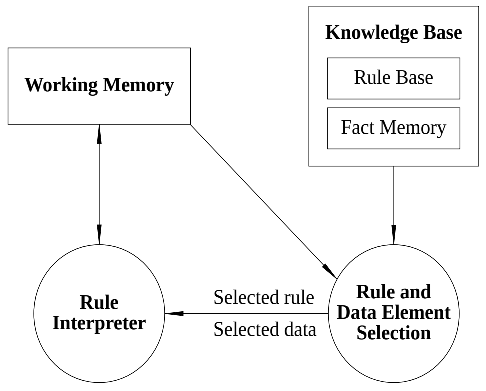


Figure 2. Principal components of a rule-based system (adapted from Hayes-Roth [6]).

code. This problem is addressed by what we call *structural relations* (see Sections 3 and 4).

The first available frameworks, such as Smalltalk's Model-View-Controller (MVC) framework [7] and MacApp, revealed that a framework's complexity is a burden for its user. (By the term user we mean the programmer who uses a framework to produce a specific application.) In order to adapt it, a framework user must become familiar with its design, that is, the design of the individual classes and the interaction between these classes, and maybe with basic object-oriented programming concepts and a specific programming language as well.

This is why *framework cookbooks* have come to light. Their recipes describe in an informal way how to use a framework to solve specific problems. For example, in a framework for reservation systems a recipe could describe how to adapt the rate calculation for rental items.

A programmer has to find the recipe that is appropriate for a specific framework adaptation. A recipe is then used by simply adhering to the steps that describe how to accomplish a certain adaptation task.

A cookbook recipe is typically structured into the sections purpose, procedure (including references to other recipes), and source code example(s). Cookbook recipes with their inherent references to other recipes lend themselves to presentation as hypertext. Recipes usually do not explain the internal design and implementation details of a framework.

What we call *informal relations* (see Sections 3 and 4) evolved from recipes as means to actively support framework adaptation at hot spots.

3. Rule-Based Active Cookbooks

Hayes-Roth [6], for example, describes the principal components of a rule-based system. Figure 2 illustrates these components and their relationships.

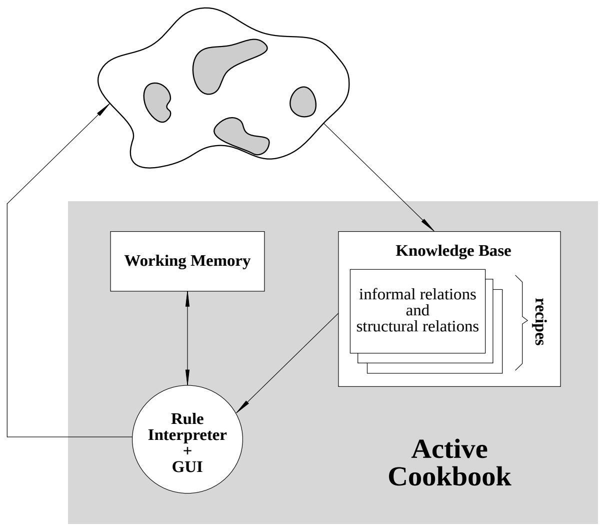


Figure 3. Rule-based active cookbook.

Garlan and Shaw [5] view a rule-based system as a special case of an architectural style called table-driven interpreter: The knowledge base represents the pseudo-code to be interpreted. The rule interpreter forms the core of the inference machine and can be considered as the interpretation engine. The control state of the interpreter is represented by the rule and data element selector. The working memory stores the current state of the interpreted program.

3.1 Knowledge Representation

Conventional ways to represent a software engineer's know-how, such as descriptive logic as used in the software component construction environment CD-CON [12] and algebraic specifications, proved to be inappropriate means to support framework-centered software development. Thus suitable knowledge representation formalisms had to be explored to tame the complexity of state-of-the-art frameworks.

Recipes constitute a viable vehicle for this purpose: Framework components require only certain localizable steps in order to be completed. Recipes describe these steps, ignoring the numerous irrelevant details of involved components. Recipes rely on either informal or structural relations as outlined below. Figure 3 shows the rule-based system underlying active cookbooks.

3.2 Informal Relations

As mentioned in Section 2, frameworks introduce a sort of language on an abstraction level significantly higher than the particular underlying object-oriented programming language. Basic elements of such a framework language are the class and method names. Typically a group of classes and methods works together in order to provide a certain service. “Working together” usually means a complex interaction—i.e., method calls—between components. The various building blocks and interactions constitute the application framework language.

Ideally programmers adapting application frameworks to specific needs do not have to know all the details of the application framework language. They only have to know and understand some aspects in order to customize hot spots. Thus they think in terms of specific rules for accomplishing a certain adaptation.

Informal relations serve the purpose of capturing these rules. They describe in an informal way the coarse interaction between components and the purpose of particular methods. They lead the developer step by step through the adaptation process. For this purpose recipes based on informal relations provide the appropriate tools to specify adaptations. Some adaptations might lend themselves for visual manipulations. Tree editors, data flow editors and resource editors are examples of visual manipulation tools incorporated in a recipe.

When such recipes are interpreted, the developer is guided actively through all development and configuration steps of a certain task. Typically, the developer just invokes the tools associated with each step of a recipe. The recipe also checks dependencies between steps: sometimes certain steps have to be accomplished before tools associated with other steps can be invoked.

3.3 Structural Relations

Framework designers anticipate its specialization by defining hot spots. But a framework can hardly meet entirely new requirements unless the design and implementation of the original framework is changed. However, when frameworks are used in practical applications, their design and implementation cannot be changed continously. On the other hand, framework users have to meet requirements that were not anticipated by the framework designers. We propose structural relations to attack this problem.

Conceptually, structural relations capture the interaction between components (see Figure 4). Usually interaction code is distributed over several framework components. Structural relations bundle this interaction. The arrows in Figure 4 express schematically the interaction between components, i.e., method calls between components. The gray components plugged into the structural relation are placeholders. If existing framework components or newly developed ones are plugged into a structural relation, they have to offer a certain method protocol depending on the particular interaction defined in the structural relation.

Structural relations offer the following advantages:

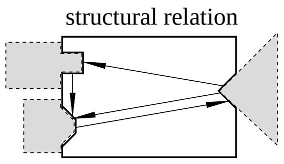


Figure 4. Capturing interaction between components in structural relations.

- An already defined interaction relationship between components can be applied to new components not yet provided by the framework. Thus interaction design is reused. Furthermore, a structural relation specifies the interface that the attached components must offer. The corresponding methods and probably parts of their implementation can be generated based on the information provided by a structural relation.
- More complex structural relations can be constructed out of basic ones. This represents another way of interaction design reuse.

Note that structural relations lend themselves to be embedded in informal relations. In this case an informal relation describes the steps to reuse a structural relation, i.e., how to attach newly defined components to a structural relation. The prototype presented in Section 4 illustrates this combination of informal and structural relations in recipes: The active cookbook user interacts with a hypertext recipe based on an informal relation. This recipe is in turn based on (a) structural relation(s).

Structural relations have further advantages:

- Component interaction need not to be mirrored in the class hierarchy of an object-oriented system. Explicit modeling of the interaction keeps the structure of the class hierarchy simple and easier to understand.
- Appropriate modeling of interaction code postpones necessary redesign steps and allows the evaluation of redesign intentions.

3.4 Rule Interpreter

Note that the rule and data element selection component of a rule-based system is conceptually unified with the rule interpreter component in an active cookbook (see Figure 2 and 3). The rule interpreter of an active cookbook:

- allows selection of a particular recipe
- presents the recipes as hypertext
- maintains temporary information accumulated during the interpretation of a recipe in working memory
- generates the source code of additional or modified framework classes

The arrow drawn from the rule interpreter to the framework in Figure 3 expresses the rule interpreter's involvement in the framework development process: the generated classes modify/extend the particular framework.

The knowledge base is typically influenced by the evolving framework. Changes in relationships between framework components might imply additional structural or informal relations that modify recipes or add new ones to the knowledge base.

In the following section the realization of these concepts is illustrated by an active cookbook prototype for a specific domain.

4. An Active Cookbook Prototype

This section presents an active cookbook prototype that demonstrates how the framework development process can be actively supported. First we outline the framework that serves as our testbed for the active cookbook prototype. As communication systems form an important strategic product category for Siemens, we chose the ubiquitous computing domain ([16]; see below), for which we developed a framework. The informal and structural relations embedded in the active cookbook prototype for that framework illustrate the realization of these concepts and their automation potential. Overall, the rule-based active cookbook whose architecture is depicted in Figure 3 offers the following features:

- Active guidance: Recipes invoke the appropriate tools that are suited for a particular adaptation.
- Context-sensitive behavior: Recipes are stored embedded in class code and are scanned during their invocation; hence specifics of certain classes influence the recipes; recipes might also change themselves while they are being processed by the interpreter component of the active cookbook.
- Incremental extensibility: All classes as well as informal and structural relations are stored in a common format so that third vendor components can be integrated.

Note that an active cookbook for a particular framework should especially help an experienced programmer develop software based on this reusable architecture. Providing as an aid for end user computing is a secondary goal of this electronic book.

4.1 A Communication System Framework as Testbed

Mark Weiser's vision of a ubiqituous computing world [16] was chosen as an ideally suited application domain in order to test the practicability of our concepts. One of the key ideas underlying a ubiqituous computing world is to offer everyone a personal working environment whatever computer is used and wherever a person resides. So, for example, personally configured desktops can be displayed no matter whether its owner is sitting in front of a workstation or using a white board on a wall. Another characteristic is that one can specify which messages (phone calls, e-mails, etc.) are to be forwarded to where the addresse currently is.

The backbone of a ubiqituous computing world is communication software. Its development forms the central issue of the testbed framework *UbiComm*, which stands for ubiqituous communication.

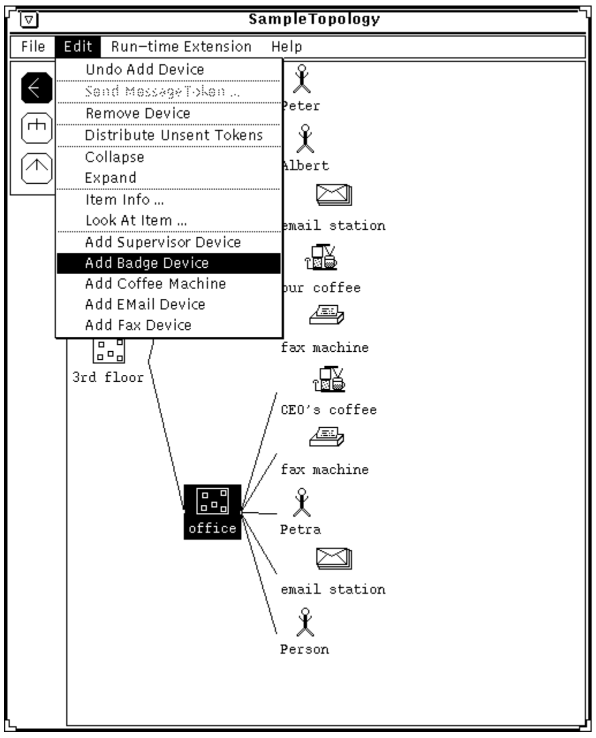


Figure 5. Screenshot of an ubiqituous computing world simulation.

Functionality of UbiComm. The UbiComm framework implements the communication control infrastructure required in an ubiqituous computing world together with a GUI for simulating this vision (see Figure 5).

Without the modification or extension of UbiComm, the framework offers the following features:

- *Interactive configuration of the topology*. The simulated ubiqituous computing world as shown in Figure 5, for example, is interactively configured by simply moving devices between different rooms with the mouse. Note that the inner nodes group devices and thus represent rooms, floors or buildings. People wear badge devices that store personal configurations and localize a person's physical position. Badge devices are depicted as persons in the simulation.

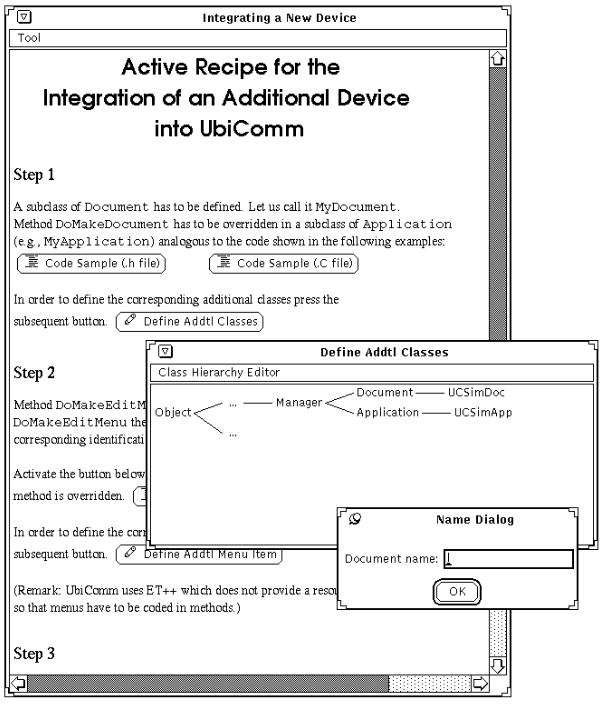
In order to insert a device into a simulated ubiqituous computing world, the user selects the device group where the additional device should be inserted and chooses the corresponding menu item in the Edit menu (see Figure 5). Without extensions UbiComm offers badge devices, e-mail devices and devices to group other devices.

- *Distribution of message tokens*. Depending on the type of a device, different message tokens can be handled. For example, an e-mail device can handle e-mail tokens. In order to simulate the sending of e-mails, the user selects a badge device and chooses the Send Msg Token menu item from the Edit menu. Depending on the devices located in a room, a particular message can be composed and sent. (Without extensions, UbiComm offers only email devices which generate email messages.) The default behavior of UbiComm is to forward a message to the room where the receiver is located. If no appropriate device could process the message token there, the message is queued and forwarding is retried later.

Typical UbiComm adaptations. The key abstractions in UbiComm are the abstract classes Device, MsgToken and MsgDispatcher. Adaptations of UbiComm are accomplished by defining subclasses and overriding the corresponding methods of these abstract classes. Adaptations of UbiComm typically extend the framework by defining additional devices and probably message tokens. A change of the message forwarding strategy constitutes another UbiComm specialization. The active cookbook provided for UbiComm significantly automates these framework specializations as shown below.

Project status. UbiComm and the corresponding active cookbook prototype were developed in order to demonstrate the feasability of our concepts and their pros and cons in the realm of a specific domain. It was definitely not the goal to develop a reusable prototype. As the results were promising, the prototype now serves as a reference system for the implementation of tools for numerous domains.

Both UbiComm and the active cookbook prototype were implemented in C++. ET++ formed the basis for providing the GUI. UbiComm comprises around 30 classes, the active cookbook around 50 classes. The prototype was developed in 35 person months. It is available on Unix workstations and PCs.



Active Recipe for the Integration of an Additional Device into UbiComm

Step 1

A subclass of `Document` has to be defined. Let us call it `MyDocument`. Method `DoMakeDocument` has to be overridden in a subclass of `Application` (e.g., `MyApplication`) analogous to the code shown in the following examples:

In order to define the corresponding additional classes press the subsequent button.

Step 2

Method `DoMakeEditM`
`DoMakeEditMenu` the
corresponding identificati

Activate the button below
method is overridden.

In order to define the corr
subsequent button.

(Remark: UbiComm uses ET++ which does not provide a reso
so that menus have to be coded in methods.)

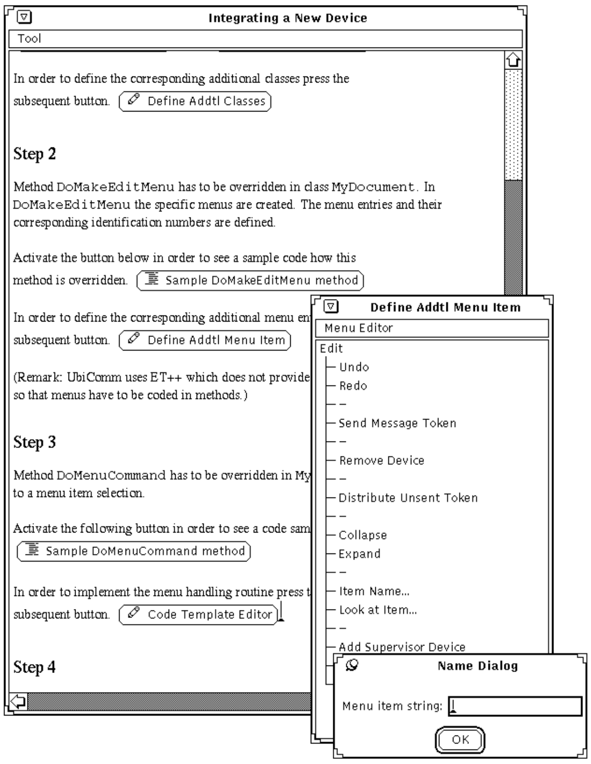
Step 3

Figure 6. Activation of a tree editor in an active recipe.

4.2 A Recipe for Integrating an Additional Device

This UbiComm adaptation is a typical one that is already anticipated in the framework design. Thus an informal relation underlies the recipe that actively guides through the process of integrating a new device. We assume that an additional fax component was already defined by means of the corresponding recipe presented in Section 4.4. This new fax component has to be integrated into UbiComm so that devices of this type can be used in a ubiqituous computing world simulation. The screenshot in Figure 5 is already based on an extended UbiComm framework that offers a fax device in the Edit menu.

Device Integration—Step 1. Figure 6 shows how the rule interpreter component of the active cookbook presents the recipe used to integrate a fax device into UbiComm. The first step requires the definition of two subclasses of the UbiComm classes Application and Document and the overriding of method DoMakeDocuments in the Application subclass. Instead of having to define the corresponding C++ header and implementation files, the user (= programmer who adapts the UbiComm framework) simply provides the class names in the tree editor shown in Figure 6. The tree editor opens after a click on the Define Addtl Classes button in the recipe.



Integrating a New Device

Tool

In order to define the corresponding additional classes press the subsequent button. Define Addtl Classes

Step 2

Method DoMakeEditMenu has to be overridden in class MyDocument. In DoMakeEditMenu the specific menus are created. The menu entries and their corresponding identification numbers are defined.

Activate the button below in order to see a sample code how this method is overridden. Sample DoMakeEditMenu method

In order to define the corresponding additional menu en
subsequent button. Define Addtl Menu Item

(Remark: UbiComm uses ET++ which does not provide
so that menus have to be coded in methods.)

Step 3

Method DoMenuCommand has to be overridden in My
to a menu item selection.

Activate the following button in order to see a code sam
Sample DoMenuCommand method

In order to implement the menu handling routine press t
subsequent button. Code Template Editor

Step 4

Define Addtl Menu Item

Menu Editor

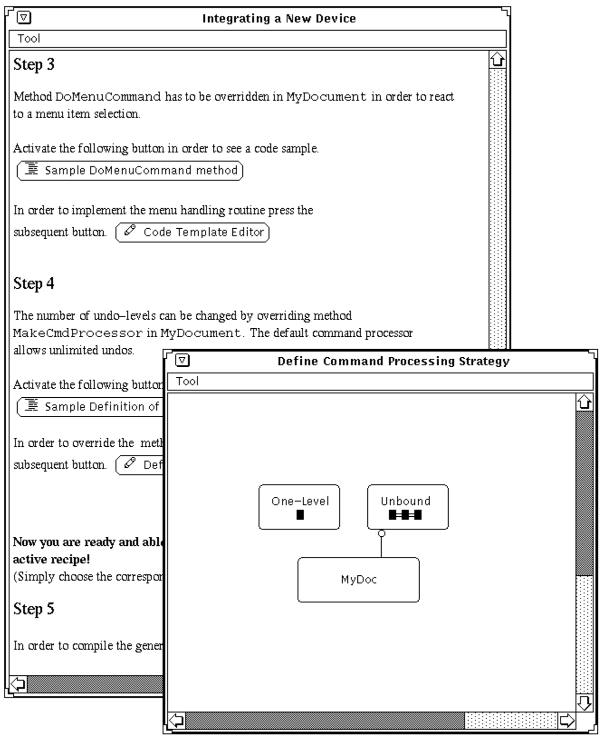
```
Edit
├ Undo
├ Redo
├ –
├ Send Message Token
├ –
├ Remove Device
├ –
├ Distribute Unsent Token
├ –
├ Collapse
├ Expand
├ –
├ Item Name...
├ Look at Item...
├ –
├ Add Supervisor Device
```

Name Dialog

Menu item string:

OK

Figure 7. Activation of a resource editor in an active recipe.



Integrating a New Device

Tool

Step 3

Method DoMenuCommand has to be overridden in MyDocument in order to react to a menu item selection.

Activate the following button in order to see a code sample.
Sample DoMenuCommand method

In order to implement the menu handling routine press the subsequent button. Code Template Editor

Step 4

The number of undo–levels can be changed by overriding method MakeCmdProcessor in MyDocument. The default command processor allows unlimited undos.

Activate the following button
Sample Definition of

In order to override the meth
subsequent button. Def

Now you are ready and abl
active recipe!
(Simply choose the correspo

Step 5

In order to compile the gener

Define Command Processing Strategy

Tool

Figure 8. Activation of a configuration editor in an active recipe.

Device Integration—Steps 2 and 3. The Edit menu (see Figure 5) has to be extended in order to provide an additional menu item that corresponds to the additional device type. For example, in case of the fax device, we define the menu item Add Fax Device. For accomplishing this task the active recipe provides a resource editor which opens after pressing the Define Addtl Menu Item button (see Figure 7).

In Step 3 the user has to define what happens if the newly defined menu item is selected. Instead of having to know which specific method to override, the user only has to provide the program code fragment that generates an instance of the particular device subclass. In case of integrating a fax device, this would be the statement `return new FaxDevice(...)`.

The active guidance often depends on the context. For example, if the tool activated in step 2 or 3 recognizes that a pen device should be integrated, the recipe itself would change and offer an additional step 3a where tablet-specific parameters can be specified.

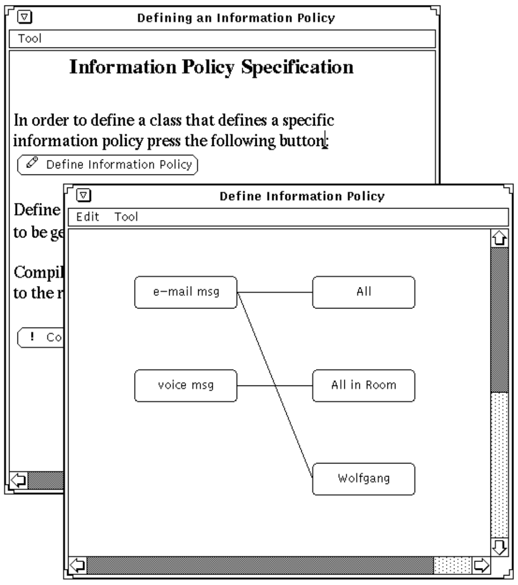
Device Integration—Step 4. Finally, the undo/redo mechanism of a ubiqituous computing world simulation can be adjusted in step 4. Depending on the user's preference, one-level or unlimited-level undo is specified by connecting the corresponding components as shown in Figure 8. Without tool support the user would have to override a specific method where the particular command processor is instantiated.

Code Generation. On completion of these adaptation steps, the corresponding C++ classes are generated, compiled and linked with the original UbiComm framework. This is done automatically by the active cookbook based on the information accumulated in the working memory while processing a recipe. The adapted UbiComm simulation now offers fax machines as an additional device type (see Figure 5).

4.3 Recipe for Adapting the Message Forwarding Policy

The corresponding active recipe offers a data flow editor for specifying the individual message forwarding policy. For example, in Figure 9 the forwarding policy is specified for a particular person as follows: if an e-mail arrives for this person, it is forwarded to all other members of the group this person belongs to and to a person named Wolfgang. A voice message is forwarded to all people in the room where the addressee currently is.

After specification or redefinition of the forwarding policy for a person, the corresponding class can be generated simply by activating the Generate Class button in the recipe. UbiComm is designed so that an


Defining an Information Policy
Tool
Define Information Policy
Define Information Policy
Edit Tool
e-mail msg
voice msg
All
All in Room
Wolfgang

Information Policy Specification

In order to define a class that defines a specific information policy press the following button:

Define
to be g

Compil
to the r

Figure 9. Activation of a data flow editor in an active recipe.

instance of a newly generated information policy class can be integrated dynamically into a running ubiqituous computing world simulation. So individual forwarding policies can be changed at run time.

4.4 Recipe for Creating a New Device

The creation of a device with specific functionality constitutes an example of a framework modification which was not anticipated. Thus structural relations underly the recipe that actively guides through this development process. The active guidance is again based on an informal relation.

Remember that a structural relation models the interaction of (several) framework components. According to the plugs offered by a structural relation, components have to be associated with it. A structural relation contains the code that implements a particular interaction. When a component is associated with a structural relation as will be demonstrated below, relation-specific interaction code is conceptually added to that component.

It depends on the implementation underlying a structural relation whether the code is added via inheritance mechanisms or by generating methods and code for associated components. A programmer working with structural relations does not see these implementation details.

Note that the use of structural relations does not affect an application framework's structure, i.e., its class hierarchy and interaction already defined in its components. Thus structural relations extend existing

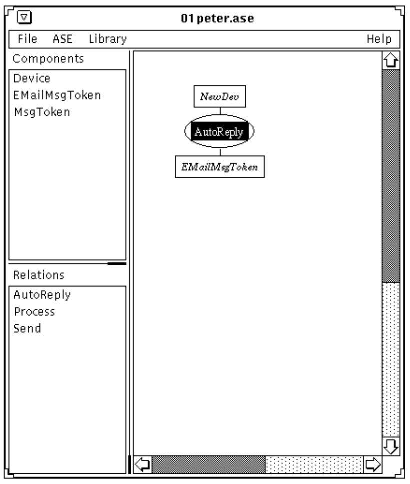


Figure 10. The *AutoReply* Relation.

application frameworks and enhance their flexibility without having to change their design and implementation. They can also be used later during framework redesign to incorporate the interaction originally added by structural relations directly in the framework.

Take a coffee machine as an example of a device with new functionality. Persons living in an ubiqituous computing environment should be able to ask a coffee machine device via email regarding its fill state. When a coffee machine receives an email it automatically returns its fill state wrapping the answer in an email and returning it to the sender.

The recipe presented to the active cookbook user is analogous to the one shown in Figure 6. The first step in the realm of constructing a new device is the specification of its iconic representation by means of a bitmap editor. When code is generated, the recipe takes care to attach the bitmap to the class according to the conventions predefined in the UbiComm framework.

The next steps in the recipe rely on structural relations. The recipe offers an appropriate construction tool (see Figure 10) in order to work with them. The left-hand-side panes of the construction editor list the structural relations defined for UbiComm (lower pane) and those UbiComm components (upper pane) that are involved in these structural relations. The AutoReply relation (shown in the right-hand-side pane) cooperates with two components, a device and a message token. The structural relation itself is represented by an oval; the plugs (which are placeholders for components) by two rectangles.

The AutoReply relation models the following functionality: It receives a message, determines its type and sender, generates a response message, and replies to the sender. This does not specify the response message

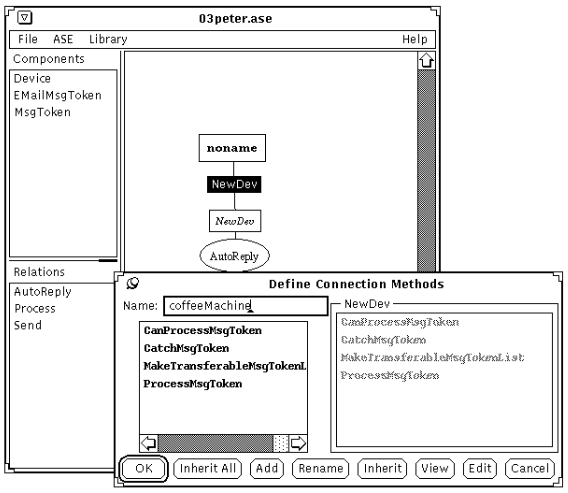


Figure 11. Specification of methods.

or when it will be sent. This has to be adjusted when AutoReply is reused. However, the code to realize the functionality described above is contained in the form of methods in the relation itself and will be inserted into the participating components, i.e., the new coffee machine device. Thus the new class can be constructed easily. It significantly extends the UbiComm Device class.

The active cookbook user now drags the Device component from the Components panel into the construction panel and connects the component labeled noname (see Figure 11) with the plug of AutoReply. This means that the new component is associated with the AutoReply relation. The same has to be done with an EMailMsgToken component and the corresponding plug.

The AutoReply relation defines the class structure of the associated components, i.e., method names, and most of their implementation. What remains to be done is to specify what is not modeled in the relation or what should differ from the interaction defined in AutoReply. As mentioned above, we just have to specify the reply and time for when a coffee machine receives an email.

Relations can be constructed from other relations or components. At the lowest granularity, however, basic or atomic relations are ultimately implemented directly in the framework. They can be clustered and grouped in hierarchies. This is illustrated in Figure 12 for the AutoReply relation, which is built from a Send and a Process relation. Relations consist of a (still) schematic body and contain ports for components to interact with. In the current implementation, ports require type conformance.

To sum up, the concept of relations offers sufficient help and tutoring, but it still leaves enough freedom. By separating the interaction code between classes in

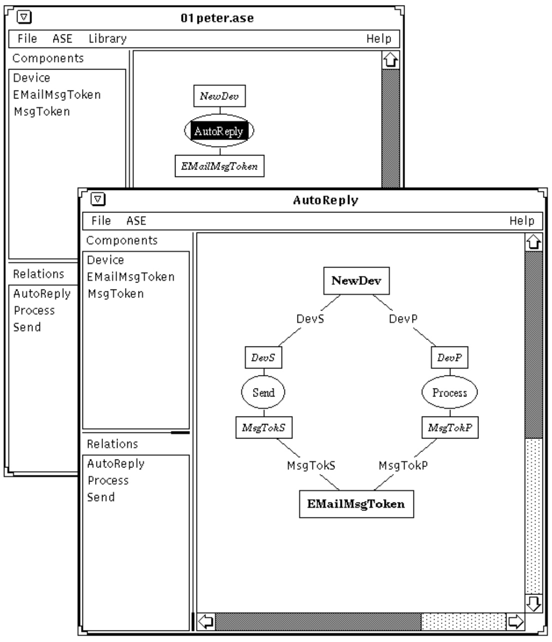


Figure 12. The Internal Structure of the *AutoReply* Relation.

relations, it becomes possible to (re)use it in the modeling and design process. This permits the reuse of interactions without changing the class hierarchy of a complex framework.

The new framework component coffee machine can be integrated into the running application by following the steps of the recipe explained in Section 4.2. Figure 5 shows a ubiqituous computing world simulation with coffee machines.

5. Summary and Outlook

An active cookbook helps to automate software development based on application frameworks. The presented prototype illustrates that such a tool is well suited to efficiently specifying certain aspects of framework adaptations. Some of the visual manipulation editors offered by the active cookbook prototype should be usable by non-programmers. To fill the gap in between, programming expertise is required. This is corroborated by the sample adaptations of UbiComm.

Future research and hands-on experience with the active cookbook prototype will reveal pros and cons of the presented approach. Research is especially necessary to define generic editors that can be integrated into active cookbooks for several different domains. It would be a tedious task to implement specific yet similar editors for numerous active cookbooks. For example, a generic data flow editor seems to be a good candidate. Also, tree editing will be required in active cookbooks independent of the underlying domain.

References

1. Fukunaga A, Pree W, Kimura T (1993) Functions as Data Objects in a Data Flow Based Visual Language. *ACM Computer Science Conference*, Indianapolis
2. Gamma E (1992) Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge. Doctoral Thesis, University of Zürich, 1991; published by Springer-Verlag
3. Gamma E, et al. (1993) Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Proceedings of the ECOOP'93 Conference, Kaiserslautern, Germany; published by Springer Verlag
4. Gamma E, et al. (1995) Design Patterns—Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison-Wesley
5. Garlan D, Shaw M (1993) An Introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering, I (Ambriola V, Tortora G, Eds.) World Scientific Publishing Company
6. Hayes-Roth F (1985) Rule-Based Systems. Communications of the ACM, Vol. 28
7. Krasner GE, Pope ST (1988) A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming, **1**(3)
8. Myers W (1995) Taligent's CommonPoint: The Promise of Objects. IEEE Computer, **28**(3)
9. Pirklbauer K, Plösch R, Weinreich R (April 1994) Object-Oriented Process Control Software. Journal of Object-Oriented Programming
10. Pree W (1994) Metapatterns: a Means for Capturing the Essentials of OO Design. In Proceedings of the ECOOP'94 Conference, Bologna, Italy; published by Springer-Verlag
11. Pree W (1995) Design Patterns for Object-Oriented Software Development. Reading, Massachusetts: Addison-Wesley/ACM Press
12. Terveen L, Selfridge P (1994) Intelligent Assistance for Software Construction: A Case Study. The Ninth Knowledge-Based Software Engineering Conference, Monterey, CA
13. Weinand A (1992) Objektorientierter Entwurf und Implementierung portabler Fensterumgebungen am Beispiel des Application-Frameworks ET++. Doctoral Thesis, University of Zürich, 1991; published by Springer-Verlag
14. Weinand A, Gamma E, Marty R (1988) ET++ — An object-oriented Application Framework in C++. In OOPSLA'88, Special Issue of SIGPLAN Notices, **23**(11)
15. Weinand A, Gamma E, Marty R (1989) Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. Structured Programming, **10**(2), Springer-Verlag
16. Weiser M (Sept. 1991) The Computer for the 21st Century. Scientific American
17. Wilson DA, Rosenstein LS, Shafer D (1990) Programming with MacApp. Reading, Massachusetts: Addison-Wesley
18. Wirfs-Brock RJ, Johnson RE (1990) Surveying Current Research in Object-Oriented Design. Communications of the ACM, **33**(9)