Check for updates

Automated Support for Software Development with Frameworks

Albert Schappert and Peter Sommerlad
Siemens AG - Dept.: ZFE T SE
D-81730 Munich, Germany
Voice: ++49 89 636-48148 Fax: -45111
E-mail: {albert.schappert,peter.sommerlad} @zfe.siemens.de

Wolfgang Pree
C. Doppler Laboratory for Software Engineering
Johannes Kepler University - A-4040 Linz, Austria
Voice: ++43 70 2468-9431 Fax: -9430
E-mail: pree@swe.uni-linz.ac.at

Abstract

This document presents some of the results of an industrial research project on automation of software development. The project's objective is to improve productivity and quality of software development. We see software development based on frameworks and libraries of prefabricated components as a step in this direction. An adequate development style consists of two complementary activities: the creation of frameworks and new components for functionality not available and the composition and configuration of existing components.

Just providing adequate frameworks and components does not necessarily yield automation and efficiency of software development. We developed the concept of relations between software components as a foundation for abstraction, reuse and automatic code generation for component interrelationship. Furthermore we suggest to supplement frameworks with an active cookbook consisting of active recipes which guide the software developer in the use of framework elements.

In this paper our concept of using relations among software components is presented and the active cookbook is illustrated as a means for developer guidance. We created a prototype to demonstrate these concepts.

1 Introduction

In the past the partitioning of software systems into components (e.g. subroutines, modules) improved productivity and quality of the software engineering process. The reasons for this were the mastering of complexity by abstraction and the possibility of reuse of existing implementations. With the introduction of object–oriented techniques it has become easier to implement domain–specific frameworks in an extensible and adaptable fashion. Such *frameworks* consist of components which are based on abstractions and their implementations. A set of abstract and concrete components builds a framework for the solution of a specific problem [Johnson & Foote, 1988]. The term *application framework* is used if the framework comprises a generic software system for an application domain.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SSR '95, Seattle, WA,USA
© 1995 ACM 0-89791-739-1/95/0004...$3.50

However, the object–oriented concepts and frameworks introduce additional complexity to software development. A framework gains its functionality by the cooperation of different components. Thus, the use of a single component may be based on its interrelationships with others that must be understood and mastered by the application developer.

Present work on software architecture also emphasizes the interrelationships between software components depending on each other to get control on this additional complexity. There the term *design pattern* is used to describe and specify component interaction mechanisms in an informal way[Pree, 1994a][Buschmann *et al.*, 1994][Gamma *et al.*, 1994]. We follow this trend of looking at the component cooperation by using explicit structural information as a base for our automation approach. The term **relation** is used to describe software component interrelationships. Thus, we consider software components and relations between them as elements our concepts are based on.

We distinguish three levels of representation for relations and components. These levels help to implement the concepts of visual interaction and automation in the prospected development tools:

visual	providing graphical manipulation and adequate presentation,
structural	containing information as a foundation for automation, and
code	for the (re)use of existing implementations.

Modern software development should build on sophisticated application frameworks that are domain–specific. The existence of such frameworks implies two complementary ways for application development **composition & configuration** and **construction** as shown in figure 1. Both ways use relations as a means for expressing (possible) component association.

- The composition and configuration of existing components in predefined/anticipated ways to solve a problem. These predefined ways of compositions are defined via relations.
- The construction of a new component or relation as a solution is necessary to provide new functionality that can't be easily obtained from the framework.

Since the use of large frameworks is complex the application developer should be guided through the development process. We propose an active guidance of the developer by an "active cookbook" presenting recipes. These recipes contain explanations and invoke tools to perform a development task. The composition and configuration spots are the places where such an active recipe sup-

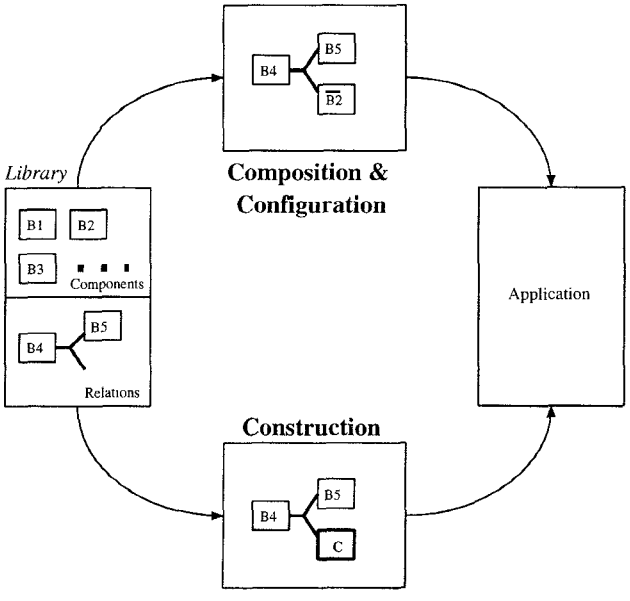

Figure 1: Two styles of development

port is viable. Thus, we extend a framework to include also recipes and domain specific tools needed by them in addition to software.

We see that our concepts can also be applied to other kinds of domain–specific libraries with prefabricated components that are not frameworks.

2 Basic Concepts

In this section the basic concepts of our approach are explained in detail. Component interrelationship can show up in different aspects. We call those that manifest on the source code level "*structural relations*". If one looks on the aspect of using a part of a framework, recipes form "*methodical relations*" defining the necessary development decisions for the configuration and composition. First we focus on the use of structural relations that are used for software construction. The methodical relations that manifest as so–called "*active recipes*" are explained second.

2.1 Relations

Structural relations are used to extract mechanisms that are usually implemented across multiple software components. Participating in such a mechanism requires a certain functionality (i.e. code fragment) at every component. By describing the mechanism explicitly the functionally is isolated and can be reused by components in different contexts.

The structural relations allow the description of such mechanisms as well as the reuse of them in different contexts. Depending on the component concept (e.g. procedure, module, class) and the structure given by components in the program source text, different approaches may be necessary to realize structural relations in a way suitable for software construction[1].

[1] Similar ideas are promoted by Mary Shaw (CMU) under the name "connector" [Shaw, 1994].

In the object–oriented context of the prototype there are (at least) three aspects of a structural relation:

- It declares the interfaces to be fulfilled by the attached components.
- It describes in an abstract way the interrelationship, such as an inter–class call–graph.
- It provides a parameterized – eventually schematic – implementation of the mechanism to be realized by it.

In the future this list will be extended by instructions for the utilization of this structural relation (an attached active recipe is explained in general in the next section).

In our prototype we have chosen C++ classes as components. In this setting a structural relation consists of two or more plugs, where components can be attached. Each plug contains the method interface and a parameterized (schema–) implementation of the component's aspects targeted by the relation.

Thus, a plug contains a set of C++ method interfaces and (partial) implementations for the relevant C++ methods. This program code linked to the relation is parameterized in the plugs. The plugged in classes provide the parameter values which are the class name and method interfaces.

A component under construction that is plugged into a structural relation obtains the interface from the plug and the implementation of the corresponding methods. The main interaction is completely visual by dragging the components in place and connecting them with the plugs via a line.

A structural relation itself can be based on other internal structural relations as well as on internal components. To construct such a combined structural relation the relations plugs are constructed as would be new components on the top level. Thus similar to components a hierarchical structure of relations is possible giving a means to abstract from lower–level mechanisms.

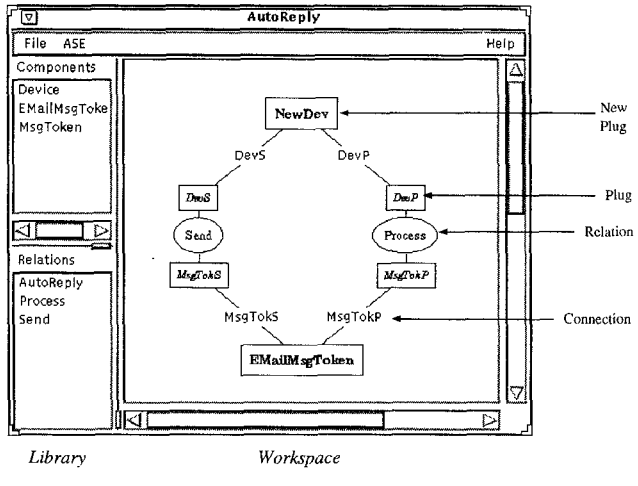

Figure 3: The inner structure of a "relation".

Based on the structural relations code generation for new components is possible. This code generation consists of parameter substitution and the combination of the code fragments in the relation's plugs and the attached components. Separate class interface and implementation files are generated for C++.

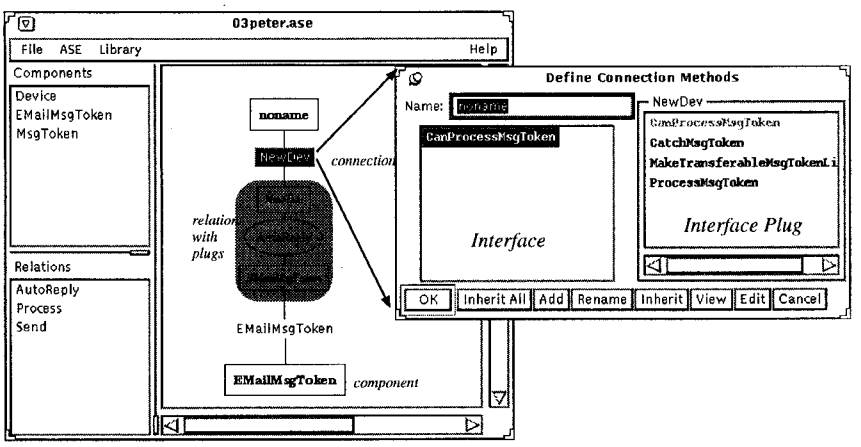

Figure 2: A relation's plug provides interface and implementation to a newly constructed component (noname).

2.2 Active Guidance

A framework contains predefined places that need to be configured and where application specific parts are composed. These places are called *hot–spots* [Pree, 1994b]. A developer using a framework must know the hot–spots for a given problem and how to adapt them to the application's needs. Large frameworks typically suffer from a steep learning curve. In the past a developer needed to consult the source code or a large reference manual to find the hot spots and how to configure and compose the building blocks of the framework.

To ease the learning curve and use of a framework the cookbook idea emerged. A cookbook consists of recipes that describe in an informal way how to adapt a framework to specific needs. Examples of such passive cookbooks for graphical user interface frameworks are the MVC cookbook [Krasner & Pope, 1988], the MacApp Cookbook [MacApp, 1989, 1989], and the ET++ Cookbook [Weinand *et al.*, 1989]. A cookbook's recipe is typically structured into the sections purpose, procedure, and source code example(s). A developer has to search for the recipe that is appropriate to the problem. This recipe is then used by adhering to the steps given in the procedure section. Very often a solution is obtained by using and extending the example source code.

We emphasize the cookbook idea and extend it to the active cookbook. First the recipes with their inherent references to other recipes and source code indicate a computer–based implementation in a hypertext structure. Second we see that some steps required by recipes are better supported by the integration of specialized (visual) tools and editors than by textual editing code. Further support of the developer is obtained by demanding required recipe steps before others and by generating program source code based on decisions of the developer. Thus the development system provides for every action exactly the appropriate and specialized tool that is needed.

As stated above especially the use of visual tools is recommended in recipes of an active cookbook. Such visual tools allow an easy arrangement of components and provide a better overview than textual ones. An example may be a visual editor displaying (part of) a class hierarchy and allowing the creation of specific subclasses visually (see figure 4).

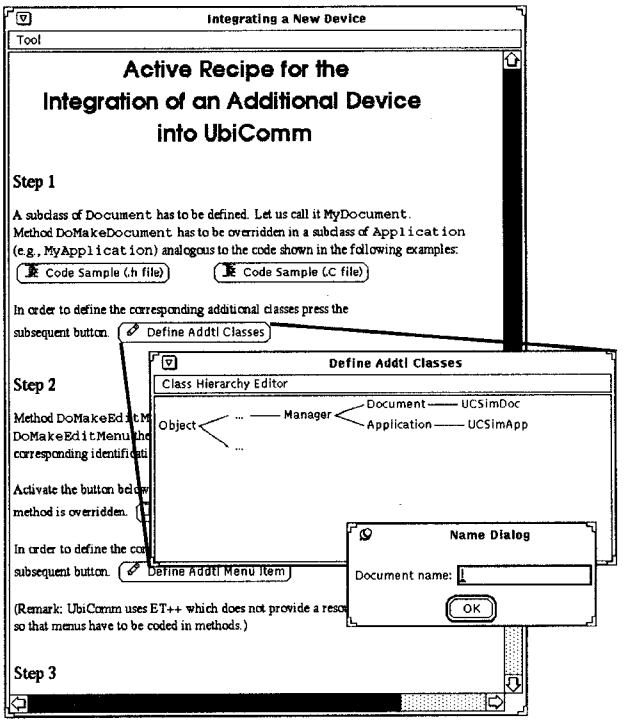

Figure 4: An active recipe with an activated tool.

2.3 Desirable Features of a Development System

There exists a set of desirable features a proposed development system should have. Those that are already demonstrated by our prototype are

- visual development,
- active conduction,
- flexible guidance,
- context–sensitive behaviour, and
- incremental extensibility.

In the prototype the features are interwoven in order to achieve a seamless support for software development. For clarity we discuss them separately in the following.

Other features not yet addressed are cooperative and distributed software development, development process control and the integration of other non-software artefacts into the development system.

Visual Development

In many cases a visual way of doing software development can be more efficient. The complex structural relationships in software systems can be better mastered by a visual representation than just having many source text windows. Especially for routine tasks a visual representation supplemented with automatic code generation will improve productivity.

Active Conduction

Each recipe includes the tools needed by its steps. These tools are activated by the recipe. In addition to the navigation through recipes they provide an active support to the developer according to the motto *"the right tool at the right place."* There is no need to search for the tool or to parameterize it for the specific needed task. These things are provided by the recipes. The activated tools signal (successful) completion of their task to the recipe so that further steps can be started.

Flexible Guidance

The active–cookbook approach allows an easy way of navigation through the recipes using hyper–links. Once a user has found the recipe he wants for further development this recipe guides him through the needed configuration and composition steps. The developer gets the amount of explanation he needs. Although the sequence of development steps are predefined, he can freely navigate between them.

Context – sensitive Behaviour

Depending on the design decisions, the development system must vary its conductive steps. For example, depending on the components or relations used when applying a recipe, different configuration activities may be necessary. This context sensitive behaviour is realized by attaching recipes to the components and relations to be used. Thus the decision for a specific component will trigger the application of the attached recipe. Further rules can be present in the recipes to handle more complex cases of changing the development procedure.

Incremental Extensibility

It is necessary to provide a way for evolution and migration, when introducing a new idea of support for software development. Evolution of framework support is provided by a recipe editor and a means for attaching recipes to existing or newly developed components. Migration is supported by using existing source code and augmenting it with structural information and with recipes.

2.4 Preparation of SE–Environment

The proposed software engineering environment has to be filled with information to become useful. A framework consisting of components and structural relations must be developed or an existing one integrated into our software engineering environment. The typical use of the framework has to be identified and described in recipes. Last the tools required by the recipes' steps must be developed.

Creating the Library

It is a major effort to develop a domain specific framework. Ideally the code of the framework is created with structural relations in mind, so that the relations and components are already part of it. Nevertheless it is possible to use existing framework source code and integrate it into our proposed environment by providing or extracting the structural information. Architecture principles and design patterns will make the selection of necessary structural relations easier.

Creating Recipes

An editor for active recipes will allow the creation and modification of the text of a recipe. Furthermore links to existing tools and editors may be included into the recipe. Editing the links provides the information on the actual tool and its parameterization. In the same manner links to other recipes can be created.

Ideally the developer of a framework creates the according recipe(s) in parallel to the development of the framework. The creation of the active guidance during framework development has some benefits:

- The later use of the framework becomes more transparent to the framework developer. He will recognize how hard it is to use and adapt the framework.
- The active guide provides a concrete way of communicating the architecture and implementation of the framework with its potential users.
- Design flaws that hinder the adaptation and use of the framework, e.g. improper abstractions, may be detected more early.
- Useful points of automation and appliance of visual tools are indicated when the guidance steps are written down.

Creating additional tools and editors

The process of writing active recipes will inspire possible automation of the development process. New specific tools or editors may become admirable or necessary. Some generic visual editors or even a simple textual editor will be sufficient for many tasks. By providing an own framework for developing new visual tools this job is more easy to accomplish. In the future we may provide such a framework and active recipes for writing new tools and editors that generate the program code.

3 Conclusion

Software development based on domain–specific framework consists of two complementary development styles:

- Composition and configuration of framework parts.
- Construction of new software parts.

We propose tools to support these development styles. These tools shall provide visual interaction, active developer guidance and the automatic generation of program code. Buy making use of existing program code and extending it we maintain a way for migration.

The use of structural information as a base for construction and code generation is an innovative idea in the tool support. We go further than browsers for object–oriented software development in activating the use of such structures by providing them as separate abstraction and reuse entities that contain also implementation code.

However, it is an open point how to structure a framework and implement it around a set of relations. A well designed framework consisting good abstractions will not only ease the design of applications but also by the reuse of relations the design of new framework elements.

The other more pragmatic way followed was augmenting frameworks with recipes and visual tools to support software development with them. There exists a similar problem to software documentation. It is necessary to maintain the guides and tools in the same progress as the framework improves. Controlling this framework evolution via active recipes will make the corresponding progress in the active guides convenient.

There is the problem of having or creating frameworks suitable to be supported by active recipes and using structural relations. Good frameworks can be the base, but their development and use is not (yet) a mature technology. Only a few high sophisticated developers are needed to create such a framework for a given domain. The cost of establishing such frameworks with integrated recipes pays off when less experienced developers are actually using the framework in an efficient way without the need to become framework experts.

The writing of active recipes during framework development will make the use of a framework more transparent to its developer. He will recognize how hard it is to use the framework, write down the typical adaptation cases and find possibilities for automation and additional tool support. If such (active) recipes are sketched in advance they provide a vehicle to communicate the use of a prospected framework with the potential users and domain experts. Such communication may improve the design and implementation of the framework.

A software engineering methodology must be formulated supporting our two styles of development with active tool support for applications. In addition the methodology must help in creating frameworks consisting of components, relations, active recipes and visual tools.

References

Buschmann, F., Jäkel, C., Meunier, R., Rohnert, H., & Stal, M. 1994. *Pattern–Oriented Software Architecture.* Siemens AG, Corporate Research and Development, Munich, Germany.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1994. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley.

Johnson, R.E., & Foote, B. 1988. Designing reusable classes. *The Journal of Object–Oriented Programming*, **1**(2), 22–35.

Krasner, G.E., & Pope, S.T. 1988. A cookbook for using the model–view–controller user interface paradigm in smalltalk–80. *Journal of object–oriented programming*, **1**(3).

MacApp, 1989. 1989. *MacApp II Programmer's Guide.* Apple Computer.

Pree, Wolfgang. 1994a. *Design Patterns for Object–Oriented Software Development.* Addison-Wesley.

Pree, Wolfgang. 1994b. Meta Patterns — A Means For Capturing the Essentials of Reusable Object–Oriented Design. *Pages 150–162 of:* Tokoro, Mario, & Pareschi, Remo (eds), *Object–Oriented Programming, ECOOP '94.* Springer–Verlag.

Shaw, Mary. 1994 (Jan.). *Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First–Class Status.* Tech. rept. CMU-CS-94-107. Carnegie Mellon University, Pittsburgh, PA 15213–2890.

Weinand, A., Gamma, E., & Marty, R. 1989. Design and Implementation of ET++, a Seamless Object–Oriented Application Framework. *Structured programming*, **10**(2), 63–87.