A Paleontological Perspective on Designing Adaptable Software Systems

Oliver Stiemerling and Armin B. Cremers

University of Bonn
Department of Computer Science III
Römerstraße 164
53117 Bonn
Germany
{os | abc}@informatik.uni-bonn.de

Abstract

This paper investigates the application of a recently developed theory in paleontology and genetics to the design of adaptable software, i.e. systems whose properties can be changed in order meet diversified or dynamic requirements. About 530 Mio years ago, hierarchical "modularization" appeared as a new property in the genome. It permits rapid and effective changes to the code on fairly high levels of abstraction (e.g. by varying the number of eyes or legs). Paleontologists today believe that this new characteristic was the cause for an era of enormous evolutionary change – the Cambrian explosion – which eventually led to the later move out of the sea onto land. Similar to living systems, for complex software systems adaptability is a useful property, because different individuals might use them for different tasks, in different environments etc. The developer of an adaptable software system has to choose an architecture that provides the necessary degree of flexibility. Traditional approaches include parameterization with ini-files or even open source policies that permit control over every aspect of the software system. These approaches have the disadvantage that their flexibility is either too restricted (changing parameters in ini-files) or too complicated (changing the source code) for many applications. This paper – adopting the way nature successfully provided the structural foundations for adaptability in the advent of the Cambrian – presents a component-based approach that provides scalable adaptability for multi-user software systems distributed over a network like the Internet.

Introduction

A software system is called *adaptable*[1] if some of its properties, which are stable during regular use, can be changed in order to meet diversified or changing requirements (see e.g. Henderson and Kyng 1991). Adaptability is useful for a variety of reasons. For instance, human users are not alike and there is no "one-best-way" in supporting their work (Ackermann and Ulich 1987). The same holds for groups of people and whole organizations. If a heavily computerized production or service process has to be changed in order to accommodate dynamic market conditions or customer profiles, adaptability of the underlying software system is almost a *sine qua non* requirement. Furthermore, with adaptable software, developers can target larger market segments. Software development is expensive and most software systems are not completely custom built anymore.

While the motivation for adaptability is quite clear, the methods for designing adaptable software are far from mature. The main reason for this lies in the fact that adaptability raises a number of quite diverse questions, requiring expertise in different areas:

- *How can we determine the necessary points and degree of adaptability*? This question concerns the design process and is driven by three factors (Trigg 1992): fluidity, diversity, and uncertainty of requirements. Especially the first and third factor involves the future use of the system and thus we will never have a complete methodological solution. However, current requirements capturing and design techniques can be extended to take anticipated future changes into account (see e.g. the *change case* methodology by Ecklund et al. 1996).
- *How are adaptations initiated, selected and implemented*? Assuming we have an adaptable system, this question concerns the way the need for an adaptation is discovered, how an appropriate adaptation is developed, how the decision for or against the adaptation is made, and how the adaptation is finally implemented. (Kühme et al. 1992) suggest these four factors as basis for a taxonomy of adaptation control mechanisms. They distinguish, which step is controlled by the user or by the system. We call a system *adaptive*, if the control resides mostly with the system, and *user tailorable* otherwise. In the case of adaptive systems, there are approaches using techniques similar to evolutionary selection for the decision step.
- *How can we support adaptability in the software architecture*? Regardless of whether adaptations are

[1] Synonymous to *adaptability*, the literature in the area of human-computer interaction (HCI) often refers to this property of a software system as *tailorability*. The term adaptability is used more frequently in the area of software technology (ST). In this paper, the terms are used in the following way: if we refer to more technical aspects, we use *adaptability*; if we are in a more user-oriented context (the sections on the MUD application and the 3D tailoring interface), we use *tailorability*.

controlled by the user or the system, they have to be supported on the technical level.

This paper addresses the last of the three questions – *software architectures providing adaptability on the technical level*. Thus, we are not concerned with mechanisms for controlling e.g. the selection of certain adaptations (second question) or with design methodologies for eliciting and documenting diverse and dynamic requirements (first question). The rest of the paper is structured as follows:

In section 2 we briefly describe the Cambrian explosion, an event of that time in evolution history that is characterized by enormous evolutionary change and diversity of life forms. In the last few years, evolutionary biologists and geneticists have pinpointed certain regulatory genes as the culprits of this explosion. The section also describes the function of these genes. In section 3 we relate them to concepts in software architecture and discuss how far we can stretch the similarities and what lessons software engineers can learn from nature for the design of adaptable software systems. Section 4 describes the EVOLVE system, our implementation of the ideas developed earlier. Section 5 describes the use of our system to provide a distributed multi-user application with the property of adaptability. Section 6 summarizes and discusses what we have gained from the paleontological perspective taken in the paper and proposes directions of future research.

The Cambrian Explosion

Life on our planet can be traced back as far as 3.5 billion years (most of the information presented in this section is taken from Erwin et al. 1997). Simple, single-celled organisms ruled the earth for about 3 billion years. Only some algae, drifting through the oceans, achieved multicellular grade about a billion years ago. 565 million years ago, other, larger multicellular organisms appear in the fossil record. 35 millions years later, an explosive increase in organism variety and complexity took place. For instance, fossils from this area are the first to reflect the presence of animals with limbs. In the following 10 million years, it appears as if nature rapidly experimented with different "animal architectures", resulting in, for instance, the fearsome creatures with five eyes in figure 1.

At the end of this period of rapid change, only about 37 distinct "body architectures" remained which are recognized among present-day animals and form the basis of the taxonomic classification of phyla. It appears that during the Cambrian explosion, the differentiation of simple-structured multicellular organisms into animals with organs, limbs, etc. was determined. Later changes only modified elements of these basic body plans.

The explanation of the sudden evolutionary development during the Cambrian explosion has recently been traced to genetics.

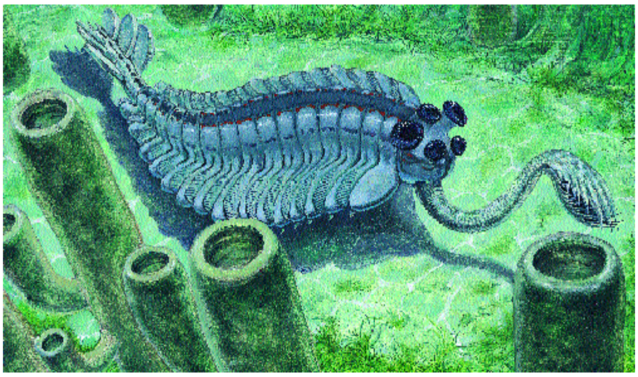


Figure 1: A (not surviving) result of the Cambrian explosion: the five-eyed Opabinia. (taken from (Erwin et al. 1997), illustration by D. W. Miller)

Regulatory genes: the genetic culprits of the Cambrian explosion

The early development of complex animals is controlled by regulatory genes. The process runs through a sequential activation of regulatory switches that in turn activate other genes. Thus the development and differentiation of different parts of the animal body are organized by hierarchically structured genetic information.

Examples for regulatory genes are *Hox* genes, which are activated very early on in the cascade and determine the basic body plans of animals. They are arranged together in a gene cluster and the position of the *Hox* gene within this cluster roughly determines the position of the body part the gene encodes. Thus, the first *Hox* genes in the cluster usually specify the head and associated structures, while the genes in the middle produce arms and so on. The exact morphology of e.g. an arm is determined later on in the cascade by other regulator genes. A consequence of this hierarchical specification is that small changes in the upper levels of the hierarchy can cause dramatic effects in the fully developed animal, like e.g. an additional eye.

Furthermore, recent research has shown, that these high level regulatory genes are extremely similar among different species. For instance, it was shown that a *Hox* gene specifying the existence and position of an eye in an octopus could be inserted into the genetic information of *Drosophila* (the common fruit fly). This causes the fly to develop additional (*Drosophila*) eyes, implying the high degree of similarity of the genetic regulatory system of both animals.

Concerning the relation between the evolution of the genetic code and the Cambrian explosion, paleontologists today propose a scenario in which the existence of a complex hierarchical regulatory system permitted nature to experiment very rapidly with many different life forms by making small but highly effective changes at high abstraction levels of the genetic information (see Erwin et al. 1997).

Summing up, we want to make two observations:

- The rapid and highly successful phase of evolutionary change during the Cambrian was made possible by the *hierarchical organization of the specification* of complex organisms.
- The regulatory system which implements this hierarchy is remarkable *similar in different species*.

How does this reflect on the design of adaptable software systems?

Designing Adaptable Software Systems

Modularization and hierarchical specification of software systems are being discussed in the software engineering community for quite some time now. (Parnas 1972) discusses the use of modularization for limiting the impact of changes within the software system and with this goal in mind suggests criteria for the decomposition of complex systems into modules. (DeRemer and Kron 1976) introduce the notion of *Module Interconnection Languages* (MILs) as the foundation for a discipline of "Programming-in-the-large" (creating new applications and new functionality by composing software at a high level of abstraction) as opposed to "Programming-in-the-small" (coding of functionality line by line in a traditional programming language). These languages allow a software system to be defined on two distinct levels: first, the modules (or components) of the system are specified in traditional programming languages. Second, the composition of these modules is specified in the MIL. Then the specifications are compiled resulting in the final software system. Today, many application development systems make use of similar techniques for rapid prototyping and development. For instance, VisualBasic (Microsoft 1996) and VisualAge (IBM 1998) use prefabricated components that are programmed "in-the-small" in C++ or Java to visually (hence "Visual...") compose new applications. (Szyperski 1998) gives an overview of the software technical foundations of component oriented-programming. The central concept behind the success of the component-oriented approach to programming is *reuse*. Reusing as many parts of other software system as possible has many obvious advantages:

- Reduction of development *time*.
- Reduction of programming *errors* ("in-the-small"), because the reused software components have already been employed in other applications.
- *Simplification* of the programming ("in-the-large") process, if one assumes, that composition of applications is easier to achieve than programming them in a traditional language.

Looking at the five-eyed fellow in figure 1, one can observe the same concept of reuse at work in natural evolution. Because of the hierarchical specification of genetic information it was as simple for nature to experiment with additional eyes or limbs, as it is for the designer of component-based software today to experiment with different compositions of basic software components.

However, we believe the structural similarities between genetic structure and component software cannot be followed much further. While there are successful approaches which model compositional aspects of software using natural science metaphors (e.g. the Chemical Abstract Machine model, see Inverardi and Wolf 1995), we currently do not see how the fine technicalities of the composition of computational entities can benefit from copying these mechanisms directly from nature. The connection of an arm to the midsection of the body is conceptually too different from two software components interacting via method call.

Coming back the two observations made at the end of the last section, we now approach the question of how one can design a software system that permits effective *adaptations at high levels of abstraction* and is *generic* in the sense that the adaptation functionality can be applied to a variety of

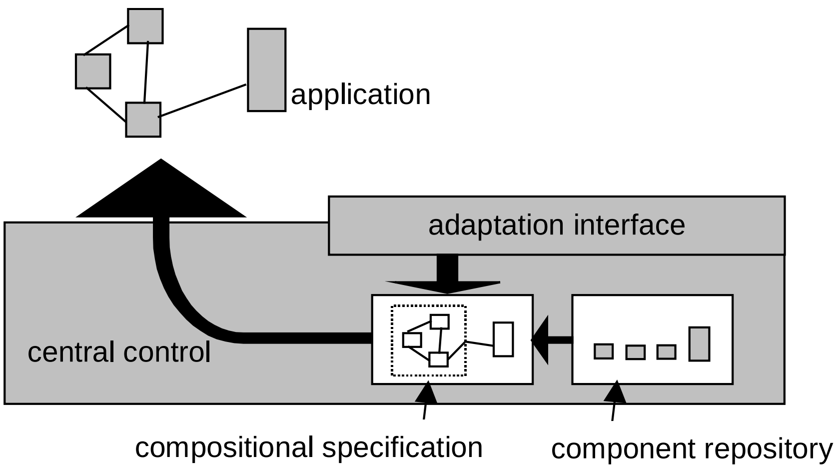


Figure 2: Conceptual architecture of an integrated run-time and adaptation environment

application domains.
So far we have described the application of the concept of hierarchical specification in the process of software development. The software resulting from these development processes, however, is just as monolithic and un-adaptable as software resulting from non-component-based processes, because the hierarchical structure is usually lost during compilation and therefore cannot be changed without changing and re-compiling the original source code.
In order to achieve the goal of adaptability during the lifetime of one version of a software system, the hierarchical structure has to be retained in manipulatable form in the system. Together with the goal of genericity, this suggests an integrated *runtime and adaptation environment*. The conceptual architecture of such an environment is depicted in figure 2.
The *central control* retains a specification of the hierarchical structure (*compositional specification*) of the application and a *component repository* containing the blueprints (or prototypes) of single components. At system startup time the compositional specification is evaluated by the central control and the components are instantiated and connected accordingly. During run-time, changes can be made to the compositional specification via the *adaptation interface*. The central control immediately implements changes to the specification within the running application. Thus, the specification and the application are kept consistent. Similar environments are used for distributed application management (see e.g. Magee et al. 1995 and Bellissard et al. 1996).
The adaptation interface in figure 2 is not a user interface. It only offers a set of methods to adaptation control modules. The function of these modules (which are not shown in figure 2) reflects the second question of the introduction: "*How are adaptations initiated, selected and implemented*?" Possible control modules are user interfaces (e.g. visual programming style interfaces, see Myers 1990) or intelligent agents that automatically adapt the application.
This concludes the description of the motivation and basic concepts of integrated run-time and adaptation environments. (Stiemerling 2000) describes the architecture in more detail in form of a mathematical model that comprises of the fundamental data structures and algorithms. In the following we describe the EVOLVE system that is our implementation of an integrated runtime and adaptation environment.

The EVOLVE Platform

The EVOLVE platform is designed to support the deployment and subsequent adaptation of arbitrary distributed component-based multi-user applications. It is independent from domain specific functionality and can thus be used to provide many different software systems with the property of adaptability. Figure 3 depicts a simple example for a component-based application made adaptable by the EVOLVE platform – a *shared to-do list* employed by two users to coordinate their tasks:

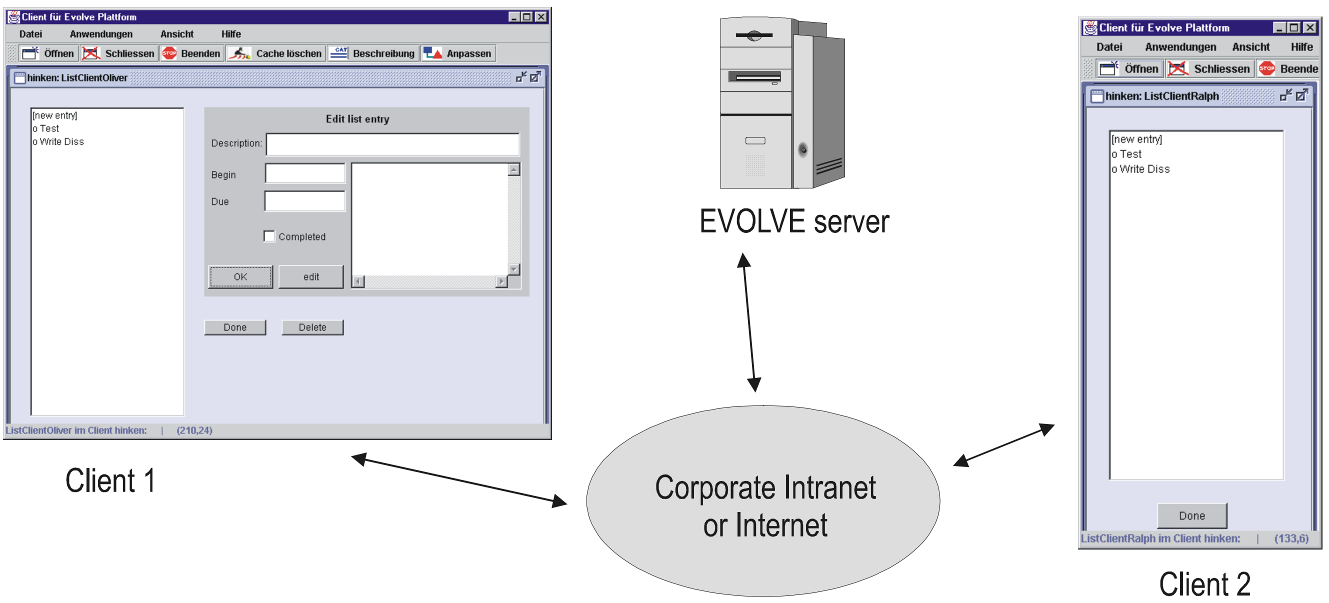


Figure 3: The EVOLVE platform supports multi-user client-server applications distributed over TCP/IP networks (here the *user perspective*, showing two clients of a shared to-do list application)

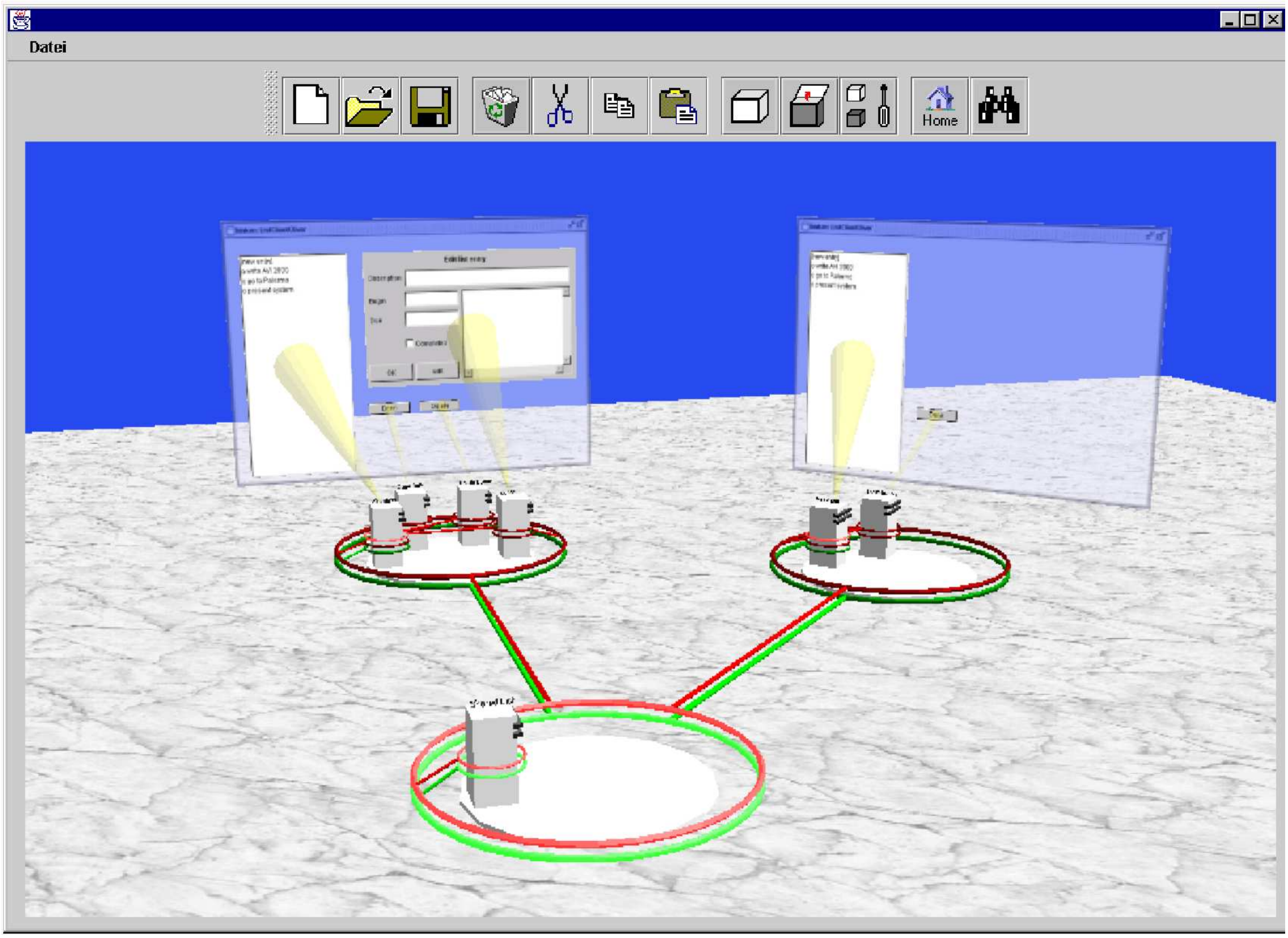


Figure 4: Screenshot of the 3D tailoring interface (showing the *tailoring perspective* of the application depicted in figure 3)

Client 1 belongs to a manager, client 2 to a subordinate. The actual data of the shared to-do list is stored on the EVOLVE server (in the middle of figure 3) employing a (invisible) server component. The clients are tailored to meet the requirements of their respective owners. While the subordinate may only see the contents of the list and mark entries as "done", the manager can actually add new entries to the list and delete them. The distributed application is built from a set of components (visible ones like the buttons, the list and the editor; and invisible ones like the data storage component on the server).

In a traditional system, the composition would be static after development and deployment. The EVOLVE platform, however, maintains – and permits the manipulation of – the system's component structure. During the use of the system, a system administrator, outside consultant or even an end user can switch to the *tailoring mode* (figure 4) in which he can inspect and manipulate the entire distributed application (if he or she has the right to do so).

Figure 4 depicts a 3D user interface for component-based tailoring which accesses the flexibility provided by the EVOLVE system. The two circles in the background represent the two clients, together with the virtual screens onto which appearance and position of the visible components are projected. All components – visible and invisible – are represented as boxes depicting the compositional structure of the application. The circle in the foreground represents the server that contains the invisible component for storing the contents of the shared to-do list.

The tailor can move around the 3D component scene and perform manipulations, e.g. concerning the positioning of the visible components on the screen or the connections of the components. He can remove component instances or add new ones from a repository (not shown in figure 4). The parameterization of component instances can also be manipulated. In short, the tailor has full control over every aspect of the application's composition, whenever the requirements of the supported group change.

In the shared to-do list example, the manager's trust in the subordinate might have increased enough to permit the subordinate to directly delete items from the shared to-do list – instead of only marking them as "done" for later control and deletion by the manager. In this case, the tailor would add a delete button component to the client of the subordinate.

While the example of adding a button is admittedly a rather simple tailoring operation, the EVOLVE platform permits the application of re-composition (and re-parameterization) operations to any part of a distributed software system. Furthermore,

- *tailoring operations can be performed during runtime*. A system administrator can add the delete button, while the user is working with the shared to-do list. The system does not have to be shut down and the state of all other components is maintained.
- *tailoring operations can be performed remotely*. The whole system can be tailored from any workstation in the network. This feature supports models of centralized and decentralized system management.
- *the effect of tailoring operations can be shared among many users*. In the example, if other users share the definition of the subordinate client, the effect of the tailoring operations is propagated to all running instances of that definition.
- *the effect of tailoring operations can be restricted to subgroups of users*. This feature permits the accommodation of individual differences. In the example, there could, for instance, be a need to exclude a user who would misuse the delete button.
- *tailoring operations can be applied to any level of the compositional hierarchy*. A system administrator might inspect and manipulate the system on a very fine-grained level, while an end user might prefer a more high level view.

Summarizing, the EVOLVE platform is responsible for maintaining component structures during runtime and for providing functionality for manipulating these structures. The second central element of the approach is the component model, i.e. the specification that tells the programmer exactly what a component is. The next subsection described what a component is in the context of the EVOLVE platform.

EVOLVE components: FLEXIBEANS

The component model of the EVOLVE platform is the FLEXIBEANS model, which is an extension of the JAVABEANS model (JavaSoft 1997). JAVABEANS is the component model of the JAVA programming language. The component model specification tells the programmer exactly what conventions a piece of code has to adhere to for it to be a component. These conventions usually concern:

1.) the way a component *interacts* with its environment, i.e. other components. The JAVABEANS component model, for instance, offers JAVA events as the only interaction primitive. A JAVA event is actually a void method call from one component instance to another component instance (or – in the case of multicast events – sequentially to a set of component instances). The void method call is parameterized with a stateful object (the event object) and permits the other component instance(s) to execute some reaction and then return the method call to the calling component instance.

2.) the way components can be *manipulated* and *inspected* by IDEs. In the case of JAVABEANS, a component capable of sending out events has to offer standardized methods for registering and de-registering other components interested in these events. When JAVABEANS instances are composed or re-composed, the IDE calls the respective manipulation method to connect (or disconnect) the two instances. Because of the standardized naming patterns of the methods, the IDE can learn everything it needs to know about the composability of a component by simple inspection.

3.) the way components are *packaged* for delivery and deployment. Some JAVABEANS components rely on a number of JAVA classes, bitmaps and other resources all of which have to be present for the component to be instantiated. These resources are packaged in JAR-files (Java ARchive) together with a “manifest file” which tells the IDE which classes are JAVABEANS and which are only of auxiliary nature.

Apart from the fact that they do not explicitly support distributed applications, JAVABEANS have – in the context of our work – the additional disadvantage that they receive events of the same type (e.g. button-click events) always on the same port (read: event handling method). In order to distinguish between two different event sources an adapter object has to be introduced which explicitly contains code to forward button-clock events to different handling methods depending on the event source (button 1 or button 2). For the purpose of run-time tailorability, dynamically generating and compiling code would be rather cumbersome. Furthermore, while events proved to be sufficient for the implementation of the search tool example, sometimes a less “pushy” interaction style between component instances appears to be more appropriate (e.g. when one instance simply wants to publish some information without forcing another instance to react to this information immediately, as the JAVA event interaction style prescribes). In order to remedy these shortcomings we have developed the FLEXIBEANS component model which extends the JAVABEANS component model with the concepts of:

- *named ports*, permitting the differentiated event handling on the compositional level mentioned above, without having to dynamically produce and compile (adapter-) code.
- *shared objects*, permitting a less strongly synchronized style of interaction in the fashion of a “pull”-like data flow (need- and not creation-driven exchange of data).
- *remote interaction*, permitting the composition of distributed groupware applications based on JAVA RMI (Remote Method Invocation). A button situated on one machine can, for instance be directly connected to a component on another machine.

The complete FLEXIBEANS specification can be found in (Stiemerling 1998).

The EVOLVE platform and the FLEXIBEANS component model serve as proof of concept for the idea of providing adaptability / tailorability by maintaining the compositional structure of an application after initial development. However, the EVOLVE platform is only that – a platform. In order to show the applicability of the approach, a number of distributed applications were development on top of the platform. The following section is devoted to one of these applications – a groupware application (i.e. an application supporting cooperative work. For an overview of groupware see e.g. Ellis et al. 1991). Another application concerns the document exchange between cooperating companies in the steel industry (the ORGTECH project, see Stiemerling et al. 1998).

An Example Application: MUD

The name "MUD" stands for MULTI USER DUNGEON. Despite this name, the system is not a game, but denotes a component set for constructing and evolving text-based, virtual meeting rooms. Users can log into these rooms, see who else is present at the moment and communicate based on text messages. (Churchill and Bly 1999) describes the use of a similar application to support remote cooperation in a research center. The primary attractiveness of a MUD lies in the fact that it is a communication tool with characteristics somewhere between email and the telephone

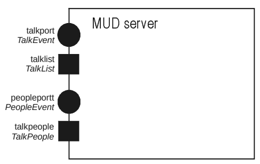
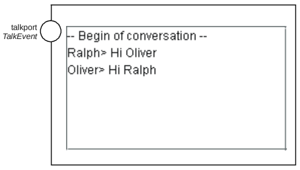
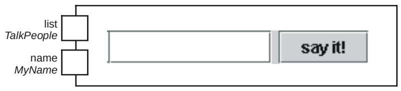
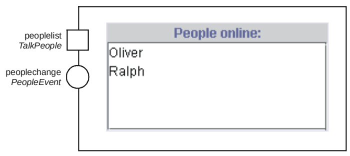
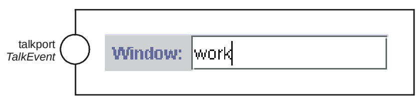
	**MUD server component** This server component is responsible for storing the list of active participants and for distributing the text messages instantly to each party. It also sends out events informing clients whenever the current list of participants has changed. Furthermore, it permits the downloading of specific sequences of the conservation history in order to help late-comers to enter the discussion.
	**Main text window** This component can receive and display the text messages distributed by the server component. Whenever the contents exceed the visible area, a scrollbar appears which permits moving the window to the part of the conversation which is currently of interest. However, new text is always added at the end.
	**Text input** This component permits the user to input and send a text string to the server component which distributes it to all other participants. If a name component (see below) is attached to it, then the name of the user is added to text string: "Joe> ....".
	**User Name** The user can type his or her name which is entered into the list of active participants maintained by the server component. The user can change the name during a session (e.g. in order to add things like "back in 5 minutes" or "out to lunch"). The change becomes instantly visible in the people online component (see below)
	**People online** This component visualizes the list of active participants maintained by the server component. Whenever a change in this list occurs, an event is received and the visualization is updated accordingly.
	**Repeater component** This component is intended to support late-comers to the MUD room. Pressing the button repeats the last x lines of text stored by the server component. The repeater component connects both to the server component and the main text window.
	**The sound warner** This components can be parameterized with a string. Whenever this string appears in a conversions (it can also be the name of another participant), the EVOLVE client emits a ringing sound. This component is intended for users who can not constantly monitor the conversation in the room, but need to be aware whenever certain key words (like "lunch") are uttered.
	**The window warner** This component provides essentially the same functionality as the sound warner. However, instead of the ringing sound, a small window appears on top of the windowing system.

Table 1: The components of the MUD component set

(Excerpt from an interview reported by Churchill and Bly 1999, p. 102): *"As I see it, it fits between where you have the telephone and electronic mail"*). It does not require constant attention like the telephone, but is still more synchronous than email, because messages are shown instantly to all collaborators in a room. (Churchill and Bly 1999) also reports on the use of the MUD application to support keeping in contact with the research center when working at home or abroad.

The implementation of the MUD as a set of FLEXIBEANS is designed to demonstrate how a (groupware) application can benefit from the adaptability provided by the EVOLVE platform. In particular it is shown how diverse requirements stemming from different uses of the tool can be met by differently tailored versions.

The MUD component set

The initial set of FLEXIBEANS components comprises of eight components which are depicted in table 1.

Different MUD compositions

The components shown in table 1 can be composed yielding MUD applications for different uses. Figure 5 depicts a number of compositions for different users and purposes.

The super user (top) has a rather complicated application with a lot of functionality. He (or she) has added a number

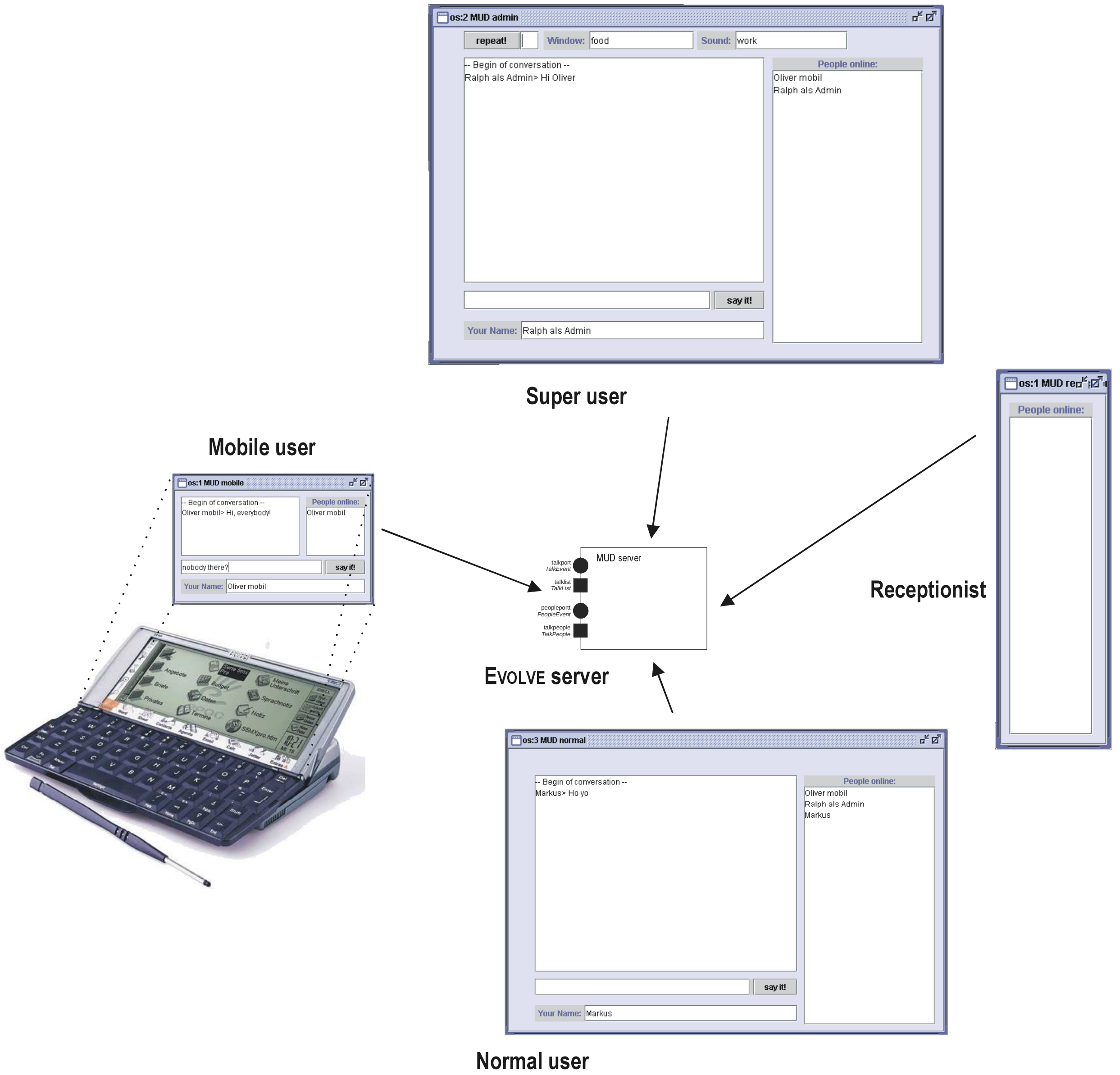


Figure 5: Different MUD compositions

of *warner* components to the application, in order to get informed, whenever a certain word is uttered in the conversation, when he is currently not watching the EVOLVE client. The receptionist of the example organization does not want to participate in the conversation. However, for him (or her) it is quite useful to have access to the list of people who are currently only. Therefore, his client only contains a elongated *people online* component (right). A normal user (bottom) might want to see the people online and participate in the conversation. Finally, a mobile user (left) might have different requirements concerning the visual appearance (size and position of visible components). All these different user applications are composed from the same component set shown in table 1.

Furthermore, there are a number of subtle ways in which MUD compositions can differ to meet diverse requirements. For instance, the *name* component could be disconnected from the *MUD server* component in order for the user not to appear in list of the *people online* component. Or a *warner* component could be connected to the repeater instead of the server in order to be able to search in the conversion history for a certain text string.

Further evolution of the system

The compositional mechanisms are not only restricted to the client of the MUD systems. One could imagine a suite of server components which integrate the MUD system with other telecommunication infrastructures. For instance, it should be possible to add a component to the MUD server, which acts as an SMS (Short Messaging System) gateway, transmitting the whole conversation (or certain key phases, similar to the *warner* components) to a mobile phone. This could be useful in a helpdesk scenario in which a system administrator is often away from his workstation. Requests for help could be sent directly to the mobile phone.

Summarizing, the MUD example demonstrates how the concept of component-based adaptability can be applied to the design of a primarily synchronous groupware system. The initial set of components can be employed to compose (and re-compose) different types of MUD applications. Additionally, the system can be further evolved by adding new components which provide functionality not initially anticipated.

Summary, Conclusions and Future Work

We have demonstrated how one can design adaptable software systems based on a structuring principle taken from the genetic code of animals. This principle – the hierarchical modularization of genetic information – is believed to have paved the way for an era of enormous evolutionary change in the Cambrian period. In order to achieve the same degree of flexibility in complex software systems (obviously on another time-scale), we have proposed the notion of an integrated runtime and tailoring environment. On top of this platform, arbitrary component-based applications can be deployed and evolved. This decision for the division in application-specific components and an application-independent platform is supported by the observation, that the regulatory genes (*Hox* genes) controlling the hierarchical modularization in the genetic code are similar across different animals (drosophila and octopus).

We have presented our prototype implementation of such an runtime and tailoring environment: the EVOLVE platform. An example application – the MUD system – demonstrates how the requirements of different exemplary types of users (*administrator/super user*, *mobile user*, *receptionist* and *normal user*) can be met by different compositions of an initial component. Furthermore, we have argued that the component-based adaptability approach also supports extensions of an application with new components in order to meet unanticipated requirements on its functionality. The prototype and the example application clearly show the feasibility of the component-based tailorability approach. Furthermore, we believe this work to be a nice example for how in particular the structural aspects of software engineering can benefit from the natural sciences.

For future work, we intend to review methods from paleontology (e.g. methods for estimating rates of evolutionary change) with respect for their usability in software engineering. We are especially interested in developing methods for giving software “the right” degree and kind of flexibility.

Acknowledgements

Ralph Hinken implemented the EVOLVE prototype as part of his Master thesis, Michael Hallenberger the 3D tailoring interface. Furthermore, we want to thank the colleagues and friends the department of computer science III for many fruitful discussion.

References

Ackermann, D. and Ulich, E., “The Chances of Individualization in Human-Computer Interaction and its Consequences,” in *Psychological Issues of Human Computer Interaction in the Work Place*, M. Frese, E. Ulich, and W. Dzida, Eds.: Elsevier Science Publishers B.V. (North-Holland), 1987, pp. 131-145.

Bellissard, L., Atallah, S. B., Boyer, F., and Riveill, M., “Distributed Application Configuration,” in: Proceedings of 16th International Conference on Distributed Computing Systems., Hong-Kong, IEEE Computer Society, pp. 579-585, 1996.

Churchill, E. F. and Bly, S., “Virtual Environments at Work: ongoing user of MUDs in the Workplace,” in: Proceedings of WACC '99 (Work Activities Coordination and Collaboration), D. Georgakoloulos, W. Prinz, and A. L. Wolf, Eds., San Francisco, ACM Press, pp. 99-108, 1999.

DeRemer, F. and Kron, H. H., “Programming-in-the-Large Versus Programming-in-the-Small,” *IEEE Transactions on Software Engineering*, vol. 2, 2, pp. 80-86, 1976.

Ecklund, E. F., Delcambre, L. M. L., and Freiling, M. J., “Change Cases: Use Cases that Identify Future Requirements,” in: Proceedings of OOPSLA '96., CA, USA, ACM Press, pp. 342-358, 1996.

Ellis, C. A., Gibbs, S. J., and Rein, G. L., “Groupware - some Issues and experiences,” *Communications of the ACM*, vol. 34, 1, pp. 38-58, 1991.

Erwin, D., Valentine, J., and Jablonski, D., “The Origin of Animal Body Plans,” *American Scientist*, vol. 85, 2, 1997, (available at: http://www.amsci.org/amsci/articles/97articles/Erwin.html).

Henderson, A. and Kyng, M., “There's No Place Like Home: Continuing Design in Use,” in *Design At Work*, J. Greenbaum and M. Kyng, Eds. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers, 1991, pp. 219-240.

IBM, “Visual Age for Java,” , 1.0 ed, 1998.

Inverardi, P. and Wolf, A. L., “Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model,” *IEEE Transactions on Software Engineering*, vol. 21, 4, pp. 373-386, 1995.

JavaSoft, “JavaBeans 1.0 API Specification,” , Version 1.00-A ed. Mountain View, California: SUN Microsystems, 1997.

Kühme, T., Dieterich, H., Malinowski, U., and Schneider-Hufschmidt, M., “Approaches to Adaptivity in User Interface Technology: Survey and Taxonomy,” in: Proceedings of IFIP '92, pp. 225-252, 1992.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J., “Specifying Distributed Software Architectures,” in: Proceedings of 5th European Software Engineering Conference., Barcelona, LNCS 989 (Springer-Verlag), pp. 137-153, 1995.

Microsoft, “Visual Basic,” , 4.0 ed, 1996.

Myers, B. A., “Taxonomies of Visual Programming and Program Visualization,” *Journal of Visual Languages and Computing*, vol. 1, pp. 97-123, 1990.

Parnas, D. L., “On the Criteria To Be Used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, 12, pp. 1053-1058, 1972.

Stiemerling, O., “FlexiBeans Specification V 2.0,” Department of Computer Science, University of Bonn, Bonn, Working Paper, 1998, (available at: http://www.informatik.uni-bonn.de/~os/Publications/ Flexibeansv20.ps).

Stiemerling, O., “Component-Based Tailorability”, Ph. D. Thesis, *Department of Computer Science III* Bonn: University of Bonn, 2000 (upcoming).

Stiemerling, O., Rohde, M., and Wulf, V., “Integrated Organization and Technology Development - The Case of the OrgTech Project,” in: Proceedings of Concurrent Engineering '98, S. Fukuda and P. L. Chawadhry, Eds., Tokyo, Japan, ISPE, pp. 181-187, 1998.

Szyperski, C., *Component Software - Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 1998.

Trigg, R. H., “Participatory Design meets the MOP: Accountability in the design of tailorable computer systems,” in: Proceedings of 15th IRIS, G. Bjerknes, T. Bratteig, and K. Kautz, Eds., Oslo, 1992.