WS 9. The First International Workshop on Unanticipated Software Evolution

Günter Kniesel[1], Joost Noppen[2], Tom Mens[3], and Jim Buckley[4]

[1] Dept. of Computer Science III, University of Bonn, Germany,
gk@cs.uni-bonn.de - http://www.cs.uni-bonn.de/~gk/
[2] Software Engineering Lab, University of Twente, Enschede, The Netherlands,
noppen@cs.utwente.nl - http://wwwhome.cs.utwente.nl/~noppen/
[3] Programming Technology Lab, Vrije Universiteit Brussel, Belgium
tom.mens@vub.ac.be - http://prog.vub.ac.be/~tommens/
[4] University of Limerick, Castletroy, Limerick, Ireland,
jim.buckley@ul.ie - http://www.csis.ul.ie/

This workshop was dedicated to research towards better support for *unanticipated software evolution (USE)* in development tools, programming languages, component models and related runtime infrastructures. The report gives an overview of the submitted papers and summarizes the essence of discussions during plenary sessions and in working groups.

1. Introduction

Many studies of complex software systems have shown that more than 80% of the total cost of software development is devoted to software maintenance. This is mainly due to the need for software systems to evolve in the face of changing requirements. Despite the importance of software evolution, techniques and technologies that offer support for software evolution are far from ideal. In particular, *unanticipated* requirement changes are not well supported, although they account for most of the technical complications and related costs of evolving software.

By definition, *unanticipated software evolution (USE)* is not something for which one can prepare during the design of a software system. Therefore, support for such evolution in development tools, programming languages, component models and related runtime infrastructures becomes a key issue. Without it, unanticipated changes often force software engineers to perform extensive invasive modification of existing designs and code.

Correspondingly, the USE workshop addressed the issues inherent in unanticipated, incremental evolution of object-oriented and component based systems. The main goal of the workshop was to increase mutual awareness of the different research groups active in this field. By fostering exchange of ideas we wanted to promote new approaches and technologies for dealing with unanticipated software evolution.

1.1 Workshop Overview

The workshop lasted one day. In keeping with the spirit and format of a workshop, USE 2002 had a highly discursive nature, with presentations in the morning and different theme-based discussion tracks ("breakout groups") in the afternoon.

For the morning session the organizers selected presentations that covered a wide range of approaches [6, 14, 4, 21, 20, 27]. Although more than half of the slot devoted to each presentation was reserved for discussion, this time was seldom long enough. The heterogeneity of approaches and of the participants' background triggered lively debates. Occasionally, these led to general agreement but more often they demonstrated the need for stronger exchange of ideas between different communities working on different facets of unanticipated software evolution.

The paper presentations were followed by an overview of all received submissions, intended as a starting point for identifying topics for breakout groups. Starting from the question "What is Unanticipated Software Evolution?", Günter Kniesel offered a tentative answer and a classification of the various dimensions of the problem and proposed approaches.

1.2. What is USE and what is USE research?

Agreeing on a common definition of unanticipated software evolution (USE) turned out to be one of the most controversial issues. The opinion that every change is anticipated at some point in time and the opposite view that every change is inherently unanticipated where both present. This is characteristic of the heterogeneity of the different views on this topic. Among others, the discussion revolved around the questions of *what* changed, *when* it changed, whether the change depended on any *hooks* built into in the previous version of the application and from *whose point of view* the change was (un)anticipated.

There was no conclusive answer to this question. Maybe the reason is that this might be the wrong question, since different definitions are sensible from different points of view. So one could step back from asking *what is* unanticipated software evolution. Instead one could ask *why* we are concerned about unanticipated evolution and what is the aim of research on unanticipated evolution. This is less controversial: the obvious goal is to reduce the costs of evolving software without sacrificing correctness and quality.

Correspondingly, *malign changes* are those that exhibit undesired side-effects or involve high costs either because they are non-incremental or because they depend on prior encoding of hooks. In contrast, *benign changes* exhibit no undesired side-effects, are incremental and do not depend on prior encoding of hooks. Then one can simply say that the aim of research on unanticipated software evolution is to *enlarge the category of benign changes.* Put differently, *the aim of USE research is to enlarge the category of changes that exhibit* no undesired side-effects *and for which a* non-invasive[1] *implementation is possible* without encoding of proper hooks *in prior versions of the changed software.*

Although this definition was not produced during the workshop we want to put it forward as a working definition that can be further elaborated in the future. It comes close to some of the opinions expressed at the workshop so we hope it can eventually evolve into a generally accepted understanding of what USE research is about.

2. Workshop Papers

All submitted papers were reviewed by the USE program committee[2]. From the 24 accepted submissions 2 were from industrial research and development departments. All the others contributions came from academic research.

Java HotSwap API The two industrial papers address the new dynamic class replacement mechanism ("hot swap API") included in JDK 1.4 and supported by Sun Microsystems' Java HotSpot Virtual Machine:
- Pazandak [21] presents ProbeMeister, a product that uses the hot swap API to attach to multiple remotely running applications, and effortlessly insert software probes to gather information about their execution. This information can be used to effect changes within the running applications, improve their operation or recover from failures.
- Dmitriev [4] describes yet unreleased improvements of the HotSwap API for fine-grained, fast and scalable bytecode instrumentation in running server programs. As an interesting case study he shows that a JVM supporting an evolved API for dynamic bytecode instrumentation allows developers to profile dynamically selected parts of a target application much more efficiently than with other techniques.

Instance Adaptation Instance adaptation is the process of changing the layout of existing objects after their class definition has changed. This is a well-known problem in object oriented databases and manifests itself also in systems that allow update of classes at run-time.
- Duggan and Wu [6] present "adaptable objects", an approach to enable interoperation of objects whose interface and layout may be updated at runtime. It relies on lazy creation of version adapter proxies that mediate where there is a version mismatch between an object and code attempting to access that object.
- Hirschfeld, Wagner, and Gybels [11] suggest that there is much to learn from the proven techniques for unplanned dynamic evolution in Smalltalk and provide an overview of Smalltalk's mechanisms for renaming, removal, and layout change of a class.

[1] An non-invasive change is one that adds something to a software without changing any of its pre-existing parts. The terms *non-invasive*, *incremental* and *additive* are synonyms.

[2] See the USE 2002 web site at http://joint.org/use/2002/ for programm committee members and additional reviewers.

- Rashid [22] presents a comparative evaluation of schema relationships and instance adaptation of four object database systems, each representing a particular category of evolution systems. The discussion aims to demonstrate the benefits of an aspect-oriented approach in the face of unanticipated changes.

Dynamic Linking The two contributions about dynamic linking differ from all others in that they do not concentrate on enabling unanticipated changes but instead demonstrate that unanticipated changes can sometimes be undesired or have undesired side-effects:

- Drossopoulou and Eisenbach [5] point out that, in some situations, dynamic linking in Java affects program execution and the integrity of the virtual machine. In order to let programmers understand precisely these effects, the papers demonstrates the process in terms of a sequence of source language examples, in which the effects of dynamic linking are explicit.
- Eisenbach, Sadler and Jurisic [7] present a series of examples in Java and C# that pinpoint problems in the way how these two languages deal with dynamic linking. The examples show similarities and differences between the two languages in the treatment of compile-time constants, interface changes and resolution of fields (and method) access. The paper shows that neither system can guarantee the integrity of distributed applications, so developers need to be aware of the employed dynamic linking process if they want to cope with "DLL hell".

Static Linking Zenger [27] presents the programming language Keris, an extension of Java with *extensible modules* as the basic units of software. The linking of modules is inferred and allows to extend systems by replacing selected submodules with compatible versions without needing to re-link the full system. Extensibility is unanticipated, type-safe and non-invasive; i.e. the extension of a module preserves the original version and does not require access to source code.

Evolving Components and Services

- Evans et al [8] describe the evolution a distributed run-time system that itself is designed to support the evolution of the topology and implementation of an executing, distributed system. The paper presents the different versions of the architecture, discusses how each architecture addresses the problems of topological and functional evolution, explains the reasons for the evolution of the architecture and discusses lessons learned in this process.
- Sora et al [26] present a component model that supports unanticipated customization of systems in a top-down stepwise refinement manner, while composing also the structure of components. This strategy allows to realize very fine-tuned compositions even when the composition decision is made automatically, as it is necessary for self-customizable systems. The presented component composition model is able to cope with evolving requirements and discovery of new component types.
- Oriol [20] argues that application evolution at runtime is prevented by connections between different software entities and proposes a discon-

nected communication architecture based on three main concepts: extreme associative naming, late binding and asynchrony of communications. Everything is a service. Service registration and invocation occur through a semantic description. The choice of the service that best matches the description of the requested service is performed at the moment of the invocation.

- Gorinsek [9] presents novel work in the field of component-based evolution of embedded systems. It addresses the necessary properties of a system supporting dynamic updating of components in an embedded platform and proposes a new approach to design such a system.
- Buckley [2] describes work towards the development an environment where software components could autonomously adapt their interfaces to each other at runtime. Such adaptation by components would allow them adjust to unanticipated software evolution by enabling their interfaces to change in response to amendments in the service components they use and in the client components that they service.
- McGurren and Conroy [14] define the notion of a dynamic system, presents a taxonomy of dynamic systems and introduces three types of adaptation that may be used in the implementation of dynamic systems: instance replacement, service level adaptation and interface adaptation. Then the X-Adapt architecture is proposed as a design to support these three types of adaptation while combining them with integrity management.

Change Impact Analysis
- Neumann, Strembeck and Zdun [18] present an approach that fosters changes of software by managing runtime-traceable dependencies of requirement specifications and test cases to corresponding architectural elements and source code fragments. In case of (unexpected) change requests it is easy to find the affected system parts, thus facilitating timely change propagation and regression testing.
- Mens, Mens and Wermelinger [15] propose *intentional software views* as an intuitive and lightweight technique to model crosscutting concerns. This technique provides the formal basis for tools that help grouping together source-code entities that address the same concern. The technique also facilitates unanticipated software evolution by providing the ability to automatically verify the consistency of views and detect invalidation of important intentional relationships among views when the software evolves.
- Mens, Demeyer, Janssens [17] addresses the need to formalise behaviour preserving software evolution as a basis for dealing with refactoring. The paper introduces a graph representation of those aspects of the source code that should be preserved by a refactoring, and graph rewriting rules as a formal specification for the refactorings themselves. The authors show that it is feasible to reason about the effect of refactorings on object-oriented programs independently of the programming language being used, which is crucial for the next generation of refactoring tools.

Dynamic Object Models: Roles, Views, Delegation
- Markovic and Sochor [13] present an formal object model unifying wrapping, replacement

and roled-objects techniques. In this model the objects with roles can dynamically change the interfaces they support. The model comprises also other techniques used to handle evolving objects.
– Sadou [25] presents an approach unanticipated evolution of distribted objects at run time. New client programs may add behavior to existing server objects, whereas old clients may continue to use the unadapted version of the server. The approach relies on a combination of the adapter pattern and of delegation. A prototype version of the Adapter system is implemented as a set of Java libraries that does not require addition of new language constructs.
– Anderson and Drossopoulou [1] present *Delta*, a first imperative calculus for object-based languages with delegation. Such languages (eg SELF) support exploratory programming by composition of objects, sharing of attributes and modification of objects' behaviour at run-time. Furthermore delegation allows objects to delegate execution of methods to other objects. These features allow for creation of very flexible programs that can accommodate changes in requirement at a very late stage.

Aspect-Oriented Approaches David and Ledoux [3] present an approach to dynamic adaptation to changing execution conditions based on distinguishing functional and non-functional concerns. These two kinds of concerns are composed together, at run-time, by a weaver which is aware of the execution conditions so that it can adapt its weaving to their evolution.

JVM Extensions for USE Redmond and Cahill [23, 24] present the Iguana/J architecture, which supports unanticipated, non-invasive, dynamic modification in an interpreted language without extending the language, without a preprocessor, and without requiring the source code of the application.

Anticipating Requirements Noppen et al [19] talk about optimisation of software development policies for evolutionary system requirements based on a Software Evolution Analysis Model (SEAM). This is a probabilistic model for evolution requirements, which helps anticipating on future requirements.

Classification Gustavson et al [10] present a classification of dynamic software evolution built the distinction of two major facets, the technical facet and the motivation facet.

The above summaries cannot give more than a raw idea of the topics addressed in each paper. Furthermore, the classification above could be replaced by many others, depending on one's personal point of view. Therefore, interested readers are invited to consult the full papers. They can be download from the USE 2002 web site either selectively or as an archive containing the entire online proceedings [12].

3. Working Group: Scenarios

The "Scenarios" working group consisted of Misha Dmitriev, Sophia Drossopoulou, Dominic Duggan, Susan Eisenbach, Robert Hirschfeld, Jens Gustavson, Günter

Kniesel, Finbar McGurren, Yahya Mirza, Manuel Oriol and Bernard Pagurek. Its goal was to collect real-life scenarios of unanticipated software evolution from the experience of the participants.

The intent of such a collection was to serve as a basis for evaluating the expressiveness of a possible taxonomy[3] and as a set of "functional benchmarks" on which to compare different approaches to USE. Another possible use could be to identify specific requirements of certain application areas.

The group discussed the scenarios summarised in the following. For scenarios from already completed projects, the participants reported also about the approach towards USE taken in their project, the individual techniques that had been applied and the reasons why they were sufficient for that scenario. For scenarios from new or planned projects the discussion focused more on identifying the particular problems of USE that arise in that scenario and on the applicable techniques.

Network management An often encountered problem in network management is how to perform a consistent change of protocols on a server and its clients. The challenge here is to perform the change as an atomic operation while the network is still up and running. Bernard Pagurek reported on a solution of this problem based on "software hot swapping". This technique was applicable because the protocols that had to be updated were stateless, so there were no complications related to state transfer between old and new object versions. A similar example was the move from IP4 to IP6.

Mobile Devices Presently, updating the operating software on mobile devices (e.g. handhelds) requires users to turn the device in to a dealer after making himself a backup of any personal data, configurations, preferences, etc. The opposite is highly desirable: remote software updates (at the user's site), without needing a reboot, without loss of data and without loss of functionality. This is still a topic of ongoing research.

Independent Evolution of Platform and Services Another challenging scenario is independent evolution of a distributed platform and of the services running on that platform. Updates in the platform must still support services written for previous versions of the platform. New and updated services must be able to run in platforms that were not prepared for that service update. All this must happen dynamically and be transparent to the user.

Change of Law Legislation is in a permanent flux. Unanticipated changes in legislation can have a significant impact on existing products and services. For example, a new law that all hearing impaired people have to be supported by communication devices might require dynamic upgrades to the user interfaces of already delivered devices. The new user interface version would need to contain functionality that was not anticipated in its original design.

Computer Games Modern computer games provide a multitude of characters and scenarios (worlds) in which these characters interact. If new characters

[3] Development of a taxonomy was the topic of another working group.

and scenarios can be loaded dynamically into a game a "protocol discovery protocol" is needed that allows the new behaviour of new types of beings to be determined. Existing character types must learn dynamically how to deal with the newly introduced ones. For instance, an existing "dog" character will have to "learn" that it should chase a new "cat" character.

Profiling Another example was selective, dynamic profiling of applications, including library code, as reported in [4].

Bank Account Evolution A bank might conduct surveys of its customers' habits and decide to offer selected customers new types of accounts that are more attractive for that particular group. Similar unanticipated changes could be possible for credit card users. The changes of account type would need to happen dynamically, too.

This list still needs to be completed and related systematically to the different existing USE approaches and to the USE taxonomy proposed by the next working group.

4. Working Group: Taxonomy

The "Taxonomy" working group consisted of Jim Buckley, Tom Mens, Awais Rashid, Salah Sadou, Stefan Van Baelen and Matthias Zenger. Its aim was the development of a taxonomy based on *characterizing mechanisms* of software evolution and the *factors that impact upon these mechanisms.*

The goal of the taxonomy is to position concrete tools and techniques within the domain of software evolution, so that it becomes easier to compare and combine them, or to evaluate their potential use for a particular maintenance or change context.

The group proposes a categorisation along the four following dimensions:

- properties of the change mechanism,
- properties of the change itself,
- system properties and
- change process.

4.1 Properties of the change mechanism

Time of change Depending on the programming language, or the development environment being used, it is possible to distinguish between phases such as design time, compile time, load time, link time and run time. Relative to these phases one can determine four times of change: **T1** = *the time (interval) when a change is requested*; **T2** = *the time (interval) when a change is prepared*; **T3** = *the time (interval) when the change becomes available for execution*; **T4** = *the time (interval) when the change is executed for the first time.*

Parallelism Software evolution may be carried out *sequentially* or *in parallel*. With sequential software evolution, only one version of the software is available at any given time. With parallel evolution, many versions of the same system can co-exist for some time.

Within parallel evolution, one can further distinguish between *convergent* change and *divergent* change. With convergent changes, as in the example above, two parallel versions can be merged together into a new unified version. With divergent change, different versions of the system co-exist indefinitely as part of the maintenance process.

Incrementality Something that is closely related to versioning is the *incrementality* of the change. For incremental changes, an altered component can be incorporated into a system while the old (non-extended) version gets preserved. This is the basis for versioning mechanisms and it is required for systems in which old and new versions coexist simultaneously.

Degree of automation In software re-engineering, numerous attempts have been made to automate, or partially automate, software maintenance tasks. Typically, the tasks suited to automation are structural transformations of the software system.

4.2 Properties of the change

Semantic or purely structural A distinction can be made between *purely structural* and *semantic* changes. While semantic changes have an impact on the behaviour of the software, structural changes aim to preserve the semantics. In contrast to several structural changes, semantic changes are very difficult to automate.

Addition, subtraction, modification Another distinction is between *addition*, *subtraction*, and *modification* of an element in the software. These activities can be structural in that files or modules can be rationalized, added to or altered structurally. Likewise the activities can be semantic, when functionality is added, removed or altered.

Granularity of change Another distinguishing feature of change would be its *granularity*. This can go from very coarse granularity (such as system, subsystem and module level) to an extremely fine granularity (such as variable, method and statement level).

Impact of change Related to the granularity is the *impact of the change*. Sometimes, seemingly local changes may have a global impact because the change is propagated through the rest of the code.

Change effort Another related issue is the *change effort*. In many cases, changes with a high change impact also require a significant effort to make the changes. However, in some situations this can be overcome by automated tools. For example, source code renaming is typically a global change with a high change impact, but the corresponding change effort is low because renaming can proceed in an automated way.

4.3 System properties

Activeness The system can be *passive* (changes are driven externally) or *active* (the system can drive the changes itself). Typically, for a system to be active, it must contain some monitors that record external and internal state. It must also contain some logic that allows self-change based on the information received from those monitors. A system is passive if change must be driven by an external agent, typically using some sort of user interface.

Openness A software system can be *open* in that it is specifically built to allow for dynamic, unanticipated evolution. Unanticipated adaptations can be specified and incorporated at runtime. For example, a system might rely on plug-ins at runtime. While the plug-in modules may be unknown in advance, the ability to add them to the system at runtime is explicitly provided.
Closed systems are those that have their complete functionality and adaptation logic specific at build time. A system however, cannot be open to every possible change. It is likely that systems will only be open to certain (unanticipated) changes.

4.4 Change process

A final classification of software evolution can be based on the change process, which is a part of the software development process, and is typically imposed by the project manager.

Planning We can distinguish between *planned* or *unplanned* evolution, based on whether the changes that are to be applied are managed in a more or less coherent manner.

Control During or after a change, we can distinguish between *controlled* or *uncontrolled* evolution. Controlled evolution typically happens in a planned context, when the constraints in the change process are explicit and enforced. For example, versioning rules may be used to impose constraints on the evolution.

After the workshop the "Taxonomy" group continued its discussion via the USE mailing list (`use@joint.org`) with occasional contributions of other workshop participants. The results of this ongoing effort are being compiled into a technical report [16] that will provide a much more complete account of all the issues involved than is possible within the limited space available here. The summary in this section reflects a snapshot of the discussion at the beginning of September 2002.

1 Working Group: Language-Level Support for USE

This group consisted of Christopher Anderson, Misha Dmitriev, Huw Evans, Joris Gorinsek, Lubomir Markovic, Kim Mens, Ioana Sora and Barry Redmond

The discussion addressed existing challenges and emerging techniques for direct support of unanticipated evolution in languages and run-time infrastructures. It focused on the following issues:

Language annotations versus tool support Language annotations could be used to explicitly specify properties of a program, that must be known at evolution time. However, changing languages is difficult and “non-standard” languages seldom find wide-spread acceptance. Therefore, tool support to help programmers reason about evolution and about the related properties of a program might be a better alternative.

Object replacement There are many problems involved in designing a powerful and generally applicable mechanism for dynamic replacement of objects or components: when should the change happen, how and by whom is it triggered, how is it performed, and at which granularity. When moving from replacement of objects to replacement of entire component instances a sort of transaction concept is needed for ensuring atomicity of the change. All this is even harder in a resource restricted system.

Dynamic update The issue of transaction is also related to the issue of ensuring consistency of a system after a dynamic update, possibly based on some constraint mechanism. If dynamic update is based on object replacement rather than addition of new objects and delegation, than state transfer between the object versions can become difficult. On one hand, encapsulation might prevent transfer of all relevant information. On the other hand, there complex interactions can exist between multiple versions.

Dependency management Dependencies between objects need to be tracked. They can occur at many levels: invocation dependencies, structural dependencies, architectural dependencies, etc. Dependencies may need to work in both directions: if change is made to A then B must be checked and vice versa. There are more open questions here than ideas of how to address them.

Performance Evolution can be an expensive operation and building evolution support into a system can be at odds with performance. One possible way out is the approach pioneered by the dynamic language Self and successfully adopted for a commercial environment by the Java HotSpot virtual machine: The idea is to perform extensive optimizations for performance but make the optimizations undoable for evolution.

5. Working Group: USE in the Software Life Cycle

Support for USE in languages, architectures and infrastructures addresses the *late* phases of the software life-cycle (implementation and deployment). Tool support for static evolution additionally addresses the design phase. There is hardly any work about what support for unanticipated evolution might mean in *early* phases (An exception is [19].). Therefore, this group explicitly addressed USE during the entire life-cycle, including requirement elicitation and analysis. The group consisted of Pascal Costanza, Thomas Ledoux, Joost Noppen, and Uwe Zdun.

In the first place, support for USE is needed because changes to requirements and their consequences for software systems are, by definition, not known in advance. So talking about USE during requirement elicitation might sound like a contradiction and it was indeed perceived as such by other workshop participants. However, there is no contradiction, as shown in the following.

5.1 No Silver Bullet Yet

For practitioners and researchers of USE techniques it is a sad but nevertheless true fact that no single approach to USE is capable of dealing gracefully with all possible types of evolution. A particular evolution can still be outside the scope of the language in which a system is programmed, of its run-time infrastructure and of the adaptation patterns implemented in the system. Including as many evolution techniques as possible is no viable solution either, since this would increase the complexity of the system and lead to problems in performance, usability, size, cost-effectiveness, etc. In the end, there will always be changes that require extensive invasive modifications of an existing application.

5.2 Selection Models for Software Evolution Techniques

Since every technique for coping with evolution addresses a certain set of evolution problems it should only be applied in the context in which it functions best. However, choosing the right technique at analysis and design time is non-trivial. It requires analysts and designers to be aware of existing USE techniques and be able to assess their strength and weaknesses with respect to certain evolution scenarios. Experience shows that awareness and skilful use of novel techniques spreads slowly in the software engineering community. In general, the pace of innovation is faster than the pace of dissemination. Even experts in a certain domain sometimes have difficulties in keeping up with new developments.

So there is ample room for tool support in this area. Its intention is to help software engineers make an informed decision about proper evolution approaches without having to spend a significant share of their work researching and compiling the newest developments in the field of software evolution.

In order to achieve such tool support, a better understanding of the software evolution process is needed. This should be expressed by a model that can support the decisions on which evolution techniques to consider without committing to the occurrence of particular evolution scenarios. For the selection of the proper design and evolution mechanisms there is no clear-cut solution, but many different research directions can provide relevant feedback on this topic, like for instance: probabilistic models, market analysis, domain models, etc.

5.3 Summary

The essence of the discussion in this group can be summarized as follows:

– No single approach to USE is capable of dealing gracefully with all possible types of evolution. There will always be changes that will require extensive modifications of an existing system.

– Being aware of this fact, one can still take advantage of the fact that some evolution scenarios are more likely to occur than others and that some approaches to USE are better suited for certain types of evolution than others.
– So, *techniques for improved upfront analysis and design do not compete with techniques for USE but rather complement them.* One obvious way to combine both is to make analysts and designers aware of existing USE techniques and of their strength and weaknesses with respect to certain evolution scenarios. Another one, advocated here, is to devise models of evolution that already include such knowledge and help programmers make an informed decision as to which evolution support techniques are most promising for a given application.

Unanticipated evolution can invalidate an existing product, even if several preparation techniques have been included. But by making a sound choice of extension mechanisms and adaptation techniques as well as a good initial architecture design it should be possible to make software systems more resilient towards evolution.

6. Conclusions

From individual discussions with participants the organisers gained the impression that the feelings about the workshop were split. On one hand, many people appreciated the chance to meet and exchange ideas in a forum in which unanticipated software evolution was the primary topic. On the other hand, many participants were unhappy about not having enough time for discussion in breakout groups and for presentation of *all* submitted papers. Also the 10 to 12 minutes discussion per presented paper were occasionally perceived as too short. Obviously, the organizers had underestimated the high interest in unanticipated software evolution and also the particularly high need for discussion in a newly forming research community. Therefore the next USE workshop, organised in conjunction with ETAPS 2003, will last *two days* (http://joint.org/use/2003/).

The workshop showed that there is considerable work on USE in many academic and industrial research centers. The majority focuses on unanticipated run-time evolution but approaches that address all other phases of the software life cycle are present too. The surprising diversity of proposed techniques shows that there is really a strong need for forming a community of USE researchers, where groups with very different backgrounds can meet and learn from each other. Establishing a common understanding of basic notions, a catalogue of benchmark scenarios, techniques, and their classification are just the very first steps towards synergetic joint work.

7. Acknowledgements

In the first place want to thank all the participants for making this a successful workshop. Big thanks are due also to all the members of the programm committee for the time and expertise they put into their work. Many participants have expressed their gratefulness for the very detailed, and *constructive* reviewer comments they received. Last but not least, the organizers of ECOOP 2002 provided an excellent infrastructure for the USE workshop.

References

[1] Christopher Anderson and Sophia Drossopoulou. Delta - an imperative object-based calculus with delegation. In [12], 2002.

[2] Jim Buckley. Adaptive component interfaces. In [12], 2002.

[3] Pierre-Charles David and Thomas Ledoux. Dynamic adaptation of non-functional concerns. In [12], 2002.

[4] Mikhail Dmitriev. Hotswap technology application for advanced profiling. In [12], 2002.

[5] Sophia Drossopoulou and Susan Eisenbach. Manifestations of java dynamic linking - an approximate understanding at source language level. In [12], 2002.

[6] Dominic Duggan and Zhaobin Wu. Adaptable objects for dynamic updating of software libraries. In [12], 2002.

[7] Susan Eisenbach, Chris Sadler, and Vladimir Jurisic. Feeling the way through DLL Hell. In [12], 2002.

[8] Huw Evans, Malcolm Atkinson, Margaret Brown, Julie Cargill, Murray Crease, Phil Draper, Steve Gray, and Richard Thomas. The pervasiveness of evolution in GRUMPS software. In [12], 2002.

[9] Joris Gorinsek. Empres: Component-based evolution for embedded systems. In [12], 2002.

[10] Jens Gustavson and Uwe Assmann. A classification of runtime software changes. In [12], 2002.

[11] Robert Hirschfeld, Matthias Wagner, and Kris Gybels. Assisting system evolution: A Smalltalk retrospective. In [12], 2002.

[12] Günter Kniesel, Pascal Costanza, and Mikhail Dmitriev (eds). Online proceedings of USE 2002 – First International Workshop on Unanticipated Software Evolution, June 2002. http://joint.org/use/2002/sub/.

[13] Lubomir Markovic and Jiri Sochor. Object model unifying wrapping, replacement and roled-objects techniques. In [12], 2002.

[14] Finnbar McGurren and Damien Conroy. X-Adapt: An architecture for dynamic systems. In [12], 2002.

[15] Kim Mens, Tom Mens, and Michel Wermelinger. Supporting unanticipated software evolution through intentional software views. In [12], 2002.

[16] Tom Mens, Jim Buckley, and other authors pending. Towards a software evolution taxonomy. Technical report, Programming Technology Lab, Vrije Universiteit Brussel, 2002.

[17] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving software evolution. In [12], 2002.

[18] Gustaf Neumann, Mark Strembeck, and Uwe Zdun. Using runtime introspectible metadata to integrate requirement traces and design traces in software components. In [12], 2002.

[19] Joost Noppen, Bedir Tekinerdogan, Mehmet Aksit, Maurice Glandrup, and Victor Nicola. Optimising software development policies for evolutionary system requirements. In [12], 2002.

[20] Manuel Oriol. Evolution of code through asynchronous services. In [12], 2002.

[21] Paul Pazandak. ProbeMeister: Distributed runtime software instrumentation. In [12], 2002.
[22] Awais Rashid. Aspect-oriented schema evolution in object databases: A comparative case study. In [12], 2002.
[23] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Object-Oriented Programming – Proceedings of ECOOP 2002*, LNCS, 2002.
[24] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In [12], 2002.
[25] Salah Sadou and Hafedh Mili. Unanticipated evolution for distributed applications. In [12], 2002.
[26] Ioana Sora, Nico Janssens, Pierre Verbaeten, and Yolande Berbers. A component composition model to support unanticipated customization of systems. In [12], 2002.
[27] Matthias Zenger. Evolving software with extensible modules. In [12], 2002.