Non-Invasive On-Demand Changes: Applying Change-Sets as Aspects

ROBERT HIRSCHFELD [a] DAVID H. LORENZ [b]

[a] *Future Networking Lab, DoCoMo Communications Laboratories Europe*
hirschfeld@docomolab-euro.com

[b] *College of Computer & Information Science, Northeastern University*
lorenz@ccs.neu.edu

Abstract

All-invasive, in-place modifications of existing classes and methods are a popular practice in making code changes to open source systems, relying on tool support to manage change integration and maintenance. Often, a less-invasive approach is taken that treats the system as a framework and employs subclasses to express changed behavior. In this paper, we propose the application of aspects to represent code changes in a completely non-invasive manner. Squeak's change sets are an example for recording changes to a system and sharing those changes with others. Traditionally, contributions to Squeak, which include extending system classes and in-place modifications of system methods, are logged and shared via change sets. This is possible because change sets are not limited to classes as the dominant unit of modularity—their level of granularity is that of method implementations. However, change sets are insufficient in supporting dynamic system integration and maintenance, on-demand changes, and selective undo operations. The representation of system changes as aspects, as an infrastructural alternative to change sets, addresses these issues.

Key words: Change-sets, Aspect-Oriented Programming, AspectS, Smalltalk
PACS:

1 Introduction

There are two popular approaches in Smalltalk [1–3] to carry out changes in the system. The first approach, and also the most popular amongst Smalltalk practitioners, can be characterized as all-invasive—the direct in-place modification of existing classes and methods. The second, less-intrusive approach employs inheritance [4] wherever possible, using subclasses to express behavioral differences.

In-place modifications lead to less code and easier-to-understand systems, but may require more tedious work during system integration. Often it requires integrating the changes with an evolving code base maintained by someone else. In other cases, in-place modification may not be an option at all. Furthermore, directly altering third-party code might be prohibited by law or by the release license of code artifacts. Meanwhile, relying on subclassing typically results in more code to add and change, and with that, more code to maintain.

The two approaches offer a trade-off between flexibility and modularity. Subclassing is preferable, but it is also too restrictive. It constrains the modifications to be within the modular structure of the base system. Difficulties in system maintenance can be attributed to the class as the main modularity construct in object-oriented programs and systems. In-place modifications provide maximum flexibility, allowing changes to cut across classes and methods, but offer no linguistic constructs.

A balance is normally found in the programming environment in the form of extra-lingual tools and mechanisms for managing the changes, namely change sets. In Squeak [5–9], change sets keep track of in-place modifications made to the system. Change sets have always been and still are the most fundamental mechanism for tracking, storing, or deploying changes made to the Squeak system. Every project is associated with a change set. Changes can be written to a file as interpreter directives in source code form, and this source code representation can be used to re-apply these changes to the same or another system. Projects in Squeak not only store the state of the complete desktop, including all visible display objects, but also reference the currently active change set so that all changes done to the system in the context of that particular project get recorded into this change set [9].

The use of change sets allows work to be shared by providing information about modifications to be done to a system to achieve desired behavior. Such modifications include changes to an existing method, the addition or removal of a method, the change of message category names, the change of class definitions, the change of class comments, the addition or removal of classes, and the change of class category names [2]. One can think of a change set as a function on a program: applying such function to a program results in an updated program. The same can be said of aspects.

1.1 Contribution

In this paper we propose to elevate change sets to be first class aspects. Often, an aspect is viewed as a source code transformation [10]. In this work we explore the other direction: Could also source code revisions be viewed and treated as aspects?

Traditionally, revision control tools for manipulating and managing source code such as SCCS [11], RCS [12], and CVS provide fine-grained single-lines of code tracking of modifications, but they also treat the program as a mere collection of text lines. While there is support for merging versions and for rolling back to an earlier version, changes cannot be invoked or revoked out of order or dynamically at runtime.

Change sets have this limitation, too. While multiple change sets can be applied to a system, they have to be applied in a certain order, they affect the whole system, and they can only be rolled-back—if at all possible—in the reverse order applied originally. However, with the availability of AspectS [13,14], a framework for *aspect-oriented programming* (AOP [15]) in Squeak, aspects become an alternative to change sets, which offer a totally non-invasive approach to carrying out changes.

Changes are crosscutting by nature, the essential phenomenon addressed by *aspect-oriented software development* (AOSD). We present the application of aspects to represent change sets to allow dynamic, on-demand changes that are non-invasive, context-dependent, and allow for selective undo operations and better system integration and maintenance. We illustrate our approach by means of AspectS and its application to the augmentation of the Squeak development environment [16,17].

1.2 Outline

The paper shows how to use aspects to implement a revision-control-like functionality in Squeak. In Section 2, we describe the application of AOP to tracking changes. We give a rationale for capturing changes as aspects, and we show how particular changes can be expressed as an aspect. In Section 3, we describe our experience in using aspects to migrate the changes from basic Squeak to AspectS. We provide a precise description and discussion of the schemes used to encode changes, and in Section 4 we give an example of a particular transformation. Finally, Section 5 summarizes and concludes the paper with an outlook at on-going work.

2 Changes as Aspects

Change sets can be described as a unit of code deployment where changes neither contain nor are limited to a particular class, but affect several classes and methods therein. Since change sets usually combine changes related to one particular task or subject, change sets embody crosscutting changes. Because of that, it is natural to represent such changes in an aspect-oriented fashion, localizing changes within aspects.

This approach is aligned with software configuration management approaches that employ AOSD techniques, for example the Sheets system [18], which is a hypercode programming environment and part of the Gwydion project [19]. The Coven system [20] also supports fragment-based versioning, however, the fragments are stored as text. In comparison, we advocate the storage of aspects as semantic change-sets.

Our observations are based on actual experience while extending Squeak's code browsers to allow developers to become aware of system parts affected by aspects introduced via AspectS, and to traverse structural relationships between aspects and system parts affected by aspects [16,17]. AspectS is an approach to general-purpose AOP in the Squeak/Smalltalk environment. Based on concepts of AspectJ [21,22] it extends the Smalltalk metaobject protocol to accommodate the aspect modularity mechanism. In contrast to systems like AspectJ, weaving and unweaving in AspectS happens dynamically at runtime, on-demand, employing metaobject composition. In addition to that,AspectS also supports the unweaving of individual aspects dynamically at runtime. Instead of introducing new language constructs, AspectS utilizes Smalltalk itself as its pointcut language. AspectS benefits from the expressiveness of Smalltalk, its elegance and power.

2.1 AspectS Case Study

In our first attempt to augment Squeak's development environment to add AspectS-related functionality, we decided not to apply in-place modifications since Squeak is supposed to be a fast changing platform, which made in-place modifications not that attractive. We subclassed all classes to be changed and provided differential behavior there instead. Due to numerous unnecessary hard-coded direct class references, we had to override many more methods to readjust those references to the subclasses of such classes provided by us, and to put mechanisms into the system that allowed us to actually integrate our code.

After finishing that part of our system extension, we decided to move all our changes into an aspect that could be installed and uninstalled on-demand, evoking or revoking AspectS related browser properties when needed. The resulting aspect allowed us to keep our changes localized in one single module, to make them appear almost like in-place modifications, and to activate or deactivate our changes if necessary.

In the following sections, we will describe how particular changes—the addition, removal, or modification of a method—can be transformed into and expressed by an aspect. In an example we show changes before and after their transformation using AspectS.

- *Added Methods.* Expressing added methods by an aspect is straight-forward: Most aspect-oriented languages offer an introduction clause which qualifies perfectly for adding new methods into a system.
- *Removed Methods.* Removing methods is almost as simple as adding them. However, just rendering removed methods silent, that is changing them to do nothing at all, is not sufficient since this will not address situations in which the removed method was overriding another method defined in a superclass. To avoid such erroneous behavior, removing a method must be done via the application of an around advice that shadows the original method and forwards the message received to '`super`.'
- *Changed Methods.* Changes of an existing method can be matched to before, after, or around advice directives. A before or an after advice are perfect candidates for additional behavior that has to be activated on entrance into or exit from a method execution. An around advice, which is more general than both a before and an after advice, was used to extend more complicated methods where it was not that obvious how to just change the entrance or the exit of a method invocation. An around advice allows to completely render original behavior inactive by shadowing the original method and providing a completely rewritten new method.

3 Coding Styles and Transformations

We gained our insights and experience while converting regular system extensions and modifications to the Squeak image into aspects using AspectS. With that, our guidelines about how to perform these conversions are influenced by the actual implementation of AspectS, its strengths and limitations.

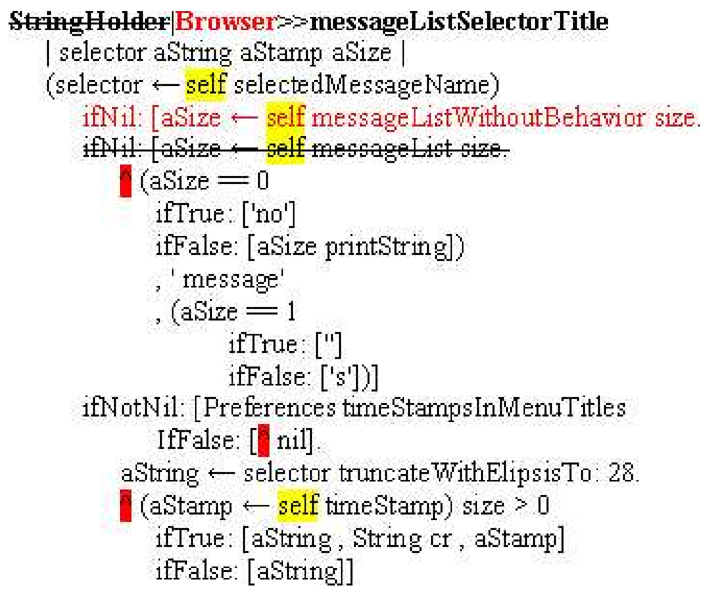
A primary design principle in the development of AspectS was to introduce the aspect modularity construct without changing Squeak—neither the language nor the basic development environment. Since aspects in AspectS are to be dynamic and late bound to the image, AspectS makes use of block objects to represent advice code. Blocks are objects often used in control structures of the Squeak system, representing a deferred sequence of actions. Blocks are not executed when defined, but at a later time when requested [1]. Such block objects require special care when accessing the receiver of message affected by aspects, the access to its state, or the returning of objects in general. In the following, we will go into some detail to further illustrate these situations.

The code for messages that are to be sent to Squeak's pseudo-variable '`self`' in the context of the receiver needs to be changed by replacing all occurrences of '`self`' with the name of the first argument of an advice block denoting the actual receiver of the affected message, which is commonly named '`receiver`' in AspectS.

- *Super Sends.* The code for messages that are to be sent to Squeak's pseudo-variable '`super`' in the context of the receiver needs to be changed, too. Compared to '`self`'-sends, '`super`'-sends need more work to be adjusted in advice code blocks. Here, we need to utilize Squeak's reflection protocol to explicitly start the method lookup in the superclass of the actual receiver of the affected method. We can achieve this by means of '`Object>>perform:withArguments:inSuperclass:`' which, as indicated above, works just like '`Object>>perform:withArguments:`' except that message look up does not begin within the class of the receiver of the message, but with the supplied superclass instead [9].
- *Direct Variable Access.* Since block objects of advice directives are defined outside of the method(s) to be affected, it is not possible to access state held by an associated object or its class directly. Object or class state can only be accessed either through reflection, which is not recommended in regular advice code, or via standard accessor methods. If accessor methods are not already available, they need to be introduced by the aspect to gain state access.
- *Single Exits.* Blocks in Squeak are in principle similar to continuations. Besides some deficiencies of their current implementation, one of their properties is that if they contain an explicit return, the flow of control does not continue after where the block was activated, but where the block was defined. Because of that, explicit returns via return statements need to be eliminated. As result, there will be no explicit return in advice blocks, leaving the block with one exit point, as in Nassi-Shneiderman diagrams [23] or flow charts.

4 A Transformation Example

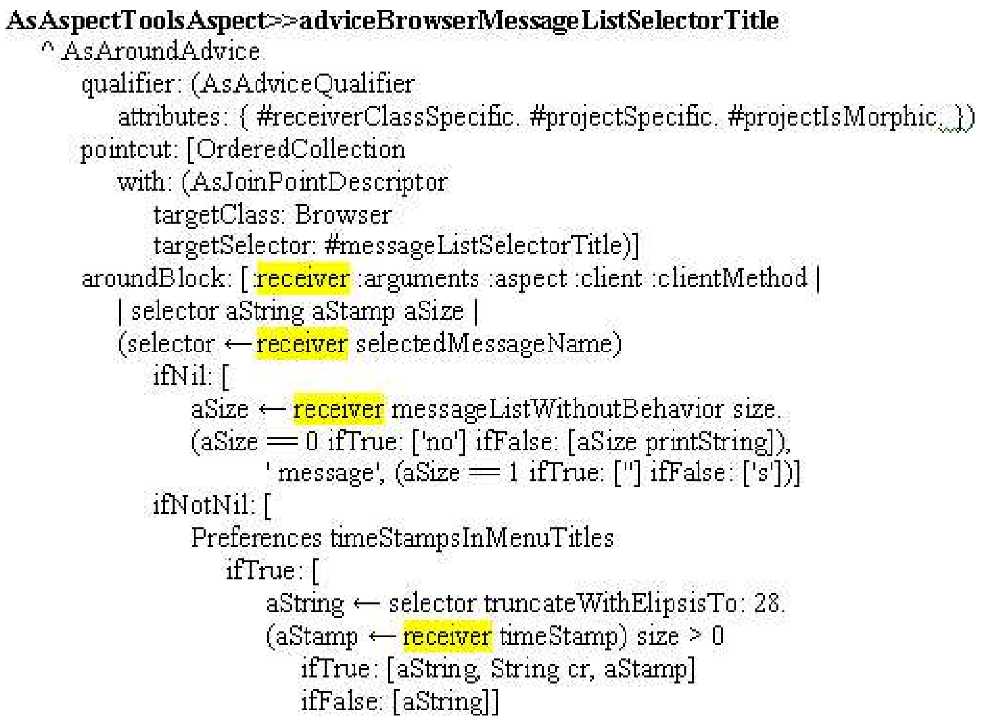
The following code (Figure 1 and Figure 2) illustrates how regular changes to Squeak can be represent as advice method of an aspects. '`StringHolder`,' a superclass of '`Browser`' implements '`messageListSelectorTitle`,' which is one of the methods that needed to be changed to allow aspect navigation for AspectS. Figure 1 shows how this change was achieved by overriding this method in '`Browser`.' The part of the code that needed to be changed from '`StringHolder`' is rendered '`strikethrough`,' and the part of the code that got actually changed in ‘`Browser`’ is printed in ‘red.’



```
~~StringHolder~~|Browser>>messageListSelectorTitle
    | selector aString aStamp aSize |
    (selector ← self selectedMessageName)
        ifNil: [aSize ← self messageListWithoutBehavior size.
        ~~ifNil: [aSize ← self messageList size.~~
            ^ (aSize == 0
                ifTrue: ['no']
                ifFalse: [aSize printString])
                , ' message'
                , (aSize == 1
                        ifTrue: ['']
                        ifFalse: ['s'])]
        ifNotNil: [Preferences timeStampsInMenuTitles
                IfFalse: [^ nil].
            aString ← selector truncateWithElipsisTo: 28.
            ^ (aStamp ← self timeStamp) size > 0
                ifTrue: [aString , String cr , aStamp]
                ifFalse: [aString]]
```

Fig. 1. Required changes

Figure 2 is the representation of the same change as an AspectS advice. To make it easier to compare with the original method, removed explicit returns were highlighted in ‘red,’ and all ‘`self`’-references that were changed to ‘`receiver`’ were highlighted in ‘yellow.’



```
AsAspectToolsAspect>>adviceBrowserMessageListSelectorTitle
    ^ AsAroundAdvice
        qualifier: (AsAdviceQualifier
            attributes: { #receiverClassSpecific. #projectSpecific. #projectIsMorphic. })
        pointcut: [OrderedCollection
            with: (AsJoinPointDescriptor
                targetClass: Browser
                targetSelector: #messageListSelectorTitle)]
        aroundBlock: [:receiver :arguments :aspect :client :clientMethod |
            | selector aString aStamp aSize |
            (selector ← receiver selectedMessageName)
                ifNil: [
                    aSize ← receiver messageListWithoutBehavior size.
                    (aSize == 0 ifTrue: ['no'] ifFalse: [aSize printString]),
                        ' message', (aSize == 1 ifTrue: [''] ifFalse: ['s'])]
                ifNotNil: [
                    Preferences timeStampsInMenuTitles
                        ifTrue: [
                            aString ← selector truncateWithElipsisTo: 28.
                            (aStamp ← receiver timeStamp) size > 0
                                ifTrue: [aString, String cr, aStamp]
                                ifFalse: [aString]]
```

Fig. 2. Changes as as AspectS advice

This transformation is simple enough to be done automatically. The change set is actually much easier to understand than its corresponding AspectS advice. However, the advice is generated, and, more importantly, it is an advice and part of an aspect with all the benefit listed earlier.

5 Conclusion and Future Work

In this paper, we propose the application of aspects to represent code changes in a manner non-invasive to the base system, which allows changes to be done or undone on-demand. The paper suggests a solution for encapsulating changes, which embody crosscutting modifications. The technique is derived from experience in performing changes to Squeak to support AspectS, using AspectS' aspects reflexively to implement such support.

Squeak's change sets are an example for recording changes to a system and sharing those changes with others. Traditionally, contributions to Squeak, which include all-invasive in-place modifications and less-invasive extensions to system classes and methods, are logged and shared via change sets. This is possible because change sets are not limited to classes as the dominant unit of modularity—their level of granularity is that of method implementations. However, change sets are insufficient in supporting dynamic system integration and maintenance, on-demand changes, and selective undo operations. The representation of system changes as aspects in AspectS, as an infrastructural alternative to change sets, addresses these issues.

Instead of requiring developers to express their changes as aspects, we would assume infrastructure or framework support to support the generation from regular changes into aspects and back. This should not be an issue since all transformations involved are mostly trivial.

While AspectS encourages selective do and undo operations of aspectualized change sets, it does not yet help in describing and enforcing dependencies, conflict detection and conflict resolution. Here, more work needs to be done to address aspect and change management adequately. Since AspectS provides another composition mechanism (that of aspects and advice behavior) on top of Smalltalk and with that adds runtime overhead for composition management, we suggest to phase-in selected aspects into the base systems once they stabilize, using Smalltalk's base composition mechanism (that of classes and methods).

The feasibility to express certain modifications as aspects raises interesting question for further research. How would such an encapsulation of program changes affect evolution of the code base? Often changes build upon each other. Would such a representation hinder or assist in changes to the aspectizedencapsulated changes? Changes also interact with each other, or with other concerns that crosscut the system. How does this change encoding facilitate or hinder the governance of interactions with either crosscutting concerns or other aspectized changes? How would this approach scale up to many layers of change-sets and how would it affect the software development cycle, e.g., debugging? AspectS is a proof that at least a complex modification like those required to implement AspectS can be done without a significant performance degrade. More work and experiment are needed to asses whether the approach will be useful in general, e.g., perhaps minor modification need to be collected and aggregated before issuing an aspect, and perhaps large modifications need to be split to several aspects?

A change set can be viewed as a sequence of program transformations that can be applied to an image to obtain a new image. We have argued that AOP can make the application of these transformations undo-able and non-intrusive. While this has not been formally proved, we have illustrated the potential of aspects to model change sets. Our experience with change sets as aspect can provide insights which might be used by subsequent language developers attempting to provide similar encoding for crosscutting changes.

On-demand changes via aspects are implementation candidates for PerspectiveS [17,24]. PerspectiveS provides multiple layers of context-dependent behavior which will be activated or deactivated by context assessors. With PerspectiveS there is no need to specifically prepare the base system that is to be decorated with context-dependent behavior. PerspectiveS was inspired by PIE (Personal Information Environment, [25–27]). In contrast to PIE where such alternatives were offered to developers during development-time, PerspectiveS permits them to concurrently exist in a deployed system at runtime. Behavior alternatives are activated or deactivated depending on the computational context accessible directly or indirectly via context assessors.

Acknowledgment

We thank Matthias Wagner for his fruitful discussions and his comments.

References

[1] A. Goldberg, D. Robson, Smalltalk-80: the language and its implementation, Addison-Wesley, 1983.

[2] A. Goldberg, SMALLTALK-80: the interactive programming environment, Addison-Wesley, 1984.

[3] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison-Wesley, 1989.

[4] G. Bracha, W. Cook, Mixin-based inheritance, in: Proceedings of the European Conference on Cbject-Criented Programming Systems, Languages, and Applications (OOPSLA/ECOOP), Vol. 25 of ACM SIGPLAN Notices, 1990.

[5] M. Guzdial, Squeak: Object-Oriented Design with Multimedia Applications, Prentice Hall, 2000.

[6] M. Guzdial, K. Rose, Squeak: Open Personal Computing and Multimedia, Prentice Hall, 2001.

[7] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, a practical Smalltalk written in itself, in: Proceedings of the 12^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'97, ACM SIGPLAN Notices 32(10) Oct. 1997, Atlanta, Georgia, 1997, pp. 318–326.

[8] G. Korienek, T. Wrensch, D. Dechow, Squeak - A Quick Trip to ObjectLand, Addison-Wesley, 2001.

[9] Squeak homepage, `http://www.squeak.org`.

[10] M. Katara, S. Katz, Architectural views of aspects, in: Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, 2003, pp. 1–10.

[11] M. Rochkind, The source code control system, in: IEEE Transactions on Software Engineering, Vol. 1, 1979, pp. 364–370.

[12] W. Tichy, Rcs - a system for version control, Software, Practice & Experience 22 (8) (1992) 637–657.

[13] Aspects homepage, `http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/`.

[14] R. Hirschfeld, AspectS—aspect-oriented programming with Squeak, in: M. Aksit, M. Mezini, R. Unland (Eds.), Architectures, Services, and Applications for a Networked World, no. 2591 in Lecture Notes in Computer Science, Springer Verlag, 2003.

[15] R. Filman, D. Friedman, Aspect-oriented programming is quantification and obliviousness, in: Proceedings of the Workshop on Advanced Separation of Concerns at the 15th European Conference on Object-Oriented Programming (ECOOP), Budapest, 2001.

[16] R. Hirschfeld, M. Wagner, Metalevel tool support in AspectS, in: Proceedings of the OOPSLA 2002 Workshop Tools for Aspect-Oriented Software Development, Seattle, Washington, 2002.

[17] R. Hirschfeld, M. Wagner, PerspectiveS – AspectS with context, in: Proceedings of the OOPSLA 2002 Workshop on Engineering Context-Aware Object-Oriented Systems and Environments, Seattle, Washington, 2002.

[18] Sheets homepage, `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/gwydion/docs/htdocs/gwyd%ion/Sheets/`.

[19] Gwydion, Gwydion homepage, `http://www-2.cs.cmu.edu/afs/cs/project/gwydion/docs/htdocs/gwydion/`.

[20] M. Chu-Carroll, S. Sprenkle, Software configuration management as a mechanism for multidimensional separation of concerns, in: Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, ICSE 2000, 2000.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: J. L. Knudsen (Ed.), Proceedings of the 15^{th} European Conference on Object-Oriented Programming, no. 2072 in Lecture Notes in Computer Science, ECOOP 2001, Springer Verlag, Budapest, Hungary, 2001, pp. 327–353.

[22] C. V. Lopes, G. Kiczales, Recent developments in AspectJ, in: S. Demeyer, J. Bosch (Eds.), Object-Oriented Technology. ECOOP'98 Workshop Reader, no. 1543 in Lecture Notes in Computer Science, Workshop Proceedings, Brussels, Belgium, Springer Verlag, 1998, pp. 398–401.

[23] I. Nassi, B. Shneiderman, Flowchart techniques for structured programming, in: ACM SIGPLAN Notices, Vol. 8, 1993, pp. 12–26.

[24] Perspectives homepage, `http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/PerspectiveS/`.

[25] D. Bobrow, I. Goldstein, Representing design alternatives, in: Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior (AISB), Amsterdam, 1980.

[26] D. G. Bobrow, I. P. Goldstein, A layered approach to software design, Tech. Rep. CSL-80-5 (Dec. 1980).

[27] D. G. Bobrow, I. P. Goldstein, An experimental description-based programming environment: Four reports, Tech. Rep. CSL-81-3 (Mar. 1981).