VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Towards Asset Based Software Engineering

Robert Biddle and Ewan Tempero

Technical Report CS-TR-99/1
January 1999

Abstract

We believe that current programming environments are not well designed to support reuse. In this paper we propose that to better support reuse, programming environments should focus on existing code assets, and so foster an asset-based approach to software development. We outline some ways this approach has worked in our exploration with prototype programming tools.

Keywords: Reusability, Tool support, Programming Environments

Workshop Goals: Explore tool support for reuse

Working Groups: Tool support for reuse

Publishing Information

This was presented at the *Ninth Workshop on Institutionalizing Software Reuse*, held in Austin, Texas, U.S.A in January 1999.

1 Background

Two important strategies to improve programmer productivity are *reuse* to reduce the programmer's work, and the development of *tools* to aid the programmer. Many factors are involved, and the impact of non-technical issues can be significant. Even so, we believe a key factor to achieve success with reuse, and success with tool support, is that they work together.

Our approach has been to first understand how programming languages support reuse. We have studied specific languages, and developed a conceptual model to document the general principles involved [BT96]. We are now developing programming tools designed to explicitly support reuse. There are tools such as code repository systems that already directly support reuse, but we believe *every* kind of programming tool has a role to play in facilitating reuse. We are investigating what kind of support is necessary, how tools could provide such support, and are developing prototype tools to explore these ideas.

We believe that the next step is to consider all the steps of the programming process, and examine how to support reuse at each step. However, as we study existing programming environments, we become dissatisfied that their structure is appropriate to properly support reuse.

2 Position

Our position is that to support reuse, programming environments should focus on existing code assets, and so foster an asset-based approach to software development.

Historically, the primary elements of programming environments are the tools involved in processing program code: editor, compiler, linker, debugger, and so on. We believe that this orientation can and should be changed to promote software development by supporting reuse.

In the field of Human-Computer Interaction, an important technique in the design of user interfaces is "task analysis" (see chapter 7 of [DFAB93]). The key principle of this technique is that study of user tasks and their structure should determine the design of the interface. In this analysis, the designer must take care to distinguish the essential user tasks from steps the user follows to accomplish the essential tasks using existing software.

The idea of design by task analysis is that the user interface should, as much as possible, support the essential user task structure, and that this will maximize productivity. For example, task analysis might suggest that word processors should be designed to focus on the document, rather than on formatting commands.

We suggest that this approach gives insight into how a programming environment would best support reuse. There are two steps involved. Firstly, the environment should focus on code, not on the functions to process code. Secondly, the environment should be explicitly oriented to highlight existing code assets. In this way, we hope to improve productivity by accentuating the value of code assets, and the role of existing assets in producing new software. We call this approach asset-based programming.

On reflection, we see the asset-based approach has many parallels in other fields. When we build a house, or when we make dinner, we typically focus on entities, not processes. We create entities, and we create them from other component entities. The processes are important, but we focus on the entities. Moreover, in most situations, our design work begins from the basis of the entities we have available. When we design a house, we take into strong consideration what building materials are available. When we plan dinner, we check to see what ingredients we have available. Of course, this is not always the case, but there are strong reasons of economy and reliability for working with assets that are available.

Our immediate concern is the design of a program development environment, and how this design can support reuse. However, the issues go well beyond source code, and the real underlying problem involves wider concepts in software development and the implications of software reuse. While our present work will involve the asset-based approach to programming environments, we believe that a significant opportunity is to explore the wider idea of asset-based software engineering.

3 Approach

Our recent work has been the development and exploration of prototype tools to support reuse in program development. In this work, we have become increasingly aware of the importance of the asset-based approach. To give some idea of how this approach affects tool design, we describe here two important affects the approach has had on our work.

Direct Manipulation

One of our prototype tools addresses static program structure, and so is similar to a CASE tool featuring a class browser. However, we have adapted the diagrammatic notation to highlight the interplay of reuse of both component and context, key ideas from our conceptual model. We also tried to simplify interaction with the tool to reduce distraction, and so instead of allowing the user to control the layout of the diagram, our tool did the layout automatically to highlight reuse. But we saw signs of a deeper effect as well.

As using the tools became simpler, the barriers between program and tool began to diminish: the diagrams seemed to *become* the programs, and the structure of the diagrams began to influence our thinking about the programs. The experience was a sensation similar to that in interaction Shneiderman calls "direct manipulation", and he argues there is a strong relationship between direct manipulation and learning (see his discussion in section 6.3 of [Shn98]).

The nature of direct manipulation involves a focus on existing entities, allowing the user to work with them to accomplish tasks. In our case, the visual entities represented existing software components, and the user works with them in creating new designs. Accordingly, we see a strong connection between direct manipulation and an asset-based approach to tool design.

Visualisation as Documentation

It is often difficult to know how to use library code because it is not clear what the code does, or how it really works. Explaining what code does and how it works is a job for documentation. However, textual documentation is often very hard to understand. As an alternative, we have been exploring documentation provided by program visualisation, where animated diagrams show pictorially how the code works.

Such visualisation can be interactive, and allow the user to explore use of the code in order to determine whether it is suitable for their purpose, and how best to use it. We have been exploring this idea by creating visualisations as documentation, and attempting to determine whether they really do help in practice. For example, the Java "GridBag" class is important for GUI layout control. However, it's very hard to understand all the parameters involved, even with the textual documentation. We developed a interactive visualisation of the GridBag that allows programmers to more quickly determine how to use the GridBag to obtain the layout they wish.

While the GridBag already has a visual aspect to its behaviour, this is not necessary for the principle to apply, because we can create visualisations of conceptual behaviour. To explore this, we are experimenting with using techniques of algorithm animation to document data structure program components.

In a development process that is oriented around existing assets, documentation becomes of even greater importance, and so any approach that can make documentation more effective is of great value. Moreover, a documentation visualisation provides an explicit visual focus on an asset. In this way the approach highlights the asset, and supports the programmer in considering how to work with it.

4 Comparison

Many tools have been developed to support programmer productivity, and many of these do fit into software engineering processes designed to support reuse, as McClure describes [McC97]. However, our approach is to consider how such tools, and tool frameworks, should be explicitly designed to support reuse.

Asset-based programming might be seen as a generalisation of the "commercial off the shelf" (COTS) software approach [DV97] , and so many of the issues involved in COTS also apply to asset-based programming. One distinction between the two is that asset-based programming would have a greater influence on system requirements, and consider all assets that may be of relevance in any decision making process.

A consequence of this difference is that asset-based programming might result in an organisation not following a COTS approach. For example, the implementation language chosen will effect which commercial products are applicable, and choice of implementation language will be strongly influenced by existing expertise and tools within the organisation. Deciding which new products to develop will be more strongly influenced by previously developed products within the organisation, and may rule out the use of COTS. Moreover, the asset-based approach would regard assets as starting points, and so would encompass changing or transforming assets where desirable.

More generally, we do not claim that the asset-based approach negates or abandons previous work in reuse or software engineering. Rather, we suggest that an asset-based viewpoint will show many disparate ideas to be parts of a coherent structure for improving software development.

References

[BT96] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*, pages 52–61. IEEE Computer Society Press, April 1996. Also available as Technical Report CS-TR-95/17.

[DFAB93] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction.* Prentice Hall, 1993.

[DV97] John Dean and Mark Vigder. System implementation using commercial off-the-shelf software. In *Proceedings of the 1997 Software Technology Conference*, Salt Lake City, Utah, April 1997.

[McC97] Carma McClure. *Software Reuse Techniques.* Prentice Hall, 1997.

[Shn98] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison Wesley Longman, third edition, 1998.

5 Biographies

Robert Biddle is a faculty member in the Computer Science department of Victoria University of Wellington, New Zealand. He is interested in research on software reusability, programming visualisation, and computer science education. He teaches courses on object-oriented programming, and human-computer interaction. He received a Ph.D. in Computer Science in 1987 from the University of Canterbury; previously he received B.Math and M.Math degrees from the University of Waterloo.

Ewan Tempero is a faculty member in the Computer Science department of Victoria University of Wellington, New Zealand. His main research interest is examining how programming languages features affect the production of code. His current focus is the impact programming languages, particularly object-oriented languages, have on the reusability of the resulting code. He is also interested in various aspects of distributed systems. He received a Ph.D. in Computer Science from the University of Washington in 1990.