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

Programming Environments for the Beyond the 90s

Robert Biddle and Ewan Tempero

Technical Report CS-TR-99/2
January 1999

Abstract

We believe that current programming environments do not provide enough support for programmers. We discuss our approach to designing and developing new tools and environments. We argue that any such design effort should be based on principles that relate to the concept the environments are meant to support. We advocate developing an environment that directly support the creation and reuse of reusable code.

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

Many people regard the only way to get the benefits from reuse is through *systematic reuse programs*. One aspect of such programs is the existence of repositories of reusable code — as Tracz says, "In order to reuse software, there needs to be software to reuse" [Tra95]. One possible inhibitor to acceptance of reuse programs is the cost of the creation and maintenance of repositories of reusable code. Studies have shown that the cost of developing reusable code is typically two times the cost of developing code for single use. Our interest is in reducing the cost of creating reusable code.

2 Position

Our position is that programming environments have not progressed much since the invention of the full-screen editor. Even current so-called "integrated development environments" (IDE) only provide a small amount of assistance to a programmer. Anyone who writes code is constantly aware of the wasted effort that it requires. Simple tasks must be frequently repeated (create a new `for` loop, use the help to remember the key-binding for the `if` template, look at the documentation for the `Layout` component, check the `Layout` documentation again to make sure the argument types are correct, look at a colleague's use of the `AccountsPayable` framework to see how it is done). All are tasks that could be automated, but typically aren't. Instead IDE's provide tools such as hierarchical browsers (that aren't used) and on-line documentation (that either isn't complete or too difficult to use to easily find what is needed). It is not that this kind of support is bad, it's that it is solving problems that aren't so important while leaving more important problems unsolved. We believe the reason that such environments exist is that they have been developed without a clear idea of what support they are supposed to be providing.

We believe that programmer productivity can be significantly improved if a *principled* approach is taken to design programming environments. By this we mean that environment designers, as well as deciding what problems to solve, should also understand *why* those problems should be solved. Just providing solutions that look useful (such as hierarchy browsers) is wasted effort if they aren't used.

Understanding what problems to solve begins by deciding what fundamental concept the environment will support. Different choices may result in quite different looking environments. In our case, we advocate developing an environment that directly support the creation and reuse of reusable code.

3 Approach and Comparison

In order to support the creation of reusable code, we must first understand what it means for code to be reusable. Our previous work has involved creating a model of how programming languages affect the reusability of code [BT96]. Our model focuses on units of reuse, which we call *assemblies*. An assembly is anything that may ultimately become valid source code. Our model describes various aspects of assemblies that are important to our ability to reuse the assemblies. Our model gives the foundations for our investigation into tool support.

Our model tells us what aspects of assemblies to focus on in order to make the assemblies more reusable. We have developed prototype tools that allow their users to explore and analyse those aspects. This is providing support for developing *for* reuse. However, there is more to reuse than just developing reusable code, there is also the process of reusing it, or developing *with* reuse.

Developing *with* reuse means using existing assemblies when creating new ones. The technical factors associated with developing with reuse can be broken down as follows (adapted from Mili et al. [MMM95]). If something is to be used, then it first must be *identified*, thus the cost of the *search* process affects whether or not reuse takes place. The cost of searching includes the cost of formulating the *query* and the *accuracy* of the result, and there is usually a tradeoff between the two. Another part of the identification process is *verifying* that the assembly is acceptable. Once an acceptable assembly has been identified, it must then be *retrieved*. When a potential assembly has been identified and retrieved, it must then be *adapted* to the new environment.

We expect tools to enhance our ability to perform tasks, and we can use the analysis above to identify tasks associated with code reuse that may profit from tool support. There are potentially many tools, even counting only tools with direct support for reuse. McClure [McC97] asserts that there are five types of tools that support the creation and use of a reuse library: repositories, classification tools, browsers, configuration management tools, and cataloguing tools. It should be noted however, that this classification does not cover tools that may be useful for developing for reuse, such as tools for designing, analysing, and testing assemblies.

The prototype tools mentioned earlier have allowed us to explore a reuse focus in static code browsing, and in tracking program execution. We now believe that *all* steps in the programming process can be assisted significantly with tool support for reuse. This could lead to an entire program development environment that supports reuse.

To explore this idea, we propose the following framework for considering the issues. We suggest focusing on the traditional detail steps in the programming process: code production, project configuration, compile-time, and run-time. At each step, we should consider programming *for* reuse (with the points from our model), programming *with* reuse (with the points adapted from Mili et al.), and then consider the kinds of tools that might provide assistance (inspired by the kinds of tools suggested by McClure).

Code Production	text editor repository
Project Configuration	dependency checker version control system
Compile-Time	class browser cross referencer
Run-Time	debugger profiler animation system

The table above itemises some of the many tools that have been developed over the years to assist in programming, and we do not mean to discount their contribution. However, we do feel that our framework will provide a principled way to assess their support for reuse. Moreover, by focusing on goals, the framework should also assist us to detect gaps in tool support, and to consider alternatives.

There are many kinds of ways it may be possible to improve support for reuse. For example, one approach might be to examine how existing tools already support reuse, and concentrate on better tailoring them to the task. Existing tools for production of code are typified by text editors, for actual entry and detail modification of code. However, we might consider especially their role in adapting code for reuse, and their role in supporting detail reuse via copy-and-paste.

A critical factor in the design of any tool is simply usability. In particular, Iivari reports that one key reason why CASE tools are generally less successful than had been hoped is the usability of the tools themselves [Iiv96]. So where tools do support reuse, it is also important that their usability be sufficiently good that programmers actually benefit from the support.

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.

[Iiv96] Juhani Iivari. Why are CASE tools not used? *Communications of the ACM*, 39(10):94–103, October 1996.

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

[MMM95] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[Tra95] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse.* Addison-Wesley, 1995.

4 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.