In Proceedings of the 1st International Workshop on the State of the Art in Automated Software Engineering, Irvine, CA, 27th June 2002.

# Recent Experiences with Code Generation and Task Automation Agents in Software Tools

John Grundy[1,2] and John Hosking[2]

Department of Electrical and Electronic Engineering[1] and Department Computer Science[2]
University of Auckland, Private Bag 92019, Auckland, New Zealand
{john-g, john}@cs.auckland.ac.nz

## 1. Introduction

As software grows in complexity, software processes become more flexible yet complex, and more developers must co-operate and co-ordinate their work, software tools providing developers editing, reviewing and management facilities are not in themselves sufficient to ensure optimal project productivity. The number of tasks developers must manually perform with their tools, no matter how effective and efficient the tools are, continues to increase. Eventually this either overwhelms developers or leads to them not performing (often critical) tasks e.g. they avoid or reduce appropriate project management metrics capture, detailed design analysis and rigorous software testing.

The solution is provision of various forms of automation in the software tools developers use - the tools carry out perhaps a wide range of activities for the developer at appropriate times and inform the developer of results of actions in appropriate ways [2, 4, 5]. Many automation facilities have been used in tools, and in recent years more and more have tended to be added. Examples of automated tool support include information analysis (i.e. checking of software artefacts for consistency); autonomous agents (that perform tasks for users, including notification, information update and change propagation, and task co-ordination); code generation (generating user interface, data management and/or information process code from specifications); and

We have focused in recent years on two areas of automation in software tools: (1) generating code from high-level software specifications; and (2) utilisation of high-level software information by agents to support collaborative work, change management and component testing. From our experiences developing a number of software tools using these automation approaches, we have learned a number of lessons for further research in these areas. These include:

- the need to support software tool meta-model extension
- the need for on-the-fly enhancement of tool notations, event processing and code generation facilities
- support for software artefact change propagation and annotation
- the need to have reflective, high-level information to running software system components
- the continuing challenges of enhancing COTS tools with these kinds of automation facilities, including the need for sharable, extensible software information models for software tools and open tool infrastructure

In the following two sections we briefly review some of our recent automated software tools. We give three examples of tools generating code from high level software descriptions, including a performance test-bed generator, a data mapping tool and an adaptive user interface generation tool. We describe three tools utilising event-driven software agents, including collaborative work components, requirements management and component testing tools. We then review the key lessons we have learned from this work and summarise future directions for our research on automated software tools.

## 2. Code Generation Examples

The three tools described in this section all generate large amounts of complex code from high-level descriptions of different aspects of software. Their ability to do so is dependent on the software information model they generate code from and the developer's ability to construct instances of this model via appropriate user interfaces and design metaphors.

### *2.1. SoftArch/MTE*

Determining if software architecture designs will meet required performance benchmarks is very challenging [3, 14]. SoftArch/MTE is a distributed system performance test-bed generator [6]. It takes high-level descriptions of software architectures and generates client and server code that is automatically deployed and run to inform developers of likely architecture performance. As real code is generated and is deployed and run on real machines, quite accurate

performance measures can be obtained very quickly by developers. Figure 1 outlines how Softarch/MTE works. A tool (SoftArch) is used to model software architectures at a high level of abstraction. This generates an XML-encoding of the architecture design including clients, servers, client requests, server operations, database operations and tables, and middleware and host characteristics. XSLT transformation scripts convert the XML into code and deployment scripts, which are uploaded and run on distributed client and server machines by deployment agents. Performance results are sent back to SoftArch/MTE and visualised with MS Excel™.

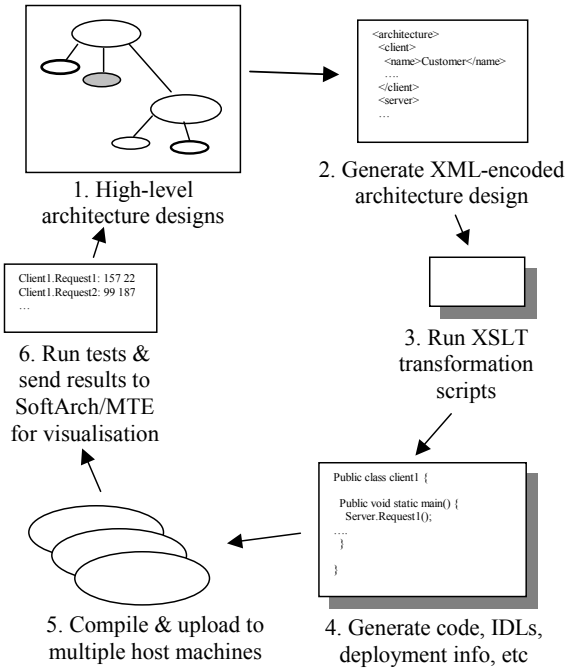


**Figure 1. SoftArch/MTE performance test-bed.**

*2.2. Form-based Data Mapping Specification*

Implementing mappings between complex data structures is needed for various domains, including business-to-business e-commerce, but is time-consuming and hard to maintain with convention languages and tools [7]. We have developed a form-based data mapping tool that provides an environment in which non-programmer end-users (business analysts) specify correspondences between complex data models [12]. These data models are rendered as "business forms", and analysts specify form element correspondences using a drag-and-drop, form-copying metaphor. A transformation implementation is then generated from this high-level correspondence specification that when run transforms data in the source form format into target form data. Figure 2 illustrates this form-based mapping specification approach. Meta-data is imported from schema files or design tools. Business form representations are generated, and then analysts specify correspondences between form elements, effectively programming-by-demonstration of mappings. XSLT transformation scripts are generated by the mapping tool that implement the data transformations specified.

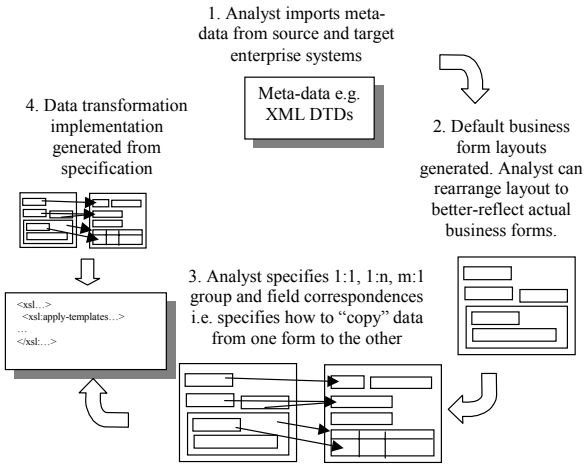


**Figure 2. Form-based data mapper.**

*2.3. Adaptive User Interface Technology*

Many systems require thin-client interfaces that will run on multiple display devices and will suit different kinds of users and user tasks [13]. For example a customer accessing an on-line store via a wireless PDA will have quite a different interface for the same functions as a staff member accessing the system from a desktop PC web browser. Building such interfaces with conventional techniques results in large numbers of very similar server-side web page implementations. We have developed a GUI design tool and adaptive interface mark-up generator to make design and implementation of such adaptive interfaces easier [8]. Figure 3 illustrates this Adaptive User Interface Technology (AUIT) system. A designer uses an abstract representation of an interface to specify generic screen components, layout and interaction. This tool generates Java Server Pages with a set of custom tags describing the adaptable interface. When deployed in a web server and accessed by a user, the tags generate a user interface tailord to the accessing user's display device, user characteristics and current task.

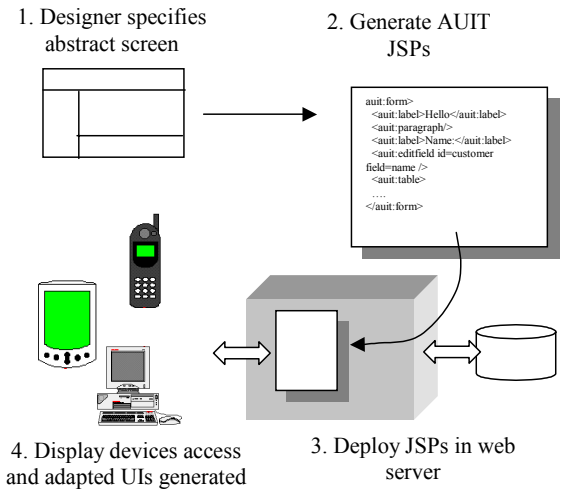


**Figure 3. Adaptive User Interface Technology.**

## 3. Task Automation Examples

The following examples are of software tools we have developed that incorporate software agents to assist developers by automating various tasks. The agents are driven by event subscription or user request. The agents access and manipulate software artefact information for the user in various ways.

### *3.1. Collaborative Work Agents*

Most software engineering tools require some degree of collaborative work support, though most hard-code this and are thus inflexible and require extensive engineering to build [1, 5]. We have developed a set of plug-in software agents that interact with tool client and server components to add collaborative work support to tools [9]. Figure 4 illustrates the basic structure of our approach. Collaboration agents support collaborative editing, group awareness and version control. Communication agents support messaging, annotation and dialogue between developers. Co-ordination agents provide change notification actioning, locking, to-do list task scheduling and even workflow co-ordination. The agents can be plugged into or removed from tools at run-time. In order to add these agents to tools, they need to determine various user interface, distribution and persistency support of the tool components. This is done by having the tool components publicise "aspects" which describe this information and can be introspected by our collaboration agents.

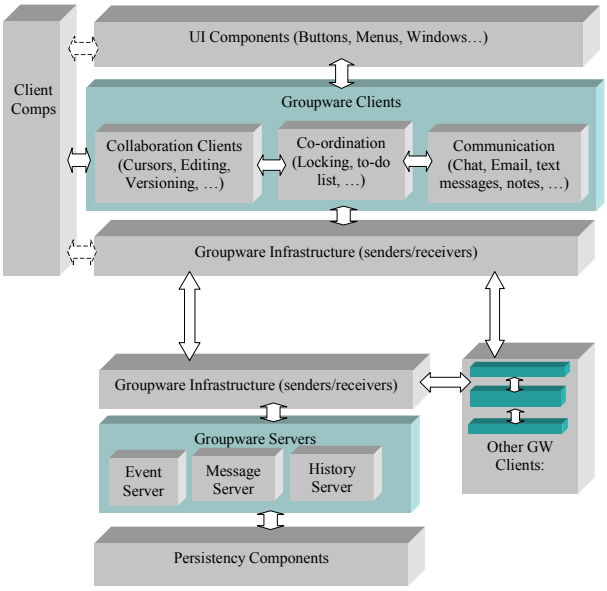


**Figure 4. Collaborative work components.**

### *3.2. Requirements Management Agents*

Based on an empirical study of software engineering practitioners use of abstract information models [17], we have built a prototype tool for managing relationships between functional and non-functional requirements, use case models, and black-box test plans.

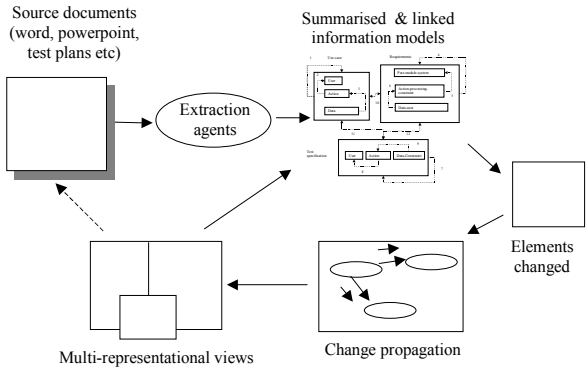


**Figure 5. Requirements management.**

This environment contains software agents that extract information about these three different abstract software representations, summarising the key parts of each information model. Other agents create implicit links between elements in different representational models or allow developers to create explicit links and modify artefact information. When elements in one representation change, descriptions of these changes are captured and sent to other models. Developers can view the impacts of these changes,

trace sources of changes, and view information from different representations in multi-representational views. We hope to provide other agents that can update source, 3rd party software artefact documents in the future. Figure 5 outlines the main facilities of this prototype tool.

### *3.3. Aspect-oriented Component Validation Agents*

Validating that deployed software components meet their required functional and non-functional constraints is very difficult [11, 15]. We have developed software agents that inspect the constraints on deployed software components and perform validation tests on these components. The components are designed with the aspect-oriented component engineering method [10]. Their implementations have information characterising system aspects, such as persistency, distribution, security and transaction processing characteristics, associated with them as XML documents. Our validation agents inspect these component aspects and formulate tests to ensure the component's aspect-encoded constraints (functional and non-functional) are met in their current deployment situation. Some agents deploy 3rd party testing tools, like SoftArch/MTE, to carry out complex tests and analyse the results produced.

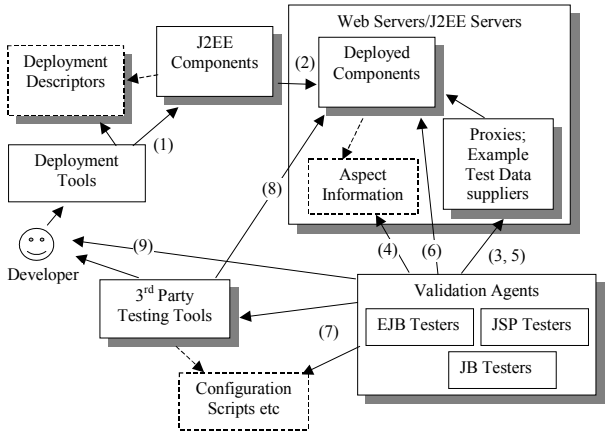


**Figure 6. Agent-based component testing.**

## 4. Key Issues and Future Research

We have identified several key issues when building the tools described in the previous two sections. These are summarised below, along with some of the research directions we are investigating to make the development of such automated software tools easier and more feasible.

### *4.1. Software Information Model Extension*

Many of our tools require extensible meta-models in order that their capabilities can be enhanced by developers as required. For example, we have added new kinds of architectural characteristic support to SoftArch/MTE as we have extended the tool to support a wider range of target test bed generation (e.g. message-based systems and web-based interfaces). Similarly, the information models the requirements management agents use needs to be extensible as different users have different degrees of detail in each model they are interested in capturing.

Our experience with these tools has indicated that ideally many automated software tools will have software information models that can be extended as required. Versioning these information models and ensuring compatibility between old and new models often may need to be supported. We are developing a new software meta-tool with a fully extensible meta-model.

### *4.2. Tool Notation and Behavioural Extension*

Many of our tools need to allow developers to add additional notational representations (in order to make use of meta-model extensions or support new kinds of artefact views), and similar require behaviour extensions (such as new code generation extensions or constraints on models built). Examples include extending the modelling notations of SoftArch/MTE, AUIT and our requirements modeller, and adding new target code generation for SoftArch/MTE, our form-based mapper, AUIT and component validation agents.

Most of our tools have very limited notational support, and limited run-time behavioural modification. This results in frustrating turn-around time when enhancing tools and requires developers to have in-depth knowledge of tool internal structures to make any enhancements. Our new meta-tool architecture supports flexible view notation definition as well as a wide range of run-time behaviour enhancement by allowing developers to incorporate new code into the tools at run-time. This code includes constraint checking, event/action rules and XSLT transformation scripts which we have found very useful for implementing code generation.

### *4.3. Change Propagation and Artefact Annotation*

Many of our tools need to track changes made to software artefacts. These include our requirements modeller, collaborative work supporting agents and component validation agents. SoftArch/MTE and our requirements modeller require support for specifying links between model elements and for annotating elements with

additional, semi-structured information such as design rationale and change explanation.

While many software tools have moved to adopting publish-subscribe event-based architectures the use of these architectural facilities is still relatively limited. We have found using this architecture important in driving many task automation agents, particularly those supporting collaborative work. The ability to link, refine and annotate software artefacts in many of our tools is important and hence should be supported within a tool infrastructure.

### *4.4. Reflection Information*

Some software tool automation facilities need access to detailed information about running tool components. Examples include the plug-in collaborative work agents, the data mapping tool and the aspect-based component validation agents. The collaborative work agents need to adapt tool component interfaces to integrate new facilities and make use of publicised component event mechanisms. The data mapper needs to obtain meta-data information from source and target structures. The validation agents need to determine what the requirements on deployed components are in order to perform appropriate tests to validate these are met.

In our recent work we have developed a mechanism to annotate software components with information about the "aspects" of a system they provide or require services [10, 9]. This is used by our collaborative work and validation agents. Interestingly, tool automation is required in order to generate this information from annotated component design models. We are investigating adding these aspects as annotations to SoftArch/MTE architecture designs to better-organise the many properties of some of its architecture abstractions.

### *4.5. Tool Integration*

Software tool integration has been a long-standing problem for tool developers and those developing automated support for tools [16, 18]. Some of our tools utilise 3rd party systems in limited ways e.g. SoftArch/MTE uses MS Excel™ to visualise performance data and our requirements modeller extracts summarised data from save files. Many of the automation support described in our tools above could however be very useful if integrated into 3rd party, commercial software development tools. For example, SoftArch/MTE test beds could be generated from (greatly) annotated Rational Rose™ deployment diagrams; mapper transformations from cross-linked MS Access™ screen designs; collaborative work agents potentially added to a very large range of tools; and inter-representation requirements change management added to integrate several different tools.

Three key problems preventing such enhancements of existing tools we have identified are lack of agreed, high-level tool information models that can be shared between tools, lack of adequate tool event and operation APIs, and sufficiently open technologies implementing these APIs and run-time inspection facilities allowing other tools to discover them. We are investigating "componentising" some of the tool automation facilities outlined in the previous section in order to add them to COTS software tools and to allow easier use of these tools by our own.

## 5. Summary

We have been developed a range of software tools with automation features, in particular ones that generate code from various high-level software information models and ones that leverage "agents" to perform various task automation facilities for developers. Some of the key issues in building such tools we have encountered include the need to support extensible information models, notations and event handling behaviour, change propagation and information element annotation capabilities, detailed reflective information encoded with software components, and tool integration. We are developing a meta-tool with these capabilities to enable us to better support the construction of various automated software tools, and developing various integration components to support the integration of our new tools and enhancement of existing COTS software tools with automation.

## References


1. Bandinelli, S., DiNitto, E., and Fuggetta, A. Supporting cooperation in the SPADE-1 environment. *IEEE Transactions on Software Engineering,* vol. 22, no. 3, December 1996, 841-865
2. Fischer, G, Domain-oriented design environments, In *Proceedings of the Seventh Knowledge-Based Software Engineering Conference*, McLean, Virginia, 1992, pp. 204-213.
3. Gorton, I. And Liu, A. Evaluating Enterprise Java Bean Technology, In *Proceedings of Software - Methods and Tools*, Wollongong, Australia, Nov 6-9 2000, IEEE CS Press.
4. Green, C., Luckham, D., Balzer, R., Cheatham, T., Rich, C. Report on a Knowledge-Based Software Assistant, Technical Report RADC-TR-83-195, Rome Air Development Center, August 1983, Reprinted in: C.H. Rich, R. Waters (eds.): *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann Publishers, Los Altos, CA, 1986, pp. 377-428,.
5. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, vol. 2, no. 5, September/October 1998, IEEE CS Press.
6. Grundy, J.C., Cai, Y. and Liu, A. Generation of Distributed System Test-beds from High-level Software Architecture Descriptions, In *Proceedings of the 2001 IEEE Automated*

*Software Engineering Conference*, San Diego, 26-29 Nov 2001, IEEE CS Press, pp. 193-2000.

7. Grundy, J.C., Mugridge, W.B., Hosking, J.G. and Kendall, P. Generating EDI Message Translations from Visual Specifications, In Proceedings of the 2001 IEEE Automated Software Engineering Conference, San Diego, CA, 26-28 Nov 2001, IEEE CS Press.
8. Grundy, J.C. and Zou, W. An architecture for building multi-device thin-client web user interfaces, In *Proceedings of the 14th Conference on Advanced Information Systems Engineering*, Toronto, Canada, May 29-31 2002, Lecture Notes in Computer Science.
9. Grundy, J.C. and Hosking, J.G. Engineering plug-in software components to support collaborative work, to appear in *Software – Practice and Experience*.
10. Grundy, J.C. Multi-perspective specification, design and implementation of software components using aspects, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 10, No. 6, December 2000, pp. 713-734.
11. Haddox, J.M., Kapfhammer, G.M. An approach for understanding and testing third party software components, In Proceedings of 2002 Annual Reliability and Maintainability Symposium, Seattle, WA, 28-31 Jan. 2002, IEEE CS Press.
12. Li, Y., Grundy, J.C., Amor, R. and Hosking, J.G. A data mapping specification environment using a concrete business form-based metaphor, In *Proceedings of the 2002 International Conference on Human-Centric Computing*, IEEE CS Press.
13. Marsic, I. Adaptive Collaboration for Wired and Wireless Platforms, IEEE Internet Computing (July/August 2001), 26-35
14. McCann, J.A., Manning, K.J. Tool to evaluate performance in distributed heterogeneous processing. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, IEEE, 1998, pp.180-185.
15. McGregor, J.D. Parallel Architecture for Component Testing. Journal of Object-Oriented Programming, vol. 10, no. 2 (May 1997), SIGS Publications, pp.10-14..
16. Meyers, S. Difficulties in Integrating Multiview Editing Environments, IEEE Software, **8** (1), 1991, pp. 49-57.
17. Olsson, T., Runeson, P., Software document use: A qualitative survey, Technical report, Dept. of Communication systems, Lund University.
18. Reiss SP. The Desert environment. *ACM Transactions on Software Engineering & Methodology*, **8** (4), Oct. 1999, pp.297-342.