Design Pattern Automation

Andy Bulka
WindowWare Multimedia
6/32 Loller street,
Brighton VIC 3186, Australia
abulka@netspace.net.au

Abstract

As *design patterns* become more mainstream, various IDE's (Integrated Development Environments) and UML modelling software environments have begun to introduce support for design patterns. For example, developers browse through a catalog of design patterns and drop one onto a UML workspace, whereupon various classes appear with the appropriate methods and attributes inserted. This paper explores the state of pattern automation software, discusses the pros and cons of various approaches and then goes on to discusses the broader issues raised by the attempt to automate something which some argue, in principle, perhaps should not be automated at all.

Keywords: Design Patterns, Automation, Tools, UML.

1 Introduction

The types of tools discussed in this paper are becoming increasingly prevalent in commercial software development environments. As far as I know, the effectiveness of design pattern automation tools has not been rigorously tested in controlled studies. This paper is a report based on personal experience and observed practice.

The type of design patterns that are being discussed in this paper are of the type described in Gamma et. al (1995). There are many sources for additional design patterns of this level of granularity e.g. the series of books arising out of Programming Languages of Program Design Pattern Conferences e.g. Martin et. al (1998).

2 Levels of Automation

There are various degrees of pattern automation offered by UML modelling tools. These range from the static to more dynamic mechanisms. See *Table 1*.

Static approaches merely insert a group of related classes onto a workspace whereas dynamic approaches attempt to integrate classes from newly inserted patterns with existing classes, renaming class and method names as required.

Better levels of automation provide wizards which allow patterns to be customised to more closely fit the problem and context of its proposed use.

Advanced levels of automation go even further - classes involved in patterns automatically and intelligently respond to changes made by the designer in other parts of the UML model. This way patterns maintain their 'integrity' and cannot be accidentally broken by the designer.

3 The State of Pattern Automation Technology

As we examine these software automation tools, we should recognise that these tools automate pattern *implementations*. That is, in the end, they create specific classes and methods on the UML workspace. They do not automate the designer's deliberations - the designer must still analyse the forces and context and then *choose* which pattern to use.

Object oriented design and modelling patterns in UML notation is a complex, high-order process - the professional software designer must still understand and know how to select the appropriate options offered by wizard dialog boxes. Thus the role of the programmer *as designer* is not automated - though some tools go some way towards assisting this process as well e.g. *Cogent*, by Budinsky et. al. and Vlissides (1996) provide wizards, much online documentation and context sensitive help to help with the designer's deliberations.

Later in the paper I will discuss whether the aforementioned complex design and modelling processes can be made easier and more accessible to less experienced designers, through the use of the new generation of design pattern automation tools.

Automation Level	Description	Pro	Con
Simple Template *e.g. UMLStudio*	Stores the structural solution of a pattern. Inserts a group of related classes onto a workspace	Prebuilt structures save modelling time	Manual customisation required.
Parameterised Templates e.g. *TogetherJ* e.g. *ModelMaker* e.g. *Cogent*	Dialog box prompts for class and method names before inserting a group of related classes onto a workspace	Integrates classes from newly inserted patterns with existing classes.	Typing class and method names may be tedious and error-prone.
Intelligent Patterns e.g. *ModelMaker*	Classes that have been inserted as parts of patterns automatically and intelligently respond to changes in other parts of the UML model	The integrity of patterns is maintained.	Developer may be confused or feel restricted when model auto-changes.

Table 1: Levels of Pattern Automation

3.1 The simple Template approach

An example of the simple template approach to design pattern automation can be found in *UMLStudio* http://www.pragsoft.com. The user of this UML modelling tool browses through a catalog of design patterns, viewing thumbnails of various designs. The user selects a design pattern implementation and it subsequently appears on the workspace. Finally, the user makes any necessary modifications to the class and method names, often moving pre-existing classes into positions occupied by the new classes from the template in order to fully integrate the pattern classes into the existing UML model.

This series of operations is illustrated in the next three figures, steps 1 to 3. The goal is to use the Adaptor Pattern to wrap the class Person.

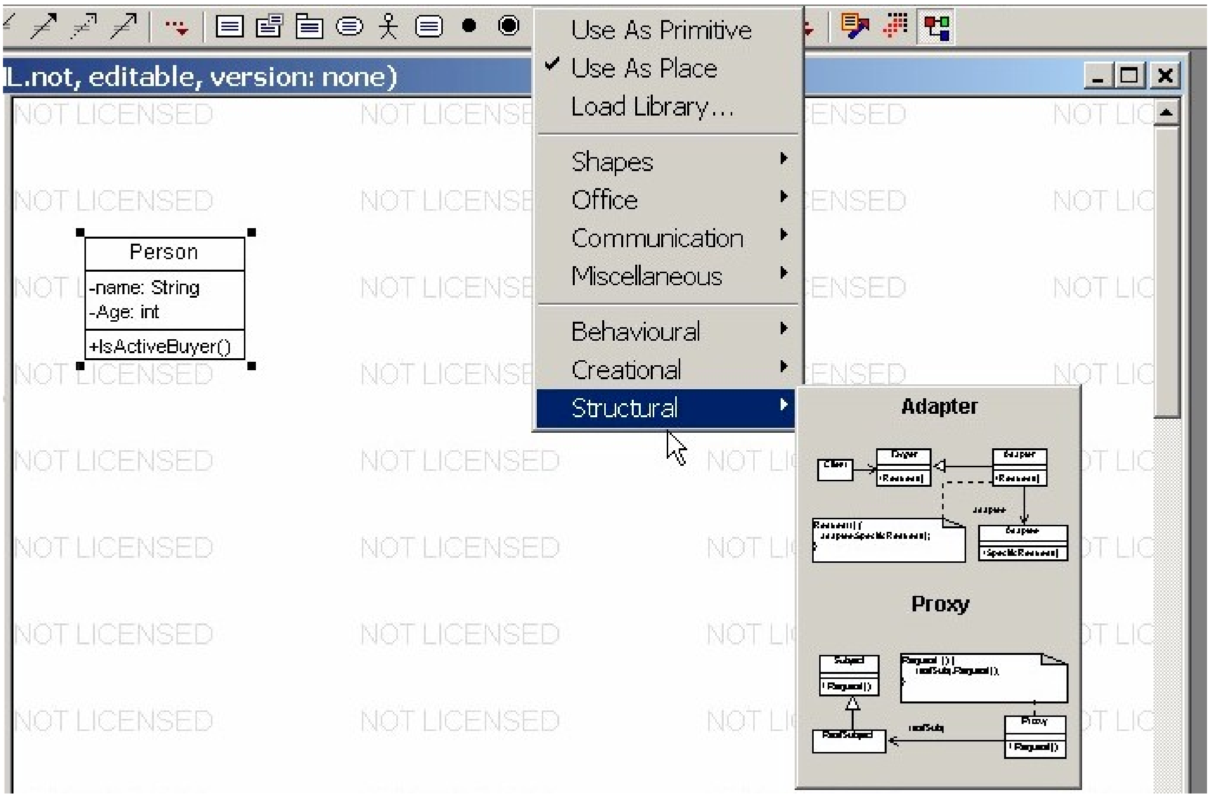

Figure 1: Browsing through the *UMLStudio* pattern catalog. Step 1 of 3.

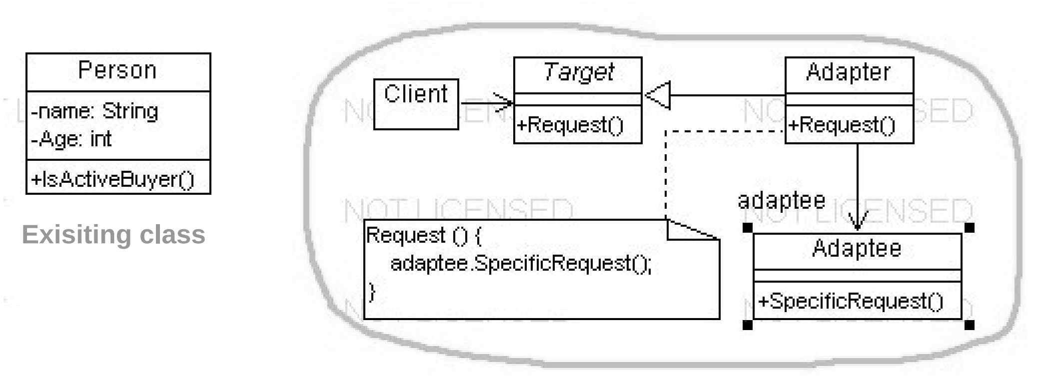

Figure 2: New classes implementing an Adaptor Pattern appear on the workspace. (UMLStudio) Step 2 of 3.

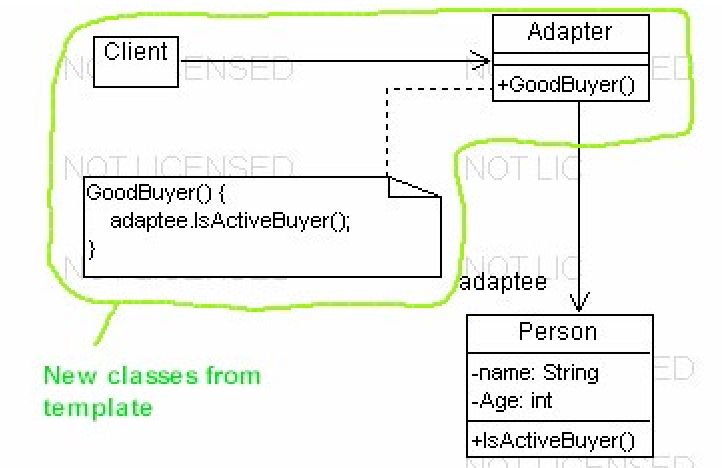

Figure 3: Workspace after manual editing. Step 3 of 3.

The point of using the Adaptor Pattern in this example is to adapt the interface of the `Person` class which consists of the method `IsActiveBuyer` to a new interface in a new class, consisting of the method `GoodBuyer`.

In order to achieve this goal, we need to customise the classes that were dropped onto our workspace from the design patterns catalog. This is a manual process since there is no further automation offered by the UMLStudio tool. Figure 3, above, shows the final result. To achieve this result we manually had to do the following:

1. Replace the new `Adaptee` class with our pre-existing `Person` class.
2. Preserve the wiring between the `Adapter` class and the `Adaptee/Person` class.
3. Rename the generic methods provided by the new Adaptor Pattern classes to methods appropriate to the class being adapted. Thus the method `Request` is manually renamed `GoodBuyer`. The method `SpecificRequest` is manually renamed `IsActiveBuyer`.
4. Write the code shown in the UML text comment box of *Figure 3*.

The need to manually customise the classes that are copied out of the template library is this technique's biggest drawback. The pre-existing `Person` class shown in step 1 is ignored by the Adaptor Pattern template - the new classes from the adaptor pattern template library are not automatically integrated with the existing `Person` class.

It would save manual editing if the user of the modelling tool first selected the class that he or she wanted to adapt, *then* applied the Adaptor pattern. The resulting group of classes would wire themselves and integrate with the selected class. Let us now examine the details of this more advanced approach.

3.1.1 Parameterised Pattern Templates

A more dynamic and intelligent approach to pattern automation is offered by *TogetherJ* http://www.oi.com/together.htm and *Modelmaker* www.delphicase.com. These UML modellers / IDE's allows you to attach patterns to existing classes and offer the ability to customise aspects of the template's application using wizard dialog boxes.

Here we use *Modelmaker* to apply the Wrapper Pattern to the `Person` class:

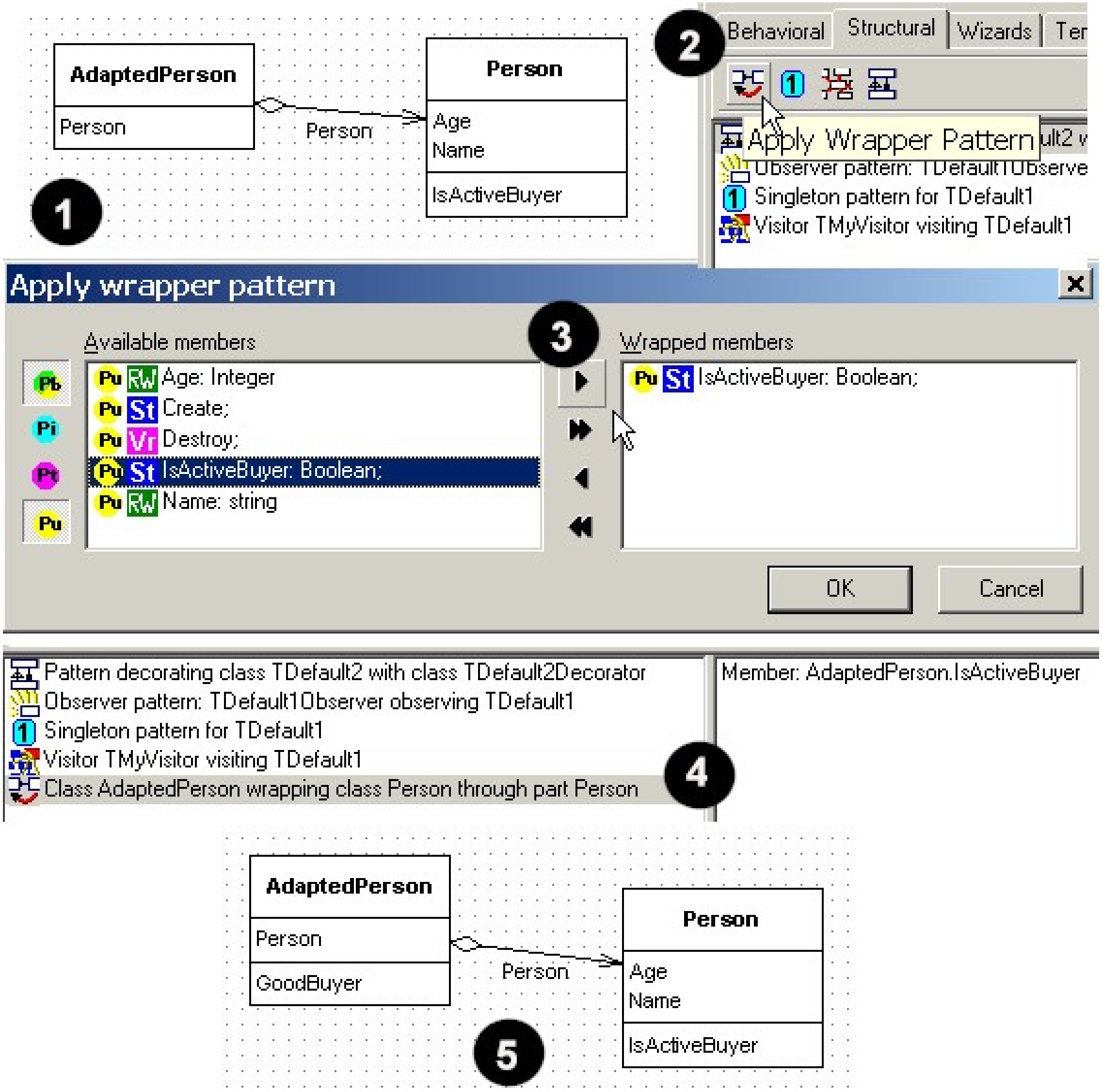

Figure 4: Steps in applying the Wrapper Pattern (*Modelmaker*).

The steps in applying Wrapper Pattern in ModelMaker as shown in Figure 4 are:

1. Locate the class you want to wrap and create a wrapping class that points to it.
2. Select the pointer attribute and click on the "Apply Wrapper Pattern" icon.
3. In the dialog box which lists the methods of the class being wrapped, select any methods that you want to appear in the new wrapper class. Click OK.
4. The pattern has now been applied and is added to ModelMaker's list of active patterns being used in the model.
5. Notice the Wrapping class now has a new method 'GoodBuyer' which simply delegates to the IsActiveBuyer methods of the wrapped 'Person' class.

The benefit of Parameterised Pattern Templates is that the new classes and methods are integrated wit h existing classes and methods, since the currently selected UML object and the 'parameters' asked for in the wizard during the design pattern automation process give the tool the necessary information to do the integration.

An additional benefit of Parame terised Pattern Templates is that usually, after the dialog completes, there is little to no manual coding to do. For example, Modelmaker actually *generates* the code mapping the call to the new method (`AdaptedPerson.GoodBuyer`) into a call to the existing adapted class method (`Person.IsActiveBuyer`).

A drawback to Parameterised Pattern Templates is that you are usually locked into a modal dialog box - typing and remembering class and method names is tedious and error-prone, unless a namespace browsing facility is provided.

Other tools, like *TogetherJ* offer additional features like a drag and drop GUI, so that for example, specifying adaptor and adaptee classes is done with a drag of the mouse.

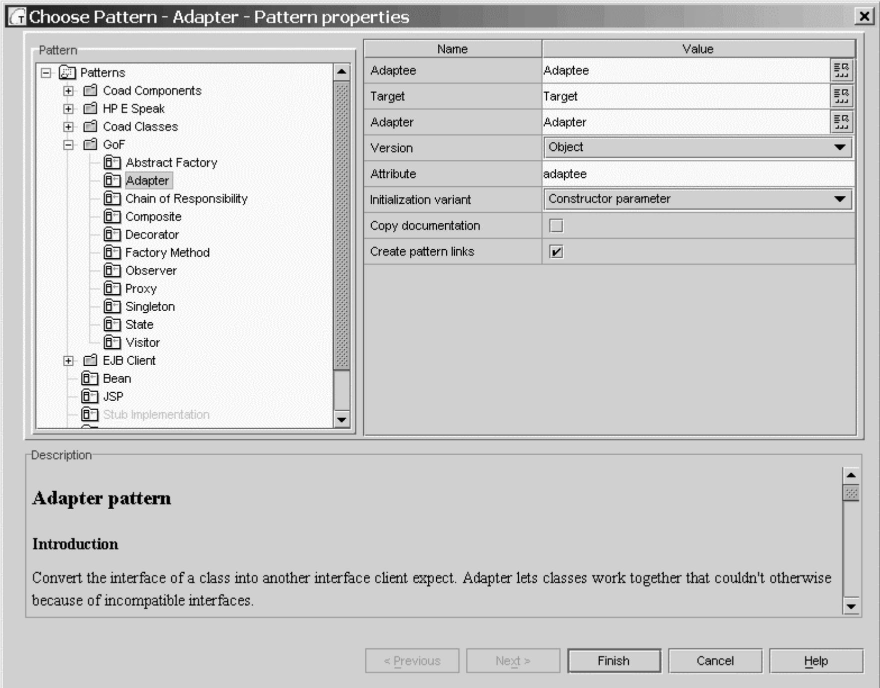

Figure 5: Select a pattern and tweak the parameters. (*TogetherJ*).

3.1.2 Super Parameterised Pattern Templates

Cogent, by Vlissides et. al. uses wizards (dialog boxes with multiple pages of parameters - *see figure 7.*) to strongly customise the pattern implementation automation process. Wizards allow the designer to conside r and choose various trade-offs, ultimately generating different code.

For example, when *Cogent* applies the *composite pattern*, the sort of questions asked are: Whether to cache. Whether to use linked list vs. stretchable arrays. Whether to access via sim ple index vs. iterator. Whether class management API should include append/prepend, insert/remove operations. Whether to declare child management operations in all classes or in composite class only. Whether to raise exception under certain conditions e tc.

The big promise of this approach is that the designer can more readily fit the pattern implementation to the exact needs of his or her problem.

3.1.3 Dynamic Intelligent patterns

Modelmaker www.delphicase.com goes one further step in pattern automation and offers the idea that patterns are dynamic first class citizens in a UML model. Once applied, the modelling tool remembers which classes are involved in the pattern and actively protects the pattern from accidental d amage.

Additionally, as parts of the model change, the classes and methods involved in the pattern are automatically updated as necessary. For example, if an adapted method name changes, the code in the wrapping class method will automatically be altered to call the renamed method. Similarly, if an adapted method is deleted, the wrapping class method will also be deleted.

This sort of dynamism can get quite complicated yet beneficial. For example, a more complicated pattern like Visitor requires that e ach visitor class has a visit_xxx method for each class to be visited. When dynamic patterns are being enforced, each time you add a new class to be visited e.g. the `CleaningLady` class, all visitor classes *automatically* get a new method called `visitCleaningLady`. Thus the integrity of the *visitor pattern* is maintained.

4 Discussion

To some, the whole idea of pattern automation is a contentious issue. This is because a design pattern is actually a reasonably complex *document* describing a problem in a given context surrounded by various forces. Any solutions offered in design pattern documents are qualified by thorough discussions of the various trade -offs of using certain designs and implementations over others, given various context and forces. Thus a pat tern is not just a solution structure consisting of a UML diagram, rather it is an essay for the designer to read, think about and ultimately adapt what is learnt to his or her needs.

In fact a designer may encounter situations where none of the sample implementations offered in the design pattern document are suitable to the designer's particular situation, or possible to implement in the implementation language available (e.g. it may be a non-object oriented language). In these cases the designer will need to invent a custom solution based solely on the *ideas* discussed in the design pattern document.

Thus the idea of *automating* this entire complex process can therefore seem ludicrous, given the amount of design, human thought, and customization of solution structures often (or perhaps always?) required.

4.1 What aspect of patterns should be automated

The rebuttal to the objection that "pattern automation should not be attempted at all" is to be found in distinguishing exactly *what* aspect of design pattern process software tools are really trying to automate. Are they really trying to replace the designer's deliberations or merely assisting the designer in the mundane and often complex task of *implementing* the pattern?

The tools so far described in fact do not replace the designer's deliberations in *choosing* which pattern to use at any given time. The tools do not replace the need for the designer to understand how and why he or she is using a particular pattern. Nor do the tools replace the need for the designer to understand the balance of forces and trade-offs involved in choosing different solution variations.

Once the designer has decided what sort of pattern solution is required, then what the pattern automation tools *can* help him or her with, is the moderately complex and careful surgery required to implement that pattern solution into a UML diagram or into source code.

Are Refactoring Tools Doing Pattern Automation?

It could be argued that refactorings Fowler (1999) are related to and in some cases similar to Patterns. They encapsulate proven 'moves' in the chess game of refactoring code. Some refactorings go so far as to encapsulate proven designs e.g. *'Change Unidirectional Association to Bidirectional'*, (p200, Refactoring) show the best way to implement a bidirectional association. Similarly, 'Duplicate Observed Data' (p 189, Refactoring) shows how to implement the Observer Pattern.

Source code editors with refactoring support are becoming more popular and moving from their original use in Smalltalk circles to the more mainstream Java community, more recently to the Python community and beyond.

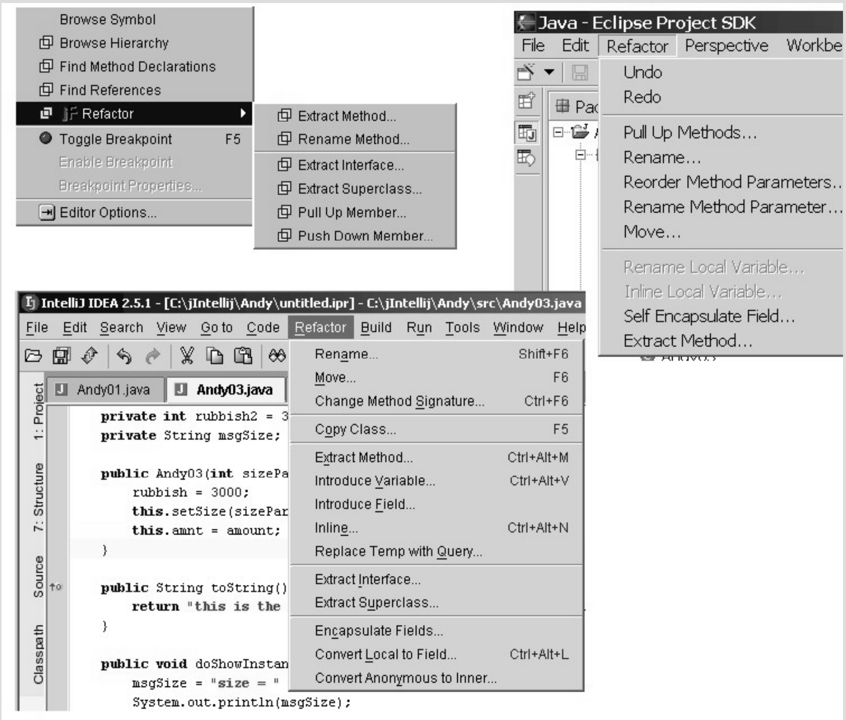

Figure 6: Examples of refactoring editors and refactoring tools that plug into existing IDE's. Clockwise from the top - Jrefactory plug-in, Eclipse, IntelliJ.

4.2 Fitting the pattern to the problem

Gamma et. al (1993) say in *Design Patterns*, p. 3: "Design Patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem. The design patterns in this book are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.*"

Thus an important argument *against* the idea of automating pattern 'implementations' is that by the time the design process has considered all the forces in the designer's context and the designer has adapted the pattern solution to fit his or her unique situation, then the implementation stage will involve coding up something that is quite unique. Pattern automation tools that offer out of the box solution structures are unlikely to ever match what the designer really wants - even when you take into account the customisation features of the tool.

The problem of fitting the automated pattern templates to the exact problem at hand might in principle always be a mismatch in the sense that say, hand crafted solutions might always be better. A custom tailored suit is always going to fit someone better than an off the rack suit. But do we go further and say that in principle each pattern implementation is *necessarily* going to be unique? Do pattern implementations *always* need to be 'custom tailored'?

Let us remember that there are also benefits to non-custom tailored approaches - standards, comprehensibility, maintainability, proven and efficient designs etc. If there is room, in principle, for pattern solution template libraries then these benefits should not be overlooked.

It is also likely that given the wide variety of patterns, that some patterns are more suitable to be automated than others. It is my experience that some patterns, with suitable wizards, are quite suitable for automation *e.g.* Adapter, Visitor, Strategy. This is not to deny that there are variations of these patterns that are not handled by my current automation tools.

4.3 Adding your own patterns to pattern solution catalogs

Designers are of course not restricted to the implementation solutions offered in a pattern catalog - some tools in fact offer the capability of allowing custom pattern implementations to be added to the pattern catalog, thereby increasing the relevance of the pattern implementations offered by the pattern automation tool. *PsiGene* Heister et. al (1997) for example describes a framework specific tool (for house/building simulation) which allows proven custom, domain specific patterns to be easily re-used. Thus developers can build catalogs of pattern solution implementations which are suited to their organisation, language and problem domain.

A drawback to custom templates which designers add to a catalog themselves, is that these pattern templates do not have the advanced customisation wizards that come with standard patterns. This is because wizards need to be carefully designed and are complex enough that they need to be implemented by the automation pattern tool makers themselves.

4.3.1 The Promise of Wizards

Despite the fact that complex wizards are probably only going to be supplied by pattern automation tool makers, and not by tool users, I believe that, besides having a variety of solution structures to choose from in a catalog, parameterised patterns with smart wizards offer the designer a promising way of generating unique solutions - better fitting his or her context and forces. For example, in my experience, a flexible Adaptor/Wrapper Pattern automation wizard copes with many adapting/wrapping situations. On the other hand, a Composite Pattern Wizard is actually not offered in *ModelMaker* because, as the author of *ModelMaker* told me: 'Every composite pattern situation is different.'!! I ended up adding a custom composite pattern to the *ModelMaker* catalog, which suited my needs at the time - and this worked very well. It was easily parameterised, asking me the names of methods and classes but of course had no smart wizard to accompany it.

Cogent, co-designed by one of the *Design Patterns* book authors, Vlissedes, on the other hand *does* take on the challenge and does offer a Composite Pattern wizard, albeit a complex one (see *Figure 7*). This wizard comes with the *Cogent* tool.

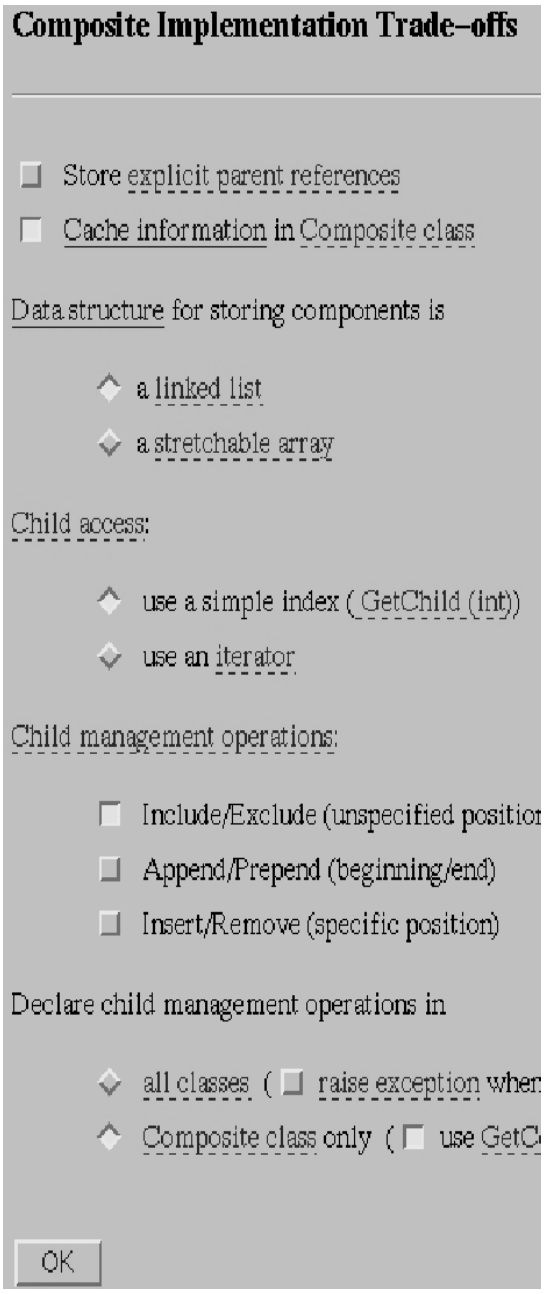

Figure 7: Composite Pattern Wizard (*Cogent*)

The big promise of wizards is that the designer can more readily fit the pattern implementation to the exact needs of his or her problem. However, until there is a large library of radically alternate implementations of patterns available, plus very smart wizards, most designers will probably still feel restricted by the available pattern template choices - despite the levels of customization being offered. Intriguingly, once such libraries and smart wizards exist, then the process of modelling may radically change and become more like piecing together smart lego blocks rather than hand crafting every class and relationship manually. In the hands of a disciplined and experienced designer such tools would indeed be powerful.

4.4 Patterns as Lego pieces in the hands of babes

There is the very real possibility that inexperienced designers will be tempted to piece together architectures using pattern templates supplied with an automation tool, without really understanding the traditional role of patterns in resolving the forces arising in a Object Oriented analysis and design process.

And there is also the danger that designers will attempt to twist their designs in order to take advantage of groovy pattern implementations offered by their pattern automation tool. This may even lead to situations where a particular pattern implementation may do only 80% of what is really required yet the designer still uses it, hoping to take advantage of time savings. In this case the designer will need to implement the rest of the functionality manually.

Conversely, if a particular pattern implementation does *more* than required, the excess code in the design will confuse programmers, unless it can be safely removed. Furthermore some dynamic intelligent pattern tools (e.g. as used in *Modelmaker*) may actually resist such manual fine tuning (hence preserving pattern integrity), thus restricting the programmer, and forcing upon the design too much code, some of which is unnecessary.

Then again, if the software works and delivers the functional requirements asked of it, does it really matter all that much if the solution has been over-engineered, or built out of pre-fabricated parts? These same age-old philosophical questions relating to hand crafted vs. factory produced surely arise in other industries too? e.g. engineering, tailoring and furniture manufacturing etc.

In the end, I think that yes, in some situations, highly customised solutions will be necessary, and no automated tool is going to help you. In other situations, standard implementation solutions will actually be sufficient, *especially with the ability to customise a choice of standard implementations using complex wizards*.

Interestingly, automation facilities, especially wizards, will probably serve to *educate* designers and programmer as to the trade offs they should be considering, since reading dialog boxes and being guided through choosing options representing trade-offs will be helpful in supplementing one's reading and understanding of original design pattern documents.

5 Other Aspects of Pattern Automation

5.1 Comparing Refactoring Automation and Pattern Automation

Interestingly, the idea of automating '*refactorings'* is not as controversial as pattern automation. Refactoring editors [1] certainly save time and keystrokes. Nobody is concerned about the pitfalls of having smarter, more automated code editors.

Then again, the current range of 'moves' offered by refactoring tools [2] are suitably low level such that they cannot spark any controversy relating to attempting to replace the design process. For example, the automated refactoring *'Extract Method'* wizard takes the selected lines of code, prompts you for a method name, and automatically replaces the selected code with a method call - placing the selected code into a new method for you. This is no doubt a helpful tool.

However as soon as you automate other more advanced refactorings like *'Duplicate Observed Data'* (Observer Pattern) etc. you then stray into the same controversial territory as pattern automation. Namely, you are automating a particular implementation of a pattern rather than using one's understanding of a pattern to inform your design deliberations and choices.

5.2 Software Assistance in Keeping Track of Patterns

As patterns (especially dynamic, intelligent, self protecting patterns) become first class citizens of a model, it makes sense to know which patterns have been used in a model and where these patterns live. Whilst there are conventions for representing the presence of patterns in a UML diagram Booch et. al (1999) there is no standard way to represent their presence in a modelling tool.

[1] Java refactoring support can be found in the *Jrefactory* plug-in from *Instantiations* (see footnote below), Eclipse (free from www.eclipse.org and IntelliJ www.intellij.com. Python refactoring support can be found in *Bicycle Repair Man* http://bicyclerepair.sourceforge.net.

[2] Interestingly, Instantiations (2002) www.instantiations.com makers of refactoring support tools for Java in the JBuilder, VisualAge and WebSphere IDE's have expanded their offerings to more than just automated refactoring tools. They have recently added to their product line wizard based pattern automation support for nine patterns found in the *Design Patterns* book, plus three other patterns specifically related to the Java language. This development further indicates the related nature of refactoring tools and pattern automation tools.

An interesting approach has been adopted by the modelling tool *Modelmaker*. It lets you view a list of patterns that have been applied in the model. This view allows the designer to see which classes and methods are affected by each pattern.

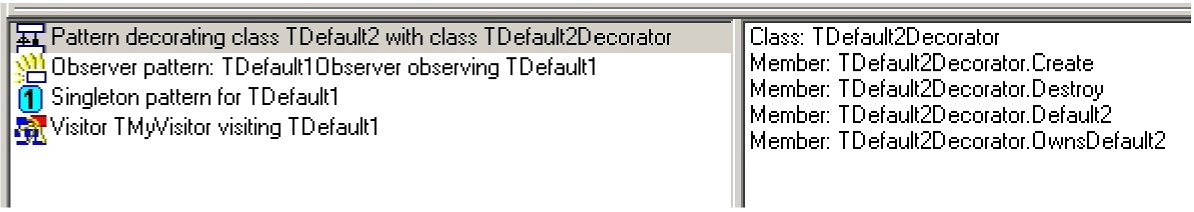

Figure 8: The list of patterns used in a UML model. (*Modelmaker*).

Clicking on each pattern in the left hand pane, in *Figure 8* will display the affected classes in the right hand pane. Pattern properties can be modified from this view. Patterns can also be deleted in their entirety from the model, using this view.

5.2.1 Aspects and Code Sections

When a pattern is implemented its implementation may be scattered over a number of classes, and involves numerous methods and attributes. More to the point, a pattern implementation may involve not just entire new methods but rather, *just a few lines of code in an existing method*.

Aspect Oriented Programming promises to elevate these 'snippets of code' within methods to first class entities. Aspects can be weaved into classes as required.

A similar but alternative approach used by *Modelmaker* is to break methods up into things called *code sections*. Each code section is 'owned' by either the user of the modelling tool, or by a pattern. Code sections (or aspects) owned by patterns are read-only and cannot be modified, except by altering the pattern properties. Sometimes entire methods or attributes are owned by a pattern, allowing this particular modelling tool to 'protect' its living patterns from being clobbered by developers thus enforcing the patterns' integrity.

6 Conclusion

Pattern Automation tools are useful tools - as long as they are used wisely by designers who are already educated in patterns. Pattern Automation tools cannot replace the designers understanding of when and where to use patterns.

As long designers appreciate that they have at their disposal pattern *implementation* automation tools, and as long as designers resists the temptation to use the same fixed implementation every time they need a particular pattern - just because it is included in the tool, then pattern automation can play an increasingly important role in the construction of software, especially when smarter wizards and larger repositories of pattern solutions become available.

As a rule of thumb, given these provisos, if the facilities offered by automation tools help your implementations - go ahead and use them. If your needs are more specialised, then of course you are on your own and will need to hand code your solution.

My own hope is that pattern automation tools will continue to provide advanced facilities for the maintenance, self-preservation and identification of where patterns have been applied. In this way patterns will live as first class entities in designs rather than being mere inspirations for design that perhaps corrode and are forgotten about over time.

7 References:

GAMMA, E, HELM, R, JOHNSON, R and VLISSIDES, J. (1993): *Design Patterns: Abstraction and Reuse of Object-Oriented Design.*

MARTIN, R, RIEHLE, D and BUSCHMANN, F (1998): *Programming Languages of Program Design* Volume 3. See also volumes 1, 2 and 4.

FOWLER, M (1999): *Refactoring: Improving the design of existing code.*

BUDINSKY, F et. al. and VLISSIDES, J. (1996): *Cogent: Automatic Code Generation from Design Patterns*. IBM Research Journal Volume 35, Number 2, 1996 http://www.research.ibm.com/journal/sj/352/budinsky.html

HEISTER, F, RIEGEL, J, MARTIN S, SCHULZ, S, ZIMMERMANN, G (1997): *PsiGene: A Pattern-Based Application Generator for Building Simulation.* Computer Science Department, University of Kaiserslautern, Germany

BOOCH, G, RUMBAUGH, J, JACOBSON, I (1999): *The Unified Modelling Language User Guide,* p.388

7.1.1 Additional reading

SCHMIDT D.C, STAL, M, ROHNERT, H AND BUSCHMANN, F (2000): *POSA2*: *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* has some references to further pattern automation literature.

FLORIJN, G, MEIJERS, M AND WINSEN, P. V. (1997): *Tool support for object -oriented patterns*. Proceedings of ECOOP, 1997 describes the design internals of how patterns can be represented inside software automation tools. www.serc.nl/people/florijn/work/patterns.html