Maplesden, D., Hosking, J.G. and Grundy, J.C. Design Pattern Modelling and Instantiation using DPML, In Proceedings of Tools Pacific 2002, Sydney, 18-21 February, 2002, CRPIT Press

Design Pattern Modelling and Instantiation using DPML

David Mapelsden, John Hosking and John Grundy
Department of Computer Science, University of Auckland,
Private Bag 92019 Auckland, New Zealand
{dmap001,john,john-g}@cs.auckland.ac.nz

Abstract

We describe the Design Pattern Modelling Language, a notation supporting the specification of design pattern solutions and their instantiation into UML design models. DPML provides a set of modelling constructs allowing design pattern solutions to be modelled and reused. A corresponding notation links design pattern solution elements to UML model elements, verifying fulfilment of the design pattern in the UML model. A prototype tool is described, together with an evaluation of the language and tool..

Keywords: design patterns, pattern instantiation, visual languages, tool support

1 Introduction

Design patterns are a method of encapsulating the knowledge of experienced software designers in a human readable and understandable form. They provide an effective means for describing key aspects of a successful solution to a design problem and the benefits and tradeoffs related to using that solution. Using design patterns helps produce good design, which helps produce good software.

Design patterns to date have mostly been described using a combination of natural language and UML style diagrams. This leads to complications in incorporating design patterns effectively into the design of new software. To encourage the use of design patterns we are investigating tool support for incorporating design patterns into program design. We describe the DPML (Design Pattern Modelling Language), a visual language for modelling design pattern solutions and their instantiations in object oriented designs of software systems.

We begin by describing previous work in design pattern tool support. We then overview DPML and describe its use in modelling design pattern solutions and pattern instantiation. We then describe an implementation and evaluation of DPML before describing future work.

2 Previous Work

Design patterns were popularised in the "Gang of Four" book of the same name (Gamma et al, 1994). A design pattern describes a common design solution to a programming problem, and they have become very widely used in object-oriented software development. Design patterns are typically described using a combination of natural language, UML diagrams and program code (Gamma et al 1994, Grand 1998). Such descriptions lack design pattern-specific visual formalisms, leading to pattern descriptions that are hard to understand and hard to incorporate into tool support. While the UML has become an industry standard for object modelling, it has inherent shortcomings for modelling design patterns. The proposed UML standard for modelling design patterns relies upon parameterised collaborations (Object Management Group, 2000). The main drawback is that because these are constructed using similar concepts to object models, they become simply prototypical examples of an object model. This does not allow enough freedom to model patterns effectively. There is no construct for modelling groups or sets of objects. Thus there is also no construct for mapping relations, such as associations or generalisations, between groups of objects. Several attempts have been made to develop new visual representations for design patterns.

The LePUS language (Eden et al 1998) uses higher order monadic logic to express solutions proposed by design patterns. Primitive variables represent the classes and functions in the design pattern, and predicates over these variables describe characteristics or relationships between the elements. LePUS also includes a visual notation for LePUS formulas consisting of icons (squares, ovals and triangles) that represent variables or sets of variables and annotated directed arcs representing the predicates. The main drawback with LePUS is that it is based on mathematics and formal logic, making it difficult for average software developers to work with and providing a weak basis for integrated tool support. Proposed tool support for LePUS is based on Prolog and lacks support for the visual notation. The notation defines many abstractions to make diagrams terse. Thus there are many different syntactic elements leading to diagrams that, while compact, are difficult to interpret. LePUS concentrates solely on defining design pattern structures, and has no mechanism for integrating instances of design patterns into program designs or code.

Florijn et al (1997) represent patterns as groups of interacting "fragments", representing design elements of a particular type (eg class, method, pattern). Each fragment has attributes (e.g. classname), and roles that reference other fragments representing pattern relationships, e.g, a class fragment has method roles referencing the method fragments for methods of that class. The fragments actually represent instances of

Copyright © 2002, Australian Computer Society, Inc. This paper appeared at the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia. Conferences in Research and Practice in Information Technology, Vol. 10. James Noble and John Potter, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included..

patterns. Pattern definitions are represented by prototype fragment structures; a one-level approach to defining patterns where the patterns. kept in a separate repository, are identical to the pattern instances in the fragment model. This system lacks support for the definition of design patterns and also lacks a strong visual syntax; visual aspects only provided by the tool interface. The single level architecture means patterns are only defined as prototypical pattern instances. We argue that concepts exist at the pattern level that do not at the pattern instance level, thus patterns can't be specified in the most general way using only prototypical instances.

Lauder and Kent (1998) propose an extension to UML to aid in "precise visual specification of design patterns". They use a 3-layer model with a visual notation for expressing models. The notation is an amalgam of UML and "Constraint Diagrams" a notation to visually specify constraints between object models elements. A second notation expresses object dynamic behaviour and can represent generalised behaviour of design patterns. We found their notation difficult; the differentiation between the diagrams at different levels was unclear and it seemed difficult to understand the reason why some abstractions were made at one level and not another.

Some approaches use textual rather than visual languages (Eden et al 1998, Florijn et al 1997, Reiss 2000, Hedin 1997). While these have good ideas and aspects, we are interested in a visual language for modelling design patterns. We are particularly interested in applying to design pattern modelling the approach UML (Bosch 1996, Sunye et al 2000) takes to object modelling.

3 Overview of DPML

DPML defines a metamodel and a notation for modelling design pattern solutions and solution instances within object models. It is important to stress that DPML can only be used to model the generalised *solutions* proposed by design patterns, not complete design patterns, that also contain information such as when the solution should be applied and consequences of using the pattern. A design pattern solution is a design pattern with information describing particular classes, methods and relationships used to realise the design pattern. A more abstract *design pattern* may not necessarily specify such particular implementation choices.

DPML is designed so that it can be used as a stand-alone modelling language for design pattern solutions or in conjunction with UML to also model solution instances within UML object models i.e. where in a UML model a pattern is used. A *design pattern solution instance* describes the relationship between design pattern solution elements modelled in DPML and system design elements modelled in UML. This allows tracking of the usage of a design pattern solution in a UML design and the validation of the usage of this pattern solution i.e. checking whether the pattern has been completely and consistently expressed in the UML design.

DPML supports incorporation of patterns at design-time, as opposed to implementation of design patterns during program coding (Wild 1996; Budinsky et al 1996). We feel design-time is the vital stage at which to include design patterns in he software engineering process, the assumption being that if design patterns can be effectively incorporated into the UML object model then converting the object model into code is, relatively speaking, straight-forward.

DPML has been developed specifically with automated tool support in mind. It is designed to be relatively easy to implement, particularly in conjunction with UML (Sunye et al 2000). We have carried out a detailed investigation into the implementation issues for the DPML and the processes that can be supported for working with the DPML.

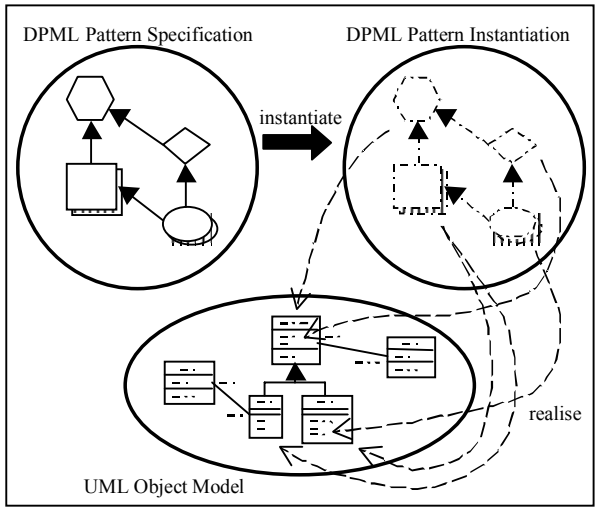

Figure 1: Core concepts of DPML.

The core concept of DPML (shown in figure 1) is a design pattern specification model which is used to describe the generalised structures of design patterns that are of interest to or useful to the user. This entails modelling the participants (interfaces, methods etc) involved in the pattern and the relationships (e.g. this interfaces declares this method) between them. Participants represent roles in the design pattern solution that program elements must play, and the relationships represent the constraints these elements must satisfy.. During the OO modelling process, where UML is used to create an OO model of a system, if a modeller sees an opportunity to use a design pattern previously defined, s/he can create an instance of that design pattern from the original definition. Once an instance of the pattern is created, the realisation process consists of either linking the participants in the design pattern to elements of the OO model, or creating new model members where required. The language's well-formedness rules define which members from the OO model are eligible for fulfilling each participant's role. In this way the user can be sure of creating a valid instance of the design pattern and so be sure of gaining the benefits of using the design pattern.

The design pattern instance model also allows each individual design pattern instance to be tailored. By default a design pattern instance contains members for all objects and constraints on these objects specified by the pattern definition, however certain parts of the pattern

may be relaxed or extended on a case by case basis allowing pattern instances that are variations on the base pattern.

4 Modelling Design Pattern Solutions

In DPML design pattern solution models are depicted using Specification Diagrams, the basic notation for which is shown in Fig. 2. DPML models design pattern solutions as a collection of participants; dimensions associated with the participants and constraints on the participants. A participant represents a structurally significant feature of a design pattern, that when instantiated, will be linked to objects from the object model to realise the pattern. Constraints represent conditions that must be met by the objects filling the roles of the participants in a design pattern instance for it to be considered a valid instance of the design pattern. Dimensions are constructs associated with participants to indicate that the participant potentially has more than one object linked to it in an instantiation. They indicate that a participant represents a set of objects in the object model, instead of just a single object.

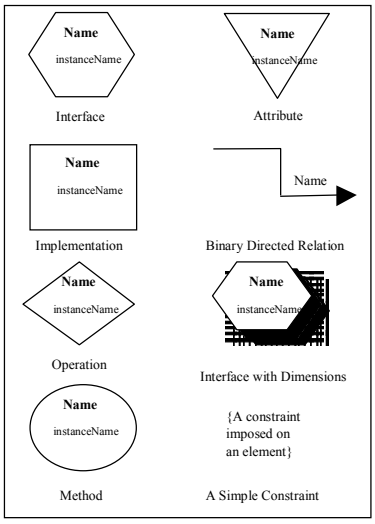

Figure 2: Basic DPML notation

Participants are interfaces, implementations, methods, operations or attributes. An interface (hexagon) represents a role that must be played by an object that declares some behaviour i.e. it exhibits an interface in the object model. In a traditional object model this means an interface or a class can fill an interface role as both declare a set of operations that provide behaviour. An implementation (rectangle) represents a role played by an object that defines or actually implements some behaviour. In a traditional object model an implementation would map to a class. The key concept with an implementation is that it defines no interface itself: its type or the declaration of its behaviour is defined entirely by the interfaces it is said to implement. This is different from the traditional concept of a class, which embodies both an interface and an implementation in the one object. This split is designed to allow a clearer definition of roles of objects in a design. A single object, in the case of a class, can play the roles of both an interface and an implementation.

An operation (diamond) is the declaration of a piece of behaviour while a method (oval) is the definition or implementation of a piece of behaviour. This mirrors the abstract/concrete split of interface/implementation. An attribute (inverted triangle) is a declaration of a piece of state held by an implementation. An attribute defines a role played by an attribute of a class in the object model.

Constraints are either simple constraints or binary directed relations. Simple constraints (plain text in curly brackets) define a condition specified in natural language to be met by the object bound to a single participant. Binary directed relations (lines with arrowheads) define a relationship between two participants, implying a relationship must exist between the objects in the object model playing the roles the participants define. The type of the binary directed relation determines the exact relationship that is implied. For example the 'implements' relationship between an implementation and an interface implies the object filling the role of the implementation must implement the object filling the role of the interface. Other examples of binary directed relations are *extends*, *realises*, *declared in*, *defined in* and *refers to*.

A more complex subclass of binary directed relations is the extended relations. These define the mappings of a binary directed relation between participants that have dimensions associated with them. These participants have sets of objects associated with them and therefore you need to specify how the base relation maps between the two sets of objects. There are four possible mappings: the relation exists between every possible pair of objects; exactly once for each object; for every object in one set but not necessarily the other; and only between one pair of objects. These respectively are a total, regular, complete and incomplete relations.

Dimensions specify that a participant can have a set of objects playing a role. The same dimension can be associated with different participants in a pattern and this specifies not only that these participants can have some multiple number of objects associated with them but that this number of objects is the same for both participants.

Consider modelling the Abstract Factory design pattern from (Gamma et al 1994) (Fig. 3). This pattern is used by designers when they have a variety of objects ("Products") which are subclasses of a common root-class to create. A set of "Factory" objects are used to create these related "Product" objects.

In this pattern there are six main participating groups of objects. The abstract factory interface declares the set of abstract create operations that the concrete factories will implement. This can be modelled by the DPML with an interface named *AbstractFactory* and an operation named *createOps*.

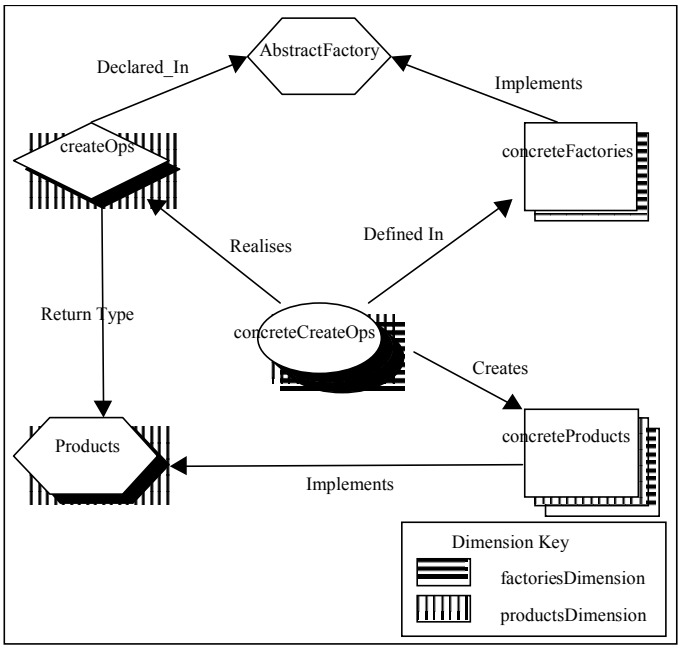

Figure 3: Example specification of Abstract Factory pattern using DPML.

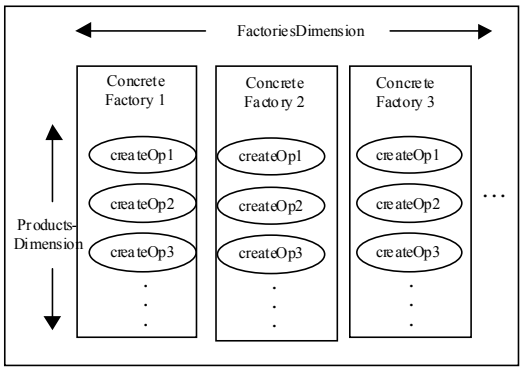

Figure 4. Object structure implied by DefinedIn relations associated with ConcreteCreateOps method

The *createOps* operation represents a set of operations so it has an associated dimension (*productsDimension*). There is also a complete *Declared_In* relation running from *createOps* to *AbstractFactory*. This relation implies that all methods linked to the *createOps* operation in an instantiation of the pattern must be declared in the object that is linked to the *AbstractFactory* interface.

In Abstract Factory the other groups of participating objects are the factory implementations, the method implementations that these factories define, the abstract product interfaces used by the abstract factory and the concrete products that the factories produce. These are modelled by a *concreteFactories* implementation, a *concreteCreateOps* method, a *Products* interface and a *concreteProducts* implementation respectively.

The *concreteFactories* implementation has a dimension, *factoriesDimension,* to indicate it represents a number of concrete implementations. A complete *Implements* relation runs from *concreteFactories* to *AbstractFactory*, implying all the *concreteFactories* must implement the *AbstractFactory* interface.

The *concreteCreateOps* method represents all methods from the set of *concreteFactories* that implement one of the sets of *createOps* so it is associated with both the *factoriesDimension* and *productsDimension* dimensions. It has a regular, complete *Defined_In* relation running from it to the *concreteFactories* implementation. This extended relation specifies that for every concrete factory there is a set of *concreteCreateOps* as shown in figure 4. A similar extended relation *Realises* runs from *concreteCreateOps* to *createOps*.

The *Products* interface has the *productsDimension* associated with it to imply there are a number of abstract product interfaces, and the same number of abstract *createOps* operations. A regular *Return_Type* relation runs from *createOps* to *Products*, implying each of the *createOps* operations has exactly one of the *Products* as its return type.

Finally the *concreteProducts* implementation has both *productsDimension* and *factoriesDimension* dimensions associated with it. This is because there is exactly one *concreteProduct* for each abstract product and concrete factory i.e. each concrete factory produces one concrete product for each abstract product interface. The *concreteProducts* implementation also takes part in an *Implements* extended relation with the *Products* interface and a *Creates* extended relation with the *concreteCreateOps* method. These specify that each *concreteProduct* implements one *Product* interface and that each *concreteProduct* is created (instantiated) in exactly one of the *concreteCreateOps* methods.

5 Design Pattern Instantiation and Realisation

A DPML *Instantiation Diagram* models design pattern solution instances and their realisation within object models. The design pattern instances are regarded as part of the object model, providing another construct that can be used in the description of a program. Every design pattern instance is derived from a design pattern solution. It is possible to have numerous instances of the same design pattern solution in an object model, each having its own identifying name and Instantiation Diagrams and connected to a different set of realising elements in the UML object model.

Instantiation Diagrams look similar to a Specification Diagrams; all of the basic symbols are the same shape, however the 'proxy' element icons are drawn with a dashed (or coloured) outline to distinguish them from

'real' elements as shown in Fig. 5. 'Proxy' elements are participants and relations inherited from the base design pattern solution and are therefore immutable in the instance; they can only be altered by changing the base design pattern solution. 'Real' elements are participants and relations added to a design pattern instance to tailor that instance. In this way we can ensure the basic structure of all instances of the same design pattern is the same, but still allow instances to be specialised for a particular task.

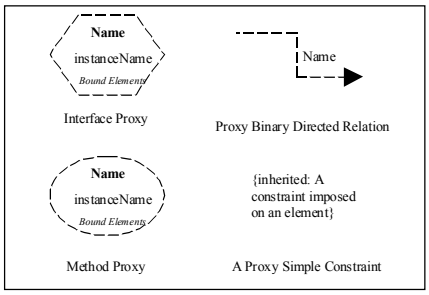

Figure 5: Instantiation Diagram Notation

In a design pattern instance every participant ('proxy' or 'real') has a link that maintains a binding from the participant to some number of UML model elements. Participants with no dimension can only be linked with a single UML model element; those with one or more dimensions can be linked with multiple UML model elements. The names of the UML model element(s) that a participant is linked to are shown in an italicised, textual annotation attached to each participant symbol in an Instantiation Diagram. If no model elements are linked to that participant then there is no annotation.

For example consider an instantiation of the Abstract Factory pattern. Assume we are implementing a GUI toolkit that allows programmers to create a GUI with windows, menus, icons, buttons etc. We wish to use the Abstract Factory pattern to allow the look and feel of the GUI to be changed at runtime. We declare a GUIFactory interface that has methods to construct all GUI elements required in our GUI toolkit. Different implementations of this GUIFactory create different sets of elements with different look and feel, allowing a change of look and feel by changing factories. A UML model for this design is shown in Figure 6.

If we assume there are no additions to the basic design pattern structure defined earlier in this instantiation then it is simply a matter of realising the design pattern by linking the participants from our design pattern instance to suitable objects in the UML model. This is done by the following steps:

- A design pattern instantiation model is made by copying the structure of the design pattern solution and substituting interface proxies, method proxies and binary directed relation proxies for pattern interfaces, methods and relations respectively.
- Each proxy in the design pattern instance model is linked to a UML design element. Some can be linked to multiple UML design elements e.g. createOps to createMenu, createButton etc.
- Once all design pattern instance elements are linked to one or more UML design elements, consistency checks are made: Are all pattern instance elements linked? Are correct kinds of pattern instance elements linked to UML design elements?

The final instantiation diagram is illustrated in Figure 7. The first instance element to UML design element link is the easiest, a link between the *AbstractFactory* interface in our design pattern instance and the *GUIFactory* interface in the UML model. Next we have a link between *concreteFactories* implementation and our two factory classes *SpaceFactory* and *MetalFactory*. We linked our *createOps* operation to the three methods declared in the *GUIInterface* interface namely, *createScrollBar*, *createMenu* and *createButton*. We linked our *Products* interface to the three interfaces *ScrollBar*, *Menu* and *Button*. The *concreteCreateOps* method is linked to the six implementing methods in our two factory classes: createScrollBar(): MetalScrollBar, createScrollBar(): SpaceScrollBar, createMenu():MetalMenu, createMenu():SpaceMenu, createButton():MetalButton, createButton ():SpaceButton. Similarly the *concreteProducts* implementation is linked to the six implementing product classes: MetalScrollBar, SpaceScrollBar, MetalMenu, SpaceMenu, MetalButton and SpaceButton.

6 Tool Support: DPTool

As mentioned earlier the DPML was designed specifically to facilitate the provision of tool support. We have successfully implemented a prototype DPML CASE tool we call DPTool. A screen dump showing the tool in use is shown in Fig. 8. The tool supports, among other things:

1. UML object models and the specification of these models using UML class diagrams.
2. DPML models and their specification using DPML specification and instantiation diagrams.
3. Consistent, multiple views of UML and DPML.
4. Model management mechanisms, so users can create, save and reload UML and DPML models.
5. An automatic design pattern instantiation mechanism, to create pattern instances from design patterns.
6. An automated consistency mechanism between design pattern instances and base design patterns.
7. An automatic model verification mechanism, which ensures the UML object model, and all design pattern instances within that object model, are correct If not, suitable error messages generated.
8. A pattern instance realisation support mechanism, which assists users in realising a design pattern instance by highlighting the valid model elements that can be bound to any given participant in an instance. This also allows users to create new UML model elements "on the fly" to be bound to a participant, if no suitable element exists in the UML model.

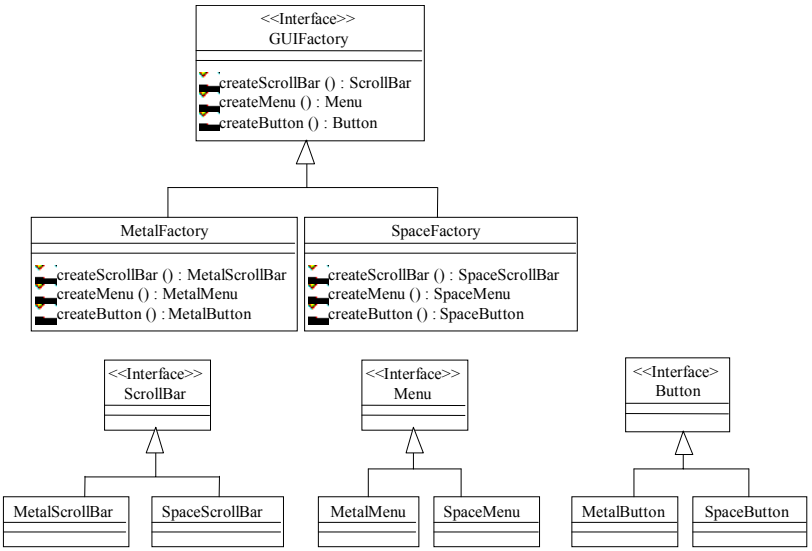

Figure 6. GUIFactory UML object model.

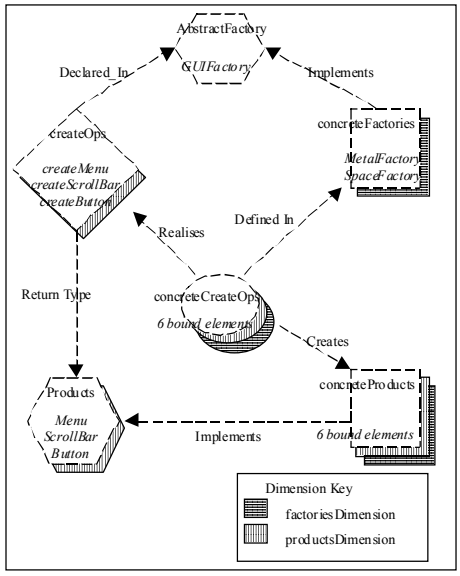

Figure 7. Instantiation of the AbstractFactory pattern.

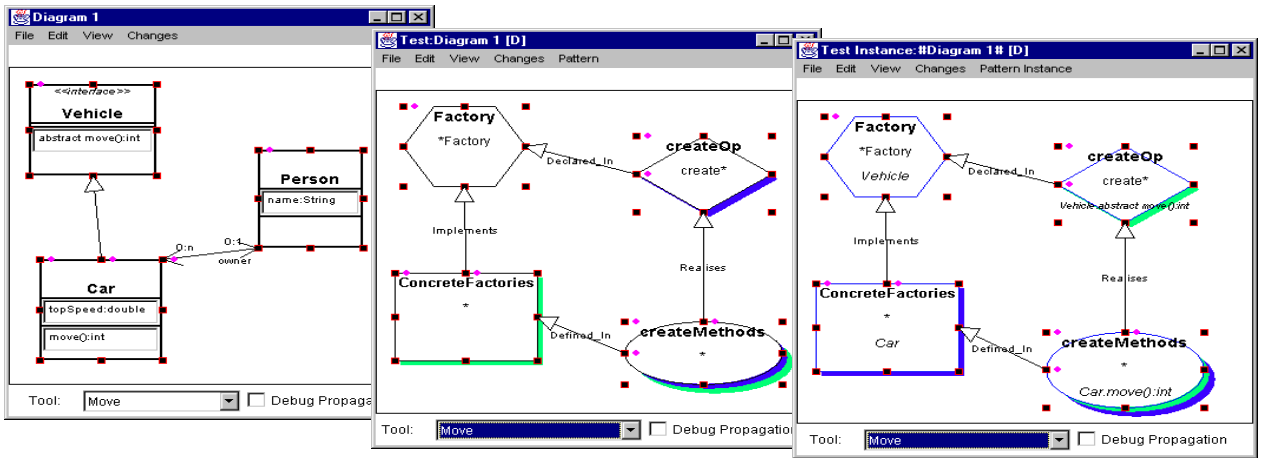

Figure 8: Three diagrams implemented in the prototype DPML tool: (left) A UML Class Diagram. (centre) A Design Pattern Specification Diagram. (right) A Design Pattern Instantiation Diagram. Note these diagrams use colour to convey information.

The designer creates pattern solution diagrams and UML design diagrams as required. Pattern instantiation diagrams are created when the user wants to either link existing UML design elements to design pattern elements they implement or wants to instantiate a design pattern with a partially complete UML design. After creation, the user specifies links between pattern instance elements and UML design elements, as illustrated in Figure 9. DPTool shows the designer all available UML design elements that can be linked to the design pattern instance proxies. The designer can link the pattern instance proxy to no UML model, can ask for a new UML design element to be created to which the pattern instance proxy is then linked, or can select from the list of items identified by DPTool as being valid UML design elements the pattern instance proxy can be linked to.

The designer may request DPTool to validate their UML design in terms of the usage of design patterns with the design as expressed by design pattern instance links to UML design elements. DPTool identifies incompleteness and inconsistency in the design pattern usage and reports this to the designer. Figure 10 illustrates an example of an error report generated by DPTool after attempting to validate a UML design. An example is also shown of DPTool's pattern and pattern instance manager, allowing designers to select patterns for instantiation from a catalogue. Pattern definitions are saved to a separate project file than UML designs and pattern instantiations allowing reuse of patterns across multiple design projects.

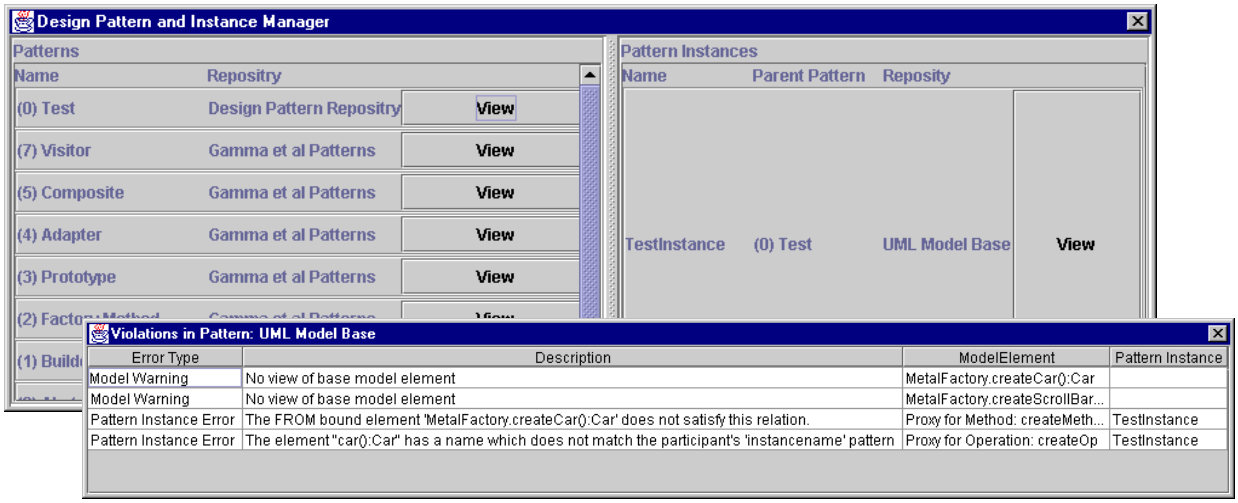

Figure 10. Pattern and instance manager and pattern validation manager examples.

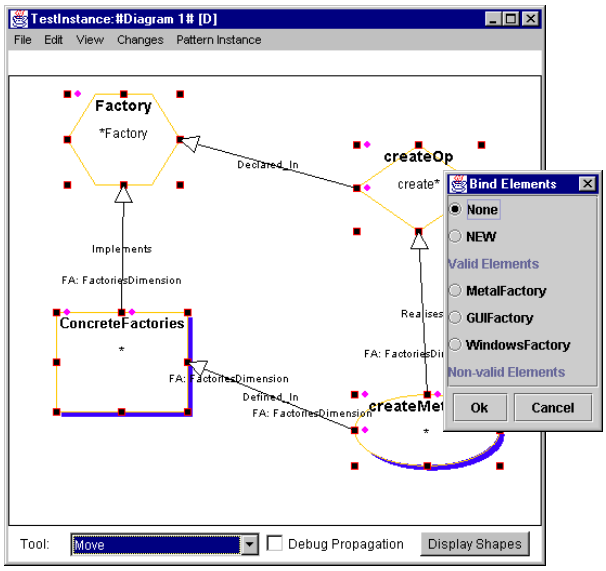

Figure 9. Example of binding pattern instance elements to UML model element(s).

7 Evaluation

We performed an evaluation of our tool and the DPML. This evaluation consisted of a user survey and a cognitive dimensions framework (Green and Petre 1996) evaluation. The user survey consisted of a tutorial for the DPML tool that the survey participants carried out and a number of open-ended questions for them to answer. The tutorial was a long and relatively detailed introduction to the tool that guided the users through tasks such as: creating a design pattern solution; creating a simple UML object model; instantiating the design pattern solution to create a design pattern instance; realising the instance within the UML object model; validating the object model and tracking any errors; and carrying out modifications to the design pattern after the instance has been created.

The survey questions covered two general topics. The first focussed on how intuitive the users found the concepts represented in the tool. Questions explored which concepts in the tool the users could and could not easily understand. The second group of questions focussed on the usability of the tool. They were designed to find the good and weak points of the tool, and which operations were easy to carry out and which were not.

The general survey response to the tool was quite enthusiastic. The majority of the language concepts were found to be easy to understand and use. In particular, the explicit separation between design patterns, design pattern instances and object models was easy to follow and effective in managing the use of design patterns. The design pattern definition process was also well understood. The almost universally mentioned difficult points were the dimension concepts. The inability to attach comments to model elements was highlighted as a weak point by several survey respondents.

The majority of the respondents found the tool effective to use for its primary tasks of creating and instantiating design patterns models. The pattern realisation (binding) process was mostly approved of, although it had some shortcomings, particularly in the visualisation of dimension categories. Respondents had a range of suggested extensions and improvements ranging from support for reverse engineering from object designs to design patterns (pattern abstraction) to inclusion of support for aspects. Most encouragingly all respondents thought that the tool was worth using if design patterns were a part of your design process.

The cognitive dimensions framework evaluation explored the full range of dimensions. Here we comment on some of the more important issues arising.

Abstraction Gradient. The DPML is a mixed abstraction system. There are few explicit abstractions in the notation, which mostly consists of combinations of primitive elements. However the starting level of the abstractions in the system is very high, i.e. the primitive elements in the system represent quite advanced abstractions in their own right.

Closeness of mapping. DPML models conceptual abstract design structures and their realisation in concrete OO models. It defines a set of concepts we felt designers most

commonly use to formulate these conceptual abstract designs, providing a language that closely matches their thinking. Most concepts are common to OO modelling. Two concepts, implementation and dimension, are not commonly used, however they provide more useful abstractions than the more common class and set.

Consistency. The DPML is consistent. Dimensions are denoted in a consistent way on all participants, be they methods or interfaces or in patterns or pattern instances. Participant shape is consistent for both patterns and pattern instances e.g. interfaces represented by rectangles and methods by ovals. The consistent treatment of methods and operations as full participants similar to interfaces and implementations has advantages over the specialised treatment they receive in many other languages.

Diffuseness/Terseness. Generally DPML is more diffuse than other visual languages for design patterns such as Kent and Lauder's (1998) or LePUS (Eden et al 1998). Both of these languages use additional abstractions or more detailed notations resulting in terser, smaller, diagrams.

Hard Mental Operations. We have reduced the need for hard mental operations in DPML, by avoiding some of the dense, terse notation of other languages. The dimension concept, that is mostly unfamiliar to new users of the language, can be difficult to understand. This does not fall into the category of a 'hard mental operation' however because in our experience once familiarity with the concept is gained it is easy to apply and the layering of dimensions does not overly increase their complexity.

Hidden Dependencies. The design pattern to design pattern instance link is a hidden dependency; it is not possible to tell explicitly from a design pattern the design pattern instances related to it. Also, it is also not possible to tell from a UML model element which roles it is playing, if any, in design pattern instances.

Progressive Evaluation. The automatic model verification mechanism, which locates errors in a model being developed, can be run at any stage in development.

Role-Expressivenes. Every object in a diagram has an obvious shape, denoting its type, and a prominent name, to further clarify the role it is playing in the diagram. The model/view separation and multiple view support makes it possible to create modularised models, with each view of the model displaying a single related group of entities from the model. The view can be given a relevant name indicating the role the group of entities has in the model.

*Secondary Notation and Escape from Formalism.*The potential for secondary notation is high. Layout is not constrained in any way, naming conventions can easily be used and multiple views can be used to break models up into logical pieces. Escape from formalism is not so well supported. The ability to add free text to model elements as a form of documentation is essential, as is the ability to add free text and annotations to diagrams in general, but are not yet supported in our tool.

Visibility and Juxtaposability. The DPML tool suffers from poor visibility in one notable area; the indication of bound elements for design pattern instance participants. While it is easy to display a list of the bound elements for any one participant, it is not possible to display the list of bound elements for more than one participant at a time. Also the lists are just lists of elements; there is no visual representation of the realisation links between UML model elements and the pattern instance participants. In other areas DPML has good visibility. DPML has good juxtaposibility with the ability to have multiple views open side by side, displaying different parts of the same model or related parts of different models.

9. Summary

DPML is a visual language for modelling design patterns and their instances. We feel DPML offers several benefits over other modelling languages for design patterns and as a basis for tool support for design patterns.

The ability to work with design patterns in conjunction with UML is a major benefit. UML is now a standard for OO modelling and its use is widespread. Compatibility with UML makes our approach more palatable for many programmers and designers as they are already familiar with UML, and many of the concepts in the DPML are built upon or similar to concepts in the UML. We found the meta-modelling approach to the definition of the DPML particularly useful. Not only did this approach give us an avenue for defining the formal semantics of the language, but also we found it very sympathetic to the implementation of tool support, we were able to create a large and sophisticated tool implementing the language in a very short period of time.

The DPML's multi-level approach with design patterns, design patterns instances and object models is effective as is the basic premise of describing pattern structures in terms of participating objects and the relations between those objects. We avoided introducing unnecessary abstractions into the language, as these enlarge the language and introduce abstract structures, which, when combined together, are difficult to interpret. One of the novel concepts in the language is that of dimensions. We feel there is much promise in use of this concept in place of sets. It offers a more accurate description of the principles involved and offers a more sophisticated insight into the pattern structure.

There are a number of concerns that we did not have time to address in this initial work, but are certainly important issues for the future.

- Support for design pattern composition to create other design patterns. This could be used to support design pattern hierarchies or pattern languages.
- Support for specification of dynamic aspects of design patterns.
- The introduction of an OCL-style constraint language, for more clearly expressing constraints.
- Better tracking of the overlapping of design pattern instances in object models, particularly visualisation

of the roles a UML element is playing in different design pattern instances.

- Better visualisation of the bindings that occur during the pattern realisation process. A 3-dimensional notation for expressing this might well be appropriate.
- Support for a classification scheme for the design patterns, to assist in deciding when to apply them.
- Extending the pattern concept (as used in the tool) to allow a greater range of patterns e.g. architectural patterns or code level idioms.

8 Acknowledgements

Support for this research from the New Zealand Public Good Science Fund is gratefully acknowledged. David Mapelsden was supported by a William Georgetti Scholarship.

9 References

BOSCH, J. (1996): Language support for design patterns, In *Proceedings of TOOLS Europe '96*, pg 197-210, Prentice-Hall.

BUDINSKY, F.J., FINNIE, M.A., VLISSIDES, J.M., YU, P.S. (1996): Automatic code generation from design patterns, In *IBM Systems Journal* **35** (2).

EDEN, A.H., HIRSHFELD, Y., YEHUDAI, A. (1998): *LePUS – A declarative pattern specification language*, Technical report 326/98, department of Computer Science, Tel Aviv University, (1998).

FLORIJN, G. MEIJERS, M. VAN WINSEN, P. (1997): Tool support for object-oriented patterns, In *Proceedings of the 11th European conference on Object Oriented programming*, Springer LNCS 1241, pg 472-495.

GAMMA, E., HELM, R., JOHNSTON, R. AND VLISSIDES, J. (1994): *Design Patterns*, Addison-Wesley.

GRAND, M. (1998): *Design patterns and Java*, Addison-Wesley.

GREEN, T.R.G. AND PETRE, M. (1996): Usability analysis of visual programming environments: a 'cognitive dimensions' framework, *Journal of Visual Languages and Computing* **7**, pg 131-174.

HEDIN, G. (1997): Language support for design patterns using attribute extension, In *Proceedings of ECOOP '97*, Springer LNCS 1357, pg 137-140.

LAUDER, A., KENT, S. (1998): Precise Visual Specification of Design Patterns, In Proceedings of the 12th *European conference on Object Oriented programming*, LNCS 1445, pg114-134.

OBJECT MANAGEMENT GROUP (2000): Unified Modeling Language (UML) Specification v1.3, Document formal/00-03-01, available from http://www.omg.org.

REISS, S.P. (2000): Working with patterns and code, In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, (Abstracts) pg 243.

SUNYÉ, G., LE GUENNEC, A., JÉZÉQUEL, J-M. (2000): Design patterns application in UML, In *Proceedings of the 14th European conference on Object Oriented programming*, Springer LNCS 1850, pg 44-62.

WILD, F. (1996): Instantiating code patterns, *Dr. Dobb's Journal*, pg 72-76.