Chapter 2 in Design Patterns Formalization Techniques, Toufik Taibi (Ed)
(c) Idea Group Inc., Hershey, USA, March 2007.

A Visual Language for Design Pattern Modelling and Instantiation

David Maplesden
Orion Systems Ltd
Mt Eden, Auckland, New Zealand
Phone: +64 9 638 0600
Fax: +64 9 638 0699
Email: David.Maplesden@orion.co.nz

John Hosking
Department of Computer Science
University of Auckland,
Private Bag 92019 Auckland, New Zealand
Phone: +64 9 3737599
Fax: +64 9 3737453
Email: john@cs.auckland.ac.nz

John Grundy
Department of Computer Science and Department of Electrical and Computer Engineering
University of Auckland,
Private Bag 92019 Auckland, New Zealand
Phone: +64 9 3737599
Fax: +64 9 3737453
Email: john-g@cs.auckland.ac.nz

A Visual Language for Design Pattern Modelling and Instantiation

Abstract

In this chapter we describe the Design Pattern Modelling Language, a notation supporting the specification of design pattern solutions and their instantiation into UML design models. DPML uses a simple set of visual abstractions and readily lends itself to tool support. DPML design pattern solution specifications are used to construct visual, formal specifications of design patterns. DPML instantiation diagrams are used to link a design pattern solution specification to instances of a UML model, indicating the roles played by different UML elements in the generic design pattern solution. A prototype tool is described, together with an evaluation of the language and tool.

Keywords: *CASE tools, Language Constructs, Modeling Languages, Special Purpose Languages, Class diagram, Meta Model, Object-Oriented Design*

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 (Gamma et al, 1994).

Design patterns to date have mostly been described using a combination of natural language and UML-style diagrams or complex mathematical or logic based formalisms which the average programmer find difficult to understand. This leads to complications in incorporating design patterns effectively into the design of new software. To encourage the use of design patterns we have been developing 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 have developed two prototype tools, DPTool and MaramaDPTool, realising DPML and integrating it within the Eclipse environment. Significant contributions of this work include the introduction of *dimensions* as a proxy for collections of like design pattern participants and the instantiation of patterns into designs rather than directly into code. These both fit naturally with model driven design approaches.

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 discuss two prototype tools we have developed to support the use of DPML, together with an evaluation of their usability. We then discuss in more detail the rationale and implications of the design choices we have made in designing DPML and the potential for more general applicability of some of those design features before summarising our contributions.

PREVIOUS WORK

Design patterns, which describe a common design solution to a programming problem, were popularised by the seminal "Gang of Four" book (Gamma *et al*, 1994) and Coplien's *Software Patterns*

(Coplien, 1996). Design patterns have become very widely used in object-oriented software development, and their influence has spread to areas of software development other than design, such as the development of analysis patterns (patterns in the analysis phase) and idioms (language specific programming patterns). Design patterns are typically described using a combination of natural language, UML diagrams and program code (Gamma et al, 1994; Grand, 1998). However, such descriptions lack design pattern-specific visual formalisms, leading to pattern descriptions that are hard to understand and hard to incorporate into tool support. The UML standard for modelling design patterns relies upon UML profiles and the UML meta-model (Object Management Group, 2006). This presents difficulties for modelling design patterns, particularly because they are constructed using similar concepts to object models and hence are simply prototypical examples of that object model. This does not allow enough freedom to model patterns effectively. Design pattern representations look like existing UML models and linking pattern elements to standard UML elements is often not supported. Several attempts have been made to improve design pattern representation with UML (e.g. Mak et al, 2004; Fontoura et al, 2002; Guennec et al, 2000) but all use conventional UML diagram representations or minimal extensions. Stereotypes and related approaches to delineating patterns makes the diagrams considerably more complex and discerning pattern elements from standard UML elements is difficult.

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.

There have been a number of approaches proposed for alternative visual representations for design patterns. LePUS (Eden, 2002) uses a textual 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. A visual notation for LePUS formulae consists of icons that represent variables or sets of variables and annotated directed arcs representing the predicates. LePUS' basis in mathematics and formal logic makes it difficult for average software developers to work with and provides a weak basis for integrated tool support, being well removed from typically used design and programming constructs. LePUS tool support is based on Prolog and lacks support for the visual mean that while diagrams are compact, they are difficult to interpret with a high abstraction gradient. LePUS concentrates solely on defining design pattern structures, and has no mechanism for integrating instances of design patterns into program designs or code. Mak et al (2003) have proposed an extension to LePUS which addresses pattern composition.

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 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 approach lacks support for the definition of design patterns and also a strong visual syntax. 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, i.e. you cannot specify all patterns in the most general way using only prototypical instances. Their approach to pattern definition also has no formal basis. It is limited to defining patterns only relative to Smalltalk programs represented in the fragment model. We feel it would be advantageous to define an exact unambiguous meaning for the pattern representation in use so that it can be discussed without confusion and applied appropriately to a range of programming languages.

RBML (Kim et al, 2003; France et al, 2004) adopts a similar to approach to ours, and has been influenced by initial work we have presented in this area (Mapelsden et al, 2002). It uses a meta-modelling approach to the specification of pattern representation, extending the UML metamodel to achieve this. They place more emphasis on behavioural specification than we have in the development of DPML, which has focussed more on structural representations of patterns. They also adopt a cardinality approach to specification of multiplicities as opposed to our dimension concept. The tradeoffs involved are discussed further in Section 0.

Some approaches use textual rather than visual languages (e.g. Reiss 2000; Sefika et al, 1996; Taibi and Ngo, 2003). While these present useful concepts, our interest is in a visual language for modelling design patterns. Domain-specific visual languages like DPML offer a higher level of abstraction and representation, particularly for design-level constructs. We are also particularly interested in applying to design pattern modelling the approach UML (Object Management Group, 2006) takes to object modelling, i.e. providing a common formalism that is accessible to the average designer/programmer, while abstracting away from lower levels of design.

OVERVIEW OF DPML

DPML defines a metamodel and a notation for specifying design pattern solutions and solution instances within object models. The metamodel defines a logical structure of objects, which can be used to create models of design pattern solutions and design pattern solution instances, while the notation describes the diagrammatic notations used to represent the models visually. It is important to stress that DPML can only be used to model the generalised *solutions* proposed by design patterns, not complete design patterns. A complete design pattern also contains additional information such as when the solution should be applied and the consequences of using the pattern.

DPML can be used as a stand-alone modelling language for design pattern solutions or, more commonly, in conjunction with UML to model solution instances within UML design models i.e. whereabouts in the UML model the pattern is used and the various bindings that result from that usage. DPML supports incorporation of patterns into a UML model at design-time, rather than instantiation directly into program code. We feel design-time is the vital stage at which to include design patterns in the 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.

There were three primary goals for the development of the DPML. Firstly to provide an extension to the UML so that design patterns could be raised to first class objects within the modelling process. Secondly to provide for design patterns some of the same benefits that the UML provides for object oriented modelling: a common language to facilitate the exchanging of models; a formal basis for the understanding of such models; and a basis to facilitate the provision of automated tool support for modelling. By raising design patterns to first class objects in the design process the DPML allows design patterns to become an integral part of the design process. Thirdly, we have aimed for a formalism to express design patterns that is *accessible* to typical programmers. Our aim is for sufficient formalism to provide a robust representation, while avoiding complex mathematical formalisms that restrict the use of our approach to a very small set of mathematically inclined programmers. This is a similar approach to formalism as has been taken in the development of UML. We feel that by providing an easy-to-use yet powerful method for creating and instantiating design pattern solutions, designers will be encouraged to think at higher levels of abstraction about the problems they are facing and come up with more general, re-usable solutions which can, in an iterative manner, be abstracted and then encapsulated in a design pattern for future use.

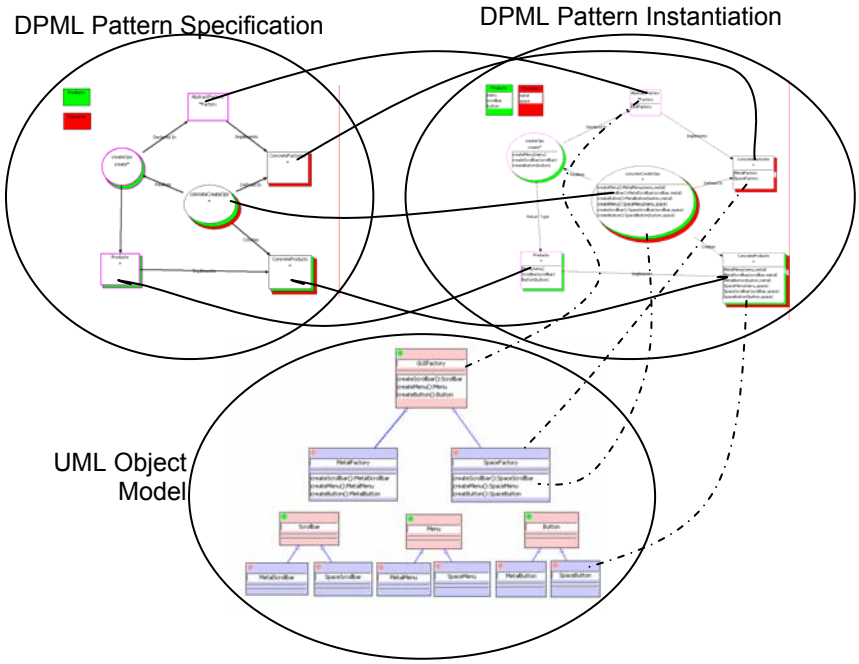

Figure 1: Core concepts of DPML.

The secondary goals of the DPML include firstly, a common standard language for the definition of design patterns that will allow patterns to be exchanged amongst designers in a more readily accessible manner than written text and diagrams, hopefully spreading useful design abstractions and robust designs and improving program design and secondly, providing a basis for tool support for design patterns. 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. 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 and developed two proof-of-concept tool implementations.

The intended uses of the DPML then is to capture areas of good design within an OO model in a design pattern and then reuse them within the same and different models. By specifying the constructs in the design pattern that capture the essential parts of the good design you can ensure that these elements exist every time the design pattern is employed. Indeed by encouraging more experienced designers to create the design patterns to be used by less experienced designers in their work the DPML provides a practical way to encapsulate and reuse the expertise of good designers.

The core concept of DPML (as shown in Figure 1) is that a design pattern specification model is used to describe the generalised design 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 between them. The user can then use the UML to create an object oriented (OO) model of a system that they are interested in or developing. During the OO modelling process, if the user sees an opportunity to use a design pattern that they have previously defined, they can create an instance of that design pattern from the original definition. The instantiation process consists of linking the roles of the elements in the design pattern with members from the OO model, or creating new model members where required. The well-formedness rules at this stage define which members from the OO model are eligible for fulfilling each 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. This recognises the fact that pattern instantiation often involves small adaptations of the pattern to suit the particular context it is being applied to (Chambers et al, 2000).

MODELLING DESIGN PATTERN SOLUTIONS

Pattern Specification

In DPML, design pattern solution models are depicted using Specification Diagrams, the basic notation for which is shown in Figure 2. It should be emphasised that the surface syntax is relatively unimportant (and we have used at least two variants of this in our work). In addition, in our MaramaDPTool the surface appearance can be altered by the user by use of a meta-tool designer. Of much more importance is the abstract metamodel which is described as a UML class diagram in Figure 3. 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.

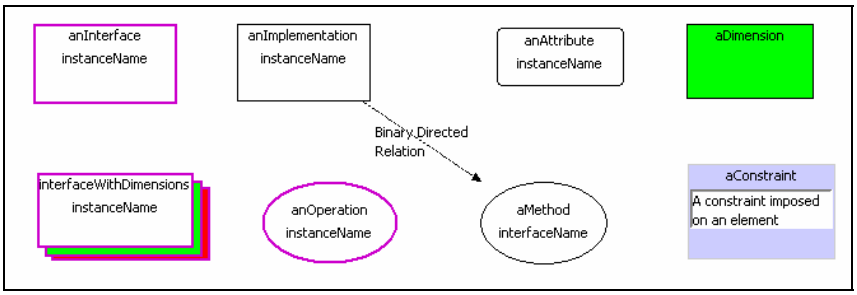

Figure 2: Basic DPML notation

Participants can be interfaces, implementations, methods, operations or attributes. An interface (lighter and thicker bordered rectangle) represents a role that must be played by an object that declares some behaviour i.e. it exhibits an interface or signature in the object model. In a traditional UML class 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 (darker, thinner bordered rectangle) represents a role played by an object that defines or actually implements some behaviour. In a conventional UML 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 the participants in a design. Modellers can specify precisely whether an object is intended to be a declaration of type, an interface, or a definition of behaviour, an implementation. A single object, in the case of a class, can play the roles of both an interface and an implementation.

A relationship similar to the one between interfaces and implementations exists between operations and methods. An operation (lighter thicker bordered oval) is the declaration of some form of behaviour while a method (darker thinner bordered oval) is the definition or implementation of that behaviour. An operation represents a role that must be played by an object in the object model that declares a single piece of behaviour e.g. it can be played by a method or an abstract method in a conventional UML model. A method can only be played by an object that actually defines behaviour and so must be played by a concrete UML method. An attribute (rounded rectangle) is a declaration of a piece of state held by an implementation and defines a role played by a class attribute in the UML model. Constraints are either simple constraints or binary directed relations. Simple constraints (plain text inside a grey box) define a condition specified in either natural language or OCL 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 each of 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*, *creates, declared in*, *defined in*, *return type* and *refers to*.

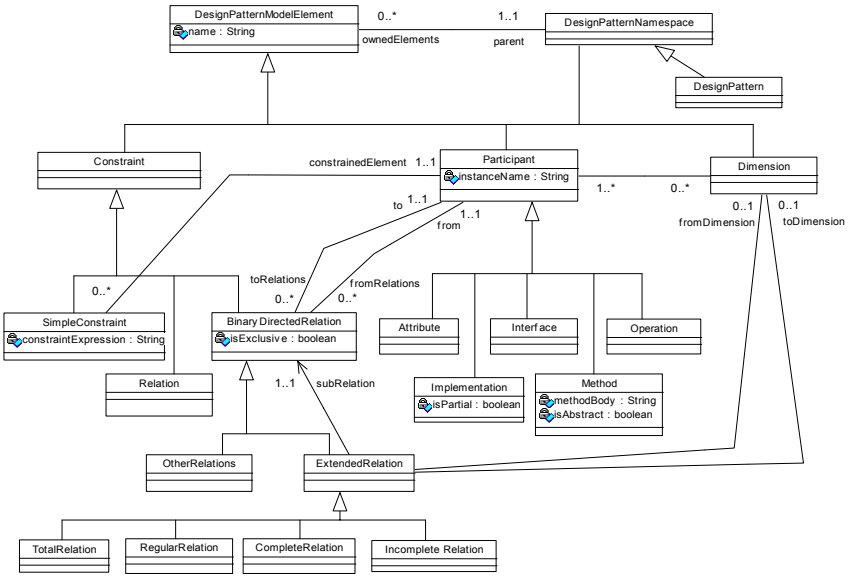

Figure 3: DPML Specification Diagram metamodel

A more complex subclass of binary directed relations is the set of extended relations. These define the mappings of a binary directed relation between participants that have dimensions associated with them. Because these participants have sets of objects associated with them we need to specify how the base relation maps between the sets of objects involved. There are four possible mappings: the relation exists between every possible pair of objects (a *total* relation); exactly once for each object (a *regular* relation); for every object in one set but not necessarily the other (a *complete* relation); and only between one pair of objects (an *incomplete* relation). Extended relations are more fully specified as expressions of the form *extendedRelationName(fromParticipantDimension, toParticipantDimension, subRelationSpecification)*. An example later in this section will illustrate this further.

Dimensions (indicated by a coloured rectangle and coloured shading of participant icons to indicate they are associated with a dimension) 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.

Pattern Specification Examples

Consider modelling the Abstract Factory design pattern from (Gamma et al, 1995) (Figure 4). 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*. The *createOps* operation represents a set of operations so it has an associated dimension (*Products*) since there is one operation for each abstract product type we want to create. 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. The *Products* interface has the *Products* dimension associated with it to imply there is the same number of abstract product interfaces, as there are 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.

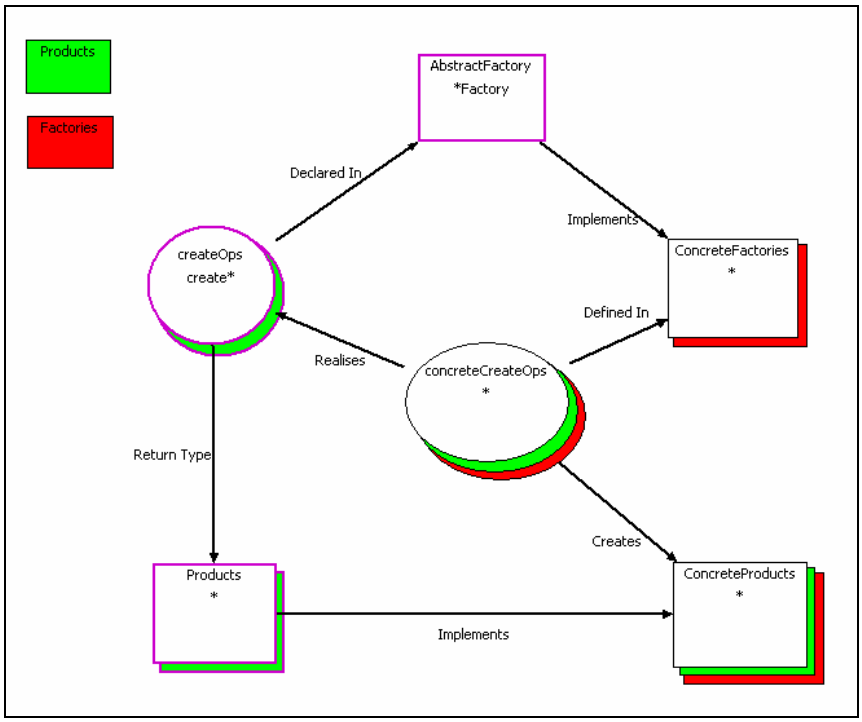

Figure 4: Example specification of Abstract Factory pattern using DPML.

The above set of participants defines the abstract part of the Abstract Factory pattern. The other set of participants define the concrete part of the pattern: the factory implementations, the method implementations that these factories define, and the concrete products that the factories produce. These are modelled by a *concreteFactories* implementation, a *concreteCreateOps* method, and a *concreteProducts* implementation respectively.

The *concreteFactories* implementation has a dimension, *Factories,* to indicate it represents a number of concrete implementations, one for each type of Factory implemented. 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 *Factories* and *Products* dimensions. It has a regular relation with a complete sub relation that has a *Defined_In* sub relation running from it to the *concreteFactories* implementation. This extended relation sounds complicated but it implies simply that for every concrete factory there is a set of concreteCreateOps that it defines, one member of that set for each *Product*. This can be stated in a more compact expression form, as:

regular(Factories, Factories, complete(Products, , Defined_In))

where we see that the regular relation is associated with the *Factories* dimension, ie for each of the Factories the complete subrelation holds. This subrelation is between the *Products* dimension on the

concreteCreateOps side and no dimension on the *concreteFactories* side and specifies that every *concreteCreateOp* associated with a Product is defined in each *concreteFactory*.

Similarly there is a regular relation with a complete sub relation which has a *Realises* sub relation running from *concreteCreateOps* to *createOps* which implies that for every *createOps* operation there is a set of methods in *concreteCreateOps* that realise it (one in each concrete factory). This can be stated in a more compact expression form, as:

regular(Products,Products , complete(Factories, , Realises).

The overall effect of the two extended relations can be seen in Figure 5 which shows the object structure implied by them.

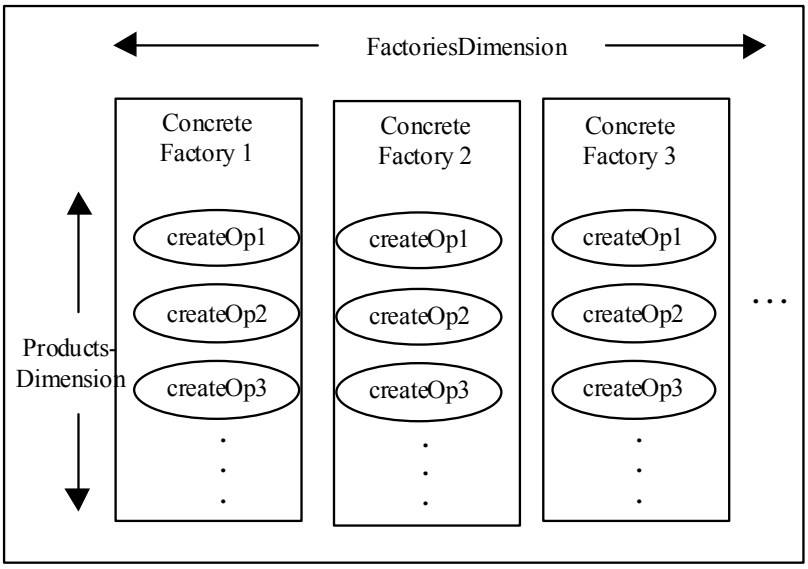

Figure 5: Object structure implied by Defined In and Realised relations associated with ConcreteCreateOps method

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 (regular, complete, Impelments) with the *Products* interface and a *Creates* extended relation (regular, regular, Creates) 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.

As can be seen, the full name of a relation in expression format can be long and can clutter the diagram when the base relation is the important part. So for ExtendedRelations just the name of the base relation can be used on the diagram to improve readability of the diagram, this means the full ExtendedRelation needs to be specified elsewhere (in our proof of concept tools this is specified in a property window). Generally the type of the ExtendedRelation can be deduced from the diagram because relations tend to follow common patterns. Usually (but not always) a relation between two

Participants with the same Dimension will be a RegularRelation and between a Participant with a Dimension and a Participant without a Dimension will be a CompleteRelation. Occasions when TotalRelations and IncompleteRelations are used are much less common and it is advisable in these cases to show the full ExtendedRelation name in the diagram.

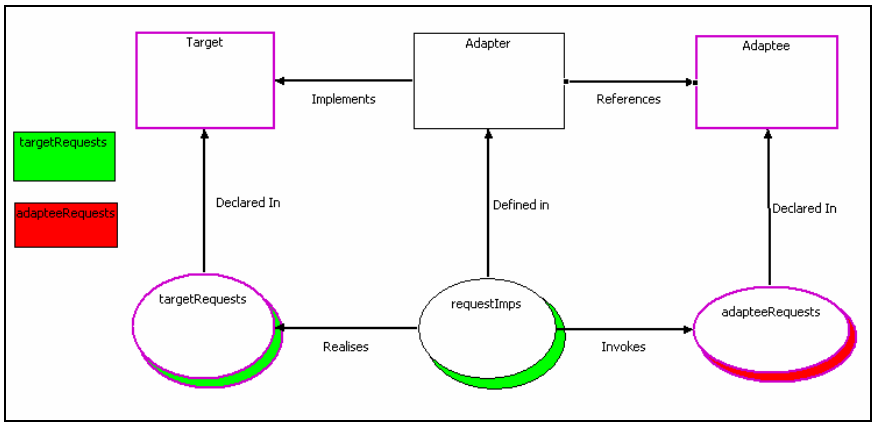

Figure 6: Specification Diagram for the Adapter Pattern

Figure 6 shows another specification diagram, this time for the Adapter Design pattern, also from (Gamma et al, 1995). This specifies how an *Adapter* implementation can be used to map operations specified by a *Target* interface to an *Adaptee*, which has different signatures for its operations. Here we see that the *Target* interface declares a set of *targetRequests*, the set being associated with the *targetRequests* dimension. The *Adapter* implements *Target's* interface, in the process defining a set of *requestImps* methods, also associated with the *targetRequests* dimension. A regular *Realises* relation between *requestImps* and *targetRequests* specifies that each *targetRequests* operation is realised by one *requestImps* method. The *Adapter* has a *References* relation with the *Adaptee* interface. The *Adaptee* declares a set of *adapteeRequests* one for each member of the *adapteeRequests* dimension. The *Invokes* relation between *requestImps* and *adapteeRequests*, indicates each of the *requestImps* may invoke one or more of the *adapteeRequests*.

Behavioural Specification

In designing DML, we have concentrated on structural specification. However, more dynamic aspects, such as method calling mechanisms, can be represented using an extended form of UML sequence or collaboration diagram. Figure 7 shows an example sequence diagram for the Adapter pattern. This uses standard sequence diagram notation, but includes participants, acting as proxies for the final bound objects. Dimensions, as is the case in the specification diagram, are represented using coloured shading, in this case, the *targetRequests* and *requestImps* invocations are annotated with the *targetRequests* dimension colour and the *adapteeRequests* invocation is annotated with the *adapteeRequests* dimension colour. This component of the formalism needs further development. In particular, in should be possible to use dimensions to indicate looping constructs with a similar set of invocation relationships having a similar set of extended relationship variations as for the structural diagrams. This remains as further work.

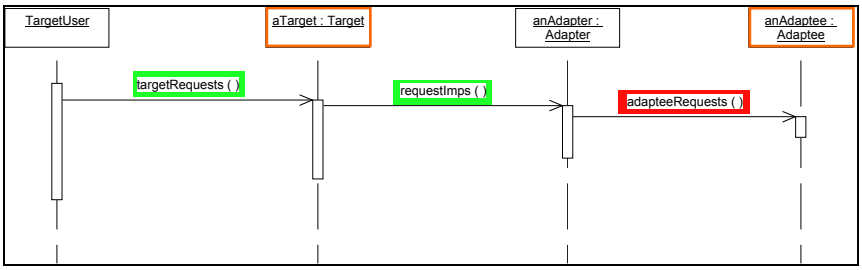

Figure 7: Behavioural Specification for Adapter Pattern

Pattern Instantiation

Instantiation Diagrams provide a mapping from the pattern specification to its realisation in a UML design model. In addition, as mentioned earlier, the instantiation process can adapt a pattern solution through addition of extra participants or modification of existing participants, an important part of our DPML design. Pattern solutions are rarely instantiated directly, the instantiation almost always involves some measure of adaptation or refinement. Accordingly, Instantiation Diagrams have a very similar look and feel to Specification Diagrams. The basic notation is very similar and, in fact, all of the modelling elements (except addition of new dimensions) that are used in Specification Diagrams may also be used in Instantiation Diagrams. These modelling elements are used to model the pattern adaptations. In addition, however, an Instantiation Diagram also includes *proxy* elements (which will typically be the majority of the diagram's elements). These represent the original participant specifications in an instantiated pattern, but are elaborated with information about the actual UML design elements that they are bound to in this particular instantiation of the pattern. Figure 8 shows the proxy element notation. As can be seen, the syntactic form is similar to that of the specification diagram equivalents. In the case of interface, implementation, operation, method, and relation proxies, they differ from the originals by having dashed borders or lines, and lists of bindings. For constraints, an "inherited" keyword precedes the constraint expression, and for dimensions, a list of the names of the category bindings for that dimension.

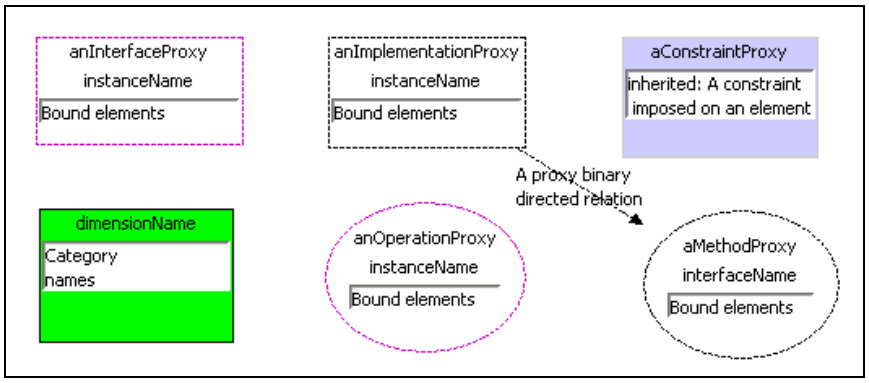

Figure 8: Additional notational elements for Instantiation Diagrams

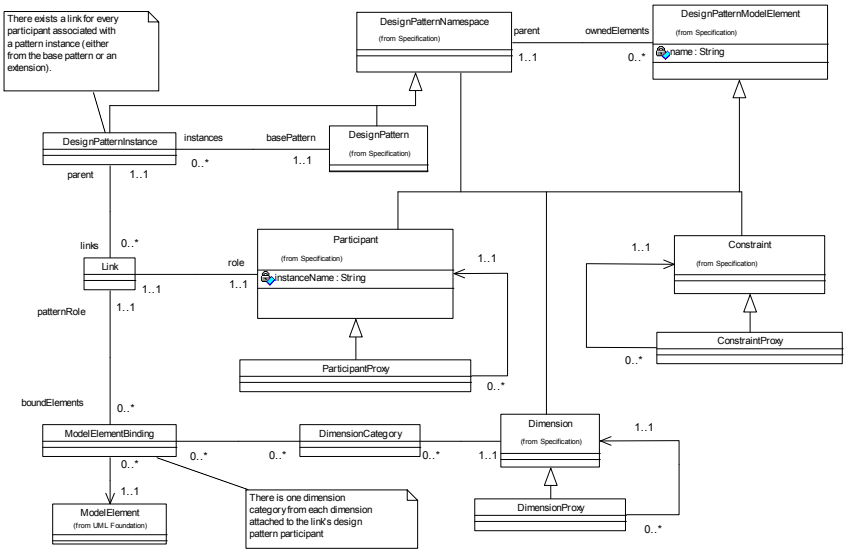

Figure 9: DPML Instantiation Diagram metamodel

Figure 9 shows the metamodel for Instantiation Diagrams. In a DesignPatternInstance (ie the model for an Instantiation Diagram) every Participant (whether they are a 'proxy' or 'real') has a Link associated with it that maintains a binding from the Participant to some number of UML model elements in an object model (*UMLModelElement* is part of the UML Foundation Package). The names of the bound UMLModelElements are those displayed in the bound elements lists in the proxy participant icons. When model elements are bound to Participants with Dimensions each model element is associated with a DimensionCategory for each Dimension. These DimensionCategories are specified as a simple list of their names which are displayed in the Dimension proxy. The number of DimensionCategories for a Dimension establishes the 'size' of the Dimension for the Instance.

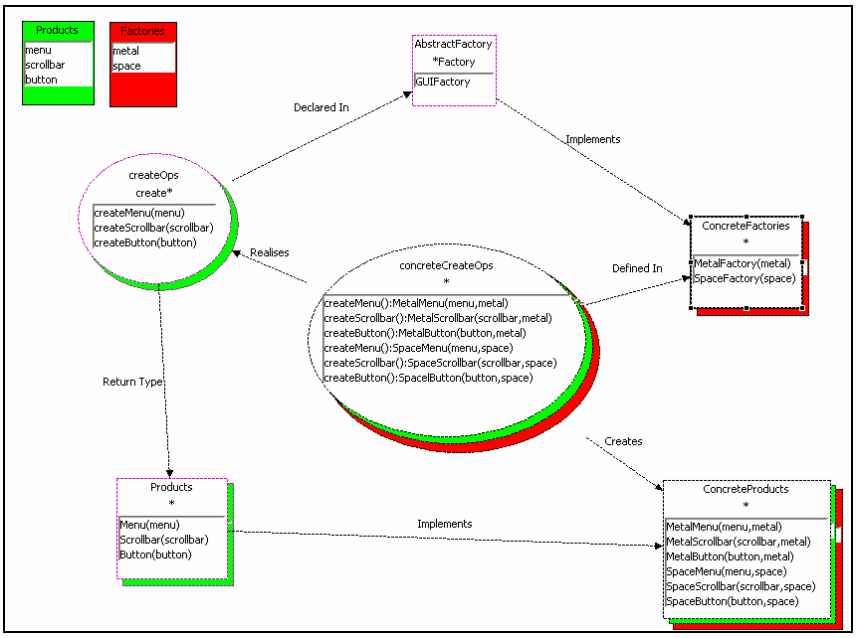

Figure 10: Instantiation Diagram for GUIFactory

Pattern Instantiation Example

As an example, consider the Instantiation of the Abstract Factory pattern shown in Figure 10. Assume we are implementing a GUI toolkit that allows programmers to create a GUI with windows, menus, icons, buttons etc which users can change the look and feel of at runtime and we want to use the Abstract Factory design pattern to do this. The Instantiation Diagram illustrates the bindings for this. The *GUIFactory* UML interface is bound to the AbstractFactory interface participant and this interface is implemented by two *ConcreteFactories*, *MetalFactory* and *SpaceFactory*, one for each dimension category (metal, space) in the Factories dimension (the relevant dimension category name is shown in brackets after the bound element name). *GUIFactory* declares three operations: *createMenu*, *createScrollbar* and *createButton*, one for each of the product dimensions (menu, scrollbar, button) each of which has a return type of the corresponding *Product* type (*Menu*, *Scrollbar*, *Button*). The *concreteCreateOps* method participant has 6 bound elements, a set of 3 methods (each creating a corresponding *ConcreteProduct*) for each of the two *ConcreteFactories*.
Figure 11 shows a UML class diagram representing the UML model elements bound in Figure 10. This could have been independently developed and bindings made manually in the Instantiation Diagram. Alternatively, having specified the dimension category names for each dimension, and some other key bindings (in this case only the *GUIFactory* binding) simple regular expressions (defined in the specification diagram) specifying naming convention patterns combined with the extended relation expressions, and a small amount of manual intervention (notably for incomplete relations) could be used to directly *generate* the bound element names and from them, the bound elements themselves, and thence the equivalent UML model for those elements. For example, the *ConcreteFactories* bindings may be generated by pre-pending the string "Factory" with the Factory dimension category name with its first letter capitalised, while the *ConcreteProducts* bindings may be generated by the cross product of the Factory dimension category names (1st letter capitalised) and the Products dimension category names (1st letter capitalised). The asterisks in the examples indicate names or parts of names which can be generated in this way.

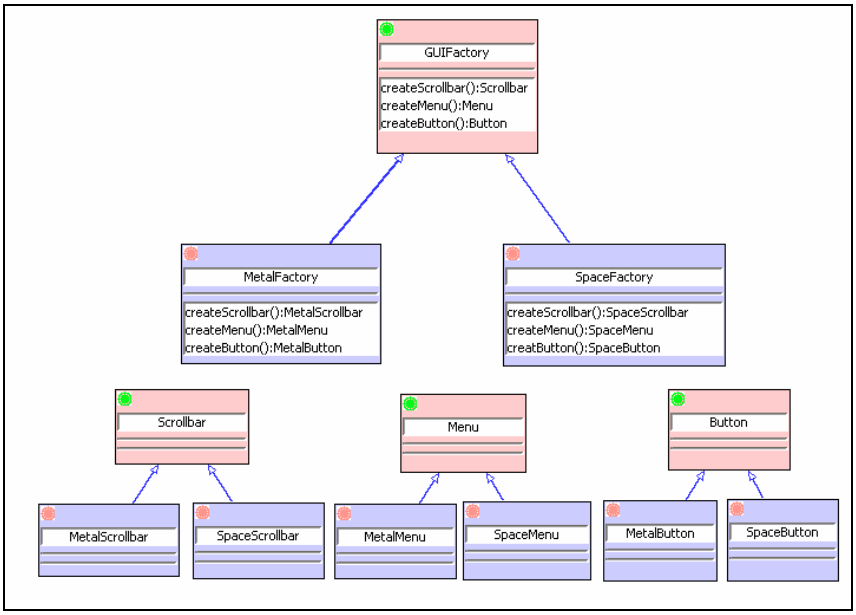

Figure 11: UML Design for GUIFactory

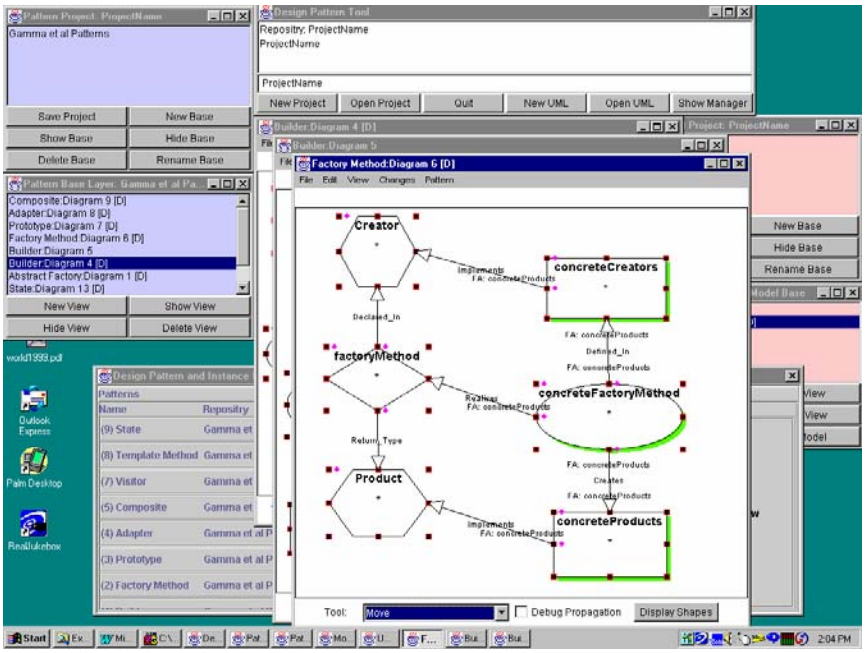

Figure 12: The Prototype DPTool in use

TOOL SUPPORT

We have developed two proof of concept tools to support the use of DPML for pattern modelling and instantiation. The first, reported in (Mapelsden et al, 2002), is a standalone tool, DPTool, implemented using our JViews/JComposer meta-toolset. Figure 12 shows this tool in use. The second, MaramaDPTool, is an Eclipse (Eclipse Foundation, 2006) plugin generated using our Ponamu/Marama meta-toolsets (Zhu et al, 2004; Grundy et al, in press), and which use Eclipse's EMF and GEF frameworks for model and view support respectively. The bulk of the diagrams presented earlier in this paper were generated using MaramaDPTool. Three views of this tool in use are shown in Figure 13. Of the two implementations, DPTool is the most developed in terms of functionality, however MaramaDPTool, based as it is on the Eclipse framework, has far better potential for integration with other programmer productivity toolsets.

Both tools provide the following functionality:

- Modelling views for specification, instantiation, and UML class diagrams, including specification of naming convention patterns.
- For each type of view, multiple views can be modelled, with consistency maintained between the views, meaning that complex patterns or UML models can be broken down into a collection of partial specifications, each contributing to an underlying model
- Support for instantiation of a pattern, through generation of an instantiation diagram with the same layout as the specification diagram it is derived from, but with participants replaced by proxies
- Consistency management between specification and instantiation views, so changes to a specification are reflected in each of the instantiations.
- Support for binding UML model elements to instantiation diagram participants and proxies.

- Consistency management between UML class diagram views and instantiation diagram bindings so that changes to the pattern instantiation can be reflected in the UML class diagrams and vice versa.
- Support for instantiation of multiple, overlapping patterns into a UML model through the use of multiple instantiation diagrams contributing to a common UML model.
- Model management support, to allow saving and loading of models (DPML and UML), undo-redo, etc.

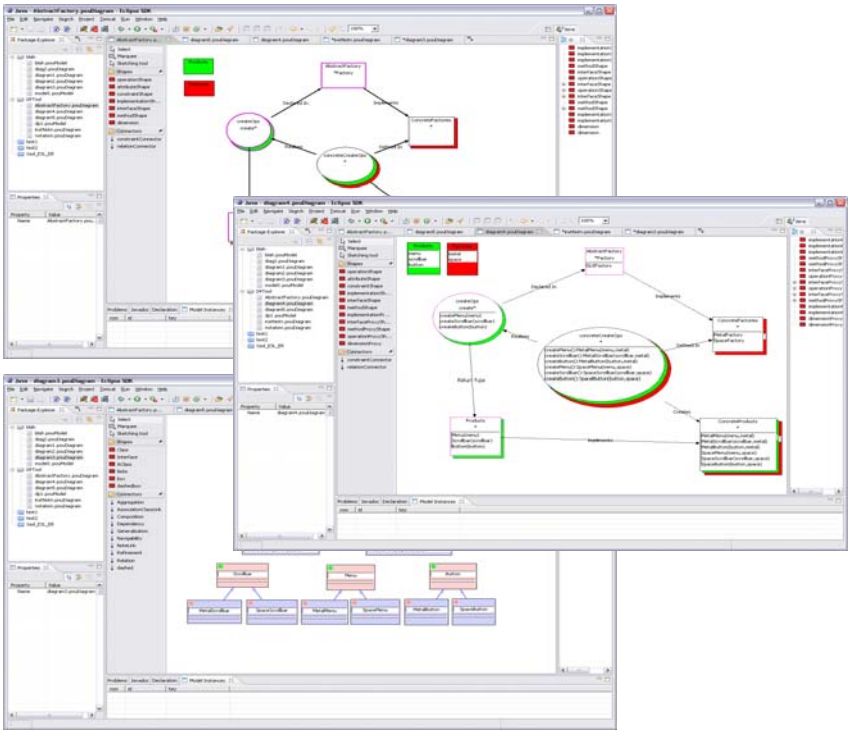

Figure 13: Three views using the MaramaDPTool (top) Design Pattern Specification Diagram (centre) Design Pattern Instantiation Diagram (bottom) UML Class Diagram.

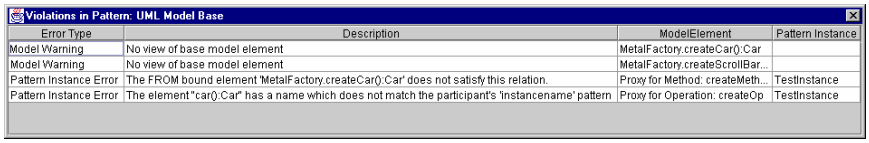

Figure 14: The Error Display in the DPTool

In addition, the JViews based DPTool has better support for pattern realisation and model validation, together with better repository support through provision of a library of predefined patterns. Realisation support includes recommendations on UML model elements that could be validly bound to participants. Validation support checks validity of UML models against the pattern specification, checking for incompleteness, i.e. participants that aren't bound, and inconsistency, i.e. violations of the DPML or UML well formedness rules and violations of the participant naming convention patterns. Errors discovered are displayed in a window, as shown in Figure 14. Validation in the DPTool is a user instigated operation. In our MaramaDPTool, we are implementing an Argo critic style of validation

support which can execute in the background generating a to-do list of identified errors (Robbins et al, 1999).

Pattern Abstraction is the process of identifying a useful or interesting structure or design in an object model and abstracting from that object structure to a suitable DesignPattern model. This mechanism forms the third leg (the first two being pattern instantiation and pattern realisation) of a complete round trip engineering process for design patterns. In Pattern Abstraction a group of elements from an object model are identified and used as a blueprint for creating a DesignPattern model. This is not currently well supported in either of our tools. Currently this requires manual construction of Instantiation and Specification diagrams and binding of participants to the UML model. In our MaramaDPTool we are developing complementary support to the pattern instantiation mechanisms permitting selection of UML model elements and generation of a Pattern Instantiation diagram, which in turn can be automatically abstracted to Pattern Specification diagram. This is more complex than the pattern instantiation process, however. In general it is impossible to create a fully accurate and defined design pattern automatically from a collection of object model elements. This is because there are any number of ways certain arrangements of elements could be abstracted to a design pattern model. A particular difficulty is the recognition of repeating sets of elements as elements that should be represented by Participants with Dimensions. Accordingly, the abstraction support will only be able to automatically perform parts of the abstraction process requiring manual assistance from the user for aspects such as dimension identification. An additional diagram type more explicitly showing binding relationships between DPML and UML elements may be a useful component in solving this problem.

EVALUATION

We have evaluated DPML, and, in particular, the DPTool via an end user based usability study and a cognitive dimensions (Green & Petre, 1996) assessment. Results of these evaluations have been reported in (Maplesden et al, 2002). In brief, the user study, which was qualitative in its nature, showed that the tool was regarded favourably by the survey group, as was much of the language and notation used by the tool. 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 and users found the tool useful and usable for its primary tasks of creating and instantiating design patterns models. Weaknesses and difficulties noted included a difficulty in understanding the dimension and dimension category concepts due to poor visualisation of these and the lack of annotation capability. Both of these have been addressed in MaramaDPTool, the former by having explicit iconic representation of dimensions and dimension category bindings (these were not in the original notation) and the latter by supporting textual annotations. Pattern abstraction support was also highlighted as a desirable feature, which we are currently addressing in MaramaDPTool. The integration of DPML support into Eclipse via MaramaDPTool allows code generation and consistency management between UML models and target code to be maintained. It also potentially allows us to use MaramaDPTool pattern descriptions with 3rd party UML tools via their XMI-based UML meta-models.

The cognitive dimensions assessment highlighted the strong abstraction gradient and difficult mental operations introduced by the dimension concept. These have been mitigated, as noted above, by more explicit iconic representation of dimensions. However, it is worth noting that DPML is already better in both aspects than other, more formal notations, such as Kent and Lauder's (1998) or LePUS (Eden et al 1998). DPML exhibits good closeness of mapping, and consistency, and DPTool provides good

progressive evaluation support via its validation mechanism. Hidden dependencies are an issue, as is the case with any multi-view tool. Secondary notation capability in DPTool is poor, but has been addressed through the enhanced annotation capability in MaramaDPTool.

DISCUSSION AND FUTURE WORK

As described in the introduction, we feel that in addition to DPML (particularly the DPML meta model) and the prototype tool support we have developed for it, our most significant contributions have been in the area of dimensions and instantiation of patterns into designs. These both have some novel characteristics which have more general applicability. Accordingly it is instructive to understand how we developed these concepts in comparison to alternatives.

In the DPML metamodel we wanted a concept that would allow us to create models including groups of objects of arbitrary size. There were various other approaches we could have taken besides using the Dimension concept. One simple approach would have been to allow the cardinality of a Participant to be specified directly, indicating how many objects it represents; this is the approach taken in RBML (Kim *et al*, 2003). This would have allowed groups of objects to be detailed by a single Participant but lacked a certain expressive power. One could not, for example, express the fact that a Participant represents objects that can be classified into sets by multiple criteria, that it effectively has sets of sets of objects. Another method we considered was the set-based approach of LePUS. This would have allowed us to create a set of objects and then a set of sets of objects etc. The main drawback is that it implies an unnecessary ordering on the groupings of the objects, you must group them into sets according to some criteria first and then group the sets into sets according to a second criteria and so on. The order in which you apply the criteria is arbitrary but fixed, once specified you cannot go back and re-order the groupings to suit the different relationships the groupings may be involved with.

We came up with the concept of a Dimension to get around this problem. Designers can specify that a Participant has a certain number of dimensions but no ordering of those Dimensions is (nor should be) implied. The objects linked to a Participant then can be classified according to their position within a particular Dimension and we can consider the Dimensions in any order we wish, each order creating a different sequence of classifications. No order then is implied by the specification of the Dimensions, simply that this Participant has another Dimension in which the objects attached to it can (and indeed must) take up a position. Another advantage of the Dimension concept is that the same Dimension can be applied to different Participants to imply they have a similar cardinality in that one direction. This enables us to easily specify constraints, such as two Participants have exactly the same cardinality, by giving them both the same Dimension. Also if one Participant has a Dimension and another Participant has that Dimension plus a second Dimension then we are saying the second Participant represents a group of objects for every object in the first Participant.

The proxy elements were also an interesting design decision. We considered, initially, replicating a Design Pattern's structure in a Design Pattern instance by simply linking the original objects from the Design Pattern into the instance. However this technique would result in an unnecessarily messy and inelegant object structure in our DPML models. The same object would have been linked into many instances and had potentially many different instance-specific alterations added to it. The proxy elements allow us to maintain a certain separation between the instances and the original pattern while still having a mechanism for keeping the structures consistent. It is not possible in the metamodel to

express the fact that, in an implementation of the DPML, the proxy elements should listen to their base element and make changes, when required, to maintain consistency with that base element. However you can specify that, in a correct model, proxy elements must be consistent with their base elements. The proxy elements then take part in all the relationships and activities that have instance-wide scope, while alterations and relationships that have pattern-wide scope take place in the pattern and, in a tool implementing the DPML, can be propagated to the instances via the proxy elements. Proxy elements help us maintain the self-contained nature of the original design pattern models so that they can be used independently from models of their instances (although not vice versa!).

In both cases, dimensions and proxies, we feel that the approach we have taken is more generally applicable, in particular to any model driven development situation where an element in the model can represent and be instantiated as a multiply categorised set and where models have multiple, tailored instantiations respectively. We are, for example, exploring the use of dimensions in our meta tool model specification languages as an alternative to the cardinality approaches we are currently using. The decisions about what object structures to model as Participants and what to model by using Constraints or Relations were the most difficult ones we had. The set of Participants we came up with was fairly standard by most measures. However arguments can be made for modelling an Attribute as a Constraint on an Implementation, rather than as a Participant as we have done, or for modelling constructs such as Associations as Participants, rather than with a BinaryDirectedRelation, which is the approach we took. We were largely influenced in our decisions by what structures in an OO design have a clear definition in an implementation (such as a class, a method or an attribute) because these fell naturally into roles as Participants. If associations between classes in a mainstream programming language are ever given a clearer, more encapsulated definition than as a pair of object references, our perception of associations could well change. This would be an argument for having Associations as Participants rather than as a condition that exists between two other Participants.

Another area that requires further work is pattern composition. It is obvious after working with DPML for a while that some combinations of Participants or mini-patterns are very common and are repeatedly re-used in design after design. This was an aspect also noted by our survey respondents. While our model and tool support instantiation of multiple patterns into a UML model, there is no support for composing patterns from other patterns. In the DPML metamodel the direct superclass of DesignPattern and DesignPatternInstance, DesignPatternNamespace, is a direct subclass of DesignPatternModelElement. We gave some consideration to making DesignPattern a subclass of Participant to facilitate a form of design pattern composition. With the ability to include a whole DesignPattern as a Participant of another pattern you could specify a simple mini-pattern in a DesignPattern and use it in many other patterns, even with Dimensions to specify multiple groups of the DesignPattern. There are complicating issues that arise with this addition to the metamodel however, particularly in the changes required to the instantiation metamodel and in the semantics of linking additional Constraints to the Participants involved in the sub-pattern, which leads to the whole notion of visibility of Participants from outside the DesignPattern. We have not been able to address these issues satisfactorily yet and so we have not as yet been able to incorporate this idea for pattern composition into our metamodel or tool support. By contrast, Mak *et al* (2003) have developed an extension to LePUS (exLePLUS) which provides pattern composition capability. Noble and Biddle (2002) provide a deeper discussion of the relationships that exist between patterns that provides a useful basis for further developing pattern composition tool support.

Generalising, it can be seen that other, e.g. more domain specific, participants could be selected creating a variety of DPML-like variants for use in model driven development. Our metamodel is readily adaptable to this approach, and the meta-toolset based implementations we have been using for implementation likewise allow for rapid adaptation to incorporate new participant types and related icons. We thus see our developing MaramaDPTool as an underlying framework for rapidly developing domain specific modelling languages that support instantiation into UML designs as part of an overall model driven development approach. In this respect we have similar aims to those behind the development of Microsoft's DSLTool meta toolset (Microsoft, 2006). This type of approach is likely to suit constrained situations, such as product line configuration, where domain specific visual notations afford a more accessible approach for end user configuration than existing conventional coding approaches.

CONCLUSIONS

We have described DPML and associated toolset for specifying and instantiating design patterns. Patterns specified using DPML can be instantiated into UML designs. The instantiation process supports customisation, to adapt the pattern instance for the particular context it is applied to. Two proof-of-concept tools have been developed supporting the use of DPML. Evaluations, both user and cognitive dimensions based, demonstrate the usefulness and effectiveness of the language and tool support. Of particular novelty is our approach to specifying multiplicity of participants in a design pattern, using the dimension concept, combined with binary extended relations that interpret the dimensions in a particular context. We feel these have broader applicability in other areas of model driven development.

ACKNOWLEDGEMENTS

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

REFERENCES

Chambers, C., Harrison, B., & Vlissides, J. (2000). A debate on language and tool support for design patterns, In Wegman M. & Reps, T. (Eds) *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, (pp 277 – 289) Boston ACM Press.

Coplien, J.O. (1996). *Software Patterns*. SIGS Management Briefings. SIGS Press.

Eclipse Foundation (2006), Eclipse, retrieved 30 March, 2006 from http://www.eclipse.org/

Eden, AH. (2002). A visual formalism for object-oriented architecture", In *Proc Integrated Design and Process Technology*, Pasadena, California.

Florijn, G., Meijers, M., & van Winsen, P. (1997). Tool support for object-oriented patterns", in *ECOOP '97 – Proceedings of the 11th European conference on Object Oriented programming, LNCS 1241*, 472-495

Fontoura, M., Pree, W., & Rumpe, B. (2002). *The UML Profile for Framework Architectures*. Addison-Wesley.

France, R.B.; Kim, D.-K.; Ghosh, S., & Song, E. (2004). A UML-based pattern specification technique, *IEEE Trans SE* 30(3), 193-206.

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 & Petre, M., (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework, *Journal of Visual Languages and Computing, 7*, 131-174
Grundy, J.C., Hosking, J.G., Zhu, N., & Liu, N. (in press). Generating domain-specific visual language editors from high-level tool specifications, In *Proc 2006 ACM/IEEE Conference on Autaomted Software Engineering*.
Guennec, A.L., Sunye, G., & Jezequel J. (2006). Precise modeling of design patterns. In *Proceedings of UML'00* (pp. 482-496)
Kim, D., France, R., Ghosh S., & Song, E. (2003). A UML-Based Metamodeling Language to Specify Design Patterns, *Proceedings of Workshop on Software Model Engineering (WiSME), at UML 2003*, San Francisco.
Lauder, A., Kent, S. (1998). Precise Visual Specification of Design Patterns, In *Proc ECOOP'98 Workshop reader on OO technology, LNCS 1445*, pg114-134.
Mak, J.K.H., Choy, C.S.T., & Lun, D.P.K (2003). Precise specification to compound patterns with ExLePUS, In *Proc. 27th Annual International Computer Software and Applications Conference*, 440-445
Mak, J.K., Choy, C.S.T., & Lun, D.P.K. (2004). Precise Modeling of Design Patterns in UML, In *26th International Conference on Software Engineering (ICSE'04)*, 252-261.
Mapelsden, D., Hosking, J.G., & Grundy, J.C. (2002). Design Pattern Modelling and Instantiation using DPML, In *Proc. Tools Pacific 2002*, Sydney, IEEE CS Press
Microsoft (2006). *Domain-Specific Language Tools*. retrieved 30 March, 2006 from http://msdn.microsoft.com/vstudio/DSLTools/
Noble, J., & Biddle, R. (2002) Patterns as Signs. In *Proceedings of the European Conference on Object Oriented Programming*, Spain. (ECOOP). pp368-391. Springer-Verlag
Object Management Group (2006). Unified Modelling Language (UML) Specification v 2.0., retrieved 30 March, 2006 from www.omg.org/technology/documents/formal/uml.htm.
Reiss, S.P. (2000). Working with patterns and code, In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, Volume: Abstracts*, (pp 243 –243).
Robbins, J.E., & Redmiles, D.F. (1999). Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML, In *Proc CoSET'99* (pp. 61-70) Los Angeles: University of South Australia.
Sefika, M., Sane, A., & Campbell, R.H. (1996). Monitoring Compliance of a Software System with its High-Level Design Models. In *Proceedings of the 18th international conference on Software engineering*. Berlin, Germany. 387 – 396
Taibi, T., & Ngo, D.C.L. (2003). Formal Specification of Design Patterns – A Balanced Approach, *Journal of Object Technology 2(4)*, 127-140
Zhu, N., Grundy, J.C., & Hosking, J.G. (2004). Pounamu: a meta-tool for multi-view visual language environment construction, In *Proc. IEEE Visual Languages and Human Centric Computing Symposium*, Tokyo, 254-256.