Basic Relationship Patterns

James Noble
MRI, School of MPCE,
Macquarie University, Sydney.
kjx@mri.mq.edu.au

Abstract

Relationships between objects are almost as important to designs as objects themselves. Most programming languages do not support relationships well, so programmers must implement relationships in terms of more primitive constructs. This paper presents five basic patterns which describe how objects can be used to model relationships within programs. By using these patterns, programs and designs can be made smaller, more flexible, and easier to understand and maintain.

Introduction

Relationships, also known as collaborations or associations, are very important in object oriented design. For example, collaborations make up one third of the CRC (Class Responsibility Collaboration) design technique [25], and Rumbaugh has claimed that relationships are complementary to (and as important as) objects themselves [20]. Unfortunately, most object oriented programming languages do not support relationships well. For example, designs may use one-way or two-way relationships between objects; they may associate one object with one other object, one object with many other objects, or many objects with many others; they may be complete — always relating objects — or partial, sometimes relating objects and sometimes not. Programming languages support relationships mainly through attributes (also known as variables, slots, or data members), which are one-way links from one object to one other object. This means that programmers must somehow build up other relationships using the raw materials the languages provide — objects, messages, and attributes [22].

This paper presents five basic patterns for implementing relationships between objects. The patterns address the most basic kind of relationships where one object needs to be able to refer to another object at runtime. These relationships are often called associations or collaborations, to distinguish them from the more complex aggregation and inheritance. The first two patterns describe the fundamental ways to model relationships — either using attributes or using objects. The following three patterns describe how one-to-one, one-to-many and two-way relationships can be modelled. The elementary content of these patterns should not surprise any object oriented software practitioner, and they can all be found in the general OO literature [25, 2, 21, 22]. Rather, these patterns attempt to record well known design and programming folklore, describing techniques to new programmers, and, for more experienced programmers, illustrating when particular techniques are appropriate. The patterns are mostly language independent, however, they will work best in languages like Smalltalk, where (almost) everything is an object, especially as they do not address explicit memory management. The patterns were motivated by experience teaching object oriented design and programming to students who had backgrounds in data modelling, or who had attended a course on object oriented analysis techniques, but otherwise had no practice in object oriented design.

Figure 1 summarises the problems dealt with by this collection of patterns, and the solutions they provide. Although these patterns do not form a complete pattern language, there is some structure to these patterns, which is illustrated in Figure 2. This diagram illustrates two relationships between patterns in the language. One pattern can *refine* another pattern, providing more detail on how to implement that pattern. Alternatively, two patterns can *conflict*, indicating that the two patterns are mutually exclusive, each providing a different solution to a common problem.

Pattern	Problem	Solution
Relationship As Attribute	How do you design a very simple relationship?	Make an attribute to represent the relationship.
Relationship Object	How do you design a big, important, or common relationship?	Make a Relationship Object to represent the relationship.
Collection Object	How do you design a one-to-many relationship?	Make a Collection Object to model the relationship.
Active Value	How do you design an important one-to-one relationship?	Make an Active Value to model this relationship.
Mutual Friends	How can you represent a two-way relationship?	Make a consistent set of one-way relationships.

Figure 1: Summary of the Patterns

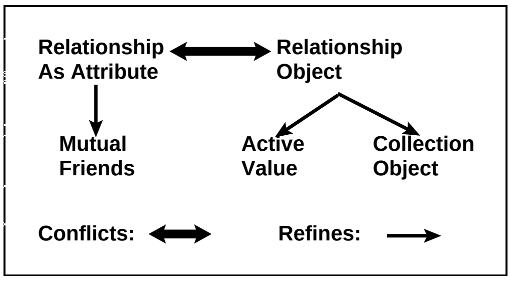

Figure 2: Structure of the Patterns

Forces

Each of these patterns resolves a number of different forces, and some conflicting patterns (such as **Relationship as Attribute (1)** and **Active Value (4)**) resolve similar problems in different ways. Many of the patterns consider the *ease* of *reading* or *writing* of a particular solution — generally, solutions which are easy to write are more likely to be chosen by programmers, and solutions which are easy to read are likely to be easier to maintain. Several patterns also address the *cohesion* and *coupling* of the resulting designs, as designs with high cohesion within objects and low coupling between them are more flexible, understandable, and easier to maintain. This is often related to whether a relationship is represented *explicitly* by a single element of a design, or whether it is *dispersed* across several objects, attributes, and methods. Representing a relationship explicitly generally makes it easier to *identify* the relationship within the design, to *change the implementation* of the relationship if necessary, to maintain *consistency* in two-way relationships, and to *reuse* both the relationship and other participating objects elsewhere. The patterns are marginally concerned with *efficiency* — the *time* and *space* cost of a design, and the *number of objects* it requires.

Example

The patterns in this paper use examples drawn from a simple system for an insurance office, see Figure 3. The most important object in the system is Policy, which represents an insurance policy. Each Policy belongs to a Client (although a single client can have many policies), and a policy can also have a number of Endorsements to change its wording. When a policy is issued, it is associated with one or more Underwriters, who receive a share of the premium in return for meeting a share of the costs of any claims. Finally, a PolicyWindow, not shown in the diagram, provides a GUI interface for the system.

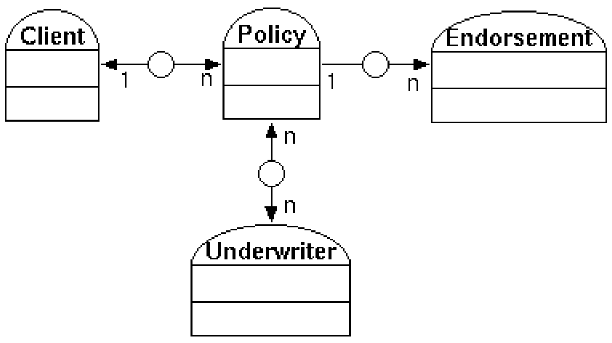

Figure 3: Insurance Analysis Model

Form

The patterns are written in a modified Portland form. Each begins with a question (in italics) describing a problem, followed by a one paragraph describing the pattern's context, and a second paragraph describing the forces the pattern resolves. A boldface "**Therefore:**" introduces the solution (also italicised) followed by the consequences of using the pattern, an example of its use, and some known uses and related patterns. All the examples are presented in Smalltalk, although the patterns are applicable to most object oriented languages.

1 Relationship As Attribute

How do you design a very simple relationship?

Simple, one-way, one-to-one relationships are very common in object oriented models. For example, an insurance Policy object must record the date and time the policy was issued, and an client object must record the name and address of the client it represents. Such relationships generally carry very little weight in the application domain — the objects themselves are important, but the fact that these particular objects are related is coincidental. Often, the fact that the objects are related is important to only one of the related objects (the date is important to the policy, but the policy isn't important to the date). Changes in the relationship — for example, a client changing their address — are only important insofar as they record the changing circumstances of that object in the program domain, and don't have far-reaching effects within the program.

Because these kinds of relationships arise so often, they need to be simple to represent in a program, so that they are easy to write and can be immediately understood by programmers later reading the program. For similar reasons, they must be implemented so that they impose a minimal overhead on the program — both in terms of the space required to represent the relationship, and the time taken to manipulate it.

Therefore: *Make an attribute to represent the relationship.*

Object oriented programming languages provide attributes (variables, slots, data members) which are local to each object instance. These attributes are ideal for modelling one-to-one relationships. Add accessor messages if the relationship should be publicly available, and mutator messages if the relationship should be publicly changeable.

Example

Consider modelling an insurance policy. Each Policy object must record the date and time it was issued, the sum insured, and any excess. The Policy object's behaviour doesn't depend upon the precise values of any of these objects, and they can be changed as necessary, without any consequent effects on the program's model of the domain. Because the relationships are very simple, all this information can be stored as attributes of the Policy object.

Consequences

Setting and accessing attributes is easy and efficient in every object oriented programming language. This pattern tightly couples the object with the attribute to the object referred to by the attribute. Attributes occupy space in objects even if they are not needed in a particular case, for example, if a policy has no excess, an attribute to store the excess will still be allocated in the Policy object.

This pattern is fundamental to object oriented design, just as object's attributes (that is, state) are fundamental to object oriented programming. Although this pattern is simple it is not necessarily trivial, and in my experience needs to be learned, especially by programmers with a predilection for modelling objects as tuples using collections. More importantly, it provides alternatives to the other patterns in this language which deal with more complex relationships.

Known Uses

Every object oriented program represents simple relationships as attributes.

Related Patterns

Relationship Object (2) is a complementary pattern which describes how to represent important relationships with objects.

2 Relationship Object

How do you design a big, important, or common relationship?

Relationships between objects can be very complex, involving two-way communication between many participating objects which are essentially peers. For example, the risks (and premiums) of all the policies issued in an insurance office must be distributed carefully to underwriters and reinsurers. This distribution does not depend solely on the details of a single policy. Rather it must take into account contracts with underwriters, reinsurance treaties, and arrangements made for other policies. These kinds of relationships embody important concepts and constraints from the programs' domain, and can require complex state or behaviour to implement.

Complex relationships can be implemented directly using features of programming languages such as attributes (see **Relationship As Attribute (1)**), but this has several disadvantages [2, 21]. The relationship is dispersed among its participating objects — it cannot be easily identified within the program, and thus cannot be easily located or maintained. Participating objects are tightly coupled, so a change to the relationship necessitates a change to several of the participating objects. Participating objects' cohesion is reduced, because they must model a portion of the relationship as well as the abstractions they represent. If individual developers or teams "own" each object, they will need to cooperate to implement the relationship.

Therefore: *Make a* **Relationship Object** *to represent the relationship.*

Move any methods or variables associated with the relationship from the participating objects, and place them into the **Relationship Object**. Change the implementations of the participating objects so that they refer to each other via the **Relationship Object**. The **Relationship Object** may need to use internal subordinate objects to implement the relationship, but these should not be visible through the **Relationship Object's** interface.

Example

Consider designing a system to support an insurance underwriting office. The core of such a system is a relationship between policies and underwriters which records the business written by the office. This relationship is many-to-many (a policy can be underwritten by many underwriters, and a underwriter can underwrite many policies), partial (each policy is underwritten by a few underwriters, and each underwriter underwrites a small fraction of the policies). The relationship is subject to a large number of business rules. Distributing the implementation of this relationship over the Policy and Underwriter objects would unnecessarily complicate both these objects.

Introducing a **Relationship Object** — a Portfolio — simplifies the design (see Figure 4). All the details of the relationship are collected into the Portfolio. The Portfolio object can use whatever internal implementation structures are needed for an efficient implementation, and these can be changed as necessary without affecting Policy or Underwriter. The Portfolio can implement the business constraints and behaviour required by the relationship.

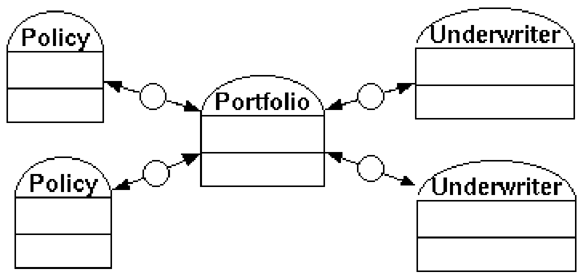

Figure 4: Relationship Object

Consequences

A **Relationship Object** explicitly represents a relationship in the program. The other objects involved in the relationship are independent of it (coupling is reduced), so the relationship object can be changed more easily, and may be able to be reused in other contexts. An explicit **Relationship Object** also increases cohesion, since the other participating objects do not need to model part of the relationship. Introducing **Relationship Objects** increases the number of objects in the program, which can increase memory footprint, and accessing objects indirectly via **Relationship Objects** can reduce execution efficiency. **Relationship Objects** are currently under discussion in the literature on object oriented design [10, 18].

Known Uses

ParcPlace Smalltalk provides several examples of **Relationship Objects**. Collections and ValueHolders are specialised kinds of **Relationship Objects** (**Active Value (4)** and **Collection Object (3)** respectively). ObjectWorks introduced DependentsCollection objects to record and manage the relationship between an object and its dependents [19], where the earlier Smalltalk-80 implemented this relationship using a global dictionary [9].

Related Patterns

Relationship As Attribute (1) is a complementary pattern which describes how to represent simple one-to-one relationships efficiently. **Observer** and **Mediator** [8] and **Director** [5] describe how **Relationship Objects** can coordinate or control the behaviour of their participating objects. **Relationship Objects** can also record supplementary information, such as the time span of the relationship [3]. Martin Fowler has described **Relationship Objects** in detail [6].

3 Collection Object

How do you design a one-to-many relationship?

One-way one-to-many relationships are almost as common as one-way one-to-one relationships. For example, a standard insurance policy can be endorsed to deal with special situations. One policy can have multiple endorsements, so this should be modelled as a many-to-one relationship between Policy objects and Endorsement objects. Since Endorsements are meaningless outside their particular policies, this is a one-way relationship.

Because they are so common, one-to-many relationships need to be implemented as easily and as efficiently as possible. Unfortunately, in most object-oriented programming languages, one-to-many relationships are rather more difficult to implement than one-to-one relationships. The **Relationship As Attribute (1)** pattern does not extend cleanly to relationships where one object is linked to more than one object. Although the one object can have multiple attributes, each attribute must be accessed individually — the attributes cannot be managed as a group. Multiple attributes can be used to implement multiple one-to-one relationships, but not one-to-many relationships. One-to-many relationships can be implemented by hard-coding the relationship into *every* participating object. This disperses the relationship across all participating objects, couples them very tightly together, and has to be rewritten for each relationship in a design.

Therefore: *Make a* **Collection Object** *to model the relationship.*

Most object oriented libraries provide a variety of container or collection objects which can be used to represent one-to-many relationships. The "one" object simply stores a collection which holds the "many" objects participating in the relationship.

Example

Consider modelling an insurance policy which can have one or more endorsements attached. This relationship can be represented by a List **Collection Object**. The Policy class simply keeps a List of all its Endorsements (see Figure 5). If necessary the List can be replaced with a specialised collection object (perhaps an EndorsementList) to implement any additional constraints or behaviour required by the relationship.

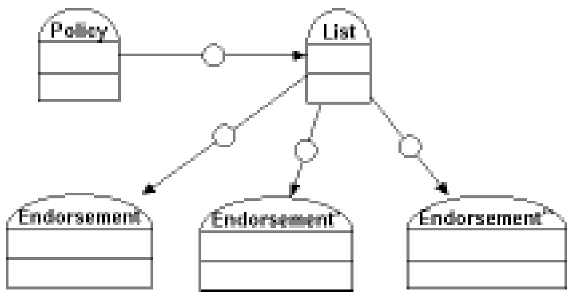

Figure 5: Collection Object

Consequences

Using a **Collection Object** to represent a one-to-many relationship explicitly greatly simplifies a program, especially as in most object oriented languages, collections are almost as easy to use as attributes. The **Collection Object** and the objects stored inside it are very weakly coupled, and so each can be reused in different contexts. A **Collection Object** introduces one or more extra runtime objects, depending on its implementation, and the extra level of indirection reduces execution speed.

Known Uses

Collections are ubiquitous throughout object oriented programming. They are the most common kind of **Relationship Object (2)**, and they form the core of many class libraries [2, 9, 16]. Most collections implement one-to-many one-way relationships, although some, such as Smalltalk's Dictionaries, can implement many-to-many relationships.

Related Patterns

Pattern-Oriented Software Architecture [4] describes the *collection-members* variant of the **Whole-Part** pattern. *Smalltalk Best Practice Patterns* [1] includes a number of patterns about using Smalltalk's **Collection Objects**.

4 Active Value

How do you design an important one-to-one relationship?

Some one-to-one relationships are important in themselves, rather than simply connecting two objects. For example, a PolicyWindow object (a GUI entry window for a Policy) needs to store the values entered by the user. The PolicyWindow object has a one-to-one relationship with the value of each of its entry fields. If one of these values (such as the sum insured) is changed, code embodying business rules must be run to validate the entered value.

One-to-one relationships are often important because of their position in the program's architecture. That is, objects in the program which do not participate directly in the relationship may be affected if the relationship changes — in the example, changing the sum insured field means that business rules must be called to verify the entered values. Alternatively, one-to-one relationships may be important to a program due to the underlying domain. That is, a change in the relationship represents a change to an important parameter of the domain, which the program must deal with. The importance of the relationship can be judged by the consequences of a change to the relationship.

A one-to-one relationship can be implemented using **Relationship As Attribute (1)**. This requires that code to detect and handle changes in the relationship (i.e. assignments to the attribute) must be included in the object containing the attribute, and must be invoked whenever the attribute could have changed. **Observer** [8] can be used to notify dependent objects of any changes, but this requires code which detects and signals changes to be written especially for each attribute. This reduces the cohesion of the class containing the attributes, and increases that class's complexity.

Therefore: *Make an* **Active Value** *to model this relationship.*

An **Active Value** is essentially an object which represents a single variable. It should have an attribute to hold the variable's value, and should understand two messages — an accessor to retrieve the value of the variable, and a settor to change the value. To use an **Active Value**, make it an attribute of the source object of the relationship, and access the relationship by sending messages to the **Active Value**, rather than to the source object. The **Active Value** can detect when its value changes, and then act as the Subject in **Observer** [8] to update any dependent objects.

Example

The PolicyWindow can use **Active Values** to store the values of the entry fields — see Figure 6. Business rules for data validation can access the **Active Values** directly, rather than via the Policy Window. Each **Active Value** can detect when it has changed, and then notify interested BusinessRules. The PolicyWindow needs no specialised code to signal updates, provided all accesses to its fields are directed via the **Active Values**. Similarly, a BusinessRule does not need to access the PolicyWindow because it can access the information it needs via the **ActiveValues**.

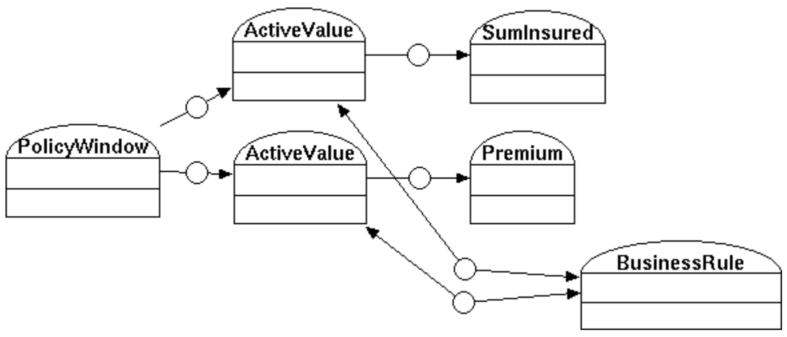

Figure 6: Active Value

Consequences

Like **Relationship Object (2)**, this pattern reduces the coupling and increases the cohesion of the objects involved. Code to detect and signal updates in the relationship can be written once for all **ActiveValues**, rather than being dispersed within the participating objects. **Active Values** complicate the program's structure by introducing many small objects, with the indirection involved increasing the time and space costs of the program. **Active Values** also reduce the clarity of program text, since otherwise simple variable accesses and assignment statements must be replaced by messages sent to **Active Values**.

Known Uses

Active values were first provided as part of the language in LOOPS [23]. VisualWorks's ValueModel framework is based on this pattern [19]. In particular, the ValueHolder object is a generic **Active Value**. The Cooldraw [7], Unidraw [24], and QOCA [11] constraint solvers all use **Active Values** to represent variables explicitly: the last two uses are mentioned in *Design Patterns* [8].

Related Patterns

Understanding and Using the ValueModel Framework in VisualWorks Smalltalk [26] describes how to use **Active Values** in VisualWorks. **Observer** [8] is often used to link **Active Values** to the objects which depend on the relationship. Several **Active Values** can form a **Connected Group** [13].

5 Mutual Friends

How do you design a two-way relationship?

You have a two-way relationship (also known as a bidirectional or mutual relationship) where all participating objects are equally important. For example, in the insurance system a Client object needs a two-way relationship with its Policy objects — clients need to be able to enumerate their policies to compute the total premium due and policies need to know their clients in the event of a claim.

In a two-way relationship, each participating object needs to be easily accessible from every other object. A change in any one participating object may affect all other objects in the relationship. If an object joins or leaves the relationship, the other objects must be informed so the relationship remains consistent. Note that a two-way relationship may be a one-to-one, a one-to-many, or even a many-to-many relationship.

Unfortunately object oriented programming languages' support for two-way relationships is even more limited than their support for one-way relationships. While one-way one-to-one relationships can be modelled with attributes (**Relationship As Attribute (1)**), and one-way one-to-many relationships modelled with collections (**Collection Object (3)**), there are no obvious language or library constructs which can model two-way relationships.

Therefore: *Make a consistent set of one-way relationships.*

Implementing **Mutual Friends** has two steps. First, the two-way relationship should be split into a pair of one-way relationships. Second, the one-way relationships must be kept consistent.

1. Splitting the relationship. The simplest kind of **Mutual Friends** involves a one-to-one two-way relationship. This can be split into a pair of one-to-one one-way relationships, typically implemented using **Relationship As Attribute (1)**. A one-to-many two-way relationship can be split very similarly, except that one object will need a one-way one-to-many relationship, such as a **Collection Object (3)**. A two-way many-to-many relationship can be split using two collections.

2. Keeping the split relationships consistent. Choose one of the **Mutual Friends** as a *leader*, and have it manage the other objects as its *followers*. The leader object should provide an interface to manage the whole relationship. The followers' implementation of the relationship can be made much simpler, because they should only be invoked by the leader. Language mechanisms (such as C++'s friends or Eiffel's selective export) can be used to enforce this restriction. In a one-to-many relationship, the "one" object is usually the better choice for the leader, since this centralizes the responsibility for maintaining the relationship in a single object. If one object in the relationship creates the other participating objects, this object should be the leader. If a follower needs to make a change to the relationship, it should delegate the change to its leader.

Example

Consider again the relationship between Policies and Clients. This is a two-way, one-to-many relationship, and it can be implemented by making clients and policies **Mutual Friends**. The relationship can be broken down into two one-way relationships — a one-to-many relationship from Clients to Policies, and a one-to-one relationship from a Policy to its Client (see Figure 7). A Client uses a List **Collection Object (3)** to record its Policies. The Policy object's one-way relationship can be implemented using a client attribute, as described in **Relationship As Attribute (1)**.

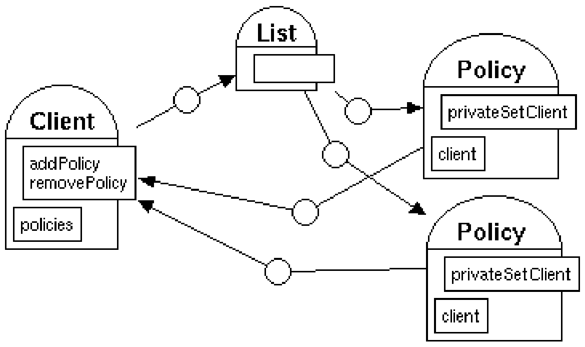

Figure 7: Mutual Friends

The Client objects are the leaders of their **Mutual Friends**, because they are the "one" objects in the relationship. Client provides addPolicy: and removePolicy: messages to acquire and dispose of Policies. These messages set the Policy's client attribute via a privateSetClient message. In Smalltalk, the relationship could be implemented in Client as

```
addPolicy: aPolicy
    policies add: aPolicy.
    aPolicy privateSetClient: self.


removePolicy: aPolicy
    aPolicy privateSetClient: nil.
```

and in Policy as

```
privateSetClient: aClient
    client := aClient.
```

although a robust implementation should check for errors in the parameters. If a Policy needs to change its own Client, it should not change its client variable directly, rather it should delegate this to its leader, that is, to Client:

```
setClient: aClient
    aClient addPolicy: self
```

Consequences

This pattern disperses the implementation of a two-way relationship among all the objects participating in the relationship. This increases the coupling between objects, because each participating object depends upon the details of the other object's implementation. It similarly decreases their cohesion, because each object must include part of the relationship. This pattern can be difficult to write correctly, because both ends of the relationship must be kept synchronised to maintain overall consistency.

Known Uses

The Smalltalk class hierarchy uses a two-way relationship between subclasses and superclasses. A class and its subclasses are **Mutual Friends**, with the superclass as the leader. Each class keeps a collection of it subclasses, and each subclass has a pointer to its class [9]. Similarly, Smalltalk views and controllers are **Mutual Friends**, as are VisualWorks' VisualPart and CompositePart [19].

Related Patterns

If the control flow across a relationship is primarily in one direction, you may be able to model the relationships as if it were a one-way relationship, and use **Self Delegation** [1] for access in the other direction. **Composite** [8] can use **Mutual Friends** to maintain a two-way relationship between containers and leaves. A **Relationship Object (2)** can model an entire complex relationship, and one or more **Observers** [8] can be used to maintain two-way links between the participating objects [12, 18].

James Rumbaugh [22] and Martin Fowler [6], amongst others, have described how two-way relationships can be split into mutual one-way relationships. Ward Cunningham wrote a version of **Mutual Friends** on the WikiWikiWeb, and it was discussed by Ralph Johnston and Steve Metsker [15]. Steve Metsker subsequently described how concurrent two-way relationships can be managed by a **Judge** [14]. Databases often provide support for two-way relationships. For example, all relationships in relational databases can be traversed in either direction, and ODMG-93 object oriented databases automatically maintain the consistency of inverse relationships [10].

Acknowledgements

This paper is a revised version of a paper presented at TOOLS Pacific 1996 [17]. Thanks are due to Wolfgang Keller, the EuroPLOP'97 shepherd for this paper, to John Grundy for his collaboration on *Explicit Relationships in Object Oriented Design* [18], and to Michael Richmond, Jonathan Tidswell, John Potter and David Holmes for their comments on drafts.

References

[1] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice-Hall, 1996.

[2] Grady Booch. *Object Oriented Analysis and Design with Applications.* Benjamin Cummings, second edition, 1994.

[3] Lorrie Boyd. Patterns of association objects in large scale business systems. In *Pattern Languages of Program Design*, volume 3. aw, 1997.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[5] Jens Coldewey. Decoupling of object-oriented systems: A collection of patterns. Technical report, sd&m — software design & management GmbH, April 1997.

[6] Martin Fowler. *Analysis Patterns.* Addison-Wesley, 1997.

[7] Bjorn N. Freeman-Benson. Converting an existing user interface to use constraints. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, 1993.

[8] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[9] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[10] Ian Graham, Julia Bischof, and Brian Henderson-Sellers. Associations considered a bad thing. *Journal of Object-Oriented Programming*, 9(9), February 1997.

[11] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Third Eurographics Workshop on Object-Oriented Graphics*, 1992.

[12] Wolfgang Keller. Mapping associations from OODBs to RDBMs. Technical report, sd&m — software design & management GmbH, 1995.

[13] Jairong Li. Connected group, 1996. Reviewed at EuroPLOP.

[14] Steve Metsker. The judge pattern. Submitted to the Journal of Object Oriented Programming, April 1997.

[15] Steve Metsker, Ward Cunningham, and Ralph Johnston. Symmetrical reference. `http://c2.com/cgi/wiki?SymmetricalReference`.

[16] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries.* Prentice Hall, 1994.

[17] James Noble. Some patterns for relationships. In *TOOLS 21*, Melbourne, 1996.

[18] James Noble and John Grundy. Explicit relationships in object-oriented development. In *TOOLS 18*, Melbourne, 1995. Prentice-Hall.

[19] ParcPlace Systems. *VisualWorks Smalltalk User's Guide*, 2.0 edition, 1994.

[20] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA Proceedings*, 1987.

[21] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, New Jersey, 1991.

[22] James E. Rumbaugh. Models for design: Generating code for associations. *Journal of Object-Oriented Programming*, 8(9), February 1996.

[23] M. J. Stefik, D. G. Bobrow, and K. M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1), January 1986.

[24] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3), 1990.

[25] Nancy M. Wilkerson. *Using CRC Cards: An Informal Approach to Object-Oriented Design.* SIGS Books, 1995.

[26] Bobby Woolf. Understanding and using the ValueModel framework in VisualWorks Smalltalk. In *Pattern Languages of Program Design.* Addison-Wesley, 1994.