Explicit Relationships in Object Oriented Development

James Noble
Centre for Object Technology, Applications, and Research.
University of Technology, Sydney.
Email: `kjx@socs.uts.edu.au`

John Grundy
Department of Computer Science.
University of Waikato, Hamilton, New Zealand.
Email: `jgrundy@cs.waikato.ac.nz`

Abstract

Traditional object oriented analysis methodologies are based not only upon objects, but also upon relationships between objects. Object oriented programming languages do not provide support for relationships, and so analysis relationships must be expressed indirectly within a program's design, and then incorporated into implementations of other objects in a program's code. By using explicit relationships in design and implementation, analysis relationships can be expressed directly within a program's code. Programs which use explicit relationships are often smaller and easier to comprehend than traditional OO programs, and are generally quicker to write and easier to maintain.

1 Introduction

Seamlessness is one of the most important benefits of object orientation (Henderson-Sellers 1994). In an object oriented system development lifecycle, the same conceptual model (the object model) is used to organise the analysis of a problem, the design of a solution, and the implementation of a running program. Seamlessness promotes *traceability* (Pfleeger 1991), so that an object in a program can be easily traced back to an object in the program's design, and further back to an object identified by an analyst and understood by the program's customer. The seamlessness of object oriented development makes quite a contrast to the discontinuities of the more traditional forms of analysis and design, where the models built for analysis (such as data-flow diagrams) bear little relation to the models built for design (such as structure charts or entity-relationship diagrams), or to the final program.

In practice, object oriented development is not as seamless as it may first appear. This is because object oriented analysis techniques (such as OMT (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen 1991), OOADA (Booch 1994) or MOSES (Henderson-Sellers and Edwards 1994)) actually construct models using both objects and relationships between objects. In contrast, object oriented programming languages (such as Smalltalk (Goldberg and Robson 1983) or C++ (Stroustrup 1986)) are based solely upon objects: they do not support relationships other than object containment and object references. Thus, while an object in an analysis can correspond to an object in a program, relationships from analysis are lost during design and implementation.

This paper describes an approach to object oriented design and implementation which considers relationships as well as objects. In this approach, analysis relationships are transformed into objects during design. These design objects can then be implemented directly in an object oriented language. In this way, analysis relationships (as well analysis objects) can be seamlessly traced through design into objects in the implemented program.

This paper is centred around a case study — the implementation of a simple invoicing system. Section 2 describes the system's requirements, using generally accepted techniques of object oriented analysis. Section 3 describes how standard object oriented design and implementation can be used to implement the invoicing system based upon this analysis, and shows that although these techniques preserve the objects identified in analysis, they do not preserve the relationships between objects. Section 4 then applies our alternative design approach to the same invoicing system, and shows how this approach maintains analysis relationships through design to implementation. Section 5 compares this explicit relationship approach to design with the standard object oriented approach. Section 6 discusses related work, and Section 7 presents our conclusions.

2 Object Oriented Analysis

Most object oriented analysis techniques focus on the encapsulation of attributes and methods within objects or classes (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen 1991; Booch 1994; Henderson-Sellers and Edwards 1994). These objects are then inter-related via generalisation, aggregation and association relationships. Generalisation relationships specify type and inheritance hierarchies between objects. Using object oriented design and implementation, generalisation relationships are guaranteed to be preserved as inheritance — which, therefore, we do not consider further in this paper.

Aggregation relationships specify the aggregation of objects to form larger object models. Association relationships indicate one object makes use of another object in some way. Aggregation and association relationships are generally refined into one or more direct references between objects during design and implementation. Unlike generalisations, there is no guarantee that aggregation and association relationships will remain explicitly present in the program.

2.1 A Case Study

Consider a simple invoicing system which stores information about customer purchases. In terms of data requirements, the company has a number of customers, each having a customer code, name and address. Each customer buys products, with each group of purchases recorded line by line on a dated invoice. Each customer has an account, and accounts have transactions which are used to record the amount of each purchase for each invoice line. Customers have a credit limit, and accounts hold a balance and year-to-date balance. Products are made up of individual parts, with the price of the product calculated from the price of its parts.

The functional requirements of this system include the ability to:

- add and delete customers, invoices, accounts, transactions, and products.
- modify an invoice number and date, a customer name or address, or product names, parts, and prices.
- print, find, and sort customers, invoices, accounts, and products.
- calculate the cost of each invoice line purchase (including tax).
- calculate account balance and YTD balance.

2.2 Analysis

Figure 1 contains a high-level analysis diagram for the data requirements of this system. Only attributes are shown here. There are a variety of relationships between classes used in this diagram, either aggregation (unarrowed) or association (arrowed). Company objects are an aggregate of the accounts, customers and products the company has (all one to many relationships). An account holds zero or more transactions, and a product is made up of one or more parts. A customer has one account and zero or more invoices. Each invoice has one or more lines, with each line having one transaction and one product.

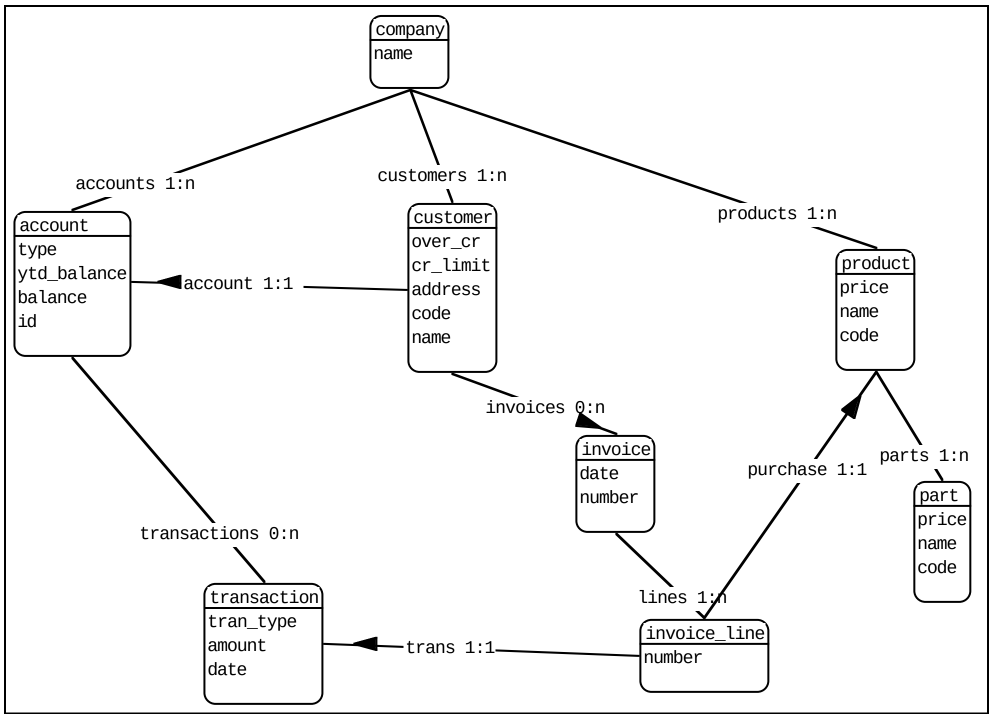

Figure 1: Analysis of Invoicing System Data Requirements

We can extend this data model to provide basic methods associated with each class to carry out the functional requirements described in the previous section (see Figure 2). We have associated these methods with the class most likely to carry out this functionality. For example, the company class manages the top-level functions of adding, deleting and locating account, customer and product information. The account class manages its own initialisation and deletion, posting and reversal of transactions, and updating its YTD balance. The customer class manages its own initialisation and deletion, the adding, deleting, printing and finding of invoices, and modification of its code or name.

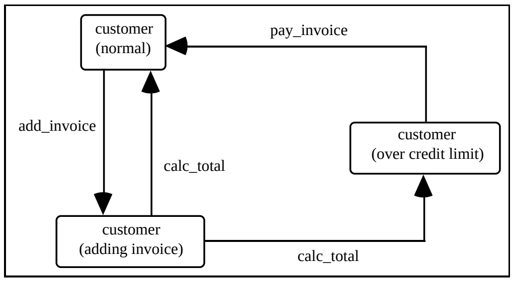

Figure 3: Dynamic Model of Customer Objects

The system functionality embodied in methods can be captured using dynamic models and functional models (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen 1991). These describe the changes in objects' states, and data and control flows between objects. For example, Figure 3 shows a dynamic model describing the possible states of customer objects. If a customer's account balance exceeds their credit limit, then new purchases for the customer are rejected.

Figure 4 shows the functional specification for adding an invoice for a customer. The invoice is first created, then successive invoice lines and their associated transactions created. If the credit limit for the customer is exceeded, no more purchases are allowed. If not, further lines are added. When adding of lines is completed, a receipt is printed for the customer.

Note that this analysis only represents some of the classes, relationships, attributes and methods that may comprise a large system. For example, the account, customer and product classes would normally be used by other subsystems, such as inventory, payroll, job costing, creditors, etc. Thus many of these classes would have other relationships and many more attributes and methods, making their interac-

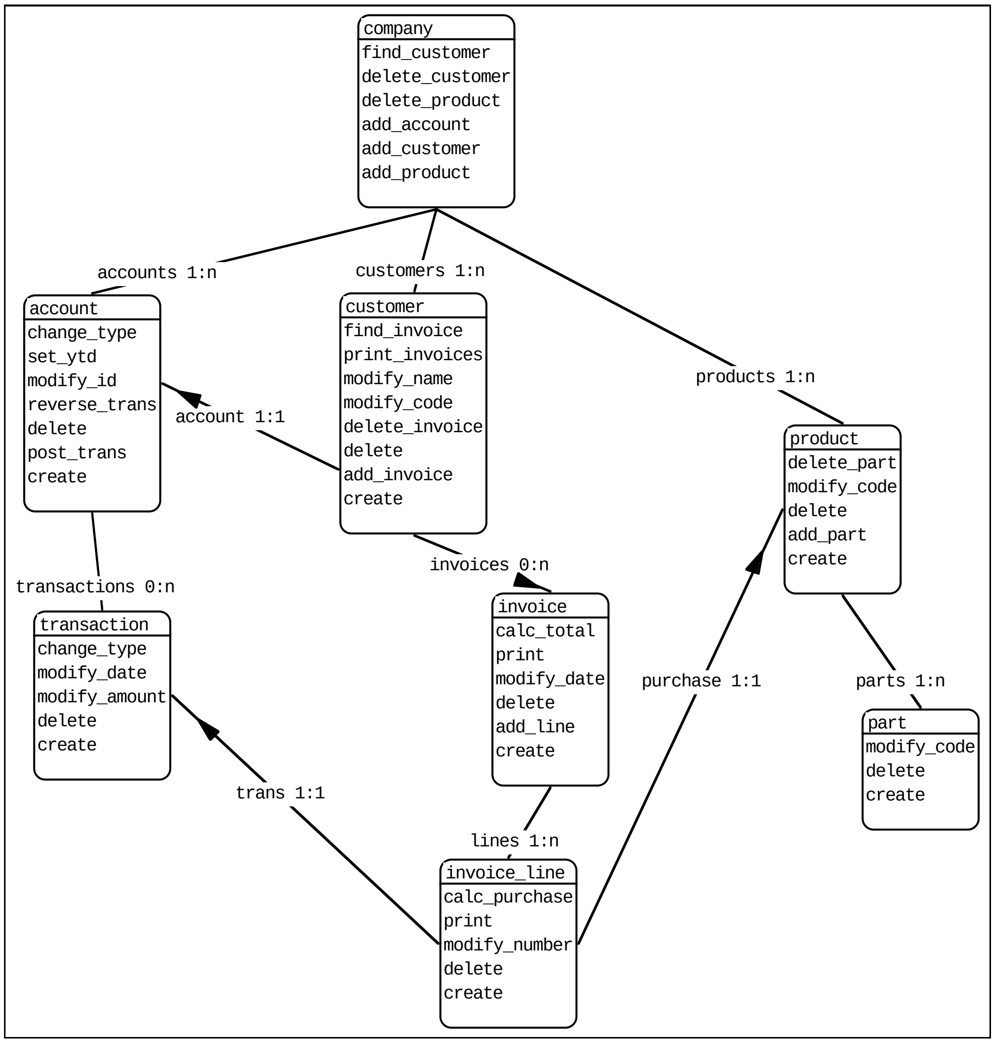

Figure 2: Analysis of Invoicing System Functional Requirements

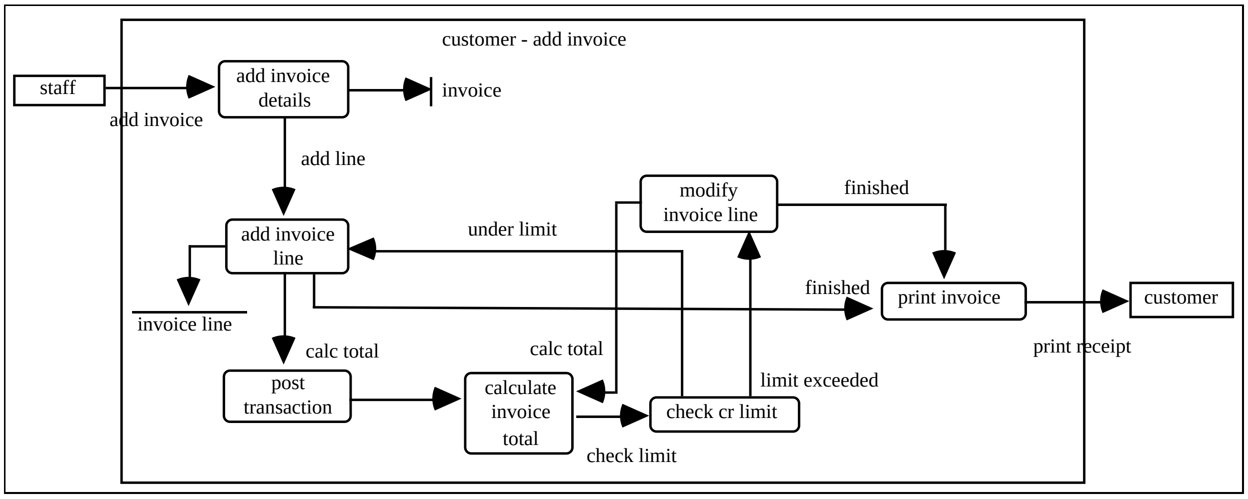

Figure 4: Functional Specification for Adding an Invoice

tions with each other very complex.

3 Traditional OO Design and Implementation

Using traditional OO design techniques, the analysis model from the previous section is refined to a design which uses only those relationships supported by the chosen programming language. In particular, aggregation and association relationships are replaced with simpler relationships which do not embody functionality of their own. New methods are required to encode relationships' behaviour, and these are added into the objects which participate in the relationships.

Figure 5 shows the resulting object model for the account, transaction, customer, invoice and invoice line objects. Reference attributes have been used to represent the analysis relationships, which, therefore, are no longer explicit in the diagram. For example, the "transactions 0:n" relationship between accounts and transactions in Figure 1 has been replaced by a reference from the account class to the transaction class (via the account's "trans_list" attribute). This refers to the first transaction associated with that account. Each transaction stores a reference to the account's next transaction in the "next_trans" reference attribute, and a reference back to their owning account in the "account_ref" attribute. Other object reference attributes include, "invoices_list", "next_invoice" and "customer_ref" representing the "invoices 0:n " relationship, and "lines_list", "next_line" and "invoice_ref", representing the "lines 1:n " relationship.

Extra operations have been added to these classes to carry out the basic functions of the system. For accounts this includes updating the account balance and YTD balance, changing the account's name and type, and deleting the account's transactions. Transaction operations include changing the transaction date, amount and type. Customer operations include checking the customer credit limit. Invoice operations include checking the customer credit limit, deleting lines and modifying line numbers. Invoice line operations include modifying line dates, numbers and amounts, and updating the invoice line total. All of these methods include extra functionality to ensure that the relationships are maintained. For example, to ensure account totals are updated correctly for the account type, account transactions are only deleted if other objects linked to the transactions are also deleted, ensuring removed invoice lines are removed from their invoice list and their transactions are reversed.

Extra design refinements may be carried out at this stage. These might include improving the efficiency of object linkages, adding extra references to improve access efficiency, designing algorithms to carry out functional behaviour, or caching values as attributes for efficiency.

3.1 Traditional OO Implementation

This design would normally be implemented using traditional OO implementation languages. Operations would be encoded in methods and attribute types would be defined and implemented. The operational behaviour described above would be implemented by sending messages between appropriate objects to carry out an algorithm.

For example, when adding an invoice line the invoice::add_line method would call: the invoice_line::create method, the invoice::lines linked list insert method, the account::post_trans method, the invoice_line::add_trans method, the invoice::calc_total method, the invoice::check_cust_limit method, and then the customer::check_cr_limit method. This algorithm must be carefully encoded to ensure all methods are called in the right order and with the right parameters.

Any change to this algorithm, or any relocation of methods to other classes, would require major modification to several of these methods. This is because much of the functionality is not directly associated with any one particular object, but rather serves to support the relationship between several objects. This functionality must be arbitrarily assigned to one of the objects, or equally arbitrarily split between two or more objects. For the same reason, a similarly complex sequence of method calls would be required when modifying or deleting invoice lines, with many constraints needing to be checked by each method to ensure both the correctness of the operation and correct modifications to related objects.

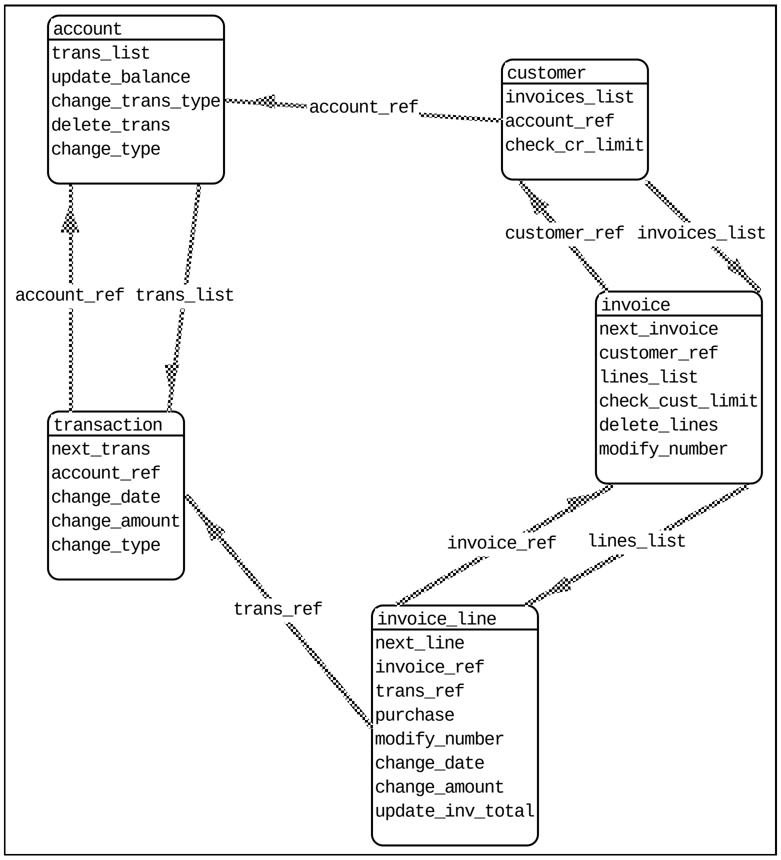

Figure 5: Design using Reference Attributes

3.2 Main Problems of Traditional OO Design

This simple example shows the complexity that is introduced by associating all system behaviour with objects and none with the relationships between objects. Classes quickly become quite large, and class methods quickly start to embody complex calculations and constraint checking dependent on inter-object relationships.

For example, consider the account class. This needs to handle the functionality of the "transactions 0:n" relationship between itself and the transaction class and possibly the "account 1:1" relationship with the customer class. As accounts are used in many parts of an information system, there will be many more relationships from account to jobs, purchase orders, creditors, and so on. If an account or one of these related objects is modified, the account class may have to embody many complex procedures. For example, if the account type is changed, there will be many constraints to check depending on the kind of relationships account participates in e.g., check valid account type for things account related to. This functionality will have to be embodied in one or more methods, which may become large and complex.

The adding and modifying of invoice lines illustrates that constraints and calculated values impose some especially difficult problems. The complex method calling sequences and splitting up of constraints and calculations illustrates this. As another example, consider the calculation of the account balance and the checking of the customer credit limit required when an invoice line is added or modified. It is easy to make errors by omitting functionality or by calculating things in the wrong order. This is compounded if a method must embody a large amount of functionality relating to different functional behaviour, such as invoice::calc_total, used by several invoice line processing functions. Designers and implementers must ensure these methods are called at the appropriate times.

The more relationships an object participates in, the more difficult it becomes to ensure this and to maintain and reuse classes.

Maintenance is made more difficult with methods embodying code for many different system functions. Trying to split up this code into several methods makes the classes much more complex and more difficult to reuse, as method calling sequences and arguments become very difficult to manage.

4 Relationship Oriented Design and Implementation

A relationship oriented design approach preserves the relationships from the analysis phase by representing them as objects in their own right. These *relationship objects* are then allocated responsibility for much of the inter-class relationship functionality. Note that relationship objects are no different from any other objects used in design or implementation, except that they represent analysis relationships, rather than analysis objects. Relationship oriented implementation techniques provide efficient ways of writing these relationship objects, and ensuring relationship behaviour is always carried out even after system modification.

4.1 Relationship Oriented Design

The key idea of relationship oriented design is to use extra objects to represent relationships explicitly. Behaviour which is associated with the relationship can then be put into the relationship object. In this section we present an alternative design for the invoicing system, in which some of the relationships from the analysis model have been converted into explicit relationship objects. The decision on which relationships to convert and which to leave as references is based on how much functionality can be embodied in the relationships themselves.

The alternative design is shown in Figure 6. In this design, all the aggregation relationships have all been refined into relationship objects. These objects exhibit behaviour common to all aggregations, in that when one object is copied or deleted, all other objects participating in the relationship must also be copied or deleted. This behaviour must not only ensure objects are properly copied, but that the new objects are correctly connected together, and that any constraints upon the aggregate are maintained — for example, that totals are recalculated appropriately. Similarly, when an object is deleted, the other participating objects must be disconnected from other objects, disposed if necessary, and any constraints on the aggregate must be maintained.

The aggregation relationships in Figure 6 also embody behaviour specific to their particular place in the system. For example, consider the "account_trans" relationship. This handles the addition of new transactions into the relationship; the deletion of a transaction or all transactions (if the account is deleted); changing of all transaction account types if the account type is modified; updating the balance of the account if a transaction is added, deleted or updated. It could also be extended to handle sorting, printing or copying transactions for the account, and location of single or groups of transactions via various key values. Some operations attempted may violate constraints e.g., the account type change might be invalid as relationships from the account's transactions to other objects may not allow this new account type, or a customer code change might be invalid as another customer already has this code. Such operations should be aborted.

The "invoices 0:n" association relationship between the customer class and the invoice class has been refined to the "customer_invoices" relationship object. This does not embody aggregation behaviour, but does embody behaviour about the relationship between customer and invoice objects. For example, if an invoice number or date is changed, any lookup table for finding invoices for customers needs to be appropriately updated, and the operation aborted if it would duplicate invoice numbers. This relationship also handles the creation and deletion operations on invoices previously handled by the customer class itself, thus reducing the customer class size and complexity.

The two 1:1 association relationships involving the invoice_line class (the "trans 1:1" relationship with the transaction class, and the "purchase 1:1" relationship with the product class) have not been refined to relationship objects.

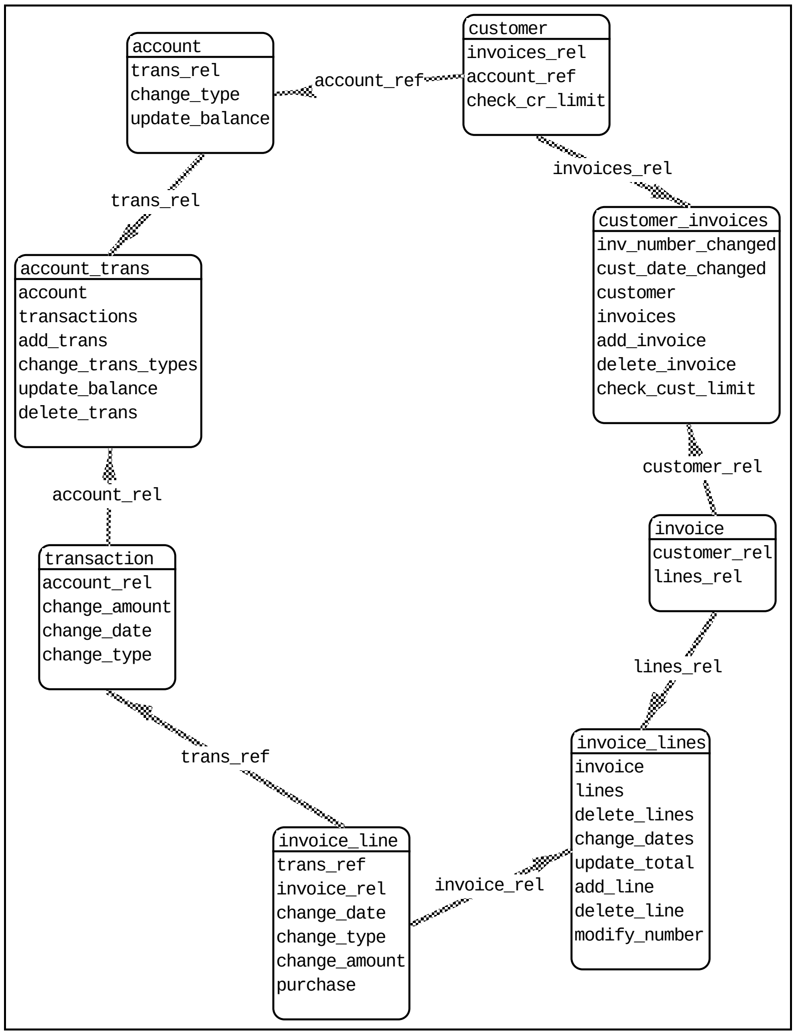

Figure 6: Design using Explicit Relationship Objects

This is because it would be rather expensive in terms of object storage to do so, as there will be many transaction and invoice line objects. Also, if relationship objects were used to represent these relationships, almost all of the "trans 1:1" relationship behaviour, such as changing dates and amounts and adding or reversing transactions would be factored from the invoice_line objects into this relationship object, and the invoice_line objects would simply be placeholders for one end of the relationship.

4.2 Relationship Oriented Implementation

There are several implementation approaches which can be adopted in order to implement the relationship objects described above. The most conservative simply involves associating methods with these objects which are called at appropriate times. More abstract approaches include the notification of an object's dependents, the propogation of reified update descriptions, and the automatic monitoring of participating objects. These approaches allow programmers to ensure functionality associated with relationship objects is carried out at the correct times, and result in less complex and much more easily extended and maintained systems.

Methods in Relationship Objects

The relationship oriented design in Figure 6 uses classes to define the relationship objects. These relationship classes incorporate references to the objects participating in the relationship, and methods to manage adding and deleting the objects being related. Objects in relationships keep a reference to each relationship they participate in.

Methods embodying functionality particular to the relationship are then associated with these relationship object classes, and the objects participating in the relationship are modified to call them at appropriate times. Some methods can be given very general names, such as "check_constraints", "object_deleted", "object_updated", and so on. This allows methods in participating classes to be implemented so that they always call these standard relationship methods. Extra system functionality or modifications can then be incorporated more easily than in traditional OO implementation, as they are added in standard places (the relationship objects).

Libraries of relationships with useful, generic functionality can also be built up. For example, generic aggregation relationships can support an "object_updated" method which always sends an update message to every other object participating in the relationship when one of the objects is updated.

Relationships as Dependents

A more abstract approach to the implementation of relationships is via a dependency mechanism as found in Smalltalk (Goldberg and Robson 1983) or the Observer pattern (Gamma, Helm, Johnson, and Vlissides 1994). Each object which participates in relationships stores their relationships in a list of dependents, rather directly referring to their relationships. When a participating object is changed, it sends a single "update:" message to all relationships on its list of dependents, rather than sending a message to each relationship individually.

For example, if a transaction object is added, modified or deleted, the TransOfAccount relationship's update method is called and this then updates the account's balance appropriately (see Figure 7). If the update method is given arguments describing the change to related objects, then it can be implemented more efficiently. If update is called for TransOfAccount with "trans_amount" as one argument and the amount's old and new values as others, then it is straightforward to update the account's balance. This approach is more abstract than the previous one as objects have less knowledge about the relationships in which they participate. A system designed so all object classes use this protocol (as in Smalltalk) is generally easier to extend.

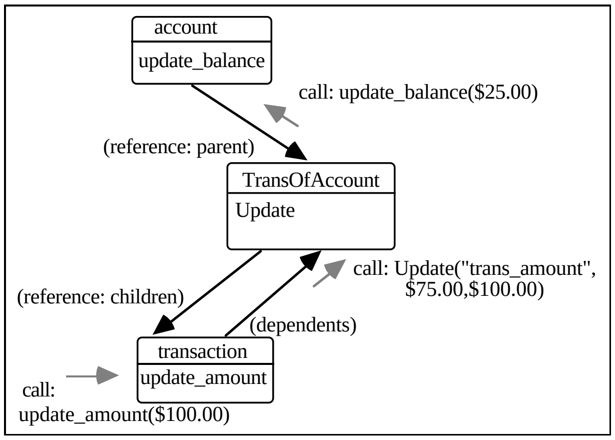

Figure 7: MVC-style Indirect Invocation

Update Description Propagation

A still more general approach is the propagation of reified updates to relationship objects (Grundy and Hosking 1993a; Grundy and Hosking 1994). This is similar to the dependency model described in the previous subsection, as each object stores a list of relationship objects representing the relationships in which they participate. Whenever a participating object receives an event or is updated, rather than simply sending a message to its relationships, it converts the event or update into an object describing the update or event. This "update object" is then sent to all relationships the object participates in. Relationships interpret these update objects and respond to them by updating their participating objects.

Using explicit update description objects is more flexible than simply notifying relationship objects via message-sends. Update descriptions can be forwarded directly to other, related objects. They can also be grouped into transactions of related sequences of updates, stored to document changes objects have undergone, reversed to support undo and abort mechanisms, and can be used to drive efficient, incremental constraint and calculation mechanisms (Grundy

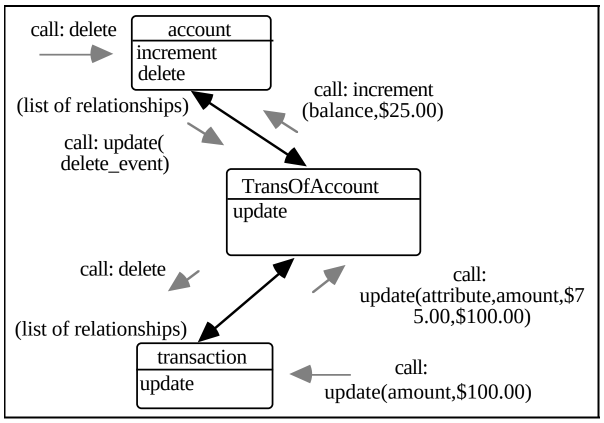

Figure 8: Update Description Propagation

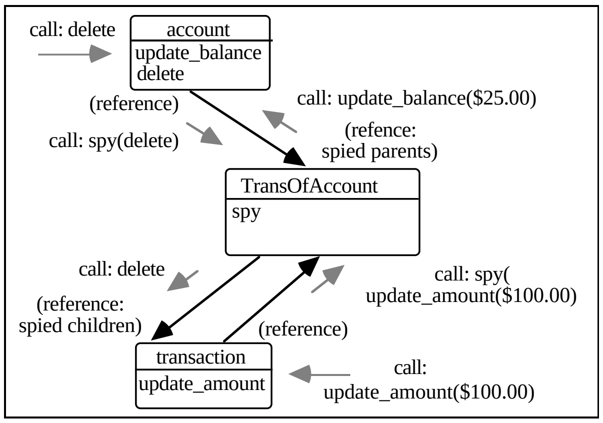

Figure 9: Monitoring Abstract Updates

and Hosking 1994). Figure 8 shows the propagation of update descriptions between objects.

As long as objects utilise a standard set of descriptions, this approach is very general. It is very easy to build up libraries of generic, reusable relationship objects which support aggregation, constraints and transactions (Grundy and Hosking 1995).

Monitoring Abstract Updates

In all these approaches, methods which update objects or correspond to important events must be annotated to notify the objects' relationships by sending either messages or update descriptions (Brown and Hershberger 1991; Grundy and Hosking 1993b). In a reflexive language (Maes 1987) relationships can monitor the execution of the participating objects, and automatically detect changes and generate update descriptions (Noble and Groves 1992; Noble, Groves, and Biddle 1995). This increases the independence of objects and the relationships in which they participate, as methods in participating objects need not be annotated, and objects do not have to be modified to participate in relationships. Figure 9 shows an example of this approach.

4.3 Advantages of Explicit Relationships

Using this relationship oriented design and implementation approach, objects tend to be smaller and more reusable. For example, the relationship oriented version of the account class embodies much less functionality, as much has been abstracted out into its relationships. This version is much smaller, as it concentrates on modelling the core functionality of an account, and so should be more reusable. Any new relationships in which account participates, or modifications to relationships or system functionality, can be more readily incorporated by associating them with appropriate relationship objects.

It is generally easier to understand and modify a design or implementation based on this approach, particularly if the behaviour associated with the relationship between objects changes. For example, as the behaviour associated with account class relationships is now abstracted into methods of relationship objects, new constraints and functionality associated with these relationship objects is easier to add to the evolving system design. In addition, if the relationships themselves change, it is easier to identify where the behaviour associated with the old relationship structures can be modified to suit the new structures.

Explicit relationship objects markedly improve the traceability of object oriented development. Every object in the design (in Figure 6) can be traced back to either a single object or single relationship from analysis (Figure 1). Every analysis object can be traced forwards to a single design object, and every analysis relationship can be traced forwards either to a single object, or a single reference attribute.

5 Discussion

There are several possible objections to the use of relationship objects. Using relationship objects may be decried as "not being pure object orientation". This criticism presumes there is a definition of "pure object orientation", and that such a definition does not include objects which can be seen to represent relationships between other objects. From an object oriented modelling view, where objects are supposed to represent abstractions in the domain of the real world, objects representing relationships may be problematic since they do not represent something in the real world. From our perspective, relationship objects are important precisely because they do represent important things in the real world — although, they represent things that are intangible. If the relationships were not part of the real world, they would not be in the analysis model and would not need to be implemented within the program.

An object oriented design using relationship objects may also seem more complex than an equivalent design without relationship objects: in particular, the relationship oriented design will use more objects. If both programs are basically equivalent, then the information about the relationships must be contained somewhere in the program, and in the more basic design it will be spread across several objects. Thus, although the more basic design will involve less objects, the individual objects will be larger, and their mutual interactions will be more complicated.

5.1 Cohesion and Coupling

The use of relationship objects decreases the size of the participating objects, as data and behaviour are relocated into the relationship objects. Relationship objects simultaneously increase the cohesion of the participating objects, as those objects now contain only a core of data and behaviour, while reducing coupling, as the behaviour within the participating objects relating to each other has been moved into the relationship object. Relationship objects will, of course, introduce coupling between the participating objects and the relationship objects, however, this will be many-to-one coupling, since the many objects participating in a relationship are coupled to one relationship object. This should be more manageable than the many-to-many coupling required if the responsibility for maintaining the relationship is spread over all participating objects. It is important to realise that, since the relationship was identified in analysis and is part of the program's model of the real world, the classes must be coupled in some way to be able to implement the program's requirements[1].

5.2 Encapsulation

Extra relationship objects existing "outside" their participating objects may also be seen as breaking the participating object's encapsulation (Rumbaugh 1987). The first point to note here is that many relationships occur between objects which are themselves parts of another aggregate object: that is, the relationship and the participating objects may all be encapsulated by another object. The second point here is that if encapsulation is broken by the relationship, this is because the encapsulated objects need to be accessed by the relationship object in order to implement the semantics of the relationship. Without the explicit relationship object, the analysis relationship would have to be implemented in another way, by being built in to the participating objects. If the relationship requires access to the "inside" of an object, breaking its encapsulation, these objects would therefore need to break each other's encapsulation anyway. In short, using an explicit relationship object cannot worsen breaches of encapsulation. The root of the problem is not the relationship *object* (i.e. how the relationship is implemented), but the existence of the relationship as part of the problem domain.

In some circumstances, relationship objects may actually increase encapsulation, as the implementation of the relationship itself becomes encapsulated against the participating objects when it is moved in to a separate relationship object.

[1]Colloquially, *"There ain't no such thing as a free lunch".* (Raymond and Steele 1993)

5.3 Implementation

Finally, there is the question of implementation: are programs which use relationship objects larger or slower than those which do not? At this point we are not aware of any systematic studies to determine the matter, however, we believe that there is no reason why programs which use relationship objects should be greatly slower that those which do not. Various types of relationship objects have been used successfully in performance-sensitive applications such as CASE tools (Grundy, Hosking, Mugridge, and Fenwick 1994) and user-interface systems (Maloney, Borning, and Freeman-Benson 1989; Berlage 1992). The Hotdraw event-based drawing tool (Johnson 1992) was converted to use relationships (in this case, arithmetic constraints) with no noticeable change in performance (Freeman-Benson 1993). Finally, it appears that compilers will be able to use information about relationships (when explicitly retained in programs) to optimise programs' execution (Vander-Zanden 1994).

5.4 Finding the Objects

Designing programs using explicit relationship objects involves three stages: first, important relationships from analysis are reified as objects; second, data and behaviour associated with a relationship are moved into the relationship object from the other objects participating in the relationship; and third, implicit invocation and reified events can be used to link the relationship objects to the other participating objects.

This suggests an alternative perspective on the technique of designing with explicit relationship objects. A fundamental problem in object oriented analysis is how to "find the objects", since the objects are the foundation on which the remainder of the analysis will be built. Many techniques for finding objects have been proposed, including underlining verbs (Booch 1994), using index cards (Wirfs-Brock, Wilkerson, and Wiener 1990), and design patterns (Gamma, Helm, Johnson, and Vlissides 1994) — Henderson-Sellers (Henderson-Sellers and Edwards 1994) provides a comprehensive survey. Explicit relationship objects can be seen as another technique which can be used to find objects: one takes analysis relationships and refines them into objects.

6 Related Work

Several of the published OO methodologies do address the difference between analysis (with relationships) and design (without relationships), however we consider that this support is quite superficial. For example, the MOSES (Henderson-Sellers and Edwards 1994) and OOADA (Booch 1994) methodologies effectively include two sets of notations for relationships — one set for relationships at the analysis level (such as aggregations and associations), and a second set for relationships at the implementation level (such as reference attributes, embedded objects, and friends). These methodologies provide little guidance on the design process required to deal with analysis relationships, other than to note that relationships must be manually translated into patterns of reference attributes (as in the example in Section 4) or collection classes (in languages like Smalltalk).

Rumbaugh's OMT (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen 1991) methodology does take greater cognisance of the existence of relations in design. However, OMT's "relation objects" simply link the objects participating in a relationship, in a similar manner to the relations in relational databases (Rumbaugh 1987). Using attributes to control operation propagation over relation objects (Rumbaugh 1988) does allow relation objects to distribute control flow to their participating objects, but does not support application-specific functionality within relation objects. Rumbaugh also advocates language support for relation objects, and in his "Object-Relational" language DSM (Shah, Hamel, Borsari, and Rumbaugh 1989) (and implicitly in OMT), relations and classes are equally important constructs. In contrast, we have described how analysis relationships can be modelled by objects in standard OO languages.

Soukup's Pattern Classes (Soukup 1994b) are distantly related to relationship objects (or relationship classes), even though they were proposed for implementing design patterns (Gamma, Helm, Johnson, and Vlissides 1994) rather than implementing relationships. Soukup's example Composite pattern class (Soukup 1994a) has several features in common with a relationship object: it represents a relationship between objects, and the relationship is managed via the relationship object rather than by directly accessing the participating objects. Pattern classes are distinguished from our relationship objects in that pattern classes contain only the executable code necessary to maintain the relationship: the relationship's data are distributed into the objects participating in the relationship.

Research continues into the modelling of relationships in object oriented design and programming languages. For example, Civello (Civello 1993) describes a set of categories which can be used to classify relationships at the design stage. Kristensen (Kristensen 1994) identifies two categories of association relationships and describes language extensions which can be used to implement these relationships.

6.1 Existing Systems using Relationship Objects

One of the earliest uses of objects to model relationships was the addition of constraints between objects' attributes. A constraint is much like a formula in a spreadsheet which defines a relationship between spreadsheet cells. When a cell changes, other cells are recalculated so the relationship between cells is maintained. In ThingLab (Borning 1981) objects' attributes are linked by constraints, which can read and update other objects. When one attribute changes, the constraint causes other attributes to be recalculated. More recently, systems such as Garnet (Myers, Guise, Dannenberg, Vander Zanden, Kosbie, Pervin, Mickish, and Marchal 1990), Rendezvous (Hill, Brinck, Rohall, Patterson, and Wilner 1994) and Snart (Fenwick, Hosking, and Mugridge 1994) have added constraints to the basic object model. In these systems, constraints are modelled with objects; these constraint objects are thus special cases of relationship objects.

6.2 Relationship Objects in Smalltalk

The evolution of the Smalltalk system also provides several examples of objects being used to explicitly represent relationships, resulting in a simpler design. In the original Smalltalk-80 system (Goldberg and Robson 1983), a Smalltalk object can have dependents — other objects which are notified if the primary object changes. Section 4.2 described how the dependency mechanism can be used to implement other relationships; however, this mechanism can be viewed as a relationship in its own right. In Smalltalk-80, behaviour to add and remove an object's dependents, and to broadcast change notifications is included in the Object class, which also maintains a global database of objects' dependents. ObjectWorks (ParcPlace Systems 1992), a later version of Smalltalk, provides the Model class which improves the handling of dependents. The Model class records its dependencies in an instance of a DependentsCollection. The DependentsCollection both stores an object's dependents, and contains special behaviour for accessing them and broadcasting change notifications. The Model class does not require the global dependents database, or behaviour to manage dependencies. From the perspective of this paper, the DependentsCollection explicitly represents the relationship between an object and its dependents.

VisualWorks (ParcPlace Systems 1994), an even more recent Smalltalk version, also uses another kind of explicit relationship object to enhance the ObjectWorks implementation of dependents. Many Smalltalk objects need to inform their dependents whenever one of their attributes changes. Using Models and DependentCollections, change notifications can be efficiently broadcast to an object's dependents, but the change notification must be triggered explicitly by the programmer. VisualWorks introduces the ValueHolder class (Woolf 1994) which can automatically generate the required change notifications. A ValueHolder is interposed between a primary object and another object which is the value of the primary object's attribute. To change the attribute, the primary object stores the new attribute value into the ValueHolder, and the ValueHolder automatically generates the required change notification. ValueHolders are thus essentially relationship objects which model the relationship between an object and its attributes.

7 Conclusion

Traditional object oriented methodologies have difficulty dealing with relationships. In particular, relationships are used in analysis but not in design or implementation, and this counteracts some of the benefits of seamlessness and traceability generally produced by object oriented techniques. We have described how using objects to represent relationships explicitly can finesse this difficulty, allowing the relationships to persist from analysis through design and into the implemented program. Using explicit relationship objects typically makes programs smaller and easier to understand, and thus quicker to write and to maintain.

We intend to continue our investigations into the use of explicit objects to represent relationships in object oriented development. In particular, we are interested in the methodological implications of relationship objects. For example, if relationships in design and implementation are best represented as objects, how much special consideration of relationships in notation and process activities is needed? We are continuing to refine experimental systems we have built which use relationship objects (Noble and Groves 1992; Grundy, Hosking, Mugridge, and Fenwick 1994), and are also very interested in finding further examples of existing systems where relationships are modelled by objects.

Acknowledgements

We would like to thank Lindsay Groves for his helpful comments on the first draft of this paper. This work is supported by the Centre for Object Technology, Applications, and Research at the University of Technology, Sydney, and the University of Waikato.

References

Berlage, T. (1992). Using taps to separate the user interface from the application code. In *Proceedings of the ACM Symposium on User Interface Software and Technology.*

Booch, G. (1994). *Object Oriented Analysis and Design with Applications* (Second ed.). Benjamin Cummings.

Borning, A. (1981, October). The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transations on Programming Languages and Systems 3*(4).

Brown, M. H. and J. Hershberger (1991, December). Color and sound in algorithm animation. *IEEE Computer 25*(12).

Civello, F. (1993). Roles for composite objects in object-oriented analysis and design. In *OOPSLA Proceedings.*

Fenwick, S., J. G. Hosking, and W. B. Mugridge (1994). Visual debugging of object oriented systems. In *TOOLS Pacific.*

Freeman-Benson, B. N. (1993). Converting an existing user interface to use constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology.*

Gamma, E., R. Helm, R. E. Johnson, and J. Vlissides (1994, October). *Design Patterns.* Addison-Wesley.

Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

Grundy, J. C. and J. G. Hosking (1993a). Constructing multi-view editing environments with MViews. In *Proc. IEEE Visual Languages Workshop.*

Grundy, J. C. and J. G. Hosking (1993b). The MViews framework for constructing multi-view editing environments. *New Zealand Journal of Computing 4*(2).

Grundy, J. C. and J. G. Hosking (1994). Constructing integrated software development environments with dependency graphs. Working Paper 94/4, Department of Computer Science, University of Waikato.

Grundy, J. C. and J. G. Hosking (1995). Supporting flexible consistency management via discrete change description propagation. Working Paper 95/2, Department of Computer Science, University of Waikato.

Grundy, J. C., J. G. Hosking, W. B. Mugridge, and S. Fenwick (1994). Connecting the pieces. In M. M. Burnett, A. Goldberg, and T. G. Lewis (Eds.), *Visual Object-Oriented Programming*. Prentice-Hall.

Henderson-Sellers, B. (1994). *A BOOK of Object-Oriented Knowlege*. Prentice-Hall.

Henderson-Sellers, B. and J. M. Edwards (1994). *BOOKTWO of Object-Oriented Knowlege: The Working Object*. Prentice-Hall.

Hill, R. D., T. Brinck, S. L. Rohall, J. F. Patterson, and W. Wilner (1994). The Rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer-Human Interaction 1*(2).

Johnson, R. E. (1992, October). Documenting frameworks using patterns. In *OOPSLA Proceedings*.

Kristensen, B. B. (1994). Complex associations in object-oriented modelling. In *OOPSLA Proceedings*.

Maes, P. (1987). Concepts and experiments in computational reflection. In *OOPSLA Proceedings*.

Maloney, J. H., A. Borning, and B. N. Freeman-Benson (1989). Constraint technology for user-interface construction in ThingLab II. In *OOPSLA Proceedings*.

Myers, B. A., D. A. Guise, R. B. Dannenberg, B. Vander Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal (1990). Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer 23*(11).

Noble, J., L. Groves, and R. Biddle (1995). Object oriented program visualisation in Tarraingím. Technical report, COTAR.

Noble, R. J. and L. J. Groves (1992, December). Tarraingím — A Program Animation Environment. *New Zealand Journal of Computing 4*(1).

ParcPlace Systems (1992). *ObjectWorks Smalltalk User's Guide* (4.1 ed.). ParcPlace Systems.

ParcPlace Systems (1994). *VisualWorks Smalltalk User's Guide* (2.0 ed.). ParcPlace Systems.

Pfleeger, S. L. (1991). *Software Engineering: The Production of Quality Software* (second ed.). MacMillan.

Raymond, E. and G. L. Steele (1993). *The New Hacker's Dictionary* (Second ed.). MIT Press.

Rumbaugh, J. (1987). Relations as semantic constructs in an object-oriented language. In *OOPSLA Proceedings*.

Rumbaugh, J. (1988). Controlling propagation of operations using attributes on relations. In *OOPSLA Proceedings*.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991). *Object-Oriented Modeling and Design*. New Jersey: Prentice Hall.

Shah, A. V., J. H. Hamel, R. E. Borsari, and J. E. Rumbaugh (1989). DSM: An object-relationship modeling language. In *OOPSLA Proceedings*.

Soukup, J. (1994a). Implementing patterns. In *Pattern Languages of Program Design*. Addison-Wesley.

Soukup, J. (1994b). *Taming C++: Pattern Classes and Persistence for Large Projects*. Addison-Wesley.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley.

Vander-Zanden, B. T. (1994). Optimizing toolkit-generated graphical interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology*.

Wirfs-Brock, R., B. Wilkerson, and L. Wiener (1990). *Designing Object-Oriented Software*. Prentice-Hall.

Woolf, B. (1994). Understanding and using the ValueModel framework in VisualWorks Smalltalk. In *Pattern Languages of Program Design*. Addison-Wesley.