The Graphical Depiction of Design Patterns

C. W. Irving and *C. Atkinson*
University of Houston - Clear Lake

Abstract

If design patterns are to fulfill their promise of dramatically improving the quality, timeliness and reusability of software it is essential that techniques be found to document them concisely and accurately. Graphical notations offer a potentially powerful approach, but unfortunately current techniques for depicting object-oriented design patterns rely on notations that were initially developed for capturing analysis information, and are thus unable to describe some important aspects of patterns. This paper introduces some notational concepts which are tailored to the description of the implementation-level mechanisms from which design patterns are constructed, and are therefore better suited to their documentation. The concepts are first illustrated through their use in the depiction of the well known Observer (or Publisher-Subscriber) pattern, and then in their depiction of the essential elements of a new design pattern called Advertisement.

1. Introduction

In recent years object-oriented design patterns have emerged as a promising vehicle for improving the quality, timeliness and reusability of software. By capturing the essence of commonly recurring arrangements of classes and objects, patterns allow design experience to be packaged and catalogued in a form suitable for reuse by other engineers. Over time it is expected that widely-used patterns will be catalogued, and that the names of these patterns will become part of the vocabulary of software engineering. Engineers will then be able to develop new applications by identifying, instantiating and assembling predefined design patterns rather than by building complete systems from scratch.

Clearly the accurate, concise and complete documentation of design patterns is a key prerequisite for this vision to become reality. A software engineer interested in using a design pattern in his/her work must be able to understand its structure and behavior without wading through reams of text, or grappling with complicated formal notations. Engineering experience has shown that one of the most powerful ways of accurately conveying detailed information is by diagrams. Therefore, diagrams will probably play an important role in any future scheme for describing the structure and properties of patterns. The current approaches to design pattern documentation already make heavy use of graphical notations ([Gamma et al., 95], [Coad et al., 95]).

In general, patterns can be discerned in virtually all the techniques and models used by software engineers, ranging from large-scale architectural frameworks, to smaller libraries of predefined classes, or toolkits. However, *design* patterns, of the form that have recently received widespread attention in the literature ([Coad, 95], [Coplien and Schmidt, 95] [Gamma et al., 95] and [Pree, 95]) on the whole focus on implementation practice (i.e. design and coding) using object-oriented languages. In particular, they codify ways of combining four fundamental features of object-oriented languages which give them much of their power, and collectively distinguish them from other programming paradigms, namely :

- inheritance
- abstract classes
- polymorphism
- dynamic binding

Polymorphism and dynamic binding are closely related mechanisms which are often confused. Polymorphism is the mechanism which allows instances of a class to be substituted by instances of its subclasses. Thus, polymorphism allows different servers to service the same client at different points in a programs execution. Dynamic binding is the mechanism by which the appropriate method implementation is determined when the client invokes a method of the server.

The notations currently used to describe design patterns are adapted from notations originally developed for analysis, such as the Object Modeling Technique (OMT) [Rumbaugh et al., 91] and Coad/Yourdon [Coad and Yourdon, 91]. Unfortunately, these focus on the description of high-level, problem-oriented concepts and tend to neglect some of the important implementation-level features used in the application of design patterns. Although this abstraction mismatch has to a certain extent been alleviated by the addition of new implementation-level facilities (such as pseudo code annotations and creation relationships [Gamma et al., 95]) these notations still fail to describe some of the most important aspect of design patterns. In particular, they do not explicitly indicate where polymorphism and dynamic binding are being exploited, nor define the set of objects which may be used to service a particular client (polymorphically) at run-time. In other words, although notations such as OMT and Coad/Yourdon do a good job of describing the first two features in the list above, they provide little support for the latter two features.

In this paper we introduce a set of notational concepts which are specifically tailored to the description of the implementation-level abstractions used in the application of object-oriented design patterns. These concepts are currently embodied in a notation called ION (Implementation-Oriented Notation) [Atkinson and Izygon, 95] which was developed to complement OMT in a NASA reengineering project [Eichmann, 95]. ION reuses many of the best features of existing object-oriented notations such as OMT, Booch [91] and Fusion [Coleman et al., 94], but recasts them within a more consistent and comprehensive framework that ignores analysis level issues. By focusing exclusively on implementation concerns, ION is intended to fill a perceived gap in the range of notations available to software engineers, and thereby to complement existing languages and methods. We believe this notation offers some concepts which provide a consistent, readable platform for documenting design patterns, and can graphically depict many of the implementation details that make them work. The burden on the associated textual description to describe every little implementation detail can thus be reduced.

In the following section we introduce the notational concepts by using them in the depiction of a familiar pattern called *Observer* by Gamma et al. [95] and *Publisher-Subscriber* by Coad [95]. The use of a common example also allows a direct comparison of the various notations. We then illustrate the concepts further by using them to depict a new pattern which we call *Advertisement*.

2. Pattern Notations

The Observer pattern has its roots in the Model-View-Controller (MVC) framework of Smalltalk [Krasner and Pope, 88] which was devised as a means of separating the presentational aspects of a user interface from the application data. MVC achieves this by packaging the presentational details within distinct objects which are notified by the application object(s) whenever the relevant data values are changed. The Observer pattern is a generalization of this scenario which consists of two high-level participants: the “subject” (or “publisher”) having a state that needs to be displayed and the “observer” (or “subscriber”) providing the viewing mechanism for the state information. A particular subject is related to zero or more observers, each of which needs to be notified of changes in the subject's state.

Coad's representation of this pattern is shown in figure 1.

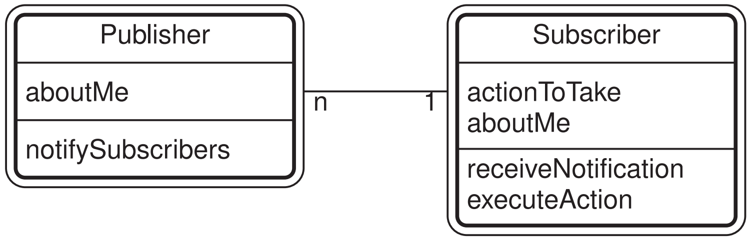


Figure 1. Publisher-Subscriber Pattern: Coad Notation

The rounded rectangles in figure 1 are examples of a “class-with-object”, that is, a normal class which can be instantiated. A round edged rectangle without the external (thin) boundary represents a class that cannot be instantiated, in other words an abstract class. Each rectangle is divided into three sections, the top section containing the name of the class, the middle section listing the attributes of the classes, and the bottom section listing the methods (or services in Coad's terminology).

The line between the classes indicates that they are related in some way. Connection constraints at the ends of the line define how many relationships of that type a particular instance of the class can have. Two special kinds of relationship are available : generalization/specialization (inheritance) relationships, indicated by semicircular icons, and whole-part (aggregation), indicated by triangles.

The other main diagram type supported by the notation, but not illustrated in figure 1, is the scenario view. This is a form of interaction diagram which indicates how objects interact to implement a service provided by the system.

Gamma et al. use two diagrams to depict their version of the Observer pattern: a class diagram, illustrated in figure 2, and an interaction diagram, illustrated in figure 3. Together these provide much more information about the structure and behavior of the pattern than Coad's diagram.

The notation for class diagrams is heavily influenced by OMT. A class is represented by a rectangle which has the familiar three sections containing its name, attributes and methods . However, unlike OMT, (and most other notations including Coad's) the methods are listed in the central section and the attributes in the lower section. Symbols retained from OMT include the diamond to represent aggregation (not used in Figure 2), the triangle to represent inheritance, and the small circle to represent associations with a multiplicity of “many”.

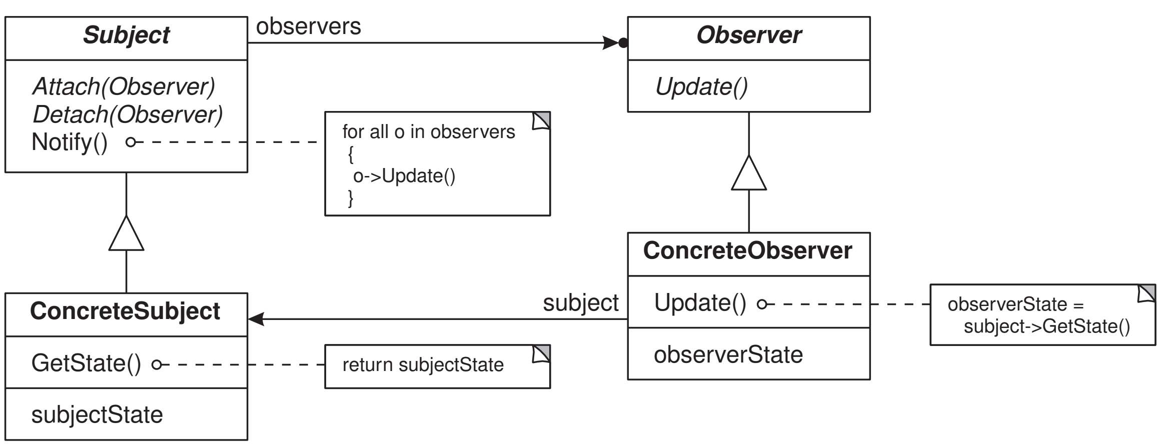


Figure 2. Structure of the Observer Pattern: Gamma et al. Notation

There are three main extensions to the OMT base which are particularly useful for depicting design patterns:

- abstract classes and methods are highlighted by an italic font,
- small pseudo code fragments may be attached to methods to describe their implementation,
- dashed arrows can be used to indicate the creation of instances of a class by another class.

The first two of these are used in figure 2 to show that the Subject and Observer classes are abstract classes with concrete subclasses. The purpose of this arrangement is to introduce type-compatibility between observers of a common concrete subscriber class, and make the subscriber unaware of the precise nature of its observers.

Although not used in this pattern, Gamma et al. also use object diagrams to depict the run-time relationships between typical class instances. In such diagrams objects are represented by rounded rectangles.

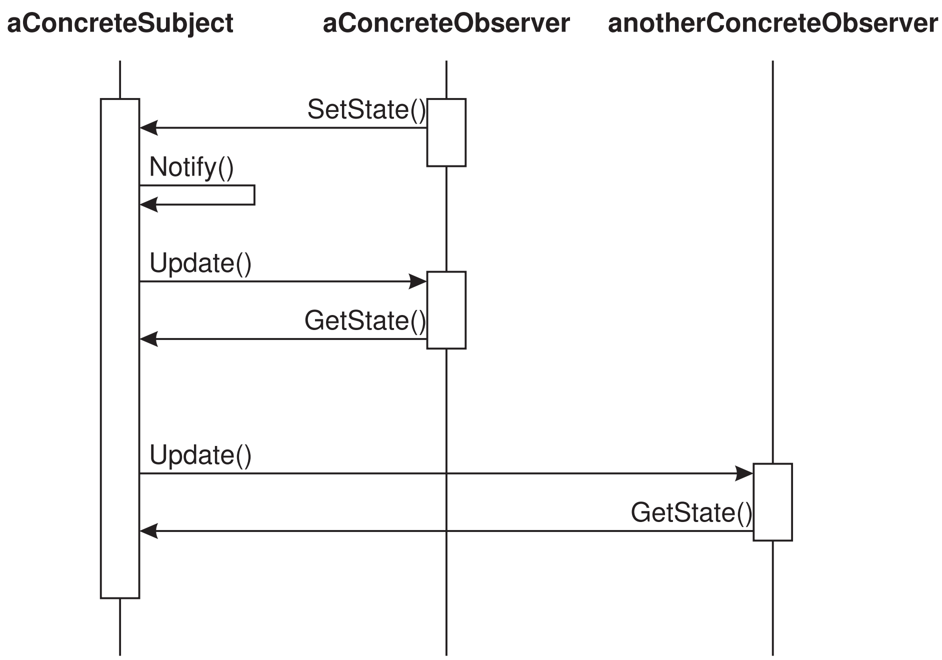


Figure 3. Interactions in the Observer pattern: Gamma et al. Notation

The third type of diagram used in Gamma et al. is the interaction diagram. This type of diagram illustrates a typical sequence of run-time interactions between typical objects. As illustrated in figure 3, the objects are represented as vertical lines and interactions as horizontal arrows crossing between them. This shows that after the subject's SetState() method has been called to initialize the structure, an invocation of its Notify method causes it to send Update() messages to each observer. These respond by invoking the subject's GetState() method.

In both the Coad [95] and Gamma et. al [95] approaches, these diagrams are accompanied by supporting text. Although they serve the need for a visual foundation upon which the rest of the documentation can be based, the diagrams have a number of shortcomings. In particular, they give little, if any, information about:

- how inter-class and inter-object relationships are implemented,
- how the interaction structure is established (i.e. how objects gain visibility others),
- how the structure changes over time,
- what information is exchanged in method calls,
- which interactions (method calls) are alternatives or repeated,
- where polymorphism is being used,
- where dynamic binding takes place.

In the following subsection we illustrate how a more implementation-oriented notation can address a lot of these problems, and provide more complete documentation of the Observer pattern.

2.1 An Implementation-Oriented Pattern Notation

Our approach employs three different types of diagrams -

- class dependency diagrams
- object dependency diagrams
- object interaction diagrams

The first two are taken from ION [Atkinson and Izygon, 95], with minor adaptations. The third type of diagram is obtained by combing object icons from ION with Fusion-style interaction graphs [Coleman et al., 94]

2.1.1 Class Dependency Diagrams

Figure 4 illustrates the use of the first type of diagram to capture the relationships between the classes. Classes are represented as rounded rectangles with two horizontal lines in the center. These lines are important because they distinguish classes from objects and split the class icon into the three familiar regions: class name, attributes and methods. They are also the means by which abstract classes are distinguished from normal classes, the former having dashed horizontal lines and the latter full lines.

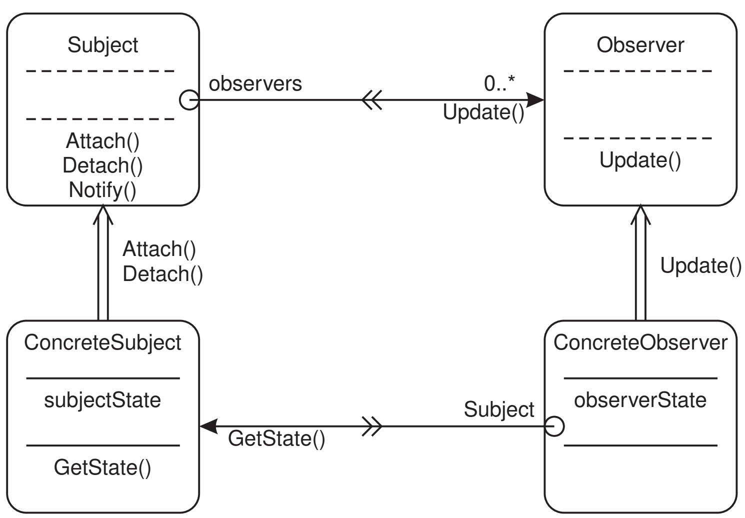


Figure 4. Observer Class Dependency Diagram

For the purpose of depicting patterns only two kinds of relationships are recognized in class diagrams: clientship and inheritance. A clientship relationship is an asymmetric relationship which indicates that an instance of one class, the client, has direct visibility of an instance of another class, the server. In implementation terms, the client holds a reference to, or the value of, the server. Clientship is therefore similar to the more abstract associations represented in analysis notations, but describe the relationships as they actually exist in the implementation. For example, the white circles at the beginning of the clientship arrows in figure 4 indicate that the client holds a reference to the server.

In total, four different possible implementation options apply to a clientship relationship in a class dependency diagram. These are summarized in table 1.

Variation	Icon	Meaning
Attached vs. detached	filled vs. hollow circle at client end of arrow.	Attached: Client holds the server by value. Detached: Client holds a reference to the server.
Intimate vs. proper	Arrow head at server inside vs. outside class boundary.	Intimate: Client accesses private server resources. Proper: Client only accesses public server resources.
Direct vs. indirect	Continuous vs. "broken" arrow between client and server.	Direct: Client accesses the server directly. Indirect: Client accesses the server through intermediary class(es).
Permanent vs. transient	Circle at client end of arrow inside vs. outside class boundary.	Permanent: Lifetime of clientship exceeds a single client method execution. Transient: Clientship within the scope of a single client method execution.

Table 1. Summary of Clientship Alternatives

As can be seen from figure 4, inheritance is depicted by a double-lined arrow pointing from the subclass to the superclass.

Both types of relationship can have additional annotations. In the case of clientship, there are four types of optional annotations: procurement annotations, references annotations, call annotations and multiplicity constraints.

Procurement annotations are used to indicate how the client gains visibility of the server. In figure 4, the "double arrow" symbol (<<) in the middle of the upper clientship line, pointing towards the client, indicates that the client imports a references to (or the value of) the server object from an external source via a method call. The double arrows pointing towards the server in the lower clientship line indicate that the client creates the server and exports visibility to it. As a third alternative, if the two arrows are opposed (><), this indicates that the client creates the server but keeps it private. Call annotations are method names placed at the server's end of the arrow, and list the methods which the client calls through the relationship. Thus the annotation Update() next to the class Observer specifies that instances of Subject call this method. Reference annotations are used to identify the names (i.e. variables or parameters) by which the client accesses the server. The name is placed at the client's end of a clientship line. Finally, as in most object-modeling notations, multiplicity constraints indicate how many instances of the related classes may participate in a given instance of the relationship.

In the case of inheritance only one optional annotation is available; the redefintion annotation. This takes the form of method names placed next to the inheritance line, and indicate the set of methods which are provided with a new implementation in the subclass, either because they are redefined, or because they are deferred by an abstract superclass. Thus the Attach() and Detach() annotations next to the inheritance relationship between Subject and ConcreteSubject indicate that the corresponding methods are given new implementations in ConcreteSubject.

One problem which arises when depicting designs patterns is how to judge what information to show and what information to omit. It is usually not possible to depict every detail of a pattern or system of objects in a single diagram, otherwise the diagram would become far to cluttered. Unfortunately, given their origin, notations such as OMT do not provide much notational power to depict when elements of a pattern or system of objects have purposefully been omitted from a diagram. However, the ION notation has a mechanism to handle such cases.

When the designer wishes to show that further clientship relationships, inheritance relationships, instance variables or methods exist in the implementation of an object-oriented program but are not immediately appropriate for the diagram on hand, he/she can insert corresponding "ellipsis" marks indicating their existence. Inheritance and clientship ellipses are shown as shortened versions of their corresponding graphical representation, and indicate that a class has further parents or servers, respectively, which are not shown on the diagram. For example, the Advertisement pattern (section 3) indicates that DataField and FormWindow classes derive from a larger user interface framework (which itself is out of the scope of the pattern) by showing inheritance ellipses for the classes. Instance variable and method ellipses are shown as three dots ("...") appended to the appropriate sections of class icons, and indicate that the class has further instance variables or methods which are not shown on the diagram.

2.1.2 Object Dependency Diagrams

Figure 5 shows an example of an object dependency diagram. There are actually two different types of object dependency diagrams: "object history diagrams" and "object snapshot diagrams". As its name implies, an "object history diagram" shows the various clientship relationships which an object enters into during the life of the system. In contrast, an "object snapshot diagram" shows the various relationships which an object has while it is in a particular state.

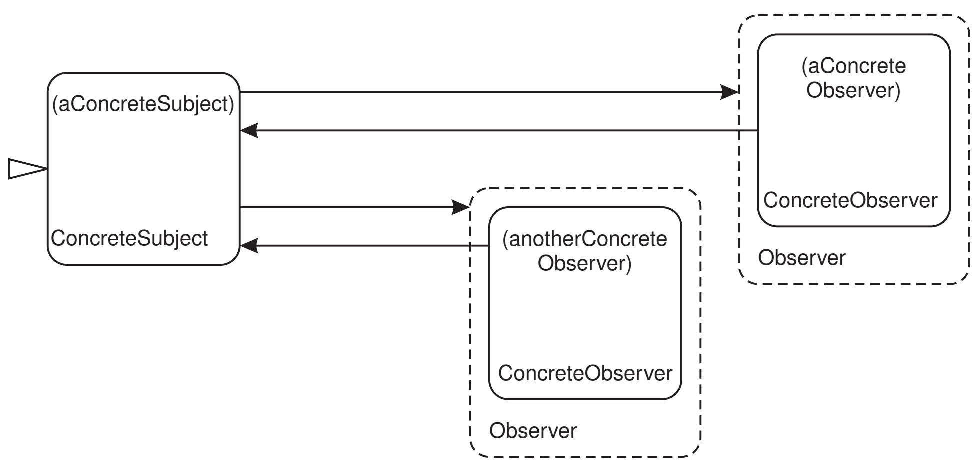


Figure 5. Object History Diagram for Observer.

Another extension to OMT by Gamma et al. [95] is the explicit labeling of creation relationships on class diagrams using dashed arrows. In our notation similar information can be garnered from the visibility decorations used in class dependency diagrams. However, to further highlight situations in which the client is the creator of the server, object dependency diagrams include a special form of clientship relationship known as the creation relationships (denoted by double arrowheads).

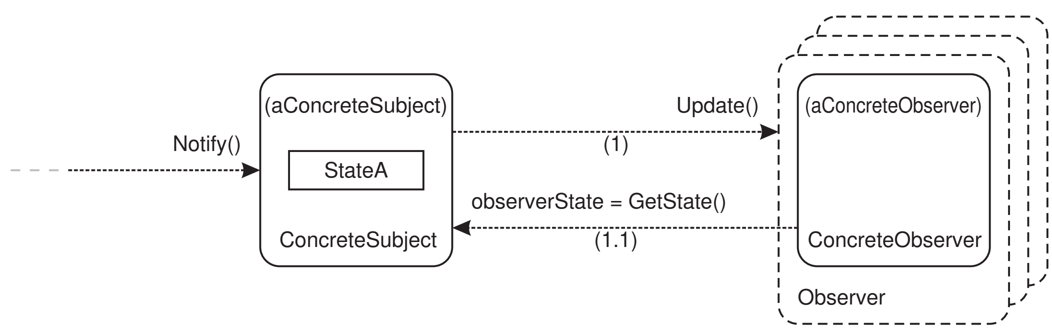
Object diagrams are centered around a particular object, known as the root of the diagram, which is the focus of the reader's attention. Figure 5 shows a history of the dependencies between the aConcreteSubject object and its related observers. Typically the other objects in an object diagram are offspring of the root object (i.e. they are created either directly or indirectly by the root object) although in this case it is clearly not so. The root of a diagram is identified using a thin triangle set with its point against the root object icon.

Polymorphism is shown directly on object diagrams by surrounding server objects with a dashed boundary corresponding to the supertype it is used as. For instance, aConcreteObserver and anotherConcreteObserver in figure 5 are both servers of aConcreteSubject but are referred to polymorphically via pointers to instances of the Observer class. They are thus called *polymorphic servers*.

The object diagram counterpart of class relationship multiplicity is the notion of *sets*. Using the simple notational device of replicating an object boundary three times (in a stacked configuration), the designer shows that a set of objects participate in the given clientship. If the client object is replicated, this means that every member of the set is a client of the server. On the other hand, a replicated server indicates that the client object is in a clientship relationship with each member of the set. By combining this set notation with the concept of polymorphic servers, subtle nuances in the use of polymorphism can be depicted. For example, the arrangement on the right hand side of figure 6 below indicates that the client is related to a set of Observer objects, each of which at run time is substituted by a ConcreteObserver object. On the other hand, if the polymorphic sever were a normal (non set) icon, and the element inside it a set icon, this would indicate that the client is logically related to a single Observer object which, at run-time, can be substituted by one of a set of ConcreteObserver objects.

2.1.3 Object Interaction Diagrams

The third type of diagram is illustrated in figure 6. Unlike object dependency diagrams which show clientship relationships, the arrows in an interaction diagram show dynamic messages (method invocations) which are executed as part of a certain operation. To visually distinguish interactions from relationships, the former are dashed.



States:
StateA: Following attachment of one or more observers.

Figure 6: Interaction Diagram for the Notify() Operation.

The basic structure of the diagrams, and the form of the message annotations, is taken from Fusion [Coleman et. al., 1994]. Thus, every interaction graph has a controller object with an incoming arrow representing the operation being elaborated. In other words, all the interactions depicted are part of the implementation of the incoming message. Every interaction is marked with the name of the method being invoked, and optionally the input and output parameters. In addition, the position of the message in the overall sequence of interactions is indicated as a “sequencer” annotation in the form of a Dewey decimal number. Thus, in figure 6, the aConcreteObserver object invokes the GetState() methods (1.1) as part of the execution of the Update() method (1) invoked by aConcreteSubject.

This diagram also shows that many of the icons available in object dependency diagrams can be reused in object interaction graphs. In particular, the polymorphic server notation can be used to indicate which class of object a client "thinks" it is invoking. Thus, in figure 6, aConcreteSubject believes it is invoking the Update() methods of a set of Observer objects, whereas in fact it is interacting with objects of class ConcreteObserver. The diagram therefore directly shows how polymorphism and dynamic binding are utilized in the pattern.

2.2. Observations

In order to adequately cover both the static and dynamic aspects of patterns, notational support for class, object and interaction diagrams are clearly necessary. We believe the triad of diagrams described above meet this in a simple, elegant and synergistic way.

The class dependency diagrams in our notation are basically the same as the class diagrams used by Coad and Gamma et al. However, the alternative properties of the clientship relationship, and the variety of optional annotations allow considerably more implementation detail to be provided. For example, using the distinction between permanent and transient clientship it is possible to distinguish relationships which last for the duration of the system (e.g. the subject-observer relationship) from those that last only for the duration of one method call. Such transient relationships form the key of some patterns such Visitor [Gamma et. al, 95].

Object dependency graphs represent a larger divergence from the currently used notations. Not only do they provide a sounder conceptual foundation from which to define groups of related objects, providing a clean distinction between history diagrams and snapshot diagrams, but they also support the description of hitherto neglected aspects of patterns. The main example is the explicit depiction of polymorphism using the concept of the polymorphic server.

Object interaction graphs provide an important additional view of object systems over and above that provided by object dependency diagrams alone. The style of interaction graph introduced by Fusion has proved to be very successful in practical software development projects, and is strengthened by cross fertilization with constructs from ION object diagrams. Moreover, this style of diagram is better at showing repetition and optionality than the traditional "ladder" or "railroad" style of interaction diagram [Coleman et al. 94], and can typically show many more interactions. However, engineers who prefer "ladder" diagrams can still construct them using the more general form of interaction diagram presented here simply by elongating the object icons, and organizing interactions between them chronologically.

The choice between interaction diagrams and pseudo code annotations as in Gamma et al. is not so straightforward, however. It is arguable whether a set of interaction diagrams can always economically replace pseudo code annotations when the readership generally is very much in tune with code to start with. There is, however, no reason why pseudo code annotations could not be added to ION diagrams, nor is there any reason to require them.

ION is not the only notation to support these various diagrams. The Booch notation [Booch. 91], in particular, supports many of the features described in the previous sections, and can usefully be applied to the depiction of patterns. However, Booch's notation is somewhat awkward, due to the many irregular and apparently ad hoc icons (e.g. clouds), and complicated, due to the sheer number of different abstractions covering all the main phases of development (analysis, design, implementation). Moreover, to our knowledge, no notation (even Booch) directly facilitates the description of polymorphism and dynamic binding in a manner comparable to ION. The forthcoming Unified Modeling Language [Booch, 96] will significantly simplify and systematize the features provided by Booch and other leading notations.

In the following section we illustrate how this notation would be used in the practical documentation of a pattern as part of a catalog of patterns. The example is a pattern which the authors have found useful in the construction of user interfaces, and is as yet unpublished. The description of the pattern follows the approach pioneered by Gamma et al. [95].

3. Advertisement

3.1 Intent

Allow data exchange between data fields within a data entry window, simplifying the implementation of dependencies between data fields.

3.2 Motivation

Graphical user interface (GUI) applications often have a need for data entry windows where individual data fields are logically related. Fields may have (default) values that can be deduced from others or may have validation mechanisms that are dependent on the value of other fields. We will call the fields that depend on others *data sinks* and those they depend upon their *data sources*.

One approach to implementing such windows is to have the data sink fields explicitly access the contents of their data sources as needed (e.g., during field validation). This approach, however, has two major drawbacks: each data sink field implementation needs to know the identity of its data sources (making fields difficult to reuse within other windows of the same application or other applications in the same domain) and data sinks are not notified of changes to data sources (making it impossible to dynamically update sinks following data source changes).

Another approach would be to use the Observer pattern between data sources and sinks. This approach addresses the issue of inter-field change notifications but, unfortunately, still does not mitigate the need for the participants in the publish-subscribe links to be explicitly identified. The difficulty is especially apparent in systems where a large number of data entry windows containing varying mixes of data sources and sinks must be created. In this case the Observer pattern's requirements for observers to at least be able to explicitly identify their corresponding subjects is rather cumbersome. A connectionless solution similar to Observer would be far preferable.

The key to the Advertisement pattern is to rely on the underlying user interface framework’s ability to iterate over the contents of windows and dispatch messages to their children. The actual implementation of this capability varies from framework to framework (e.g., method calls, windowing system events, etc.) but this capability is always there. By using this capability (although at the cost of some inherent communication overhead), we can decouple data sources and sinks. Doing so allows domain-specific data fields to be reused with a minimum of fuss. In the case of applications with a significant number of data entry forms, the construction and maintenance of the user interface forms is substantially simpler when no explicit inter-field references need to be created or maintained – especially if the forms are complex.

For example, in the case of a banking application, one can imagine a data entry form containing (amongst other things) a bank card number and the corresponding bank name (Figure 7). These fields are dependent: the bank name field is a data sink for the bank card number since the bank name can be determined from the card number (a bank identification number is encoded within the card number). Each data field does not know of the other’s existence, but when the card number field is updated and its value is found to be valid it advertises the availability of a new bank number while the bank name field listens for such advertisements and uses the advertised bank number to update its contents.

Figure 7. Example Data Entry Form.

In implementation terms, this means that we will have a hierarchy of data field classes (probably rooted in a generic “Window” class -- although that is implementation-dependent), initially separating data fields based upon the physical data type they subclass, followed by a domain-specific hierarchy of data field types closely mirroring those found during domain analysis (or analysis of the application, for “one-off” applications). The main participants in our example are shown in figure 8:

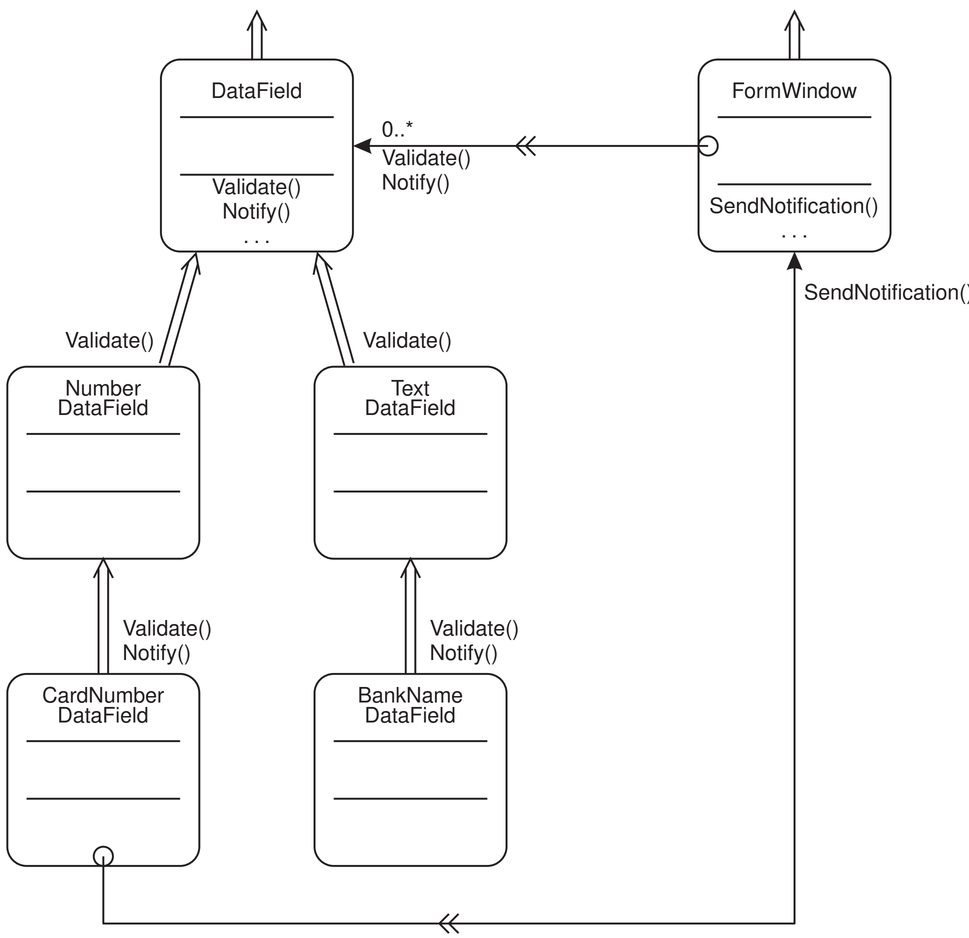


Figure 8. Typical Advertisement Participants

Each data field class has a Validate() method which is called by the environment when the field contents have changed and their validity is to be verified. Each class also implements a Notify() method that is called for each data field when a data source has been updated. The argument to the Notify() method is a notification message object containing a message type identifier, data source identification and the actual updated data. Obviously, since the various data types that are likely to be transferred are not all assignment-compatible, the notification message class needs to be specialized to store the various types involved (numbers and text, in our example):

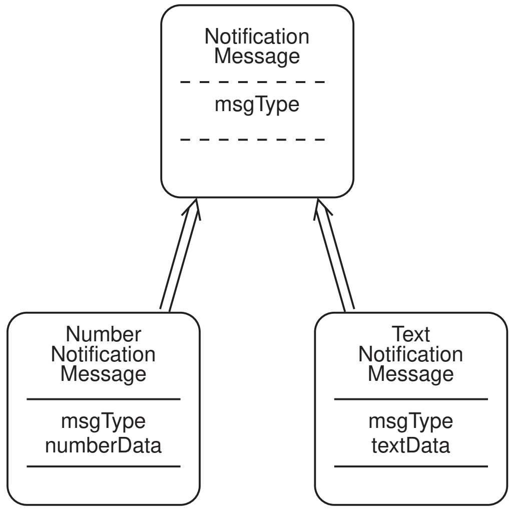
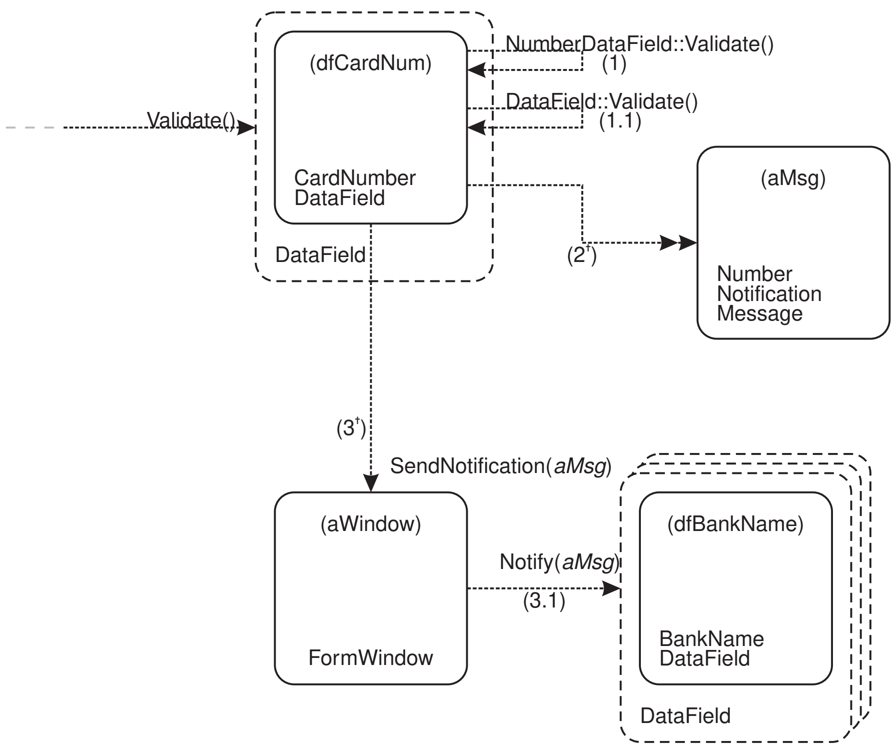


Figure 9. Typical Notification Messages



Annotations:
†: If field validation was successful.

Figure 10. Interactions in Data Source Validation

The usual sequence of events for data propagation is that the card number data field is updated, its contents are found to be valid and are then advertised to the other data fields in the window (which we'll call *form window* in order to avoid confusion with the windowing system's Window base class) through a call to the SendNotification() method of the window class. As shown in figure 10, the SendNotification() method takes a notification message object as parameter, iterates over the window's (data field) contents and sends the notification message to each in turn via a call to their Notify() method.

The situation before the call to the card number object's Validate() method is summarized by object snapshot diagram in figure 11:

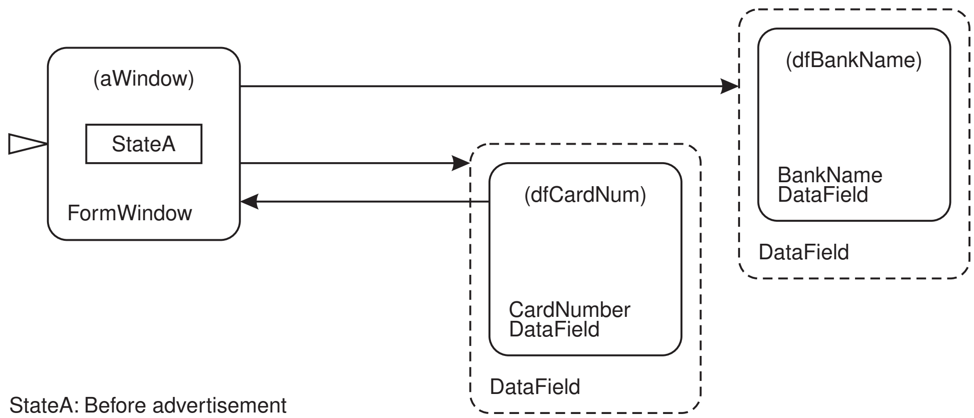


Figure 11. Snapshot of Object Structure before Advertisement

When the form window propagates a reference to the notification message object (arc 3.1 in the interaction diagram), the message object created by the data source is known to all the participants in the pattern: data source, form window and the other data fields -- including the data sink(s):

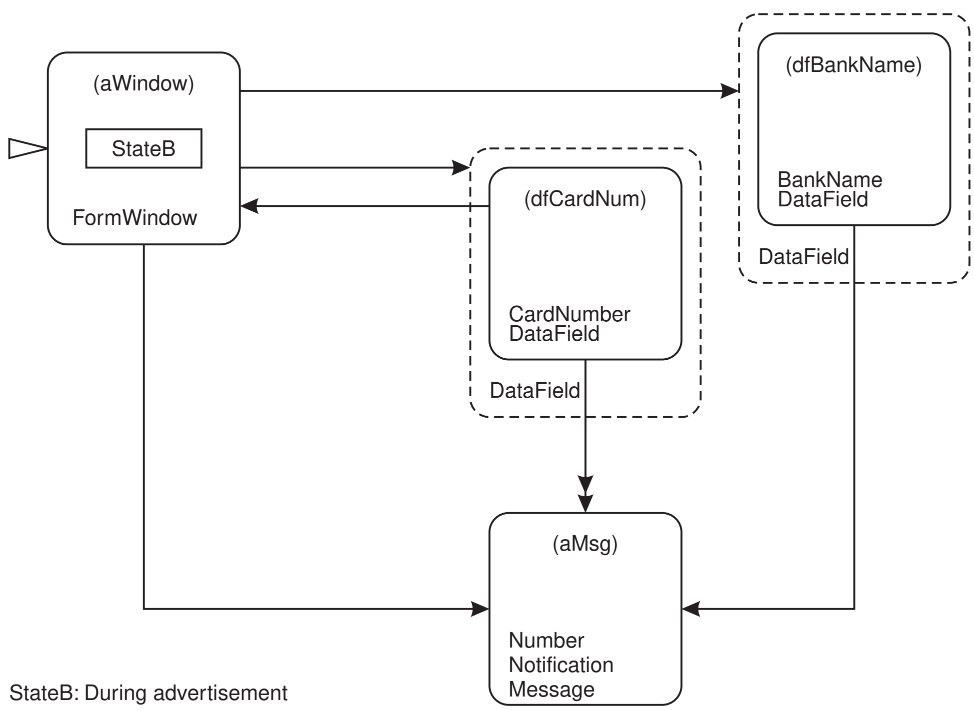


Figure 12. Snapshot of Object Structure during Advertisement

3.3 Applicability

Use the Advertisement pattern when one or more data entry windows in the application (domain) include data fields that depend on each other to:

- Propagate derived values from one field to the other. E.g., bank card number to bank name.
- Validate fields. E.g., operation types depending on bank card capabilities encoded into the card number.

The data propagation requirements must not specifically call for point-to-point communication between fields independently of their (domain, not physical) types.

3.4 Structure

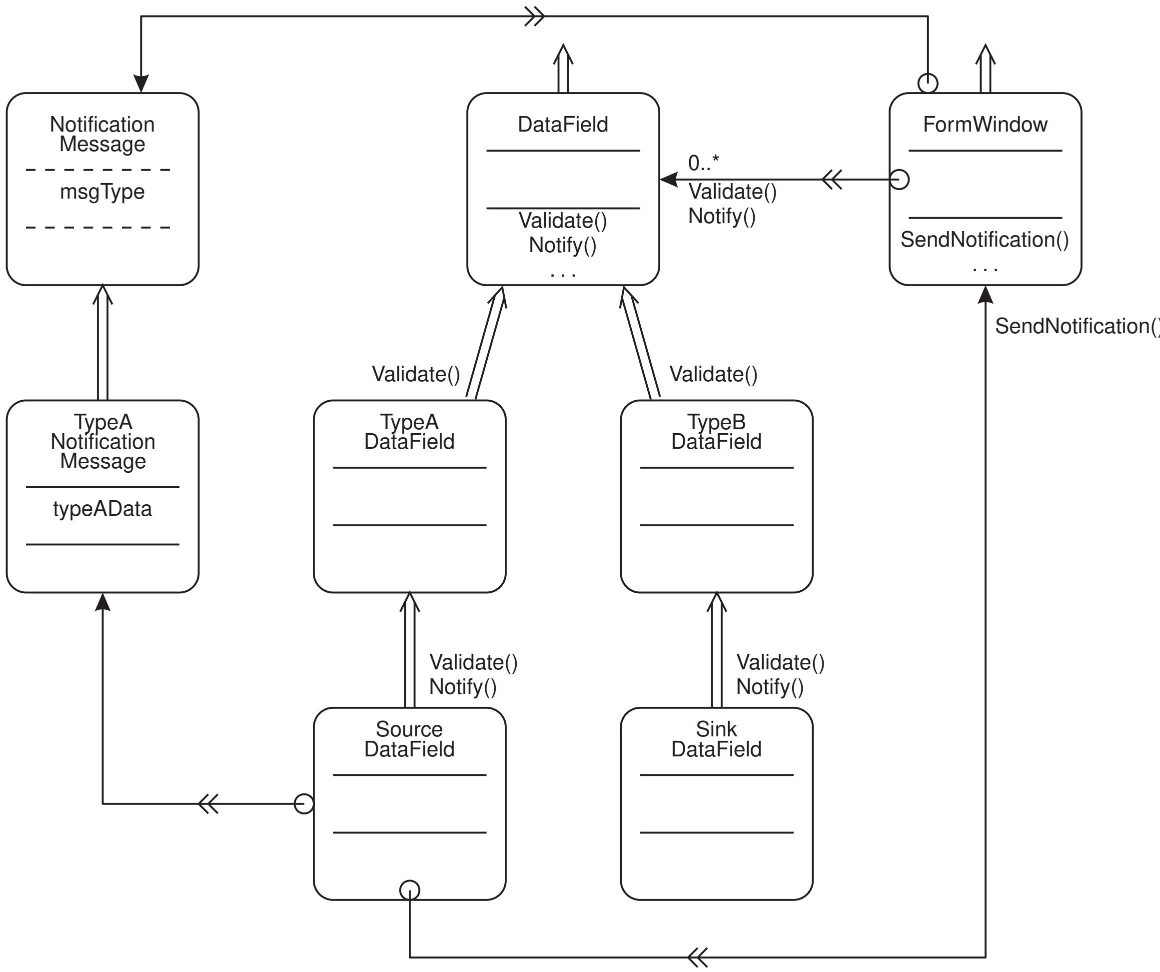


Figure 13. Advertisement Class Structure

3.5 Participants

FormWindow

- Contains all the data fields
- Propagates notifications to the data fields.

Data fields

- Derive type-compatibly from a common "data field" supertype.
- Support a validation method (i.e., Validate()) which checks the field contents and returns distinct values for valid or invalid (possibly others) field contents.
- Fields that send data to others are called data sources.
- Fields that receive data from others are called data sinks.
- Can be data sources, data sinks or both for the same or different data types.

Notification Messages

- Carry the data from sources to sinks.
- Have a domain type identification
- Derive type-compatibly from a common NotificationMessage class with specializations per physical data type.

3.6 Collaborations

- Specialized data fields can rely on the validation mechanisms provided by their superclasses.
- Data fields pass the notification message they want to advertise to the containing form window. In other words, data fields never deal with other data fields directly.
- Form windows notify their content data fields of available data by calling each one's Notify() method with the notification message as argument.

3.7 Consequences

- Data fields do not need to know of the existence or location of their collaborating peers. Changes to the form layout, contents naming, etc. have no impact on the function of the data propagation mechanisms.
- Data entry windows within the domain can be assembled out of a set of domain-specific data field classes and be fully-functional with respect to validations and data propagation without any customization work whatsoever.
- The overhead necessary to advertise notification messages following data source updates is considerably higher than for other solutions, due to the broadcast nature of the mechanism. This pattern thus imposes a performance penalty proportional to the number of fields present in the same window as the data source.

3.8 Implementation

Method-Based vs. Message-Based processing

Practical implementations of this pattern can either be method call based (like the example given in the motivation section) or message based. Message-based implementations using the underlying windowing system’s message queue use specific application-defined event types defined for data field validation and data notification, sending (as opposed to posting) them through the event handling functions of the corresponding windows instead of calling actual object methods.

Notification Message Implementation

In the case of languages that do not allow dynamic type coercion (or where type coercion is frowned upon: e.g., Ada language coding standards) and in the case of message-based implementations where the underlying event queue does not support inclusion of complex data types in events (e.g., Gupta SQLWindows) it is possible to rely on a global data exchange buffer -- as long as the window manager is single-threaded. One data structure containing one field of each domain data type visible to all participants in the pattern can be used as a “well known” location where data sources place the data they wish to advertise before calling the form window's SendNotification() method and where data sinks read the data from when processing the notification message or Notify() method call.

It is important to have one field per domain data type in the buffer rather than per physical data type because processing of a notification could conceivably result in the advertisement of another data element of a data type that is physically assignment-compatible but of a distinct domain data type. E.g., the advertisement of a card number could cause a bank number to be advertised, both of which are physically *numbers*, but are of distinct domain types. The single common buffer approach is practical in most single-threaded applications because the very nature of the graphical user interface means that only one data entry window will be in focus at one time and, since the interactions between data sources and sinks are via method calls or sent (as opposed to posted) events, all processing takes place within the lifetime of a single user interface event's processing.

Location of Processing

Flexibility in implementing this pattern includes the location of any transformation between source and sink data types. For example, the conversion between bank identification number and bank name could be performed either by the card number field or the bank name field. Furthermore, if it is the latter, there is a choice between the card number field advertising the availability of a new card number and only advertising the bank identification number.

Conditional Advertisement

A further refinement in implementation is to allow advertisement of specific data types to be turned on or off on a per-field instance basis. E.g., in a data entry window where there are two card numbers, only one should be the source for the bank name field. If the form contains multiple card numbers and multiple bank names forming pairs, it is necessary to subclass the card number bank name data field classes to support the specific meaning of the data within the form and tag the data notification messages accordingly.

3.9 Related Patterns

Observer (a.k.a. **Publisher-Subscriber**): Use this pattern for data propagation when the data source and/or sink need to be explicitly specified. Advertisement and Observer are very similar in purpose but differ significantly when it comes to the compromises made to allow subjects and observers to communicate. Observer employs a connection-based interaction protocol which is the most efficient arrangement, but requires observer instances to know the exact identity (e.g. to have a reference to) the corresponding subjects. Advertisement makes the opposite compromise: a broadcast-based connectionless interaction protocol employing a content type hierarchy which allows data sources and sinks to exchange data without knowledge of their counterpart's exact number or location. However, the cost of this greater simplicity is reduced performance.

Sender-Pass Through-Receiver [Coad, 95]: Has three similar roles -- data sources are the senders, the form window is the pass-through and data sinks are receivers. **Advertisement** is a concrete application of this principle.

4. Conclusion

Because they were originally developed for constructing analysis models, many of the abstractions provides by current notations used to document design patterns are at too high a level for this purpose. As a consequence, important aspects of a pattern's structure and/or implementation are either omitted, or are relegated to the accompanying text. We believe the emerging field of design patterns would benefit from a notation which matches the abstraction level of the components which they contain. We have illustrated some notational features which we believe can provide a more comprehensive graphical description of patterns, relieving the pattern documentor from having to describe a lot of tedious detail in text.

The notation we used is far from perfect. Its main drawback is a lack of familiarity by practitioners both from a conceptual and a practical point of view. Overt references to implementation features are generally frowned upon when designing at higher levels of abstraction, yet are the essence of what constitutes a pattern. Our experience in adapting an existing implementation-oriented notation leads us to believe that a careful reimplementation of graphical pattern notations using existing (familiar) icons could greatly mitigate the familiarity problem. The forthcoming Unified Modeling Language [Booch et al. 96] will hopefully go along way towards alleviating this situation.

Another potential problem with the notation is that diagrams can quickly become cluttered with information. However, since most of the features shown in the diagrams are optional, it is a matter for the pattern documentor to determine the optimal balance of detail between the diagrams and the accompanying text. In fact, ION provides more support than many notations for handling diagram clutter because it provides features for showing explicitly when features exist but are missing from a diagram. The diagrams shown in this paper are more cluttered than they would normally be in practice in order to illustrate as many features of the notation as possible.

Since it was designed as a general purpose program depiction notation, ION has other features which are not directly applicable to pattern description. Examples include constructs for representing different types of classes, such as generic classes, persistent classes and active class as well as other implementation level concepts such as modules, and network processing sites. As patterns are assembled with other reusable artifacts into large reuse repositories, additional features of this nature may also be brought to bear on the description of the resources available.

Patterns are one of the most important recent developments in object technology. We feel that a notation of the kind discussed in this paper will make an important contribution to the documentation of patterns, and thus ultimately to their widespread adoption.

Acknowledgments

This work was partially supported by the RBSE project under the auspices of RICIS at the University of Houston - Clear Lake. We thank the university, RICIS and NASA for the opportunity to work on this project.

References

Atkinson C. and Izygon M., 1995,
"ION – A Notation for the Graphical Depiction of Object-Oriented Programs," *RICIS Report RB.02a.1*, Research Institute for Computing and Information Systems, University of Houston - Clear Lake.

Booch, G., 1991
Object-Oriented Analysis and Design with Applications, Benjamin Cummings, Menlo Park, CA.

Booch, G., Rumbaugh, J. and Jacobson, I., 1996
The Unified Modeling Language for Object -Oriented Development, Documentation Set version 0.9 Addendum, Rational Software Corporation, Santa Clara, CA.

Coad P. and Yourdon E., 1991,
Object-Oriented Analysis, 2nd ed. Yourdon Press, Englewood Cliffs, NJ.

Coad P., 1995.
Object Models: Strategies, Patterns, & Applications, Peter Coad, Prentice Hall,

Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F. and Jeremaes P., 1994
Object-Oriented Development: the Fusion Method, Prentice Hall.

Coplien, J. O. and Schmidt, D. C. (eds), 1995.
Pattern languages of Program Design , Addison Wesley.

Eichmann D., 1995
"The Repository Based Software Engineering Program", *Proc. Fifth Systems Reengineering Technology Workshop,* Monterey, CA, Feb. 7-9.

Gamma E., Helm R., Johnson R., and Vlissides J., 1995
Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

Krasner G.E. and Pope S.T., 1988
"A Cookbook for using the model view controller user interface paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, **1**(3).

Pree W., 1995
Design Patterns for Object-Oriented Software Development, Addison-Wesley.

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W., 1991
Object-Oriented Modeling and Design, Prentice Hall.