Patterns for Finding Objects within Designs

James Noble

MRI, School of MPCE,

Macquarie University, Sydney.

kjx@mri.mq.edu.au

August 1, 1997

Abstract

To design a program, first find your objects. Unfortunately, the right objects are not easy to find, and as a result most programs are not as well designed as they could be. The patterns in this language describe how objects can found within existing programs. By using these patterns, programs and designs can be made more simple, more general, and more easy to change.

Introduction

Finding objects is probably the most important part of object oriented design. The patterns in this paper describe one way to find objects — by examining the design of the program. These patterns address two questions: how can you find objects within a program? and, once you have found some objects, what should you do with them? These patterns do not aim to describe novel research results, rather, they attempt to capture and present well known design techniques, illustrate when particular techniques are applicable, and outline their relative strengths and weaknesses.

These patterns in this paper are intended to be part of a larger pattern language, provisionally called *Found Objects* [11]. The language is called Found Objects rather than *Introduced* or *Implementation Objects* because it is written from the perspective that the found objects are latent in the program. Objects can often be found within a program because they were overlooked in the earlier stages of the program's analysis and design. At first, precisely because they were missed, the newfound objects seem out of place in the program, but as the program evolves, the found objects become better integrated into the design, and some can be recognised as modelling concepts from the application domain. Finding

Pattern	Problem	Solution
Found Object	How can an object play its full part in a design?	Revisit your analysis to see if you have any important Found Objects.
Independent Abstraction	What should an object represent?	An object should represent one Independent Abstraction.
Single Place	How should you model a domain abstraction?	Model an abstraction in a Single Place.
Remote Place	How can you access an abstraction modelled in a Single Place?	Access the abstraction via a Remote Place.
Rich Protocol	How can you use an object in a variety of contexts?	Make the object's protocols sufficiently rich that they implement the operations required in expected contexts.

Figure 1: Summary of the Patterns

these objects is much more a process of discovery than a process of introducing extra objects for the convenience of the implementation.

Figure 1 gives an overview of all the patterns in this paper. **Found Object** is a very abstract pattern, and the following patterns (**Independent Abstraction**, **Single Place**, **Remote Place**, and **Rich Protocol**) describe aspects of this pattern in more detail. Figure 2 illustrates three relationships between patterns in the language. One pattern can *use* another pattern in its solution, or a pattern can *refine* another pattern, meaning one pattern is a more specific version of the other. Alternatively, patterns can *conflict*, meaning that the patterns are mutually exclusive, each providing a different solution to a similar problem [22].

The patterns are written in a form that begins with the name of the pattern, then a question (in italics) describing a problem, followed by a description of the pattern's context and forces. A boldface "**Therefore:**" introduces a summary of the solution (also italicised) followed by the consequences of using the pattern, an example of its use, and known uses and related patterns.

1 Found Object

Implementation objects may be more important than you think. How can an object play its full part in a design?

An object in a program should model an abstraction from the application domain. For this reason, many object oriented analysis and design methods begin by identifying a set of candidate objects from the domain, and then refine the set of candidate objects [20, 3]. Unfortunately, it is not always clear what

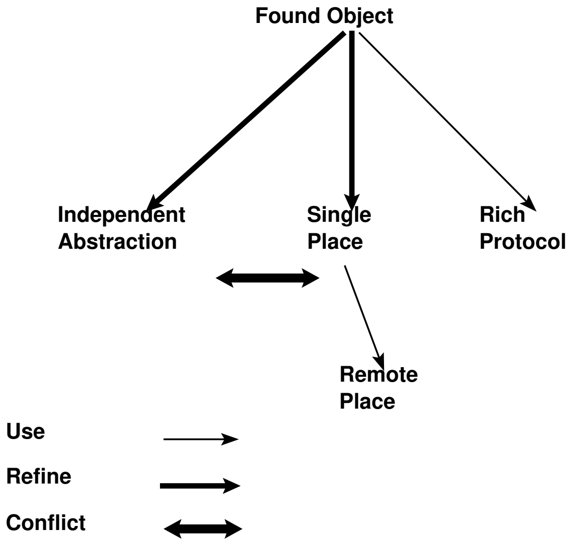

Figure 2: The Structure of the Language

objects or abstractions there are in the domain, or how to determine whether a particular program object models a domain object. This is compounded because the application domain is not ultimately fixed and objective — application abstractions may be as arbitrary as objects contained within a program.

Therefore: *Revisit your analysis to see if you have any important* **Found Objects**.

Object oriented development should use a spiral [2] or fractal [7] lifecycle. That is, the program's design should be refined during development, so that objects can be introducing during any phase of development. In particular, objects can be found during coding, as well as during analysis and design. Some of these objects will be nothing more than convenient mechanisms for implementation, but many will be important to the overall design of the program, and even to the model of the application domain.

Objects can be found from various parts of a program's design. The most obvious is that some objects in a program may be very large and complex — these objects can often be replaced by several smaller objects. Objects may similarly be found by considering the protocols by which objects communicate, or the computations performed by the program. Objects may also be found by examining a program's methods, relationships, and types.

Example

Early versions of a program visualisation system were based on a two-part application domain model based on objects to be displayed, and visualisations which monitored and displayed those objects [15]. In the original design, Target objects were directly connected to their View objects. After experience with this design, Watcher objects were introduced between Target objects and View objects, so that the design of graphical displays could be separated from the strategy for monitoring the Target objects (in the Watcher objects). These Watcher objects proved very useful, so the application model of visualisation was revised to include a third component between the Target objects and the View objects, which represented the monitoring strategy. Discovering the **Found Objects** in the program changed the model of the application domain.

Consequences

Found Object and the other patterns in this language tend to generate designs with many small, simple objects[1]. All these patterns simplify a design locally (for example, by changing one particular protocol) but impose global changes to the program (because extra objects are needed to implement the changed protocol). As a result, although objects may be easier to understand in isolation, the global design of the program may become confused. To quote Alan Perlis: *In the long run every program becomes rococo — then rubble.* [17]. If these patterns are applied injudiciously they will accelerate this process.

Because the patterns change designs and introduce extra objects, applying them can cause some rework if they are used in the midst of an existing development. This is particularly problematic for large projects with multiple developers. Generally, for larger systems, more objects will be found and more rework will be required.

In this situation, just as with a single developer, the key test for a **Found Object** is whether it really represents an important abstraction from the application domain or from the program's architecture. If a **Found Object** is not incorporated into the program's design, then any subsequent change or extension to the abstraction will require changes throughout the program, and the costs of these changes will be distributed throughout the project team. If the cost to incorporate the **Found Object** is paid once, then subsequent changes involving the abstraction, and the costs of those changes, can be localised to the **Found Object** and its clients. Although it is important to incorporate **Found Objects** quickly to

[1] This is not limited solely to this pattern language. Many other patterns have this effect, including those in *Design Patterns* [8] and *Smalltalk Best Practice Patterns* [1], as do Parnas's criteria for program decomposition [16].

minimise rework, the refactoring needs to be managed. For example, a project schedule could regularly suspend normal development activity so that **Found Objects** can be discovered and incorporated, and the program retested.

More mundanely, increasing the number of objects in a design can increase the amount of memory required by the program and decrease the program's execution speed.

Known Uses

This pattern is ultimately about the way programmers understand their designs. For this reason, **Found Objects** are difficult to identify directly, although they can be identified from historical records showing the evolution of the design over time. In a small way, every object is a **Found Object**.

Related Patterns

A **Found Object** should represent an **Independent Abstraction (2)** stored in a **Single Place (3)** with a **Rich Protocol (5)**. A **Arguments Object**, **Result Object** [13] or **Relationship Object** [14] can represent an important pattern of communication between objects. Many patterns in other pattern languages depend on finding objects, often called *reification* or *objectification* [22, 21].

2 Independent Abstraction

What should an object represent?

You have found some objects, either by using traditional analysis techniques, or by applying the **Found Object (1)** pattern to mine objects from your program. Very generally, the more objects you have in your design, the better — the more flexible your design will be, and the easier it will be to adapt the design to different situations. The finer the grain of the objects in your program, the more likely some objects will be able to be reused in the future. This is, of course, provided that you have the "right" objects — simply adding extraneous objects or arbitrarily subdividing existing objects provides no benefit. How can you determine if your objects are the right ones to represent your application domain?

Therefore: *An object should represent one* **Independent Abstraction***.*

Objects should model abstractions which make sense in the application domain and fit within the program's architectural design. Consider each object in the design, and compare it with your best estimate of the important abstractions from the application domain. Ensure that each program object does not model more than one domain abstraction, subdividing objects if necessary so that each independent abstraction is modelled by an independent object.

Example

Consider the Watcher objects in the program visualisation system described above. In the original design, View objects represented two abstractions from the program's domain — the visualisation display, and the display's connection to the target objects in the program being visualised. In the redesign, View objects represent one of these **Independent Abstractions**, and Watcher objects represent the other.

Consequences

In this context, an object or abstraction is *independent* if it can be defined and understood separately from other objects or abstractions. Abstractions (and objects) are of course also *interdependent* — one independent abstraction can use or depend upon other abstractions in a variety of ways. Each important abstraction must be considered at the appropriate level of abstraction. For example, a whole table conceptually consists of a table top and the table legs, and the table top physically relies on the legs for support. The whole table, the table top, and the table legs are each an independent abstraction in an application domain about building tables, and so each should be represented independently within the design. If the conceptual whole-part relationship or the physical support relationship are important, they should also be modelled as independent abstractions.

The idea that **Found Objects** can represent **Independent Abstractions** is one of the most important contributions of object orientation, and provides many advantages, in particular in making programs (and the objects they contain) more flexible, extensible, and reusable. Modelling **Independent Abstractions** increases cohesion within objects, but can also increase coupling, since the independent objects must cooperate to model the whole domain. Unfortunately, a design with many independent objects, although very flexible, will be difficult to learn and may have large space and time overheads.

Known Uses

As with **Found Object (1)**, it is difficult to identify uses of **Independent Abstraction** after the fact, because the pattern is very general. Every well designed object embodies this pattern — Smalltalk's Collection, Date, and Point classes are three small examples.

Related Patterns

An object should represent the **Whole Value** [5] of the **Independent Abstraction** it represents. It should be the **Single Place (3)** where that abstraction is modelled in the program, and should be available to the rest of the program via a **Rich Protocol (5)**.

3 Single Place

How should you model a domain abstraction?

You have a collection of **Found Objects (1)** which model **Independent Abstractions (2)** in the program's domain. These abstractions and objects are interdependent — some abstractions will depend on other abstractions.

Information should not be duplicated among several objects in the design. Any individual data item should be stored in only one object, to minimise memory consumption and avoid update anomalies. Similarly, each operation should be implemented in only once place, to avoid inconsistencies when the design is changed.

Therefore: *Model an abstraction in a* **Single Place***.*

Consider the application domain abstractions modelled by each object in the program. Ensure that each domain abstraction is modelled in a single place — ideally, by a single object. If an abstraction is modelled in more than one place, keep it in the place that best models that abstraction, and eliminate the other places from the design.

Example

The introduction of Watcher objects in the program visualisation system is also an example of the **Single Place** pattern. In the earlier design, a variety of different View objects re-implemented the same monitoring strategies, and several apparently independent views differed only in the strategy they implemented. Each monitoring strategy was implemented in a number of different places — inside each View which needed that strategy. Modelling each strategy with an individual Watcher object ensured that each strategy was modelled in a **Single Place**.

Consequences

Independent Abstraction (2) and **Single Place** are complementary patterns. **Independent Abstraction** says an object (or method, variable, etc.) should represent a single abstraction, while **Single**

Place says an abstraction should be represented by a single object (or other program construct). Carried to extremes, **Independent Abstraction** generates a design with a huge number of very small independent objects, while **Single Place** generates one very large single object. In practice, these patterns are applied together, each pattern working to contain the excesses of the other, resulting in found objects which balance the underlying tension between independence and isolation.

Unlike most other patterns in this language, this pattern can reduce the number of objects in a design by eliminating duplicates, depending upon the context where it is applied. This pattern can also increase the size of a design, however, where similar parts of several objects are moved into a new object.

Known Uses

The HotDraw framework was redesigned so that all user input would be handled in a **Single Place** [4]. Many programming languages provide constant definitions, so that important literal values can be recorded in a **Single Place**, rather than being repeated throughout the program. Relational database normalisation uses functional dependencies between data items to divide the data model into separate parts, and each part is stored in a **Single Place** — a single relational table [6].

Related Patterns

A **Remote Place (4)** gives access to the **Single Place** from other places. A **Rich Protocol (5)**, **Protocol Object** [13] or **Relationship Object** [14] can ensure important abstractions are defined in a **Single Place**. Many other patterns also embody **Single Place**, for example, **Write a Loop Once** [9] describes how very common loops can be written in a **Single Place**.

4 Remote Place

How can you access an abstraction modelled in a **Single Place***?*

You have a collection of **Independent Abstractions (2)** each modelled in a **Single Place (3)**. Unfortunately, domain abstractions (and the objects that implement them) are interdependent, as well as independent — that is, they depend upon other abstractions or objects. Client abstractions need to be able to access the server abstractions on which they depend.

Therefore: *Access the abstraction via a* **Remote Place**.

A local variable is an appropriate **Remote Place** for an object that is needed by a few client objects.

A global variable is an appropriate **Remote Place** for an object used throughout the program. A variable pool or local namespace can organise a set of **Remote Places** needed by some fraction of objects in the program.

Example

In the program visualisation system, the View and Watcher objects are **Independent Abstractions (2)**, each modelled in different **Single Places (3)**. Views need to communicate with Watchers to arrange monitoring of their Target objects, and Watchers need to communicate the results of the monitoring to the Views. Because one View is associated with each Watcher, and vice versa, View and Watcher objects each simply have a variable referring to their associated object. These variables are **Remote Places** which refer to the other objects.

Consequences

This pattern increases interobject communication, and generally interobject coupling. This coupling is necessary, however, because it models a dependence in the application domain. For this reason, the **Remote Place** pattern is very commonly used together with **Single Place (3)**.

Known Uses

The proxy objects used for distribution in operating systems (like NextStep [10]) and databases (like Gemstone [18]) are **Remote Places** which provide access to the **Single Place** in other address spaces where objects are really stored [8]. Just as relational database normalisation ensures each data item is stored in a **Single Place (3)**, a database foreign key is a **Remote Place** by which one table can refer to data in another table.

Related Patterns

A **Facade** [8] is a **Remote Place** that provides access to a subsystem of objects. A **Factory Method** [8] is a **Remote Place** which creates new objects. *Some patterns for relationships* describe how the association between an object and a **Remote Place** can be maintained [12, 14].

5 Rich Protocol

How can you use an object in a variety of contexts?

Objects in the real world can be used in a variety of contexts. For example, a plastic desk ruler can be used to draw straight lines, measure short distances, and exterminate small insects. Similarly, objects in a program act as servers for a variety of client objects. A client object uses a server object via the server's protocol. Clients will have their own responsibilities, and will make different demands on the server. How can you best design protocols so that objects can act as servers to a variety of clients?

One way to do this is to make protocols small and simple, so an object embodies only the very core of the **Independent Abstraction (2)** it represents. Small objects are easy to write, easy to learn, and can be implemented quite efficiently. Unfortunately, small server objects tend to create large client objects. Consider an array with a minimal interface, including only element accessor and assignment messages. Clients of such an array would repeatedly re-implement a variety of other operations, such as searching or sorting the array, or iterating through all the array elements.

Therefore: *Make the object's protocols sufficiently rich that they implement the operations required in expected contexts.*

Just as objects in the real world can be used in a variety of contexts, program objects modelling the real world should be usable in a similar variety of contexts. Inspect each context in the design where an object is used as a server, and ensure that the object's protocol supports all the operations required in that context. In particular, if a client performs a composite operation on a server (if it sends several separate messages to the server, and then further processes the results) extend the server's protocol to support the composite operation. Examine each protocol in the program, and add messages to fill any obvious gaps. If two protocols are very similar, consider merging them to produce one richer protocol.

Example

Consider designing the protocol for a `List` object from scratch. Assume an inspection of the contexts in a program's design where the `List` will be used has identified three operations — items are added to either end of the list, but are only removed from the front. A minimal `List` protocol for this context would consists of three messages — `addFirst`, `addLast`, and `removeFirst`.

A more through inspection may reveal that list items are often "cycled" through the list — that is, they are removed from the front only to be immediately added to the end. With the minimal interface, this cycle operation must be programmed explicitly by each client object. Adding a `cycle` message to the List protocol means that the cycle operation will be implemented in a **Single Place (3)** — the List object — so every client does not need to re-implement it.

On further examination of the protocol, a removeLast message could be added to fill the gap in the symmetry of the add and remove messages. A size message could also be added, because this is useful in many contexts.

Consequences

A rich protocol reduces the coupling between a server and its clients, because the protocol is more likely to include the operations the clients need. **Rich Protocols** are also easier to use, because clients can invoke operations directly, rather than having to piece them together using whatever messages the server actually provides. Of course, richer protocols are more work to implement that more minimal protocols, harder to learn, and may increase the program's runtime space requirements.

The patterns **Single Place (3)** and **Independent Abstraction (2)** constrain the application of this pattern. **Single Place** gives impetus to this pattern, making objects' interfaces richer, so any given behaviour is only defined in a single place. **Independent Abstraction** restrains this pattern, so objects and protocols keep their conceptual integrity.

Known Uses

Smalltalk provides many examples of **Rich Protocols**. The simplest of these is the printing protocol defined in class Object, which ensures that every Smalltalk object can print itself. The Smalltalk collection classes have **Rich Protocols**, providing behaviour to access, enumerate, and update their elements. The Tools and Materials metaphor uses aspect classes so that materials provide **Rich Protocols** to tools [19].

Related Patterns

A **Rich Protocol** may often be a large or complex protocol. The **Arguments and Results** patterns [13], amongst others, can be used to simplify rich protocols.

Acknowledgements

These patterns are revised versions of some patterns from the *Found Objects* pattern language [11], and were inspired by discussions on the *patterns-discussion* mailing list and the *WikiWikiWeb*.

The authors and reviewers at the EuroPLOP'96 writers workshop provided many pertinent comments and suggestions. Lorraine Body described the implications of these patterns for software development teams, and Linda Rising and Charles Weir provided detailed comments on the conference draft, as did David Holmes back at MRI. Ward Cunningham, this paper's EuroPLOP "shepherd", provided many important suggestions regarding both the paper's form and content, and in particular proposed the subdivision of **Found Object**.

References

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

[2] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.

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

[4] John Brant and Ralph E. Johnson. Creating tools in hotdraw by composition. In *TOOLS Europe 13*, 1994.

[5] Ward Cunningham. The CHECKS pattern language of information integrity. In *Pattern Languages of Program Design.* Addison-Wesley, 1994.

[6] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, 1986.

[7] Brian Foote. A fractal model of the life cycle of reusable objects. In Jim Coplien, Russel Winder, and S. Hurtz, editors, *OOPSLA'93 Workshops on Process Standards and Iteration*, 1993.

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

[9] Robert Martin. Discovering patterns in existing applications. In *Pattern Languages of Program Design.* Addison-Wesley, 1994.

[10] NeXT Computer, Inc. *NeXTSTEP Object-Oriented Programming and the Objective C Language*, 1993.

[11] James Noble. Found objects, 1996. Reviewed at EuroPLOP.

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

[13] James Noble. Arguments and results. Submitted to PLOP'97, 1997.

[14] James Noble. Basic relationship patterns. In *EuroPLOP'97 Proceedings*, 1997.

[15] R. James Noble, Lindsay J. Groves, and Robert L. Biddle. Object oriented program visualisation in Tarraingím. *Australian Computer Journal*, 27(4), November 1995.

[16] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3), March 1985.

[17] Alan Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9), September 1982.

[18] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an object server with other worlds. *ACM Transactions on Information Systems*, 5(1), January 1987.

[19] Dirk Riehle and Heinz Küllighoven. A pattern language for tool construction and integration based on the tools and materials metaphor. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.

[20] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[21] Kirk Wolf and Chamond Lui. New clients with old servers: A pattern language for client/server frameworks. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.

[22] Walter Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.