The Object System Pattern

James Noble
Microsoft Research Institute,
Macquarie University,
Sydney, Australia
kjx@mri.mq.edu.au

June 4, 1998

Abstract

Classical object oriented systems require every object to be an instance of a class, and all classes to be written before the program is deployed. Some programs need flexible, configurable, dynamically extensible representations of objects, which cannot be determined in advance. The Object System pattern describes how you can build your own object system from scratch. Using the Object System pattern, you can build precisely the kind of objects you need for your application, but these objects will require more memory space and execution time than the more rigid objects supported by programming languages.

Intent

Some programs need flexible, configurable, dynamically extensible representations of objects. The Object System pattern lets you build precisely the kind of objects you need for your application.

Also Known As

Frame System, Prototype-based system, Prototype-instance system.

Motivation

Some systems need to represent flexible, dynamic objects. This may be to support end-user tailorable or programmable systems, rapid prototyping or evolutionary development, or modelling domains where the structure and content of information in the domain is constantly changing.

Consider a system to support the initial development and analysis of civil engineering designs. This system needs to allow users (civil engineers with some computing experience) to describe their designs, and then use the information about the designs to compute properties of the designs. The system needs to allow engineers to edit all aspects of their designs — adding objects to the design (such as beams, walls, floors, and so on), deleting them, changing objects' properties (such as the cross-section, length, or composition of a beam), adding new properties to objects (such as giving one particular beam a new colour property with the value grey), and changing objects' classes (such as replacing a concrete-beam with a reinforced-steel-girder). Engineers also make need to make new kinds of objects in the middle of producing their designs, for example, creating a particular kind of beam and then using it repeatedly throughout the design. So that designs can be analysed, engineers need to be able to add procedural code to the objects in their design, and run the code to analyse the design, or export the design to existing analysis software. For example, a beam should be able to calculate its volume based on its length, width, thickness, and cross-section.

There are a number of possible approaches to produce this kind of system. The most traditional approach is to write a program (typically in FORTRAN or C) to model the design. While this program

will undoubtably execute efficiently, and can handle complex analysis calculations, such programs require skilled programmers to construct, and once built are typically quite difficult to modify. More modern programmers will choose an object-oriented language, but most object oriented languages do not support arbitrary dynamic modification and extension of the objects in the program.

Alternatively, the design could be modelled using a tool designed for end-user programming, such as a spreadsheet or scripting language. Both spreadsheets and scripting languages provide a large amount of flexibility, and they are designed to support programming on-the-fly. Unfortunately, these types of tools do not provide much support for modelling. Spreadsheets can be (and, all too frequently, are) used to model anything, but they do not provide much support for modelling data other than numbers. Most scripting languages provide simple textual lists and associative arrays, but these are not really first class objects in the language, as they do not support linked data structures.

The Object System pattern describes how these requirements can best be met. Rather than fighting to add flexibility to an existing programming language or to impose structure onto a spreadsheet or scripting language, you can construct your own implementation of an object system. By implementing the objects yourself, you can ensure that new objects and new kinds of objects can be created at runtime, and that the objects can be modified and extended directly. An object system can be extended to provide extra system-wide features like persistence or constraints, to interface with external systems, and to provide good error checking and debugging support to the programmer.

Consider once again the problem of modelling engineering designs — see Figure 1. At a high level of abstraction, everything an engineer needs to manipulate when constructing a design — beams, walls, floors, and even intangible considerations like traffic flow, loaded weight, and ambient lighting — can be seen as a single kind of object, a *Design Element*. Each design element has a number of properties, such as its weight, height, composition, or the other design elements which are supporting it. Because each design element is treated individually, its precise properties can depend upon what it is modelling and how it is used. Using the object system, engineers can describe a design by configuring design elements, linking them to one another, and adding, removing, and adjusting their properties. Engineers can analyse a design by attaching procedural code — methods — to design elements and then executing the methods to analyse the design.

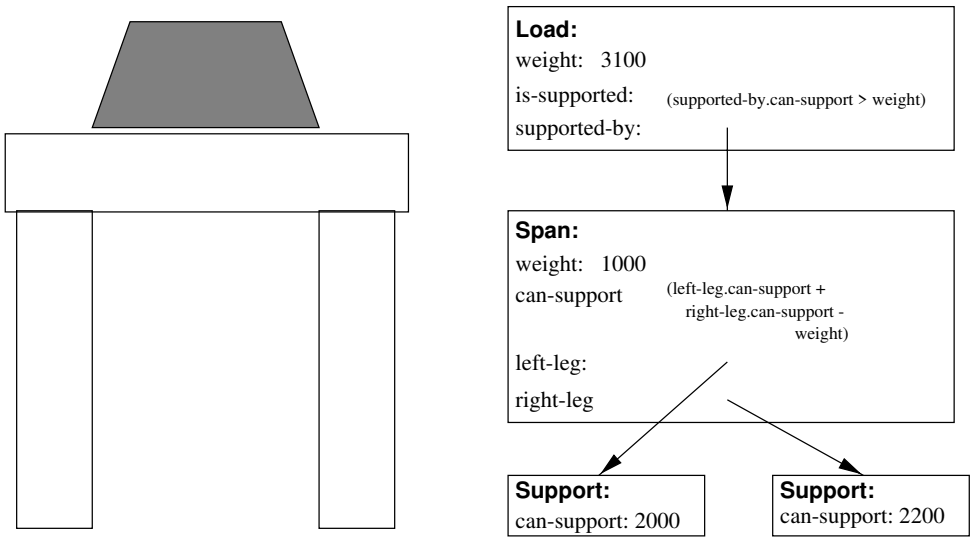


Figure 1: A Design for a Bridge

Figure 1 shows a very simple design for a bridge. A main Span design element is supported by two Support design elements. The Span itself has to support a given load, also modelled as a design element. The designer has described each design element by attaching properties to them — describing their width, length, composition, and so on — note that every design element has a different set of properties. The designer has attached methods to the Span and Load elements to calculate how much

weight the load can-support and whether or not the load is-supported.

Figure 1 shows how this is implemented. Each design element and each element property is modelled as an individual instance of the implementation-level classes DesignElement and ElementProperty respectively. A DesignElement stores a list of its ElementProperties, and provides protocol for adding, removing, and modifying properties.

Figure 2 shows the class structure. The system has two kinds of DesignElement objects, which share a superclass to define a common interface. Most design element are BasicDesignElements which can be manipulated by the user, but the system also has PrimitiveDesignElements so that properties can store primitive values like integers and weights. Similarly, there are two kinds of ElementProperties in the system, SimpleProperties (like a Slab's weight or construction properties) which store a reference to another DesignElement, and MethodProperties which store methods like the Slab's can-support property and the Load's is-supported property. Note that both DesignElements and ElementProperties have a clone method which produces a copy of the respective element or slot (a DesignElement copies all it's slots when it is cloned), so that any DesignElement can act as a prototype to create other design elements.

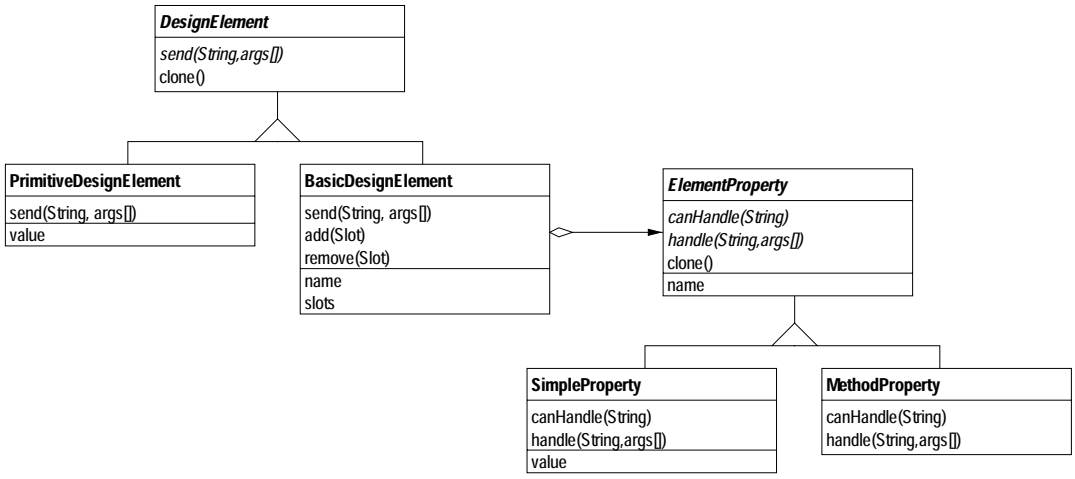


Figure 2: Design Elements

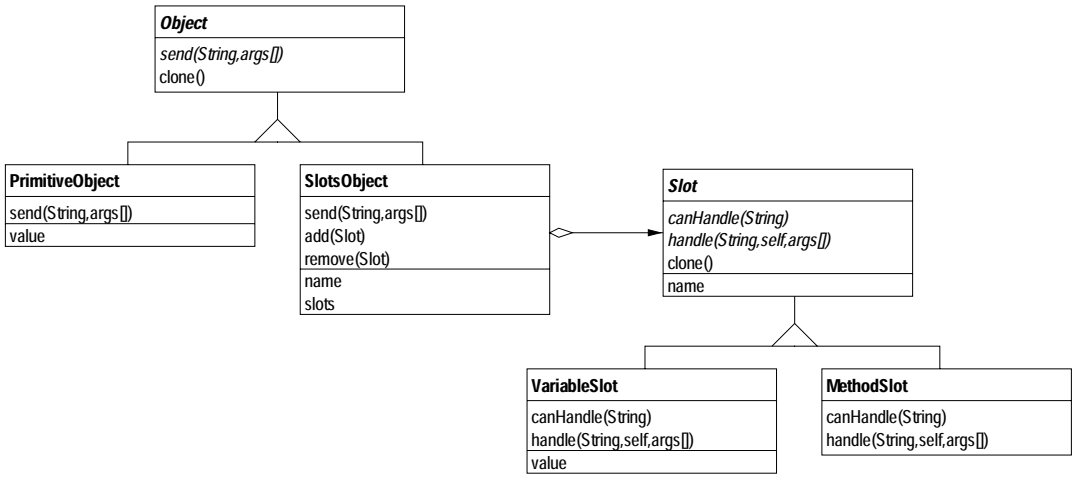
Object systems are useful in any system which needs to represent or analyse designs flexibly. As well as straight design domains, object systems are also useful in CASE tools and user interface design systems. This flexibility comes at a cost, because object systems require more memory, execute more slowly, and provide less static safety than the objects provided by a programming language.

Applicability

Use an Object System when:

- You need objects which can be flexibly created and modified at runtime.
- You can afford to pay significant time and space costs to buy this flexibility.
- You wish to avoid building a hierarchy of very similar classes.
- You don't need the security of a static type system.
- You need to incorporate system-wide extensions to objects' behaviour, such as persistence, change detection, or constraints.
- You've tried (or considered) using simpler patterns, such as **Prototype** [9], **Type Object** [12], **Property List** [25, 21], **Anything** [25], **Extension Object** [8], **Interpreter** [9], and **Reflection** [5] and they are insufficient.

Structure

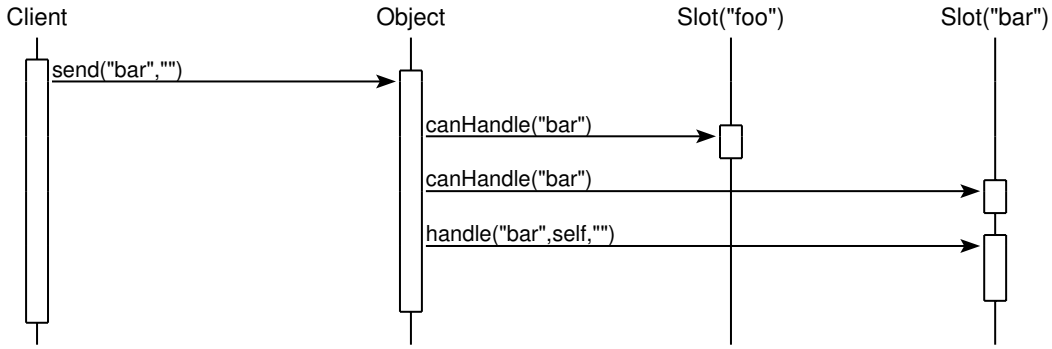


Participants

- **Object** (DesignElement)
  - Provides an interface for all objects in the system
  - Allows objects to handle messages.
- **SlotsObject** (BasicDesignElement)
  - Represents the user's objects.
  - Delegates messages to its slots.
  - Allows the user to add, remove, and modify the slots comprising the object.
- **PrimitiveObject** (PrimitiveDesignElement)
  - Wraps a primitive object from outside the object system, such as an object provided by the implementation language, a library, or an external resource such as a window system.
  - Delegates messages as appropriate
  - Allows external objects to be used seamlessly from within the object system.
- **Slot** (ElementProperty)
  - Describes a property of an Object.
  - Responds to messages as appropriate.
- **VariableSlot** (SimpleProperty)
  - Records a single name variable local to an object.
  - Assigns and returns this value in response to messages.
- **MethodSlot** (MethodProperty)
  - Holds procedural code for a method attached to an Object.
  - Executes the method in response to messages.

Collaborations

- A client sends a message to an object. The object searches for a slot which implements the message, then delegates the message to the slot.



Consequences

The Object System pattern has the following **benefits**:

- *Dynamic Configuration* The objects in an object system can be constructed and modified at runtime, whatever the underlying implementation language. Empty objects can be created from scratch and then filled with slots, or existing objects can be treated as prototypes, creating new objects by cloning.
- *Flexibility* An object system can provide as much flexibility as is needed by the application. Any kind of object can be built simply by adding properties into an object, without reference to preexisting objects or rigid classes. This flexibility is both a benefit and a liability — it is a benefit when the flexibility is required, and when it is used in a disciplined way, but if it is used carelessly it can result in a large number of badly-designed objects — a maze of twisty little objects, all slightly different.
- *Integration* Because of its flexibility, an object system can sit at the centre of a system's architecture, and integrate diverse external components. The components can either be linked directly into the object system (as PrimitiveObjects), or the object system can provide facilities to read and write its objects so that they can be understood indirectly by completely separate components.
- *System-wide Extensions* Because you've implemented the whole object system yourself, it is easy to add features such as persistence, change detection, or constraints, to every object in the system.
- *User Support* It is easy to add features to the system purely to support the user. For example, every object can be given a human-readable name to support debugging. Type errors can be checked dynamically, and error messages produced at runtime. Every individual object or object property can record the date it was last edited and the programmer who made the change.
- *Portability* With care, your object system (or at least the objects supported by the system) can be portable across different underlying languages, operating systems and architectures. If you have made little use of platform-specific primitive objects, you can write out the objects from one version of the system, and read them into another.

The Object System pattern also has the following **liabilities**:

- *Efficiency* An object system will execute much more slowly (up to a thousand times slower), and will require more space (perhaps a hundred times as much space) as objects provided directly within a programming language. The degree of degradation depends greatly upon the sophistication of the implementation techniques used.

- *Static Checking* Although an object system can do a lot of dynamic checking and provide a lot of information to the programmer at run time, it cannot generally provide much compile time support. In particular, while object systems may be dynamically type safe, so the program will never crash if an object receives a message it does not understand, type checks probably cannot be performed statically at compile time.

- *Method Syntax* An object system can require convoluted syntax for the methods attached to objects. If the methods are embedded within another programming language (typically the language implementing the object system), sending a message to an object system object can require many extra macro or function calls, compared with sending a message to an implementation language object. If the methods are implemented using a parser and interpreter, programmers have to learn the syntax of a whole new language.

- *Complexity* Introducing an additional object system makes the overall programming model more complex. Programmers must decide what should be supported directly by the infrastructure of the object system, what should be programmed using the object system, and what should be left to the underlying implementation language.

- *Standardisation* Your object system won't be a standard, unless you are Microsoft or a strategic group in a large company. This may make it more difficult to find users for your object system, since they may be unwilling to expend effort in developing for a narrow base. This may also make it difficult to reuse code developed for other systems, unless the object system can be interfaced to it directly. Although you can extend your object system to provide features like persistence, it will be difficult to buy or reuse existing products to provide these features.

- *Maintenance* You must maintain and support the object system as well as the applications which are built within it, because all the applications will depend upon the continued operation of the object system. This means you will need programmers who understand the implementation of the object system, as distinct from how to use the object system to support actual applications.

- *Cognitive Dissonance* You will build up a strong emotional attachment to your object system, and your organisation may make large investments in developing and maintaining it. This can make it difficult for you and your organisation to move to more modern, standard, third party products, rather than continuing with your own object system.

- *Overuse* Object systems are often overused. Because object systems are flexible, they can be used to solve almost any problem, but because they have many liabilities, they should be used only where other less expensive approaches (such as **Prototype** [9], **Type Object** [12], **Property List** [25, 21], **Anything** [25], **Extension Object** [8], **Interpreter** [9], or **Reflection** [5], or an appropriate object oriented programming language) are not applicable.

Implementation

Consider the following implementation issues when applying the Object system pattern.

1. *Implementing methods.* Methods can be implemented in an object system in two ways — either by using an **Interpreter** [9] or by incorporating methods directly from the programming language. Using an Interpreter is more flexible, and will make it much easier for the user to extend the systems methods at runtime. Methods implemented in the underlying programming language (perhaps using macros, functions, or blocks) will execute more quickly than interpreted methods, but will be harder to change at runtime and more difficult to write. An object system can easily incorporate both kinds of methods, by providing two MethodSlot subclasses, one for interpreted methods and one for primitive methods. This can provide the best of both worlds — interpreted methods can be written on the fly, implementation language methods can be written where speed is essential, and the choice can be made on a per-method basis without affecting the rest of the object system's framework. Some object systems do not support methods, that is, objects and

slots can be used solely to provide flexible and extensible data representation, and computation can be carried out elsewhere.

2. *Accessing Primitive Objects.* You need to provide access to objects outside the object system itself. In simple cases, these are things like integers and floats, but they can be more complex, like windows, widgets, files, and so on. A primitive object is an **Adaptor** [9] which links these programming language level objects into the object system. In most languages (including C++ and Java) you will have to implement a specific primitive object for every class of object you wish to incorporate. In reflexive languages (like Smalltalk) you can build generic adaptors which can incorporate any implementation language object into the object system [9]. Primitive objects can also be used to extend the facilities of the object system itself, or to make it more efficient. For example, many object systems provide arrays implemented as primitive objects, so that common array operations will be executed rapidly and collections of objects will not occupy too much space.

3. *Creating new kinds of Slots.* A slot is similar to a property in a **Property List** [25, 21] or a **Variable State** [2], except that the slots in an Object System can also contain methods. You can extend an object system by creating new kinds of slots. For example, you could create"typed" slots which can be assigned only restricted kinds of objects, slots which are not copied into any clone of their enclosing object, slots which can contain more than one object (like the supported-by slot in the Slab example) or slots which execute a method (a demon [20]) whenever they are assigned to.

4. *Optimising message lookup.* Looking up slots in response to message sends is a major source of inefficiency in object systems. Using a linear search is quite slow, but may be acceptable if most objects contain only a few slots, and testing whether a particular slot implements a particular message is quick. Hash tables can be used for more efficient slot lookup. Slot names can also be represented as symbols or instances of enumerated types, rather than strings, so that they can be compared or hashed quickly. Using an enumerated type to represent slot names makes adding new slot names more complex, but can also protect the programmer against misspellings [18].

5. *Using structure descriptors and virtual copying to represent clone families.* Most objects in object systems will be structurally identical to many other objects in the system. In particular, every object created by cloning a given prototype and not subsequently changed — (the prototype's *clone family*) will have the same structure. Naïve implementations of object systems require large amounts of memory to store the structure of every object in the system. To reduce memory use, you can eliminate this duplication by applying the **Type Object** [12] pattern, using structure descriptors and virtual copying. Make a read-only descriptor for each object structure, and share the descriptor between all objects with a clone family. The individual objects then need only store a pointer to the descriptor and the contents of their variable slots. Slot descriptors can provide the memory-conservation benefits of class-based systems, while preserving the flexibility and user-level simplicity of a prototype-based system.

6. *Propagating structural changes.* You may want to propagate structural changes made from an object used as a prototype to all objects cloned from it. This can keep the structures of the objects in the system more consistent. This can be implemented by keeping a record of each clone family. Whenever a family's prototype's structure is changed, that is, when slots are added to or removed from it, the same change can be made to each object in the clone family. If you are using structure descriptors, they can act as a **Manager** [24] to record the objects in the family and manage the propagation.

7. *Providing Inheritance.* Object systems can support inheritance between objects, so that objects can share structure and behaviour. For example, one way to add single inheritance between objects is to add a new kind of slot (a ParentSlot) which is a subclass of VariableSlot. When a SlotsObject receives a message, and the object does not contain a slot which implements the message, the message should be send on to the object contained in the ParentSlot. To support true inheritance (avoiding the *self problem* [14]) the message should be resent by a variant of the

basic send operation which ensures that self-references in the inherited message are bound to the object which originally received the message. Object systems can be extended to support multiple inheritance by using multiple parent slots and more sophisticated method lookup algorithms.

8. *Providing classes.* If your object system makes heavy use of clone families, inheritance, and change propagation, you might prefer to implement a class-based system, rather than a classless prototype-based system. Essentially, the **Type Object** [12] structure descriptors are turned into classes by making them visible to the object system's users. A class based system makes managing groups of objects easier than managing objects individually, and makes optimisations like using structure descriptors and optimised slot lookups much easer. Using explicit classes, however, will make an object system more complex and less flexible.

9. *Adding System-wide features.* Object systems make it easy to extend the system as a whole. For example, all the objects in the system can be made persistent simply by making the implementation of Object persistent, typically by adding methods which allow an object, its slots, and the objects reachable from it to be written into a file. Similarly, constraint systems (which need to detect when objects change) can be integrated into an object system by changing the implementation of VariableSlots so that they generate change notifications whenever they are assigned to.

Sample Code

This example is a simple object system written in Java, called Ego, after Self [28] and Kevo [26]. A Java interface called `Ego` defines the main interface to the object system. The main method in the `Ego` interface is `send`, which sends a message and its arguments to an object. To work around Java's limited polymorphism, this interface also defines some convenience functions which send messages with no arguments, one argument, or a single integer argument.

```
interface Ego {
  public Ego send(String message, Ego args[]);

  // convenience functions
  public Ego send(String message);
  public Ego send(String message, Ego arg);
  public Ego send(String message, int i);
}
```

Objects containing slots are instances of the `EgoSlotsObject` class. Every `EgoSlotsObject` has a name, a collection of `EgoSlots`, and methods to add new slots into the collection.

```
class EgoSlotsObject implements Ego, Cloneable {
  private String name;
  private Vector slots;

  public EgoSlotsObject(String name_) {name = name_; slots = new Vector();};
  public EgoSlotsObject add(EgoSlot slot) {slots.addElement(slot); return this;};
  String objectName() {return name;};
```

`EgoSlotsObject` must understand `send`, because they implement the `Ego` interface. In this very simple implementation, an `EgoSlotsObject` handles a message with a linear search of its slots to find one which can handle the message, and then delegates the message to the slot. Finally, `EgoSlotsObject` must implement the convenience functions from the `Ego` interface — these are quite straightforward, so we will omit them for brevity.

```
  public Ego send(String message, Ego args[]) {
    for (int i=0; i<slots.size(); i++)
      {
```

```
        if (slotAt(i).canHandle(message))
          {return slotAt(i).handle(message, this, args);};
      };
    throw new Error("DoesNotUnderstand: "+message);
  };

  private EgoSlot slotAt(int i) {return (EgoSlot)slots.elementAt(i);};
}
```

Ego slots are defined by the abstract class `EgoSlot`. All slots have a name, and methods for querying whether they can handle a particular message, and for actually handling a message.

```
abstract class EgoSlot implements Cloneable {
  protected String name;
  public boolean canHandle(String message) {return name.equals(message);};
  public Ego handle(String message, Ego self, Ego args[]) {return null;};
}
```

An `EgoVariableSlot` is a slot which holds a variable. A variable slot named "`name`" understands two messages — "`name`" which returns the value stored in the slot, and "`name:`" which assigns a value into the slot. As with `send`, for convenience `EgoVariableSlot` actually has a number of overloaded constructors which initialise its value with different Java types.

```
class EgoVariableSlot extends EgoSlot{
  private Ego value;

  public EgoVariableSlot(String name_, Ego value_) {
    name = name_;
    value = value_;
  };

  public boolean canHandle(String message) {
    return name.equals(message) || (name+":").equals(message);};

  public Ego handle(String message, Ego self,  Ego args[]) {
    if (name.equals(message)) {
      return value; };
    if ((name+":").equals(message)) {
      value = args[0];
      return value; };
    throw new Error("InternalErrorWrongVariableSlot: "+name+ "message:" +message);
  };
}
```

Methods are defined by `EgoMethodSlot` objects. An `EgoMethodSlot` simply calls its `body` when it handles a message. For simplicity, Ego doesn't have an interpreter, so its methods are written in its implementation language, Java.

```
class EgoMethodSlot extends EgoSlot{

  public EgoMethodSlot(String name_) {name = name_;};

  public Ego handle(String message, Ego self, Ego args[]) {
    if (name.equals(message)) {
      return body(message, self, args); };
    throw new Error("InternalErrorWrongVariableSlot: "+message);
```

```
  };


  public Ego body(String message, Ego self, Ego args[]) {
    throw new Error("Undefined Message: "+message);};
}
```

To write a useful method, you make an anonymous subclass of `EgoMethodSlot` — here's a method which returns 42.

```
  EgoMethodSlot m = new EgoMethodSlot("fortyTwo") {
    public Ego body(String message, Ego self, Ego args[])
      {return new EgoValue(42);};
    };
```

The last class, `EgoValue`, supports primitive objects by wrapping Java objects or values. `EgoValue` implements the `Ego` interface but is not an extension of the `EgoSlotsObject` class, and it implements `send` using a series of `if` statements. `EgoValue` will throw an exception if an `EgoValue` holding one type is accessed as another type — for example, if an `EgoValue` holding an integer is sent the message `booleanValue`.

```
class EgoValue implements Ego {
  private Object value;
  EgoValue(Object value_) {value = value_;};
  public Object value() {return value;};

  // convenience functions
  EgoValue(int value_) {value = new Integer(value_);};
  EgoValue(boolean value_) {value = new Boolean(value_);};
  public int intValue() {return ((Integer)value).intValue();};
  public boolean booleanValue() {return ((Boolean)value).booleanValue();};

  public Ego send(String message, Ego args[]) {
    if (message.equals("+")) {
      return new EgoValue(new Integer(
        (((Integer)value).intValue()) + (((EgoValue)args[0]).intValue()))); };
    if (message.equals("-")) {
      return new EgoValue(new Integer(
        (((Integer)value).intValue()) - (((EgoValue)args[0]).intValue()))); };
  // and so on
}
```

Finally, here's how the `Load` object from the Motivation section can be defined in Ego.

```
    EgoSlotsObject load = new EgoSlotsObject("Load")
      .add(new EgoVariableSlot("supported-by",span))
      .add(new EgoVariableSlot("weight",3100))
      .add(new EgoMethodSlot("is-supported") {
        public Ego body(String message, Ego self, Ego args[]) {
          return self
            .send("supported-by").send("can-support")
            .send(">",self.send("weight"));  }; } );
```

Known Uses

Object systems are derived from the frame systems used ubiquitously for knowledge modelling in artificial intelligence, and which contributed to the development of conventional object-oriented programming

[4]. More recently, object systems have been used to model complex technical domains. The engineering example at this start of this pattern was inspired the object system for supporting Civil Engineering developed at the University of Strathclyde [27]. The Cystic Fibrosis Society uses a language based on an object system to record information about clinical surveys [11]. Intellicorp, amongst others, sells modelling environments based on dynamic object systems.

Object systems have also been used to provide powerful scripting languages for application extension. The Basic Object System (BOS) [7] and Lua [10] link object systems to applications written in C and C++, and provide good examples of techniques for seamless integration between the object system and the implementation language. Obliq [6] uses an object system to support distributed "migratory" applications. More whimsically, many adventure games and Muds are based on object systems [1, 13].

A number of object systems have been implemented as part of graphical user interface toolkits. Perhaps the best known is NewtonScript [23], which is the main programming environment for the Newton MessagePad. Garnet [19] and Amulet [18], implemented in Common Lisp and Amulet respectively, are user interface systems for more conventional machines. They use object systems to support user-defined constraints between objects. VisualObliq [3] extends Obliq to support user interface construction.

Object systems can be used to store and manage application specific data. The Any framework [17] has been used to provide application and machine independent data integration for the Beyond-Sniff development environment. Although the Any framework does not allow behaviour to be attached to directly to objects, it is programmable via the Python scripting language.

Related Patterns

The Object System pattern is the structural complement of the **Interpreter** [9] behavioural pattern and the **Prototype** [9] creational pattern. Where Object System supports flexible modelling, Interpreter supports flexible behavioural modelling, and Prototype allows objects to be constructed easily. Together, Object System, Interpreter, and Prototype can implement a whole prototype-based object oriented programming language.

The Object System pattern is related to the **Reflection** pattern [5] in the sense that the implementation of the object system itself (and any associated interpreter) are the metalevel for the object system. Whether an object system is meta-reflexive depends on whether it can compute about its own structure and behaviour [16]. In practice, this means whether the implementation level operations such as `send` and `add(slot)` are available to programs within object system itself, so that these programs can modify their own structure and behaviour. In a meta-reflexive system, for example, you can `send` an object system level message to an object system object and have it invoke the implementation level message `add(slot)`.

The Object System pattern also relies on other patterns. PrimitiveObjects are examples of the **Adaptor** [9] pattern — they adapt the underlying primitive objects to the interface expected by the Object System. Using separate structure descriptors is an example of the **Type Object** [12] pattern, because the descriptors describe a clone family of objects with the same type. The **Dynamic Template** [15] pattern describes a complex class-based Object System as an extension of Type Object. Because the descriptors are shared and are read-only, they are also **Flyweights** [9]. The **Serializer** [22] pattern can be used to make Object Systems persistent or distributed, by storing objects on disk or sending them around a distributed system. An object model must be defined as part of the implementation of the **Broker** [5] pattern, and the Object System pattern can be used to implement this object model.

The Object System pattern is one of the most complex patterns in a sequence of patterns for making programs more flexible. The simpler patterns in this sequence are smaller scale patterns, describing how to increase the flexibility of single objects, while the more complex patterns have a larger scale impact, and describe flexible architectures for whole programs. The patterns in this sequences are: **Prototype** [9] to create objects without knowing their class; **Property List** [25, 21] or **Variable State** [2] to represent a flat list of extra attributes; **Anything** [25] to represent a nested tree of objects and attributes; **Extension Object** [8] to add behaviour to objects; **Interpreter** [9] to implement that behaviour at runtime; **Object System** to implement user defined objects with user defined behaviour; and **Reflection** [5] to implement a general reflexive system.

Acknowledgements

Thanks to Peter Sommerlad, this paper's EuroPLOP'98 shepherd, for his interesting and useful comments, especially on the use and dangers of Object Systems within large organisations.

References

[1] Richard Bartle. *Artificial Intelligence and Computer Games*. Century Communications, London, 1985.

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

[3] Krishna Bharat and Marc H. Brown. Building distributed multi-user applications by direct manipulation. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pages 71–82, 1994.

[4] Daniel G. Borbrow and Terry Winograd. An overview of KRL, a Knowledge Rrepresentation Language. *Cognitive Science*, 1(1):3–46, 1977.

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

[6] Luca Cardelli. Obliq: A language with distributed scope. Technical Report Research Report 122, Digital Equipment Corporation Systems Research Center, 1994.

[7] Allen Dutoit, Sean Levy, Douglas Cunningham, and Robert Patrick. The basic object system: Supporting a spectrum from prototypes to hardened code. In *OOPSLA Proceedings*, 1996.

[8] Erich Gamma. Extension object. In *Pattern Languages of Program Design*, vol. 3. Addison-Wesley, 1997.

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

[10] Roberto Ierusalimschy, Luiz Henrique de Figueriedo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software—Practice and Experience*, 26(6), 1996.

[11] Michael Johnson. PreQL: A prototype based object oriented epidemiology study description language". In *ECOOP'96 Workshop on Prototype-based Programming*, 1996.

[12] Ralph E. Johnson and Bobby Woolf. Type object. In *Pattern Languages of Program Design*, vol. 3. Addison-Wesley, 1997.

[13] P. David Lebling, Marc S. Blank, and Timothy A. Anderson. Zork: A computerized fantasy simulation game. *IEEE Computer*, 12(4):51–59, April 1979.

[14] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA Proceedings*, November 1986.

[15] Fernando D. Lyardet. The dynamic template pattern. In *PLOP Proceedings*, 1997.

[16] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA Proceedings*, 1987.

[17] Kai-Uwe Mätzel and Walter Bischofberger. The Any framework. In *COOTS Proceedings*, 1996.

[18] Rich McDaniel and Brad A. Myers. Amulet's dynamic and flexible prototype-instance object and constraint system in C++. Technical Report CMU-HCII-95-104, Human Computer Interaction Institute, Carnegie Mellon University, 1995.

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

[20] Eric Raymond and Guy L. Steele. *The New Hacker's Dictionary*. MIT Press, second edition, 1993.

[21] Dirk Riehle. A role based design pattern catalog of atomic and composite patterns structured by pattern purpose. Technical Report 97-1-1, UbiLabs, 1997.

[22] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. Serializer. In *Pattern Languages of Program Design*, vol. 3. Addison-Wesley, 1997.

[23] Walter R. Smith. Using a prototype-based language for user interface: The Newton project's experience. In *OOPSLA Proceedings*, 1995.

[24] Peter Sommerlad. Manager. In *Pattern Languages of Program Design*, vol. 3. Addison-Wesley, 1997.

[25] Peter Sommerlad and Marcel Rüedi. Do-it-yourself reflection. In *EuroPLOP Proceedings*, 1998.

[26] Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-oriented Programming*. PhD thesis, University of Jyväskylä, 1993.

[27] Iain Tulloch and Douglas McGregor. Using prototypes to model engineering designs. In *ECOOP'96 Workshop on Prototype-based Programming*, 1996.

[28] David Ungar and Randall B. Smith. SELF: the Power of Simplicity. *Lisp And Symbolic Computation*, 4(3), June 1991.