The Comparand Pattern

Pascal Costanza
University of Bonn, Römerstraße 164
D-53117 Bonn, Germany
costanza@cs.uni-bonn.de

Arno Haase
Arno Haase Consulting, Noeggerathstraße 1
D-53111 Bonn, Germany
Arno.Haase@Haase-Consulting.com
Arno.Haase@acm.org

Thumbnail

The COMPARAND pattern provides a means to interpret different objects as being the same for certain contexts. It does so by introducing an instance variable in each class of interest – the comparand – and using it for comparison. Establishing the sameness of different objects is needed when more than one reference refers to conceptually the same object. In distributed systems, the COMPARAND pattern provides for efficient comparison of (possibly) remote objects.

Example

Suppose you want to implement the Java Platform Debugger Architecture (JPDA), a specification of a debugging framework for the Java Virtual Machine (JVM).

The JPDA consists of three levels: the Java Virtual Machine Debug Interface (JVMDI), an API that is to be implemented in native code, at the level of the JVM; the Java Debug Wire Protocol (JDWP), that allows debuggers to remotely employ the capabilities offered by the JVMDI; and finally, the Java Debug Interface (JDI), a high-level Java API that abstracts from the details of the other levels and thus allows for the implementation of a concrete debugger in a pure object-oriented fashion (see fig. 1).

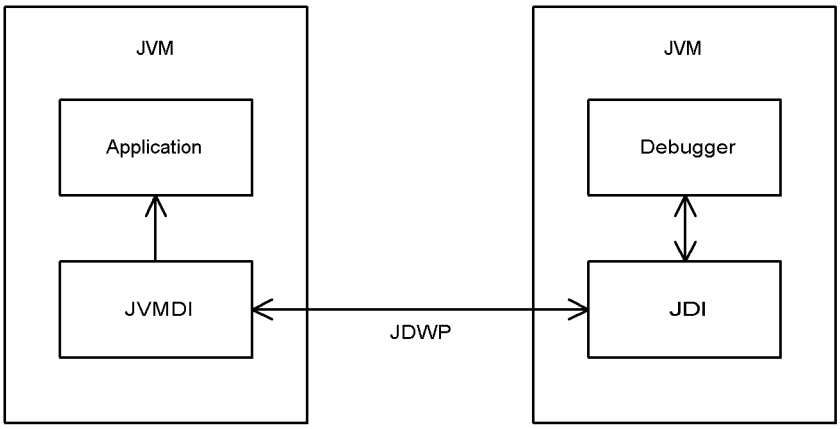

Fig. 1: The Java Platform Debugger Architecture

This architecture expects a debugger to be executed on an instance of the JVM which is different from that of the target application. Therefore the target application's objects cannot be directly referred to in the debugger by references as offered by the Java Programming Language. Instead they have to be represented as objects that act as remote references.

If a debugger needs to compare variables holding such remote references in order to determine if they refer to the same remote object, care has to be taken to do so correctly. Different remote references might refer to the same remote object, since they can be created independently, for example by consecutive retrieval operations. Therefore, if comparison of remote references yields `false`, it is not guaranteed that they actually represent different remote objects.

The straightforward solution is to execute an operation on the remote system that determines the correct answer. However, beyond the performance penalty that this solution incurs, it also interferes with the goal of the Java Platform Debugging Architecture which is to isolate the debugger from the target application as far as possible in order to avoid potential side effects.

The COMPARAND pattern solves this problem by adding an attribute to the remote references, the so-called comparand. The comparand of a particular reference is assigned a value that uniquely identifies its remote object.

Consequently, only a comparison of comparands is needed in order to determine sameness or difference of the respective remote objects. They can therefore be used to carry out the comparison operation efficiently. Interferences with the execution of the target application are reduced to the actual creation of comparands inside the JVM of the target application.

Context

Comparison of objects with reference semantics without comparing their references.

Problem

Object comparison is usually taken to mean either comparison for sameness (object identity) or comparison of state. The first of these approaches corresponds to so-called *reference semantics*, usually based on comparison of references or pointers; the second approach corresponds to *value semantics*, using all or a subset of the attributes.[1]

However, neither of these approaches is sufficient when reference semantics is to be maintained but reference comparison does not ensure sameness. This is the case when there are different references that can refer to conceptually the same object. In the motivating example, remote references are represented as objects on their own: for this reason, the target object together with its remote references form a conceptual entity that should be indistinguishable from the outside. So the issue is not how to change reference semantics to value semantics but how to have reference semantics with a comparison operation that does not simply compare the references. As another example, particularly in distributed systems, several proxies [8] in the same address space refer to either the same or to different remote objects (see fig. 2). In this case, the fact that a lack of a direct reference mechanism for remote objects has to be overcome results in potentially ambiguous references.

---

[1] Other semantics for object comparison include more complex equality operations that, for example, take structural equivalence into account. See [1] and [9] for discussions on the range of possible equality semantics.

Another example is an implementation of the DECORATOR pattern [8], where not only different decorators may be applied to the same core object, but they can even decorate each other since decorators and decorated objects have the same interfaces in general. (See [8] for examples.) Here, comparison of references might not reveal that they actually refer to the same core object, but there is a need to establish sameness for decorator objects that are strictly different.[2]

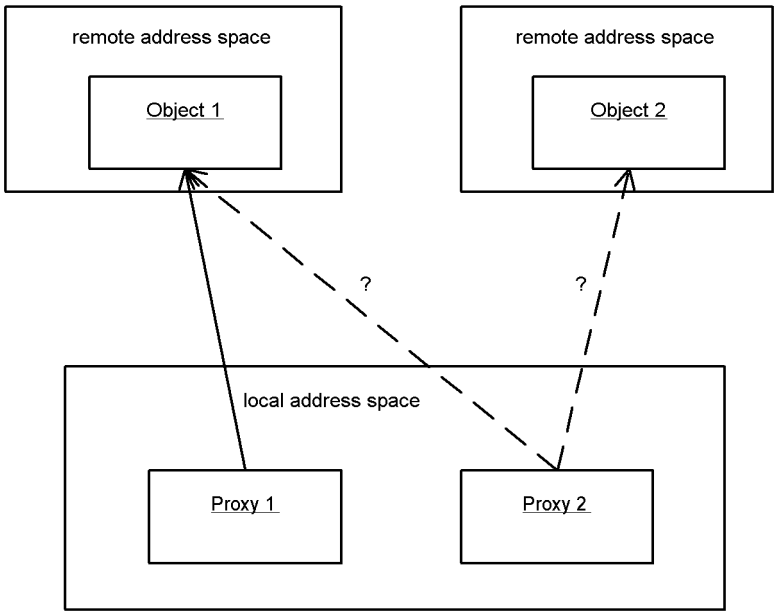

Fig. 2: Do the two proxies refer to the same remote object or not?

In such circumstances, the following forces have to be balanced:

- A comparison of object state does not yield the intended result since reference semantics is desired.
- A comparison of references cannot be relied upon since one wants to consider different objects to be the same. These objects might even be instances of different types.
- The sameness of objects in general depends on the context. Objects that are considered the same in one context can be different in another.
- If an object is copied[3], care must be taken how to define comparison between original and copy. There are cases where comparison between the two should yield `true` – leaning more towards value semantics – but others where they must be distinguished.

---

[2] The latter is also known as an example of a split object [2].

[3] For the purposes of this paper, the `clone` method is just one means of allowing an object to be copied. Therefore the two are used interchangeably except where implementation details are discussed.

- Sometimes comparison of objects of different types must yield `true`, for example different decorators of the same object, especially decorators of decorators.
- A system may want detailed control of the possible results of object comparisons. For example, when the cost of object creation is to be lowered by introducing a recycling mechanism, the expected result of comparison even changes over time.
- Comparison is performed frequently. Therefore, it must be a cheap operation in terms of runtime overhead. This requires particular attention in distributed systems.
- In a distributed system, it is usually non-trivial to determine sameness of object references locally and executing a remote call for comparison introduces a significant performance overhead.
- The additional memory overhead associated with achieving the desired comparison behavior often needs to be small, especially if many objects are involved.

Solution

Introduce an instance variable in each class of interest – the comparand – that does not belong to the conceptual state of their respective objects, and compare objects by comparing their comparand values.[4]

```
public class MyClass {

   protected static long comparandCounter = 0;

   protected long comparand = comparandCounter++;

   public boolean equals(Object obj) {
      if (obj instanceof MyClass) {
         MyClass that = (MyClass)obj;
         return this.comparand == that.comparand;
      }
      return false;
   }

   // rest of class body
   ...

}
```

[4] We have chosen the artificial name COMPARAND for this pattern to stress that this instance variable is a passive entity that is not used for referencing, but within comparison operations only. Elsewhere, names like "key" and "identifier", or acronyms like "OID" and "id" are used for this concept, but these names are used ambiguously and with overloaded meanings throughout the literature. Many brainstorming sessions have not revealed a better name, so we have opted for COMPARAND.

The comparands stored in the objects under consideration can be either values of a primitive type or instances of a compound type. Primitive values of 64 bits are large enough to allow 10 billion unique comparands per second to be created for half a century which is good enough for almost all applications.[5] In this case, unique comparands can always be created efficiently by just increasing a global counter. Therefore, in a local context that allows the management of comparands to be centralized, there is no reason to use compound comparands with the associated performance and memory overhead.[6]

Implementation

There are some subtle issues when applying the COMPARAND pattern, which are discussed in the following sections.

The "Right" Comparison Semantics

It is important to thoroughly understand what exactly object comparison is supposed to mean in the context at hand. The Comparand pattern is applicable only if the intended behavior is that of reference semantics, but not that of value semantics or even of some intermediate semantics.[7]

Sometimes different contexts require different comparison semantics. For example, after application of the DECORATOR pattern the core object and its decorators represent the same conceptual entity. However, certain clients expect the comparison of decorator objects to determine if their respective core objects are the same, whereas other clients need to differentiate between the decorators. The introduction of more than one comparison operation (for example `equals` and `equalsDecorator`) is advisable under these circumstances.

Comparison of Clones

If an object can be copied or cloned, typically afterwards both objects have exactly the same state, but they are not identical. The COMPARAND pattern offers the flexibility to define any desired degree of sameness. There is a free choice to assign the copy the original comparand or a new one which can even be based on dynamic properties of the environment. However, then one must consider the question what the correct behavior

---

[5] 32 bit values, on the other hand, are usually not big enough to ensure uniqueness for long-running applications. At a rate of 1000 comparands per second, they wrap around after roughly 1 ½ months. In the rare cases when even 64 bits are insufficient, two or more long integers can easily be combined in a customized value type with a larger range of numbers.

[6] This need only arises in the case of distributed applications. See *Comparands in Distributed Environments* in the Implementation section for further details.

[7] Sometimes, comparison of objects needs to take sophisticated aspects into account, for example structural equivalence of complex object types. A thorough overview of these issues is given in [9].

should be in a given context. In the general case, a new comparand should be assigned by default, as illustrated in the following example, since clones can usually be regarded as independent instances.

```
public class MyClass implements Cloneable {

   // comparand and equals() as above
   ...

   public Object clone() {
      try {
         MyClass myClone = (MyClass)super.clone();
         myClone.comparand = comparandCounter++;
         return myClone;
      } catch (CloneNotSupportedException e) {
         // since MyClass implements Cloneable
         // this exception cannot occur
         throw new InternalError();
      }
   }
}
```

However, an example of a system that may need to treat objects and their clones as equal is one that offers transactional services. It creates copies of objects to operate on them instead of the original ones, so that a rollback operation is easily implemented by just discarding these copies. From a system programmer's point of view, the disambiguation of copies from original objects is clearly needed, but from an application programmer's point of view it is not desirable to distinguish between them. Again, the introduction of more than one dedicated comparison operation (with different access rights, if applicable) may solve this problem.

Which Classes Are Comparable To Each Other?

Another important issue is the determination of the classes that are supposed to be comparable. If it is required to potentially establish identity for any two objects of arbitrary type, further effort is needed. For example, in Java an interface can be introduced that otherwise unrelated classes can implement, allowing their objects to be compared as follows.

```
public interface Comparable {
   public long getComparand();
}
```

Since in this case the creation of comparands does not naturally belong to one of the comparable classes anymore, it should be factored out into a class of its own.[8]

[8] Note that the `getNewComparand()` method must be synchronized in the presence of multi-threading.

```
public class ComparandFactory {

   private static long comparandCounter = 0;

   public long getNewComparand() {
      return comparandCounter++;
   }
}

public MyClass implements Comparable {

   protected long comparand = ComparandFactory.getNewComparand();

   public long getComparand() {
      return this.comparand;
   }

   public boolean equals(Object obj) {
      if (obj instanceof Comparable) {
         Comparable that = (Comparable)obj;
         return this.comparand == that.getComparand();
      }
      return false;
   }

   // rest of class body
   ...

}
```

Provided that each `Comparable` class implements `equals` in this way, any two `Comparable` objects can be made the same by assigning their comparands the same value.

Boundary Conditions of a Given System

A good understanding of the properties and the "feel" of the environment at hand is important. Are there standard ways to establish and determine sameness? For example, in C++ sameness is usually determined via the `operator==`, so it is advisable to redefine it accordingly, whereas in Java the == operator cannot be redefined, but instead the `equals` method has to be overridden and used.

What kinds of comparison and guarantees of uniqueness are provided or required by the programming language and the libraries and frameworks to be used?

For example, libraries for collections usually expect comparison operations to behave well, as in the case of Java's Collection Framework that requires the standard `hashCode` method to return the same result for two objects that are equal in terms of the standard `equals` method. In fact, the comparand should be used as a hash code value for this reason, as shown in the following code fragment.[9]

---

[9] See the JDK documentation on `hashCode()` in `java.lang.Object` for further details [18].

```
public class MyClass {
   // comparand and equals() as above
   ...

   public int hashCode() {
      return (int)this.comparand;
   }
}
```

Reuse of an Existing Attribute

There are cases where there is no need to define and create comparands specifically. For example, in frameworks that map objects to table entries in relational database systems, primary keys are good candidates for comparands, especially when they are created by some kind of sequence number generator inside the database system. However, care must be taken to ensure that the preexisting attribute exactly reflects the intended comparison semantics. There are deceptive cases where an attribute "accidentally" reflects the intended semantics without being conceptually bound to it, in which case it is better to introduce a dedicated attribute.

Execution of Comparison Operations

There are two options in this dimension of variance. On the one hand, the objects that hold the comparands can offer methods to carry out the comparison, hiding the fact that the COMPARAND pattern is used for this purpose (*internal comparison*). This allows one to change the implementation later on and base it on a technique other than the COMPARAND pattern as required. The implementation can be scaled down to even a comparison of plain references as offered by the programming language when the reasons for an advanced solution have vanished.[10]

On the other hand, objects can allow one to access the comparands and perform the comparison directly (*external comparison*). This variant may be opted for when comparands offer additional functionality and there is therefore already a need to access them. For example, comparands can also serve as keys for later retrievals of the same object. In this case, comparison of two objects looks like follows. (Note that casts to the `Comparable` interface are not always necessary.)

```
if ((obj1 instanceof Comparable) &&
    (obj2 instanceof Comparable)) {
   Comparable comp1 = (Comparable)obj1;
   Comparable comp2 = (Comparable)obj2;
   if (comp1.getComparand() == comp2.getComparand()) {
      ....
   }
}
```

[10] Other details of the specific implementation are also encapsulated and therefore easily exchanged, like the issues of primitive types vs. compound types, and so on. See *Comparands in Distributed Environments* for further details on compound comparands.

The two options are not mutually exclusive: an object can offer both internal and external comparison. However, in this case, the specific advantage that internal comparison hides the implementation details of the COMPARAND pattern vanishes, and therefore, pure internal comparison is a better alternative in the general case.

Comparands in Distributed Environments

Especially in the case of distributed systems, the COMPARAND pattern can significantly reduce the runtime overhead of comparison operations. When the comparand of a remote object is cached within each of its remote references[11], comparisons do not require any remote execution at all (see fig. 3). Instead of allowing various remote references for the same remote object to coexist, a system can choose to unify remote references as soon as they enter an address space. Since this guarantees the uniqueness of remote references they can directly be compared as such.

However, in order to check if an old reference must be reused or a new one must be created, the system has to keep a table that maps comparands, which are determined via the underlying communication mechanism, to the actual remote references.[12]

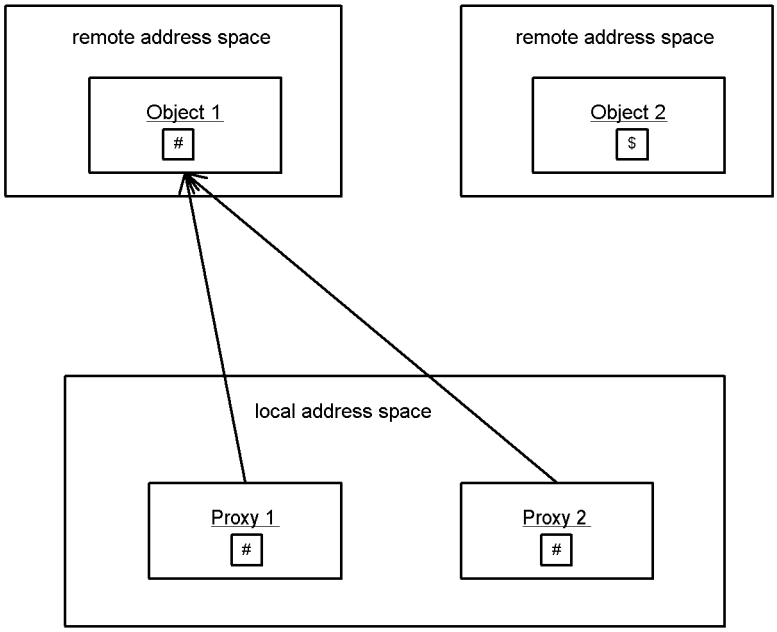

Fig. 3: The sameness of a remote object can be determined locally by comparing the comparands.

In distributed systems, the goal of unique comparands can be achieved only at great expense. Uniqueness can be complicated even further when a heterogeneous application has to be built which consists of independently developed subsystems. The following variants of the COMPARAND pattern offer different solutions for this problem.

---

[11] thus making it a simple instance of the CACHE PROXY Pattern [16]

[12] Note that comparands should always be implemented with value semantics rather than reference semantics, since only values can be copied across machine boundaries.

Ambiguous Comparands Instead of trying to achieve the goal of globally unique comparands, the requirements can be relaxed by letting all participating subsystems independently create potentially overlapping sets of comparands.

In this case, two objects might have equal comparands by accident. Therefore, one needs to know whether these comparands stem from the same subsystem in order to definitely determine sameness. As a last resort, the comparison operation has to be executed remotely. However, two objects that have different comparands are guaranteed to be different. The aim of avoiding remote invocations is not fully achieved, but the looser coupling of the systems involved outweighs this loss of performance, depending on the frequency of comparison operations.

Compound Comparands Instead of sacrificing uniqueness of comparands, compound comparands can store identifiers for the process in which the respective objects live. Comparand creation is then a process that involves several steps, such as the creation of a unique number within a server and incorporating a server identifier into comparands within clients.

The basic implementation scheme for compound comparands is as follows.

```
public class Comparand {

   protected java.net.URL remoteSystem;
   protected int processNo; // identifies an address space
   protected long remoteComparand;

   public Comparand(java.net.URL remoteSystem,
                    int processNo,
                    long comparand) {
      this.remoteSystem = remoteSystem;
      this.processNo = processNo;
      this.remoteComparand = comparand;
   }

   public boolean equals(Object obj) {
      if (obj instanceof Comparand) {
         Comparand that = (Comparand)obj;
         return this.remoteSystem.equals(that.remoteSystem) &&
               (this.processNo == that.processNo) &&
               (this.remoteComparand == that.remoteComparand);
      }
      return false;
   }

   public int hashCode() {
      return (int)remoteComparand;
   }

   // note: no redefinition of clone()!

}
```

A class for remote references that uses compound comparands looks as follows.

```
public class MyRemoteReference {

   protected Comparand comparand;

   public MyRemoteReference(java.net.URL host,
                            int processNo,
                            long comparand) {
      this.comparand = new Comparand(host, processNo, comparand);
   }

   public boolean equals(Object obj) {
      if (obj instanceof MyRemoteReference) {
         MyRemoteReference that = (MyRemoteReference)obj;
         return this.comparand.equals(that.comparand);
      }
      return false;
   }

   public int hashCode() {
      return this.comparand.hashCode();
   }

   // note: no redefinition of clone()!

}
```

Note that there are some fundamental differences between this implementation and the example that is given for non-distributed applications earlier in this paper. Firstly, remote references do not request the creation of a totally new comparand but let a comparand be initialized with given values that identify an existing remote object. This information must be determined via the underlying communication mechanism (for example IP). Secondly, the `clone()` method is not redefined since a clone of a remote reference refers to the same remote object by definition.

Computed Comparands Comparands may be computed by an algorithm that takes considerable effort to ensure global uniqueness. For example, GUIDs in the Microsoft Component Object Model (COM) can be used as 128 bit comparands. Again, counters that are global for the current machine are taken into account, together with the local machine's network address and the current time in order to ensure (world-wide) global uniqueness [3]. Since GUIDs store all this information in a standardized way, they may still be regarded as a special case of compound comparands. However, since GUID creation imposes a significant runtime overhead, in the general case ambiguous comparands and compound comparands are preferable.

Coordinated Comparands Another viable alternative for ensuring unique comparands is the assignment of non-overlapping sets of comparands to each node of a distributed application. Then each node is responsible for providing objects with unique comparands from the range of permitted comparands. This implies the need for a central comparand server that coordinates the creation of these non-overlapping sets and their assignment to the respective nodes. A possible disadvantage of this approach is the dependency on the availability of the comparand server. On the other hand, the

access rate can be scaled by the number of comparands that are granted on each request.[13]

Consequences

Using COMPARANDs to compare objects yields the following benefits.

Flexibility The use of comparands makes it easy to define sameness of objects in an arbitrary way. It is even possible to change sameness at run time without affecting the objects' state. In addition, it is possible to make objects of different types equal, for example different decorators wrapping the same object.

Comparison is cheap Comparison using primitive comparands is about as cheap as possible in terms of performance overhead.

Comparison of remote objects Proxies of remote objects can cache comparands locally, allowing remote references to be compared without the need for network traffic. This provides an efficient way to implement unification of remote references.

There are however several liabilities.

Complexity As is often the case, flexibility comes at the cost of increased complexity. The use of comparands makes it more difficult to understand which objects are the same by looking at their implementation. The code that determines equality of objects can be part of objects other than those being compared, scattering the definition of sameness across several classes.

Collections If the default comparison mechanism of the language (`equals` in Java, `operator==` in C++) is implemented with comparands, care must be taken when container classes are used. Many container implementations rely on comparison of the contained objects, and if several objects have the same comparands, unexpected behavior can result.

Memory overhead The COMPARAND pattern relies on the introduction of an additional attribute, incurring some memory overhead. This can be an issue if the number of objects is large or compound comparands are used.

Known Uses

Java Platform Debugger Architecture

The JPDA [11] does not only include a set of specifications, as introduced above, but also a standard implementation of all key components. The implementation of the Java Debug Interface uses comparands extensively to compare general ("user-defined") objects, strings, arrays, class loaders, and threads as well as reified types, fields and methods.

The comparands are implemented as `long` integer values (field `ref` in class `com.sun.tools.jdi.ObjectReferenceImpl`). In principle, the implementation

[13] There is no completely satisfactory solution to this problem because of the inherent unreliability of distributed applications. For example, see [6].

allows a debugger to connect to more than one virtual machine at the same time. For this reason, objects that have the same comparands are not necessarily the same. Therefore, a representation of the originating virtual machine is also taken into account during comparison. Consequently, the comparands can be created independently by their respective hosts.

Although the Java Debug Interface offers methods to retrieve the comparands of remote references, these comparands cannot be used to carry out comparison operations because of their ambiguity. Therefore, dedicated comparison methods are offered in addition.

See [10] for the source code of JDK 1.3, which also includes the sources of the standard implementation of the Java Platform Debugger Architecture.

Remote Method Invocation

In Java RMI [12], remote objects are represented by objects that implement the `java.rmi.server.RemoteRef` interface. In the standard implementation of RMI (as of JDK 1.3), this interface is implemented by the `sun.rmi.server.UnicastRef` class. This class uses the COMPARAND pattern to compare remote objects by comparing the field ref of type `sun.rmi.transport.LiveRef`, that is defined for this class. This field consists of a representation of a server ("Endpoint"), a unique address space within that server ("UID") and a unique `long` integer value corresponding to an object within that address space.

This representation of remote objects allows each server to create their own respective sets of values representing actual objects. Since remote references always record the execution context of their remote objects, they are the same if and only if they have the same comparands.

The `java.rmi.server.RemoteRef` interface does not allow the retrieval of the comparands of remote references, but it completely hides the fact that comparands are used in the standard implementation. Instead, a `remoteEquals` method is offered to carry out the comparison operation.

Again, see [10] for the source code of JDK 1.3, which also includes the sources of the standard implementation of RMI.

CORBA Relationship Service

In principle, CORBA does not provide any means to compare components. However, the Relationship Service Specification [14] defines the CosObjectIdentity module which includes an `IdentifiableObject` interface. It defines a `long` integer attribute as a comparand ("ObjectIdentifier").

Since this value is not guaranteed to be unique, two objects that have the same comparands are not necessarily the same. In order to definitively determine if two component references refer to the same component, an `is_identical` operation is also defined that has to be carried out remotely.

The "ObjectIdentifier"-comparands are explicitly meant to be used as keys in hash tables. Therefore they can be accessed directly as readonly attributes.

Enterprise JavaBeans

In Enterprise Java Beans [17], the so-called entity beans offer primary keys which can be obtained by `getPrimaryKey` methods. For example, they can be used to retrieve or remove the components they represent and they can also be used as comparands.

Again, comparison of such primary keys does not completely determine whether two references refer to the same component. If they are equal, it must be determined whether they are obtained from the same execution context (the so-called "home") or otherwise an `isIdentical` method has to be invoked remotely.

Whereas primary keys are technically realized as instances of possibly user-defined primary key classes, these classes are restricted to be legal Value Types in RMI-IIOP [13]. These Value Types are constrained in a way that essentially leads to classes with value semantics rather than reference semantics. For example, they are required to redefine Java's standard `equals` method accordingly.

Ginko

Ginko [15] is an email client for the Apple Macintosh (including Mac OS X), and is implemented in Objective-C. One of its features is the unified handling of different copies of the same email. Emails are represented as objects and can be stored into more than one folder whilst keeping the same set of attributes, such as status information and priority markers. The repeated receipt of the same email is also detected by Ginko.

Different instances of the same email are identified by comparison of the standard `MESSAGE-ID`, as specified by the Internet Request For Comments document number 822 [5]. As RFC 822 states, the "uniqueness of the message identifier is guaranteed by the host which generates it". Therefore in Ginko, these `MESSAGE-ID`s are used as comparands and they are equal if and only if the corresponding emails are the same.

Related Patterns

Several of the standard patterns from [8] employ some kind of delegation to let methods of one object operate on behalf of another. If a multitude of objects delegate to a single object, implementations of these patterns can apply the COMPARAND pattern instead of delegating requests for comparison to the respective target objects. The patterns that can take advantage of the COMPARAND pattern in this way are ADAPTER, BRIDGE, DECORATOR and PROXY.[14]

The OID pattern from [4], which can be regarded as a special case of the COMPARAND pattern, is restricted to the context of integrating objects and relational database systems. It discusses only primitive types (integer or strings) as candidates for comparands, and favors the use of sequence number generators, which are built into some relational database systems, as sources for comparand creation.

---

[14] Other patterns from [8] that also use delegation are STATE and STRATEGY. However, they are not candidates for the application of the COMPARAND pattern, since in these cases, the respective target objects do not play an "identifying" role, so it makes no sense to compare them at all.

Conclusion

There are several techniques for implementing object comparison, depending on the desired semantics and the context of its use, with reference comparison being built into almost all programming languages and therefore being most widely employed. An interesting distinction between the COMPARAND pattern and reference comparison is the following asymmetry. With the COMPARAND pattern, two objects are guaranteed to be the same if their comparands are equal; with reference comparison, two objects are guaranteed to be equal if their references are the same. The latter case is often utilized to optimize otherwise complex comparison operations.

We believe that these considerations could be extended into a useful pattern language covering the realm of object comparison. Other sources that should be taken into account are [1] and [9], which discuss various aspects of object comparison, and the EXTRINSIC PROPERTIES of [7], which can also be used as a means to determine object equality, to name just a few.

Acknowledgements

The authors thank James Noble for shepherding this paper; the other members of this paper's Writers' Workshop at EuroPLoP 2001 - Fernando Lyardet, Juha Pärssinen, Gustavo Rossi, Dietmar Schütz and Sherif Yacoub; and furthermore, Tom Arbuckle, Michael Austermann, Peter Grogono, Axel Katerbau, Günter Kniesel, Thomas Kühne, Markus Lauer, Oliver Stiemerling, Clemens Szyperski, Dirk Theisen and Kris De Volder for many fruitful discussions on earlier drafts and related publications which led to substantial improvements.

Pascal Costanza's contribution to this work has been carried out for the TAILOR project at the Institute of Computer Science III of the University of Bonn. The TAILOR project is directed by Armin B. Cremers and supported by Deutsche Forschungsgemeinschaft (DFG) under grant CR 65/13.

References

[1] Henry Baker, "Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same", ACM OOPS Messenger 4, 4, October 1993.

[2] Bardou, D., C. Dony, "Split Objects: a Disciplined Use of Delegation within Objects", OOPSLA '96, Proceedings, 122-137, ACM Press, 1996.

[3] Don Box, "Essential COM", Addison-Wesley, 1998.

[4] Brown, K., B. Whitenack, "Crossing Chasms: A Pattern-Language for Object-RDBMS Integration", in J. Vlissides, J. Coplien, N. Kerth (eds.), "Pattern Languages of Program Design 2", Addison-Wesley, 227-238, 1996.

[5] Crocker, D. H., "Standard for the Format of ARPA Internet Text Messages", Internet Request For Comments (RFC) 822, 1982, http://www.ietf.org/rfc/rfc0822.txt

[6] Peter Deutsch, "The Eight Fallacies of Distributed Computing", http://java.sun.com/people/jag/Fallacies.html

[7] Fowler, M., "Dealing with Properties", 1997. Downloadable via http://www.martinfowler.com

[8] Gamma, E., R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Systems", Addison-Wesley, 1995.

[9] Grogono, P., M. Sakkinen, "Copying and Comparing: Problems and Solutions", in: E. Bertino (ed.), "ECOOP2000 – Object-Oriented Programming", Proceedings, Springer LNCS 1850, 226-250, 2000.

[10] Java 2 Platform, Standard Edition (J2SE), Sun Community Source Licensing, http://www.sun.com/software/communitysource/java2/

[11] Java Platform Debugger Architecture, http://java.sun.com/products/jpda/

[12] Java Remote Method Invocation, http://java.sun.com/products/rmi/

[13] Object Management Group, Inc., "Java Language to IDL Mapping Specification", June 1999. Downloadable via http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm

[14] Object Management Group, Inc., "Relationship Service Specification", Version 1.0, April 2000. Downloadable via http://www.omg.org/technology/documents/formal/relationship_service.htm

[15] Objectpark Group, http://www.objectpark.org

[16] Rohnert, H., "The Proxy Design Pattern Revisited", in J. Vlissides, J. Coplien, N. Kerth (eds.), "Pattern Languages of Program Design 2", Addison-Wesley, 105-188, 1996.

[17] Sun Microsystems, Inc., "Enterprise JavaBeans Specification, Version 2.0", October 2000. Downloadable via http://java.sun.com/products/ejb/docs.html

[18] Sun Microsystems, Inc., "Java 2 SDK, Standard Edition Documentation, Version 1.3.1", http://java.sun.com/products/jdk/1.3/docs/, 2001.