Value Object

Dirk Riehle

SAP Research, SAP Labs LLC

3475 Deer Creek Rd, 94304 Palo Alto, CA, U.S.A.

+1 650 215 3459

dirk@riehle.org, www.riehle.org

1 Intent

Implement datatypes as immutable classes so that their instances can be handled similar to built-in values.

2 Also Known As

Whole Value.

3 Motivation

Consider a financial services application for handling deposits and withdrawals to and from accounts. You are dealing with monetary amounts like '$42' or '€107'. Let's assume you need to maintain the balance for some account. It is tempting then to represent the balance using a currency symbol and a BigDecimal for the monetary amount.

```
public class Account {
    protected char currency = '$';
    protected BigDecimal balance =BigDecimal.ZERO;
    …
}
```

After a short consideration, you decide it is better to create a dedicated Money class to hold both the currency symbol and the monetary amount. The Money class will have methods for getting and setting the currency symbol, as well as methods for doing some arithmetic with the monetary amount, like adding or subtracting money.

```
public class Money {
    public static final Money ZERO = new Money(
        '$', BigDecimal.ZERO
    );

    protected char currency = '$';
    protected BigDecimal amount =
        BigDecimal.ZERO;
```

```
    public Money(
        char myCurrency,
        BigDecimal myAmount
    ) {
        currency = myCurrency;
        amount = myAmount;
    }

    public synchronized void add(
        Money otherMoney
    ) {
        addAmount(otherMoney.getAmount());
    }

    public synchronized void addAmount(
        BigDecimal otherAmount
    ) {
        amount = amount.add(otherAmount);
    }

    …
}
```

An Account class then makes use of Money objects.

```
public class Account {
    protected Money balance = Money.ZERO;
    …

    public synchronized void deposit(
        Money moreMoney
    ) {
        balance.add(moreMoney);
    }

    public synchronized void withdraw(
        Money lessMoney
    ) {
        balance.subtract(lessMoney);
    }

    …
}
```

So far so good. You go ahead and use the Account and Money classes for implementing a money transfer method on a 'financial application server' class. (We are ignoring transaction and failure handling to keep it simple.)

```
public class FinAppServer {
    public static synchronized void transfer(
        Money money, Account from, Account to
    ) {
        from.withdraw(money);
        to.deposit(money);
    }
    …
}
```

A simple unit test checks this implementation:

```
public void testTransfer() {
    Account from = new Account();
    Account to = new Account();

    from.deposit(new Money('$', 42));

    FinAppServer.transfer(
        new Money('$', 3.14), from, to
    );

    BigDecimal fromAmount =
        from.getBalance().getAmount();

    assert(
        fromAmount.doubleValue() == (42 - 3.14)
    );

    BigDecimal toAmount =
        to.getBalance().getAmount();
    assert(toAmount.doubleValue() == 3.14);
}
```

If you are like me, you'll be surprised to learn that this test case fails. Both from and to accounts hold a '$42' balance, which is wrong. (The from account should have an amount value of 42 − 3.14 and the to account should have an amount value of 3.14.)

You probably quickly figure out what's going on: Both accounts started out with the ZERO Money object as its balance; the initial deposit of '$42' to the from account also set the to account balance to '$42', after all, it is the same Money object. The subsequent subtraction (in the withdraw method) and addition (in the deposit method) of '$3.14' evened out the value change in the balance object. Since both accounts are holding the same object as their balance, both will display '$42'.

This is the aliasing or side-effect problem. Here, it may appear trivial to avoid, but it easily gets more complicated: The side-effects of changing an object that is referenced from another place are not always easy to comprehend. If we can avoid such nasty surprises, we will be well served.

To quickly fix the bug, you change the balance initialization code of the Account class:

```
protected Money balance = new Money('$', 0.0);
```

The testTransfer test case works nicely now, and you decide to enhance the money transfer functionality. Specifically, you want a transfer to take place only if the account being withdrawn from does not fall below $0. So you rewrite the transfer method.

```
public static synchronized void transfer(
    Money money, Account from, Account to
) {
    Money fromBalance = from.getBalance();
    fromBalance.subtract(money);
    if (fromBalance.isLowerThan(Money.ZERO)) {
        throw new RuntimeException(
            "Insufficient funds!"
        );
    }

    from.withdraw(money);
    to.deposit(money);
}
```

This time we don't have to write a test case to see that this won't work. The subtraction in line 3 not only changes the local variable

fromBalance but also the value of the underlying balance of the from account. If we were to run a transfer where the from account had enough money, it would find the transfer amount to be withdrawn twice. Firstly, from the test to check whether the withdrawal will not overdraw the account, and secondly, from the real withdraw method call.

One possible solution is to make the Account object hand out only copies of its balance Money object. This way, no client could ever change the balance of an account without going through its regular methods. This also leads to a lot of copies if you check balances frequently. Many of these copies may never be changed and are likely to be discarded quickly. Basically, you are buying protection by copying eagerly without knowing whether you'll ever need that protection.

It is possible to pay the price of protection only when it is needed. For this, you need to make each Money object immutable, which means you make it return a new object every time where otherwise it would change its state. Let's look at the addAmount method to see what this means:

```
public Money addAmount(BigDecimal otherAmount) {
    BigDecimal newAmount =amount.add(otherAmount);
    return new Money(currency, newAmount);
}

public Money subtractAmount(
    BigDecimal otherAmount
) {
    BigDecimal newAmount = amount.subtract(
        otherAmount
    );
    return new Money(currency, newAmount);
}
```

Rather than changing its internal fields, a Money object returns a new Money instance that holds the desired values. This way, the original Money object doesn't change. Read-only methods won't require a new object. You only create a new object if a state change occurs.

As a consequence, you have to change the way you program with Money objects. For example, the deposit method of the Account class now looks like this:

```
public synchronized void deposit(
    Money moreMoney
) {
    balance = balance.add(moreMoney);
}
```

Contrast this code with how it would have looked like if you had used a double instead of our Money class:

```
public synchronized void deposit(
    double moreMoney
) {
    balance = balance + moreMoney;
}
```

The code is structurally equivalent. Now that they are immutable, the code for dealing with Money objects looks like the code for dealing with built-in value types (data types). The Money class we have designed behaves like a value type. Its instances are called value objects, and the class itself is an application of the Value Object pattern.

4 Applicability

Use the Value Object pattern if

- the domain concept you are implementing represents a value type

and if

- the resulting class does not become too heavyweight as to slow down performance significantly.

We say that we implement "value objects" using a "value object class" that represents a "value type" as derived from modeling your application domain. Value type means the same as data type, but is more specific.

It would best if programming languages had language features that explicitly marked classes as value types, for example, by using the keyword "valuetype" rather than "class". (This is finally in the making, see [9], [12].)

The Value Object pattern is widely applicable. Value types are as fundamental as object classes, making the distinction between values and objects explicit. Examples of value types in the financial services domain are monetary amounts and currencies, in the Internet domain they are protocol names, domain names, and URLs, in math they are percentages, explicit fractions, and integrals, and in engineering they are the metric system and its units.

5 Structure
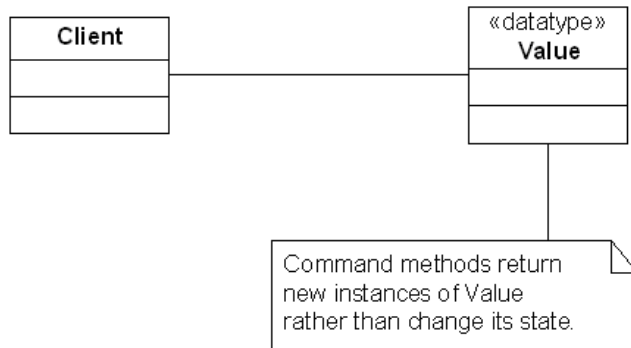
6 Participants

The *Client* class uses Value instances like built-in datatypes (to the extent that this is possible with your implementation language). Whenever the Client wants to change the value of one of its attributes, it replaces the attribute object rather than changing it. You write code like attr = attr.calculate(); rather than attr.calculate().

The *Value* class represents a domain-specific value type. It is implemented such that Value instances are immutable objects, meaning that its instances cannot change their state. Rather, as the result of some computation, a new Value instance with the desired internal state is returned.

7 Collaborations

A client creates new Value instances. Necessary data for initialization is provided as part of the constructor call or through a special initialization protocol. No state-changing methods should be published to clients and be used by them, even if these methods are public.

A Value instance provides information to a client through read-only query methods (a.k.a. observer methods). For what would traditionally be state-changing methods, the value object creates a new instance of its class and initializes it with the data that represent the result of the computation the current method is supposed to carry out.

A client that calls a command method of a value object receives a new Value instance back; it typically drops the old value object and keeps computing with the new value. Most notably, if the value is stored in an attribute of the client, the client replaces the value object representing the attribute's value with the new value object.

8 Consequences

The Value Object pattern provides the following general benefits:

- *Better domain modeling and understanding.* Recognizing a domain concept as a value type and implementing it as such reduces the gap between the domain model and its implementation. This eases code comprehension.

- *Safer programs.* Implementing value objects with immutable classes eliminates a whole class of bugs that comes from unwanted side-effects. Thus, your programs get safer.

- *Potentially better performance.* Value objects can improve system performance, because value objects are copied only if they are about to be changed. No superfluous copying happens.

There are more specific advantages, depending on where and how you use value objects.

- *Concurrency.* Because you can't change a value object's state, you don't have to synchronize methods or lock the object in other ways. Hence, you improve performance.

- *Persistence.* Value objects don't have to maintain an object id, hence you can directly write them to a relational database table without having to maintain them in separate tables with primary ids.

- *Serialization.* Similar to persistence, you can simply dump a value object as data into a data stream and you don't have to worry about dangling references from other parts of the object graph being serialized.

- *Distribution.* Also, as you copy regular objects across process boundaries, you always copy enclosed value objects too---you never pass on references to value objects across process boundaries.

- *Memory consumption and garbage collection.* By sharing value objects (see below for a discussion), you can poten-

tially safe a lot of memory and ease the burden on the garbage collector (like in the Flyweight pattern).

There are also some disadvantages.

- *More complicated code.* Value object classes require a little more code upfront than regular object classes. So clients need to understand that they are dealing with value objects, and need to look at how they are handled before using them.

- *Changes in coding style.* Programming with value objects may feel strange at first if you are not used to it. Effectively, you never change the state of a value object but rather assign a new value object to a client's attribute, whenever you made a computation with that attribute.

- *Potentially lower performance.* Unqualified use of value objects may reduce system performance. In particular heavyweight value objects that go through computations where a lot of these objects are created and dropped quickly may drag down performance.

In a given system, if it would distinguish value types from regular classes cleanly, the majority of domain concepts would be value types rather than object classes.

9 Implementation

A Value Object class is implemented like a regular immutable class. Query methods (getters, boolean query methods, etc.) simply return the requested state information. Command methods (setters, etc.) carry out the requested computation using method-local fields. They return a new instance of the Value Object class that gets initialized with the results of the computation taken from the method-local fields.

There are a variety of implementation issues to be considered.

- *Immutability.* Making an object immutable means that there are no state changing operations published to clients. You can make your life substantially easier (and your programs safer) if you can mark the fields of a value object class as final. This way, the compiler can catch any attempts to change a value object's state after initialization has taken place. It also makes your immutable classes thread-safe.

- *Identity, equality, and hash codes.* Values don't have identity, so it is important to properly implement anything that has to do with equality. In Java, this means to implement the equals() and hashcode() methods. Two value objects that are distinct objects may still mean the same value, and equality checking needs to realize this. In contrast to this, two regular objects that are distinct are never equal (otherwise they would be values).

- *Dropping synchronization.* With value objects, you can drop all synchronization code from your Value implementation. Please note that this does not hold true if you are sharing value objects to make sure that there is only one instance, see the discussion on value object sharing below.

- *Separating concerns with the Body/Handle idiom.* In some languages, it is easy to separate the copying of the value from a method's computation. In particular, in C++, using the Body/Handle idiom [3], you can use a copy-on-write pol-

icy to create a copy on the fly so the Value's body implementation itself does not have to worry about making the copy. Rather the handle object does it for its body. This makes implementation easier.

- *Mapping values to database tables.* For persistence, you can (and usually should) map complex values to one or more columns of an enclosing table. For example, the ACCOUNT table in an RDBMS should have both a BALANCE_CURRENCY column and a BALANCE_AMOUNT column to hold an account's balance. As you can see from the prefix BALANCE, we are mapping the balance attribute to two technically separate but conceptually related columns.

- *Dealing with heavyweight value types.* For heavyweight value objects, you sometimes give up on immutability. The problem here is that depending on how they are used, a lot of heavyweight value objects may be created only to be dropped quickly, because they are only an intermediate computation result. Such programming may put a significant burden on the garbage collector and downgrade performance in general. In such cases, it may be best to let the client handle the copying. This puts the burden of watching over possibly unwanted side-effects on the client. In spirit, this is still a value type, even though it may look like just a regular class.

- *Higher-order value types.* Some value types are best viewed as instantiations of value type constructors. A value type constructor is a value type that needs to be configured to give you a concrete value type. Traditional examples are ranges as in Pascal or Modula-2 (and many other programming languages). To get a datatype that accepts only integers in the range of 1..12 you write this: INTEGER MONTHS= INTEGER([1..12]); Another example are subsets. Unfortunately, Java supports neither range nor subset definitions.

- *Language support.* Structured or object-oriented programming languages typically don't support first-class value types. However, some well-known concepts can be used for value types. In Java 5, for example, you can use enums now to have an easily recognizable value type of finite cardinality. You still have to make sure your enum class is immutable, for example, by making its fields final---the compiler doesn't help you ensure it.

An important strategy for improving Value Object performance is sharing value objects (Value instances). Sharing value objects means making sure only one instance (or a defined pool) of any given value exists in the system. If you manage to efficiently share value objects, you can get a variety of performance benefits, for example, checking for object equality can be reduced to checking for object identity. Also, memory consumption and garbage collection can be reduced significantly.

- *Using Factory and Flyweight.* For sharing, you typically have a factory create and track the Value instances, much like an Abstract Factory creates objects, and much like a Flyweight tracks its instances [7]. It is common to hide the Value constructors and require clients to go through factory methods for new objects.

- *Retrieving vs. creating.* Retrieving a shared object is typically more expensive than creating a new object. This is because you usually need to initialize an empty value object

4

with the data used to retrieve the shared instance to create a hash code. In some circumstances, if the hash code is trivial, this may not be a problem.

- *Reintroducing synchronization.* Sharing makes your life more complicated though. It reintroduces synchronization, because you can't have two Value instances denoting the same value in the system. Hence, in the retrieval process for the shared value object you need to synchronize if you are about to create a new Value instance because none existed yet for the requested value. So you take a speed penalty.

- *Distinguishing by value type cardinality.* Sharing makes most sense if the value type's cardinality (possible number of Value instances) is finite. Currency has a finite cardinality, Money has not. As a consequence, at some point of time, you have created all the Currency objects and can rest assured that this is the upper limit for memory consumption. With Money objects, of which there may be any number, you don't know this.

- *Improving performance through eager initialization.* If you know that your application makes extensive use of a specific finite-cardinality value type, then it makes sense create all instances when the system starts up; otherwise you should initialize them on-demand. A hybrid strategy, employed by the JDK's BigInteger class is to create a frequently used subset during system startup and to create further instances on-demand.

10 Sample Code

Money and Currency are value types that you can implement as classes or enums. A simple straightforward implementation of Currency in Java 5 might utilize enums and look like this:

```
public enum Currency {
    USD("USD", '$', 2),
    EUR("EUR", '€', 2),
    JPY("JPY", '¥', 0);

    protected final String isoCode;
    protected final char symbol;
    protected final int noFractionDigits;

    private Currency(String myISOCode,
        char mySymbol, int myNoFractionDigits
    ) {
        isoCode = myISOCode;
        symbol = mySymbol;
        noFractionDigits = myNoFractionDigits;
    }

    …
}
```

Enums are a good tool to implement value types of known cardinality. If you know exactly what values there are and you can enumerate those, enums are your friends. Enums aren't automatically made immutable, so you still need to implement them properly. Please note that we achieved immutability by making all fields final.

If you don't know the exact number of your possible values, or if this number is simply too large to be enumerated easily, it is better to use a regular class. This is what the JDK does for currencies

with the java.util.Currency class, which represents all 200+ ISO currencies there presently are.

```
package java.util;
…

public final class Currency
    implements Serializable {

    private final String currencyCode;
    transient private final int
        defaultFractionDigits;
    …

    private static HashMap instances =
        new HashMap(7);
    …

    private Currency(String currencyCode,
        int defaultFractionDigits
    ) {
        this.currencyCode = currencyCode;
        this.defaultFractionDigits =
            defaultFractionDigits;
    }

    public static Currency getInstance(
        String currencyCode
    ) {
        return getInstance(
            currencyCode, Integer.MIN_VALUE
        );
    }

    private static Currency getInstance(
        String currencyCode,
        int defaultFractionDigits
    ) {
        synchronized (instances) {
            …
        }
    }

    public String getCurrencyCode() {
        return currencyCode;
    }

    …
}
```

A couple of things are interesting about java.util.Currency. First, it is a true Value Object class, that is, it is immutable and it its instances are used like values.

Secondly, it shares its instances: There can never be two Currency instances that both represent the US$. This sharing is realized by making the constructor private and forcing clients to use the static getInstance() method. A consequence of there never being duplicate Currency objects is that the Currency class does not have to implement equals() and hashcode(); the default implementations inherited from Object are sufficient.

Sharing is implemented in a straightforward way: Existing instances are maintained in a hash map. If an instance does not yet exist, it is created on the fly. This leads to an interesting trade-off that may or may not work for some clients. Within the getInstance() method, the code synchronizes on the instances hash map, incurring locking overhead. This is to avoid the problems with the double-checked locking pattern in Java [1]. The only way to avoid this performance penalty is to fully create all currencies upon system initialization, which has the downside of creating objects that most applications may never use.

Hidden in the Currency class code is the use of java.util.CurrencyData, which provides the initialization data for a new Currency value object. It is contains ISO 4217 currency code data. Unfortunately, there is no way of configuring this data so that your own reference data could be injected. For example, if you have your own symbols for precious metals, as many banks do, you may not be able to transparently use them as currencies. Also, java.util.Currency does not provide all information you may need. For example, minor units that belong to a currency (like 'cent' for the Dollar or Euro) are missing. So you may end up having to write your own currency class after all.

11 Known Uses

The distinction between objects and values in structured and object-oriented programming was first spelled out in the seminal article by MacLennan [10]. It has taken a while, but with the new breed of object-oriented programming languages like X10 [9] and Fortress [12], first-class value types seem to finally have arrived.

The most widely known uses of the Value Object pattern can probably be found in the Java JDK implementations. The String, Integer, and Float classes are close-to-the-system Value Object implementations. Further and somewhat cleaner examples are the BigInteger and BigDecimal classes, which are in fact implemented as immutable classes, with some caching for common numbers (e.g. values between -10 and +10). The JValue Value Object framework provides more examples: Hierarchical names, URLs, monetary amounts, UML datatypes, and others [11].

As a pattern, Value Object has first been described by Cunningham [4] and later by Fowler [6] and Evans [5]. We provide a lengthy discussion of Value Object design and implementation considerations in traditional programming languages in ([BRSetal98]).

12 Further Development

This paper is being developed on the web in a wiki at http://wiki.moredesignpatterns.com. You can find the latest revision there and comment on it. Any such comments are highly appreciated! Make sure you send me email too in order to get properly acknowledged.

ACKNOWLEDGEMENTS

REFERENCES

[1] David Bacon et al. "The 'Double-Checked Locking is Broken' Declaration." Available from http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

[2] Dirk Bäumer et al. Values in Object Systems. Ubilab Technical Report 98.10.1. UBS AG, 1998. Available from http://www.riehle.org/computer-science/research/1998/ ubilab-tr-1998-10-1.html

[3] James O. Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1992.

[4] Ward Cunningham. "The Checks Pattern Language of Information Integrity." In Pattern Languages of Program Design. Addison-Wesley, 1996.

[5] Eric Evans. Domain-Driven Design. Addison-Wesley Longman, 2004.

[6] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object Oriented Design. Addison-Wesley, 1995.

[8] James Gosling et al. The Java Language Specification. Addison-Wesley, 2005.

[9] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing." In Proceedings of the 2005 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005). ACM Press, Page 519-538.

[10] B. J. MacLennan. "Values and Objects in Programming Languages." ACM SIGPLAN Notices 17, 12 (December 1982). Page 70-79.

[11] Dirk Riehle. The JValue Value Object Framework. 1999. Available from http://www.jvalue.org

[12] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr, Sam Tobin-Hochstadt. The Fortress Language Specification Version 0.903. Available from http://research.sun.com.