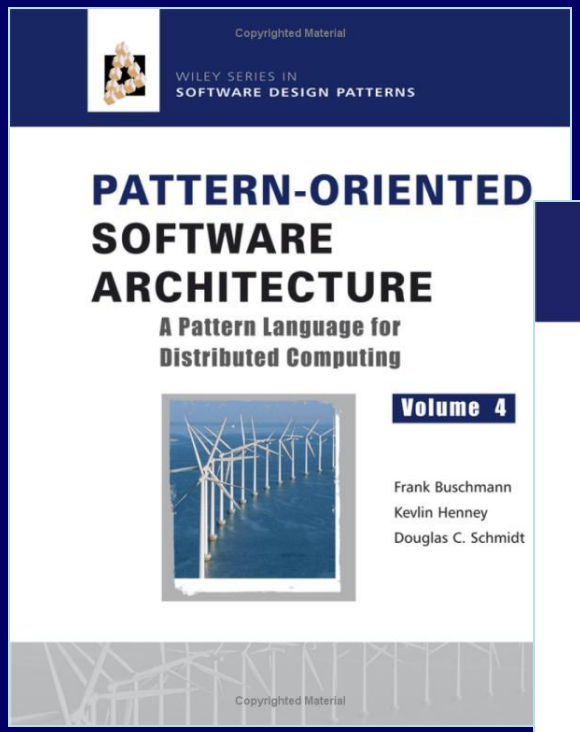
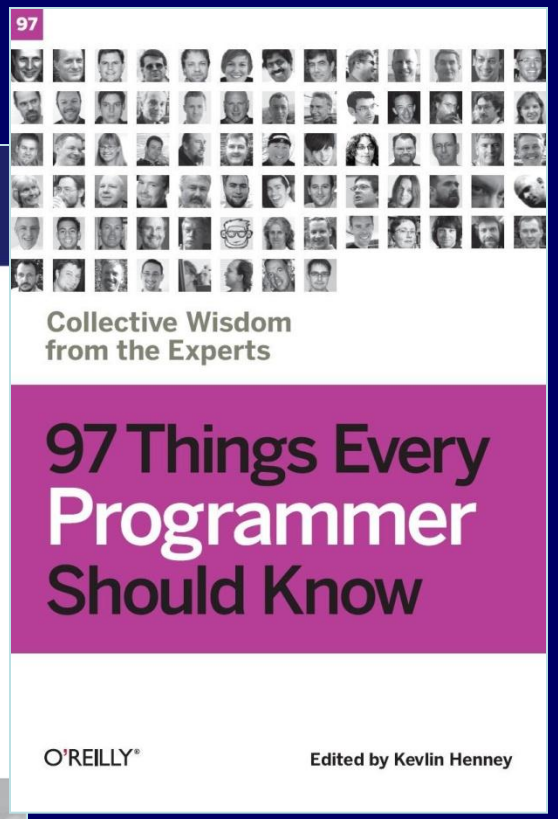
Objects of Value

Kevlin Henney

kevlin@curbralan.com

@KevlinHenney

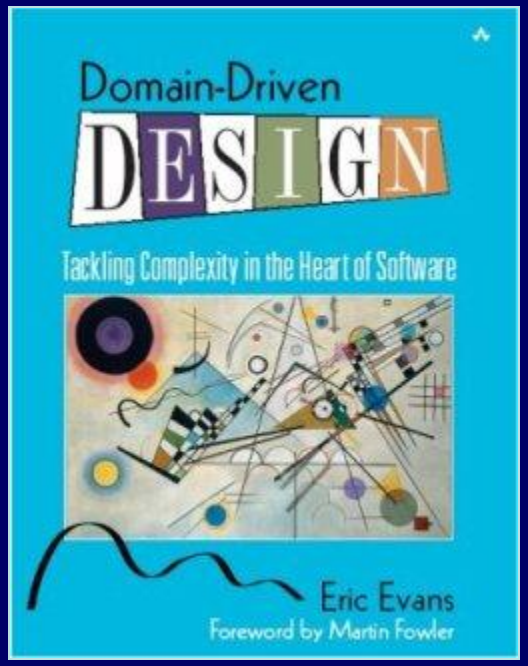




See *http://programmer.97things.oreilly.com*
(also *http://tr.im/97tepsk* and *http://tinyurl.com/97tepsk*)
and follow *@97TEPSK*

What Do We Mean by *Value*?

- The term *value* is used in many overlapping and contradictory ways
  - The mechanism of *pass by value*
  - A declarative construct, e.g., C# *structs* define programmatic *value types*
  - A kind of object representing fine-grained information in a domain model
  - The general notion of quantity or measure of something in the real world

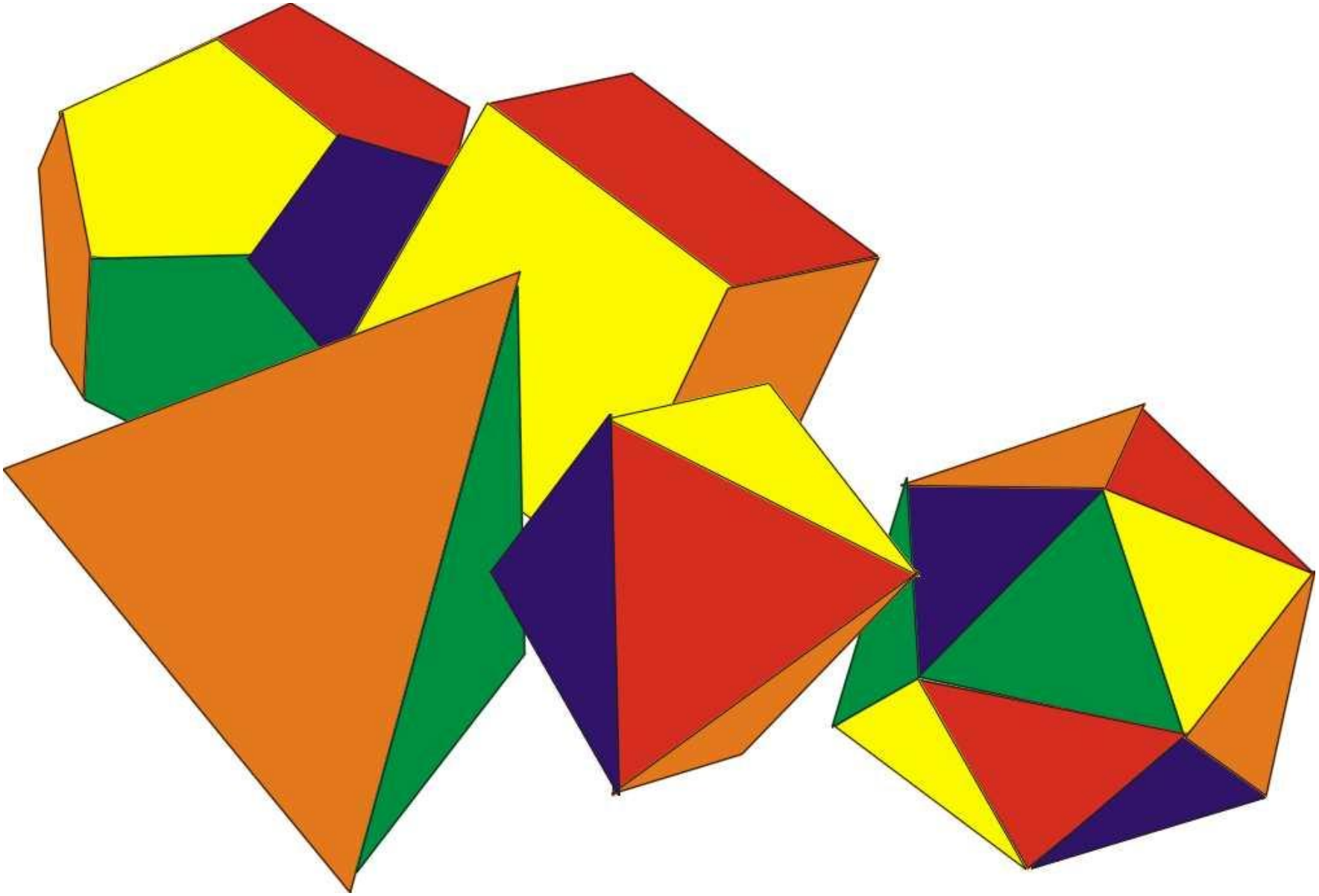


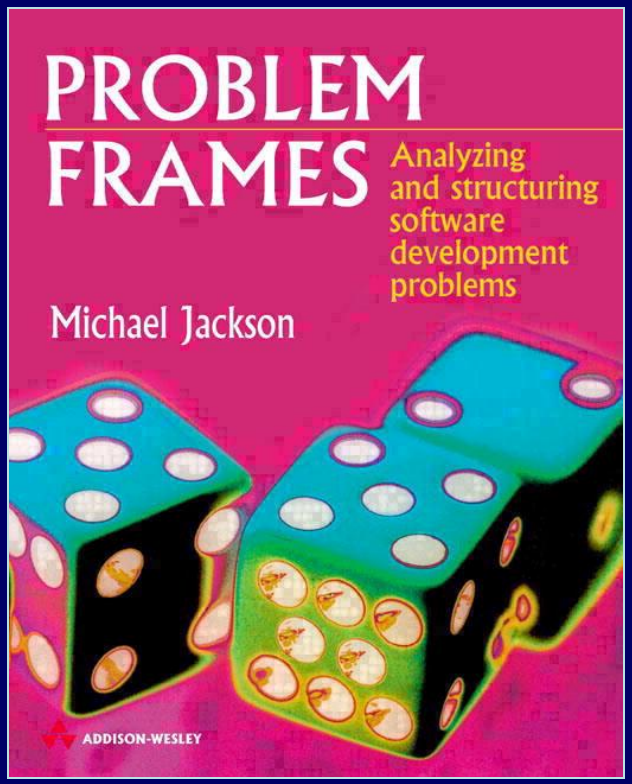
Many objects have no conceptual identity. These objects describe some characteristic of a thing. [...]

When you care only about the attributes of an element of the model, classify it as a VALUE OBJECT. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the VALUE OBJECT as immutable. Don't give it any identity and avoid the design complexities necessary to maintain ENTITIES.

Complementary Perspectives

- The Platonic Perspective
  - An idealised view of what values are in terms of maths and the physical world
- The Computational Perspective
  - A model-based view of what values are in terms of programming concepts
- The Language Perspective
  - The computational view bound to the specifics of a programming language

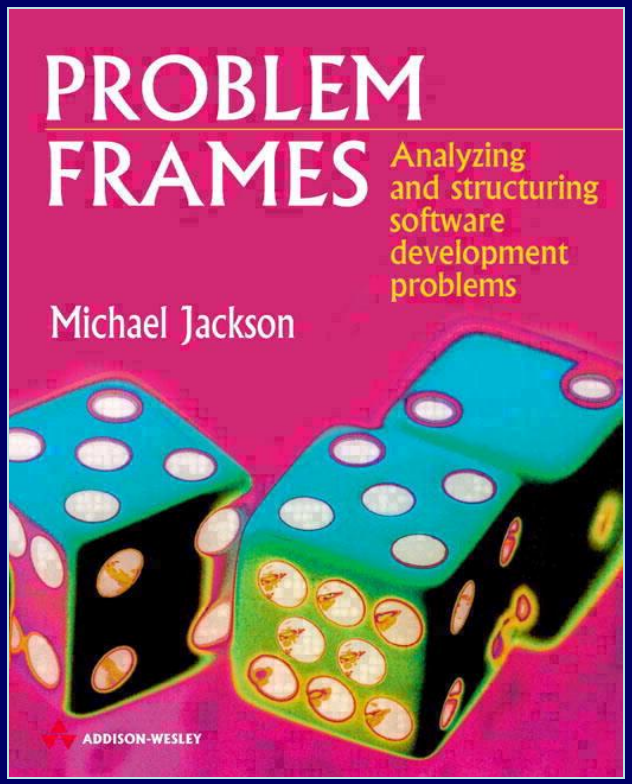




phenomenon (plural: phenomena): An element of what we can observe in the world. Phenomena may be *individuals* or *relations*. Individuals are *entities*, *events*, or *values*. Relations are *roles*, *states*, or *truths*.

individual: An individual is a *phenomenon* that can be named and is distinct from every other individual: for example, the number 17, George III, or Deep Blue's first move against Kasparov.

relationship: A kind of *phenomenon*. An association among two or more *individuals*, for example, *Mother(Lucy, Joe)*. Also, generally, any pattern or structure among phenomena of a *domain*.



Events. An event is an individual happening, taking place at some particular point in time. Each event is *indivisible* and *instantaneous*.

Entities. An entity is an individual that persists over time and can change its properties and states from one point in time to another.

Values. A value is an intangible individual that exists outside time and space, and is not subject to change.

States. A state is a relation among individual entities and values; it can change over time.

Truths. A truth is a relation among individuals that cannot possibly change over time.

Roles. A role is a relation between an event and individuals that participate in it in a particular way.

On the Origin of Species

- Value types differ in the generality and focus of their domain
  - Some are mathematical, e.g., integers
  - Some are programmatic, e.g., strings
  - Some are real world, e.g., ISBNs
- Value types reflect constraints
  - E.g., ISBNs have a well-formedness rule
  - E.g., *int* is a bounded subset of integers

Systems of Values

- Operations, relationships and constraints form systems of values
  - E.g., a point in time is a value, as is the difference between two points in time, but *time point*, *time period* and *time interval* are not equivalent types
  - E.g., distance divided by time yields speed (and displacement divided by time yields velocity)

The Nature of Order

- Some value types have an intrinsic or natural ordering
  - Ordering may be based on magnitude, lexicographical or ordinal criteria
- Be careful about imposing an ordering on unordered types
  - Ordering is not an essential feature for all value types
  - Consider external comparators instead

povo, *sm*.

1. Conjunto de indivíduos que falam (em regra) a mesma lingua, têm costumes e hábitos idênticos, uma história e tradições communs.
2. Os habitantes duma localidade ou região; povoação.
3. V. *povoado*.
4. Multidão.
5. V. *plebe*.

Minidicionário da Língua Portuguesa

Whole Value

Besides using the handful of literal values offered by the language (numbers, strings, true and false) and an even smaller complement of objects normally used as values (date, time, point), you will make and use new objects with this pattern that represent the meaningful quantities of your business. These values will carry whole, useful chunks of information from the user interface to the domain model.

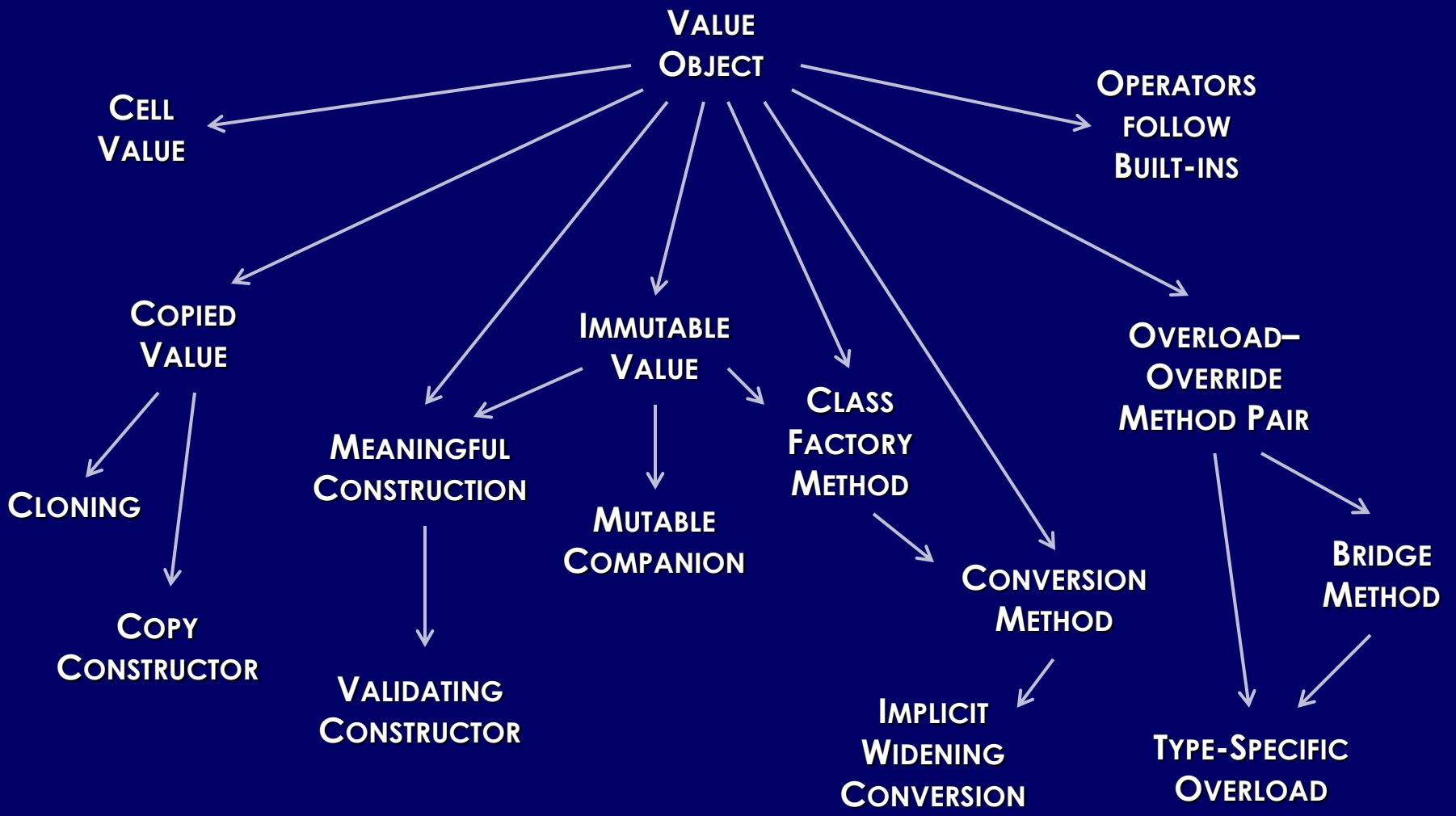
Construct specialized values to quantify your domain model and use these values as the arguments of their messages and as the units of input and output. Make sure these objects capture the whole quantity, with all its implications beyond merely magnitude; but keep them independent of any particular domain. (The word *value* here implies that these objects do not have identity of importance.)

Ward Cunningham
"The CHECKS Pattern Language of Information Integrity"

Values as Objects

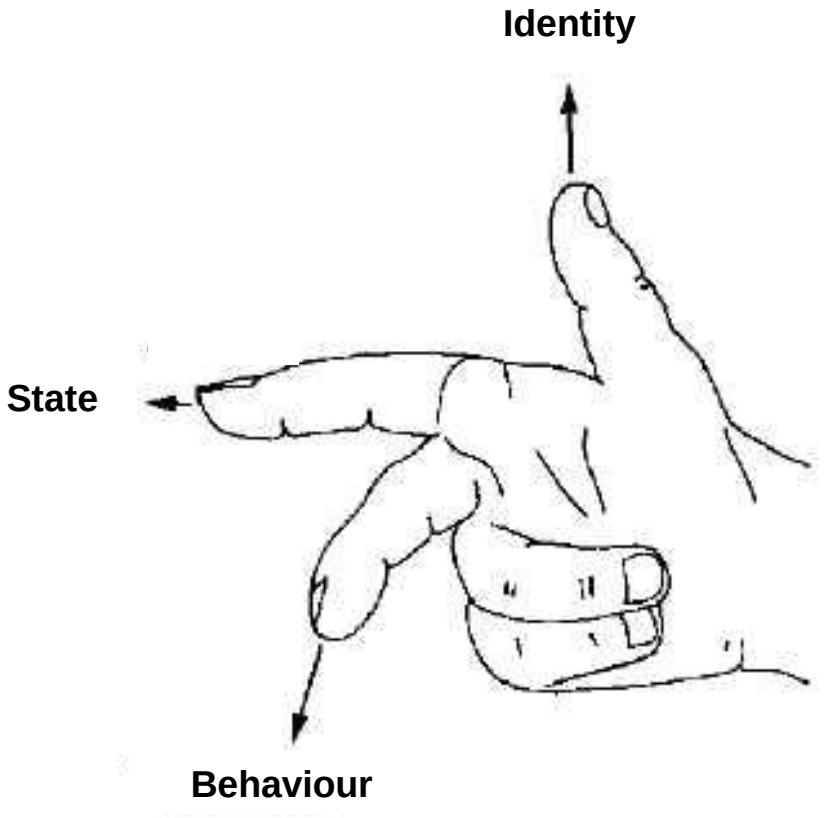
- From a programming perspective, we can model values as objects
  - Hence *value objects* and *value types*
  - Value objects have significant state but insignificant identity
- But there is no dichotomy or conflict between *values* and *objects*
  - A *value object* is a kind or style of *object* that realises a *value*

Patterns of Value



Good Value

- Construction should result in a meaningful and well-formed object
  - Partial initialisation is problematic for objects — especially for value objects
- Also consider the constructor to be transactional
  - I.e., a correct object or an exception
- Respect and enforce state invariant

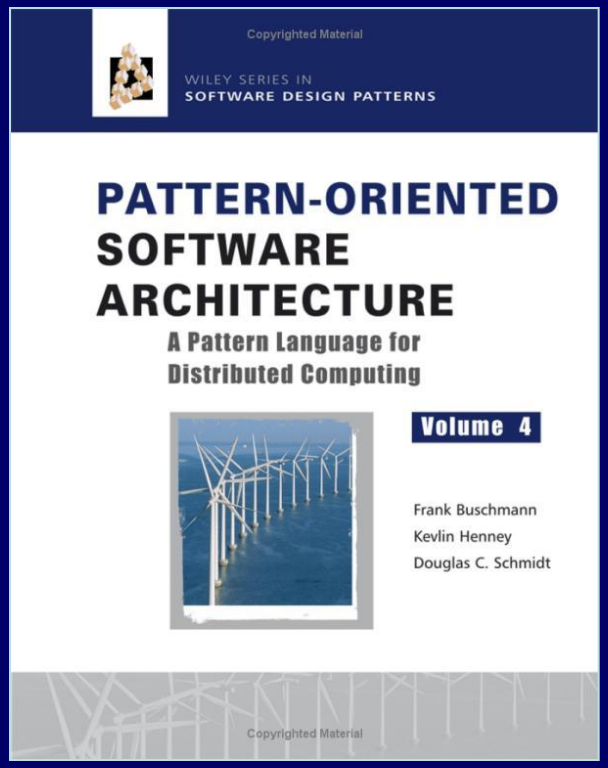

Identity
State
Behaviour


beyond
SHARED PATH
Please consider
other path users
THIS IS A WORK
ALL USERS SHO
CYCLISTS ARE A
No liability will be accep

Referential transparency and **referential opaqueness** are properties of parts of computer programs. An expression is said to be referentially transparent if it can be replaced with its value without changing the program (in other words, yielding a program that has the same effects and output on the same input). The opposite term is referentially opaque.

While in mathematics all function applications are referentially transparent, in programming this is not always the case. The importance of referential transparency is that it allows a programmer (or compiler) to reason about program behavior. This can help in proving correctness, simplifying an algorithm, assisting in modifying code without breaking it, or optimizing code by means of memoization, common subexpression elimination or parallelization.

http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)

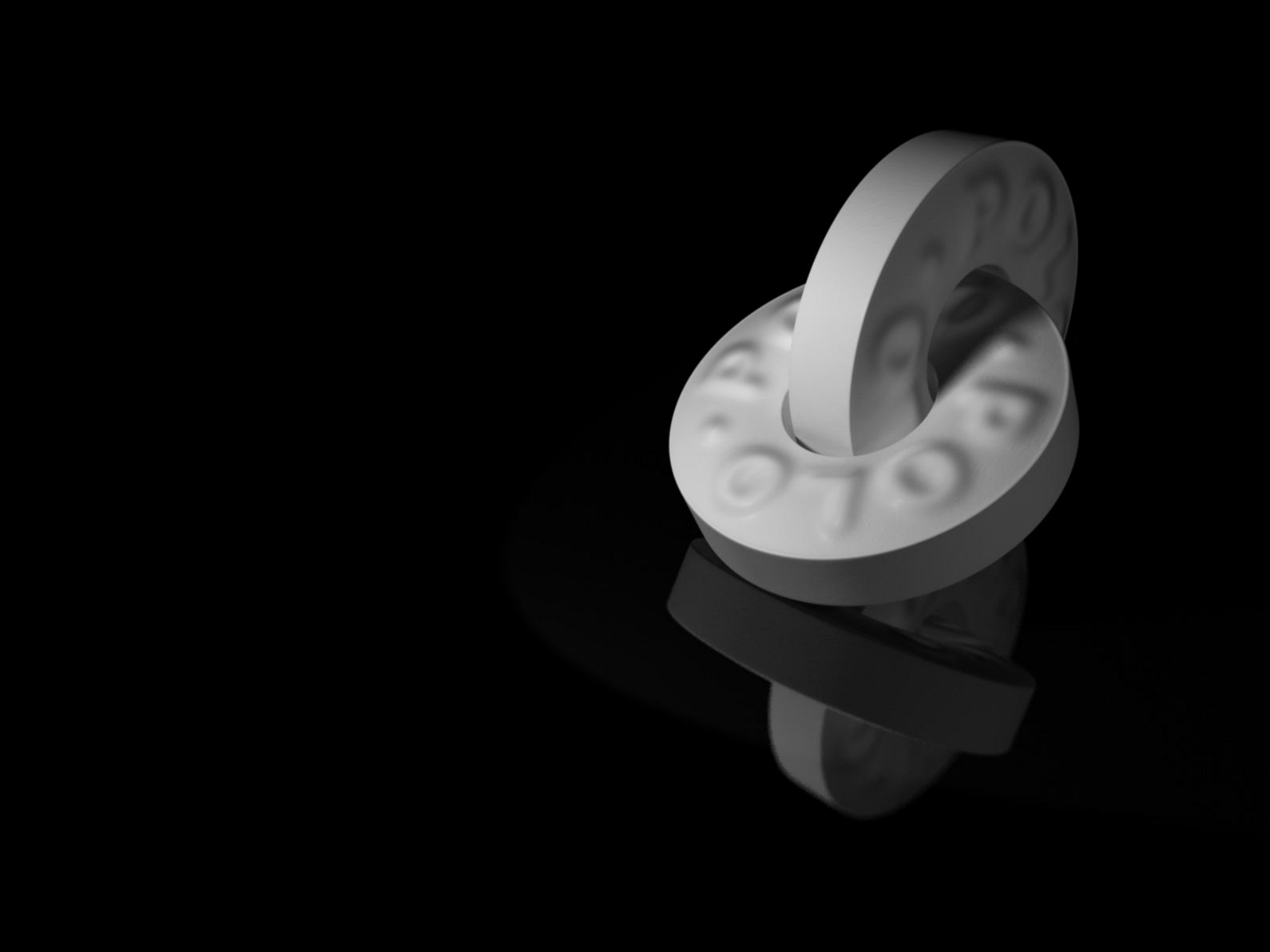


Immutable Value

Define a value object type whose instances are immutable.

Copied Value

Define a value object type whose instances are copyable.



General POLO Anatomy

- Construction...
  - Constructor results in valid object
- Comparison...
  - Equality is a fundamental concept
  - Total ordering may or may not apply
- Classification and conversion...
  - Neither a subclass nor a superclass be
  - May support conversions

Value Object Smells

- Anaemia
  - Little behaviour beyond field access
- Role creep
  - Too much responsibility and external dependencies
- Unwanted affordances
  - Non-meaningful constructor
  - Constructor without useful enforcement or sufficient quality of failure

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getDayInMonth() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setDayInMonth(int newDayInMonth) ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
    private int year, month, dayInMonth;
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    public void setYear(int newYear) ...
    public void setMonth(int newMonth) ...
    public void setWeekInYear(int newWeek) ...
    public void setDayInYear(int newDayInYear) ...
    public void setDayInMonth(int newDayInMonth) ...
    public void setDayInWeek(int newDayInWeek) ...
    ...
    private int daysSinceEpoch;
}
```

When it is not necessary to change, it is necessary not to change.

Lucius Cary

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public int getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public int getDayInWeek() ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int getYear() ...
    public Month getMonth() ...
    public int getWeekInYear() ...
    public int getDayInYear() ...
    public int getDayInMonth() ...
    public DayInWeek getDayInWeek() ...
    ...
}
```

```
public final class Date implements ...
{
    ...
    public int year() ...
    public Month month() ...
    public int weekInYear() ...
    public int dayInYear() ...
    public int dayInMonth() ...
    public DayInWeek dayInWeek() ...
    ...
}
```

Does a value type have a meaningful default value? If so, is it an in-band or out-of-band value?

```
date today;

today.set(2010, 4, 16);

today = date(2010, 4, 16);
```

For copied values, assignment should be considered the preferred way to rebind a value. Try to avoid adding redundant modifiers that duplicate the role of constructors.

Language Defines a Context

- Design is context sensitive
  - And design detail is, therefore, affected by choice of programming language
- Different languages encourage and enable different choices and styles
  - Culture and idioms, features for immutability, transparency of copying, presence of nulls, support for operator overloading, ease of equality, etc.

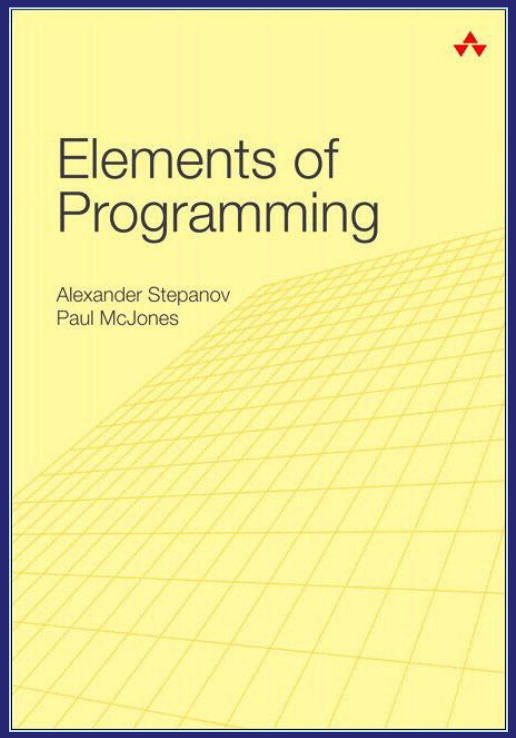
Operator Overloading

- When overloading operators, ensure completeness of operator families
  - It's jarring and awkward when one operator is provided but its logical companions are not
- Be courteous and unsurprising
  - C# demands and simplifies much politeness... C++ does not, but that's no reason to be impolite

```
bool operator<(const Type & lhs, const Type & rhs)
{
    return magnitude, lexicographical or ordinal comparison;
}

bool operator<=(const Type & lhs, const Type & rhs)
{
    return !(rhs < lhs);
}

bool operator>(const Type & lhs, const Type & rhs)
{
    return rhs < lhs;
}

bool operator>=(const Type & lhs, const Type & rhs)
{
    return !(lhs < rhs);
}
```

Two values of a value type are *equal* if and only if they represent the same abstract entity. They are *representationally equal* if and only if their datums are identical sequences of 0s and 1s.

If a value type is uniquely represented, equality implies representational equality.



If we want to emphasize the programmatic aspect of a type that has an associated `operator==`, we say "objects compare equal", but never "objects are equal". [...]

We deliberately avoid equivocal phrases such as "objects are equal", "objects are the same", or "objects are identical".

John Lakos

Normative Language to Describe Value Copy Semantics

http://www.open-std.org/jtc1/sc22/WG21/docs/papers/2007/n2479.pdf

What I propose *must* be "the same" after copy construction is "whatever the associated homogeneous *equality comparison operator defines* (documents) to be the *salient attributes* for that type" – i.e., "the specific attributes that must respectively compare equal in order for the objects as a whole to compare equal." Hence, the (observable) *values* of these *salient attributes*, and not the raw instance state used to represent them, comprise what we call the *value* of the object.

John Lakos

Normative Language to Describe Value Copy Semantics

http://www.open-std.org/jtc1/sc22/WG21/docs/papers/2007/n2479.pdf

```
bool operator==(const Type & lhs, const Type & rhs)
{
    return
        ∀ attribute ∈ SalientAttributesOf(ValueType) •
        lhs.attribute == rhs.attribute;
}

bool operator!=(const Type & lhs, const Type & rhs)
{
    return !(lhs == rhs);
}
```

Reflexivity: I am me.

Symmetry: If you're the same as me, I'm the same as you.

Transitivity: If I'm the same as you, and you're the same as them, then I'm the same as them too.

Consistency: If there's no change, everything's the same as it ever was.

Null inequality: I am not nothing.

No throw: If you call, I won't hang up.

Hash equality: If we're the same, we both share the same magic numbers.

equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

Remarks

The default implementation of Equals supports reference equality for reference types, and bitwise equality for value types. Reference equality means the object references that are compared refer to the same object. Bitwise equality means the objects that are compared have the same binary representation.

Note that a derived type might override the Equals method to implement value equality. Value equality means the compared objects have the same value even though they have different binary representations. For example, consider two Decimal objects that represent the numbers 1.10 and 1.1000. The Decimal objects do not have bitwise equality because they have different binary representations to account for the different number of trailing zeroes. However, the objects have value equality because the numbers 1.10 and 1.1000 are considered equal for comparison purposes since the trailing zeroes are insignificant.

Notes to Implementers:
This method can be overridden by a derived class. For example, many of the base data types return true if both objects represent the same value; otherwise, false.

This method only compares primitives and objects. It must be overridden to compare more complex structures, such as arrays of objects.

The following statements must be true for all implementations of the Equals method. In the list, x, y, and z represent object references that are not a null reference (Nothing in Visual Basic).

- x.Equals(x) returns true, except in cases that involve floating-point types. See IEC 60559:1989, Binary Floating-point Arithmetic for Microprocessor Systems.
- x.Equals(y) returns the same value as y.Equals(x).
- x.Equals(y) returns true if both x and y are NaN.
- (x.Equals(y) && y.Equals(z)) returns true if and only if x.Equals(z) returns true.
- Successive calls to x.Equals(y) return the same value as long as the objects referenced by x and y are not modified.
- x.Equals(a null reference (Nothing in Visual Basic)) returns false.

See GetHashCode for additional required behaviors pertaining to the Equals method.

Implementations of Equals must not throw exceptions.

For some kinds of objects, it is desirable to have Equals test for value equality instead of referential equality. Such implementations of Equals return true if the two objects have the same "value", even if they are not the same instance. The type's implementer decides what constitutes an object's "value", but it is typically some or all the data stored in the instance variables of the object. For example, the value of a String is based on the characters of the string; the Equals method of the String class returns true for any two string instances that contain exactly the same characters in the same order.

Types that implement IComparable must override Equals.

Types that override Equals must also override GetHashCode; otherwise, Hashtable might not work correctly.

If your programming language supports operator overloading and if you choose to overload the equality operator for a given type, that type must override the Equals method. Such implementations of the Equals method must return the same results as the equality operator. Following this guideline will help ensure that class library code using Equals (such as ArrayList and Hashtable) behaves in a manner that is consistent with the way the equality operator is used by application code.

Guidelines for Overloadin...

http://msdn.microsoft.com/en-us/library/ms173147(VS.80).aspx

Any derived class that can call **Equals** on the base class should do so before finishing its comparison. In the following example, **Equals** calls the base class **Equals**, which checks for a null parameter and compares the type of the parameter with the type of the derived class. That leaves the implementation of **Equals** on the derived class the task of checking the new data field declared on the derived class:

Don't

VB | C# | C++ | F# | JScript

Copy

```
class ThreeDPoint : TwoDPoint
{
    public readonly int z;

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        this.z = z;
    }

    public override bool Equals(System.Object obj)
    {
        // If parameter cannot be cast to ThreeDPoint return false:
        ThreeDPoint p = obj as ThreeDPoint;
        if ((object)p == null)
        {
            return false;
        }

        // Return true if the fields match:
        return base.Equals(obj) && z == p.z;
    }

    public bool Equals(ThreeDPoint p)
    {
        // Return true if the fields match:
        return base.Equals((TwoDPoint)p) && z == p.z;
    }

    public override int GetHashCode()
    {
        return base.GetHashCode() ^ z;
    }
}
```

A Note on Unit Testing

- The contract for equality does not automatically translate to test cases
  - Recast the relationships into a more propositional form
- It's OK to use value objects directly in other tests
  - Don't mock out values — "Every time a mock returns a mock, a fairy dies"

```
[Test]
public void IdenticallyConstructedValuesCompareEqual()
...
[Test]
public void DifferentlyConstructedValuesCompareUnequal()
...
[Test]
public void ValuesCompareUnequalToNull()
...
[Test]
public void IdenticallyConstructedValuesHaveEqualHashCodes()
...
```

```
[Test]
public void Identically_constructed_values_compare_equal()
...
[Test]
public void Differently_constructed_values_compare_unequal()
...
[Test]
public void Values_compare_unequal_to_null()
...
[Test]
public void Identically_constructed_values_have_equal_hash_codes()
...
```

"Write that down," the King said to the jury, and the jury eagerly wrote down all three dates on their slates, and then added them up, and reduced the answer to shillings and pence.

Lewis Carroll

Alice's Adventures in Wonderland