

Value Objects

Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

ADAP C06

Licensed under [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/)

Agenda

1. Values vs. objects
2. Implementing value types
3. QuantityUnit value type
4. Value type constructors
5. Value types in practice

1. Values vs. Objects

Values

- Are timeless abstractions
 - No life-cycle, no birth or death, no change
 - No identity, cannot be counted, there is only one “copy”
- Consequences for programming
 - Often implemented as immutable objects
 - Program state changes by assigning values to fields
- Values are instances of value types
 - Also often called “data” and “data types”
 - We avoid the potential confusion

Objects

- Are virtual or physical entities from the modeled world
 - Exist in time, have a life-cycle
 - Can be created, changed, shared, destroyed
 - Have identity independent of internal structure
- Consequences for programming
 - Can be implemented as traditional classes with mutable state
 - Leads to side-effects a.k.a. aliasing and source of bugs
- Objects are instances of object types (classes)

Examples of Value Types

- So-called “primitive data types”
 - Numbers, strings, characters, ...
- Common general value types
 - Names, coordinates, postal codes, ...
- Domain-specific value types
 - SI unit (quantity units), ranges, restrictions, ...
 - Currency, monetary amount, interest rate, stock ticker symbol, ...
 - Protocol names, URLs, HTTP return codes, ...
 - ...

Value and Object Representations

Shared Inter-subjective Immaterial Reality

2 : Integer

EUR 42 : Money

sa12 : SavingsAccount

Shared Material “Physical” Reality

“II” on paper

“EUR 42” on statement

sa12 as booklet

“zwei” on paper

€42 as cash

sa12 in audit trail

Software System

10

“2”

2.0

42

€

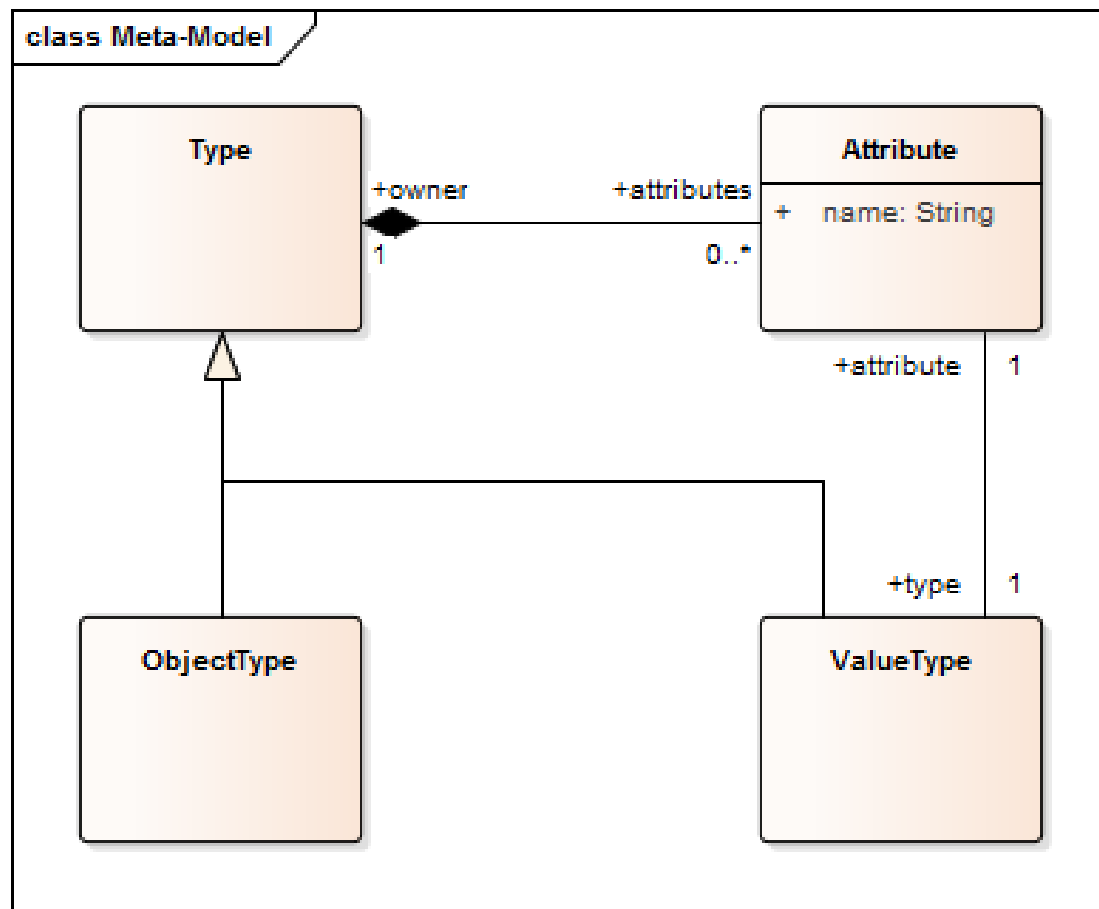
42.0

sa12 : SA

sa12 : SA

Objects and Values

- A software system consists of both objects and values
 - Object attributes have values
- The system is an object graph with values as leaves



Benefits of Domain-Specific Value Types

- Brings program closer to problem domain
- Restrains a major source of possible bugs (aliasing)
- May enhance system performance (see implementation)

Quiz: Modeling PostalAddress

- **How would you model a PostalAddress class?**
 - As a value type
 - As an object type
 - As something else

Answer: Modeling PostalAddress

- **How would you model a PostalAddress class?**
 - As a value type
 - **Conceptually:** Yes. A postal address does not have a life-cycle and does not change.
 - **Pragmatically:** No. May be too heavyweight, with little sharing possible.
 - As an object type
 - **Conceptually no, pragmatically yes. See above.**
 - As something else
 - **Tertium non datur.**

Object Identifiers / References

- Plain main memory reference
- Handle (specialized pointers)
- External object identifiers
- Primary key to relation

2. Implementing Value Types

Implementation 1 / 4: General Semantics

- Implement java.lang.Object equality contract correctly

```
public boolean equals(Object o) {
    if ((o == null) || !(o instanceof Name)) return false;

    Name n = (Name) o;
    int noComponents = getNoComponents();
    if (n.getNoComponents() != noComponents) return false;

    for (int i = 0; i < noComponents; i++) {
        if (!getComponent(i).equals(n.getComponent(i))) return false;
    }

    return true;
}
```

```
public int hashCode() {
    return asString().hashCode();
}
```

Implementation 2 / 4: Immutability

- Value types as classes defining immutable objects
 - Do not change the state of the object; rather return a new one
 - Affects the interface; no mutation methods of return type void
 - Use Java's final fields to ensure immutability
 - Adjust client code to accept new object

```
public Name remove(int i) {  
    assertIsValidIndex(i);  
    ...  
    Name result = doRemove(i);  
    ...  
}
```

```
protected Name doRemove(int index) {  
    int newSize = getNoComponents() - 1;  
    String[] newComponents = new String[newSize];  
    ... // copy components skipping component at index  
    return getName(newComponents);  
}
```

Implementation 3 / 4: Sharing

- Value types as classes defining shared objects

```
public Name getName(String[] components) {  
    return getStringArrayName(component);  
}
```

```
public StringArrayName getStringArrayName(String[] components) {  
    String nameString = NameHelper.asNameString(components);  
    StringArrayName result = allStringArrayNames.get(nameString);  
    if (result == null) {  
        synchronized (this) {  
            result = allStringArrayNames.get(nameString);  
            if (result == null) {  
                result = new StringArrayName(components);  
                allStringArrayNames.put(nameString, result);  
            }  
        }  
    }  
    return result;  
}
```


Benefits of Sharing Value Objects

- Trivial equality contract implementation
- More difficult if you have different implementation classes

```
public boolean equals(Object o) {  
    return this == o;  
}
```

```
public int hashCode() {  
    return super.hashCode();  
}
```

Implementation 4 / 4: Handle / Body Idiom

- Handle / Body Idiom [C95]
 - Pass around only the handle, which holds the body
 - Forward all method calls from handle to body
- Copy-on-write (mutation method call)
 - Upon mutation method call to handle, copy body
 - This way, the client gets isolated from source context
- Benefits of handle / body idiom
 - Protects client from aliasing effects
 - Minimizes memory consumption

Implementation Benefits of Value Objects

- Immutable objects are
 - Safe for concurrency
 - Perform well (no need for synchronization)
- Shared objects
 - Make equality easy to implement
 - Minimize memory consumption
- No identity of value objects allows for free copying
 - Database benefit: No need for separate database table
 - Serialization benefit: Value object can be serialized in-line
 - Distributed systems benefit: No cross-process reference

Quiz: Implementing Base Contracts

- How to implement on `java.lang.Object`
 - For either a value or an object type
 - These comparison methods ...
 - `boolean isSame(Object o)`
 - `boolean equals(Object o)`
 - These creation methods ...
 - `Object clone()`
 - Constructor
 - These other methods ...
 - `int hashCode()`
 - `getId()`

Answer: Implementing Base Contracts

- Value types

- These comparison methods ...
 - boolean isSame(Object o)
 - **N/A**
 - boolean equals(Object o)
 - **By attribute comparison**
- These creation methods ...
 - Object clone()
 - **return this;**
 - Constructor
 - **Is hidden when sharing values**
- These other methods ...
 - int hashCode()
 - **Calculate hash function**
 - getId()
 - **return ids.get(hashCode())**

- Object types

- These comparison methods ...
 - boolean isSame(Object o)
 - **return this == o;**
 - boolean equals(Object o)
 - **By attribute comparison**
- These creation methods ...
 - Object clone()
 - **Create shallow clone**
 - Constructor
 - **Whatever**
- These other methods ...
 - int hashCode()
 - **return super.hashCode()**
 - getId()
 - **Safe version of return (int) this**

Value Types in Java

- Not yet, but ... probably never. Oh well, updated in 2019, but nothing final yet



JEP 169: Value Objects

[OpenJDK FAQ](#)
[Installing](#)
[Contributing](#)
[Sponsoring](#)
[Developers' Guide](#)

[Mailing lists](#)
[IRC](#) · [Wiki](#)

[Bylaws](#) · [Census](#)
[Legal](#)

JEP Process

Source code

[Mercurial](#)
[Bundles \(6\)](#)

Groups

[\(overview\)](#)
[2D Graphics](#)
[Adoption](#)
[AWT](#)
[Build](#)
[Compiler](#)
[Conformance](#)
[Core Libraries](#)
[Governing Board](#)
[HotSpot](#)
[Internationalization](#)
[JMX](#)
[Members](#)
[Networking](#)

<i>Owner</i>	John Rose
<i>Created</i>	2012/10/22 20:00
<i>Updated</i>	2014/09/23 18:58
<i>Type</i>	Feature
<i>Status</i>	Draft
<i>Component</i>	hotspot
<i>Scope</i>	SE
<i>Discussion</i>	mlvm dash dev at openjdk dot java dot net
<i>Effort</i>	L
<i>Duration</i>	L
<i>Priority</i>	4
<i>Issue</i>	8046159

Summary

Provide JVM infrastructure for working with immutable and reference-free objects, in support of efficient by-value computation with non-primitive types.

Goals

- Support user-defined and library-defined abstract data types with performance profiles similar to Java primitive types.

3. QuantityUnit Value Type

Design Exercise

- Design a function that accepts a distance and a speed as input and returns the time it takes to go that distance at that speed

Missing Information

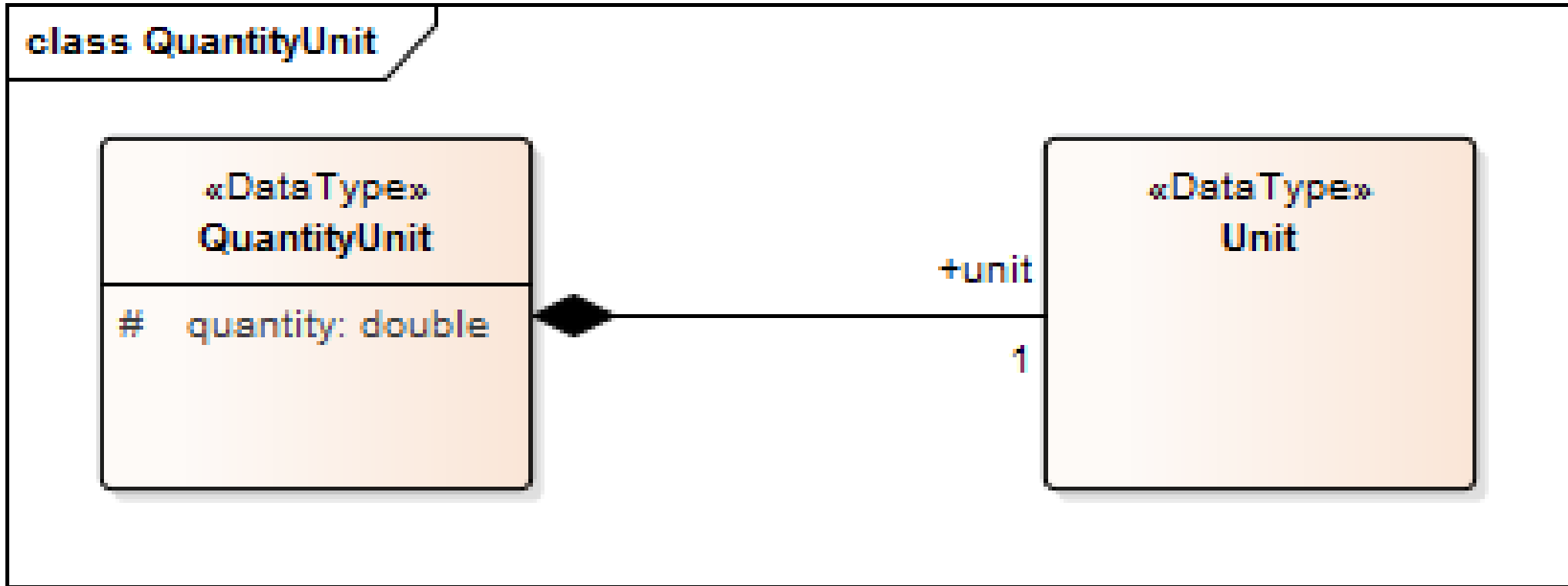
- Functional properties
 - Precision of calculation? (Assume double)
 - What types of units? (Assume metric system)
 - Handling of decimal multiples (Assume no)
 - ...
- Non-functional properties
 - Speed of calculation? (Assume as fast as possible)
 - Concurrent computations? (Assume yes)
 - Behavior in boundary cases? (Assume ignore)
 - ...

Solution in Pseudo Code

```
function calculate_duration
  in: speed
  in: distance
  out: duration

begin
  duration = distance / speed
end
```

QuantityUnit



Base Units (of the Metric System)

Quantity	Base Unit	Symbol
Length	meter	m
Mass	kilogram	kg
Time	second	s
Electric Current	ampere	A
Thermodynamic Temperature	kelvin	K
Luminous Intensity	candela	cd
Amount of Substance	mole	mol

State Pattern vs. enum

```
public enum BaseUnit {  
    m(0),  
    kg(1),  
    s(2),  
    A(3),  
    K(4),  
    cd(5),  
    mol(6);  
}
```

Unit (of Measure)

```
public class Unit {  
    // example: m/s = { 1.0, 0, -1.0, 0, 0, 0, 0 }  
    protected double exponents[7];  
    ...  
}
```

Quantities with Units

```
public class QuantityUnit {  
    protected double quantity;  
    protected Unit unit;  
    ...  
}
```

4. Value Type Constructors

Value Type Constructors

- Arrays
- Enumerations
- Parameterized types
 - Quantity units (SI units)
 - Ranges and range bounds
 - Expressions, restrictions
 - ...

Enums as Value Type Constructors

- Enumerations provide shared values
 - Constructors can only be private
 - Fields can be mutable, however

Param. Types as Value Type Constructors

```
public class RangeRestriction<T extends Comparable<T>>
    extends Restriction<T> {

    protected Range<T> range;

    public RangeRestriction(T lowerBound, T upperBound) {
        this(new Range<T>(lowerBound, upperBound));
    }

    public RangeRestriction(Range<T> range) {
        this.range = range;
    }

    @Override
    public boolean isSatisfiedBy(T value) {
        return range.includes(value);
    }
}
```

5. Value Types in Practice

Reality Check [B+97]

- Large financial system (1997)
 - More than 2500 (regular) C++ classes
- Utilization of value object (classes)
 - About 50 unique value types (implemented as classes)
 - About 20 unique value type constructors
 - More than 200 code (enum) like value types

The JValue Value Object Framework

- JValue, originally a framework for value objects in Java
 - Since 1998 but in its third (modern Java) incarnation
 - Only recently on <http://github.com/jvalue/value-objects>
 - Also new version for TypeScript in development
- **Contributions are welcome! Final theses possible as well.**

Review / Summary of Session

- Values vs. objects
- Implementing value types
- QuantityUnit value type
- Value type constructors
- Value types in practice

Thank you! Questions?

dirk.riehle@fau.de – <https://oss.cs.fau.de>

dirk@riehle.org – <https://dirkriehle.com> – [@dirkriehle](#)

Credits and License

- Original version
 - © 2012-2021 Dirk Riehle, some rights reserved
 - Licensed under Creative Commons Attribution 4.0 International License
- Contributions
 - None yet