The Story of a Framework

Dirk Riehle

SKYVA International
www.skyva.com, www.skyva.de, www.skyva.ch

dirk@riehle.org, www.riehle.org

Minneapolis, MI: OOPSLA 2000
Last updated August 9, 2000.

Purpose/Approach of Tutorial

- Purpose: Tutorial
  - to review and extend our vocabulary for designing and implementing object-oriented frameworks so that we become more effective in developing them
- Approach: Tutorial
  - shows the design and implementation of a framework using a running and evolving real-world example

Tutorial: Structure

- Part I: Programming (30 min.)
- Part II: Design (90 min.)
- Part III: Frameworks (60 min.)

Part I: Programming

Dirk Riehle

SKYVA International
www.skyva.com, www.skyva.de, www.skyva.ch

dirk@riehle.org, www.riehle.org

Part I: Topics

- Classes and interfaces
- Inheritance and code reuse
- Method types
- Unit testing

Part I: Outline

- Step 1: A simple class
- Step 2: An interface and two classes
- Step 3: A class hierarchy with a reusable superclass

Step 1: Topics

- GenericName class
  - a simple class for handling generic names
- Method types
  - query methods
  - mutation methods
  - helper methods
- Unit testing
  - simple unit testing of individual classes using JUnit

Homogeneous Generic Names

- Definition: Homogeneous Generic Name
  - a name that consists of an arbitrary number of components
  - homogeneous, because components are strings and not interpreted further
  - usually has a string representation with designated delimiter char
- Examples
  - "/usr/local/bin", "~riehle/java", "C:\WINNT\system"
  - "java.lang.util", "org.jvalue.Name"
- Counterexamples (heterogeneous names)
  - URL's like "http://developer.java.sun.com/developer/TechTips/index.html"

GenericName Class

- The GenericName class provides one field
  - a Vector, holding the name components
- It defines default delimiter and escape characters

GenericName: Fields

```
public class GenericName {
    public static final char DELIMITER_CHAR= '#';
    public static final String DELIMITER_STRING= "#";
    public static final char ESCAPE_CHAR= '\\';
    public static final String ESCAPE_STRING= "\\";

    private Vector fComponents;
}
```

GenericName: Behavior

- GenericName provides methods for
  - creating a name
  - querying, comparing, and representing the name
  - changing the name

GenericName: Methods

```
public class GenericName {
    public GenericName(String name) { ... }
    public GenericName(Vector components) { ... }

    public String asString() { ... }
    public String asDataString() { ... }

    public boolean isEmpty() { ... }

    public String getFirstComponent() { ... }
    public void setFirstComponent(String nc) { ... }
    public String getLastComponent() { ... }
    public void setLastComponent(String nc) { ... }
    public String getComponent(int index) { ... ]
    public void setComponent(int index, String nc) { ... }

    ...
}
```

Method Types

- Definition: Method Type
  - describes purpose of a method
  - describes what a client is to expect from a method
- Three main categories
  - query methods
  - mutation methods
  - helper methods

Query Methods

- Definition: Query Method (a.k.a. Assessing Method)
  - returns information about this object
  - does not change the state of this object
- Method types in this category
  - get methods (getters); prefix: get (if any)
  - boolean query methods; prefix: is, has, may, or can
  - comparison methods; prefix: is
  - conversion methods; prefix: as, to
- Standard Java examples
  - Object::hashCode(), Object::getClass(), Object::equals()

Mutation Methods

- Definition: Mutation Method
  - changes the state of this object
  - usually does not return any information
- Method types in this category
  - set methods (setter); prefix: set (if any)
  - command methods; prefix: handle, execute
  - initialization methods; prefix: init, initialize
- Standard Java examples
  - Vector::addElement(), Vector::insertElementAt(), Vector::removeElementAt()
  - JComponent::repaint(), JComponent::setVisible(), JComponent::add*Listener

Helper Methods

- Definition: Helper Method
  - performs some support task for this object
  - does not change state of this object, but only of argument objects
  - frequently a static method put into a separate helper class
- Method types in this category
  - general helper methods; prefix: no specific
  - factory methods; prefix: new, create, make
  - assertion methods; prefix: assert, ensure, check
- Standard Java examples
  - String::valueOf()

Unit Testing 1/2

- What is unit testing?
  - unit testing tests quality of an implementation unit
  - unit may be individual class, component, package
  - unit testing is a form of regression testing
- Goals of unit testing
  - gain confidence in quality of implementation
  - increase design through second streamlined view
  - communicate expected behavior (specification)
  - discover and isolate bugs quickly and early on
  - automate testing itself and make it repeatable

Unit Testing 2/2

- Process of unit testing
  - write regular code and test code in parallel
  - evolve regular code and test code in parallel
  - run tests after every non-trivial change

GenericName: Test Case Setup

```
import junit.framework.*;
import org.jvalue.value.name.*;

public class GenericNameTest extends TestCase {
   protected GenericName fGN0; // ("")
   protected GenericName fGN1; // ("##Eliot")
   protected GenericName fGN2; // ("T.##Eliot")
   protected GenericName fGN3; // ("T.#S.#Eliot")

   protected void setUp() {
        fGN0= createGenericName("");
        fGN1= createGenericName("##Eliot");
        fGN2= createGenericName("T.##Eliot");
        fGN3= createGenericName("T.#S.#Eliot");
   }

   ...
}
```


GenericName: Example Test Cases

```
public void testAsString() {
    assertEquals("", fGN0.asString());
    assertEquals("Eliot", fGN1.asString());
}

public void testAsDataString() {
    assertEquals("##", fGN0.asDataString());
    assertEquals("##Eliot", fGN1.asDataString());
}

public void testSetLastComponent() {
    fGN0.setLastComponent("Thoreau");
    assertEquals(createGenericName("##Thoreau"), fGN0);
    fGN1.setLastComponent("Thoreau");
    assertEquals(createGenericName("##Thoreau"), fGN1);
}
```

GenericName: Example Test Run

- The output is simple and reassuring

```
java org.jvalue.value.name.test.GenericNameTest

..........
Time: 0.525

OK (11 tests)
```


Step 2: Topics

- GenericName interface
  - an interface for generic name objects
  - two implementation classes StringName and VectorName
- Interface concept
  - separating interfaces from implementations
  - interface design (method types, design by contract)
- Unit Testing
  - testing an interface
  - test case and test setup inheritance

Implementing Generic Names

- StringName
  - represents a generic name as a single string
  - example: "java.lang.Object" represented as "java#lang#Object"
  - advantage: efficient memory use
  - disadvantage: slow access to individual components
- VectorName
  - represents a generic string as a Vector of strings
  - example: "java.lang.Object" represented as { "java", "lang", "Object" }
  - advantage: fast access to individual components
  - disadvantage: inefficient memory use
- Which class to choose? What if you need both?

GenericName Interface

- GenericName interface
  - defines the generic name abstraction
  - hides different implementation classes behind shared definition
- StringName and VectorName
  - implement the GenericName interface
  - provide behavior within the limits of the GenericName definition
- How to abstract from StringName and VectorName?
- How to make GenericName concrete as implementations?

GenericName: Interface

```
public interface GenericName {
    public String asString();
    public String[] asStringArray();
    public boolean isEmpty();

    public String getComponent(int i);
    public String getFirstComponent();
    public String getLastComponent();
    public NameEnumeration getComponents();
    public int getNoComponents();

    public void append(String component);
    public void insert(int index, String component);
    public void prepend(String component);
    public void remove(int index);

    ...
}
```


StringName: Fields and Methods

```
public class StringName implements GenericName {
    protected String fName;
    protected int fNoComponents;

    // implementation of GenericName interface
    public int getNoComponents() {
        if (fNoComponents == -1) {
            if (!isEmpty()) {
                fNoComponents= getNoChars(fName, getDelChar(), getEscChar())+1;
            } else {
                fNoComponents= 0;
            }
        }
        return fNoComponents;
    }
    ...

}
```

VectorName: Fields and Methods

```
public class VectorName implements GenericName {
    protected Vector fComponents;

    // implementation of GenericName interface
    public int getNoComponents() {
        return fComponents.size();
    }

    ...
}
```


Interfaces

- Definition: Interface
  - a contract between a client and a service object that provides the interface
  - acts as the coupling link between anonymous client and service object
- Contracts are mutual: two perspectives result
  - client perspective: external view
  - implementation perspective: internal view
- Java provides explicit interface concept
  - statically type-checked and guaranteed by the compiler
  - interface specification only on the level of method signatures
  - please note: used for many purposes, we focus on one use

Client and Implementation Perspectives

- Clients ...
  - make use of the interface only
  - must observe contract specified by interface
  - must stick to the interface and ignore implementations
  - are unknown to implementations
- Implementations ...
  - provide behavior defined by the interface
  - may only vary within the limits set up by the interface
  - are unknown to clients that work with interfaces only

Technical Benefits of Interfaces

- Implementations can be evolved without affecting clients
  - improved behavior (smaller, faster)
  - bug fixes behind the scenes
- New implementations can be introduced easily
  - an interface defines no superfluous baggage
  - supports incremental system evolution/delivery
- Implementations can be selected dynamically
  - client gets best service available upon request
  - allows for dynamic loading of services

Organizational Benefits of Interfaces

- Supports teamwork
  - systems are subdivided along interfaces into subsystems
  - teams decouple their work through interfaces
- Supports incremental delivery
  - provide standard implementations first
  - later introduce implementations with higher quality-of-service

Disadvantages of Interfaces

- Increased Complexity
- More typing

Use of the Word "Interface"

- Explicit interfaces (a Java interface)
  - defined using keyword interface
  - example: GenericName
- Implicit interfaces (interface of a Java class)
  - defined by the available methods of a class
  - usually refered to as "class interface"
  - example: interface of GenericName class

Interface Design

- Design by Contract
  - define state space
  - define possible transitions in state space
  - define expected client behavior
- Method design
  - make a method serve one and only one purpose
  - define preconditions/exceptions

Method Design

- Methods serve exactly one purpose
  - query methods
  - mutation methods
  - helper methods
- Mixing purposes spells trouble
  - unless you follow well-established idioms/patterns
  - unless your interface is very simple
  - unless you want to ensure atomicity of operation

StringName: getComponent(int index)

```
public String getComponent(int i) {
    assertIsValidIndex(i);
    return doGetComponent(i);
}

protected String doGetComponent(int i) {
    int startPos= getStartPosOfComponent(i);
    int endPos= getEndPosOfComponent(i);
    String component= fName.substring(startPos, endPos);
    return asUnmaskedString(component);
}
```

VectorName: getComponent(int index)

```
public String getComponent(int i) {
    assertIsValidIndex(i);
    return doGetComponent(i);
}

protected String doGetComponent(int i) {
    return (String) fComponents.elementAt(i);
}
```


Design by Primitives

- Definition: Primitive Method
  - do exactly one specific "primitive" thing
  - encapsulate access to object's state
- Primitive methods ...
  - can be readily composed
  - are typically protected and not accessed from the outside
  - rely on context to make sure preconditions of operation are met
- Definition: Design by Primitives
  - base a class implementation on primitive methods
  - compose client-oriented methods using primitive methods

StringName: Primitive Methods

```
public StringName implements GenericName {
    ...

    protected String doGetComponent(int i);
    protected void doSetComponent(int i, String c);
    protected void doInsert(int i, String c);
    protected void doRemove(int i);
    ...
}
```


Unit Testing of Interfaces

- Testing interfaces or implementations?
  - the quality of an implementation
  - the compliance of an implementation to an interface
- How to test implementations behind interfaces
  - write test code against the interface
  - supply objects of different implementations
  - set up test cases in which different implementations are matched

GenericName: Test Case Setup

```
public abstract class AbstractGenericNameTest extends TestCase {
    protected GenericName fN0; // ("")
    protected GenericName fN1; // ("org", '.')
    protected GenericName fN2; // ("lang.Object", '.')
    protected GenericName fN3; // ("java.lang.Object", '.')

    protected void setUp() {
        fN0= createGenericName("");
        fN1= createGenericName("org", '.');
        fN2= createGenericName("lang.Object", '.');
        fN3= createGenericName("java.lang.Object", '.');
    }

    protected abstract GenericName createGenericName(String name);
    protected abstract GenericName createGenericName(String name, char delChar);
    protected abstract GenericName createGenericName(Vector components);
    ...

}
```


GenericName: Example Test Cases

```
// from AbstractGenericNameTest
public void testAsString() {
    assertEquals("", fN0.asString());
    assertEquals("Object", fN1.asString());
    assertEquals("lang Object", fN2.asString());
    assertEquals("java lang Object", fN3.asString());
}


// from AbstractGenericNameTest
public void testIsEqual() {
    assertEquals(fN0, createGenericName(""));
    assertEquals(fN1, createGenericName("Object"));
    assertEquals(fN2, createGenericName("lang#Object"));
    assertEquals(fN3, createGenericName("java#lang#Object"));
}
```

Step 3: Topics

- AbstractGenericName superclass
  - sharing code through an abstract superclass
  - providing convenient standard implementations
  - moving common helper methods into helper classes
- Abstract superclasses
  - inheritance interface
  - design by primitives
  - narrow inheritance interface principle

Example Method Implementations

```
public String getFirstComponent() {
    return getComponent(0);
}

public void append(String component) {
    insert(getNoComponents(), component);
}

public String asString(char delimiter, char escape) {
    if (isEmpty()) { return ""; }
    StringBuffer sb= new StringBuffer();
    for (int i= 0; i<getNoComponents(); i++) {
        sb.append(doGetComponent(i));
        sb.append(String.valueOf(delimiter));
    }
    sb.setLength(sb.length()-1);
    return sb.toString();
}
```

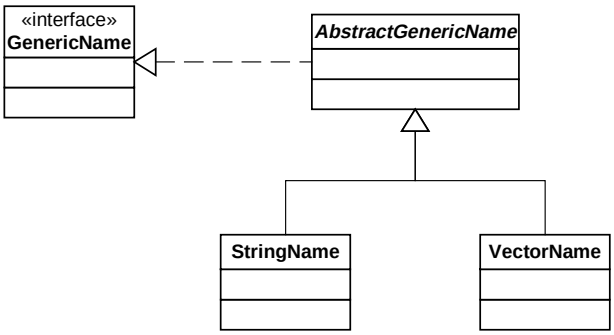
Analysis of Method Implementations

- Similarities between StringName and VectorName methods
  - many similar method implementations
  - all method implementations based on small set of primitive methods
- Only these primitive methods use implementation state

AbstractGenericName Superclass

- AbstractGenericName superclass
  - implements GenericName interface
  - implements all methods except for primitive ones
  - declares primitive methods as abstract
- StringName and VectorName
  - are subclasses of AbstractGenericName
  - implement primitive methods

GenericName: Class Model



AbstractGenericName: Class Definition

```
public abstract class AbstractGenericName implements GenericName {
    public String asString() { ... }
    public int getNoComponents() { ... };
    public String getComponent(int i) { ... };
    public void setComponent(int i, String c) { ... };
    public NameEnumeration getComponents() { ... }
    public void append(String component) { ... }
    public void prepend(String component) { ... }
    public void insert(int i, String c) { ... };
    public void remove(int index) { ... };
    ...
}
```

StringName: doGetComponent(int index)

```
class AbstractGenericName implements GenericName {
    public String getComponent(int i) {
        assertIsValidIndex(i);
        return doGetComponent(i);
    }
    ...

    protected abstract String doGetComponent(int i);
    ...
}

class StringName extends AbstractGenericName {
    protected String doGetComponent(int i) {
        int startPos= getStartPosOfComponent(i);
        int endPos= getEndPosOfComponent(i);
        String component= fName.substring(startPos, endPos);
        return asUnmaskedString(component);
    }
    ...
}
```


VectorName: doGetComponent(int index)

```
class AbstractGenericName implements GenericName {
    public String getComponent(int i) {
        assertIsValidIndex(i);
        return doGetComponent(i);
    }
    ...

    protected abstract String doGetComponent(int i);
    ...
}

class VectorName extends GenericName {
    protected String doGetComponent(int i) {
        return (String) fComponents.elementAt(i);
    }
    ...
}
```

Abstract Superclasses

- Definition: Abstract Class
  - an abstract class cannot be instantiated (= is abstract)
  - in Java, abstract classes are marked by keyword abstract
  - an abstract class is always meant to be a superclass
- Purpose of abstract superclasses
  - to be reused by subclasses
  - to provide common code for subclasses
  - to define a skeleton of control flow for subclasses

Inheritance Interface

- Definition: Inheritance Interface
  - set of abstract methods defined by an abstract superclass
  - inheritance interface serves abstract superclass
  - to be implemented by concrete subclasses
- Design of the inheritance interface
  - should consist of simple methods (design by primitives)
  - should be minimal (narrow inheritance interface principle)

AbstractGenericName: Inheritance Interface

```
public abstract class AbstractGenericName implements GenericName {
    ...

    protected abstract String doGetComponent(int i);
    protected abstract void doGetComponent(int i, String c);
    protected abstract void doInsert(int i, String c);
    protected abstract void doRemove(int i);
    protected abstract GenericName createGenericName(String n);
    ...
}
```


Default vs. Specific Implementation

```
// from AbstractGenericName (default implementation)
public String asDataString() {
    if (isEmpty()) {
        return "";
    }
    StringBuffer sb= new StringBuffer();
    for (NameEnumeration i= getComponents(); i.hasMoreElements();) {
        String s= i.nextComponent();
        s= maskString(s);
        sb.append(s);
        sb.append(DELIMITER_CHAR);
    }
    sb.setLength(sb.length() - 1);
    return sb.toString();
}
```

Specific Implementation

```
// from StringName (fast implementation)
public String asDataString() {
    return fName.toString();
}
```


Part I: Summary

- Classes and interfaces
- Inheritance and code reuse
- Method types
- Unit testing

Part II: Design

Dirk Riehle

SKYVA International
www.skyva.com, www.skyva.de, www.skyva.ch

dirk@riehle.org, www.riehle.org

Part II: Topics

- Value objects
- Design aspects
- Design patterns
- Role modeling
- UML collaborations

Part II: Outline

- Step 4: Value Objects
- Step 5: Design Patterns
- Step 6: Role Modeling

Step 4: Topics

- Value objects
  - values and objects
  - working with value objects
- Value object design
  - values as shared immutable objects
  - first take at a value framework design

Object vs. Value Types

- Object types
  - object identity
  - reference semantics
  - referential integrity
- Value types (think ints and floats)
  - no identity, only "occurrences of representations"
  - value semantics ("copy semantics")
  - no integrity constraints on value level
- Programming languages should provide value concept
  - please note the "efficient classes proposal" for Java by James Gosling

Programming with BigInteger

```
// helper method: sums up list of BigInteger objects
public static BigInteger sum(Vector biList) {
    BigInteger result= BigInteger.valueOf(0);
    Enumeration i= biList.elements();
    while(i.hasMoreElements()) {
        BigInteger element= (BigInteger) i.nextElement();
        result= result.add(element);
    }
    return result;
}
```

Programming with Point

```
// how it should be with Point objects... but isn't.
public void move(Point delta) {
    fOrigin= fOrigin.add(delta);
    // rather than: fOrigin.x+= delta.x, fOrigin.y+= delta.y
}
```


63 of 172

Categories of Value Types

- Primitive value types
  - Character, Integer, String, etc.
- Business domain value types
  - GenericName, Address, Currency, Money, etc.
- Technical domain value types
  - FileName, URL, IPAddress, SecurityTicket, etc.

64 of 172

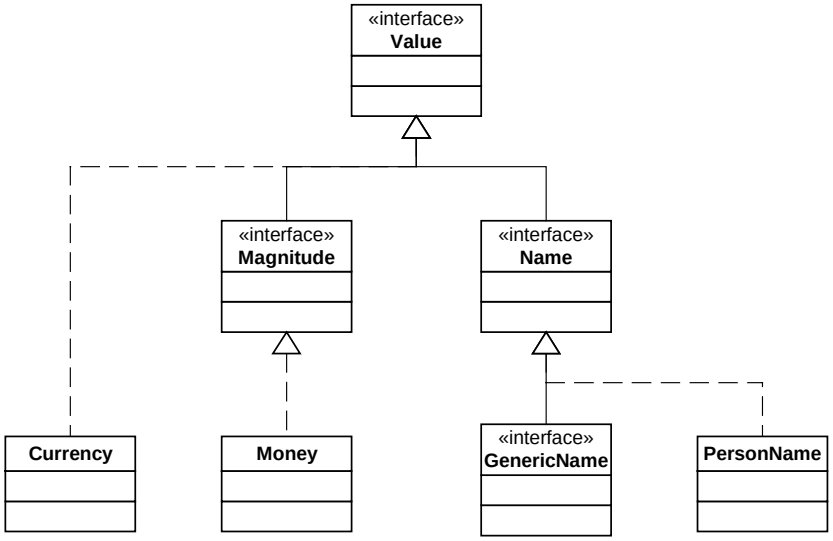
How to Implement Value Objects

- Immutable objects
  - no mutation methods
  - only query and helper (factory) methods
- Shared objects
  - use manager object to control instances
  - protect constructors

Why Use Value Objects?

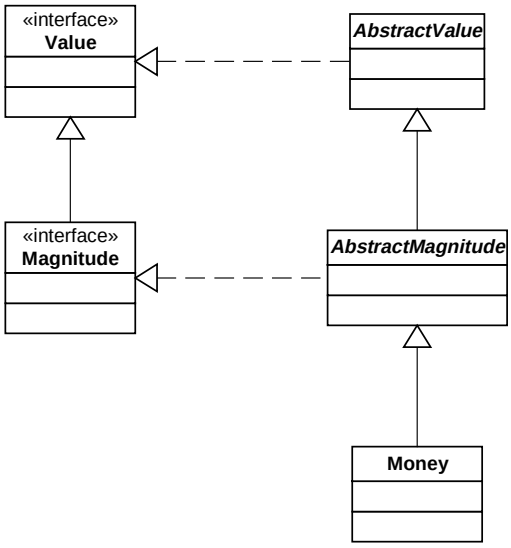
- Less programming errors
  - no side-effects!!
- Better in-memory performance
  - sharing objects (heap allocated)
  - copying objects (stack allocated, see efficient classes proposal)
  - concurrency: no locking overhead
- Better performance across process boundaries
  - serialization: always copy, no need to check references
  - persistence: no need to maintain referential integrity
  - distribution: always copy, no need to worry about references

Value Interface Hierarchy



67 of 172

Value Class Hierarchy Implementation



68 of 172

Value: Interface

```
public interface Value {
    public String asDataString();
    public String asString();
    public boolean isEqual(Value value);
    public boolean isValid();
    public String getTypeName();
    ...
}
```


Magnitude: Interface

```
public interface Magnitude extends Value {
    public boolean isGreater(Magnitude value);
    public boolean isGreaterOrEqual(Magnitude value);
    public boolean isLess(Magnitude value);
    public boolean isLessOrEqual(Magnitude value);
}
```

Money: Class Definition

```
public class Money extends AbstractMagnitude {
   protected Currency fCurrency;
   protected double fAmount;

   protected Money() { ... }

   public double getAmount() { ... }
   public Currency getCurrency() { ... }

   public Money add(Money m) { ... }
   public Money subtract(Money money) { ... }
   public Money multiply(double multiplier) { ... }
   public Money divide(double divisor) { ... }
   public Money negate() { ... }
}
```

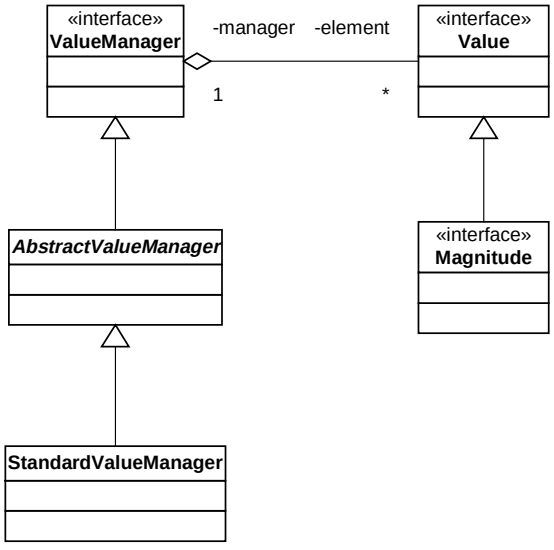

Programming with Money

```
public class Account {
    public Money getBalance() {
        return fBalance;
    }

    public void deposit(Money deposit) {
        // checking, logging, ...
        doSetbalance(getBalance().add(deposit));
    }

    protected void doSetBalance(Money balance) {
        fBalance= balance;
    }

    ...
}
```

ValueManager

- A value manager...
  - maintains list of value object for given type
  - returns value object upon request
  - registers new value objects
- A "proto-" value...
  - serves as a placeholder until new value object is requested
  - is returned as new value object

ValueManager: Class Model



ValueManager: Interface

```
public interface ValueManager {
    public int getNoDistinctValues();
    public int getNoTotalValues();

    public Value getProtoValue();
    public void setProtoValue(Value pv);

    public Value getValue(Value value);

    ...
}
```


Value Object Retrieval/Creation

- Client requests new value object
- Value initializes proto-value
- Value calls on value manager for real value object
- Value manager returns new/matching value object or null
- If null, proto-value is returned

Step 5: Topics

- Design aspects
  - sharing type information: *Type Object*
  - managing objects of same type: *Manager*
  - providing convenient access: *Singleton*
  - reading/writing value objects: *Serializer*
- Design patterns
  - concept of design patterns
  - pattern examples

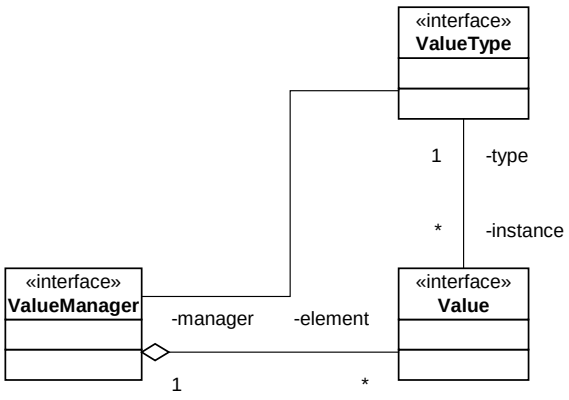
Design Aspect: Value Type Information

- Need to provide type information
  - name of value type
  - cardinality finite/infinite
  - default comparison algorithm
- One possible solution
  - implement type information as static methods
  - but: you program where you should just provide information
  - also: programming is more error-prone

ValueType

- ValueType instance
  - represents value type (class)
  - provides type information for one value type
  - centralizes all type information common to that specific value type
  - is configured at runtime
- ValueType provides operations
  - to access the type name
  - to determine whether value type is of finite cardinality
  - to access and change the current value implementation status
  - and more, as the values are used in more contexts (GUI, etc.)

ValueType: Class Model



ValueType: Interface

```
public interface ValueType {
    public String getName();

    public boolean hasFiniteCardinality();
    public boolean hasInfiniteCardinality();

    ...
}
```


AbstractValue: Type Object Access

```
public AbstractValue implements Value {
    public String getTypeName() {
        return getTypeObject().getName();
    }

    public ValueType getTypeObject() {
        return getValueManager().getValueType();
    }

    ...
}
```

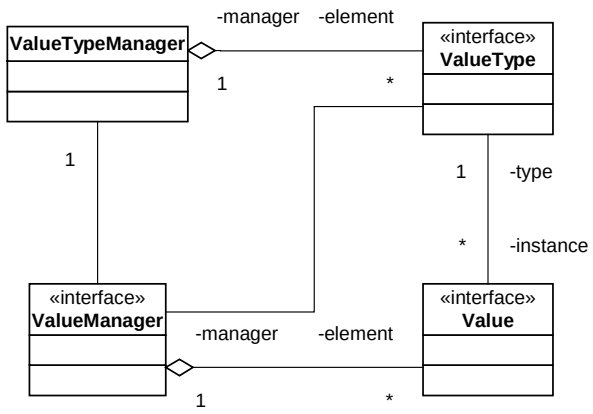
Design Aspect: Managing Objects

- How to learn about value types?
  - need convenient access to value types by string (type name)
- How to learn about available value types?
  - need to iterate over all value types
- Used in GUI's, for serialization, etc.

ValueTypeManager

- ValueTypeManager
  - maintaines ValueType objects
  - serves as convenient access point (by type name)
  - lets you iterate over available value types
- Not an interface, but a single class
  - because it is so simple
  - because it is unlikely to be extended any time soon

ValueTypeManager: Class Model



ValueTypeManager: Class Definition

```
public final class ValueTypeManager {
    protected Hashtable fValueTypes;
    protected static ValueTypeManager sInstance= null;

    public ValueTypeManager() { ... }

    public boolean hasValueType(String name) { ... }
    public ValueType setValueType(String name) { ... }
    public synchronized void addValueType(ValueType vt) { ... }

    public Enumeration getValueTypes() { ... }

    public static final ValueTypeManager getInstance() { ... }
}
```

Design Aspect: Convenient Access

- How to access ValueTypeManager object?
  - you want only one instance that must be shared by all clients
  - passing it as an operation parameter to every client bloats method signatures
- Provide global access point to single instance
  - access points ensures that there is only one instance
  - multithreading may require one instance per thread/operational context
- Single instance is called a Singleton

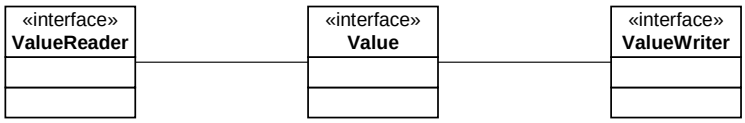
ValueTypeManager: Access

```
public final class ValueTypeManager {
    protected static ValueTypeManager sInstance= null;

    // see Double-Checked Locking pattern by Schmidt et al.
    public static final ValueTypeManager getInstance() {
        if (sInstance == null) {
            synchronized(ValueTypeManager.class) {
                if (sInstance == null) {
                    sInstance= new ValueTypeManager();
                }
            }
        }
        return sInstance;
    }

    ...
}
```

Design Aspect: Serialization of Values

- How to read and write value objects?
  - you need different formats for different purposes/backends
  - you don't want to change value type implementations for a new backend
  - you don't want clients to depend on specific value types

ValueReader/ValueWriter: Class Model



ValueWriter: Interface

```
public interface ValueWriter {
    public void writeBoolean(String fieldName, boolean value) throws IOException;
    public void writeBoolean(String fieldName, Boolean value) throws IOException;
    ...

    public void writeString(String fieldName, String value) throws IOException;
    public void writeVector(String fieldName, Vector value) throws IOException;
    ...

    public void writeInitialValue(Value value) throws IOException;
    public void writeValue(String fieldName, Value value) throws IOException;
    ...
}
```


Money: writeOn(ValueWriter)

```
public void writeOn(ValueWriter writer) throws IOException {
    writer.writeValue("currency", fCurrency);
    writer.writeDouble("amount", fAmount);
}
```

Serialization: Writing a Value Object

- Client retrieves/creates ValueWriter object
- Client calls writeInitialValue on writer with value as argument
- Writer calls writeOn on value with itself as argument
- Value calls write* methods of writer object

- Recursive descent is ended by primitive value types

Design Patterns

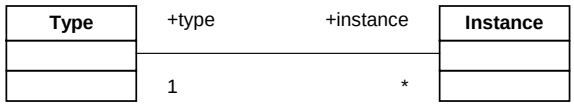
- Definition: Design Pattern
  - the abstraction from a recurring form in a specific context
  - form is frequently a problem solution (but not always!)
  - grow out of experience with designing and implementing systems
  - are documented in a structured form (see *Design Patterns* book)

- Experience and design patterns
  - you need experience to understand and apply a design pattern
  - design patterns are seldom helpful to novices (or as teaching devices)

Type Object: Explanation

- Intent: Type Object
  - represent type as object to decouple instances from their classes
  - be able to change type and create new types at runtime
- Definition: Type Object
  - is an object that represents a given type
  - is independent of the implementing class
- Examples
  - ValueType
  - (Class, Method, etc.)

Type Object Pattern: Structure Diagram



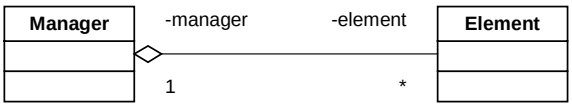
Type Object: Applicability

- Applicability (when to use)
  - if you need to flexibly provide type information
  - if you have a proliferation of subclasses
  - if you need to create new types dynamically
  - ...
- Consequences
  - you configure types rather than program them
  - you avoid subclass proliferation
  - you can change types dynamically
  - you increase design complexity
  - ...

Manager: Explanation

- Intent: Manager
  - encapsulate the management of instances of a class in a manager object
  - be able to vary management behavior independently of managed objects
- Definition: Manager
  - is an object that centralises management of set of managed objects
  - may comprise full life-cycle: creation, access, deletion
- Examples
  - ValueManager, ValueTypeManager
  - (java.awt.Container, java.rmi.Registry)

Manager: Structure Diagram



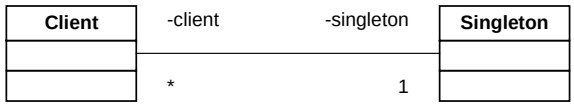
Manager: Applicability

- Applicability (when to use)
  - if elements of set of objects need to be managed properly
  - if you want to separate management from managed objects
  - ...
- Consequences
  - management is handled explicitly rather than buried in code
  - management can be changed without affecting managed objects
  - ...

Singleton: Explanation

- Intent: Singleton
  - ensure that a class has only one instance
  - provide a global point of access to it
- Definition: Singleton
  - is a globally accessible one-of-a-kind object
  - in multithreaded systems, there may be singletons-per-thread, etc.
- Examples
  - ValueTypeManager

Singleton: Structure Diagram



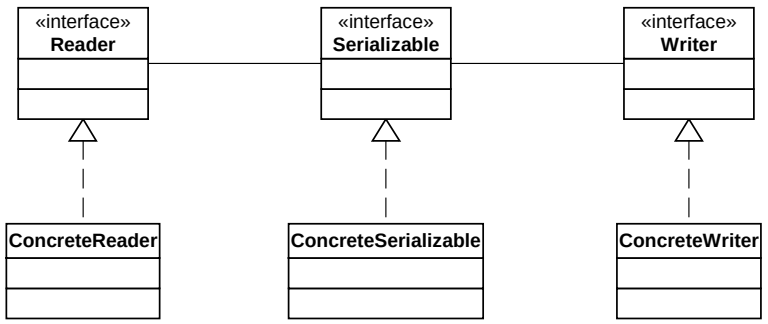
Singleton: Applicability

- Applicability (when to use)
  - if there must be exactly one instance of a class (in a given context)
  - if you want to change implementations without affecting clients
- Consequences
  - you get controlled access to the singleton
  - you can change implementations without affecting clients
  - the singleton is globally accessible
  - ...

Serializer: Explanation

- Intent: Serializer
  - encapsulate reading/writing algorithm of object structures in dedicated objects
  - thereby, efficiently read and write object structures from varying backends
  - backends may be flat files, relational databases, all kinds of buffers, etc.
- Definition: Serializer
  - is an object that either reads (Reader) or writes (Writer) an object structure
- Examples
  - Value/ValueReader/ValueWriter
  - Serializable/ObjectInputStream/ObjectOutputStream

Serializer: Structure Diagram



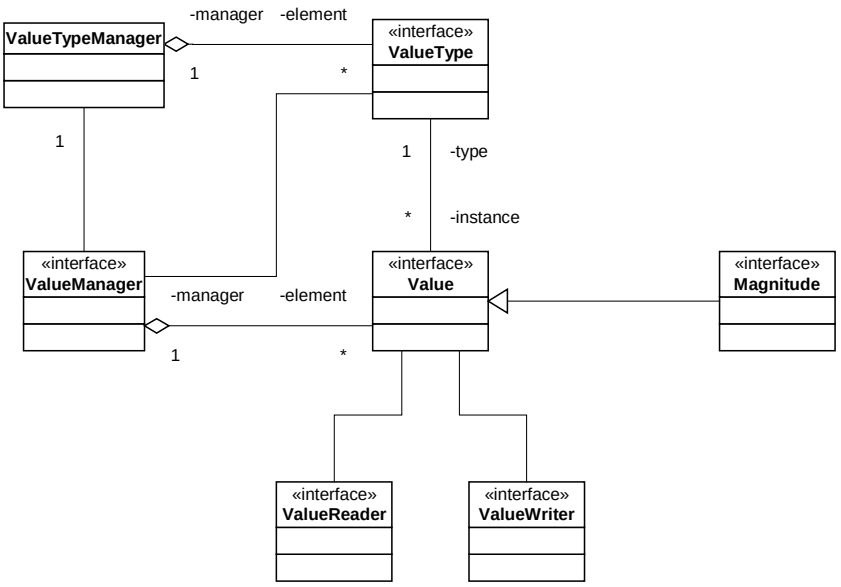
105 of 172

Serializer: Applicability

- Applicability (when to use)
  - if you need to read/write objects in varying formats
  - if you need to add new formats without affecting the objects to be read/written
  - ...
- Consequences
  - it is easy to add new/change existing formats and backends
  - makes primary objects simpler to implement
  - requires implementation of a Serializable interface
  - weakens encapsulation due to generic field access protocol
  - ...

106 of 172

Value: Overall Class Model



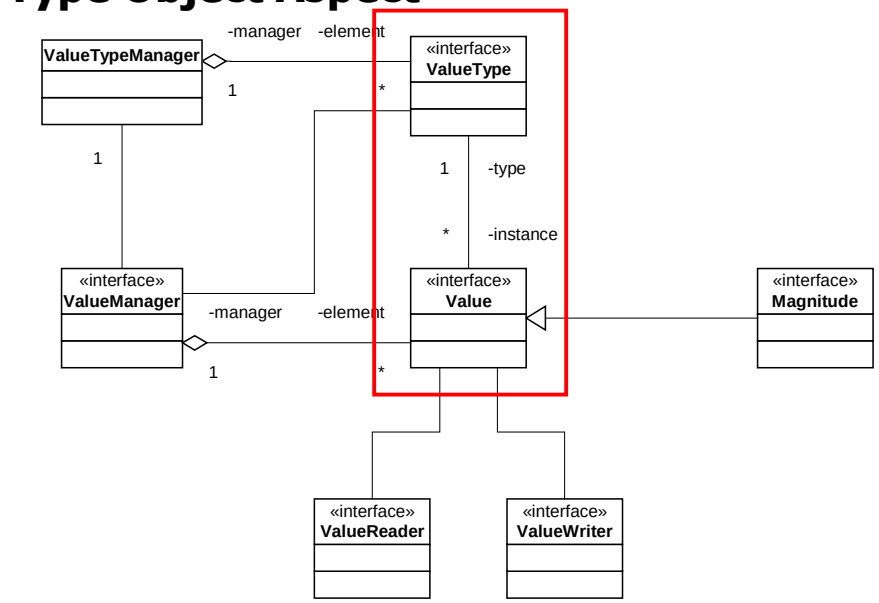
107 of 172

Value Class Model: High Pattern Density

- Sharing type information: *Type Object*
- Managing objects of same type: *Manager*
- Providing convenient access: *Singleton*
- Reading/writing value objects: *Serializer*
- And others more...

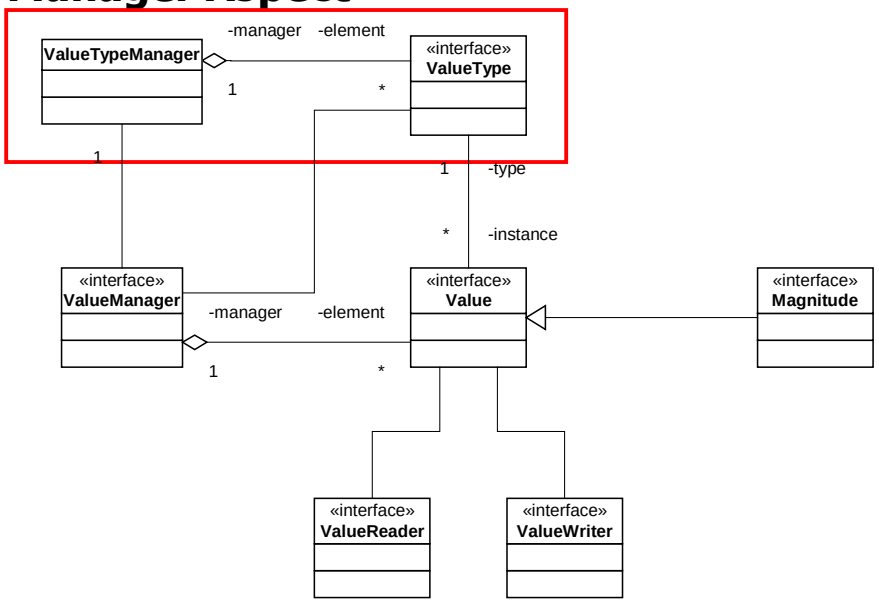
108 of 172

Type Object Aspect



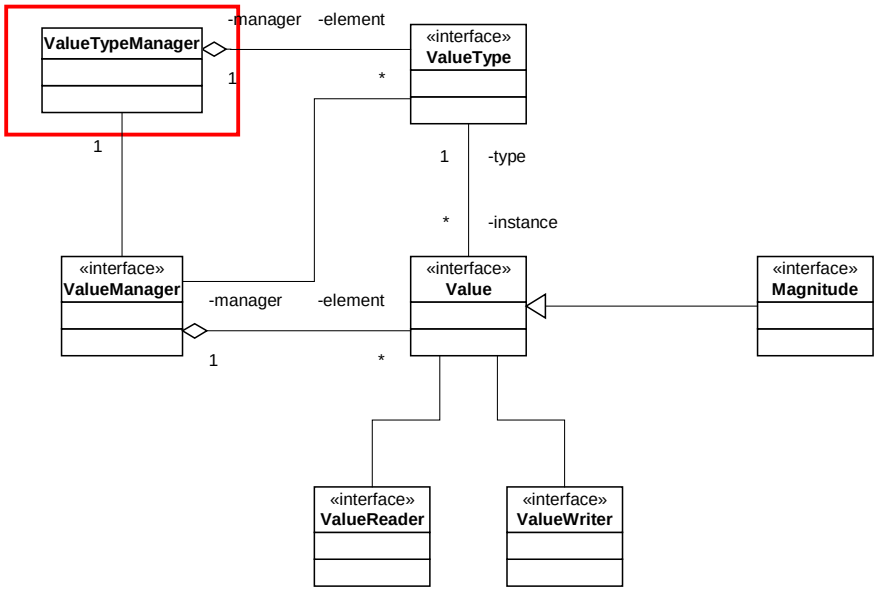
109 of 172

Manager Aspect



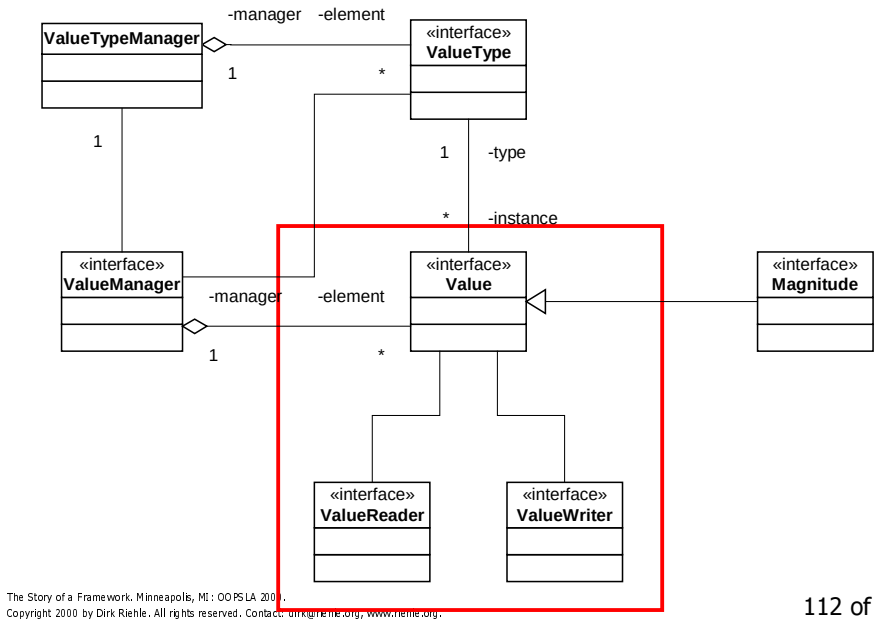
110 of 172

Singleton Aspect



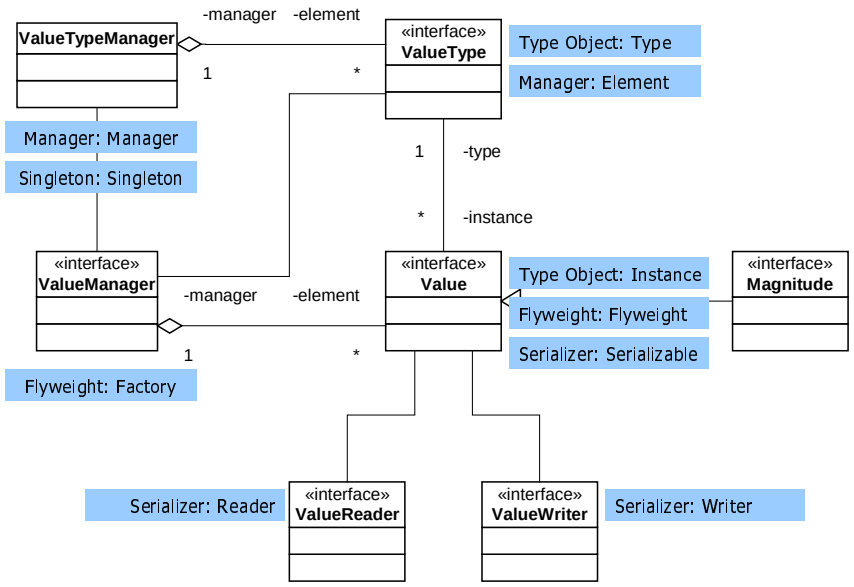
111 of 172

Serialization Aspect



112 of 172

Value Objects: Pattern Annotation



Step 6: Topics

- Role modeling
  - object role play
  - object collaboration
- UML collaborations
  - collaboration specifications
  - describing design patterns

Money: Design Aspects

- Design patterns, as discussed:
  - Flyweight, Type Object, Manager, Singleton, Serializer
- Design aspects, not discussed:
  - Value implements java.io.Serializable
- Design aspects, inherited from Object:
  - keyable, comparable, printable, multi-threaded
- Hidden design aspects, without methods:
  - immutable

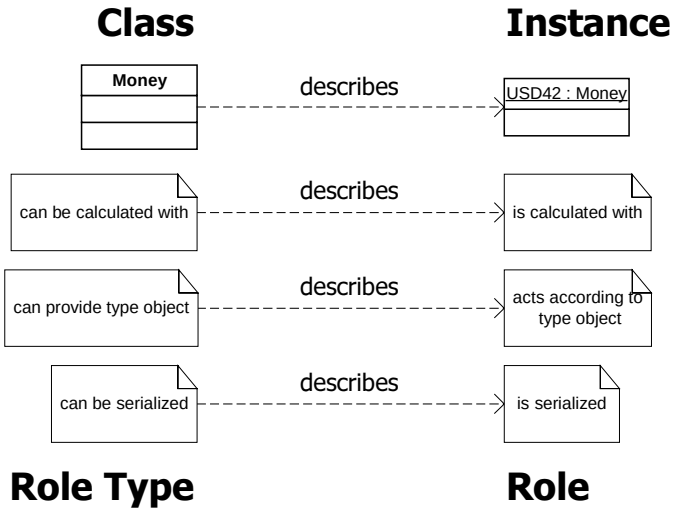
Complexity: Aspects Interact

- Value
  - createValue (ProtoValue aspect) and hashCode (keyable aspect)
  - initializing a Value with new data changes the hashcode
- Object
  - hashCode (keyable aspect) and equals (comparable aspect)
  - equal objects must provide same hashcode
- Still, aspects can be separated!

Roles: Aspects of a Specific Kind

- Definition: Role
  - an observable behavioral aspect of an object in a specific context
- Definition: Role type
  - a type that defines the behavior of a role an object may play
  - may or may not have methods
- Objects and roles: m-to-n relationship
  - objects play roles to act in different contexts
  - a role may be played by different objects over time (think call center)

Roles: Type/Instance Distinction



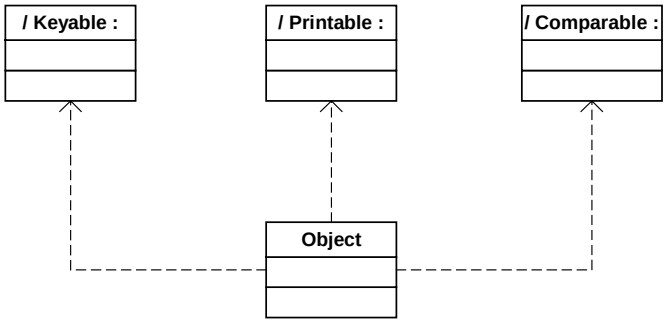
UML and Role Modeling

- UML provides support for role modeling...
  - but it is not complete
  - syntax and semantics document contradict each other
  - semantics document is much more precise than syntax document
- Heated debates about what constitutes role modeling
  - even after 1.3 release (or, perhaps, just then)
- Major influences on role concept in UML
  - Trygve Reenskaug: OOram

UML: ClassifierRoles

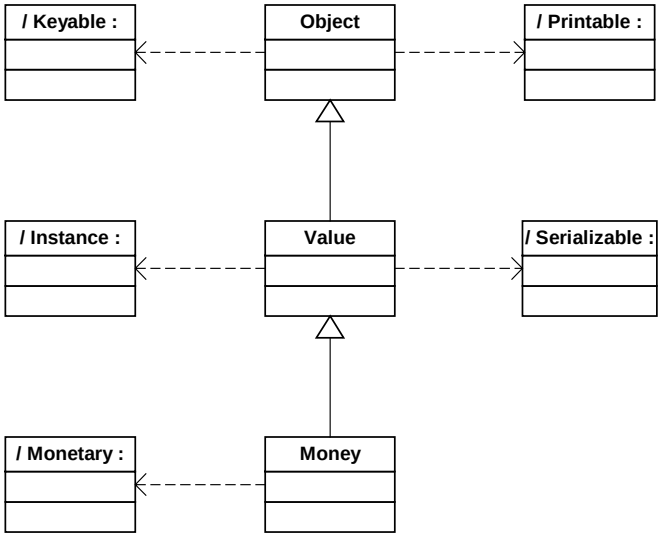
- UML concept for role: none
  - no instance level concept for roles
  - objects interact, but this is not filtered through roles
- UML concept for role type: ClassifierRole
  - a ClassifierRole is a Classifier
  - a ClassifierRole is attached to a Classifier
  - a Classifier may have several ClassifierRoles
  - ClassifierRoles are used in collaborations (discussed later)
- Notation: Object / ClassifierRole : Classifier
  - reminder: a Classifier may be a class or an interface or a datatype or ...

java.lang.Object: ClassifierRoles



121 of 172

Money: ClassifierRoles



122 of 172

Benefits of Modeling with Roles

- Reduces complexity
  - by breaking up complex (class) interfaces into parts
- Increases reuse
  - by separating roles from classes, we can reuse them

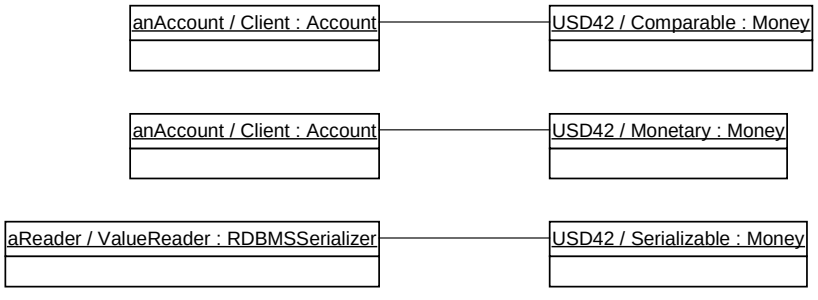
Implementing Roles

- Variant 1: as a set of methods in a class
  - simple but efficient implementation
  - accounts for about 80% of all cases
- Variant 2: as a dedicated interface
  - lets you reuse role for different classes
  - accounts for about 18% of all cases
- Variant 3: as a role object class
  - lets you dynamically attach roles at runtime
  - accounts for about 1% of all cases

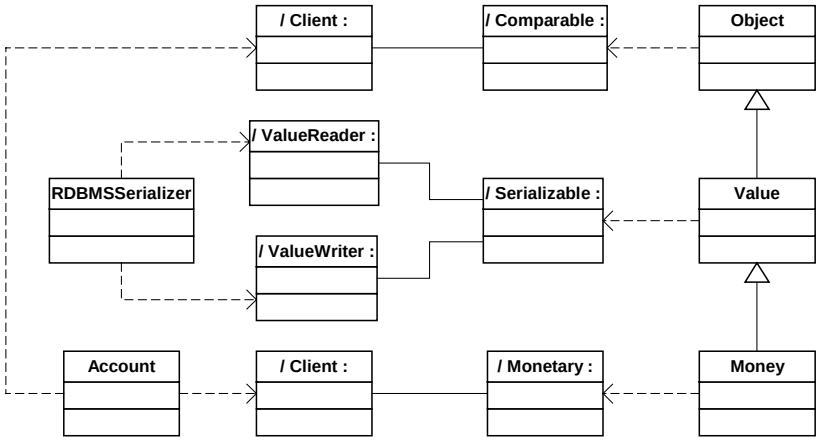
Implementation Dependencies

- Roles of one object interact
  - otherwise, there would be no synergy
  - a class is more than the sum of its roles
- Dependencies example
  - keyable: hashCode() provides hash code
  - equality: equals() must match hashcode for lookup in hashtables
  - shared immutable: equals() can be reduced to checking object identities

Money: Collaboration Instances



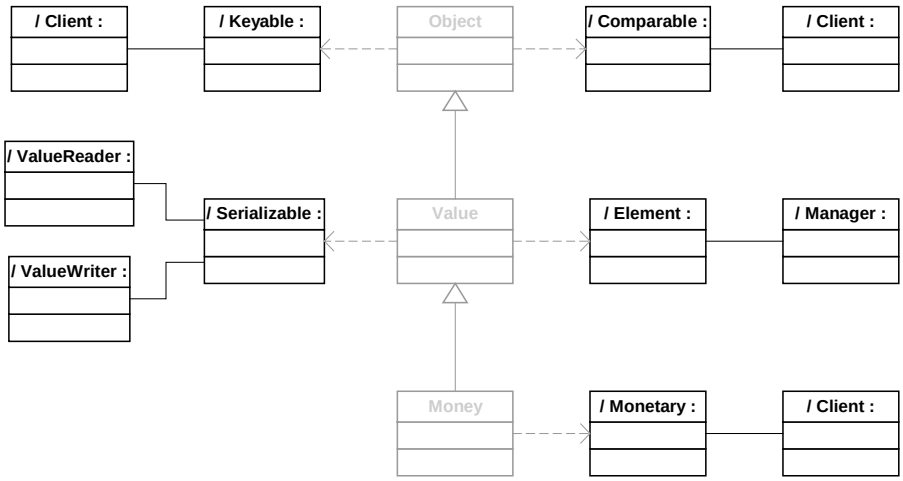
Money: Collaboration Specifications



UML: Collaborations

- Definition: Collaboration Instance
  - set of objects that collaborate by playing roles
  - Booch: "behavioral part of a collaboration"
  - UML 1.3: "collaboration on an instance level"
- Definition: Collaboration Specification
  - set of classifiers that relate to each other through classifier roles
  - Booch: "structural part of a collaboration"
  - UML 1.3: "collaboration on a specification level"
- Please note type/instance distinction again

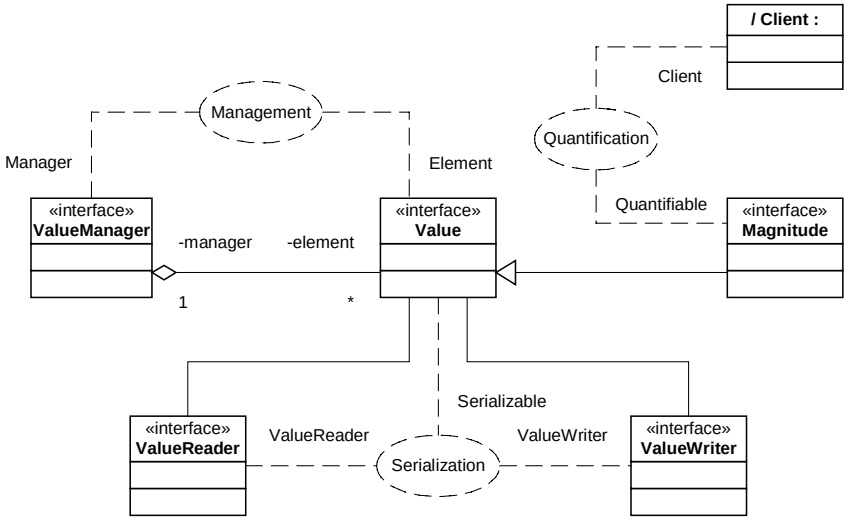
Money: Collaboration Specifications



Benefits of Collaboration Specifications

- Reduces complexity
  - by breaking up complex collaborations into parts
- Increases reuse
  - by separating collaborations from class relationships, we can reuse them

UML: Ellipse Shorthand



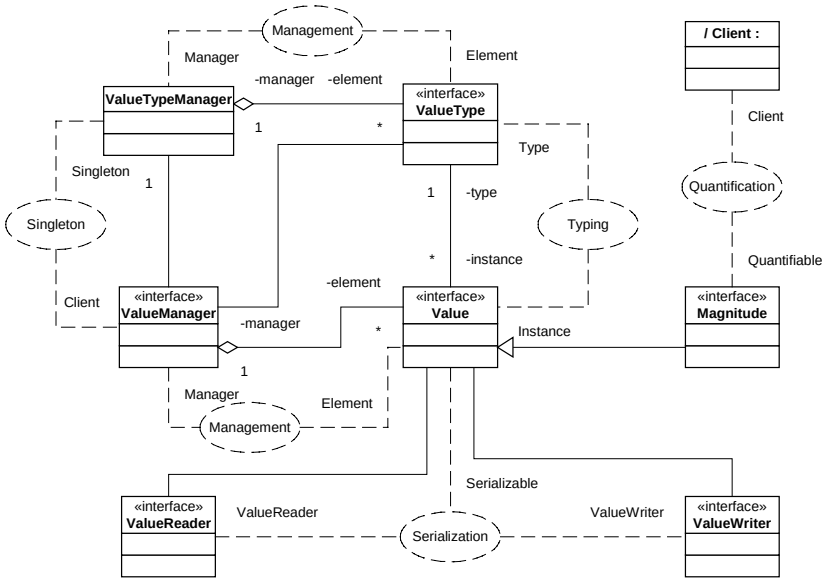
131 of 172

Design Patterns and Collaboration Specifications

- Design pattern
  - an idea of a problem solution
  - infinite and unconstrained variation in implementation
- Design template
  - a template for a specific design solution
  - infinite but constrained variation in implementation
- Design fragment
  - a specific design solution applied in the context of a larger system
  - exactly one implementation

132 of 172

Value: Overall Class Model



133 of 172

Part II: Summary

- Value objects
- Design aspects
- Design patterns
- Role modeling
- UML collaborations

134 of 172

Part III: Frameworks

Dirk Riehle

SKYVA International
www.skyva.com, www.skyva.de, www.skyva.ch

dirk@riehle.org, www.riehle.org

Step 7: Topics

- Value framework
  - interface architecture
  - implementation architecture
- Framework types
  - black-box frameworks
  - white-box frameworks
- Framework clients
  - inheritance clients
  - use-clients

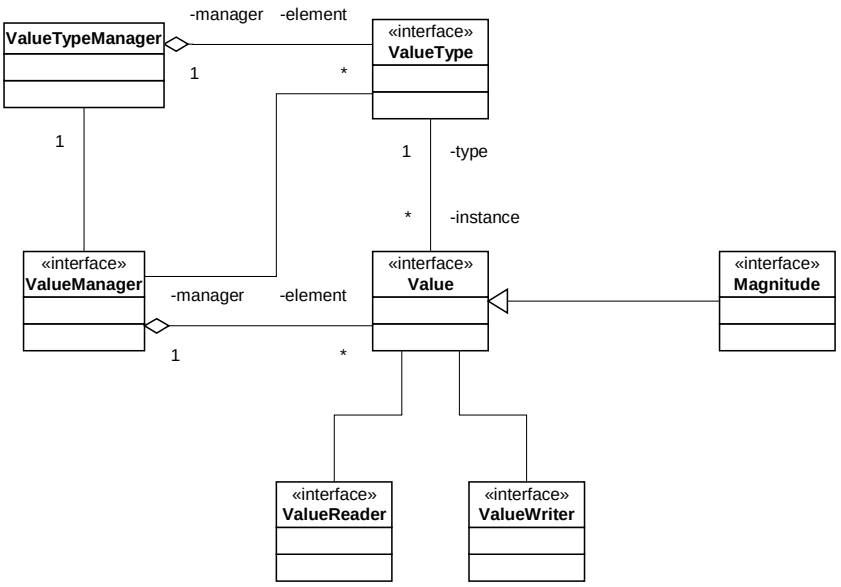
Framework (Traditional Definition)

- Definition: Framework
  - a reusable design and implementation that covers a particular domain
  - consists of an interface architecture and a (partial) implementation
- Purpose of frameworks
  - higher productivity
  - shorter time-to-market
  - less bugs
  - more homogeneous application suite
- Because of improved design and code reuse

Interface Architecture

- Definition: Interface Architecture
  - set of interfaces that determines overall structure and behavior
  - prescribes architecture for given domain or aspect thereof
- A.k.a. abstract design

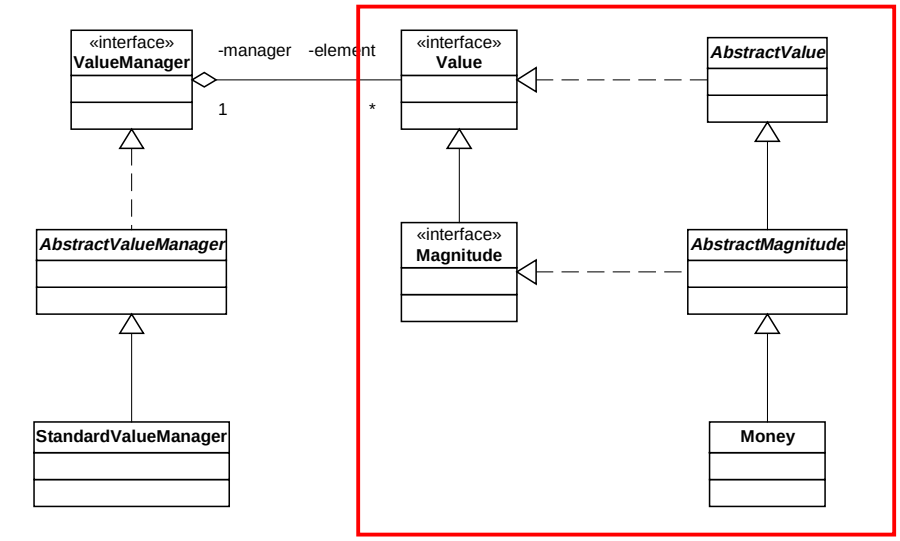
Value Framework: Interface Architecture



Implementation Architecture

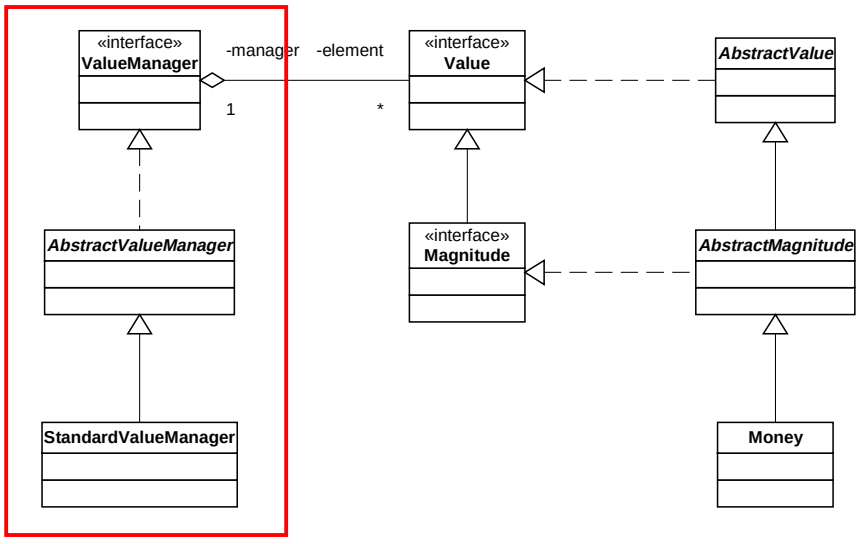
- Definition: Implementation Architecture
  - set of abstract and concrete classes that implement an interface architecture
  - provides readily usable classes as well as half-fabricated customizable classes
- A.k.a. concrete design, implementation

Value Class Hierarchy Implementation



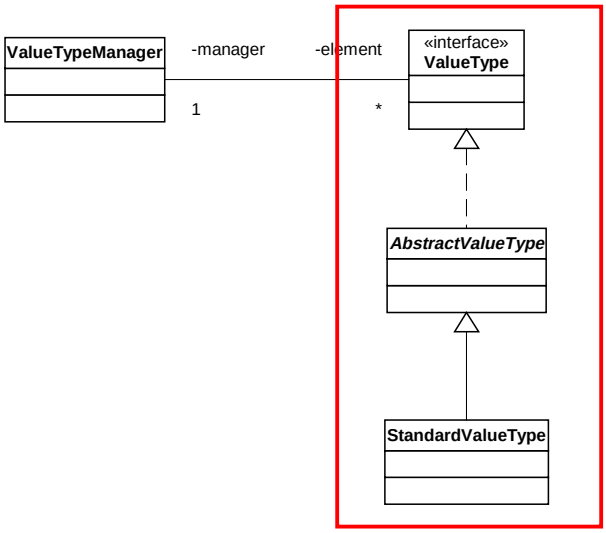
141 of 172

ValueManager Implementation



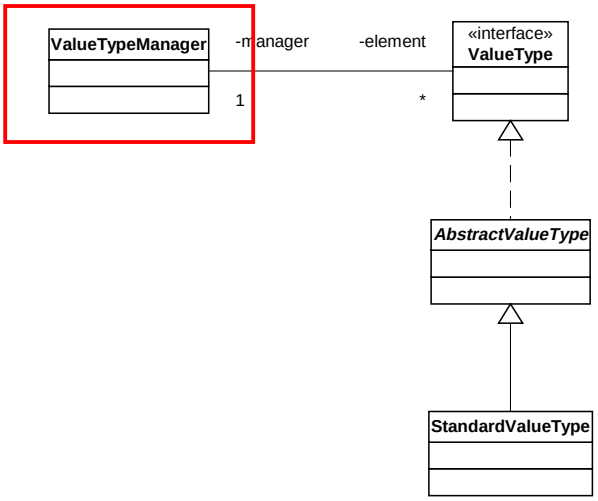
142 of 172

ValueTypeManager Implementation



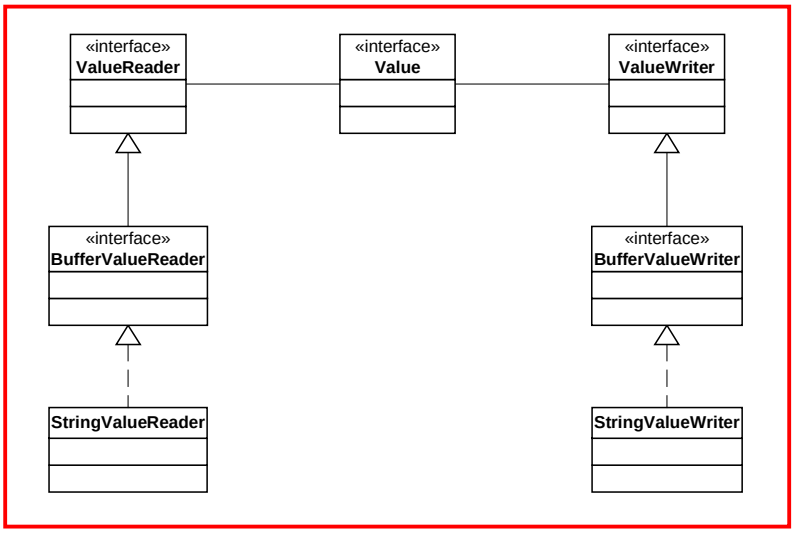
143 of 172

ValueType Implementation



144 of 172

Serialization Implementation



145 of 172

Implementation Variants

- Single class
  - ValueTypeManager
- Single interface + concrete class
  - *Reader, *Writer
- Single interface + abstract class + concrete classes
  - Value
  - ValueType
  - ValueManager

146 of 172

Framework Types

- White-box frameworks
- Black-box frameworks
- Gray-box frameworks

White-Box Frameworks

- Definition: White-Box Framework
  - a framework that provides abstract superclasses for subclassing
  - extension clients first use inheritance to customize framework
  - use-clients then compose objects to apply framework
- Examples
  - Java Object framework

Black-Box Frameworks

- Definition: Black-Box Framework
  - a framework that provides readily usable classes
  - use-clients use object composition to apply the framework
- Examples
  - Swing (most mature GUI frameworks)

Gray-Box Frameworks

- Definition: Gray-Box Framework
  - a framework that is both a black-box and a white-box framework
  - most frameworks are gray-box frameworks (except for very mature ones)
- Examples
  - Value framework
  - Swing
- Classification is not so important
- More important is how to use a framework

Using Frameworks

- Inheritance clients
  - use inheritance interface
- Use-clients
  - use collaboration specifications

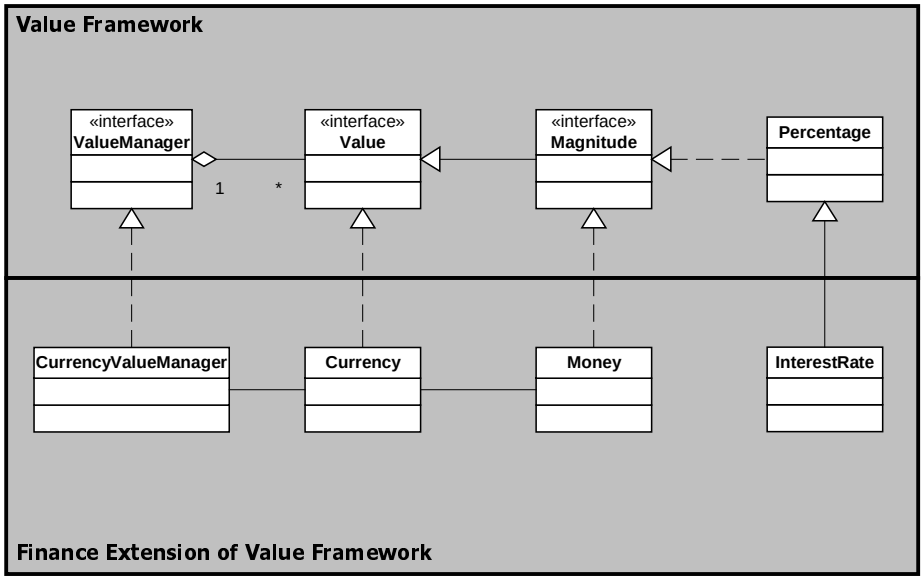
Framework (Extended Definition)

- Definition: Framework (traditional)
  - a reusable design and implementation that covers a particular domain
  - consists of an interface architecture and a (partial) implementation
- Definition: Framework (extended)
  - the class model of the framework provides
    - an extension point class set
    - a free classifier role set
    - a built-on class set

Framework Extension

- Definition: Framework Extension
  - a set of interfaces and classes that subclass a framework
  - consists of an interface architecture and a (partial) implementation
  - may be an application-specific extension or a framework itself
- Examples
  - Value framework extension for banking domain
  - Value framework for internet naming domain
- A.k.a. framework application
  - bad naming though, bears risk of confusion

Framework Extension: Finance/Currency

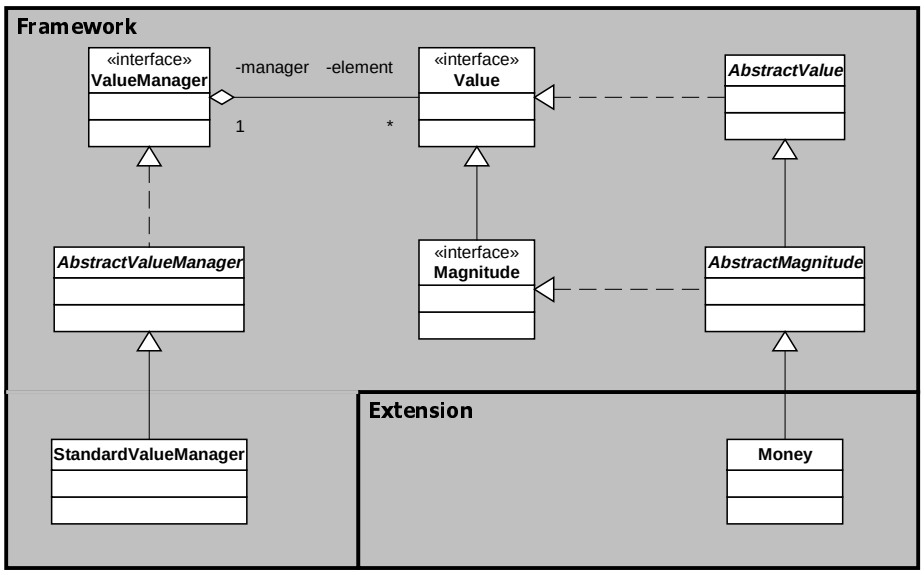


Extension Point (Classes)

- Definition: Extension Point (Class)
  - a framework class that may be subclassed by framework-external classes
  - define inheritance interface to framework extensions
- Definition: Extension Point Class Set
  - set of extension point classes of a framework
- Definition: Extension Class
  - class of a framework extension that subclasses an extension point class
  - make use of a framework's inheritance interface

155 of 172

Value Framework: Extension Points

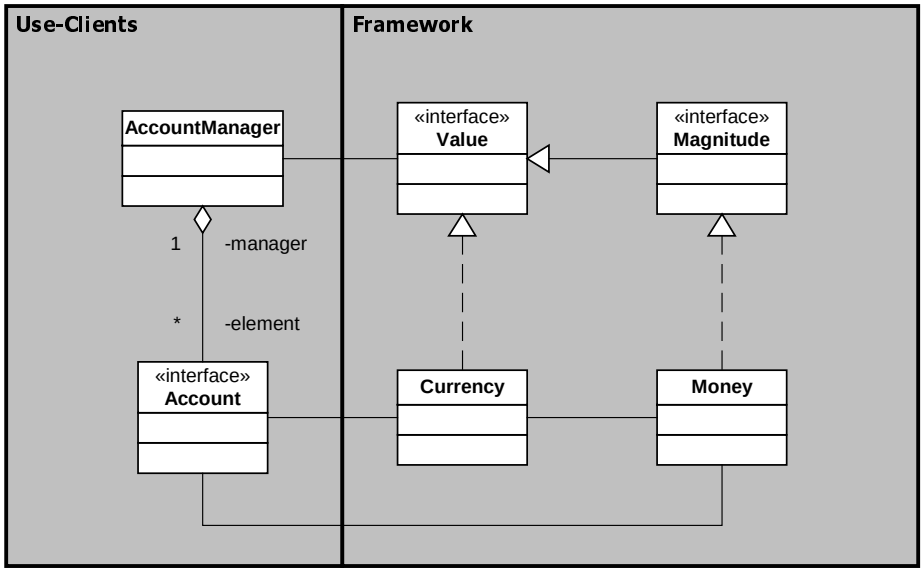


156 of 172

Framework Use by Use-Clients

- Use-clients create framework objects
  - from framework classes
  - from framework extension classes
- Use-clients compose framework objects

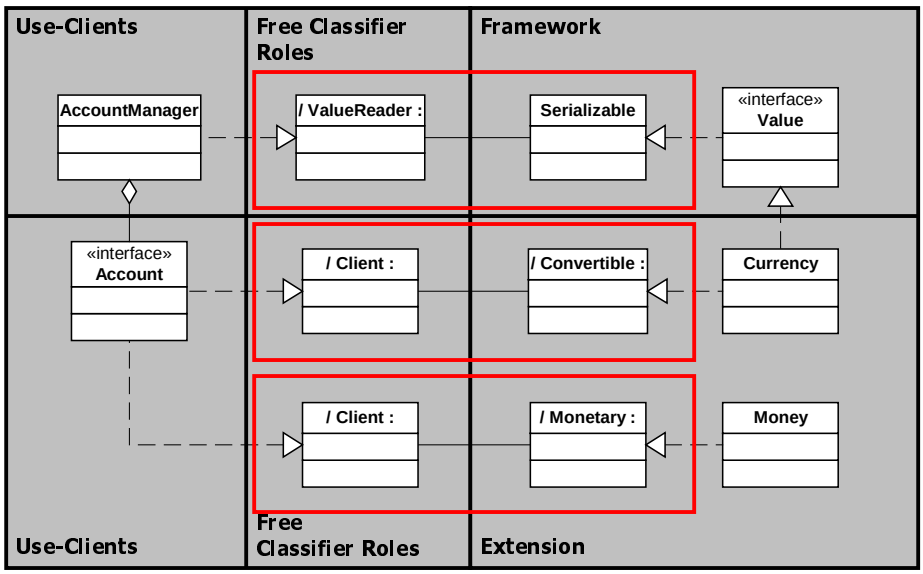
Framework Use-Clients: Finance/Account



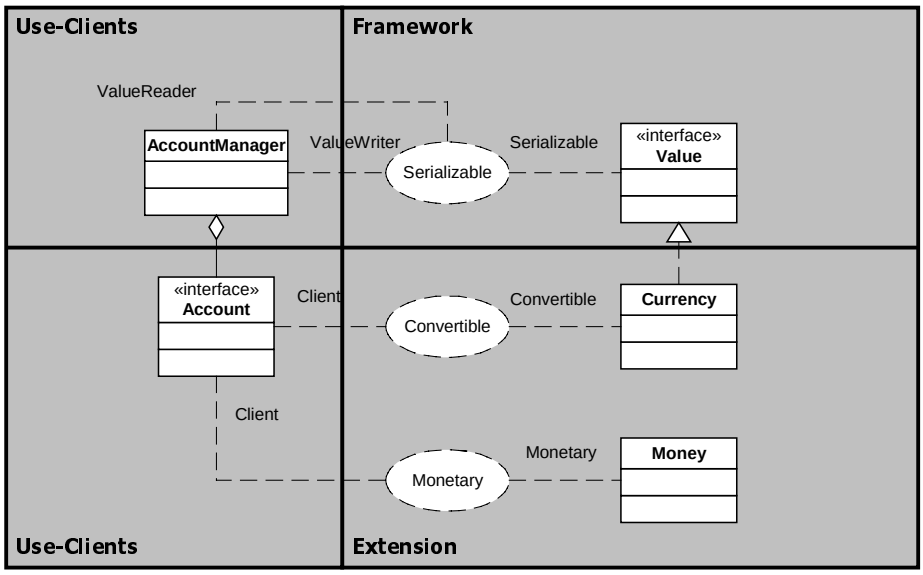
Definition: Free ClassifierRole

- Definition: Free Classifier Role (of a Framework)
  - a classifier role (ClassifierRole) that is available to use-clients
  - use-client classes pick up free classifier roles to make use of a framework
- Definition: Free Collaboration Specification (of a Framework)
  - a collaboration specification that provides free classifier roles

Use-Clients with Free Classifier Roles



Use-Clients with Ellipse Shorthand



How to Document a Framework

- Specify the use-client interface
  - determine and publish only free classifier roles to clients
  - document collaboration with free collaboration specifications
- Specify inheritance interface
  - determine extension point class set and
  - document inheritance interfaces of extension point classes
- Specify dependencies
  - determine classes a framework itself builds on
  - document relationship with built-on classes using collaboration specifications

Part III: Summary

- Value framework
  - interface architecture
  - implementation architecture
- Framework types
  - black-box frameworks
  - white-box frameworks
- Framework clients
  - inheritance clients
  - use-clients

JValue: A Java Value Object Framework

- The Value Object framework exists: JValue
  - available using the LGPL 2.0
  - contributions welcome!
- Please see: http://www.jvalue.org

Literature: Interfaces and Classes

- Dirk Riehle and Erica Dubach. "Working with Java Interfaces and Classes: How to Separate Interfaces from Implementations." *Java Report* 4, 7 (July 1999).
- Dirk Riehle and Erica Dubach. "Working with Java Interfaces and Classes: How to Maximize Design and Code Reuse in the Face of Inheritance." *Java Report* 4, 10 (October 1999).
- http://www.riehle.org/java-report

Literature: Method Types and Properties

- Dirk Riehle. "Method Types in Java." *Java Report* 5, 2 (February 2000).
- Dirk Riehle. "Method Properties in Java." *Java Report* 5, 5 (May 2000).
- http://www.riehle.org/java-report

Literature: Unit Testing

- Kent Beck and Erich Gamma. "Test Infected: Programmers Love to Write Tests." http://members.pingnet.ch/gamma/junit.htm
- Source code for JUnit: http://www.xprogramming.com/software
- http://c2.com/cgi/wiki?UnitTests

Literature: Value Objects

- Dirk Bäumer et al. Values in Object Systems. *Ubilab Technical Report 98-10-1.* Switzerland, UBS AG: 1998. http://www.riehle.org/papers/1998/ubilab-tr-1998-10-1.html
- Ward Cunningham. The CHECKS Pattern Language of Information Integrity. *Pattern Languages of Program Design 1.* Addison-Wesley, 1995.
- http://www.jvalue.org

Literature: Design Patterns 1/2

- Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides. *Design Patterns.* Addison Wesley, 1995.
- Frank Buschmann et al. *Pattern-Oriented Software Architecture.* Wiley, 1996.
- Robert Martin et al. *Pattern Languages of Program Design 3 (PLoPD-3).* Addison-Wesley, 1998.
- http://www.riehle.org/patterns/index.html

Literature: Design Patterns 2/2

- Manager. Published in *PLoPD-3.*
- Serializer. Published in *PLoPD-3.*
- Singleton. Published in *Design Patterns.*
- Type Object. Published in *PLoPD-3.*

Literature: Role Modeling

- Trygve Reenskaug. *Working with Objects.* Prentice Hall, 1995.
- Egil Andersen. *Conceptual Modeling of Objects.* Dissertation, University of Oslo, 1997.
- Michael VanHilst. *Role-Oriented Programming for Software Evolution.* Ph.D. Thesis, University of Washington, 1997.

Literature: Frameworks

- Ted Lewis et al. *Object-Oriented Application Frameworks.* Prentice-Hall, 1995.
- Taligent Press. *The Power of Frameworks.* Addison-Wesley, 1995.
- Dirk Riehle. *Framework Design: A Role Modeling Approach.* Dissertation, ETH Zürich, 2000.
- http://www.riehle.org/diss