

# Framework Design And Implementation

**Dirk Riehle, Stanford GSB**

dirk@riehle.org, www.riehle.org

**Alan Perry, SKYVA International**

aperry@skyva.com, www.skyva.com

Seattle, WA: OOPSLA 2002

Last updated.

Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

---

---

## Purpose and Approach of Tutorial

- Purpose: Tutorial
  - to show how to design and implement frameworks to achieve promised business benefits like shorter time-to-market and fewer bugs
- Approach: Tutorial
  - uses a real-world example (JValue) and demonstrates key aspects of code and design reuse using this example

## Tutorial: Structure

- Part I: Motivation
- Part II: Code Reuse for Frameworks
- Part III: Design Reuse for Frameworks
- Part IV: Frameworks as Components
- Part V: Summary

## Part I: Motivation

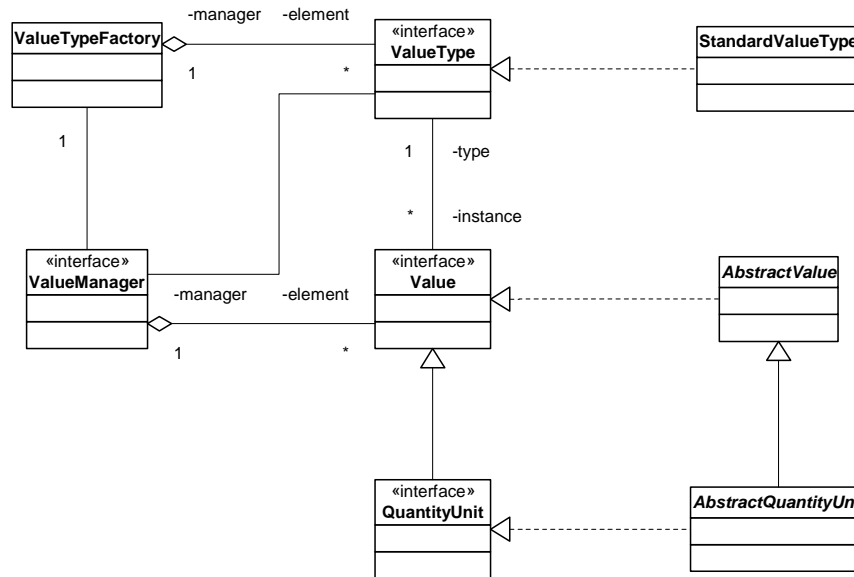
## Part I: Topics

- What is a framework?
- Business benefits of frameworks
- Key characteristics of frameworks
- Who develops and who uses frameworks?

## Framework (Traditional Definition)

- Definition: Framework
  - a reusable abstract design and implementation that covers a particular domain
- Examples from JDK
  - Object framework, Collection classes, Swing classes, RMI

## Example: JValue Framework



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

7 of 143

---

---

## Value Objects with JValue

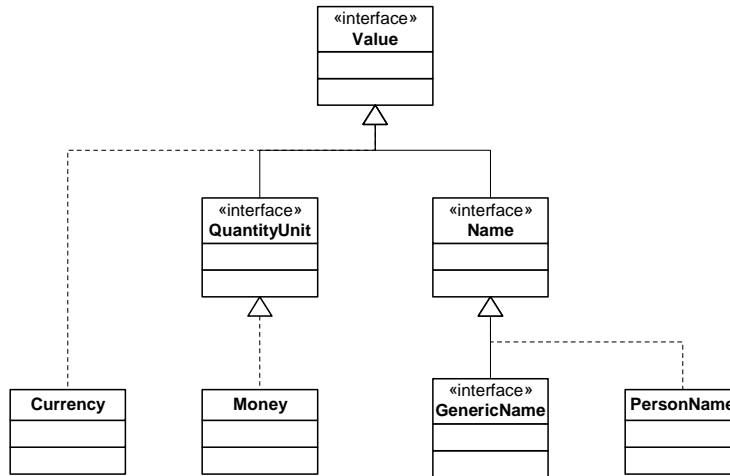
- Definition: Value Object
  - an object that represents a value, that is, has value semantics
- Value semantics
  - no identity, multiple representations
  - equality not based on identity
  - freedom from side-effects
- Value Object implementation
  - immutable objects
  - possibly shared objects (finite cardinality)

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

8 of 143

---

## Part of Value Interface/Class Hierarchy



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

9 of 143

---

---

## Value: Interface

```
public interface Value extends Serializable {
    public String toString();
    public String asDataString();

    public boolean equals(Object value);
    public boolean isDefaultValue();
    public boolean isUndefinedValue();

    public String getTypeName();
    public ValueType getTypeObject();

    public ValueManager getValueManager();

    public Object getAttributeValue(String name);

    public void writeOn(ValueWriter writer) throws ValueWriterException;
}
```

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

10 of 143

---

## Advantages of Frameworks

- Shorter time-to-market
  - you reuse existing design and code/don't start from scratch
- Higher productivity
  - frameworks make intelligent use of domain expert developers
- Fewer bugs and conceptual errors
  - you are re-using (hopefully) mature code
  - you are re-using a mature domain model
- More homogenous application suite
  - frameworks enforce adherence to architectural and programming style
  - helps development of application family/product line

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

11 of 143

---

---

## Disadvantages of Frameworks

- Design and implementation constraints
  - the very virtue of the pre-defined architecture can become a burden
- High upfront investment
  - developers typically face a steep learning curve

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

12 of 143

---

## Key Characteristics of Frameworks

- Optimal code reuse
  - well-designed abstract superclasses
  - well-defined default implementation classes
- Optimal design reuse
  - well-understood domain model
  - well-defined configurations for standard problems
- Optimal componentization
  - well-defined domain boundaries
  - well-documented purpose and limits

## Who Develops Frameworks?

- Expert programmers with sufficient domain knowledge
  - you need to understand the domain and its boundaries
- Domain experts with sufficient programming knowledge
  - you need to understand how to create a reusable design and produce reusable code

## Who Uses Frameworks?

- People who want to realize the promised business benefits
  - people who value the investment made by domain experts
  - people who want to develop a system that spans multiple domains
- Smart people!

## Part I: Summary

- Framework definition
- Business benefits
- Technological characteristics
- Are frameworks for you?

# Part II: Code Reuse for Frameworks

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

17 of 143

---

---

## Part II: Topics

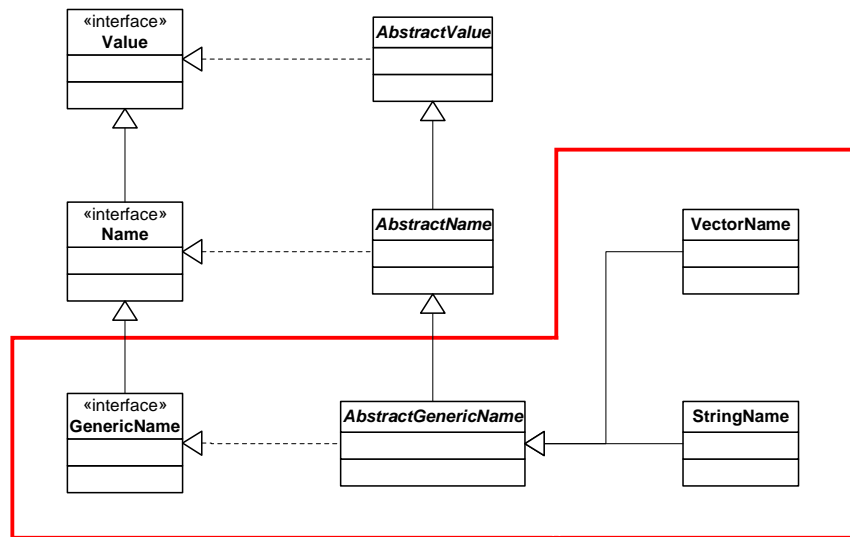
- Interface design
- Abstract superclasses
- Interfaces and implementations

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

18 of 143

---

## Example for Code Reuse



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

19 of 143

---

## 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://www.jvalue.org/jvalue-0.6.10/index.html"

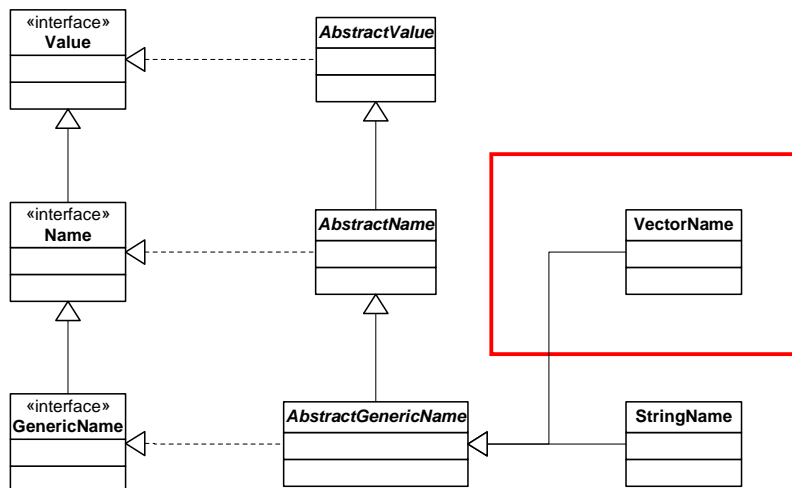
Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

20 of 143

## Topic 1: Interface Design

- Design by contract
- Method design

## Topic 1: Focus



## Interfaces as Contracts

- 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

## 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 referred to as "class interface"
  - example: interface of VectorName class

## VectorName

- VectorName provides one field
  - a Vector holding the name components
  - defines default delimiter and escape characters
- VectorName provides methods for
  - creating a name
  - querying, comparing, and representing the name
  - changing the name

---

---

## VectorName: Class

```
public class VectorName extends AbstractGenericName {
    private Vector fComponents;

    public static getValue(String name) { ... }
    public static getValue(Vector components) { ... }

    protected Value() { ... }
    protected initialize(Vector components) { ... }

    public String toString() { ... }

    public boolean isEmpty() { ... }

    public String getComponent(int index) { ... }
    public GenericName setComponent(int index, String nc) { ... }

    ...
}
```

## Design by Contract

- Define state space
  - invariants
- Define possible transitions in state space
  - preconditions, postconditions
- Define expected client behavior
  - think collaboration

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

27 of 143

---

---

## Contract Example: GetComponent(...)

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

protected void assertIsValidIndex(int i) throws IndexOutOfBoundsException {
    int upperLimit = getNoComponents();

    if ((i < 0) || (i >= upperLimit)) {
        String msg = String.valueOf(i)+" (of "+String.valueOf(upperLimit)+")";
        throw new IndexOutOfBoundsException(msg);
    }
}

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

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

28 of 143

---

## Design by Contract: Responsibilities

- Client object responsibility
  - to use service object properly as defined
- Service object responsibility
  - to ensure valid use only and reject invalid attempts
  - to fail explicitly if service cannot be provided
- Collaboration view will return...

## Method Design

- Methods should serve exactly one purpose
  - overly specialized methods are hard to reuse
  - it is easier to describe the contract of such a method
- Methods usually fall into one category:
  - query methods
  - mutation methods
  - helper methods
- Mixing categories spells trouble
  - unless you follow well-established idioms/patterns
  - unless your interface is very simple
  - unless you want to ensure atomicity of operation

## Query Methods

- Definition: Query 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`

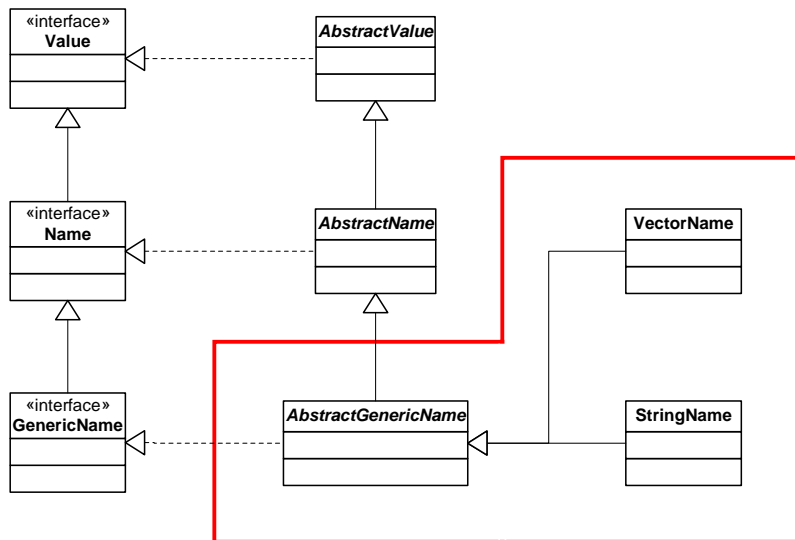
## Helper Methods

- 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()`

## Topic 2: Abstract Superclasses

- Design by primitives
- Inheritance interface

## Topic 2: Focus



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

35 of 143

---

---

## GenericName Implementations

- Using a String...
  - 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
- Using a Vector...
  - 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?

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

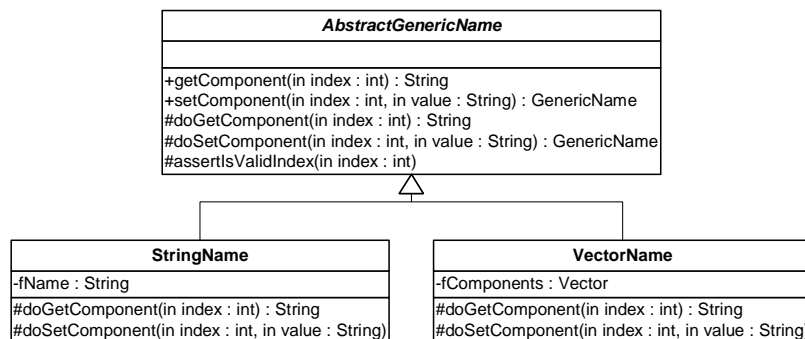
36 of 143

---

## 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

## AbstractGenericName Superclass



## AbstractGenericName: getComponent(...)

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

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

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

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

39 of 143

---

---

## 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)

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

40 of 143

---

## AbstractGenericName: Inheritance Interface

```
public abstract class AbstractGenericName extends ... {  
    ...  
  
    protected abstract String doGetComponent(int i);  
    protected abstract GenericName doSetComponent(int i, String c);  
    protected abstract GenericName doInsert(int i, String c);  
    protected abstract GenericName doRemove(int i);  
    protected abstract GenericName createGenericName(String n);  
  
    ...  
}
```

---

---

## Design by Primitives

- Definition: Primitive Method
  - does exactly one specific "primitive" thing
  - encapsulates access to object's state
- Primitive methods ...
  - can be composed (by compound methods)
  - 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

## More Method Types

- Compound methods
  - regular methods that use primitive or other compound methods
  - provide non-primitive higher-level functionality
- Hook methods
  - are meant to be overwritten
  - may have default implementation in superclass
- Template methods
  - are compound methods that rely on primitive hook methods

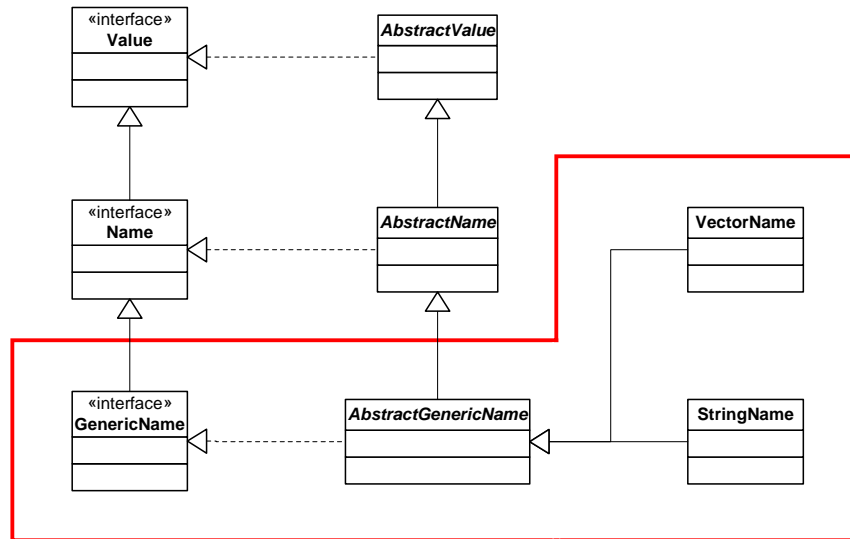
---

---

## Topic 3: Separation of Interface from Implementation

- Explicit interfaces
- Interface/abstract superclass/concrete class pattern

## Topic 3: Focus



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

45 of 143

---

---

## GenericName: Interface

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

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

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

    ...
}
```

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

46 of 143

## 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

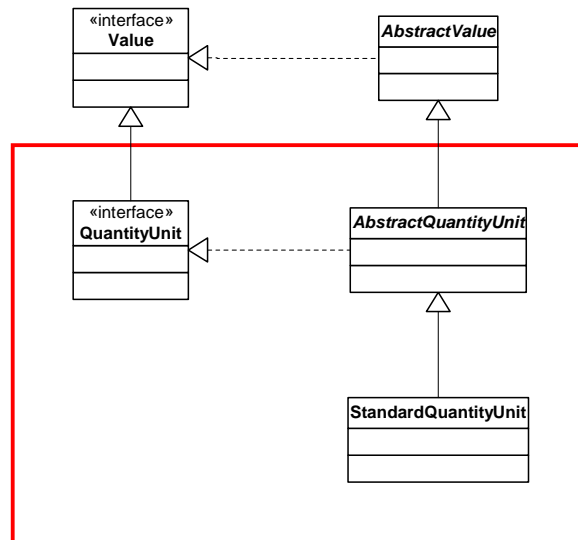
## Organizational Benefits of Interfaces

- 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

## Example: QuantityUnit



## QuantityUnit: Interface

```
public interface QuantityUnit extends Value {  
    ...  
  
    public boolean isGreaterThan(QuantityUnit value);  
    public boolean isGreaterThanOrEqualTo(QuantityUnit value);  
    public boolean isLessThan(QuantityUnit value);  
    public boolean isLessThanOrEqualTo(QuantityUnit value);  
  
    ...  
}
```

## Implementation Variants

- Single class
  - ValueTypeFactory
- Single interface + concrete class
  - ValueReader, ValueWriter
- Single interface + abstract class + concrete classes
  - Value, ValueType, ValueManager

## Part II: Summary

- Interface design
- Abstract superclasses
- Interfaces and implementations

## Part III: Design Reuse for Frameworks

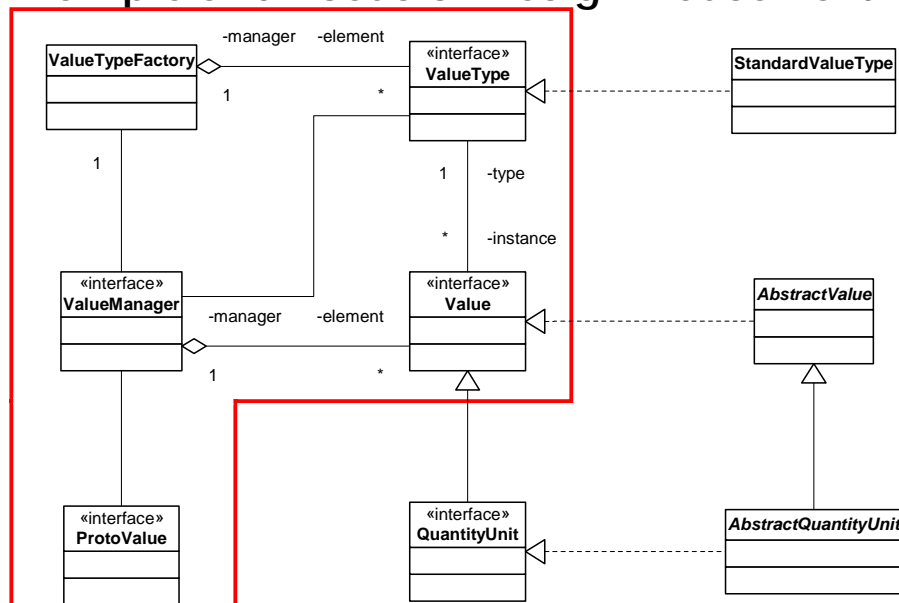
## Part III: Topics

- Abstract design
- Separation of concerns
- Design patterns

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

55 of 143

## Example and Focus of Design Reuse Part



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

56 of 143

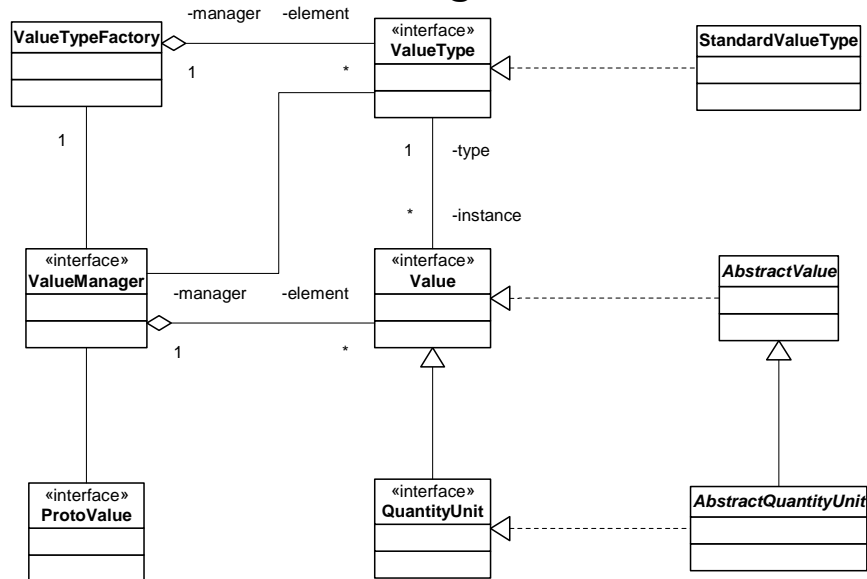
## Topic 4: Abstract Design

- JValue design
  - illustration, examples
- UML class models

## Abstract Design

- Definition: Abstract Design
  - comprises key classes and interfaces and their relationships
  - determines overall behavior and structure of applications
- UML class model
  - used to express an abstract design

## JValue: Abstract Design



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

59 of 143

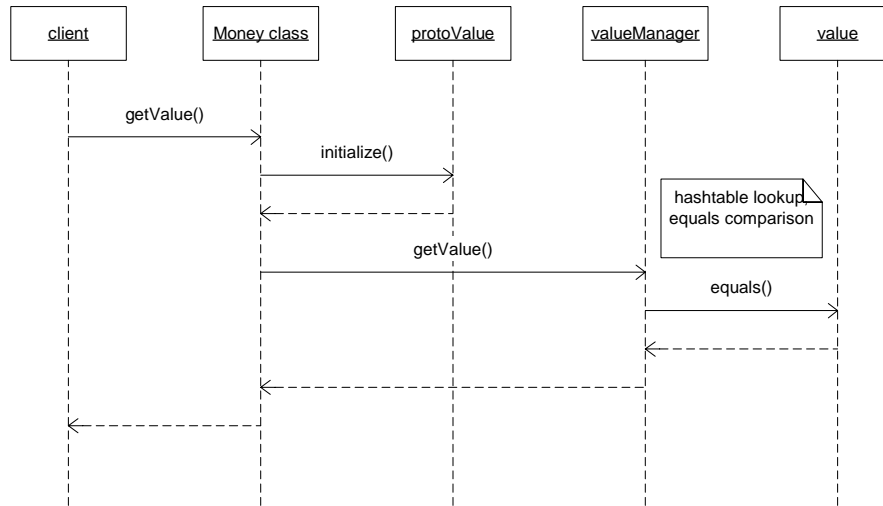
## ValueManager

- 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

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

60 of 143

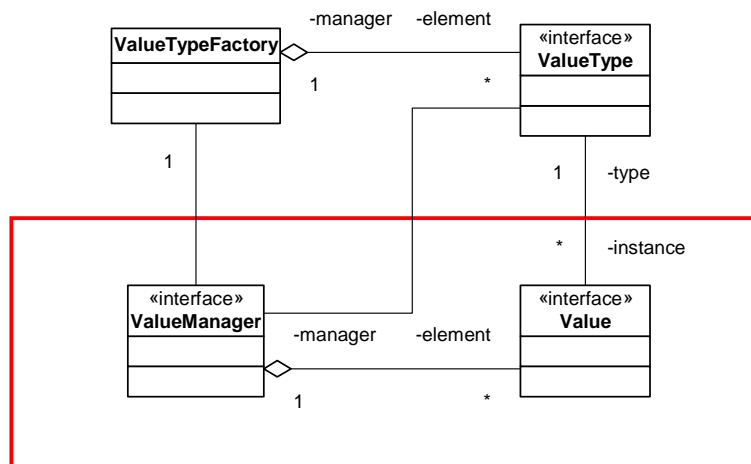
## Value Object Retrieval/Creation



Framework Design and Implementation using Java and UML, Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

61 of 143

## Design Aspect: ValueManager



Framework Design and Implementation using Java and UML, Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

62 of 143

## ValueManager: Interface

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

    public boolean hasValue(Value value);
    public Value getValue(Value protoValue);
    public void removeValue(Value value);

    public ProtoValue getProtoValue();

    public Value getDefaultValue();
    public void setDefaultValue(Value value);

    public Value getUndefinedValue();

    public ValueType getValueType();

    public int getNoValues();
    public Iterator values();
}
```

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

63 of 143

---

---

## ValueType

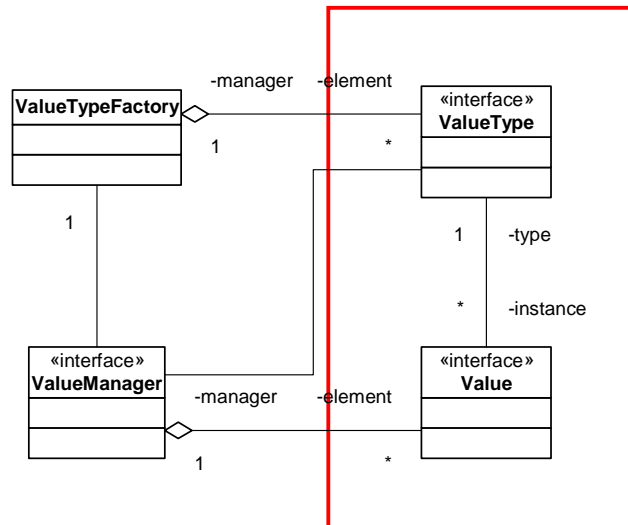
- ValueType instance
  - represents value type (class)
  - provides type information for one value type
  - centralizes all type information common to that specific value type
  - is configured at runtime
- ValueType provides operations
  - to access the type name
  - to determine whether value type 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.)

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

64 of 143

---

## Design Aspect: ValueType



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

65 of 143

## ValueType: Interface

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

    public boolean isDecorator();
    public boolean isIdentical(ValueType valueType);

    public boolean hasFiniteCardinality();
    public void setFiniteCardinality();

    public Iterator attributes();
    public Iterator allAttributes();
    public void addAttribute(String attrName, ValueType attrType);
    public void insertAttribute(String attrName, ValueType attrType, int position);
    public void removeAttribute(String attrName);

    public boolean isSupertypeOf(ValueType valueType);
    public ValueType getSupertype();
    public void setSupertype(ValueType valueType);

    public boolean isEquivalent(ValueType valueType);
    public ValueType getEquivalentType();
    public void setEquivalentType(ValueType valueType);

    ...
}
```

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

66 of 143

## ValueTypeFactory

- ValueTypeFactory
  - creates and manages 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

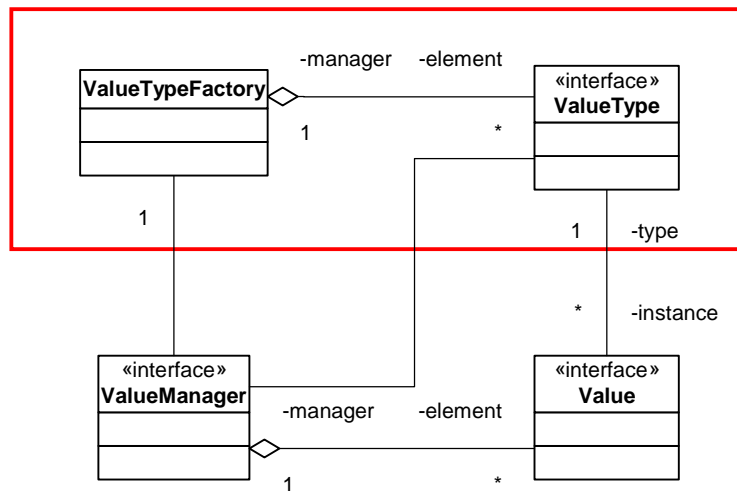
Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

67 of 143

---

---

## Design Aspect: ValueTypeFactory



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

68 of 143

---

## ValueTypeFactory: Class Definition

```
public class ValueTypeFactory {  
    protected Hashtable fValueTypes;  
    protected static ValueTypeFactory sInstance = null;  
  
    protected ValueTypeFactory() { ... }  
  
    public ValueType createValueType(String name) { ... }  
    public ValueType ensureValueType(String name) { ... }  
  
    public boolean hasValueType(String name) { ... }  
    public ValueType getValueType(String name) { ... }  
    public synchronized void addValueType(ValueType vt) { ... }  
  
    public Enumeration getValueTypes() { ... }  
  
    public static final ValueTypeFactory getInstance() { ... }  
}
```

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

69 of 143

---

---

## Topic 5: Separation of Concerns

- JValue design complexity
- UML collaboration specifications

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

70 of 143

---



## Role Modeling

- Definition: Role
  - a role is an observable behavioral aspect of an object
  - an object can play multiple roles at once, each one in a different context
  - a role may be fulfilled by one or more different objects over time (m:n)
  - every object must keep its roles integrated
  - ⇒ called *Classifier Role* in UML
- Definition: Role Model
  - a role model describes how objects playing roles collaborate for one purpose
  - roles are tied together by one shared purpose
  - object is irrelevant---only the role counts
  - ⇒ called *Collaboration Specification* in UML

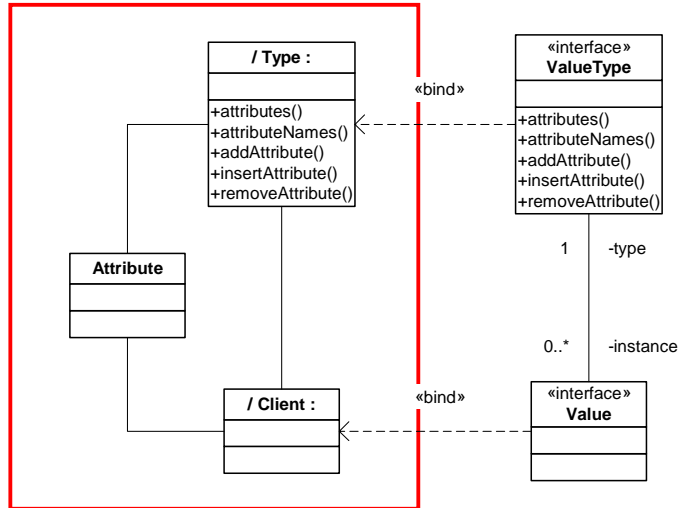
---

---

## UML Collaboration Specifications

- Definition: Collaboration Instance
  - set of objects that collaborate by playing roles
  - Booch: "behavioral part of a collaboration"
  - UML 1.x: "collaboration on an instance level"
- Definition: Collaboration Specification
  - set of classifiers that relate to each other through classifier roles
  - allows for regular classes and interfaces to be part of specification
  - Booch: "structural part of a collaboration"
  - UML 1.x: "collaboration on a specification level "
- Unfortunately, all still quite vague...

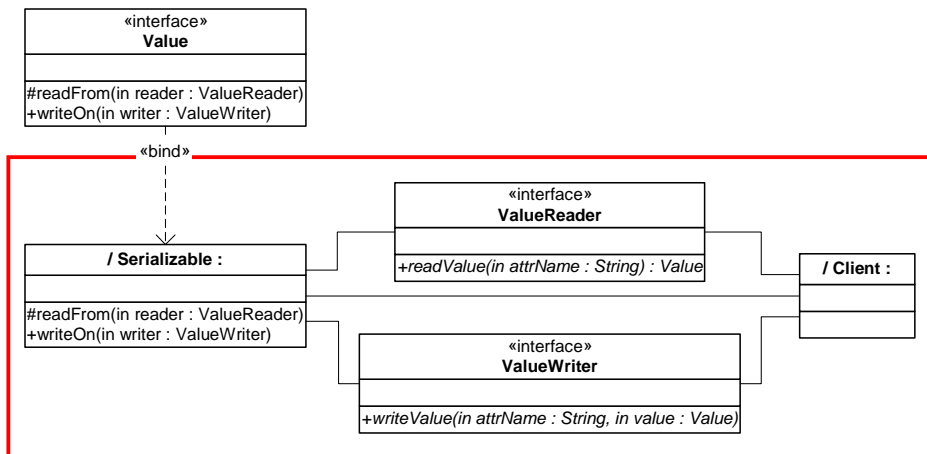
## Example: Value Type Collaboration



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
 Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

75 of 143

## Example: Serialization Collaboration



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
 Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

76 of 143

## Value Collaboration Integration

- Value binds multiple classifier roles
  - Client role of Value Type collaboration
  - Serializable role of Serialization collaboration
  - many others (not shown here)
- Value integrates roles
  - implementation of Value integrates multiple roles
  - example: reading/writing uses type-level information
- Collaboration specification associations
  - map on associations, observing multiplicities

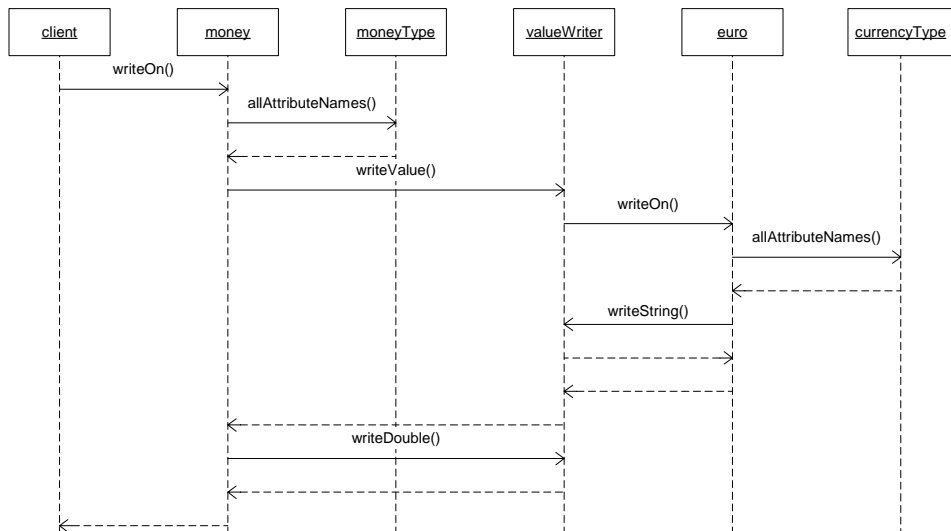
Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

77 of 143

---

---

## Integration Value Type with Serialization



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

78 of 143

---

## More Collaboration Examples

- Client/Service collaboration specifications
  - basically every class/interface provides its “default” functionality
  - for example, Value offers asDataString, isDefaultValue, etc.
- Element/Manager collaboration specifications
  - ValueManager manages Value objects
  - ValueTypeManager manages ValueType objects
  - ValueManagerHome manages ValueManager
- ValueType/Decorator collaboration
  - Value types can be wrapped by decorator

## Even More Collaboration Specifications

- Singleton collaboration specifications
  - ValueTypeManager class provides ValueTypeManager instance
  - ValueManagerHome class provides ValueManagerHome instance
  - JValueLogger class provides JValueLogger instance
- Implicit collaboration specifications inherited from Object
  - Hashtable collaboration (equals, hashCode)
  - Comparable (simple comparison function)
  - Cloneable (for cloning objects)
  - Serializable (default Java serialization)

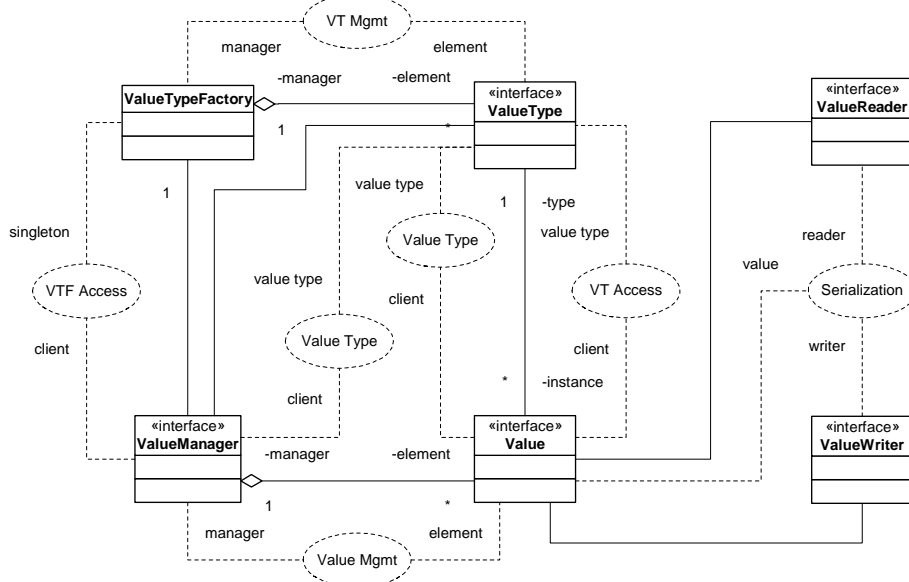
## Benefits of Collaboration Specifications

- Reduce complexity
  - by breaking up complex (class) interfaces into parts
- Increase reuse
  - by separating classifier 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 19% of all cases
- Variant 3: As a role object class
  - lets you dynamically attach roles at runtime
  - accounts for about 1% of all cases

## UML: Ellipse Shorthand



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

83 of 143

## Topic 6: Design Patterns in Frameworks

- Design patterns
- Design pattern annotations
- Design patterns using UML collaboration specifications
- Design pattern density

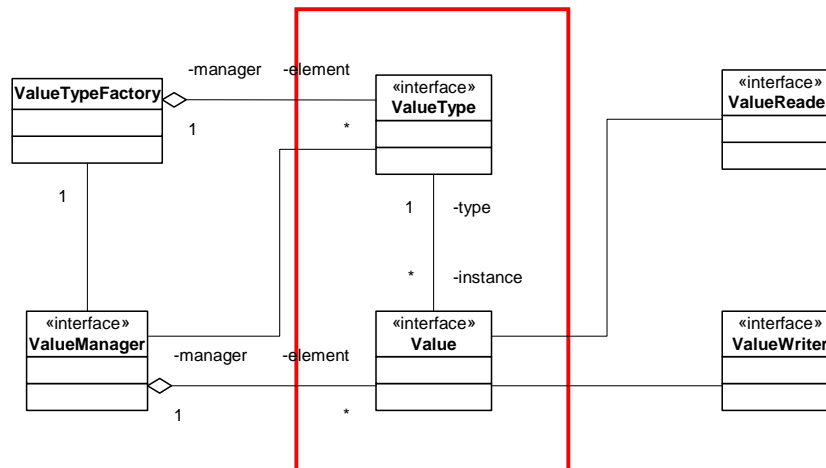
Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

84 of 143

## 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

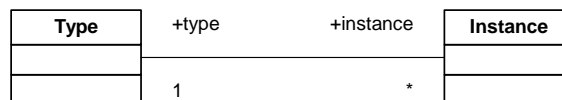
## JValue Design: Type Object Applied



## Type Object Pattern: Description

- 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.)

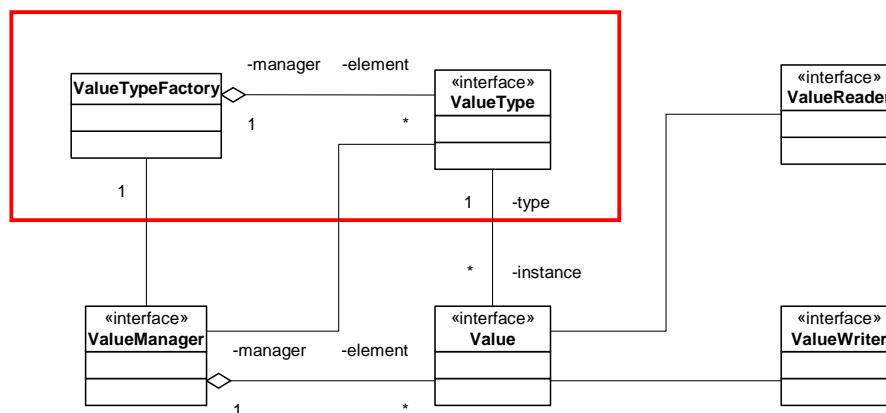
## Type Object Pattern: Structure Diagram



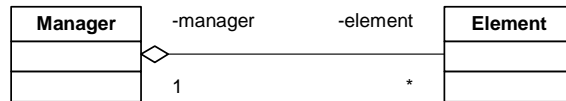
## Type Object: Applicability

- 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
  - ...

## JValue Design: Manager Applied



## Manager Pattern: Structure Diagram



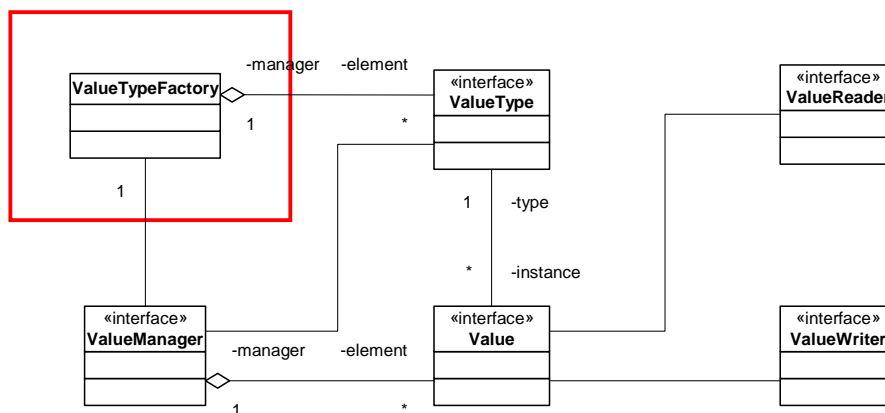
Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

91 of 143

---

---

## JValue Design: Singleton Applied

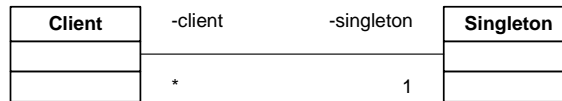


Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

92 of 143

---

## Singleton Pattern: Structure Diagram



Framework Design and Implementation using Java and UML, Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

93 of 143

---

---

## Design Patterns: Levels of Abstraction

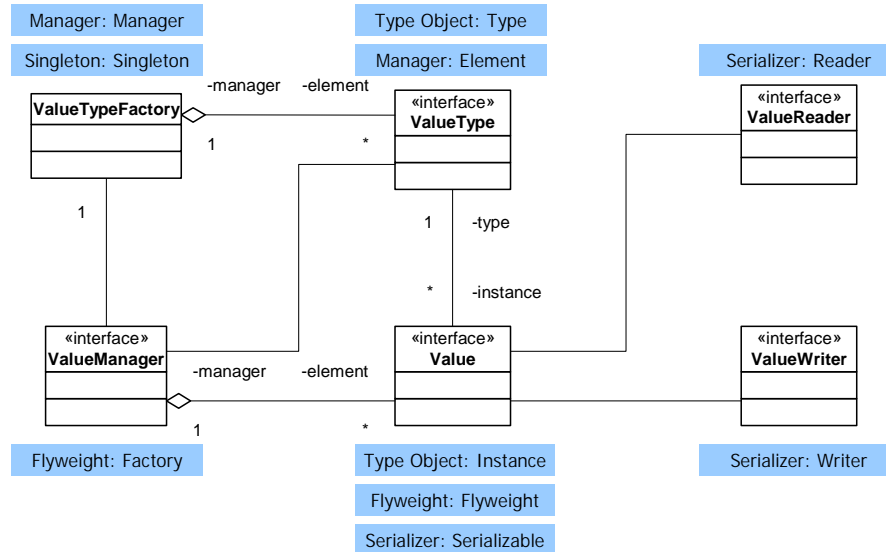
- Design pattern
  - an idea of a problem solution
  - infinite and unconstrained variation in implementation
- Design template
  - a template for a specific design solution
  - infinite but constrained variation in implementation
  - formal variant: you can instantiate it by supplying parameters
  - informal variant: you copy and paste and adjust the structure
- Design fragment/aspect
  - a specific design solution applied in the context of a larger system
  - exactly one implementation

Framework Design and Implementation using Java and UML, Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

94 of 143

---

## JValue Design with Pattern Annotation



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

95 of 143

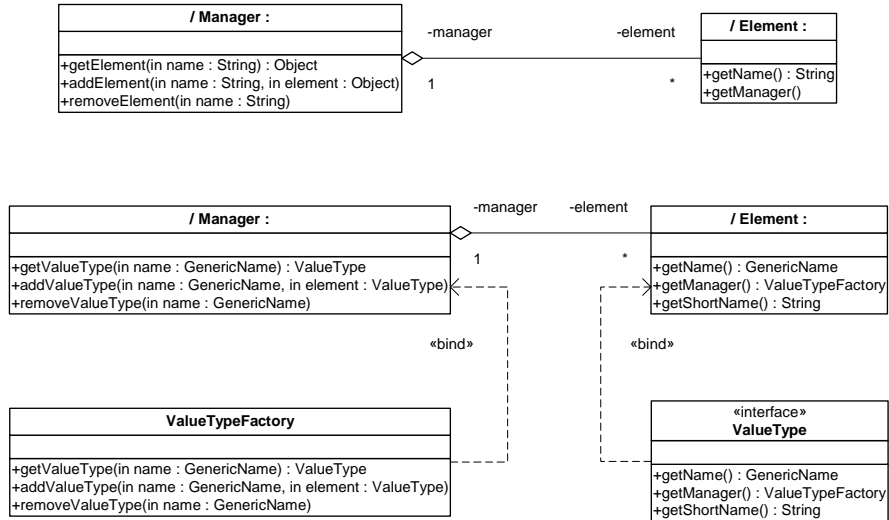
## Patterns As Collaboration Specifications

- Collaboration specifications
  - work well for role modeling problems (Observer, Type Object, Manager)
  - don't work well for static class structure problems (Singleton, Null Object)
- Use collaboration specifications as informal template
  - a collaboration specification is not a design pattern
  - one collaboration specification can serve as illustration of one common case
  - use of collaboration specification as formal template is undefined
- Do as the Design Patterns book says...
  - just use UML collaboration specifications where book uses OMT class model

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

96 of 143

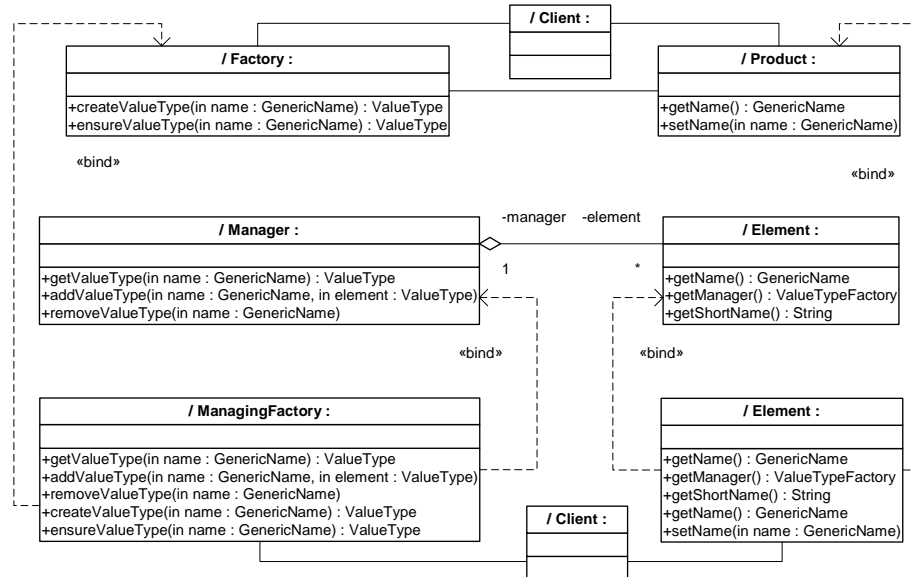
## Manager Pattern: Levels of Abstraction



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
 Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

97 of 143

## Compound Collaboration Specifications?



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
 Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

98 of 143

## Compound Design Patterns

- Definition: Compound Design Pattern
  - is first of all a design pattern
  - is best understood as composition of other design patterns
  - the whole compound design pattern is more than the sum of its patterns
- Examples of compound design patterns
  - Managing Factory (as in previous example)
  - Bureaucracy (as in GUIs), Active Bridge (as in System wrappers)
  - See literature references at the end

## JValue Design: High Pattern Density

- Value Object, Flyweight, Immutable Object
- Type Object
- Manager (n\*), Prototype
- (Abstract) Singleton (n\*)
- AbstractFactory (n\*)
- Null Object (n\*)
- Serializer
- Decorator (2\*)
- Adapter
- ...

# Part IV: Frameworks as Components

## Part IV: Topics

- Interface architecture and framework use
- Extension architecture and framework extension
- Framework types
- Framework evolution
- Framework packaging

## Frameworks as Components

- Component types
  - logical runtime components: object conglomerates
  - code components: assemblies of classes
- Framework as design and code component
  - assembly of classes and interfaces
  - membership defined through framework boundary
- How to define a framework's boundary?

## Topic 7: Interface Architecture and Framework Use

- Interface architecture
- Framework use-clients
- Free classifier roles

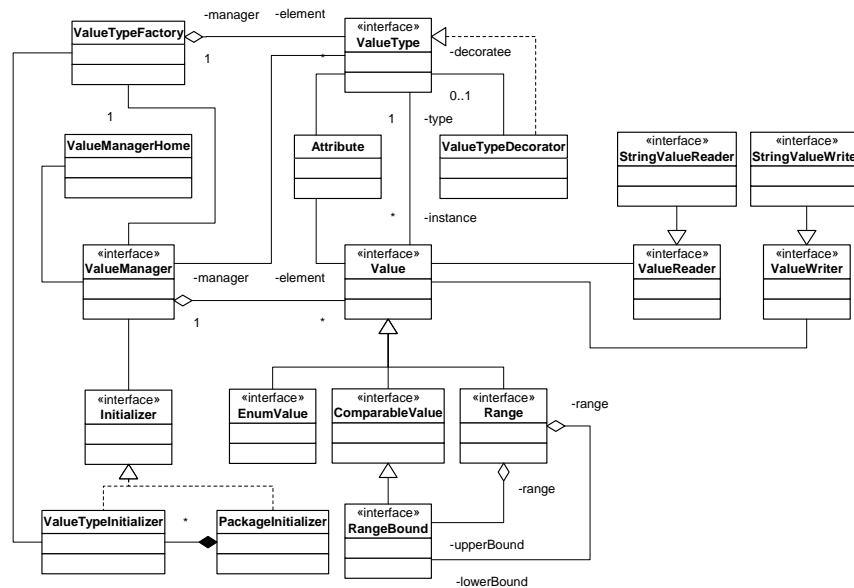
## Interface Architecture

- Definition: Interface Architecture
  - set of interfaces that determines overall structure and behavior
  - prescribes architecture for given domain or aspect thereof
- Like abstract design
  - less abstract implementation classes
- How to interact with use-clients?

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

105 of 143

## JValue: Interface Architecture



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

106 of 143

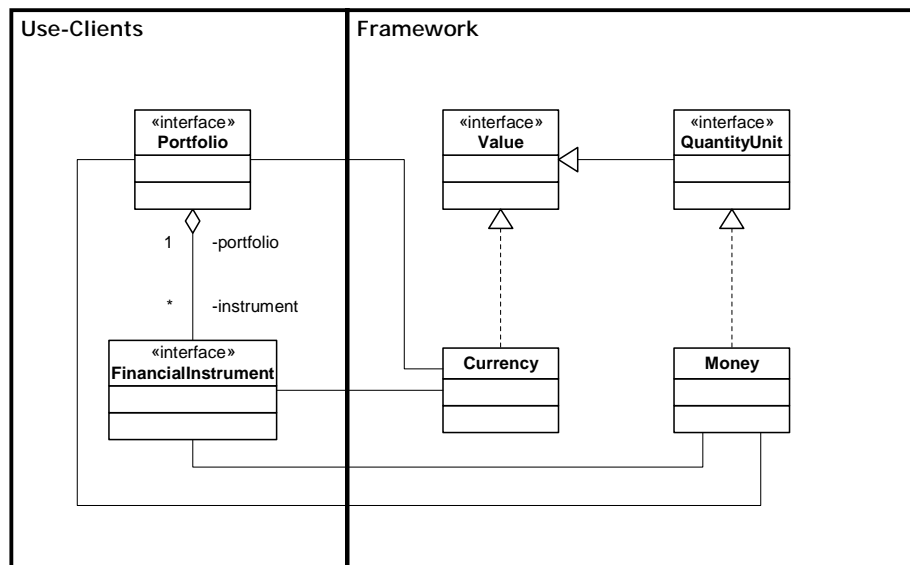
## Framework Use by Use-Clients

- Use-clients create framework objects
  - from framework classes (if available)
  - from framework extension classes (if available)
  - using a variety of object creation patterns
- Use-clients compose framework objects

Framework Design and Implementation using Java and UML, Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

107 of 143

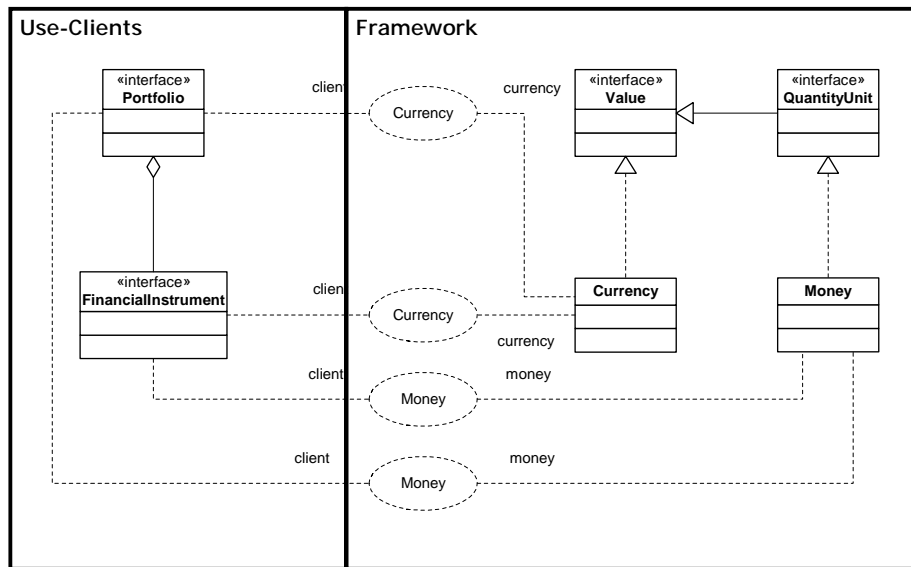
## Example Use-Clients: Finance



Framework Design and Implementation using Java and UML, Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

108 of 143

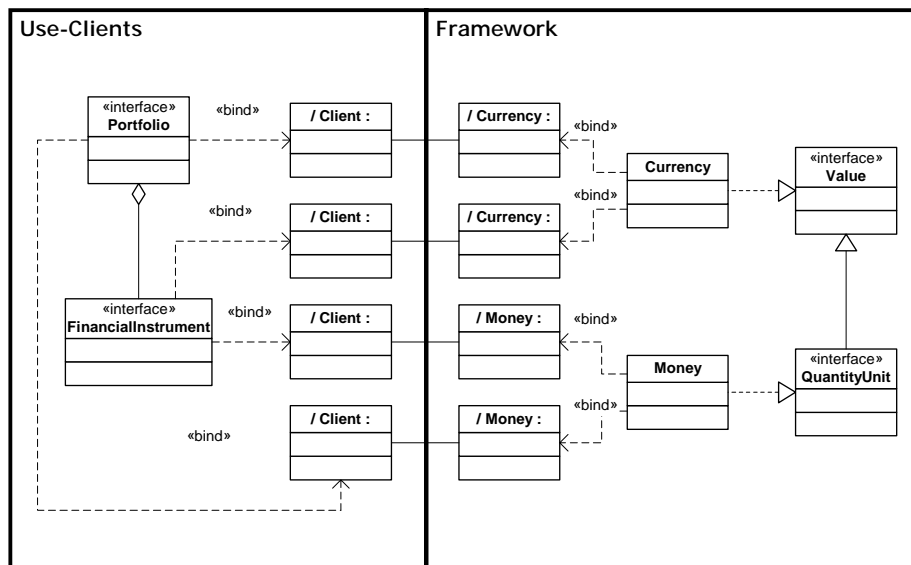
## Use-Clients: With Bound Collaborations



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
 Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

109 of 143

## Use-Clients: With Detailed Collaborations

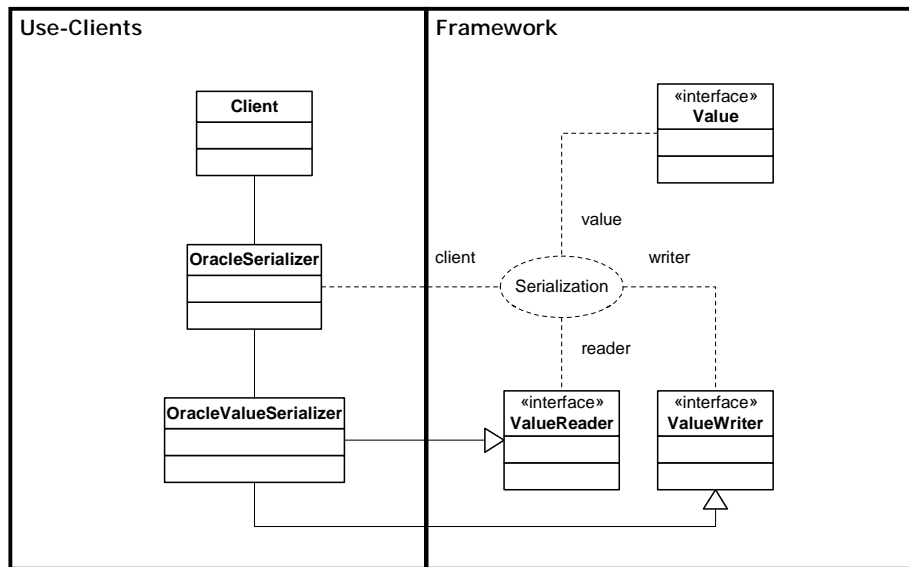


Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
 Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

110 of 143



## Example: Binding to Database



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

113 of 143

---

---

## Topic 8: Extension Architecture and Framework Extension

- Framework extension points
- Extension architecture
- Framework extension classes

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

114 of 143

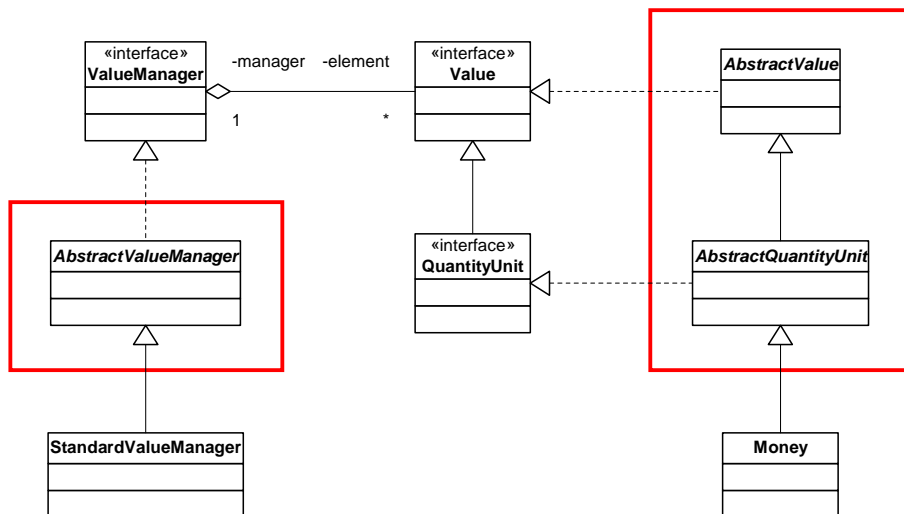
## Framework Extension Points

- Definition: Extension Point
  - a framework interface that may be implemented by external classes
  - a framework class that may be subclassed by external classes
  - if class, it defines inheritance interface to framework extensions
- Definition: Extension Class
  - class that implements or subclasses an extension point
  - makes use of a framework's inheritance interface
- Similar to Hot Spots

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

115 of 143

## JValue: Extension Point Classes



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

116 of 143

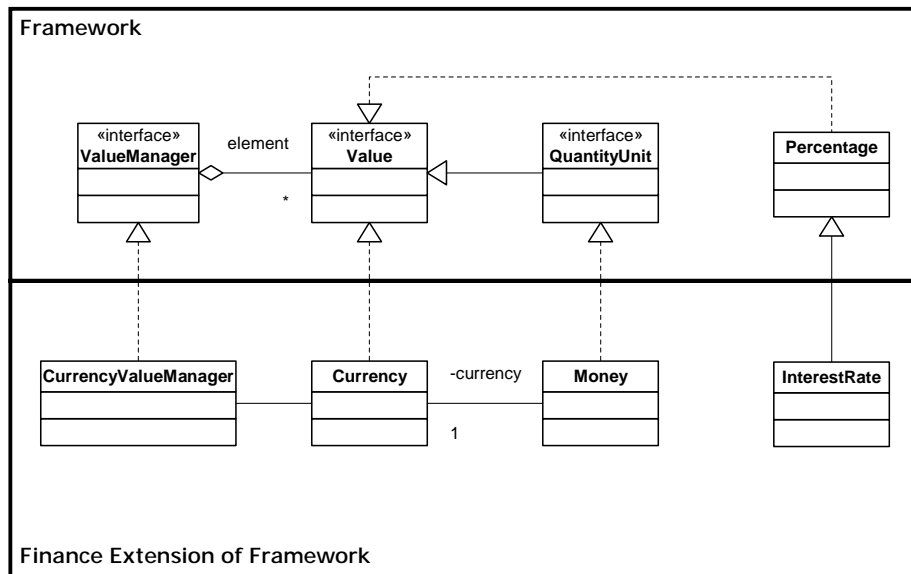
## Framework Extension Architecture

- Definition: Extension Architecture
  - complete set of framework extension points
- How to extend a framework?

## Framework Extension

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

## Framework Extension: Finance



Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

119 of 143

---

---

## Topic 9: Framework Types

- Framework definition
- White-box frameworks
- Black-box frameworks
- Gray-box frameworks

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

120 of 143

---

## Framework Definition (Extended)

- Definition: Framework (traditional)
  - a reusable design and implementation that covers a particular domain
- Definition: Framework (extended)
  - a reusable design and implementation that covers a particular domain
  - provides an interface architecture that defines how to use it
  - provides an extension architecture that defines how to extend it

---

---

## Framework Boundaries

- What's in and out is defined by design
- Collaboration specifications define where and how to use
- Extension points define where and how to extend

## White-Box Frameworks

- Definition: White-Box Framework
  - a framework that provides well-defined extension points
  - first, extension clients use inheritance to customize framework
  - then, use-clients compose objects to apply framework
- Examples
  - Java Object framework

## Black-Box Frameworks

- Definition: Black-Box Framework
  - a framework that provides readily usable classes
  - the framework has well-defined interface architecture
  - 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
- Classification is not so important
- More important is how to use a framework

## Topic 10: Framework Evolution

- Backward compatibility
- General evolution techniques
- Evolving the inheritance interface

## Backward Compatibility?

- When to provide backward compatibility
  - if your changes affect use clients
  - if your changes affect extension clients
  - if you don't control client code
  - if client code base is too big (even if you control it)
- Two-step evolutionary process
  - provide backward compatibility for one release
  - gives clients/users chance to upgrade while system is not broken
- N-step evolutionary process
  - difficult, but not always necessary

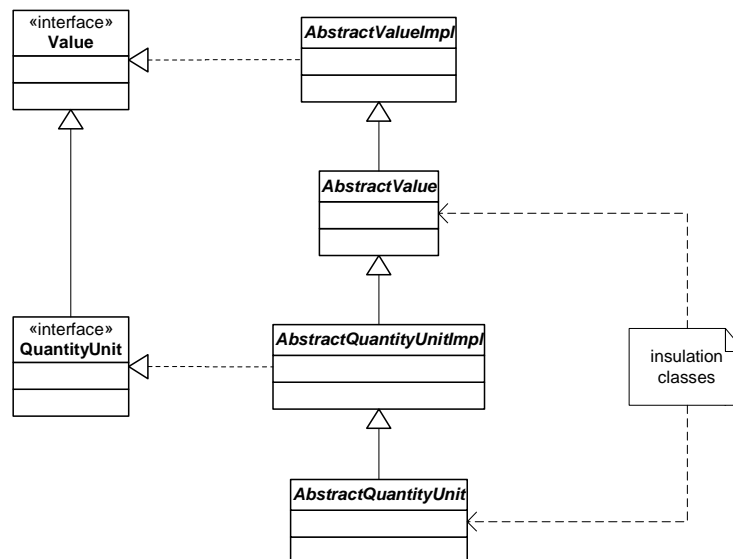
## General Evolution Techniques

- Deprecated methods
  - signal to clients that code needs to be evolved
  - in two-step process, second step removes deprecated methods
- Wrappers
  - obsolete interfaces get all methods marked deprecated
  - old implementations are changed to delegate to new implementation
  - old interfaces get new implementations that delegate to new interfaces
  - in two-step process, second step removes old interfaces and implementations

## Evolving the Inheritance Interface

- Definition: Insulation class
  - a class that redirects the inheritance interface
  - typically injected into class hierarchy beforehand (empty)
- Use of insulation class
  - changes to inheritance interface activate use of insulation class
  - old inheritance interface methods are marked deprecated
  - insulation class implements deprecated methods to new methods
  - in two-step process, second step clears out insulation class

## Example: Insulation Classes



## Topic 11: Framework Packaging

- Code packaging

## Code Packaging

- Frameworks as code components
  - class assembly, membership defined by design
- Use Java jars to package components
  - frameworks and framework extensions

## Part V: Summary

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

133 of 143

---

---

### Summary Parts I-IV

- Business benefits
- Code reuse in frameworks
- Design reuse in frameworks
- Frameworks as components

Framework Design and Implementation using Java and UML. Seattle, WA: OOPSLA 2002.  
Copyright 2002 by Dirk Riehle and Alan Perry. All rights reserved. Contact: dirk@riehle.org, aperry@skyva.com.

134 of 143

---

## JValue: A Java Value Object Framework

- JValue for Java Value Objects
  - available using the LGPL 2.0
  - contributions welcome!
- Please see: <http://www.jvalue.org>

## Appendix: 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>

## 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/software-industry-work/publications.html>

## Value Objects

- Dirk Bäumer et al. Values in Object Systems. *Ubilab Technical Report 98-10-1*. Switzerland, UBS AG: 1998.  
<http://www.riehle.org/computer-science-research/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>

---

---

## 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/practical-matters/patterns/index.html>

## 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*.

## 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.

## Frameworks

- Ted Lewis et al. *Object-Oriented Application Frameworks*. Prentice-Hall, 1995.
- Taligent Press. *The Power of Frameworks*. Addison-Wesley, 1995.
- Mohamed Fayad et al. *Various Volumes*. Wiley, 1999.
- Dirk Riehle. *Framework Design: A Role Modeling Approach*. Dissertation, ETH Zürich, 2000.
- <http://www.riehle.org/computer-science-research/dissertation>