Ch. 5.1: Part-A Advanced Structural Specification in UML

H.S. Sarjoughian

CSE 460: Software Analysis and Design

School of Computing, Informatics and Decision Systems Engineering
Fulton Schools of Engineering

Arizona State University, Tempe, AZ, USA

UML Class Diagram Revisited

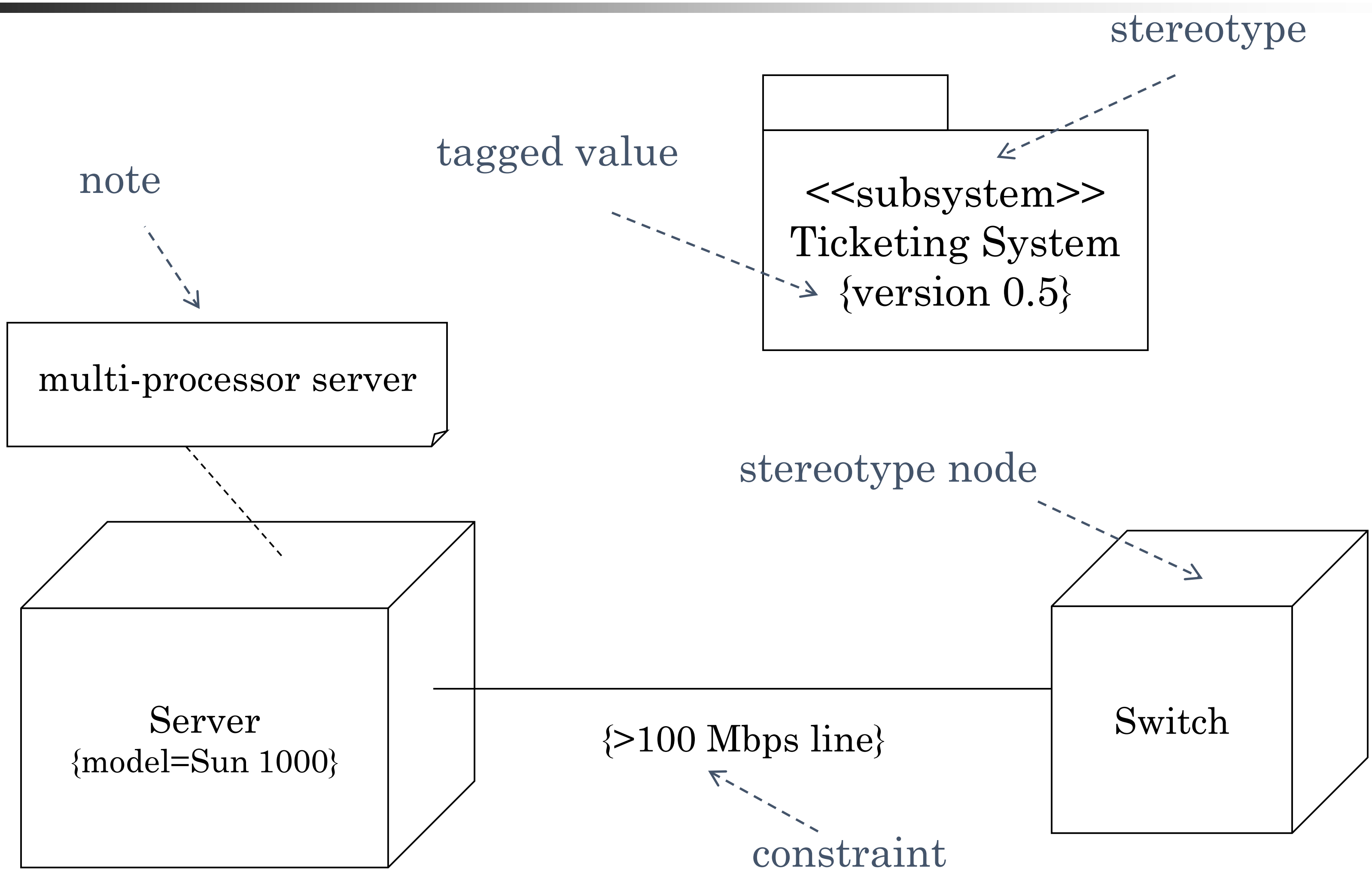
- Class diagrams are useful for modeling static views of a system
  - Each class diagram helps analyst and designer in visualization, specification, and documentation of a set of classes, interfaces, and other elements and their relationships
  - Each class diagram captures one collaboration at a time
  - Generally a family of class diagrams are necessary
- Class diagrams are the basis for constructing executable systems
- Class diagrams are the basis for component and deployment diagrams

Class Diagram (cont.)

Class diagram generally contains

- Class diagram may contain classes, interfaces, relationships (association, dependency, generalization, inheritance), packages and subsystems, and collaborations
- **Notes:** can be used to add descriptions, observations, requirements
- **Constraints:** can be used to create new rules for UML elements – extends semantics of UML elements
- **Tagged values:** can be used to create new information about UML elements – extends properties of UML elements
- **Stereotypes:** can be used to create new elements for specific domains – extends vocabulary of the UML

Notes, Tagged Values, and Constraints



Abstract, Root, and Leaf Classes

Abstract class

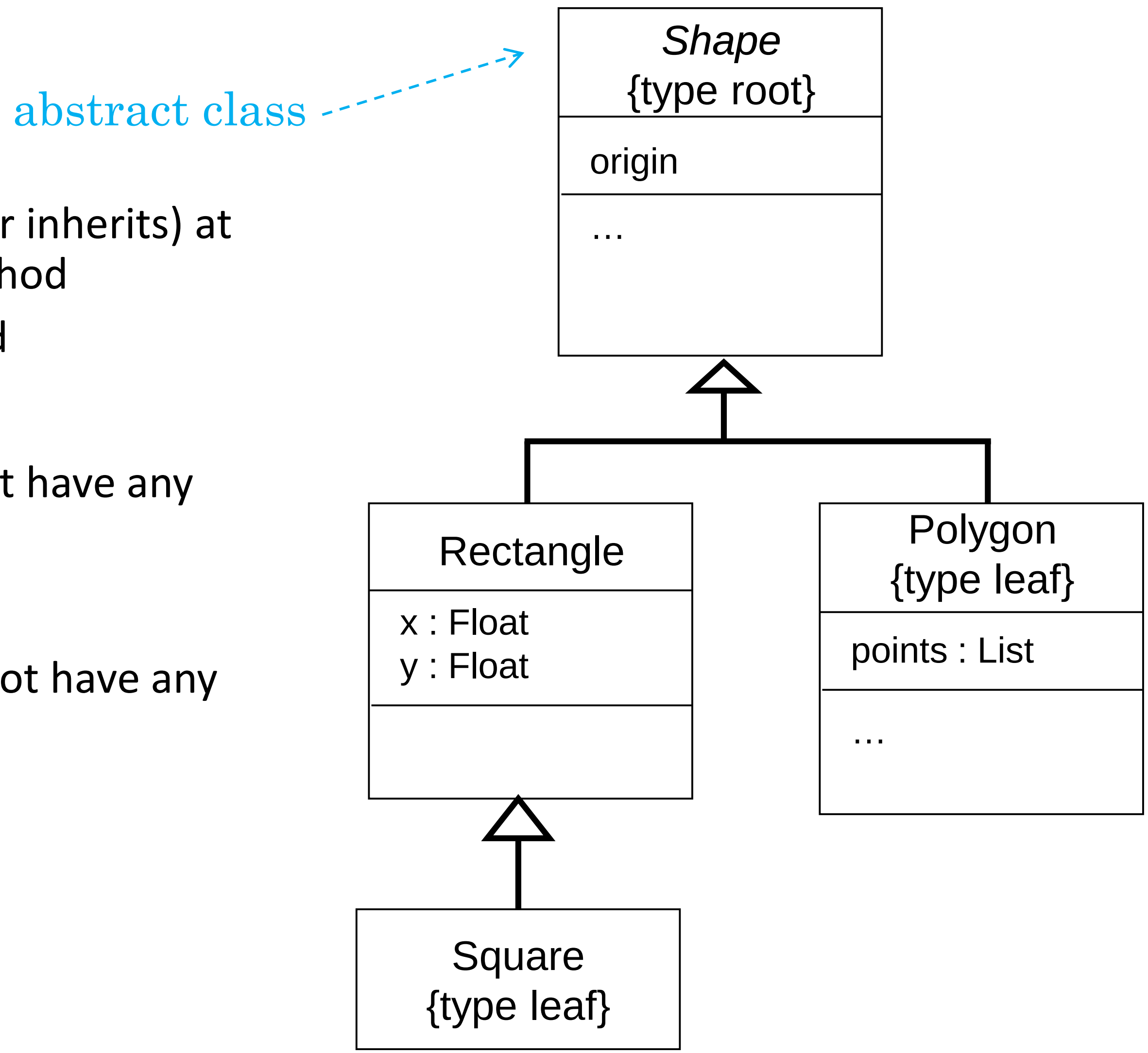
- a class that includes (or inherits) at least one abstract method
- can not be instantiated

Root class:

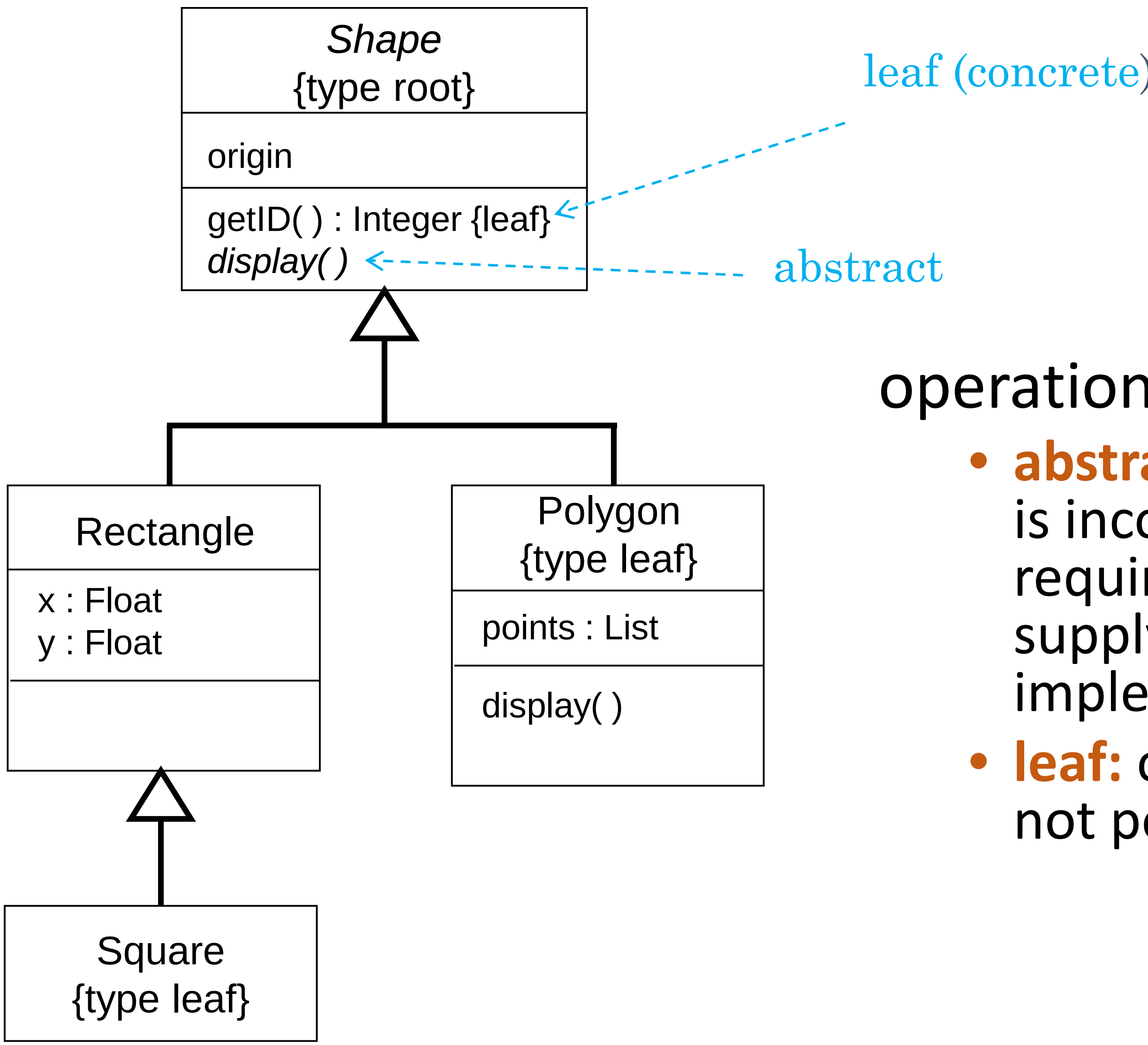
- any class which can not have any parent class

Leaf class:

- any class which does not have any children



Abstract, Leaf, and Polymorphic Operations



operations

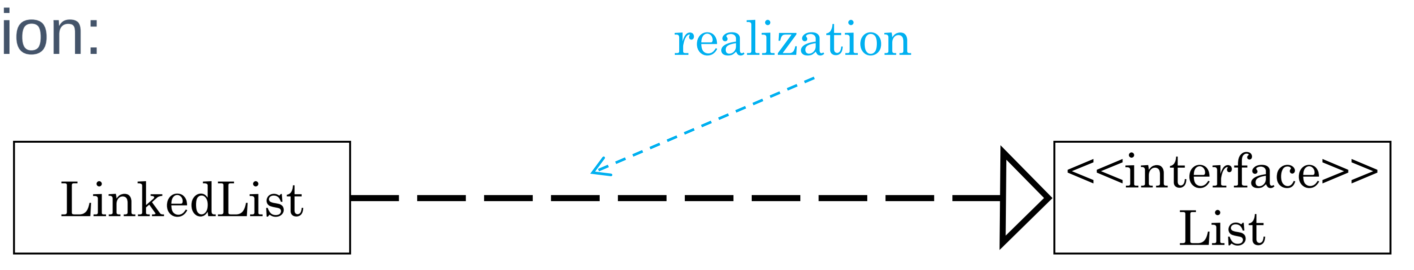
- **abstract:** operation is incomplete requiring a child supplying its implementation
- **leaf:** operation is not polymorphic.

Interface

Semantics

- an interface is a collection of operations that can be used to specify a service of a class
  - An interface is a special form of a class which specifies methods without their implementations (can not have direct instances) – interface does not include attributes and method implementations
- an interface can have a realization relationship with a class
  - Realization relationship shows that a class carries out (realizes) the contracts of an interface
- an interface can have generalization relationship with another interface
- generally an “I” is appended to the front of the interface name

notation:



Interface Example: List and LinkedList

```
public interface Collection<E> extends Iterable<E>
```

```
public interface List extends Collection

void add(int index, Object element);
        // inserts the specified element at the specified position in this list

Object remove(int index) ;
        // removes the element at the specified position in this list
```

```
public class LinkedList extends AbstractSequentialList
implements List, Cloneable, java.io.Serializable

public void add(int index, Object element) {
   addBefore(element, (index==size ? header : entry(index)));}

public Object remove(int index) {
   Entry e = entry(index);
   remove(e);
   return e.element;}
```

Interface Example: List and LinkedList

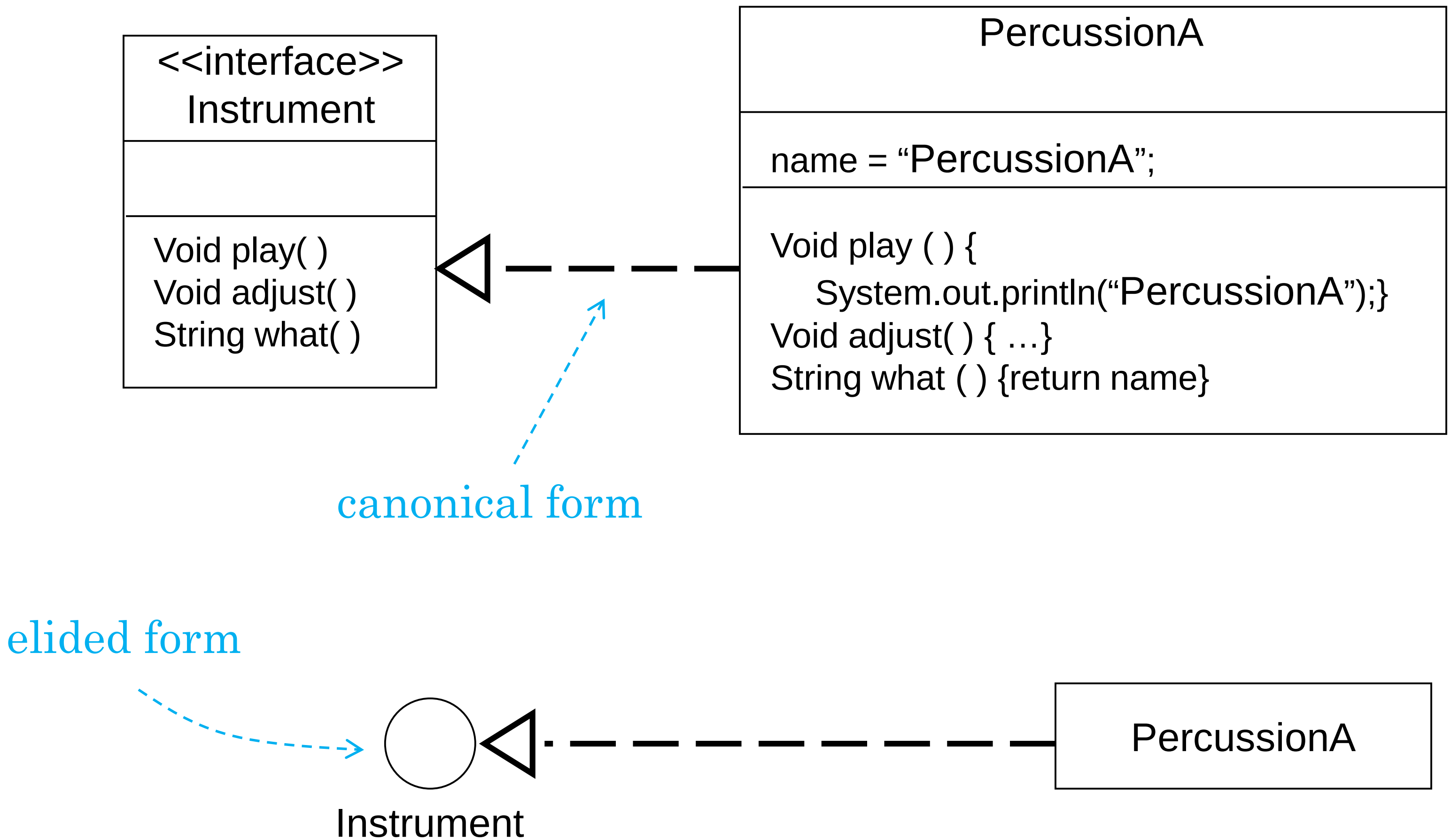
public interface Collection<E> extends Iterable<E>

Java Platform SE 11.0

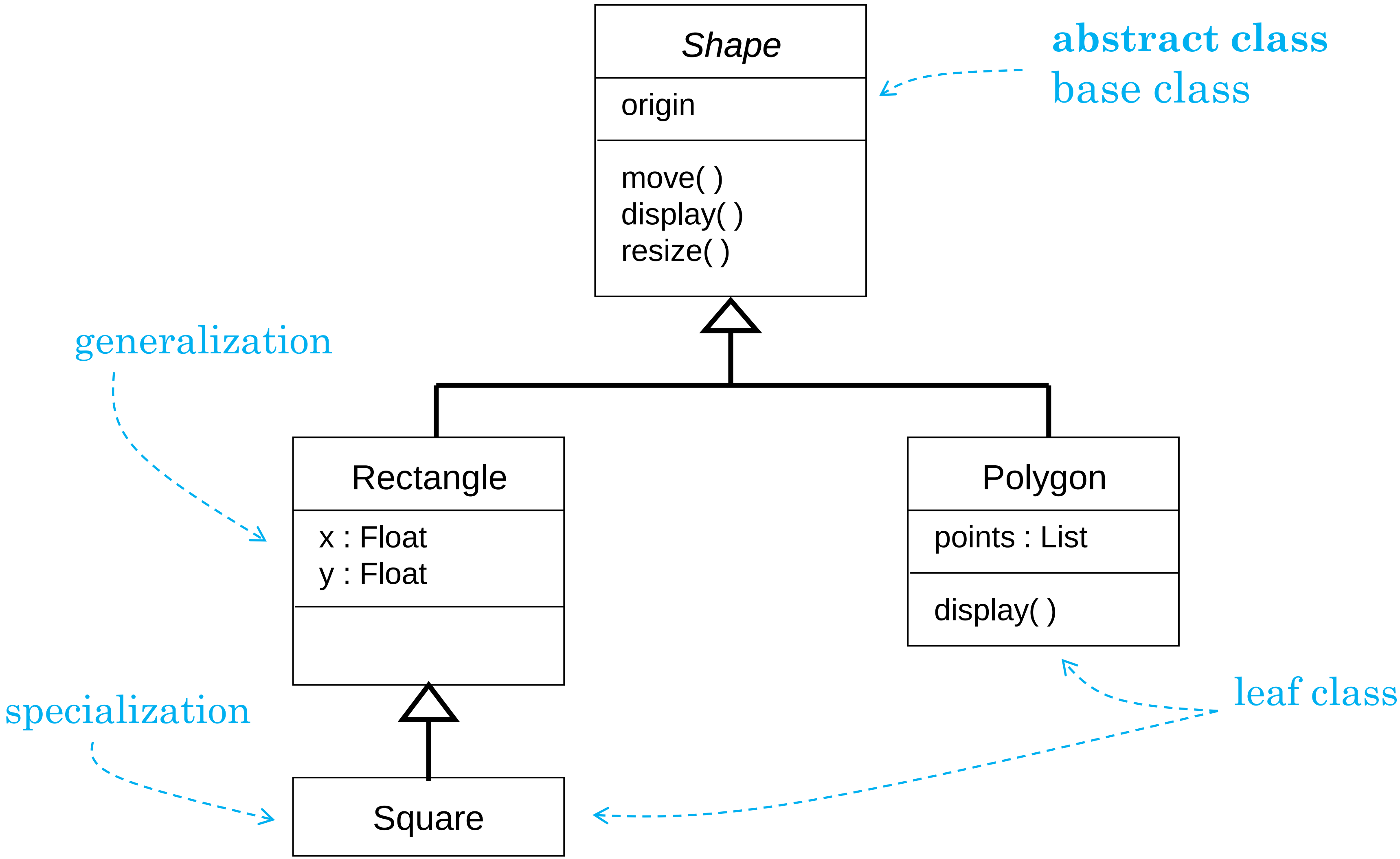
```
public interface List extends Collection

void add(int index, E element); // inserts the specified element at the specified position in this list

E remove(int index);         // removes the element at the specified position in this list
```

```
public class LinkedList<E> extends AbstractSequentialList<E>
   implements List<E>, Deque<E>, Cloneable, java.io.Serializable

public void add(int index, E element) {
   checkPositionIndex(index);
   if (index == size) linkLast(element);
   else linkBefore(element, node(index));}

public E remove(int index) {
   checkElementIndex(index)
   return unlink(node(index));}
```

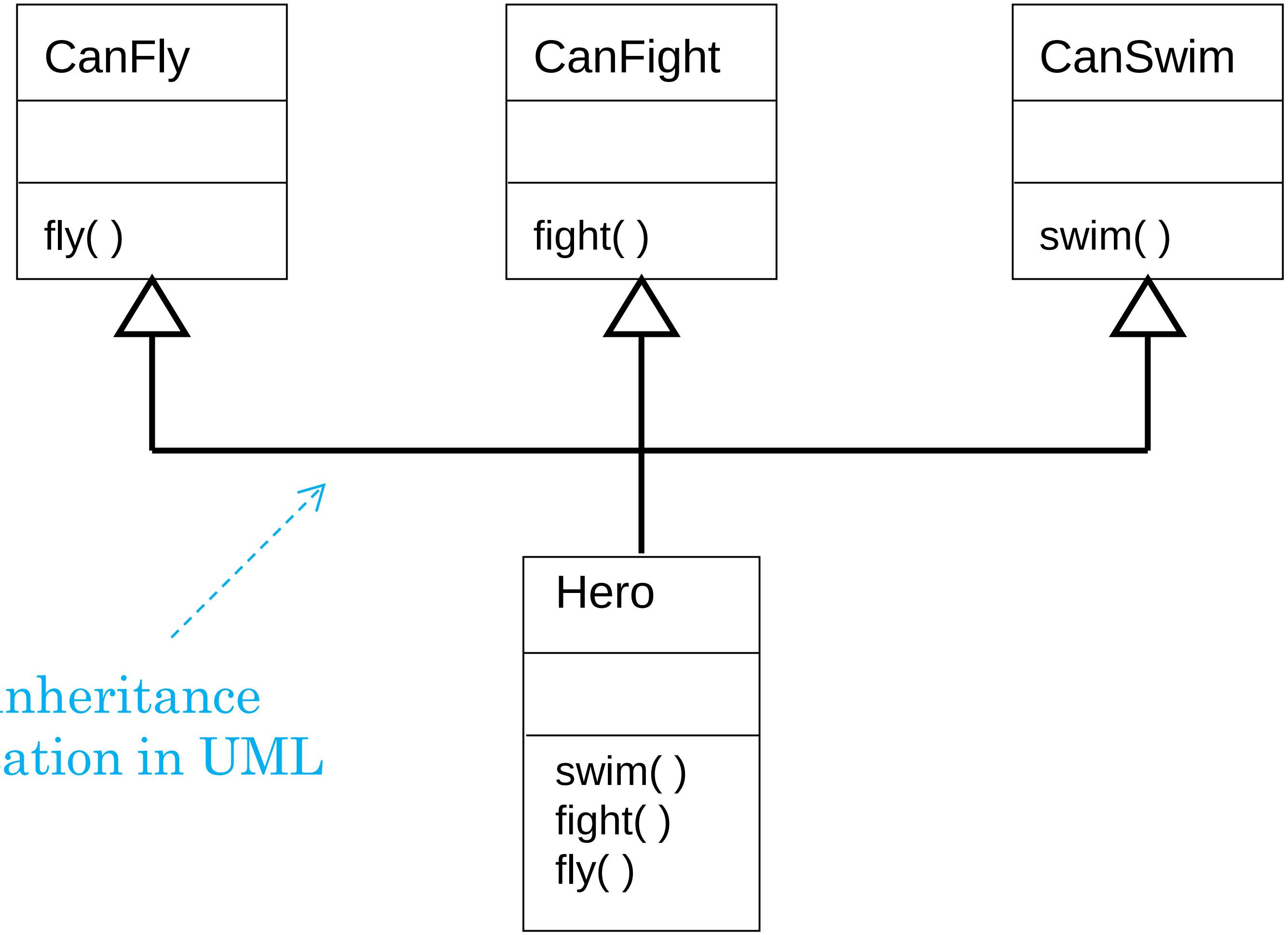
UML notation



Single Inheritance

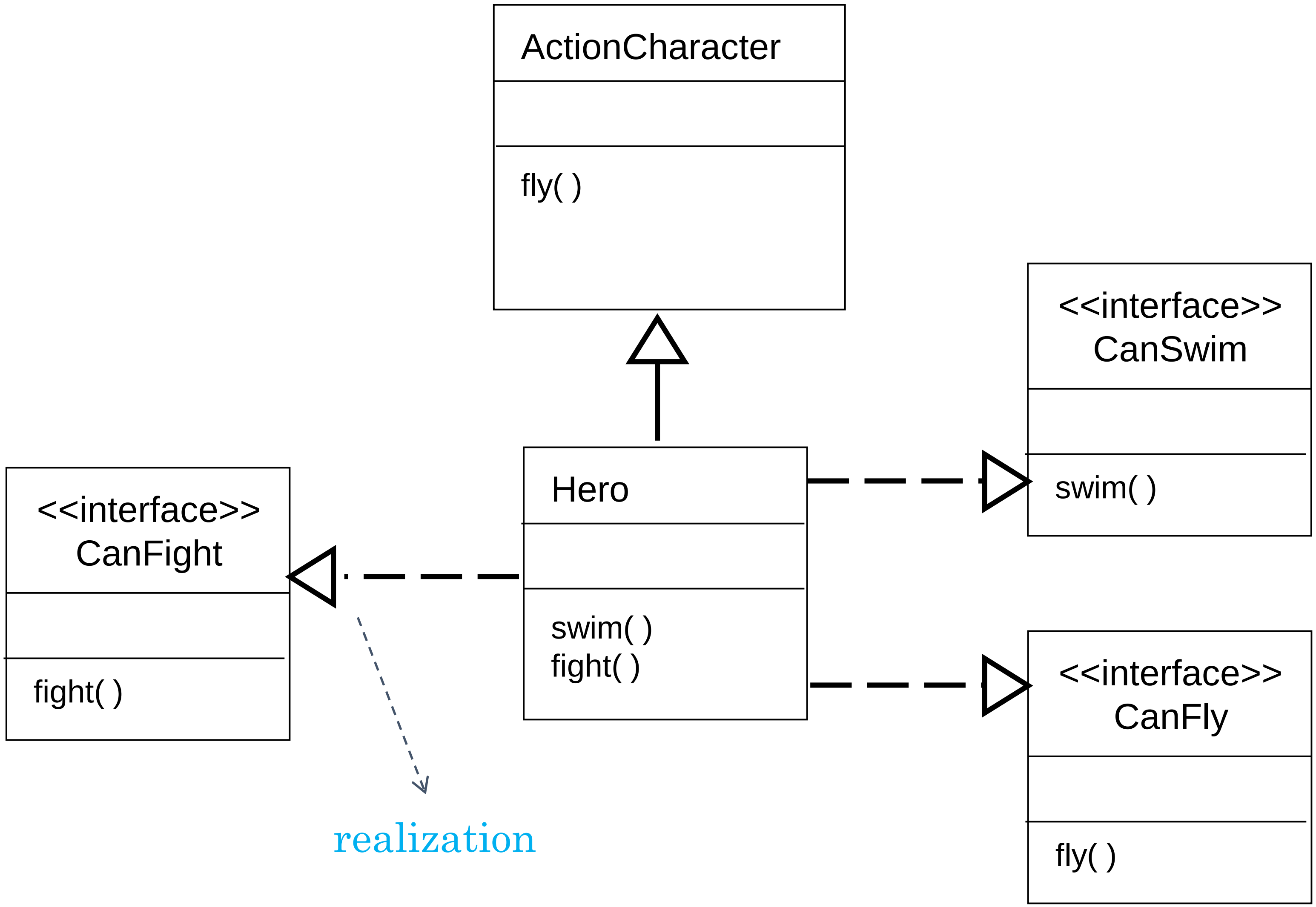


Multiple Inheritance



multiple inheritance
representation in UML

Interface Example in Java



UML Class Diagram Revisited (cont.)

Class diagrams can be used in three ways

- Model vocabulary of a system
  - Focuses on key abstractions clearly relevant to a specific system analysis or design view while excluding all others – refer to Ch4 OOAD
- Model collaborations
  - A collaboration is a society of classes, interfaces, and other elements (e.g., subsystems) that collectively provide some behavior that is bigger than the sum of all the elements
- Model a logical database schema
  - Specify design of databases

Basic Class Diagram Modeling

Modeling Procedure

- Identify mechanisms of interest – each represents some function or behavior for a part of system being modeled
- Identify classes, interfaces, and other elements that participate in this collaboration
- Identify relationships that are expected to support the collaboration
- Specify the responsibilities each element is expected to provide
- Use scenarios to examine the elements and their relationships. Revise the choice of elements and their relationships to achieve the intended behavior

it is very important to develop **use-case** and **CRC** diagrams before beginning to specify class diagrams

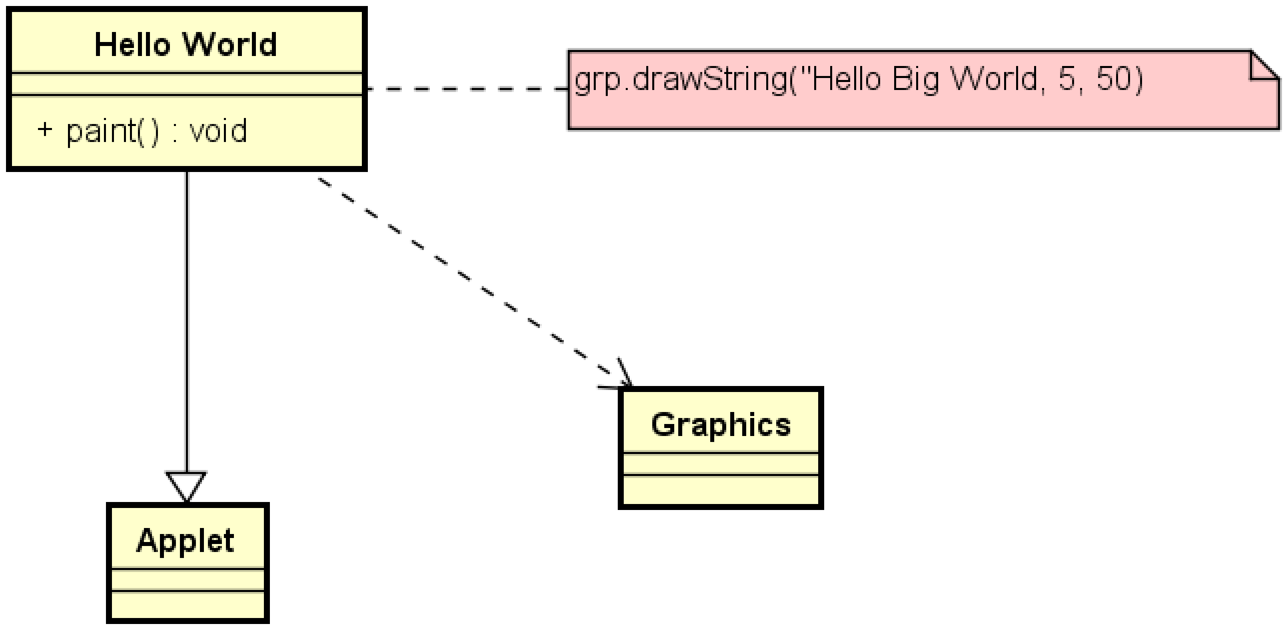
Attributes of a Well-Structured Class Diagram

- Focuses on communicating one aspect of a system's **static view**
- Contains only those elements that are essential to understanding one aspect of a ***system's static view***
- Provides detail **consistent with its level of abstraction** – only includes those adornments that are key to understanding
- Is **not so minimalist** that it misinforms the user about important semantics

Hints on Class Diagram Visualization

- Choose a ***suitable name*** – communicates the intended purpose
- Place those elements that are **semantically dependent** close to one another
- Layout elements to **minimize crossing** of relationships
- Use notes and other **visual cues** to emphasis key features
- Include **details** to the extent needed – i.e., the set of all attributes and operations for every element is usually not necessary to be shown (<u>hide details, not exclude them</u>)

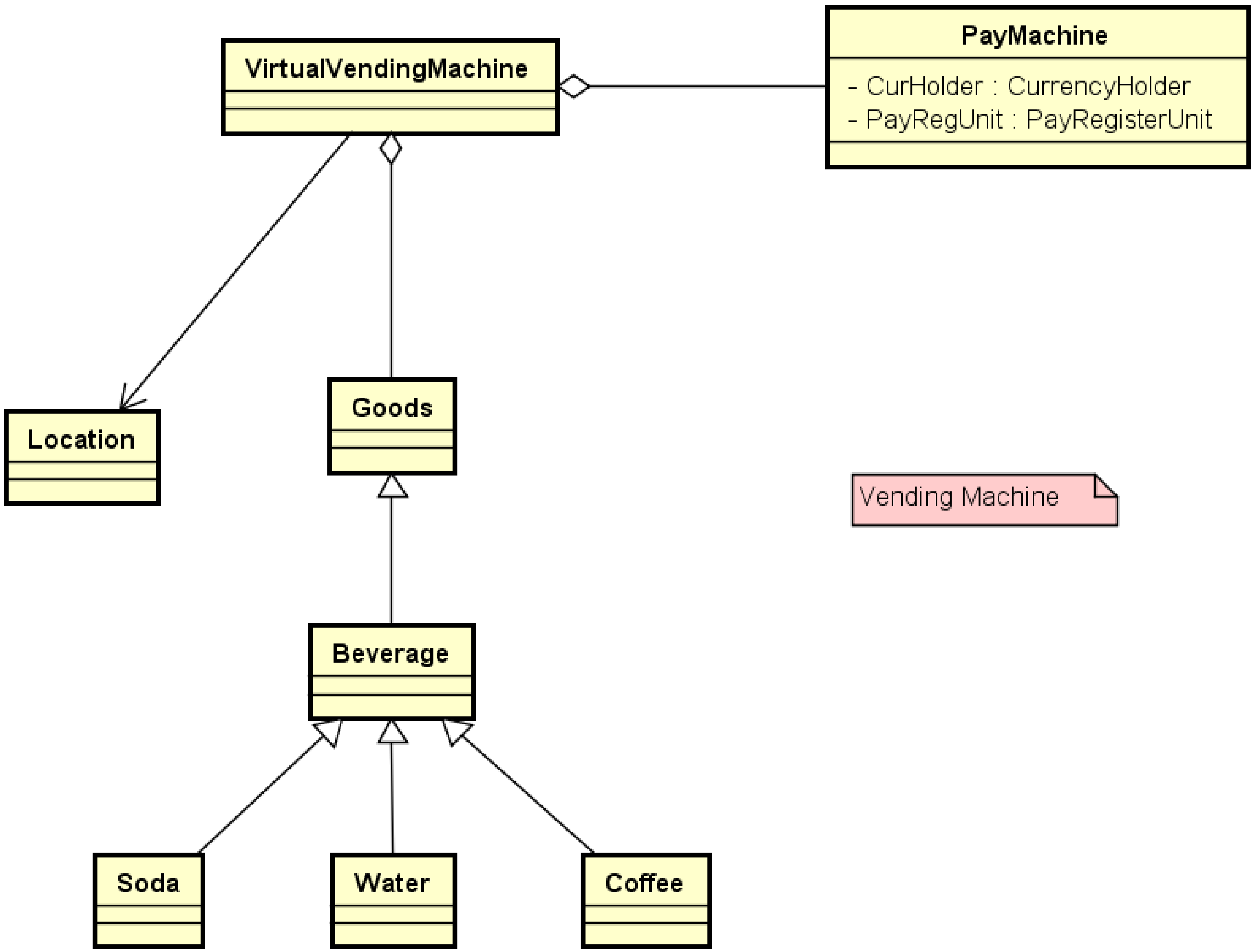
Basic Class Diagrams: HelloWorld Example

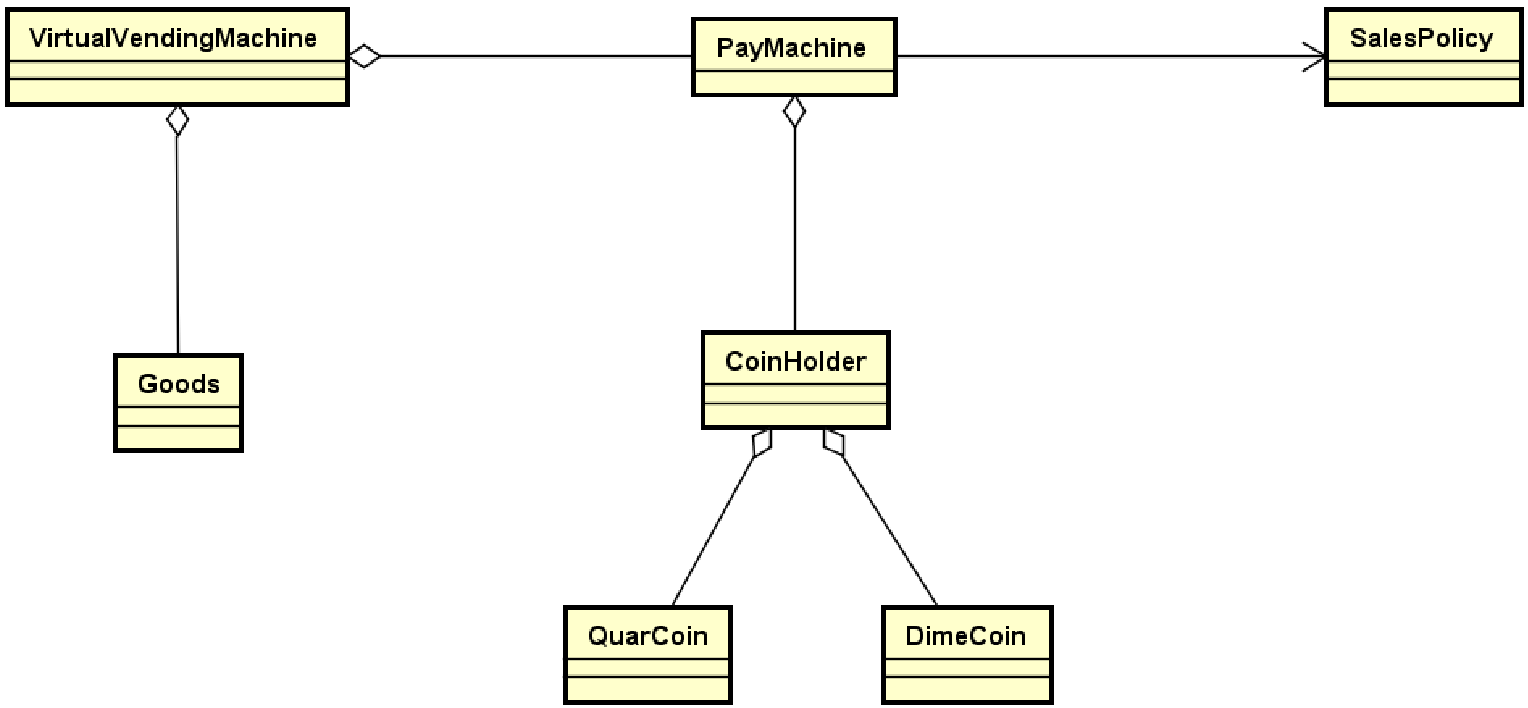


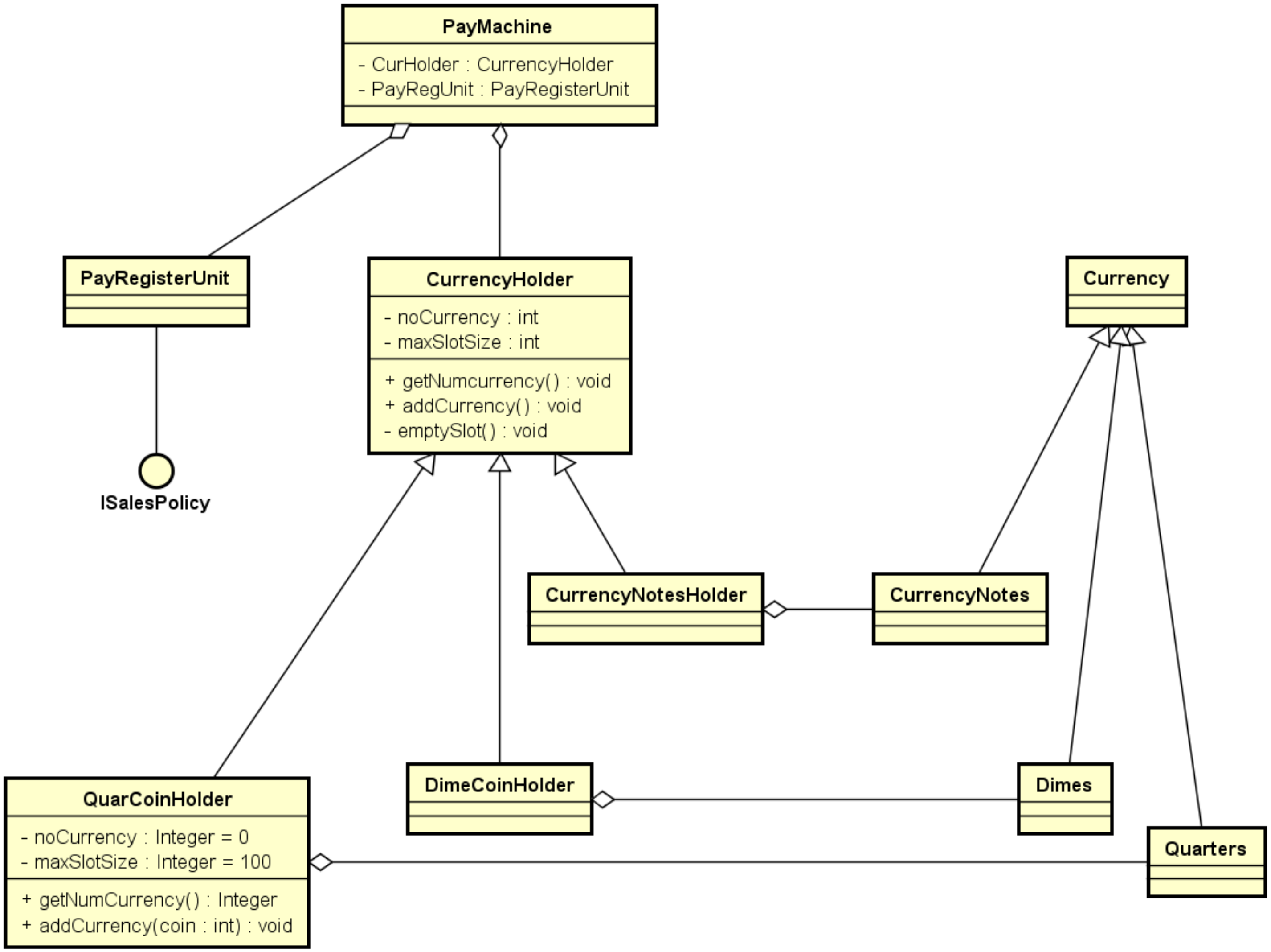
HelloWorld.java

```
import java.awt.Graphics;

class HelloWorld extends java.applet.Applet {
        public void paint(Graphics grp) {
                grp.drawString(“Hello, Big World”, 5, 50);
        }
}
```

Basic Class Diagrams: *Virtual Vending Machine Example*




VirtualVendingMachine
PayMachine
SalesPolicy
Goods
CoinHolder
QuarCoin
DimeCoin


PayMachine
- CurHolder : CurrencyHolder
- PayRegUnit : PayRegisterUnit
PayRegisterUnit
ISalesPolicy
CurrencyHolder
- noCurrency : int
- maxSlotSize : int
+ getNumcurrency() : void
+ addCurrency() : void
- emptySlot() : void
Currency
CurrencyNotesHolder
CurrencyNotes
QuarCoinHolder
- noCurrency : Integer = 0
- maxSlotSize : Integer = 100
+ getNumCurrency() : Integer
+ addCurrency(coin : int) : void
DimeCoinHolder
Dimes
Quarters

Forward and Reverse Engineering

- **Forward Engineering:** supports transforming a UML model into code through a mapping to an implementation language
  - Use of forward engineering should be decided in the early stages since it depends on the choice of programming languages – e.g., whether a programming language supports multiple inheritance or not affects analysis and design choices and methods
  - Forward engineering results in loss of information – UML has a more general and thus richer set of semantics
- **Reverse Engineering:** Supports transforming code implemented in a particular programming language into UML models – these models are generally incomplete!
- The Forward and Reverse Engineering are **complementary** – neither can be sufficient!

References

- *Object-Oriented Analysis and Design with Applications, 3rd Edition, G. Booch, et. al. Addison Wesley, 2007*
- *OMG Unified Modeling Language Specification, http://www.omg.org/spec/, 2016*
- *The Unified Modeling Language User Guide, G. Booch, J. Rumbaugh, I. Jacobson, Addison Wesley Object Technology Series, 1999*