2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Overview

- Java `class` mechanism (basics)
- Data Types
  - Enumeration Types
  - Record Types
- Data Abstraction
  - top-down (decompose), bottom-up (compose)
  - Abstract Data Type

A program that involves relations (Facebook, LinkedIn, ...)

```
boolean[][] friendship = new boolean[75][75];
    ...
    ...
friendship[1][42] = true;
    ...
    ...
if (friendship[i][j] && ! friendship[j][k]) {
    friendship[i][k] = false;
}
    ...
    ...
boolean connected[][] = new boolean[10000][10000];
    ...
    ...
```

Advantages and disadvantages of preceding code

?

Advantages and disadvantages of preceding code

Observations:

- It concerns data storage and manipulation
- It serves an abstract goal: to record a directed binary relationship

Con:

- Purpose unclear; client and implementation code are mixed
- Changing the data representation triggers changes in many places

Pro:

- Little overhead

How To Do Better?

- Apply the concepts of data type and data abstraction
- In Java, we (must) use the **`class`** mechanism for this.

Java `class` mechanism

The **`class`** is a central concept in the Java programming language.

It is very general and "powerful", and can easily be abused (cf. **`goto`**).

Initially, we used a **`class`** just to collect *static variables and methods*.

It can also involve: *instance variables and methods*, **`extends`** (inheritance), `@Override`, **`abstract`**, **`interface`**, **`implements`**, generics.

In this lecture: learn to use **`class`** to define some common data types.

- Enumeration type
- Record type
- Abstract Data Type (ADT)

Modularization: Top-down and Bottom-up Design

Design is an activity, to find and organize a solution.

- Top-down design
  - Given one (big) problem, split it into (smaller) subproblems
  - Continue until reaching trivially solvable problems
- Bottom-up design
  - Given many small solutions, combine them into bigger solutions
  - Continue until reaching a solution of the main problem

Solutions could be: expressions, statements, data definitions, ...
methods, classes, interfaces, packages, programs, ...

In practice: Yo-yo design (top-down and bottom-up combined)

Modularization for Actions

Procedural abstraction enables one to abstract from details within

- an expression
- a (possibly *compound*) statement

In Java, this is done by means of *methods* that

- operate on parameter objects, and/or
- return a primitive value, or (a reference to) an object, or **`void`**

A *method call* acts as an expression or statement.

(Limits to) Procedural Abstraction in Java

If expressions like `p*p*p*p` and `(x+1)*(x+1)*(x+1)*(x+1)/2` occur in various places, then this invites the procedural abstraction:

```
/** @return a^4 */
int pow4(int a) {
    int h = a * a;
    return h * h;
}

.... pow4(p) ... pow4(x+1)/2 ....
```

(N.B. This implementation is also more efficient, in multiplications. How to generalize that for raising to the power $n \geq 0$ efficiently?)

However, in Java it is impossible to define a procedural abstraction `inc(v)` for `v = v + 1`, where `v` is a parameter of type "**int** variable".

Modularization for Data: Data Abstraction (Bottom-up)

Combine variables of primitive types that frequently occur together into a single abstraction, instead of handling them separately.

- The numerator and denominator of a *fraction*
- The coefficients of a *polynomial*
- A pair (or triple) of coordinates of a *point* in the plane (in space)
- The given name, family name, and email address of a *person*

A data abstraction needs to be given a single name: variable vs type

In Java, grouping can be accomplished by `String`, array, and **`class`**. However, only a **`class`** introduces a name for a new type.

Class as Bundle of Variables: Example

```java
/** Data Type for fractions (simplistic). */
public class Fraction {

    /** The numerator. */
    long numerator;

    /** The denominator. denominator > 0 */
    long denominator;

    // Operations on objects of this type.
    ...
}

... Fraction q = new Fraction(2, 3);
... Fraction r = q.add(q);
```

Modularization for Data: Data Abstraction (Top-down)

What data items to **distinguish** and put in separate 'modules'?

Which concepts play a role?
Nouns in requirements serve as a hint for data abstractions.
Verbs hint at operations on the data.

Hierarchic decomposition . . .

`Surface` → `Triangle`s → `Point`s → `Coordinate`s → **`int`**/**`long`**/**`double`**

Possibly *recursive*, i.e. in terms of itself . . .

A `BinaryTree` is either empty, or has a root and two `BinaryTree`-s.
Recursion 'terminates' by using a **`null`** reference (empty tree).

(Data) Type

A (data) type is a *set of values* and *accompanying operations*.

Primitive data types in Java: **int**, **double**, **char**, **boolean**, ...
Values in subset of $\mathbb{Z}$, subset of $\mathbb{R}$, $\{\ldots,$'a'$,\ldots\}$, {false, true}

Type usage:

```
T v; // declares variable v of type T
     // during execution, v has one value of type T

... v ... // usage of v in operations; depends on T
... ++ v ... // for an int
... v = Math.sqrt(v); // for a double
... ! v && w ... // for a boolean
... v[2] ... // for an array
```

(Cf. *Type Theory* and *Type System*)

Data Type Definitions in Java: Enumerations

```
public class PrimaryColor {
    final static int RED = 0;
    final static int GREEN = 1;
    final static int BLUE = 2;
}

int v = PrimaryColor.RED; // a variable v having a primary color as value
```

or (why useful?):

```
    final static int RED = 1;
    final static int GREEN = 2;
    final static int BLUE = 4;
```

Better (since Java 5.0):

```
public enum PrimaryColor { RED, GREEN, BLUE }
PrimaryColor v = PrimaryColor.RED; // ...
```

Java Enumeration Types

```
public enum T { NAME1 , NAME2 , ... , NAMEn }
```

Set of values: the set of listed *names*

Operations (involving `v, w` of type `T`):

- constants (by their name): `NAME1, ...`
- iteration: `for (T v : T.values()) { ... v ... }`
- selection: `switch (v) { case NAME1: ...; break; ... }`
- conversion to/from string: `v.name()`, `v.toString()`, `T.valueOf(s)`
- ranking: `v.ordinal()` (ranks start at 0), `v.compareTo(w)`

Data Type Definitions in Java: ‘Records’

```java
/**
 * Record type for
 * the coordinates of a field on a chess board.
 */
public class ChessBoardField {
    public char line; // 'a' .. 'h'
    public int row; // 1 .. 8
}

ChessBoardField f; // value: the coordinates of a chess field

... f.line ... f.row ...  // using variable f
```

Java ‘Record’ Types via Classes

```
public class T { // BEGIN RECORD TYPE
    public Type1 fieldName1 ;
    ... ;
    public TypeN fieldNameN ;
} // END RECORD TYPE
```

Set of values: *Labeled product* of the value sets of the types, mapping `fieldName1` to `Type1`, etc.

Operations:

- field access (projection on field name): `v.fieldName1`, ...
- constructor, possibly parameterized to initialize fields

Advantages and Disadvantages of a Record Type

Advantages (grouping):

- Less code clutter when declaring variables:
  One record variable declaration introduces multiple variables.
- Can pass multiple variables in a record as one parameter.
- Can return multiple variables in a record as result from function.

Disadvantages (lack of information hiding):

- When changing the name of a field in a record,
  related client code must be updated.
- When changing the purpose, or mix of variables in a record,
  related client code must be updated.

Abstract Data Type (ADT): Definition of Concept

An Abstract Data Type is a type whose *specification* and *usage* abstracts from (i.e. does not depend on) *implementation* details.

The implementation deals with the choice of data representation and algorithms for operations on that representation.

The implementation of an ADT can be changed, without affecting the *usage occurrences*, provided it adheres to the ADT's *specification*.

This is called implementation hiding, also known as encapsulation.

An ADT can serve as a module of a program.

Abstract Data Type: Specification Plays Central Role

contract

$\nearrow \quad \| \quad \nwarrow$

usage $\quad \| \quad$ **implementation**

(client) $\quad \| \quad$ (provider)

- Relate usage to contract and

  relate implementation to contract.

- *Never* relate usage and implementation directly.

  That way, ‘divide’ would fail,
  leading to complexity and errors, and hence not to ‘conquer’.

Example: Java Collections Framework (JCF)

- Abstract Data Types for various containers and mappings
- Concepts of a `Collection`, `Map`
- Concepts of a `List`, `Set`, `Queue`
- Implementations: `ArrayList`, `HashSet`, `HashMap`
- Involves **`interface`**s and **`class`**es

See Ch.10.1.4 in David Eck's book

Abstract Data Type: Specification of Syntax

- Type name
- Operations (methods) with names and typed parameters:
  - Constructor, destructor (memory management)

    N.B. In Java, object destruction is implicit and automatic via garbage collection
  - Queries (state inspection), with return type
  - Commands (state change), **`void`**

ADT Specification of Syntax: Example

To make everything compile.

```java
public abstract class IntRelation {
    public IntRelation(final int n) {
    }
    public abstract int extent();
    public abstract boolean areRelated(int a, int b);
    public abstract void add(int a, int b);
    public abstract void remove(int a, int b);
}
```

Abstract Data Type: Specification of Semantics = Contract

- Set of (abstract) values
  - given *explicitly* (through *model variables*; we do this), or
  - given *implicitly* (through postulated properties of operations)
- Contracts for individual operations
  - **precondition** and **postcondition**, in terms of *abstract* values
  - **modifies** clause (for commands), thrown **exceptions**
- Public invariants: guaranteed relationships between model variables and basic queries

In Java, specification elements are stated in Javadoc comments

ADT Specification of Semantics: Example

Set of values (model) and public invariants:

```
/**
 * An {@code IntRelation} object maintains a relation on small integers.
 * The relation is a subset of {@code [0..n) x [0..n)},
 * where {@code n}, called the <em>extent</em> of the relation,
 * is given in the constructor and is immutable.
 * There are operations to test membership,
 * and to add and remove pairs.
 * <p>
 * Model: subset of {@code [0..extent()) x [0..extent())}
 *
 * @inv {@code NonNegativeExtent: 0 <= extent()}
 */
public abstract class IntRelation {
```

ADT Specification of Semantics: Example

Constructor:

```
/**
 * Constructs an empty relation of given extent.
 *
 * @param n  extent of the new relation
 * @pre {@code 0 <= n}
 * @post {@code this == [ ]  &&  this.extent() == n}
 */
public IntRelation(final int n) {
}
```

ADT Specification of Semantics: Example

Basic Query:

```
    /**
     * Returns the extent of this relation.
     *
     * @return extent of this relation
     * @pre {@code true}
     * @modifies None
     * @post (basic query)
     */
    public abstract int extent();
```

ADT Specification of Semantics: Example

Another Query:

```java
/**
 * Returns whether the elements in a pair are related.
 *
 * @param a  first element of the pair
 * @param b  second element of the pair
 * @return  whether {@code (a, b)} are related
 * @pre {@code isValidPair(a, b)}
 * @modifies None
 * @post {@code \result == (a, b) in this}
 */
public abstract boolean areRelated(int a, int b);
```

ADT Specification of Semantics: Example

Command:

```java
/**
 * Adds a pair to the relation.
 *
 * @param a  first element of the pair to add
 * @param b  second element of the pair to add
 * @pre {@code isValidPair(a, b)}
 * @modifies {@code this}, but not {@code this.extent()}
 * @post {@code this == \old(this) union [ (a, b) ]}
 */
public abstract void add(int a, int b);
```

ADT Specification of Semantics: Example

Another Command:

```
/**
 * Removes a pair from the relation.
 *
 * @param a  first element of the pair to remove
 * @param b  second element of the pair to remove
 * @pre {@code isValidPair(a, b)}
 * @modifies {@code this}, but not {@code this.extent()}
 * @post {@code this == \old(this) minus [ (a, b) ]}
 */
public abstract void remove(int a, int b);
```

Another Example of ADT Syntax: `Set<E>`

- Model: (mathematical) set over type `E` (no order, no duplicates)
- Client does not (need to) know how this is stored/implemented
- Constructor `XxxSet<E>()`: returns empty set over type `E`
- Query **`int`** `size()`: number of elements in **`this`**
- Query **`boolean`** `contains(Object o)`: whether `o` $\in$ **`this`**
- Command `add(E e)`: adds `e` to **`this`**
- Command `remove(Object o)`: removes `o` from **`this`**, if present

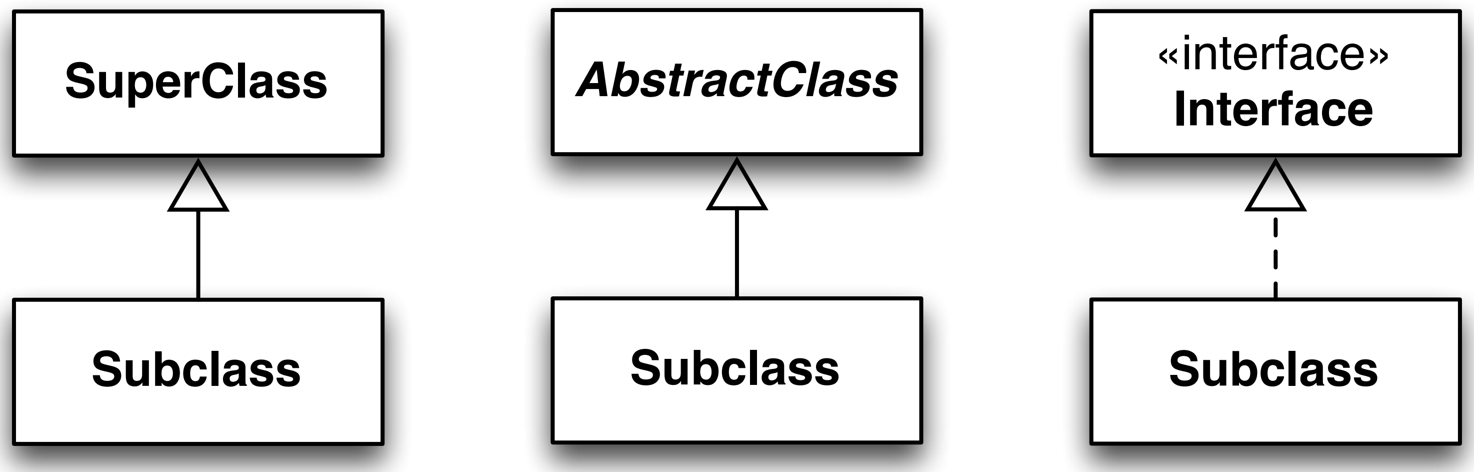
Java Interfaces, to Specify an ADT

- An **`interface`** defines a collection of method headers, with contracts.
- An interface has neither instance variables, nor method implementations. It can define constants with **`public static final ...`**
- An interface is somewhat like a class with only abstract methods.
- A class can implement (via **`implements`**) one or more interfaces, by implementing each of the methods in the interface(s).
- An interface can be viewed as a type:
  Its values are the values of all classes that implement the interface.

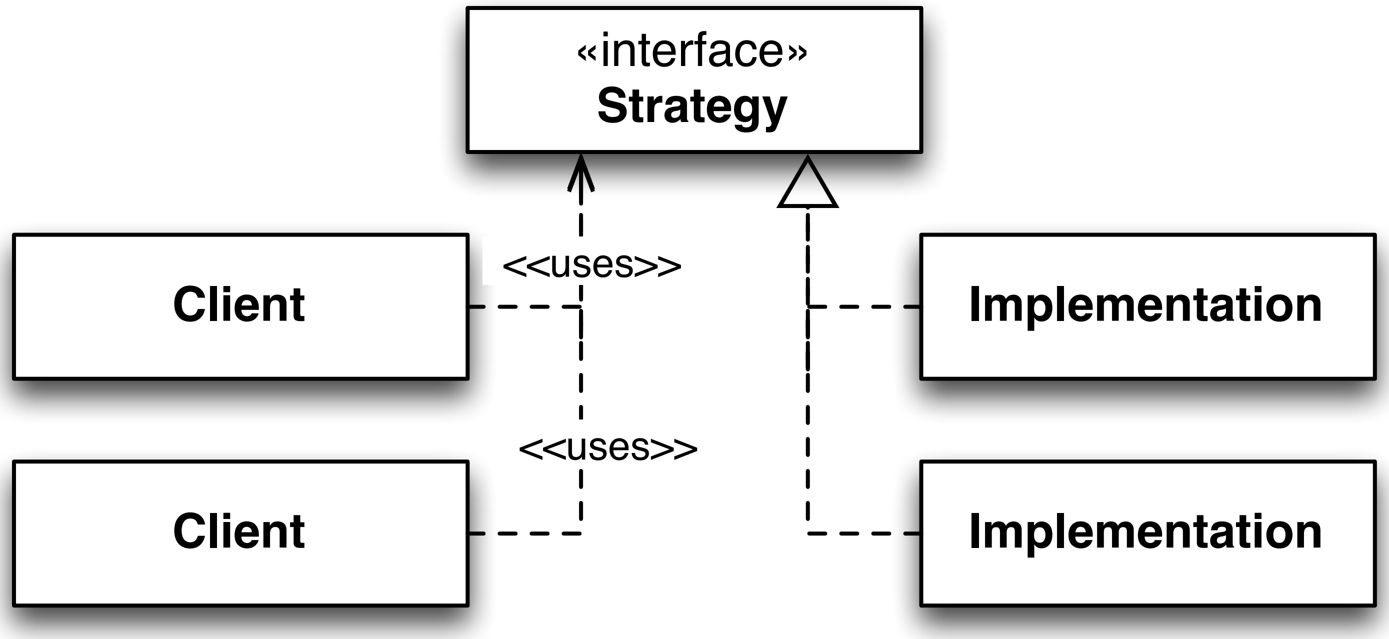
UML Class Diagrams for Software Design

UML = Unified Modeling Language

A graphical software design language based on an open standard



Compare ADT Usage–Spec–Impl to Strategy Pattern



The power is in the *missing* dependencies! Compare to Slide 20.

Clients do not depend on concrete implementations, or vice versa.

Abstract Data Type: Usage in Java

- Declare variable: `ADTname v = new ADTname(...);`

  E.g. `Fraction f = new Fraction(1, 2), g = new Fraction(2, 3);`

- Apply operations: `... v.operationName(...) ...`,
  in a state where the corresponding precondition holds

  E.g. `f.add(g)`

- Syntactically, this looks the same for all ADTs.

  (Recall that usage syntax for primitive types varies wildly.)

ADT Usage: Example Client Code

```java
IntRelation friendship;
...
friendship = new IntRelationArrays(75);
...
...
friendship.add(1, 42);
...
...
if (friendship.areRelated(i, j) && ! friendship.areRelated(j, k)) {
    friendship.remove(i, k);
}
...
...
IntRelation connected = new IntRelationListOfSets(10000);
...
...
```

Abstract Data Type: Implementation in Java (in brief)

- Provide a data representation, a representation invariant, and an abstraction function.
- For each operation, provide a method implementation adhering to the contract.

Abstract Data Type: Implementation (Values Part)

Data representation is defined in terms of instance variables that represent intended abstract values of the ADT.

```
class Fraction { private int numerator, denominator; ... }
```

Representation invariant (short: *rep invariant*) is condition to be satisfied by the instance variables, in order to make sense as a representation of an abstract value. Can be implemented in method `boolean isRepOk()`, to support unit testing of the ADT.

```
// rep inv I0: 0 < denominator
```

Abstraction function maps each *representation that satisfies the rep invariant* to the represented abstract value.
Can be implemented (in a way) in method `String toString()`.

```
// AF(q) = q.numerator / q.denominator
```

ADT Implementation: Example of Values Part

```
public class IntRelationArrays extends IntRelation {

    /** Representation of the relation. */
    protected final boolean[][] relation;

    /*
     * Representation invariants
     *
     * NotNull: relation != null
     * Extent: relation.length == extent()
     * ElementsNotNull: (\forall i; relation.has(i); relation[i] != null)
     * ElementsSameSize: (\forall i; relation.has(i);
     *     relation[i].length == relation.length)
     *
     * Abstraction function
     *
     * AF(this) = set of (a, b) such that relation[a][b] holds
     */
```

Abstract Data Type: Implementation (Operations Part)

Provide method body for each operation, adhering to its *contract*.

- Its pre- and postcondition must be re-interpreted in terms of the *data representation*, using the *abstraction function*.
- *Rep invariant* serves as additional *precondition*.
- *Rep invariant* serves as additional *postcondition* (for commands).
- Also, the *public invariant* must be honored.

ADT Implementation: Example of Operations Part

Constructor:

```
    public IntRelationArrays(final int n) {
        super(n);
        relation = new boolean[n][n];
    }
```

Postcondition `this == [ ]` $\xleftarrow{AF}$ `(\forall i, j; ; ! relation[i][j])`

Postcondition `this.extent() == n` $\xleftarrow{AF}$ `relation.length == n`

ADT Implementation: Example of Operations Part

Queries:

```java
    @Override
    public int extent() {
        return relation.length;
    }

    @Override
    public boolean areRelated(int a, int b) {
        return relation[a][b];
    }
```

In postcondition: `(a, b) in this` $\xleftarrow{AF}$ `relation[a][b]`

ADT Implementation: Example of Operations Part

Commands:

```java
    @Override
    public void add(int a, int b) {
        relation[a][b] = true;
    }

    @Override
    public void remove(int a, int b) {
        relation[a][b] = false;
    }
```

ADT Specification (...) & Implementation (___) in Java

```
/** An ADTname object provides ...
  * Model: ...
  * @inv public invariant ...
  */
public class ADTname {
    /** Representation ___ */
    private ___ ___;
    /* ___ private invariant ___ abstraction function ___ */

    /** Constructs ...  @pre ... @post ... @throws ... */
    public ADTname() { ___ }

    /** Queries ...  @pre ... @return ... @throws ... */
    public ReturnType queryName(...) { ___; return ___; }

    /** Changes ...  @pre ... @modifies ... @post ... @throws ... */
    public void commandName(...) { ___ }
}
```

Java Limitation

Unfortunately, in a Java **`class`**, *specification* and *implementation* are often combined into a single interleaved description.

Javadoc helps extract just the specification.

Alternative: Put specification in an **`abstract class`** or **`interface`**

- Java **`abstract class`** defines a *type* with partial implementation;

  (partial) data representation, (partial) method implementations

- Java **`interface`** defines a *type* without any implementation.

Week 3 / Assignments Series 3

- Refresher: book by Eck: §1.5, §2.3.4, §3.6.3, §10.1.(3–4), §10.2.4 (Enums), and §5.(1–6)
- Assignment C6: Robust `IntRelation`
- Graded assignment G1: released later
- Interim Test #1: this Friday, 9:00-10:00

Summary

- Java **`class`** can be used for

  1. Library of static constants and methods (no state)
  2. Defining an enumeration type: a set of related constants
  3. Defining a record type: a labeled-product type
  4. Defining an abstract data type (ADT)

- ADT encapsulates/hides the *representation* of the abstract data and the implementation of *operations* on that data.

- It separates the *use* of and the *implementation* of a data type, through a *specification* in the form a of a two-sided contract.

- This allows modification of the implementation without requiring modification of the using environment, and vice versa.