Lecture 20: Implementation

15.07.2013

Contents

Implementation

- **Input:** software architecture, specification of system components
- **Artifacts:** programs, documentation, test documentation, verification documentation
- **Activities:** (programming in the small)
  - refinement
  - development of algorithms and data structures
  - documentation of implementation decisions
  - coding
  - testing

Implementation Principles

Verbalization

- use meaningful identifiers
  bad: `help`, `tmp`, `var`, `store`
  better: `averageSales`, `aspectRatio`
- name constants

  ```
  static final int interest = 0.005;
  ...
  balance += balance * interest;
  ```

  better than

  ```
  balance *= 1.005;
  ```

- avoid short identifiers (typos!)
- use self-documenting programming language
- include further documentation in programs (*e.g.*, javadoc)
- avoid insignificant comments: `i++; // increment i`

Powerful programming concepts

- decrease cost of implementation and maintenance

automatic memory management (garbage collection)

- avoids manual allocation and deallocation of memory
- *e.g.*, in Lisp, Smalltalk, Prolog, ML, Haskell, Java, C#, ...
- disadvantages: slowdown (little), space usage, lack of control
- advantages: whole class of nasty errors eliminated

parametric polymorphism

- *e.g.*, in ML, Haskell, J2SE 1.5
- full type safety
  - typing errors recognized by compiler
  - no casts required
- increased reusability

Example for Generics in Java

```
// Java 1.4
static void dump(String what, Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        String s = (String) i.next();
        if (s.indexOf(what) > 0)
                System.out.println(s);
    }
}

// Java 1.5
static void dump(String what, Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); ) {
        String s = i.next();
        if (s.indexOf(what) > 0)
                System.out.println(s);
    }
}
```

first-class functions (obsoletes Command pattern)

- *e.g.*, in Lisp, Smalltalk, ML, Haskell, Python, JavaScript, . . .
- functions as parameters and results
- functions in data structures
- user-defined control structures

Example for user-defined control structure (Haskell)

```
-- example: divide and conquer
dc :: (a -> Bool) -> (a -> b) -> (a -> [a]) -> ([b] -> b) -> a -> b
dc isSimple solve partition combine problem = f problem
  where
  if   isSimple problem
  then solve problem
  else combine (map f (partition problem))

-- applied to quicksort
qsort = dc isSimple solve partition combine
  where
  isSimple  xs = length xs <= 1
  solve     xs = xs
  partition (x0:xs) = [[ x | x <- xs, x < x0]
                      ,[x0]
                      ,[x | x <- xs, x >= x0]]
  combine   xss = concat xss
```

Principle of Integrated Documentation

Goals:

- simplify orientation and maintenance
- explanation of (algorithm) design decisions
- administrative information (version numbering, authors, state, known problems)
- specification information (pre-, postconditions, invariants, complexity)

Ideally: integrated construction of code and documentation

- *e.g.*, javadoc, design by contract
- less overhead
- fewer inconsistencies
- otherwise implementation decisions may get lost

Principle of Stepwise Refinement

- Start in pseudocode style with abstract operators
- Refine operators and data structures simultaneously by decomposition, implementation, and choice (of data structure)
- Alternative refinements lead to tree structure with leaves corresponding to solutions
- Methodology formulated by Niklaus Wirth

  Program Development by Stepwise Refinement, *Communications of the ACM*, 14:4, April 1971, pp221-227
- Illustrated by example problem "Eight Queens"

Eight Queens

Problem statement Given an 8x8 chessboard and 8 queens which are hostile to each other. Find a position for each queen such that no queen may be taken by any other queen (*i.e.*, each row, column, and diagonal contains at most one queen).

- no analytic solution known

⇒ apply "generate and test"

Eight Queens: Generate and Test

- A set of candidate solutions
- p predicate for verifying a solution
- solution: $x \in A \wedge p(x)$

```
do {
  Generate the next element of A and call it x
} while(not p(x) and (more elements in A));

if p(x) then x = solution
```

- Problem: too many candidates $\mid A \mid = 64!/(56! \cdot 8!) = 2^{32}$
- Almost 5 days if 100μs per round

Eight Queens: Strategy of Preselection

- Decompose $p = q \wedge r$
- Let $B_r = \{x \mid x \in A \wedge r(x)\}$ such that
  - $\mid B_r \mid \ll \mid A \mid$
  - elements of B_r are easily generated
  - q is easier to test than p

```
do {
  Generate the next element of B and call it x
} while(not q(x) and (more elements in B));

if q(x) then x = solution
```

- Suitable r: exactly one queen in each column
- q: at most one queen in each row and diagonal
- $\mid B_r \mid = 8^8 = 2^{24}$
- 27 minutes (at 100μs per round)

Eight Queens: Stepwise Construction of Trial Solutions

- Find a representation of candidates $[x_1, x_2, \ldots, x_n]$ such that
    - generating x_j from $[x_1, \ldots, x_{j-1}]$ must be simpler than finding a complete candidate
    - $q[x_1, x_2, \ldots, x_n] \Rightarrow q[x_1, x_2, \ldots, x_j]$ for all $j < n$.

```
j := 1;
do {
  trystep (j);
  if (successful)
    advance
  else
    regress
} while (j >= 1 && j <= n)
```

- Criteria satisfied for eight queen problem.
- First solution found after testing 876 configurations.

Eight Queens: Top-level Structure

```
variable board, column, safe;

considerFirstColumn;
do {
  tryColumn;
  if( safe ) {
    setQueen;
    considerNextColumn;
  } else
    regress;
} while (not (lastColDone || regressUnderflow))
```

Abstract operators

- `considerFirstColumn`: initializes first column
- `tryColumn`: move down the column until an unthreatened square is found (then set `safe` to true) or until all squares have been considered (then set `safe` to false)
- `setQueen`: put queen in last inspected square
- `considerNextColumn`: advance to next column and initialize
- `regress`: go back to most recent column where the queen can still be moved

Eight Queens: Refinement of `tryColumn` and `regress`

```
void tryColumn () {
  do {
    advancePointer;
    testSquare;
  } while (not (safe || lastSquare))
}

void regress () {
  reconsiderPriorColumn
  if (not regressUnderflow) {
    removeQueen;
    if (lastSquare) {
      reconsiderPriorColumn;
      if (not regressUnderflow)
        removeQueen;
    }
  }
}
```

Eight Queens: Obvious Data Representation

```
boolean safe;
int column;          // 0 <= column <= 9
int board[];    // new int [9]; 0 <= board[i] <= 8
```

- `considerFirstColumn`: `board[column = 1] = 0`
- `considerNextColumn`: `board[column++] = 0`
- `reconsiderPriorColumn`: `column--`
- `advancePointer`: `board[column]++`
- `lastSquare`: `board[column] == 8`
- `lastColDone`: `column > 8`
- `regressUnderflow`: `column < 1`

To do:

- `setQueen` (vacuous)
- `removeQueen` (vacuous)
- `testSquare` (sets `safe`; complicated, but most frequently executed)

Eight Queens: Clever Data Representation

- Possible refinement step: introduce data structure such that testing for threatened row, column, and diagonal is in constant time
- Three additional boolean arrays `rowFree`, `mainDiagFree`, `minorDiagFree`
  - `rowFree[k]` iff row `k` is free; $1 \leq \mathtt{k} \leq 8$
  - `mainDiagFree[k]` iff the main diagonal with coordinate sum `k` is free; $2 \leq \mathtt{k} \leq 16$
  - `minorDiagFree[k]` iff the minor diagonal with coordinate difference `k` is free; $-7 \leq \mathtt{k} \leq 7$
- Leads to `testSquare` defined as

```
safe =  rowFree[board[column]]
     && mainDiagFree[column + board[column]]
     && minorDiagFree[column - board[column]]
```

- `setQueen` as (`removeQueen` is analogous)

```
rowFree[board[column]] =
  mainDiagFree[column + board[column]] =
    minorDiagFree[column - board[column]] = false
```

- `board[column]` should be factored out

Eight Queens: Summary

- Final solution obtained by substitution
- Original structure retained by final solution
- At choice points in algorithm design: different assignments to data structures and abstract operators
- Similar steps lead to a recursive solution
- Resulting program simple to extend to obtain all solutions
- However, there is still some redundancy in the program. . .

Transforming Models into Code

Transforming Models into Code

- Some models better suited than others:
  - \+ state charts (FSA), decision tables, class diagrams, Z, B, . . .
  - – sequence diagrams, Petri nets, . . .
- CASE tools support code generation from models (UML, Z, B,. . .)
  - rudimentary
  - sometimes also:
    - round-trip engineering, reverse engineering
    - requires program analysis (maintenance!)
  - interesting problems

Here:

- Implementation of UML class diagrams

Code Generation for Class Diagrams

- Assumption: class diagram refined to implementation/code perspective
- Class diagrams cover static aspects
    - data model
    - inheritance
    - navigability
- Dynamic aspects underspecified → stubs
- (Directly) expressible in OO PL
- Still grey areas: composition, aggregation, . . .

Code for Classes and Interfaces

Person ↦

```
public class Person {
  Person () {}
}
```

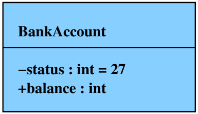
Window ↦

```
public abstract class Window {}
```

<<interface>>
Employee ↦

```
public interface Employee {}
```

Attributes — Minimalist approach



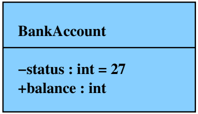
BankAccount

−status : int = 27
+balance : int

Map visibility and generate constructor

```
public class BankAccount {
  private int status = 27;
  public int balance;

  public BankAccount () {}
  public BankAccount (int balance) { this.balance = balance }
}
```

Attributes — Encapsulated approach



BankAccount

–status : int = 27
+balance : int

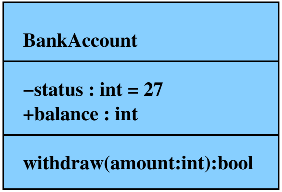
Hide all attributes, generate getter and setter methods

```
public class BankAccount {
  private int status = 27;
  private int balance;

  public BankAccount () {}
  public BankAccount (int balance) { this.balance = balance; }

  public int getBalance () { return this.balance; }
  public void setBalance (int balance) { this.balance = balance; }
}
```

Implementation decisions

- signature of constructor
- access to attributes (JavaBean naming convention: `get`*Name*, `set`*Name*)

Operations



BankAccount

−status : int = 27
+balance : int

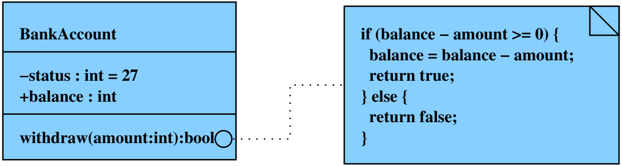
withdraw(amount:int):bool

Generate code stub:

```
public boolean withdraw (int amount) {
  // your code goes here
}
```

- Sufficient for interface or abstract class

Operations/2

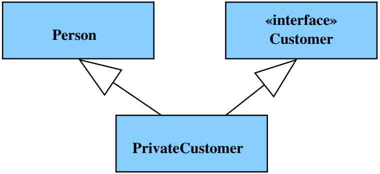


```
if (balance – amount >= 0) {
  balance = balance – amount;
  return true;
} else {
  return false;
}
```

Copy code from template:

```
public boolean withdraw (int amount) {
  if (balance - amount >= 0 ) {
    balance = balance - amount;
    return true;
  } else {
    return false;
  }
}
```

Inheritance

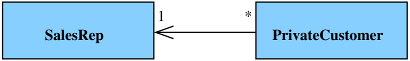


```
public class Person {...}
public interface Customer {...}

public class PrivateCustomer extends Person implements Customer {
  public PrivateCustomer () { super(); } // calls Person
}
```

- For Java multiple inheritance must be removed
- Are models independent of implementation language?

Associations

Simple directed association

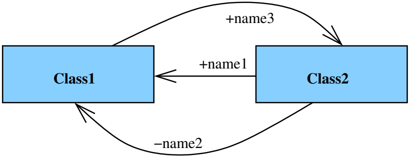


Meaning: `PrivateCustomer` objects can send messages to `SalesRep` object
Implementation:

- instance variable, here with access functions
- naming: role name, association name, or target class name

```
public class PrivateCustomer {
  private SalesRep salesRep;

  public SalesRep getSalesRep() { return salesRep; }
  public void setSalesRep (SalesRep salesRep) { this.salesRep = salesRep; }
}
```

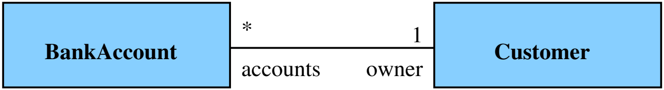
Directed, Named Associations



- One instance variable per name
- Visibility transferred

```
public class Class1 {
  public Class2 name3;
}
public class Class2 {
  public Class1 name1;
  private Class1 name2;
}
```

Association with Multiplicity

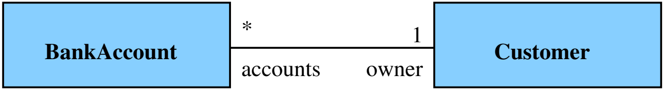


Simple approach (Rational Rose): arrays

```
public class BankAccount {
  public Customer owner;
}
public class Customer {
  public BankAccount[] accounts;
}
```

Alternatives: container classes (`Collection`), RDBMS

Refined approach:



<<Account.java>>

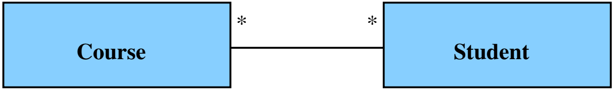
```
public class Account {
  private Customer owner;
  public Customer getOwner () {
    return owner;
  }
  public void setOwner (Customer newOwner) {
    if (newOwner == owner) return;
    if (owner != null) owner.removeAccount(this);
    owner = newOwner;
    if (owner != null) owner.addAccount(this);
  }
}
```

Public interface: `Account.setOwner()`

<<Customer.java>>

```
public class Customer {
  private Collection<Account> accounts =
    new LinkedList<Account>();
  public Collection<Account> getAccounts () {
    Collections.unmodifyableCollection(accounts);
  }
  void removeAccount(Account account) {
    accounts.remove(account);
  }
  void addAccount (Account account) {
    accounts.add(account);
  }
}
```

Many-to-many Association



Implementation depends on navigation requirements

- one-way: collections or arrays
- multi-way (*e.g.*, iteration over pairs (course, student)): separate structure (cf. DB table)
- no directly suitable Java datastructure