Java Program Style and Design

- Program Design Concepts:
  - Design Patterns
  - Refinement
  - Abstractions - Program Conceptualization
    - Object-oriented programming and abstraction
  - Pseudo-code: abstraction and refinement
  - JavaDoc
  - Software Lifecycle
  - UML - Unified Modeling Language

Programming Rules of Thumb

- **Learn *program patterns*** of general utility and use a relevant pattern (if you know one) for the problem at hand.
- **Seek inspiration** by systematically working test data by hand. Be introspective; ask yourself: “What process am I doing?”.
- **Write comments** that precisely describe the contents of each variable and each section of the program.
- **Declare variables** for each piece of information you maintain when working the problem by hand.
- Remember the problem’s **boundary conditions**.
- **Validate** your program by tracing it on simple test data.

A Useful Design Pattern

```
/* "Process" integer input values until (but not
   including) some designated stoppingValue. */
 α  // initialization
  variable = in.readInt();
  while ( variable != stoppingValue )
    {
       β      // Main processing
       variable = in.readInt();
    }
 γ   // finalization and wrapup
```

Constrast With

```
/* Do b for each integer between 1 and n. */
 α
  variable = 1;
  while ( variable <= n )     //or use for(,,) loop
    {
       β
       variable = variable + 1;
    }
 γ
```


Pattern for "processing" grades

```
int grade;     // the grade being processed.
```

```
/* "Process" grades until (but not including) a stopping
   signal of -1. */
 α
  grade = in.readInt();
  while (grade != -1 )
    {
       β
       grade = in.readInt();
    }
 γ
```

Task 1: Print the grades

```
int grade;     // the grade being processed.
```

```
/* "Process" grades until (but not including) a stopping
   signal of -1. */
  α
  grade = in.readInt();
  while (grade != -1 )
    {
       β
       grade = in.readInt();
    }
 γ
```

where

α :

β : `System.out.println( grade );`

γ :

Task 2: Print the average - similar pattern

```
int grade;     // the grade being processed.
int count;     // # of grades so far.
int sum;       // sum of grades so far.
```

```
/* "Process" grades until (but not including) a stopping
   signal of -1. */
  α
  grade = in.readInt();
  while (grade != -1 )
    {
       β
       grade = in.readInt();
    }
 γ
```

where

α : `count = 0; sum = 0;` // initialization

β : `count = count + 1;` // main processing
`sum = sum + grade;`

γ : `System.out.println( sum / count );` // finalize

Ways to *Refine P* into Subproblems

- A program pattern
  - Do *whatever* n times
  - Process input values up until (but not including) a stopping value.
- Stepwise refinement
  - Break problem into subproblems
- Iterative refinement
  - 'chip away' at the problem, each time solving a portion and leaving a 'smaller' subproblem

Programming By Stepwise Refinement

Given: a problem *P*, write Java program that solves *P*

- An "algorithm" for you to follow when programming:

```
if ( P is simple enough to code immediately )
   Write the Java code that solves P;
else {
   Refine P into subproblems;
   Write Java code that solves each subproblem;
   }
```

- The refinement of *P* into subproblems must include a description of how the code segments solving the subproblems combine to form code that solves *P*.
- You can write the Java code segments that solve the subproblems in any order.

Stepwise Refinement

- Break a **complex problem** down into a number of **simpler steps**, each of which can be solved by an algorithm which is smaller and simpler than the one required to solve the overall problem.
- Smaller and simpler therefore easier to construct and sketch in detail.
- Sub-algorithms themselves can be broken into smaller portions
- Refinement of the algorithm continues in this manner until each step is sufficiently detailed.
- Refinement means replacing existing steps/instructions with new version that fills in details.
- Example: **Making tea**. Suppose we have a **robot** which carries out household tasks. We wish to program the robot to make a cup of tea. An initial attempt at an algorithm might be:

1. Put tea leaves in pot
2. Boil water
3. Add water to pot
4. Wait 5 minutes
5. Pour tea into cup

Slide 9

Next Step of Refinement

- These steps are probably not detailed enough for the robot. We therefore refine each step into a sequence of smaller steps:

1. Put tea leaves into pot
might be refined to
1.1 Open box of tea
1.2 Extract one spoonful of tea leaves
1.3 Tip spoonful into pot
1.4 Close box of tea

Similarly
2. Boil water
might be refined to
2.1. Fill kettle with water
2.2 Switch on kettle
2.3 Wait until water is boiled
2.4 Switch off kettle

5. Pour tea into cup
might be refined to
5.1. Pour tea from pot into cup until cup is full

Substep Refinement

- Some of the sub-algorithms need further refinement. For example, the step
  - 2.1. Fill kettle with water
    might be refined as
    2.1.1. Put kettle under tap
    2.1.2. Turn on tap
    2.1.3. Wait until kettle is full
    2.1.4. Turn off tap
- The program is then constructed by translating the final refinement of each step into Java.

Stepwise Refinement - Sorting

- Sort elements x1, x2, …, xn :
  Substeps:
  - Divide into 2 sublists:
    $x_1, \ldots, x_{n/2}$ and $x_{n/2+1}, \ldots, x_n$
  - Sort sublist1 and sort sublist2
  - Merge the two sorted sublists
    - since both sublists are sorted, it is a linear process to combine lists from left to right

Iterative Refinement

- Factorial: n! is n(n-1)(n-2)….1
  - Factorial (k): // pseudo-code
    - if k > 1 then ans = k * Factorial(k-1)
      else ans = 1
    - return ans
  - **long factorial(int k) {** // recursive solution
    if (k >1) return(k * factorial(k-1));
    else return 1
    }
  - **long fact_iter(int k)** // interative solution
    {
    long ans = 1 ;
    for (int j=1; j<=k ; j++)
    ans = ans * j ;
    return ans;
    }

Program Conceptualization

- Abstraction:
  - Focus on the **essential** aspects of an **entity** (thing or concept)
  - *Ignores* or **conceals** less important aspects.
  - **Simplifies** complex situation
  - **Attributes**: properties or characteristics of entity
    - typically correspond to the **data** that is recorded
  - **Behavior**: set of actions that the object can perform.
- The **abstraction** is based on the goal or problem domain:
  - which data are essential
  - which behaviors or operations are needed
  - **constraints** which reflect real world:
    - age is non-negative
    - object cannot be in two places at same time

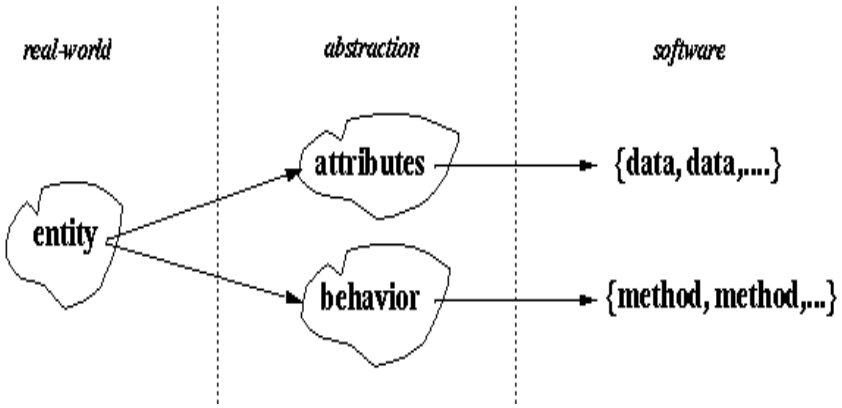
Properties of a Good Abstraction

- Abstraction:
  - A named collection of attributes and behavior relevant to modeling a given entity for some particular purpose.
- Desirable properties for an abstraction are:
  - **Well named**: suggests correct intuition, expectations.
  - **Coherent**: contains a related set of attributes and behavior that make sense from the viewpoint of the modeler.
  - **Minimal:** not contain extraneous attributes or behavior for the intended purpose.
  - **Complete**: contain all of the attributes and behavior necessary for its intended purpose.
- Good Design:
  - **Choose the 'right' abstraction for the purpose**.

Mapping Abstractions to Classes and Objects

- The **attributes** and **behavior** of an **abstraction** are mapped to:
  - *Attributes*: a set of data (variables, array, lists, complex data structures, etc.)
  - *Behavior*: a set of methods (also known as operations, functions, actions).
- The rendering of abstractions in software has always been the implicit goal of programming.
- *Object-oriented programming* offers sophisticated structures: **classes** and **objects** to represent abstractions:
  - Abstractions can be represented more easily, more directly, and more explicitly in the object paradigm.
  - **Software reuse** of classes is a major benefits of object-oriented programming.

Mapping Abstractions to Software



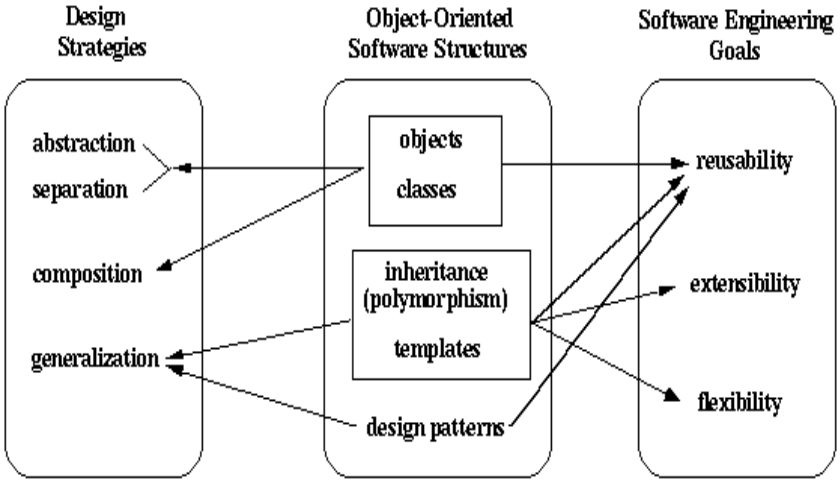
Software Objects

- **Class definition**:
  - Allows the **common structure** to be defined once and
  - **Reused** to create new objects that need that structure.
- The two main parts of an **object** are:
  - **Implementation**: data and methods are hidden inside the object.
  - **Interface**: the signature of all methods that are visible outside of the object.
- **Encapsulation**:
  - The **hiding** of the object's implementation details and data.
    - Provides *implementation independence*
  - **Restricted access** to only the non-private data members (usually methods) of the object.
  - **Object :** a distinct **instance** of a given class that **encapsulates** its implementation details and is structurally identical to all other instances of that class.

Design Strategies Embodied In Object-Oriented Programming

- **Abstraction** - simplifying to its essentials the description of a real-world entity.
- **Separation** - treating independently "what" an entity does from "how" it does it.
- **Composition** - building complex "whole" systems by assembling simpler "parts" in one of two basic ways:
  - **association**
  - **aggregation**
- **Generalization** - identifying common elements among different entities.
  - That's what classes and subclasses help provide.
  - So do Design Patterns.

Connections Among Strategies, Structures and Goals

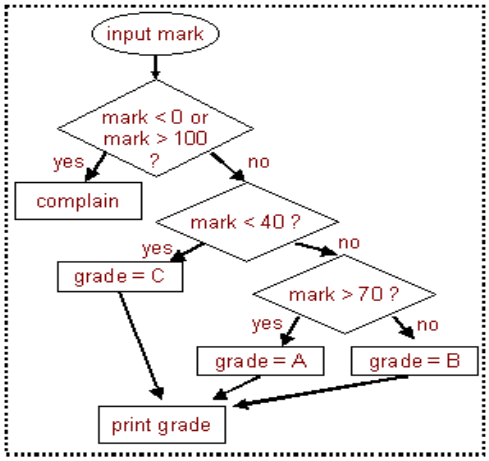


Polymorphism describes the use of variables which may refer at run-time to objects of different classes.

Pseudo-code and Flowcharts

- Two widely used notations for developing algorithms are *pseudo-code* and *flowcharts*.
  - A *flowchart*:
    - A diagram containing lines representing all the possible piecewise paths through the program,
    - These lines connect geometric shapes which represent conditional branching, subordinate steps, etc.
  - *Pseudo-code* is a form of "stylised" (or "structured") natural language.
    - Less formal than flowcharts, potentially more expressive, tho more potential for ambiguity.

Example: Flow charts & Pseudo-Code 1.5



Flow Chart

Pseudo-Code

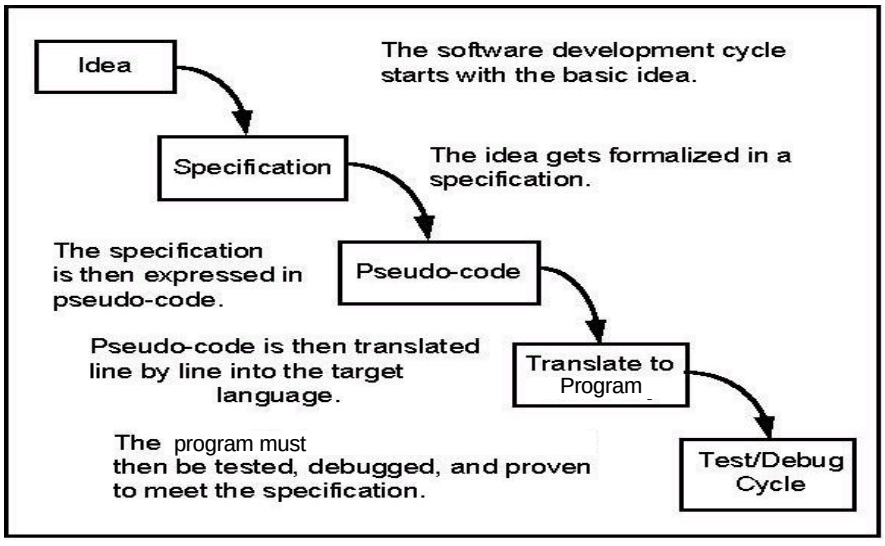
1. input mark
2. complain if mark below 0 or above 100
3. determine grade from mark (>70 = A; 40-70 = B; <40 = C)
4. print mark

"tools" to help you design/refine/debug algorithms

Pseudo code

- *Pseudo code* is a fast and effective tool that facilitates **well structured** program code.
- It provides an **intermediate step** in the process of translating from a mental conceptualization to a more precise specification.
- Pseudo-code is *less formal and detailed* than a programming language such as Java, but more precise than simple English - it need *not* be syntactically perfect.
  - It allows the programmer **to think in terms of the program flow** rather than in terms of specific language syntax.
- Pseudo code can be embedded in the target code as **comments** to provide a clear and succinct description of the program flow.
  - Additional documentation should be added to identify program details and clarify program operation.

Pseudo code for Program Development



Pseudo Code - Provides an ***Abstraction***

- For example consider the following algorithm to calculate the flight time of an aircraft using information on the timetable:

  Look up departure time
  Look up arrival time
  Subtract departure time from arrival time

- This algorithm will usually give the correct result, but the subtraction will have to take into account the special case when the plane arrives on the day after departure. Also, what about:

  Different time zones ?
  Daylight savings time ?

- ***Pseudo-Code enables details to be elided now so that essential characteristics are visible without distraction.***

- Thus the designer of an algorithm must ensure:
  - Preciseness of the algorithm
  - All possible circumstances are handled
  - Termination of the algorithm

Pseudo-Code Example

- **Play the game** (main method): // "Eight-Off" game
  - Set up the game
  - while game has not been won
    - print the game status
    - prompt user and read next move
  - Print an announcement that the game was won

- **Set up the** game (method):
  - create Deck arrays foundation[0..3], reserve[1..8], and column[1..8] (reserve[], column[] start at 0, but ignore that position) ....
- **Test if the game has been won** (method**)**: ….
- **Print the game status (foundations, reserves, columns)**
  - print "Foundations (0) "
  - for each foundation f :
    - if f is empty then print "---- "
    - otherwise print top card of f and two spaces ....
- **Prepare for the move from source to dest** (method) :
  - declare s and d to be Decks, initially set to null
  - verify src, dst are in the range -8..8 and that src is non-zero …..