Iterative Software Development

Users and **programmers** ** **cooperate** **
to produce a **usable** and **functional** system.

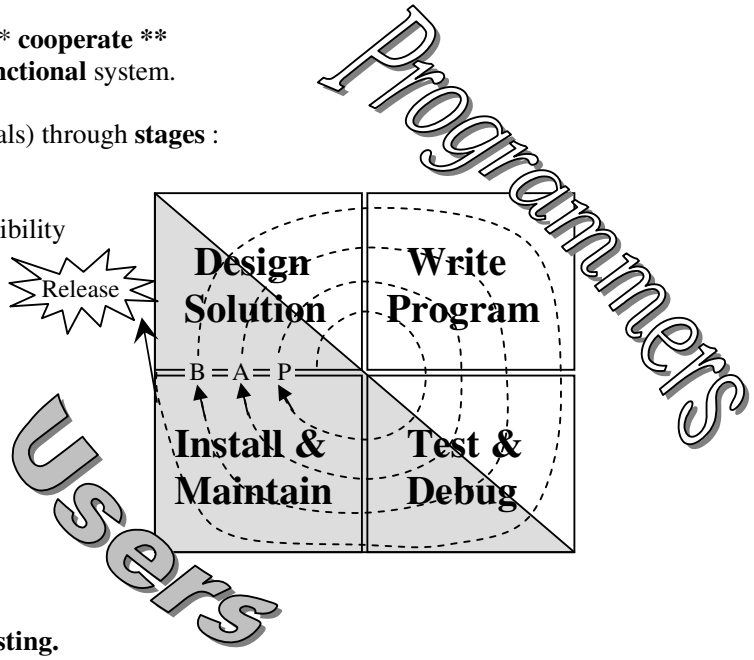
Development **iterates** (spirals) through **stages** :

Prototype
- proof of concept, feasibility

Alpha version
- barely functioning

Beta version
- *almost* finished

Release version
- Finished ! (?)



Programmers are not involved in installation or maintenance, just as users are not involved in writing the program. They **cooperate** in **design** and **testing.**

Investigate

If you can't play chess, you probably can't write a program that plays chess.

The programmers (in the real world they're called **developers**) must **investigate** the problem and **learn** something about the users' needs and problems. This facilitates later discussion and cooperation with the users. It also gathers enough information to produce a **proposal** or a **prototype.**

Prototype

The first loop around the spiral produces a **prototype** - a very simple version of the program - to **show** to the users what you "have in mind". Then you can have a **discussion** about what to do next - what to add, how to change the way it looks, etc. Most users **assume** that software development is automatic. They want to sit back and wait for the result to be produced. They are not able to engage in a useful, productive discussion about the design. The prototype helps them discuss the project.

Do the prototype quickly, don't spend much time on it. **Assume** that you will throw most of it away!

Redesign

Now the real design can start. The **user** has a better idea of the direction of the project. The **programmer** has a better idea of some of the problems they will face when writing the program.

Needs → Goals → Tasks → Interfaces + Events → Data Structures + Algorithms → Program

---------- User ---------- →

← ---------- Programmer ----------

The user carries responsibility for the left side, the programmer takes responsibility for the right side.
In the middle, some discussion is necessary to produce **interfaces** which are **both usable** and **functional.**
Remember : the user will be **part of the system -** they will need to make their parts function later.

Top-Down Design

The design of the solution must **start** with the user's needs, and **end** with a program written in a programming language - not the other way around. The **development cycle** allows feedback and redesign to occur, when it becomes obvious that some tasks or goals are unrealistic and cannot actually be programmed. However, working in the other direction (bottom-up) usually produces programs which are too simple and difficult to use.

No mess! Clean Up & Get Organized

Java provides a very sensible structure for successful top-down design and program production - this explains why it is so popular. However, it allows **too much flexibility** for a beginner (e.g. IB student). A big Java program can become very **messy** if you are not careful. The following **production model** should help you produce a project which is **well organized** and conforms to the **IB requirements.**

1. **Goals**
2. **Tasks** → **Menus**
3. **Interfaces** → **Frames = text-boxes + buttons + lists**
4. **Data Structures** → **Variables , Records, Arrays , Files, etc.**
5. **Algorithms (automation)** → **Modules → Pseudo-Code → Subs + Functions**
6. **Programming** → **Standard commands + Procedures + Parameters**

Suggestion - Use ONE Form + Menus + Frames

Multiple forms seem cool and well organized, but as the program gets bigger they will cause more and more problems. It is tricky to share code and controls between forms. It is difficult to force the user to use only one form at a time, and even more difficult to write a program that works when several forms are open.

For each **task,** make a **menu item** that makes a different **frame (window)** appear. If you have several frames on top of one another, the command to make **FrameX** visible is:

```
new FrameX
```

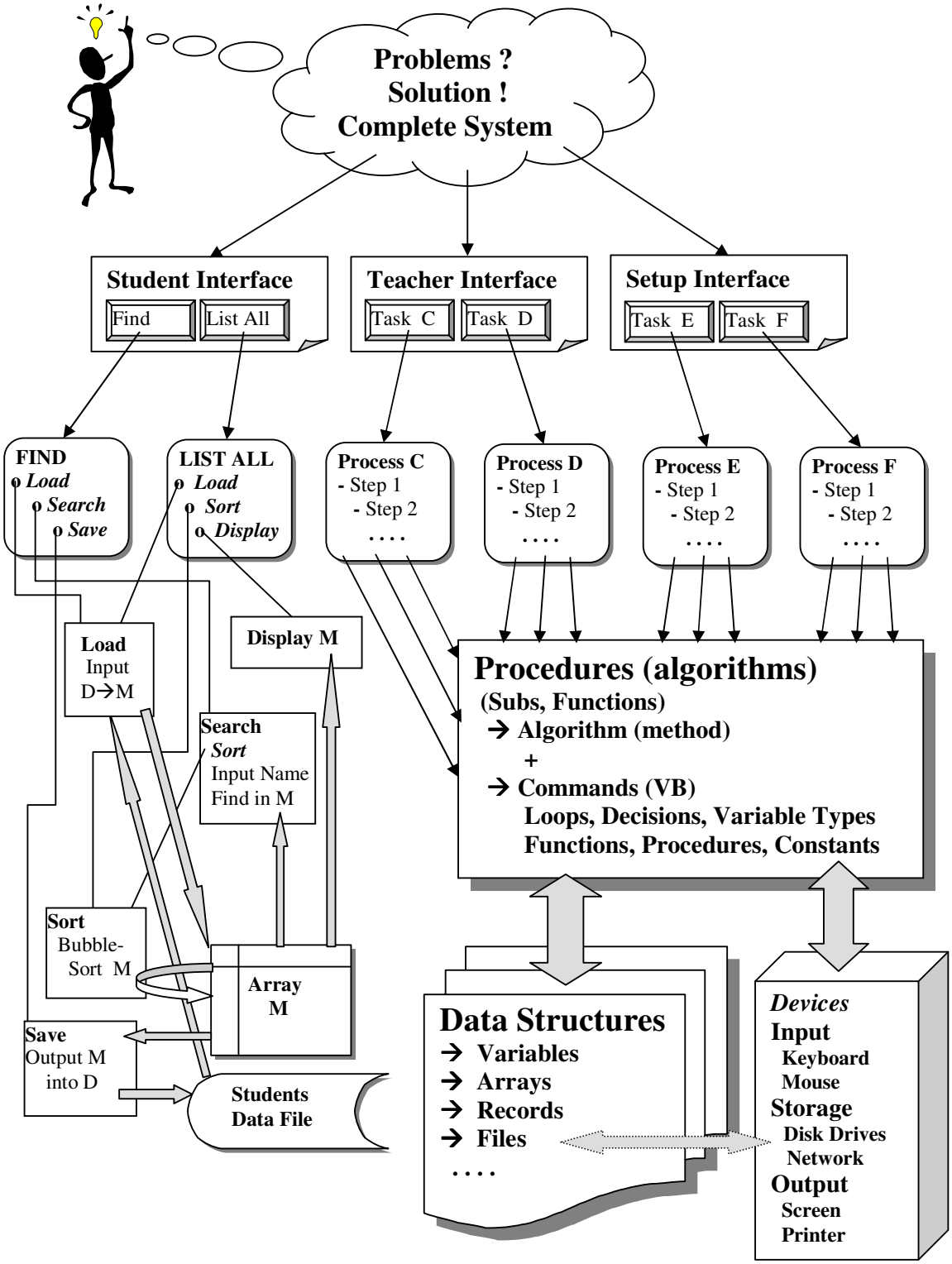
So you can write a **separate interface class** for each task. But don't get carried away – if you have more than 3 or 4 interfaces, you will WASTE TOO MUCH TIME on the user interface, and there is no reward for this in the IB assessment criteria.

Menues

Have a look in How-To at the Menues example. This is not a requirement, but you might find that it helps you organize your program.

Decomposition - Breaking Things Down

Your solution should be **hierarchical** (like an upside-down tree). It looks more or less like this:



Normally the **data-structures** do not appear on the same diagram with the **procedures.**
This was done here to illustrate the relationship between **algorithms** and **data-structures.**

Complexity

You will notice that as more and more **modules** (procedures and data-structures) are added, the diagram becomes unreasonably complex. The complex part is better displayed in the form of an **outline.**

It is tempting to write a new procedure from scratch for each new task. The apparent simplicity is deceptive. This produces many copies of similar or identical code. Any **changes** become very difficult later, as so many copies of the same code must be changed.

Re-Use - The Magic Pill for Successful Programmers

Without re-use, the tree becomes larger and larger at the bottom. Through re-use, there can be far fewer modules at the bottom. In the example, the **Load** and **Sort** procedures are **re-used** in two different tasks. Re-use keeps the program **shorter**, **simpler**, and **easier to change.**

Without re-use, the tree becomes larger and larger at the bottom. Through re-use, there can be far fewer modules at the bottom.

Naming

You need to adopt a **naming-convention**. Otherwise, re-use is extremely difficult. You will constantly be looking through your program trying to find the name of something. If the names of variables and procedures follow a pattern, you have a better chance of remembering them.

Many Small Modules

The best approach to re-use is to make every procedure as small as possible, confining it to a very specific job. Then combine and re-use these to accomplish many different tasks. Also use **parameters** to increase the flexibility of your modules. For example, a **sorting** module which can only sort one specific array is less useful than one which accepts the **array name** and then sorts that array.

Pseudocode

Write **pseudocode** BEFORE writing procedures. This lets you **design** the functionality and interface (parameters) sensibly. This takes time - you will feel like you are wasting time, writing the same ideas in pseudocode that you are going to write as Basic commands. But if you do this correctly, it will save time in the long run. As you write the pseudocode, you may realize that you are combining several tasks in one procedure, so you can split it up BEFORE you have written a long, complex procedure. You will also recognize possible problems and causes of errors ahead of time.

Documentation

Even with sensible names, you won't remember everything. This is where documentation comes in. Documentation includes:

- **Pseudocode** for all **algorithms -** at the beginning of all complex procedures - those which contain loops or if..then.. commands and do something difficult or interesting
- **Data structures -** a list of files and arrays and their intended purpose
- **Comments** in procedures, telling what various pieces do
- **Diagrams** on paper - the hierarchical diagram above is just one example
- **User documentation -** goals and instructions - this helps you remember what you were trying to do
- **Test data** - rather than making up new test data every time you run the program, it is much easier to **write down** a standard set of test data for each part of the program.

Good Names

Use variable and control names following a **naming convention –** it should be **consistent** and **clear.** For example:

Methods – the name should be a VERB, because it DOES SOMETHING

Variables – the name should be a NOUN, because it CONTAINS SOMETHING

Class – the names should be a NOUN

Controls – start with a letter or letters indicating the TYPE of the control, as suggested below:

"Official" Naming Convention
Start control names and variable names with 3-letter prefix

Control Names

Checkbox	chk
Combobox	cbo
Command Button	cmd
Form	frm
Frame	fra
Image	img
Label	lbl
Listbox	lst
Menu	mnu
Option Button	opt
Picture	pic
Shape	shp
Text Box	txt

Variable Names

Boolean	bln
Double	dbl
Date+Time	dat
Long	lng
Integer	int
Single	sng
String	str

"Official" Sample Code

```
Button btnCalc = addButton...

intAge = 21;
intBorn = 2001-intAge;
blnOK = checkAge(intAge);
if  (blnOK = = true)
{  output( "Okay") ; }
```

My Simple System
1 letter + Capitalization

Control Names

b Button	m Menu
k Checkbox	o Option Button
c ComboBox	p Picture
f Frame	s Shape
i Image	t TextBox
l ListBox	

Variable without prefix

My Simple Code

```
Button bCalc = addButton....

age = 21;
born = 2001 – age;
ok = checkAge(age);
if  ( ok = = true)
{  output("Okay"); }
```

Use **whole words** for names – you would only use a single letter for a **temporary counting variable.**

Comments

You MUST write comments BEFORE writing Java methods. This will help you clarify and focus your thinking, making the programming easier. Whether you find it easier or not, you MUST do it because it is a REQUIREMENT. You will be **heavily penalized** if you don't do this.

Each **method** requires starts with a comment containing the following:

- description / purpose
- parameters list (with brief explanations if the names are not clear)
- pre-conditions (before)
- post-conditions (after)
- return value
- **Pseudo-code** – English language step-by-step instructions. Be sure to mention loops if needed.

Sample Program - A sample program follows, showing proper **comments** and a consistent **naming convention.** This is only part of a program.

```
/* Dictionary.java
 *
 * Created on 14. November 2006, 09:51
 * @author  dave mulkey
 */

/* Dictionary ADT
 *
 * Stores pairs of Strings as (Item,Value).  For example:
 *
 *  "Font" , "Arial"
 *  "Color", "blue"
 *  "Author", "John Hancock"
 *  "Author", "Alfred Hitchcock"
 *  ...
 * Duplicates are permitted for Items.
 * Allows retrieving a Value by searching for an Item
 */
public class Dictionary {

    /*
     * items[] and values[] are parallel arrays
     * they will be created (and sized) in the constructor
     */
    String[] items ;
    String[] values;


    public Dictionary(int size)
    /* Creates a new Dictionary
     *
     *  Params: size = maximum number of item/value pairs
     *  Before: nothing required
     *  After : items[] and values[] have been created
     *          with the required size
     *
     *-- pseudocode -----------------
     *  Use new String[size] to create arrays
     */
    {
        items = new String[size];
        values = new String[size];
    }

    public Dictionary()
    /* Creates a new Dictionary
     *
     *  Params: none
     *  Before: nothing required
     *  After : items[] and values[] have been created
     *          with the required size
     *
     *-- pseudocode -----------------
     *  Use new String[1000] to create arrays
     */
    {   items = new String[1000];
        values = new String[1000];
    }

    public boolean add(String item, String value)
    /*  add a new Item/Value pair
     *--------------------------------
```

```
     *  Params :  item , value  contain Strings to add
     *  Before :  items[] and values[] have been created
     *  After  :  item/value pair have been added
     *            Return success (true) or failure (false)
     *---------------------------------
     *  Problems:
     *   must reject null item or null value (return false)
     *   if array is full, return false
     *---------------------------------
     * pseudocode:
     *  Use a loop to search for an empty spot in items[]
     *  If it reaches the end of the array (no free spots)
     *    then return false (failure)
     *  else
     *    copy item into items[] and value into values[]
     */
    {
       if (item == null || value == null)
       {  return false; }
       int p = 0;
       while (p < items.length && items[p] != null)
       {  p = p + 1; }
       if (p < items.length)
       {
           items[p] = item;
           values[p] = value;
           return true;
       }
       else
       {   return false; }
    }

    public String getValue(String target)
    /*  Search for target in items[]
     *  if found, returns matching value
     *  else returns null
     *---------------------------------
     *  Params : target is the String to find in items[]
     *  Before : items[] and values[] must exist
     *  After  : return matching value if found,
     *           else return null
     *---------------------------------
     * pseudocode:
     *   loop through the items[] array
     *      if items[p].equals(target)
     *         return values[p]
     *   if we get here, return null (not found)
     */
    {
        for (int p=0; p < items.length; p++)
        {
            if (items[p]!=null)
            { if (items[p].equals(target))
              {  return values[p]; }
            }
        }
        return null;
    }
}
```