MT 3607: Computing in Statistics

Lecture 5: Good Programming Practice

Writing code that works!

Len Thomas

2 October 2013

What is Good Programming Practice?

- Good programming practice is a set of ideas and rules designed to:
  - Minimize the amount of time required to develop working code
  - Minimize the number of programming errors
  - Maximize the re-use potential of code
- Use in conjunction with suggestions for testing and debugging (see next lecture)

1 Software design

1.1 The software life cycle

Typical software life cycle
The lifetime of a piece of software can be broken up into a set of stages:

- Requirements specification
- Design
- Implementation (i.e. coding)
- Verification (testing and debugging)
- Installation and useage
- Maintenance

1.2 Software development methodologies

Introduction

- Many software development projects are extremely large and costly:
  - Thousands of people, millions of pounds, years of time, millions of lines of code
- Need for a structured methodology to follow in order to increase the chances of successfully delivering the desired product
- Whole professions are devoted to this: systems analysts, software engineers, ...
- Here, we'll just touch on the topic (as we usually work on much smaller projects!). For more see:
  - CS3051 Software engineering
  - Somerville, I. 2004. Software engineering, 7th Edition. Pearson education.

Models of software development

- There are many different models that describe how successful software gets developed
- Two contrasting examples:
  - Waterfall model
  - Iterative model

Waterfall model (≈ "Top-down")

- Execute the stages of software development in a rigid order e.g.: requirements definition, design, implementation, installation.
- Verify that each stage has been completed before moving to the next
- Emphasis on comprehensive documentation and verification at each stage
- Example methodology that uses this model:
  - SSADM – Structured Systems Analysis and Design Methodology
- Advantages:
  - Time spent early on in exactly defining requirements and designing the system saves money and time later on.
    - Analogy: designing and building a house
  - You know what you're getting, and everyone agrees on it.
  - Extensive documentation is a good strategy for large teams, and provides robustness in the face of staff turnover.
- Disadvantages:
  - Inflexible: Can't deal with changing requirements
  - Can't go back and correct earlier mistakes

Iterative model (≈ "Bottom-up")

- Philosophy: start small and build up
- Build the whole system incrementally:
  - Start with a simple implementation of a subset of the requirements
  - Iteratively enhance (adding new capabilities and making design modifications) until system is complete
- Related ideas:
  - Agile development methods
  - Rapid application development
- Example methodology:
  - DSDM – Dynamic Systems Development Methodology
- Advantages:

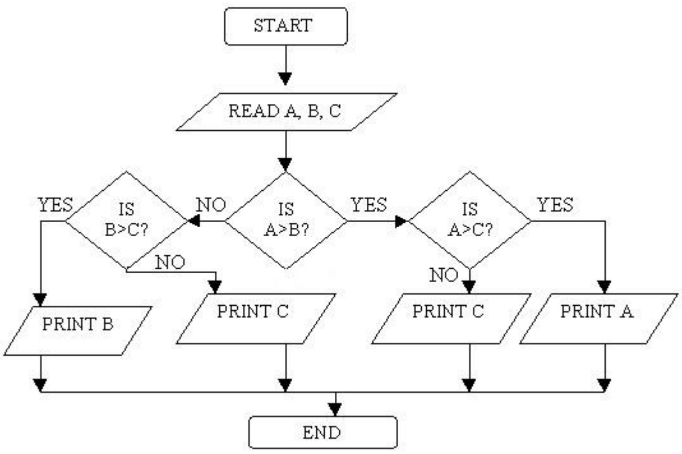
– Flexible – can respond to changing requirements
  – Good for customers who don't know exactly what they want
  – More human-focussed – emphasis on working in small, flexible teams
  – More efficient than waterfall method

- Disadvantages:
  – Not sure what you're going to get until the end
  – Less robust than waterfall method

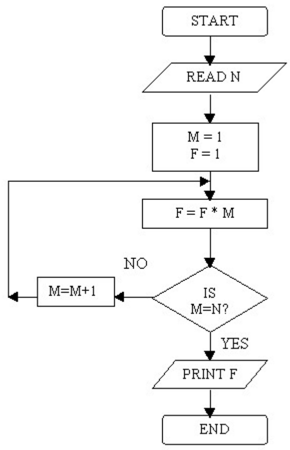
1.3 Tools for software design

Introduction

- Software engineers have developed a number of tools (many visual) to help document systems, and plan how the software will work
  – Process flow diagrams
  – Data flow diagrams
  – Entity relationship diagrams
  – Flowcharts



Find largest of 3 numbers



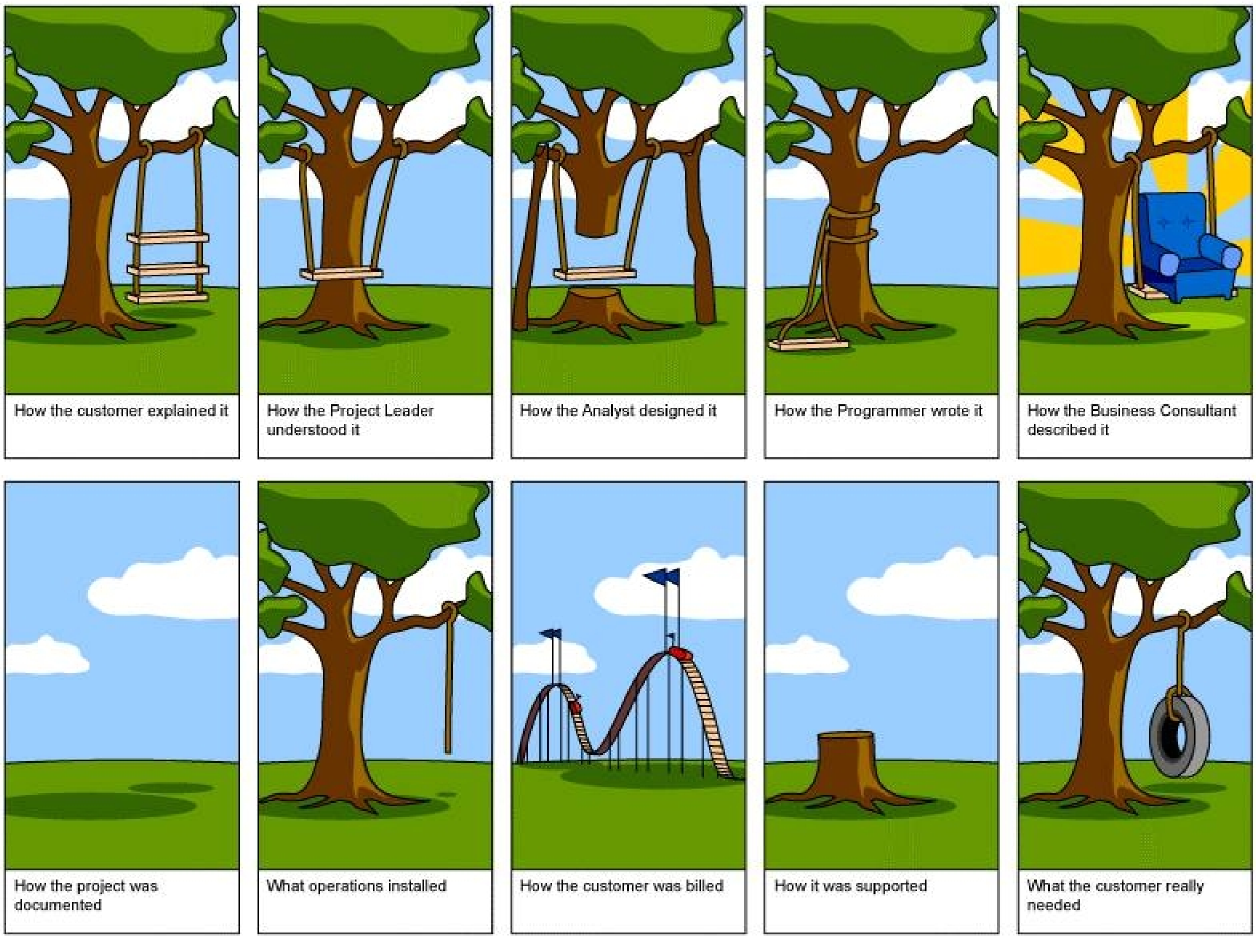
Compute factorial N

1.4 Practical software design

Software project failure

In practice, even projects that rigorously follow the latest design methodology are prone to failure. Especially true for large projects.[1]

[1]Warning: This cartoon is an example of software engineer's humour – may not be funny to ordinary human beings.



Software design for us folks

- For the projects we're involved in, it's rarely worth employing a formal design methodology
- However, it **is** worth considering design **before** implementation
- The larger the project, the more formal we want to be about the design
- In practice, we tend to blend top-down and bottom-up methods
  - Relative emphasis on each depends mostly on the context and on personal taste
  - Personally, I've tended to favour top-down methods for the projects I've been involved in so far
  - Why?
    * Requirements have been pretty clear at the outset
    * I believe that time spent thinking about design saves time later
    * I like starting with generalities and working down to specifics
    * I have a poor memory, so documentation is good!

My approach to software design
My approach (feel free to ignore!):

- For a very small job, I sketch some pseudocode on a piece of paper, and then start coding. (Often no modules (functions) needed.)
- For a larger job, I use a top-down approach:
  - Divide up the task into sub-tasks

- Turn these into module outlines – think of names and inputs/outputs
- If a module looks like it's going to be large, split into sub-modules
- Usually do all this in a text editor, so I have a record of it
- Code up a prototype/skeleton of the program, with "empty" modules that are all called by the main program, but do nothing inside.
- Then, start on the main job of coding the modules.

• For a very large job (e.g., Distance).

  - Resort to some formal methods.
  - Keep extensive "internal documentation".

2 Turning designs into code

2.1 Where to get code from

How to turn the module specifications into code
There are two approaches to this:

- Write the code yourself
- Get it from someone else (not for assignments on this course!)

Getting code/routines from someone else - a.k.a. "code theft"

- Avoid re-inventing the wheel!
  - R has an extremely rich set of built-in functions (use `help.search()` or the menu item `Help | Seach help`
  - R also has an enourmous number of add-on packages (see `Help | search r-project.org` and other resources described in Lecture 1)
  - In general, there are many code libraries freely available online
  - There are also commercial plug-ins (e.g., IMSL and NAG for 3GLs, again, see Lecture 1)
- If you are using code written by someone else (especially if it's non-commercial)
  - Acknowledge them in your code and write-ups
  - Thoroughly test it
  - If feasible, try to understand how the code works
- Other people's code is a great way to get a template for code you want to write. You can then modify the code to your own ends (after giving it a new function name).
  - For R, you can download the source code to the program
  - These end up in directories called `/src` below the main R directory – e.g., the code for the function `AIC()` is in the directory `/src/library/stats/R/AIC.R`. They don't have to be in a file with their name, so you may have to search a bit.
  - For add-on packages, you need to download the source code for each one individually.
  - These end up under `/library` - e.g., the function `gls()`, which is part of the add-on package `nlme` is in `/library/nlme/R/gls.R`.
- Warning! - R code in the base system and add-on packages can be poorly written, have few comments, and will generally used advanced techniques (such as classes) that aren't covered on this course.

Writing your own module code

- Inputs and outputs should already have been decided
- Double-check that someone else hasn't written it already!
- Determine how the module will work – the algorithm. This should be the hard part.
- Consider sketching it down as flowchart/pseudocode, especially if complex.
- Now start coding
- Start with the comments – lay out the function in comments
- Then fill in the blanks with code!
- Test and debug (see next lecture)

2.2 Programming style – general

General tips for writing good code

- Keep it local! Use principle of least privilege (even in R)
- "Optimization is the root of all evil" (Knuth)
  - Working code is better than fast code (which is why there are no optimization tips in this lecture)
  - But a balance has to be struck – e.g., vectorization in R
- Minimize repetition of very similar code – hard to maintain.
  - Turn big chunks of repeated code into modules.
  - Within modules, write code that avoids repeats
- From a previous lecture:

```
if(Smoke[i]==0) {
  cat("Subject ",i," a Non-smoker has Heartrate", Heartrate[i],"\n")
} else {
  cat("Subject ",i," a Smoker has Heartrate", Heartrate[i],"\n")
}
```

- Better:

```
cat("Subject ",i," a ")
if(Smoke[i]==0) {
  cat("Non-smoker")
} else {
  cat("Smoker")
}
cat(" has Heartrate", Heartrate[i],"\n")
```

- Better still:

```
if(Smoke[i]==0) {
  smoke.text<-"Non-smoker"
} else {
  smoke.text<-"Smoker"
}
cat("Subject ",i," a ",smoke.text," has Heartrate", Heartrate[i],"\n")
```

- Better still (in R, anyway):

  ```
  smoke.text<-ifelse(Smoke==0,"Non-smoker","Smoker")
  ```

  Then, when you want it printed out, inside your `i` loop:

  ```
  cat("Subject ",i," a ",smoke.text[i]," has Heartrate", Heartrate[i],"\n")
  ```

- Or (if 0==non-smoker and 1==smoker), using a hash (lookup) table:

  ```
  smoker.status.as.text<-c("Non-smoker","Smoker")
  ```

  Then, inside the loop:

  ```
  cat("Subject ",i," a ",smoker.status.as.text[Smoke[i]+1]," has Heartrate",
    Heartrate[i],"\n")
  ```

- Or, using a function:

  ```
  smoker.status.as.text<-function(status.as.int){
    #Purpose: Returns "Non-smoker" or "Smoker" depending on input status
    return(ifelse(status.as.int==0,"Non-smoker","Smoker"))
  }
  ```

  Then, inside the loop:

  ```
  cat("Subject ",i," a ",smoker.status.as.text(Smoke[i])," has Heartrate",
    Heartrate[i],"\n")
  ```

- Declare all variables and their types at the beginning of the module (not possible in R, unfortunately)
- Index variables (e.g., in `for` loops) can either start at 0 or 1 – be consistent (in R they usually start at 1)
- Use constants wherever possible
  - Constants can replace all explicit numbers (tolerence, etc) and characters (error messages) – puts them all in one place in your code
  - Harder in R, but can still use variables in the same way.
  - Usually not done for characters in R, however.
- Be careful when dealing with floating point numbers (see Lecture 1)
- Many higher-level functions are just a bunch of error checks and then calls to lower functions

2.3 Programming style – within modules

Tips for writing good code within modules

- Initialize everything! (can take with a pinch of salt)
- Put in lots of error checks
  - Check arguments are okay at the start of functions
  - Before certain operations
  - In R, can use `try()` to avoid the program stopping when a function call causes an error.

    ```
    res<-try(function())
    if((class(res)[1]=="try-error")) {
      #code here to deal with an error in the function
    } else {
      #code here for when the function works
    }
    ```

- Consider spreading complex expressions out over multiple lines, assigning intermediate values to a variables.
  - Instead of

    ```
    if ((((x<6)&(x>7))|((y<0)&(y>1)))&(theta>=0)&(theta<2*pi)) {
    ```

  - Could say

    ```
    x.ok <- (x<6) & (x>7)
    y.ok <- (y<0) & (y>1)
    theta.ok <- (theta>=0) & (theta<2*pi)
    if((x.ok | y.ok) & (theta.ok)) {
    ```

  - Increases readabilty and makes it easier to debug.
- Don't have too many nested control structures (say 3-4)
  - If you have this much nesting, consider putting some of it into new modules

2.4 Coding conventions

What are coding conventions?

- Coding conventions are a set of rules for writing consistent, nice looking code
- This
  - Increases readability
  - Decreases errors
  - Maximizes reuse potential
- On large software development projects, there are a written set of specifications for every aspect of code layout, variable naming, etc.

Suggested coding conventions

- Separate modules with some kind of delimiter, e.g.

```
#---------------------------------------
function code goes here
#---------------------------------------
next function goes here
#---------------------------------------
```

- Start all modules (functions) with comments giving:
  - Purpose
  - Interface (inputs, outputs, any global effects)
  - Any special implementation details (any clever tricks, things to watch out for)
- Use comments liberally throughout the code to explain what's going on
- **Make sure the comments stay up to date!**
- Use indenting and line spacing to give code a 2-d feel
  - Indent code inside control blocks
  - Group blocks of related code, then leave a line
  - Wrap lines after about 80 characters
  - ```
    } #end of for loop

    #save the best model
    best.model<-0
    min.aic<- +Inf
    for(iteration in 1:max.iterations) {
      if(!is.na(aic.history[iteration])){
        #after the loop has finished, this will be the number of iterations in
        # which models were fit
        num.iterations<-iteration
        #check if this one is the best model (needn't have been as can exit
        # before finding the best model)
        if(aic.history[iteration]<min.aic) {
          min.aic<-aic.history[iteration]
          best.model<-iteration
        }
      } #end if aic.history is na
    } #end interation

    #trim the vectors in res, and add summary stuff
    res<-list()
    res$call.history<-call.history[1:num.iterations]
    ```
- Use meaningful variable names
- Have a convention for naming variables and functions
  - In R, we tend to use a combination of upper and lower case, and dots:
    * Either VariableName or Variable.Name or variableName or variable.name
    * Use whatever suits you
  - In other 3GLs, more formal conventions exist – e.g.
    ```

    - A prefix to denote the type of variable (int, dbl, str)
    - Another prefix for global variables (gint, gdbl, gstr)
    - ALL_UPPERCASE for constants
    - etc.

- If you're declaring variables at the beginning of the module (not R) then add comments to explain what they are – otherwise do so in the code where it might not be clear

- Be consistent with the spacing around operators e.g., `x <- y + 2` or `x<-y+2`

- In R, use <− rather than =, and don't confuse it with ==