5.1 Stepwise Refinement

- A basic principle underlying many software engineering techniques
  - "Postpone decisions as to details as late as possible to be able to concentrate on the important issues"

- Miller’s law (1956)
  - A human being can concentrate on 7 ± 2 items at a time

5.1.1 Stepwise Refinement Mini Case Study

- Design a product to update a sequential master file containing name and address data for the monthly magazine *True Life Software Disasters*

- Three types of transactions
  - Type 1: INSERT (a new subscriber into the master file)
  - Type 2: MODIFY (an existing subscriber record)
  - Type 3: DELETE (an existing subscriber record)

- Transactions are sorted into alphabetical order, and by transaction code within alphabetical order
Decompose Process

- No further refinement is possible

First Refinement

- update master file
- exception report
- summary and end-of-job message

Stepwise Refinement Case Study (contd)

- Assumptions
  - **INPUT** - we believe we can produce the right record when needed by **PROCESS** and
  - **OUTPUT** - we believe we can write the correct record to the correct file at the right time when **PROCESS** says to
- concentrate on **PROCESS**

Stepwise Refinement Case Study (contd)

- What is this **PROCESS**?
  - Example:

**Transaction file**

- Abel
- Brown
- Harris
- James
- Jones
- Smith
- Townsend

**Old master file**

- Abel
- Brown
- Harris
- James
- Smith
- Townsend

**New master file**

- Abel
- Brown
- Harris
- James
- Smith
- Townsend

Stepwise Refinement Case Study (contd)

- More formally:

  1. **INSERT**: Print error message
  2. **MODIFY**: Change master file record
  3. **DELETE**: Delete master file record

  Transaction record key = old master file record key

  Copy old master file record onto new master file

  Transaction record key > old master file record key

  1. **INSERT**: Write transaction record to new master file
  2. **MODIFY**: Print error message
  3. **DELETE**: Delete master file record

  Transaction record key < old master file record key

  *Deletion of a master file record is implemented by not copying the record onto the new master file.*

Second Refinement
Third Refinement

- Now we work on refinement of I/O boxes. But this design has a major fault. Consider 2 Jones and 3 Jones

Stepwise Refinement Case Study (contd)

- The third refinement is **WRONG** - "Modify JONES" followed by "Delete JONES" is incorrectly handled
- We find this because we check each successive refinement before proceeding to the next
- If a fault is found, go back to previous refinement and proceed from there to create another third refinement

Stepwise Refinement Case Study (contd)

- After the third refinement has been corrected
  - Details like opening and closing files have been ignored up to now
  - Fix these in the fourth refinement after the logic of the design is complete
  - The stage at which an item is handled is vital
- Opening and closing files is
  - Ignored in early steps (to keep within Miller’s Law’s limits, but)
  - Essential later

Appraisal of Stepwise Refinement

- A basic principle used in
  - Every workflow
  - Every representation
- The power of stepwise refinement
  - The software engineer can concentrate on the relevant aspects
- Warning
  - Miller’s Law is a fundamental restriction on the mental powers of human beings

5.2 Cost–Benefit Analysis

- Outside of scope of this class

5.3 Software Metrics - skim

- Outside of scope of this course
- Used for project and process management and improvement
- To detect problems early, it is essential to measure
- Examples:
  - LOC per month
  - Defects per 1000 lines of code
Different Types of Metrics - skim

- **Product metrics**
  - Examples:
    - Size of product
    - Reliability of product

- **Process metrics**
  - Example:
    - Efficiency of fault detection during development

- **Metrics specific to a given workflow**
  - Example:
    - Number of defects detected per hour in specification reviews

---

The Five Basic Metrics - skim

- **Size**
  - In lines of code, or better

- **Cost**
  - In dollars

- **Duration**
  - In months

- **Effort**
  - In person months

- **Quality**
  - Number of faults detected

---

5.4 CASE (Computer-Aided Software Engineering)

- **Scope of CASE (not A for automated)**
  - CASE can support the entire life-cycle
    - Estimating resource req.
    - Drawing up the req. specification doc.
    - Performing integration testing
    - Writing the user manual
    - Etc.

- **The computer assists with drudge work**
  - It manages all the details
  - Human intervention still needed

---

5.5 Taxonomy of CASE

- **Tool - product that assists in just one aspect of the creation of software**
  - UpperCASE (front-end tool such as during requirements, analysis, design workflows) versus
  - LowerCASE (back-end tool such as during implementation workflow and postdelivery maintenance)

---

Some Useful Tools

- **Data dictionary**
  - Computerized list of all data defined within the product
    - Variable name, type, definition location, description
    - Procedure name, parameters, parameter types, prototype desc

- **Consistency checker**
  - checks that every data item is found in the design and every design item has been defined in spec. doc.

- **Report generator, screen generator**
  - produces code for report or for screen capture of data

---

Taxonomy of CASE (contd)

- **(a) Tool versus (b) workbench (Software through Pictures)**
  - versus (c) environment (Oracle Developer Suite)
5.6 Scope of CASE

- Programmers need to have:
  - Accurate, up-to-date versions of all project documents
  - Online help information regarding the
    - Operating system
    - Editor
    - Programming language
  - Online programming standards
  - Online manuals
    - Editor manuals
    - Programming manuals

Scope of CASE (contd)

- Programmers need to have:
  - General purpose tools
    - Email systems
    - Spreadsheets
    - Word processors
  - Coding tools
    - Structure editors - understands the implementation language
    - Pretty printers - part of structure editor that formats code for good visual appearance
    - Online interface checkers - part of structure editor that checks full interface information (method names, # parms, parm types) sometimes with template generation when method name is typed

Online Interface Checker

- Programming-in-the-small (programming at the individual module level by 1 programmer) uses structure editors and pretty printers
- Interface checking is an important part of programming-in-the-large (programming multiple modules which involves aspects of architecture and integration) and of programming-in-the-many (sw production by a team involves both aspects of programming-in-the-small and programming-in-the-large)

Online Interface Checker (contd)

- Example
  - The user enters the call
    ```java
    average = dataArray.computeAverage(numberOfValues);
    ```
  - The editor immediately responds
  ```java
  Method computeAverage not found
  ```
  - The programmer is given two choices
    - Correct the name of the method to `computeMean`
    - Declare new procedure `computeAverage` and specify its parameters
  - This enables full interface checking

Online Interface Checker (contd)

- Advantages
  - There is no need for different tools with different interfaces
  - Hard-to-detect faults are immediately flagged for correction
    - Wrong number of parameters
    - Parameters of the wrong type
- Essential when software is produced by a team
  - If one programmer changes an interface specification, all components calling that changed artifact must be disabled

Example

- Declaration of `q` is
  ```java
  void q (float floatVar, int intVar, String s1, String s2);
  ```
  - Call (invocation) is
    ```java
    q (intVar, floatVar, s1, s2);
    ```
    - The online interface checker detects the fault

Help facility

- Online information for the parameters of method `q`
  - Better: Editor generates a template for the call
    - The template shows type of each parameter
    - The programmer replaces formal by actual parameters

This enables full interface checking

Advantages

- There is no need for different tools with different interfaces
  - Hard-to-detect faults are immediately flagged for correction
    - Wrong number of parameters
    - Parameters of the wrong type
- Essential when software is produced by a team
  - If one programmer changes an interface specification, all components calling that changed artifact must be disabled
Online Interface Checker (contd)

- Even when a structure editor incorporates an online interface checker, a problem remains:
  - The programmer still has to exit from the editor to invoke the compiler (to generate code)
  - Then, the linker must be called to link the product
  - The programmer must adjust to the JCL, compiler, and linker output

- Solution: Incorporate an operating system front end into the structure editor

Operating System Front-End in Editor

- Single command
  - `go` or `run`
  - Use of the mouse to choose
    - An icon, or
    - A menu selection

- This one command causes the editor to invoke the compiler, linker, loader, and execute the product

Source Level Debugger

- Example:
  - Product executes terminates abruptly and prints
    - Overflow at 4806
    - Core dumped
    - Segmentation fault

- The programmer works in a high-level language, but must examine:
  - Machine-code core dumps
  - Assembler listings
  - Linker listings
  - Similar low-level documentation

- This destroys the advantage of programming in a high-level language

- We need:
  - An interactive source level debugger (like `dbx`)

Source Level Debugger (contd)

- Output from a typical source-level debugger

```
OVERFLOW ERROR
Class: cyclotronEnergy
Method: performComputation
Line 6: newValue = (oldValue + tempValue) / tempValue;
    oldValue = 3.9583       tempValue = 0.0000
```

Programming Workbench

- Structure editor with
  - Pretty printer
  - Online interface checking capabilities
  - Operating system front-end
  - Online documentation
  - Source level debugger

- This constitutes a simple programming environment
This is by no means new
- All the above features are supported by FLOW (1980)
- The technology has been in place for years

Surprisingly, some programmers still implement code the old-fashioned way
- Commercial - Sun Java Studio
- Open-source - Eclipse

During maintenance, at all times there are at least two versions of the product:
  - The old version, and
  - The new version

There are two types of versions:
  - Revisions (written to replace its predecessor) and
  - Variations (designed to coexist)

Revision
- A version constructed to fix a fault in the artifact
- We cannot throw away an incorrect version
  - Some sites may not install the new version

Perfective and adaptive maintenance also result in revisions

A variation is a version for a different operating system–hardware
Variations are designed to coexist in parallel

Version-Control Tool
- Essential for programming-in-the-many
  - A first step toward configuration management
- A version-control tool must handle
  - Updates
  - Parallel versions
Version-Control Tool (contd)

- Notation for file name, revision, and variation

![Diagram of file name, revision, and variation notation](image)

Figure 5.13

- Problem of multiple variations
  - Deltas (list of differences from stored variation to another variation)

- Version control is not enough — because of maintenance issues we need configuration control

5.8.1 Configuration Control during Postdelivery Maintenance

- Two programmers are working on the same artifact mDual/16
  - mDual version 16

- The changes of the first programmer approved and are saved as mDual/17

- The changes of the second programmer are approved and saved as mDual/18
  - The changes of the first programmer are lost

5.8.2 Baselines

- The maintenance manager must set up
  - Baseline (set of versions of all artifacts in product)
  - Private workspaces (place where prog. Investigates maintenance ideas)

- When an artifact is to be changed, the current version is frozen by one programmer
  - Thereafter, it can never be changed

5.8.3 Configuration Control during Development

- Both programmers make their changes to mDual/16

- The first programmer
  - Freezes mDual/16 and makes changes to it
    - The resulting revision is mDual/17
    - After testing, mDual/17 becomes the new baseline

- The second programmer
  - Freezes mDual/17 and makes changes to it
    - The resulting revision is mDual/18
    - After testing, mDual/18 becomes the new baseline

- While an artifact is being coded
  - The programmer performs informal unit testing

- Then the artifact is given to the SQA group for methodical testing

- An artifact must be subject to configuration control from the time it is passed by SQA
Configuration-Control Tools

- UNIX version-control tools
  - sccs
  - rcs
  - cvs
  - subversion (svn)
- Popular commercial configuration-control tools
  - PVCS
  - SourceSafe
- Open-source configuration-control tool
  - cvs
  - subversion

5.9 Build Tools

- If no configuration control tool, then use version control with build tool - example
  - UNIX make
- A build tool compares the date and time stamp on
  - Source code, compiled code
  - It calls the appropriate compiler only if necessary
- The tool then compares the date and time stamp on
  - Compiled code, executable load image
  - It calls the linker only if necessary

5.10 Productivity Gains with CASE Tools

- Survey of 45 companies in 10 industries (Myers, 1992)
  - Half information systems
  - Quarter scientific software
  - Quarter real-time aerospace software
- Results
  - About 10% annual productivity gains
  - Cost of introducing CASE technology: $125,000 per user

5.10 Productivity Gains with CASE Tools (contd)

- Newer results on fifteen Fortune 500 companies (1997)
- It is vital to have
  - Training, and
  - A software process
- Results confirm that CASE environments should be used at CMM level 3 (managerial and technical aspects are defined and measured, improvements are attempted, and reviews are used for quality) or higher
- “A fool with a tool is still a fool”