

Software Engineering

CS305, Autumn 2020

Week 10

Class Progress...

- Last Week
 - RUP phases,
 - Software Construction
 - Inspections/Reviews
- This week
 - Software Construction
 - Coding
 - Refactoring
 - Introduction to testing and unit testing (if time permits)

Coding

Coding

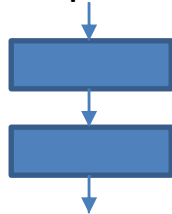
- Could involve:
 - Writing source code / programming in a chosen language
 - Automatic generation of source code using a design representation of the component to be constructed
 - Automatic generation of executable code using a *fourth-generation* language – program generating language

Human understanding is facilitated by linear sequence of logical statements

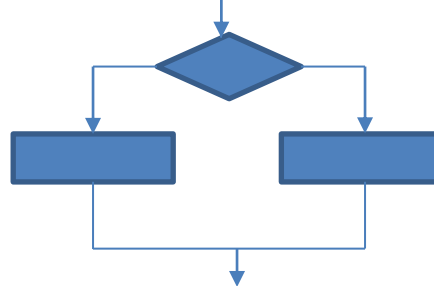
Programming Paradigms

- Unstructured Programming
 - Writing a sequence of commands or statements that access ‘Global’ data.
E.g. Assembly lang. programming.
- Structured Programming (sometimes used interchangeably with procedural programming)
 - Dijkstra’s advice on using simple logical constructs of:

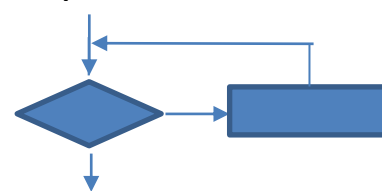
sequence



condition



repetition



- Focus on writing ‘modular’ programs. Have single-entry and single-exit for a procedure / function (control construct). E.g. C, Assembly lang. programming

Programming Paradigms

- Object Oriented Programming
 - Modeling real-world objects. Data is the centerpiece. Combine data and functions, allow code reuse, incremental dev. maintainability, modularity. *(more in Week3 lectures)*. E.g. C++, Java
- Functional Programming
 - Focus on what to do and not how to do. Don't create state that is changeable. E.g. Lisp, Racket
- Concurrent Programming
 - Focus on concurrent execution of a sequence of statements.
 - Parallel programming is a type.
 - E.g. Threads programming (Java threads), Open MP, MPI, CUDA-C.

Coding Principles

- Ensure that the problem is well-understood before coding (i.e. design is clear, programming language is clear)
- Follow Dijkstra's advice and create modular code that is highly cohesive and loosely coupled
- Select data structures that meet the design objectives
- Create readable code (have indentation, blank lines, and comments)
- Select meaningful names for variables, functions, and follow coding standards and best practices
 - tmp, temp, data are “symptoms of programmer laziness”.
 - (for GCC) <https://gcc.gnu.org/wiki/CppConventions>
- Get code reviewed by peers

Code Review – class exercise

- Review the following Fortran code

```
1  DOUBLE PRECISION FUNCTION SIN(X, E)
2  C      THIS DECLARATION COMPUTES SIN(X) TO ACCURACY E
3      DOUBLE PRECISION E, TERM, SUM
4      REAL X
5      TERM=X
6      DO 20 I=3,100,2
7      TERM=TERM*X**2/(I*(I-1))
8      IF(TERM.LT.E)GO TO 30
9      SUM=SUM+(-1**(I/2))*TERM
10     20 CONTINUE
11     30 SIN=SUM
12     RETURN
13     END
```


Code Review – class exercise

- Review the following Fortran code

- C is comment to end of line
- The `CONTINUE` statement is often used as a place to hang a statement label, usually it is the end of a `DO` loop. If the `CONTINUE` statement is used as the terminal statement of a `DO` loop, the next statement executed depends on the `DO` loop exit condition.
- `.LT.` is less than
- `**` is exponentiation (has higher priority than `*`)
- `DO label var = expr1, expr2, expr3`
 statements
Label `CONTINUE`

var is the loop variable (often called the *loop index*) which must be integer. *expr1* specifies the initial value of *var*, *expr2* is the terminating bound, and *expr3* is the increment (step).

```
1 DOUBLE PRECISION FUNCTION SIN(X, E)
2 C   THIS DECLARATION COMPUTES SIN(X) TO ACCURACY E
3   DOUBLE PRECISION E, TERM, SUM
4   REAL X
5   TERM=X
6   DO 20 I=3,100,2
7     TERM=TERM*X**2/(I*(I-1))
8     IF(TERM.LT.E)GO TO 30
9     SUM=SUM+(-1**(I/2))*TERM
10  20 CONTINUE
11  30 SIN=SUM
12     RETURN
13     END
```

Code Inspection Checklist (excerpt)

1. Data (DA)

- Is each variable correctly typed?
- Is each variable initialized before use?
- Is the initialization appropriate for the type?
- Can global variables be made local?
- Are buffer overflows checked?
- Is dynamically allocated memory freed?

2. Interface (IF)

- Are appropriate values returned from functions?
- Do function calls have correct parameter types/values?
- Are return values tested?

3. Functionality (FN)

- Do loops terminate?
- Do all loops iterate the correct number of times (no off-by-one errors)?

- Is behavior correct if a loop is never entered?
- Is there dead (unreachable) code?
- Do all switch statements have a default case?
- Do all switch arms have break statements? If not, is the "fall through" correct?

4. Input/Output (IO)

- Are files opened before use?
- Are files closed after use?
- Are error conditions checked?

5. Other (OT)

- Any defect discovered that does not fall into one of the above categories.

Further Reading

- Code Reviews:

<http://web.mit.edu/6.005/www/fa16/classes/04-code-review/>

Misc: “The Mess We’re In” - Joe Armstrong

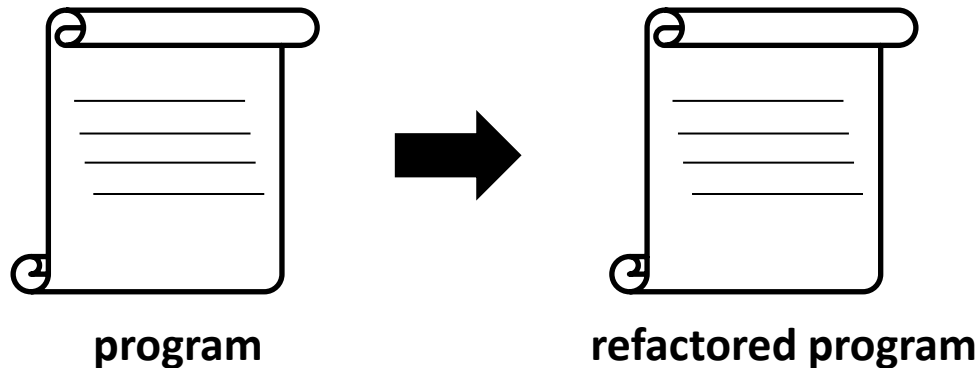
<https://youtu.be/IKXe3HUG2I4>

Pay special attention to the slide on “7 deadly sins” at around 8:00

Software Refactoring

Refactoring

- **Objective:** transform code to make it easier to read, maintain, and improve the design

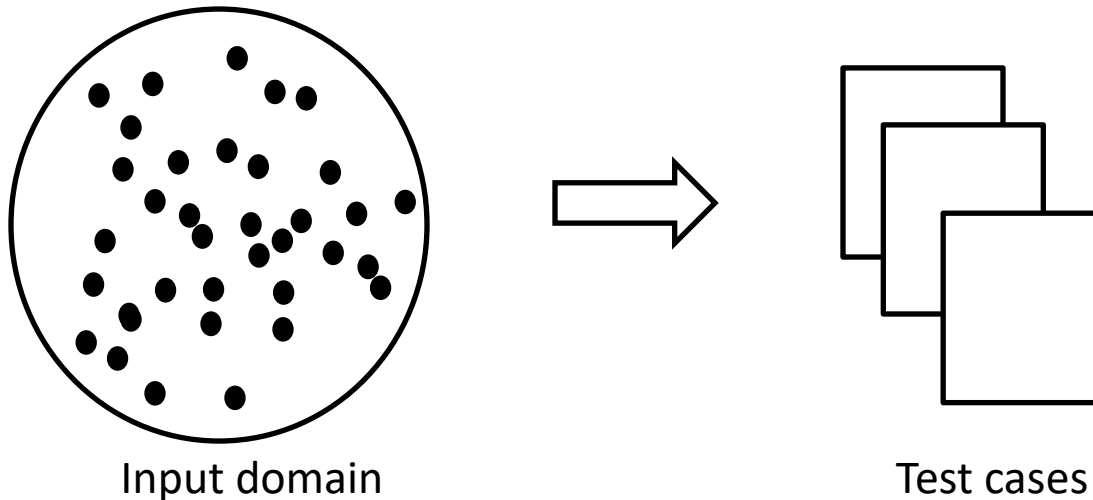


*program behavior doesn't change after refactoring –
“behavior preserving”*

You would have probably done it without actually referring to it by the name

Refactoring

- How can we “guarantee” that the transformed program is behavior preserving?
 - No guarantees. Simply test it.
 - Testing is inherently incomplete.



Why Refactoring?

- To accommodate design changes
 - Requirements Change
- To improve design
 - Add new feature
 - Make code more maintainable etc.
 - To adapt (may not have the best design in the first attempt)
- To improve “cut-paste” code

Refactoring History

- Well suited to OO languages but not limited to those languages only
 - Because of the ability of OO languages to create flexible code/design
 - William F. Opdyke's 1990 PhD thesis on refactoring for Smalltalk
- Increasingly popular (because making changes is less costly) in Agile Environments
- Martin Fowler's Book - "Refactoring – Improving the Design of Existing Code"

Refactoring Types

- Many types listed in Fowler's book
- E.g.
 - Extract Method
 - Collapse Hierarchy
 - Decompose Conditionals
 - Consolidate Conditionals
 - Extract Class
 - Inline Class

Refactoring Type – Collapse Hierarchy

Applied when:

- Class hierarchy (superclass and subclass chain) may grow over time
- Methods and attributes may move from one class to another

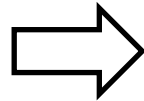
Consequence: superclass and subclass may become too similar

Fix: Merge superclass and subclass into one

Refactoring Type – Consolidate Conditionals

- **Applied when:** a set of conditional expressions with different conditional check and same result

```
double disabilityAmount() {  
    if( seniority < 2)  
        return 0  
    if (monthsDisabled > 12)  
        return 0  
    if (isParttime)  
        return 0;  
    // compute disability amount  
}
```



```
bool notEligibleForDisability() {  
    return (seniority < 2) ||  
           (monthsDisabled > 12) ||  
           isParttime ;  
}  
  
double disabilityAmount() {  
    if( notEligibleForDisability())  
        return 0  
    // compute disability amount  
}
```

Fix: Combine conditionals to have single check and single result (**combine and extract**)

Refactoring Type – Decompose Conditionals

- **Applied when:** a complex conditional check obscures what happens and why it happens

```
if( date.before (SUMMER_START) || date.after(SUMMER_END))  
    charge = quantity * winterRate + winterServiceCharge;  
else  
    charge = quantity * summerRate;
```



```
if(notSummer(date))  
    charge = winterCharge(quantity);  
else  
    charge = summerCharge(quantity);
```

Fix: **Extract methods** from conditionals, modify if-else body

Refactoring Type – Extract Method

- Large method with cohesive code snippet

Demo in Eclipse IDE

- **Fix:** create a method extracting the code snippet

More refactoring types..

- Extract Class
 - When a class is doing the work of two classes, create new class and move relevant methods and attributes.
- Inline Class
 - When a class is not doing much, move its features into another class and delete this class.

Refactoring when not to do?

- Refactoring is powerful but may introduce regression errors
- So, do not do it when:
 - Code is broken
 - Deadlines are close
 - When there is no need
 - When there is no budget (manpower, money) for manual change, test development, and maintenance

When to do refactoring?

- Bad smells – symptoms of unhygienic code
 - Duplicated code
 - Long method
 - Large class
 - Long parameter list
 - Shotgun surgery
 - Feature envy
 - . . .

When to do refactoring?

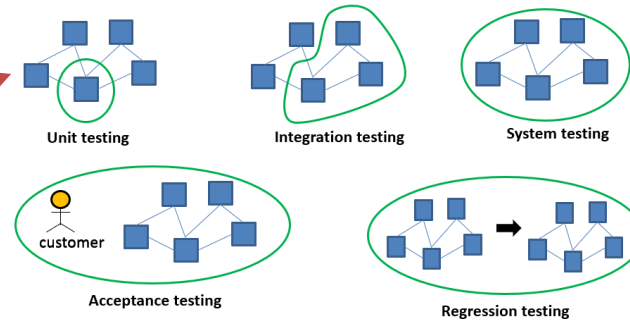
- Bad smells – symptoms of unhygienic code
 - Duplicated code: extract method
 - Long method: extract method, decompose conditionals
 - Large class: extract class (or subclass)
 - Long parameter list: ?
 - Shotgun surgery: move method/field, inline class
 - Feature envy: extract method, move method
 - . . .

Unit Testing

- Recall slides 25-36 from week4 for testing overview
 - Testing is the most common method for verifying software
- Focus: Unit Testing - Testing of individual modules in isolation
- Unit testing is an example of white-box testing. Other examples:
 - Integration testing
 - Static code analysis (for detecting errors)
 - Using code patterns and machine learning

Testing Granularity Levels

- View: software system as a bunch of interacting components



Nikhil Hegde, IIT Dharwad

27

White-box Testing Example

- Note: test without a specification (don't know how fun is called in a bigger picture)

```
1. int fun(int param) {  
2.   int result;  
3.   result = param / 2;  
4.   return result;  
5. }
```

- Execute all statements in the function
- Cons: miss catching an obvious error for a specification: input an integer and return half the value if even. Unchanged otherwise.

Nikhil Hegde, IIT Dharwad

36

Detour: IEEE Terminology

- **Failure:** observable incorrect behavior of the system
- **Fault / Bug:** related to code. Presence of a fault doesn't mean failure - necessary but not sufficient condition for failure.
- **Error:** Cause of a fault (usually a human error)

JUnit (<http://junit.org>)

- Open-source framework to write and run tests for Java programs
- You can write Unit tests
 - E.g. test individual methods of a class
- Erich Gamma and Kent Beck wrote it initially

JUnit – how to use?

- Provides annotations
 - E.g. annotations:
 - `@Test` – identifies a test method
 - `@Before` – identifies a method that is executed **before** a test is run
 - `@After` – identifies a method that is executed **after** a test is run
- Provides assertions for *verifying* methods
 - E.g. `assertEquals(3, MyClass.GetMinWordLen());`
- Provides Test runners for running the tests
- Provides features for automated running of tests and progress indicators

behaving as expected / did we get it right?

JUnit Demo

Demo in Eclipse IDE