Modifying Code

Brittany Johnson
SWE 437

Adapted from slides by Paul Ammann & Jeff Offutt
Programming for maintainability

1. Understanding the program
2. Programming for change
3. Coding style
Programming for maintainability

1. Understanding the program
2. Programming for change
3. Coding style
Major maintenance activities

We must understand an existing system before changing it
- How to accommodate the change?
- What are the potential ripple effects?
- What skills and knowledge are required?
Major maintenance activities

We must understand an existing system before changing it
- How to accommodate the change?
- What are the potential ripple effects?
- What skills and knowledge are required?

1. **Identify** the change
   - What to change, why to change
2. **Manage** the process...what resources are needed?
3. **Understand** the program
   - How to make the change, determine ripple effect
4. **Make** the change
5. **Test** the change
6. **Document** and record the change
Comprehension process

- Read documentation
- Run the program
- Read the source code

Dynamic analysis
Static analysis
What influences understanding?

**Expertise:** Domain knowledge, programming skills

**Program structure:** Modularity, level of nesting

**Documentation:** Readability, accuracy, up-to-date

**Coding conventions:** Naming style, small design patterns

**Comments:** Accuracy, clarity, and usefulness

**Program presentation:** Good use of indentation and spacing
Programming for maintainability

1. Understanding the program
2. Programming for change
3. Coding style
Avoid unnecessary fancy tricks

Write for humans, not compilers
- fully parenthesize expressions
- pointer arithmetic is anti-engineering
- clever programming techniques are not beneficial

In 1980, computers were slow and memory expensive
- Control flow dominated the running time
- Hence the undergraduate CS emphasis on analysis of algorithms

Today: Make it easier to change the program
- Readable code is easier to debug, more reliable, and more secure
- Optimizing compilers are far better than humans
- Overall architecture usually dominates running time
Document clearly

Include **header blocks** for each method (author & version)

Add a **comment** every time you stop to **think**
- Why a method does something is more important than **what**
- **What** is more important than **how**

Document:
- **assumptions**
- **variables** that can be overridden by child methods
- reliance on default and superclass **constructors**

Write **pseudocode** as comments, then write the method
- **faster** and more **reliable**

Use a **version control** system with an edit history
- **Explain why** each change was made clearly
Use white space effectively

A 1960s study asked "how far should we indent"
- 2—4 characters is ideal
- Fewer is hard to see
- More makes program too wide

Never use tabs – they look different in every editor and printer
- Mixing tabs and spaces is even worse

Use plenty of spaces
- newList(x+y)=fName+space+IName+space+title;
- newList (x+y) = fName + space + IName + space + title;

Don’t put more than one statement per line
Write maintainable Java

Be tidy
- sloppy style looks like sloppy thinking
- sloppy style creates maintenance debt

Use clear names
- Long names are simpler than short names
- Don’t make it so long it’s hard to read

Don’t test for error conditions you can’t handle
- Let them propagate to someone who does

These habits are important, if not critical, to developer jobs.
Java specific tips

Implement **both or neither equals()** and **hashCode()**

- Implementing just one can cause subtle faults

Always **override toString()** to produce **human-readable** description of the object

If **equals()** is called on the wrong type, **return false**, not an exception

If your class is **cloneable**, use **super.clone()**, **not new()**

- **new()** will break if another programmer inherits from your class

**Threads** are hard to get right and harder to modify

Don’t add **error checking** the VM already does

- array bounds, null pointers, etc.
Keep it simple, stupid

Long methods are not simple
- Good programmers write less code, not more
Bad designs lead to more and longer methods
Don’t generalize unless it’s necessary
Ten programmers...
- deliver twice as much code
- four times as many faults, and
- half the functionality as
...five programmers
Classes and objects

The point of OO design is to look at nouns (data) first, then verbs (algorithms and methods)

Think about what it is, not what it does
  - class names should not be verbs

Objects are defined by state – the class defines behavior

Lots of switch statements may mean the class is trying to do too many things
  - Use inheritance or type parameterization

Make methods that don’t use class instance variables static

Don’t confuse inheritance with aggregation
  - inheritance implements “is-a”
  - aggregation implements “has-a”
Programming for change summary

The cost of writing a program is a small fraction of the cost of fixing and maintaining it

... Don't be lazy or selfish ...

Be an engineer!

Remember that complexity is the number one enemy of maintainability
Programming for maintainability

1. Understanding the program
2. Programming for change
3. Coding style
Using style conventions

Select a set of **style** conventions
- follow them *strictly*

Follow the **existing style** when making changes
- even if you don’t like it

**Lots** of style conventions are available
- it’s more important to be *consistent* than to have perfect style

Programmers need to be told to follow teams’ style

There are two types of people.

Programmers will know.
What style guides tell us

Case for names
- Variables, methods, classes, ...
Guidelines for choosing names
Width, special characters, and splitting lines
Location of statements
Organization of methods and use of types
Use of variables
Control structures
Proper spacing and white space
Comments
Summary

Programming habits have a major impact on **readability**

**Readability** has a major impact on **maintainability**

Maintainability determines **long-term costs**

The minor decisions that engineers make determine how much money the company makes

This is what engineering means!