Coding Practices

Lecture 3a

Agenda

- What is Software Construction?
- What Does Clean Code Mean?
- Practices of Writing Clean Code

What Is Software Construction?

What Is Software Construction?

- You know what “construction” means when it's used outside software development:
  - “Construction” is the work “construction workers” do when they build a house, a school, or a skyscraper.
  - In common usage, “construction” refers to the process of building.
  - “Construction” refers to the hands-on part of creating something.

What Is Software Construction

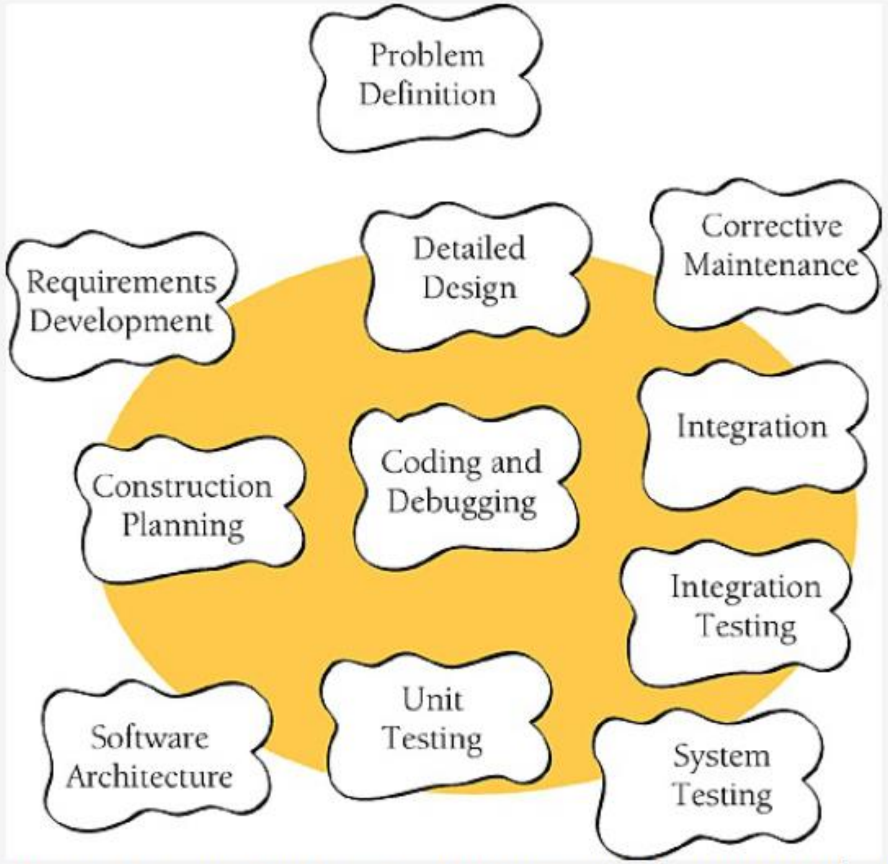


Figure 1-1. Construction activities are shown inside the gray circle. Construction focuses on coding and debugging but also includes detailed design, unit testing, integration testing, and other activities

What Is Software Construction

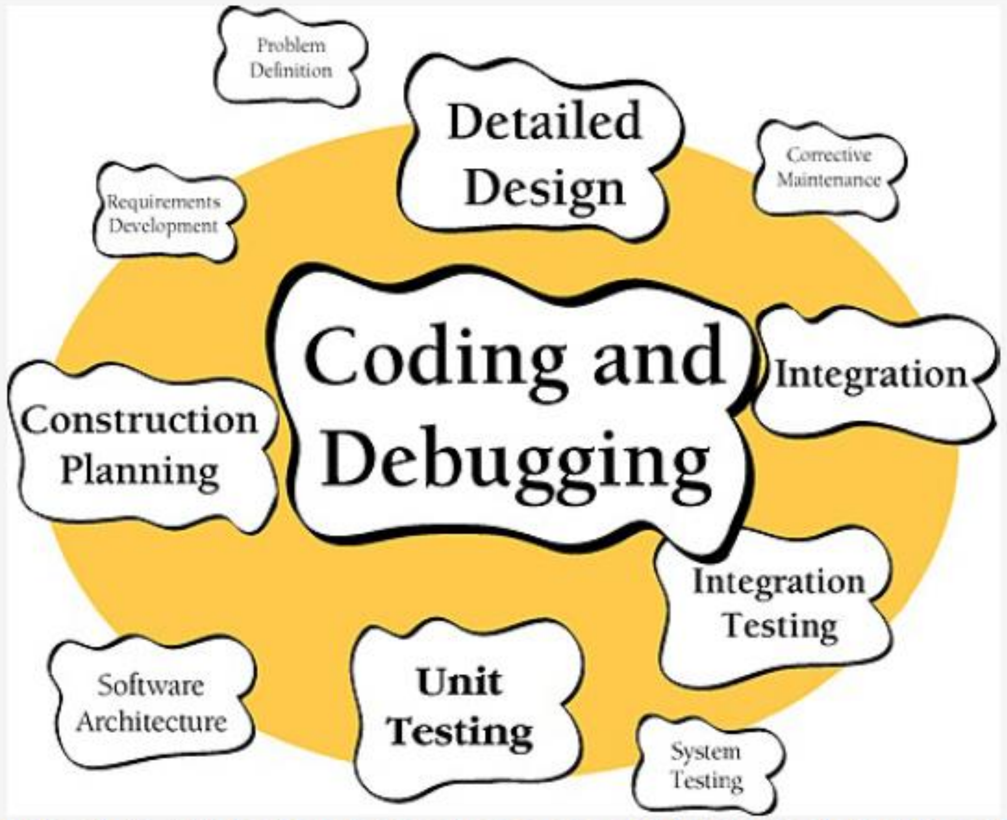
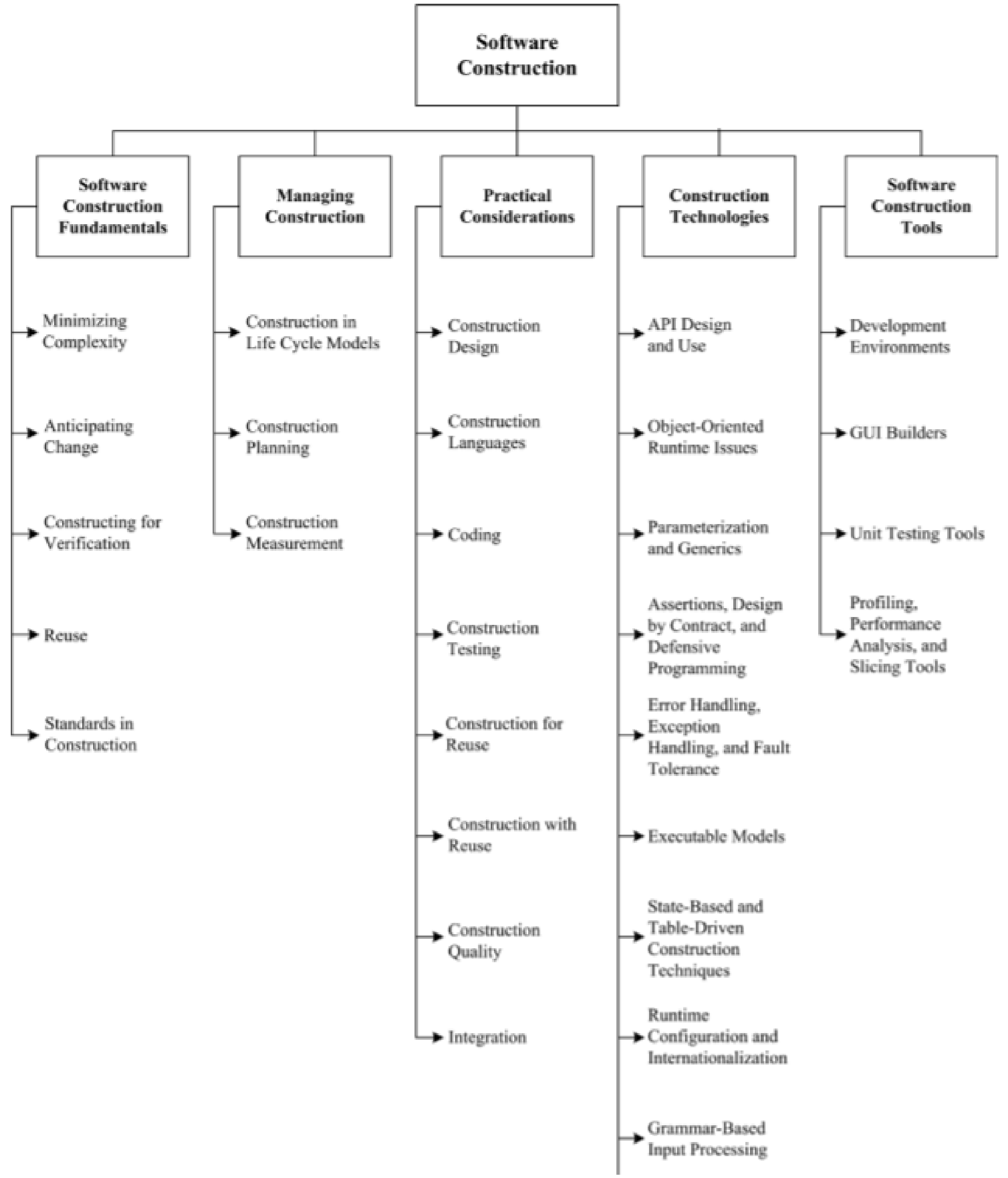


Figure 1-2. This book focuses on coding and debugging, detailed design, construction planning, unit testing, integration, integration testing, and other activities in roughly these proportions

What is Software Construction?

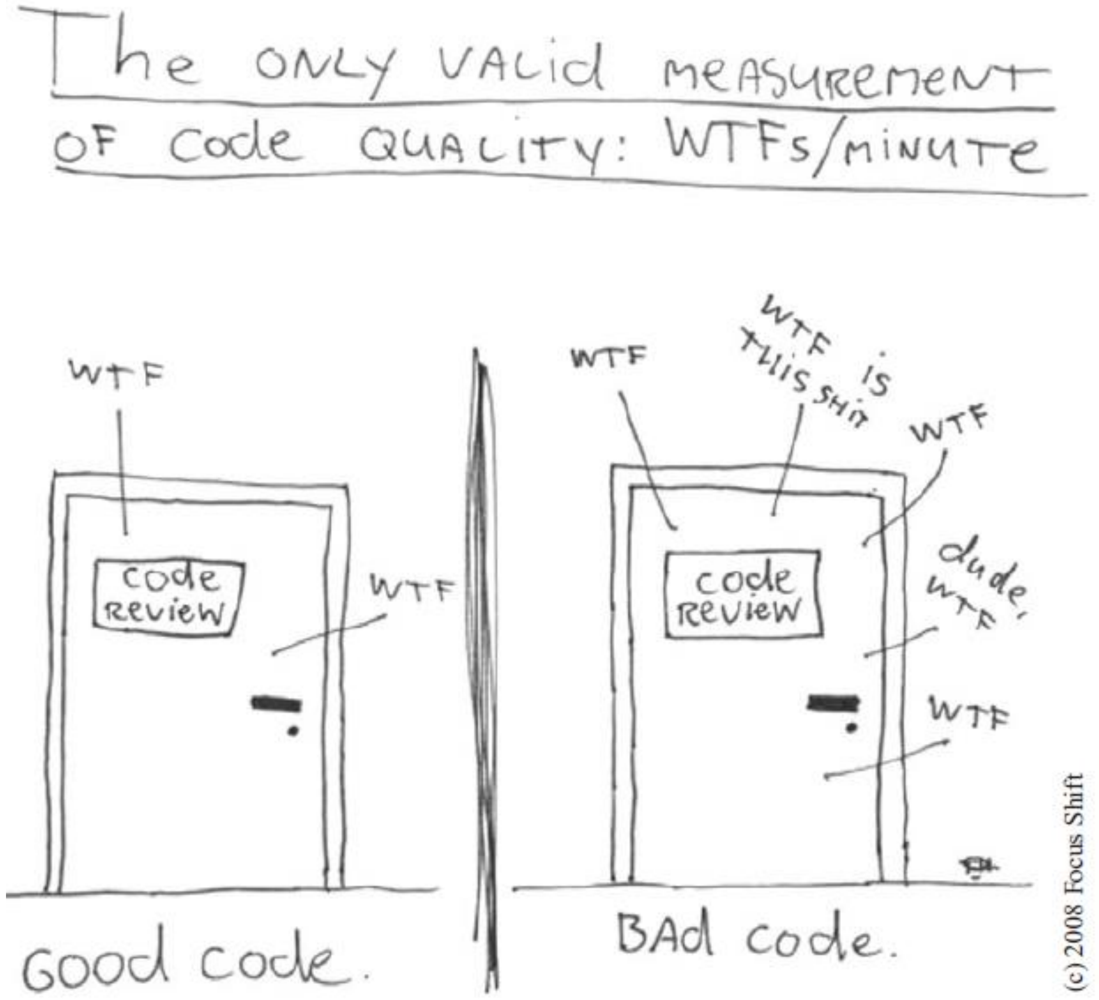
- The term **software construction** refers to the detailed **creation** of working **software** through a **combination of coding, verification, unit testing, integration testing, and debugging**.
  - According to the SWEBOK version 3 (The IEEE Guide to the Software Engineering Body of Knowledge).
- Code is the ultimate deliverable of a software project, and thus the **software quality is closely linked to the software construction**.
- The **quality** of the **construction** substantially **affects** the quality of the **software**.


Software Construction
Software Construction Fundamentals
Minimizing Complexity
Anticipating Change
Constructing for Verification
Reuse
Standards in Construction
Managing Construction
Construction in Life Cycle Models
Construction Planning
Construction Measurement
Practical Considerations
Construction Design
Construction Languages
Coding
Construction Testing
Construction for Reuse
Construction with Reuse
Construction Quality
Integration
Construction Technologies
API Design and Use
Object-Oriented Runtime Issues
Parameterization and Generics
Assertions, Design by Contract, and Defensive Programming
Error Handling, Exception Handling, and Fault Tolerance
Executable Models
State-Based and Table-Driven Construction Techniques
Runtime Configuration and Internationalization
Grammar-Based Input Processing
Software Construction Tools
Development Environments
GUI Builders
Unit Testing Tools
Profiling, Performance Analysis, and Slicing Tools

Part of the Breakdown of Topics for the Software Construction (SWEBOK version 3)

What Does Clean Code Mean?

Code Quality



Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

Extracts from Several Definitions of Clean Code

- "Clean code can be read and **enhanced by a developer other than its original author**. It has unit and acceptance tests. It has meaningful names. It provides a clear and minimal API".
  - Dave Thomas, founder of OTI, godfather of the Eclipse strategy
- "Clean code is **simple and direct**. Clean code reads like well-written prose".
  - Grady Booch, author of Object-Oriented Analysis and Design with Applications
- "Clean code always looks like it was **written by someone who cares**. There is **nothing obvious that you can do to make it better**".
  - Michael Feathers, author of Working Effectively with Legacy Code

Extracts from Several Definitions of Clean Code

- "You know you are working on clean code when **each routine** you read turns out to be **pretty much what you expected**".
  - Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader. The godfather of all those who care about code.
- "The logic should be straightforward to make it **hard for bugs to hide**, the dependencies minimal to ease maintenance, **error handling complete** according to an **articulated strategy**, and **performance close to optimal**".
  - Bjarne Stroustrup, inventor of C++ and author of The C++ Programming Language

Practices of Writing Clean Code

Categories of Practices of Writing Clean Code

1. **Meaningful Names**
   - Names are everywhere in software. We name our variables, our functions, our arguments, classes, and packages. We name our source files and the directories that contain them. We name our jar files, war files and ear files.
2. **Functions**
   - Functions are the first line of organization in any program. Writing them well is a significant topic.
3. **Comments**
   - If our programming languages were expressive enough, or if we had the talent to subtly wield those languages to express our intent, we would not need comments very much—perhaps not at all. The proper use of comments is to compensate for our failure to express ourselves in code.

Categories of Practices of Writing Clean Code

4. **Formatting**
   - You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. The coding style and readability set precedents that continue to affect maintainability and extensibility.
5. **Objects and Data Structures**
   - There is a reason that we keep our variables private. We don't want anyone else to depend on them. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?
6. **Error Handling**
   - Many code bases are completely dominated by error handling. It is nearly impossible to see what the code does because of all the scattered error handling. Error handling is important, but if it obscures logic, it's wrong.

Categories of Practices of Writing Clean Code

7. **Boundaries**
   - We seldom control all the software in our systems. Sometimes we buy third-party packages or use open source. Other times we depend on teams in our own company to produce components or subsystems for us. Somehow, we must cleanly integrate this foreign code with our own.
8. **Unit Tests**
   - Many programmers have missed some of the more subtle, and important, points of writing good tests.
9. **Classes**
   - In addition to proper composition of functions and how they interrelate, we do not have clean code until we have paid attention to higher levels of code organization such as classes.

Meaningful Names (1/9)

- **Use Intention-Revealing Names**

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

- **Avoid Disinformation**
  - Beware of using names which vary in small ways

```
XYZControllerForEfficientHandlingOfStrings

XYZControllerForEfficientStorageOfStrings
```

  - Using inconsistent spellings is *disinformation*

Meaningful Names (1/9)

- **Use Pronounceable Names**

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

to

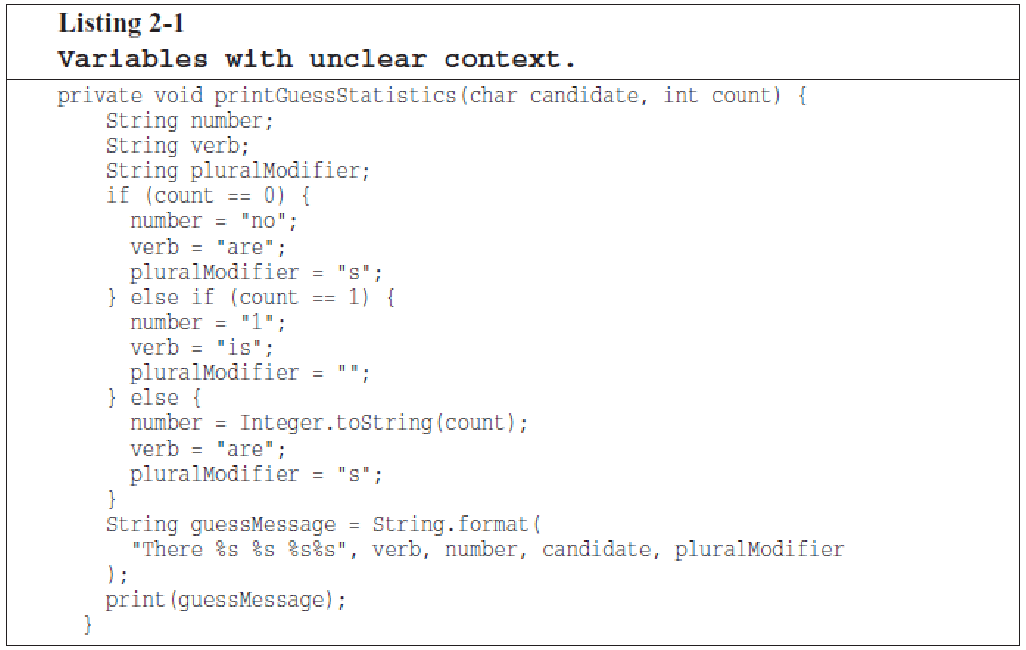
```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

- **Use Searchable Names**
  - Single-letter names and numeric constants are problematic in that they are not easy to locate across a body of text.

Meaningful Names (1/9)

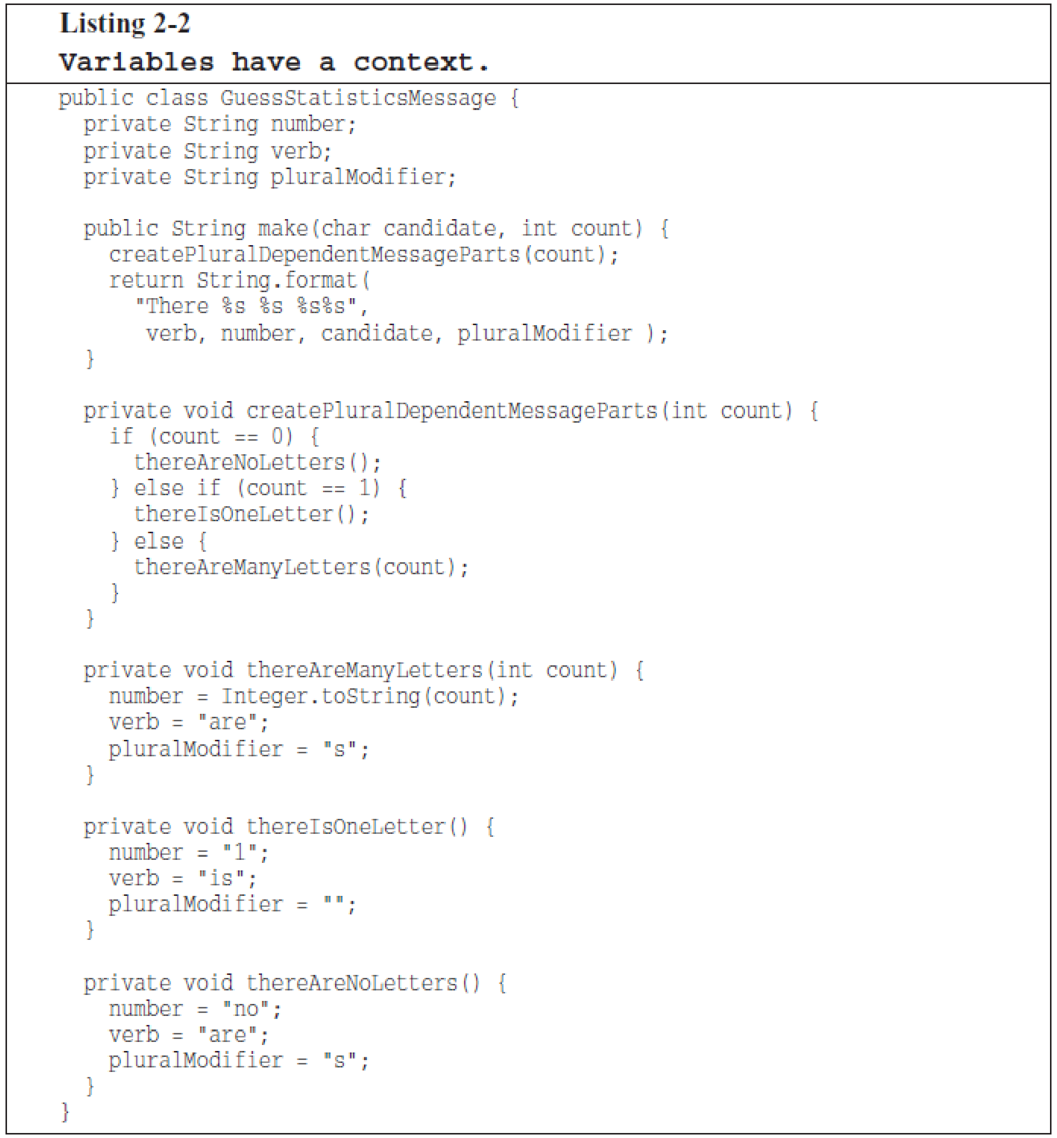
- **Avoid Encodings**
  - In days of old, Fortran forced encodings by making the first letter a code for the type
  - We don’t need to prefix member variables such as m_
  - Name interfaces without prefixes such as *I*. For example, instead calling an interface *IShapeFactory* call it *ShapeFactory* and its implementation *ShapeFactoryImp*.
- **Pick One Word per Concept**
  - Pick one word for one abstract concept and stick with it. For instance, it’s confusing to have fetch, retrieve, and get as equivalent methods of different classes.

Meaningful Names (1/9)



Listing 2-1
Variables with unclear context.

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
    } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
    } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
    }
    String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
  }
```



Listing 2-2
Variables have a context.

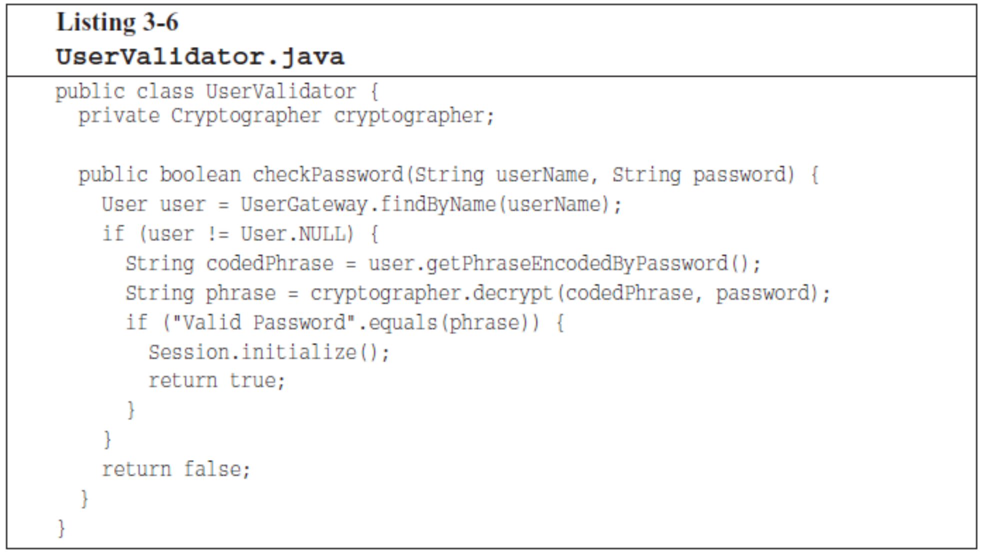
```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

Functions (2/9)

- **Functions should always be small**.
- **Blocks and Indenting**.
- **Do One Thing**.
  - Sections (such as *declarations*, *initializations*, and *sieve*) within functions.
- **Function Arguments**
  - The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification.
  - Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly

Functions (2/9)

- **Have No Side Effects**
  - Your function promises to do one thing, but it also does other hidden things.
  - Sometimes it will make unexpected changes to the variables of its own class. Sometimes it will make them to the parameters passed into the function or to system global variables. Spot the side effect:



Listing 3-6
UserValidator.java

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

Functions (2/9)

- **Command Query Separation**
  - Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object.
- **Prefer Exceptions to Returning Error Codes**
  - Extract Try/Catch Blocks.
  - Functions should do one thing.
  - Error handing is one thing.
  - The delete function is all about error processing. It is easy to understand and then ignore. The deletePageAndAllReferences function is all about the processes of fully deleting a page. Error handling can be ignored. This provides a nice separation that makes the code easier to understand and modify.

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}
```

Functions (2/9)

- **Don’t Repeat Yourself**
  - Code repetition is a problem because it will require multiple modifications if the algorithm ever requires a change. It is also an opportunity to replicate an error of omission.
  - Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it such as the Open Closed Principle (OCP) in SOLID, or the DRY principle.

Comments (3/9)

- **Comments Do Not Make Up for Bad Code**.
  - One of the more common motivations for writing comments is bad code.
- **Good Comments**
  - Legal Comments
  - Informative Comments
  - Explanation of Intent
  - Clarification
  - Warning of Consequences
  - TODO Comments

Comments (3/9)

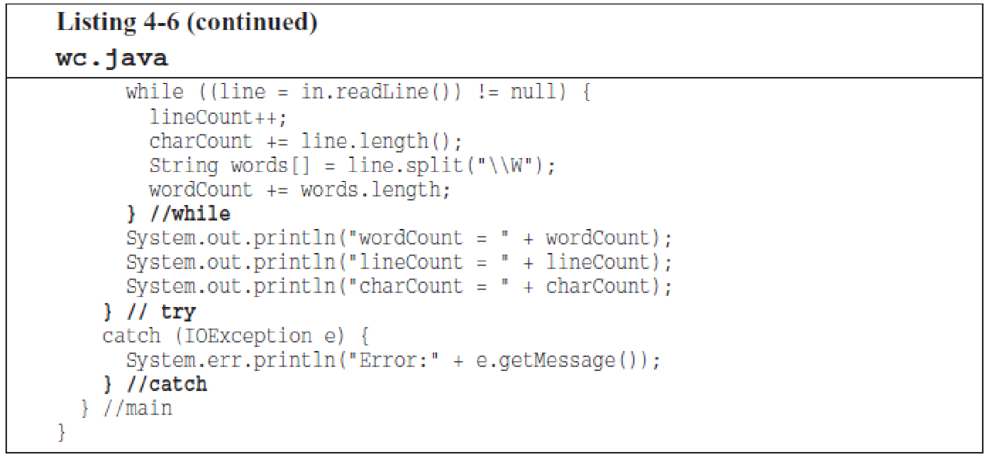
- **Bad Comments**
  - Noise Comments

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}
```

  - Position Markers

```
// Actions //////////////////////////////////
```

  - Closing Brace Comments


Listing 4-6 (continued)
wc.java

```
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;
      } //while
      System.out.println("wordCount = " + wordCount);
      System.out.println("lineCount = " + lineCount);
      System.out.println("charCount = " + charCount);
    } // try
    catch (IOException e) {
      System.err.println("Error:" + e.getMessage());
    } //catch
  } //main
}
```

Formatting (4/9)

- **Vertical Formatting**
  - **The Newspaper Metaphor**: the topmost parts of the source file should provide the high-level concepts and algorithms. Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file.
  - **Vertical Openness Between Concepts**: each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines.
  - **Vertical Density**: if openness separates concepts, then vertical density implies close association. So lines of code that are tightly related should appear vertically dense.

Formatting (4/9)

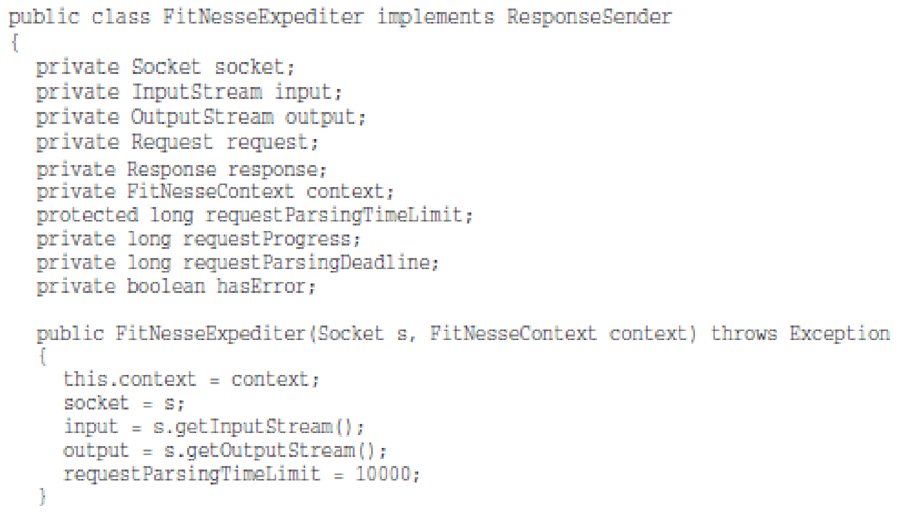
- **Vertical Formatting (continuation)**
  - **Vertical Distance**: for those concepts that are so closely related that they belong in the same source file, their vertical separation should be a measure of how important each is to the understandability of the other.
  - **Vertical Ordering**: we want function call dependencies to point in the downward direction. That is, a function that is called should be below a function that does the calling (the opposite of languages like C).

Formatting (4/9)

- **Horizontal Formatting**
  - How wide should a line be? Programmers clearly prefer short lines. The old Hollerith limit is 80. Lines edging out to 100 or even 120 may be acceptable. But beyond that is probably just careless.
  - **Horizontal Openness and Density**: we use horizontal white space to associate things that are strongly related and disassociate things that are have a weaker relation.

Formatting (4/9)

- **Horizontal Formatting (continuation)**
  - **Horizontal Alignment:** unaligned declarations and assignments are preferred, as shown below, because they point out an important deficiency. If there are long lists that need to be aligned, *the problem is the length of the lists*, not the lack of alignment.

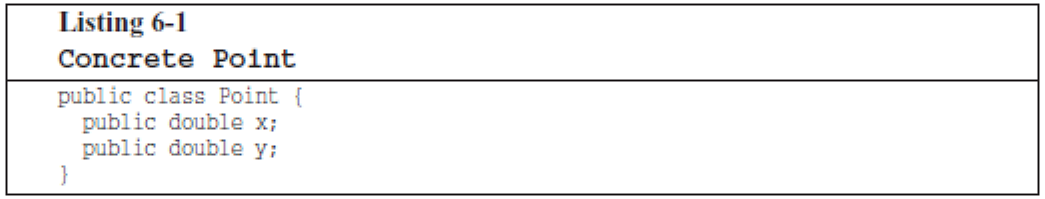


```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;
  private Response response;
  private FitNesseContext context;
  protected long requestParsingTimeLimit;
  private long requestProgress;
  private long requestParsingDeadline;
  private boolean hasError;

  public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
  {
    this.context = context;
    socket = s;
    input = s.getInputStream();
    output = s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

  - **Indentation**: A source file is a hierarchy rather like an outline.

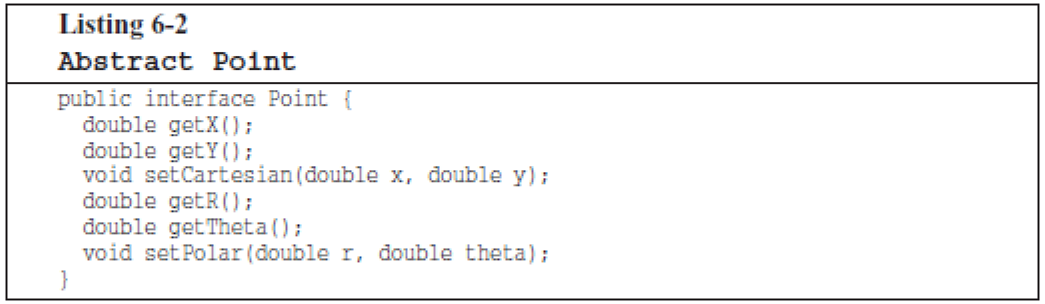
Objects and Data Structures (5/9)

- **Data Abstraction**
  - A class does not simply push its variables out through getters and setters. Rather it **exposes** abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation.



Listing 6-1
Concrete Point

```
public class Point {
  public double x;
  public double y;
}
```



Listing 6-2
Abstract Point

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Objects and Data Structures (5/9)

- **The Law of Demeter**:
  - It says that a method *f* of a class *C* should only call the methods of these:
    - *C*
    - An object created by *f*
    - An object passed as an argument to *f*
    - An object held in an instance variable of *C*
  - If *ctxt*, *Options*, and *ScratchDir* are objects, then their internal structure should be hidden rather than exposed, and so knowledge of their innards is a clear violation of the Law.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

  - Additionally, this kind of code is often called a *train wreck* because it looks like a bunch of coupled train cars. It is usually best to split them up as follows:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Error Handling (6/9)

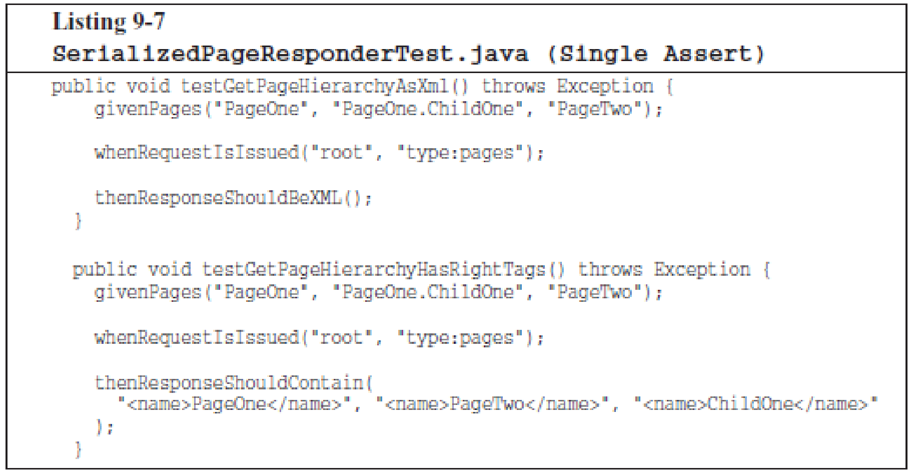
- **Use Exceptions Rather Than Return Codes**
- **Write Your *Try-Catch-Finally* Statement First**
- **Provide Context with Exceptions**
  - Each exception that you throw should provide enough context to determine the source and location of an error.
- **Define Exception Classes in Terms of a Caller's Needs**
  - Separate your business logic from your error handling.
- **Don't Return Null**
  - Returning null from methods is bad.
- **Don't Pass Null**
  - Passing null into methods is terrible.

Boundaries (7/9)

- **Exploring and Learning Boundaries**
  - It’s not our job to test the third-party code, but it may be in our best interest to write tests for the third-party code we use.
- **Learning Tests Are Better Than Free**
  - If we require an API, we need to learn how to use it. Writing tests is an easy and isolated way to get that knowledge. The learning tests are precise experiments that help increasing our understanding.
- **Clean Boundaries**
  - Prepare for change in the code that is out of our control. Good software designs accommodate change without huge investments and rework.
  - We manage third-party boundaries by having very few places in the code that refer to them.
  - We may use an *Adapter* to convert from our perfect interface to the provided interface.

Unit Tests (8/9)

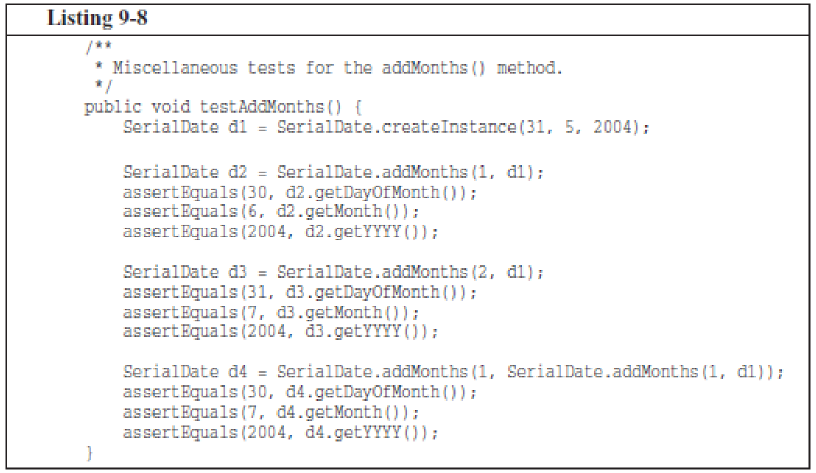
- **One Assert per Test**



Listing 9-7
SerializedPageResponderTest.java (Single Assert)

```java
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldBeXML();
  }

  public void testGetPageHierarchyHasRightTags() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldContain(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
  }
```

- **Single Concept per Test**



Listing 9-8

```java
/**
 * Miscellaneous tests for the addMonths() method.
 */
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());

    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

Classes (9/9)

- **Class Organization**
  - **Follow the standard coding convention**: a class should begin with a list of variables. Public static constants, if any, should come first. Then private static variables, followed by private instance variables
  - **Encapsulation**: keep variables and utility functions private, but without fanaticism. Sometimes we need to make a variable or utility function protected so that it can be accessed by a test.
- **Classes Should Be Small**
  - **The Single Responsibility Principle**: it states that a class or module should have one, and only one, *reason to change*. This principle gives us both a definition of responsibility, and a guidelines for class size. Classes should have one responsibility—one reason to change.
  - **Cohesion**: high cohesion is always desirable. When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole.
  - **Maintaining Cohesion Results in Many Small Classes**
- **Organizing for Change**
  - For most systems, change is continual. Every change subjects us to the risk that the remainder of the system no longer works as intended.
  - **Isolating from Change**: Needs will change; therefore code will change. We can introduce interfaces and abstract classes to help isolate the impact of those details.

Next On

Take Away Points

- Software construction
  - What is it?
  - Why is it important?
  - How is software construction connected with software quality?
- What do we mean by clean code?
- Practices of writing clean code
  - What do we mean by clean code?
  - Which are the key practices for achieving a clean code.

Further Reading

- Robert C. Martin, “Clean Code”
  - Chapters 1 to 10
- IEEE, “Guide to the Software Engineering Body of Knowledge (SWEBOK) Version 3”
  - Chapter 3: Software Construction
- Steve McConnell, “Code Complete 2nd Edition”
  - Chapter 1: Welcome to Software Construction
- Common Weakness Enumeration, “Bad Coding Practices”
  - https://cwe.mitre.org/data/definitions/1006.html

Next Lecture

- Coding Standards