Introduction to Software Engineering
Agile Clean Code and Simple Design Practices

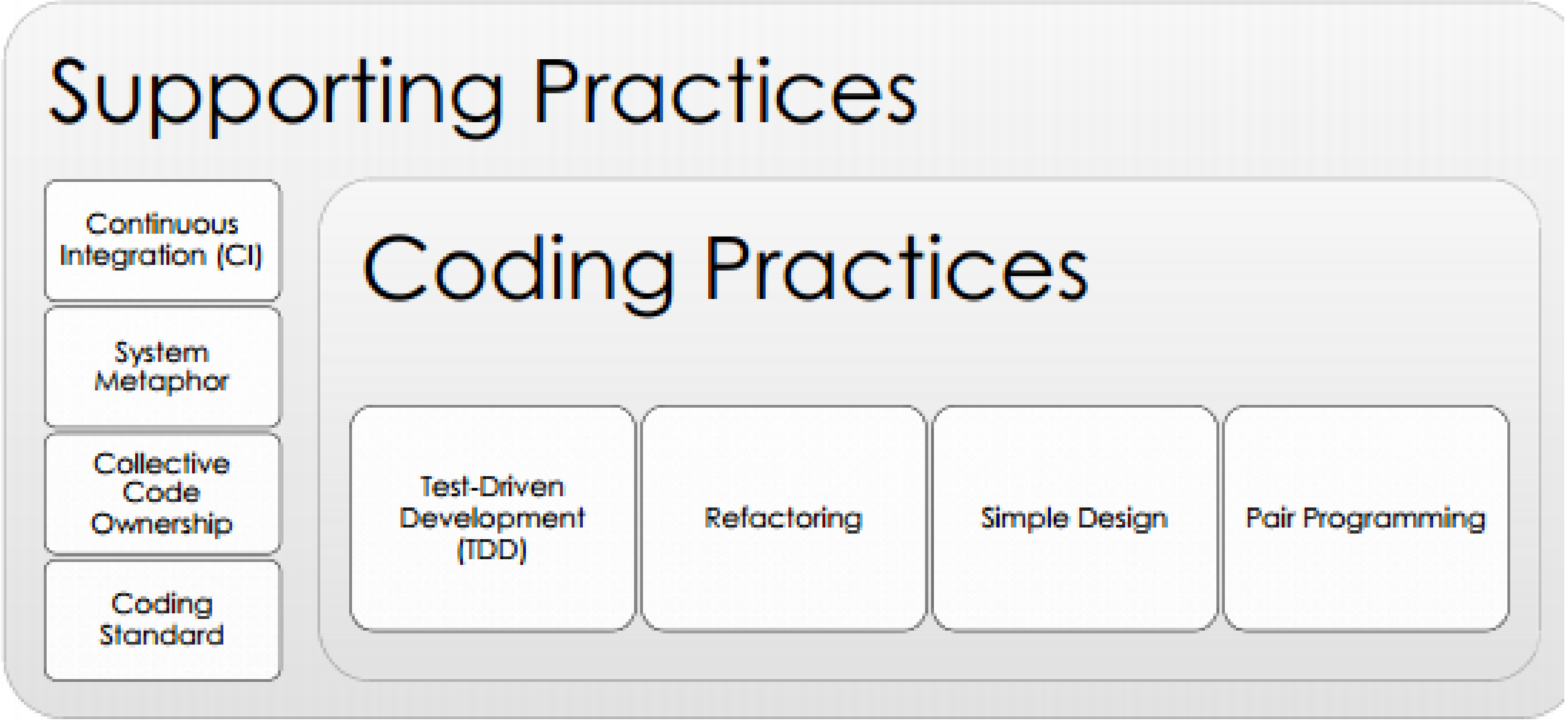
CSE115a – Spring 2025
Richard Jullig

Images: sxc.hu

Acknowledgments

- Materials drawn from
  - Construx sources
  - Martin’s *Clean Code*

Technical Practices



Supporting Practices

Continuous Integration (CI)

System Metaphor

Collective Code Ownership

Coding Standard

Coding Practices

Test-Driven Development (TDD)

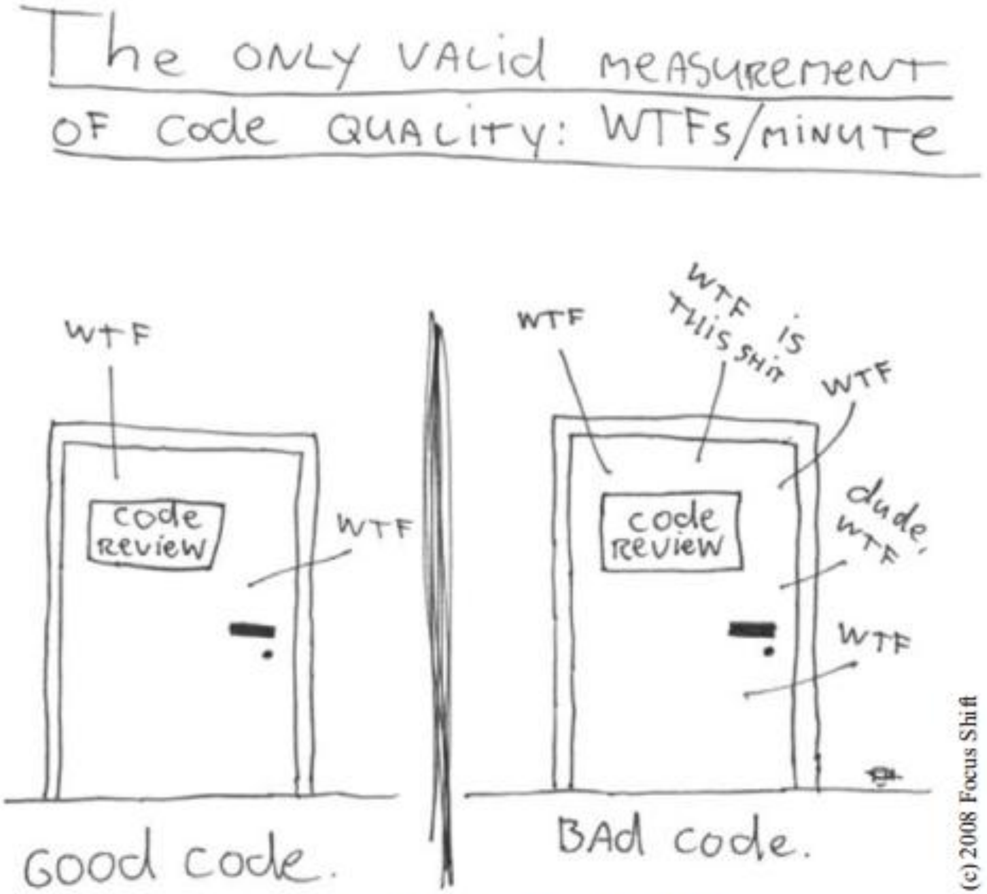
Refactoring

Simple Design

Pair Programming

- Coding Practice: Clean Code

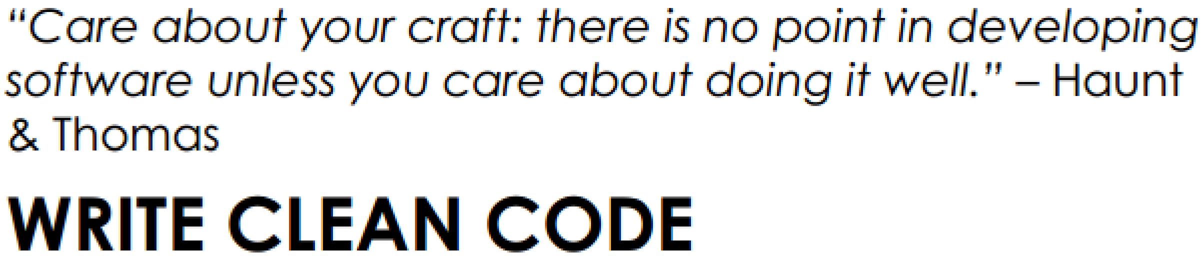
How to tell good code from bad code



(c) 2008 Focus Shift

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

Clean Code



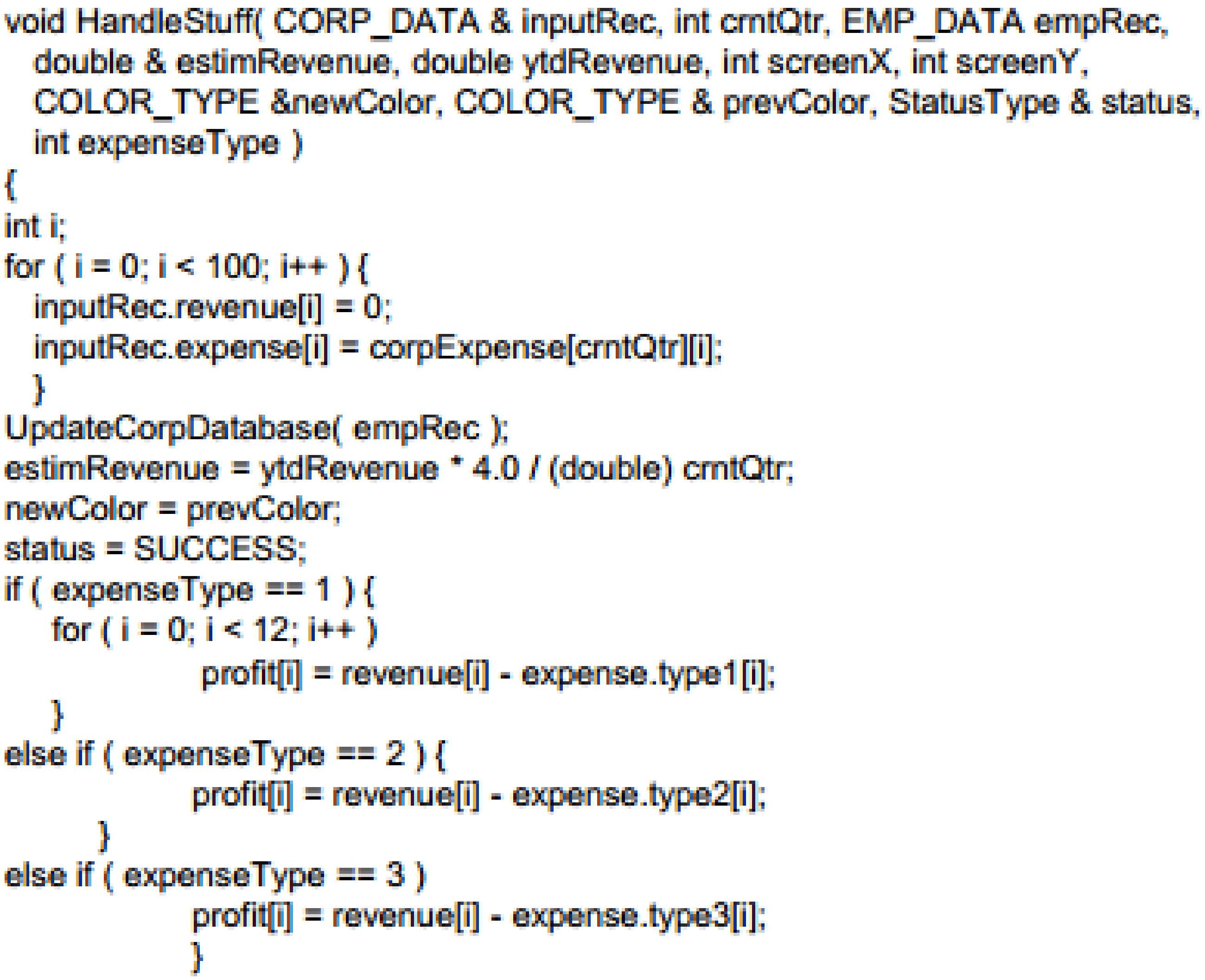
"Care about your craft: there is no point in developing software unless you care about doing it well." – Haunt & Thomas

WRITE CLEAN CODE

Useful Resources

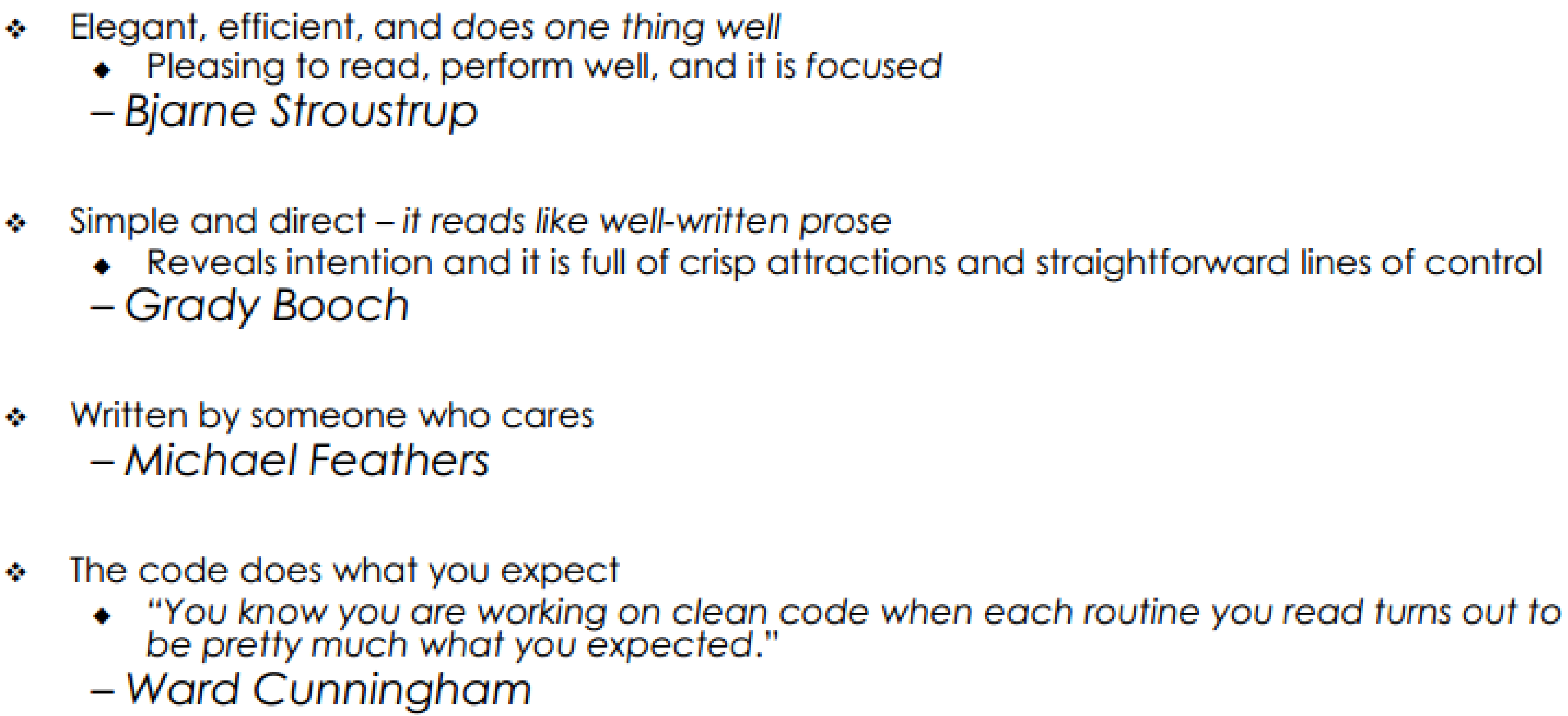
- Robert Martin, *Clean Code* (2009)
  - On Canvas > Pages > Reading Material – Software Engineering

Low Quality (not Clean) Code



```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
   double & estimRevenue, double ytdRevenue, int screenX, int screenY,
   COLOR_TYPE &newColor, COLOR_TYPE & prevColor, StatusType & status,
   int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[crntQtr][i];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
                    profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 ) {
                   profit[i] = revenue[i] - expense.type2[i];
          }
else if ( expenseType == 3 )
                   profit[i] = revenue[i] - expense.type3[i];
                   }
```

What is Clean Code?



- Elegant, efficient, and *does one thing well*
  - Pleasing to read, perform well, and it is *focused*

  – Bjarne Stroustrup

- Simple and direct – *it reads like well-written prose*
  - Reveals intention and it is full of crisp attractions and straightforward lines of control

  – Grady Booch

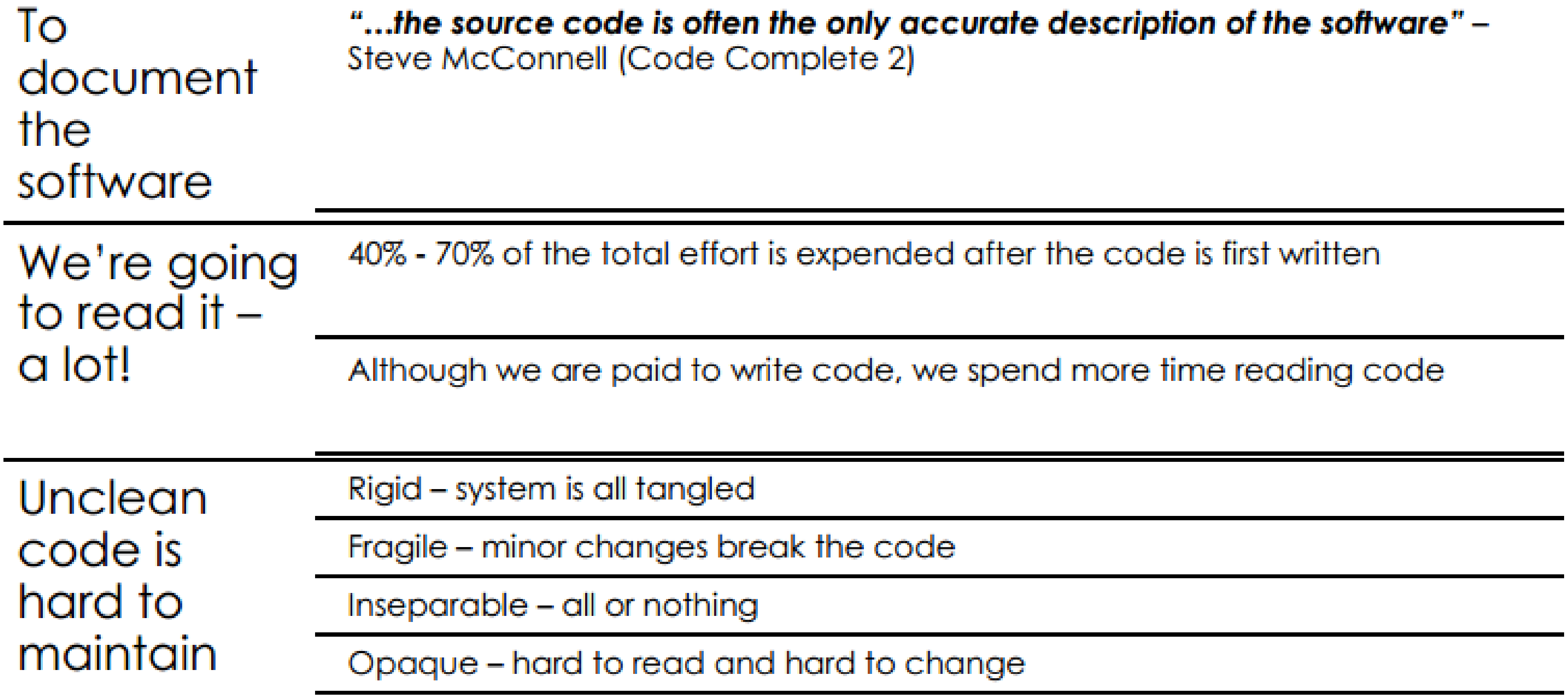
- Written by someone who cares

  – Michael Feathers

- The code does what you expect
  - *"You know you are working on clean code when each routine you read turns out to be pretty much what you expected."*

  – Ward Cunningham

Why Clean Code?



To document the software	***“…the source code is often the only accurate description of the software”*** – Steve McConnell (Code Complete 2)
We're going to read it – a lot!	40% - 70% of the total effort is expended after the code is first written
	Although we are paid to write code, we spend more time reading code
Unclean code is hard to maintain	Rigid – system is all tangled
	Fragile – minor changes break the code
	Inseparable – all or nothing
	Opaque – hard to read and hard to change

- Even while coding, time spent reading code vs. writing code is about 10:1!

Getting to Clean Code

- Adhere to standard conventions
- Avoid magic literals
- Simplify structures
- Consider cohesion & coupling characteristics
- Communicate intentions
- Comment effectively
- Solicit review

Team Working Agreements

- Definition of Done
- Acceptance Criteria

- Coding Standard/Style guide
  - Google Python Style Guide
  - Google Java Style Guide
  - Google Javascript Guide

Coding Standard

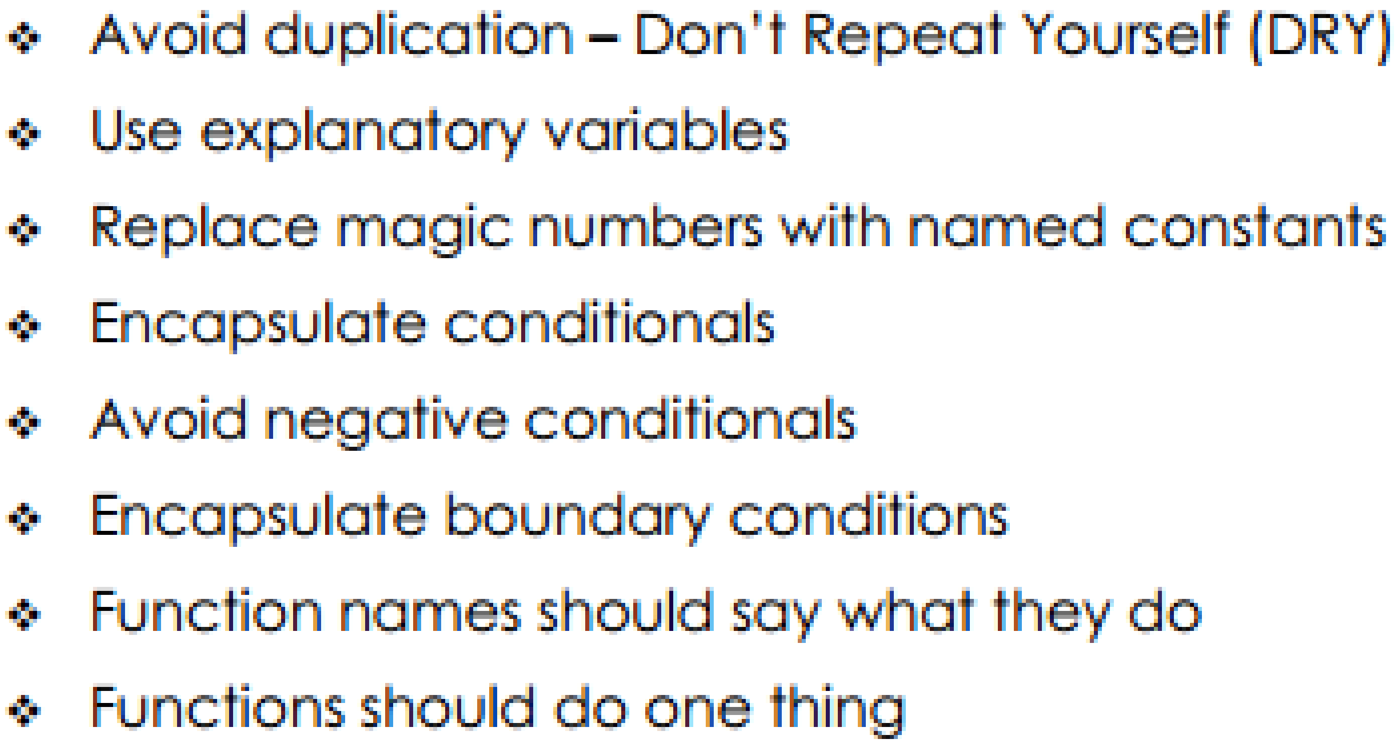
- Standardize to avoid waste
- Start with an accepted standard
- Timebox the debate
- Fit on one page
- Revisit regularly until no one cares
- The code becomes the standard

"***A coding standard*** *is foundational for working as a team that* ***collectively owns the code.***"

Criteria for Good Coding Standard

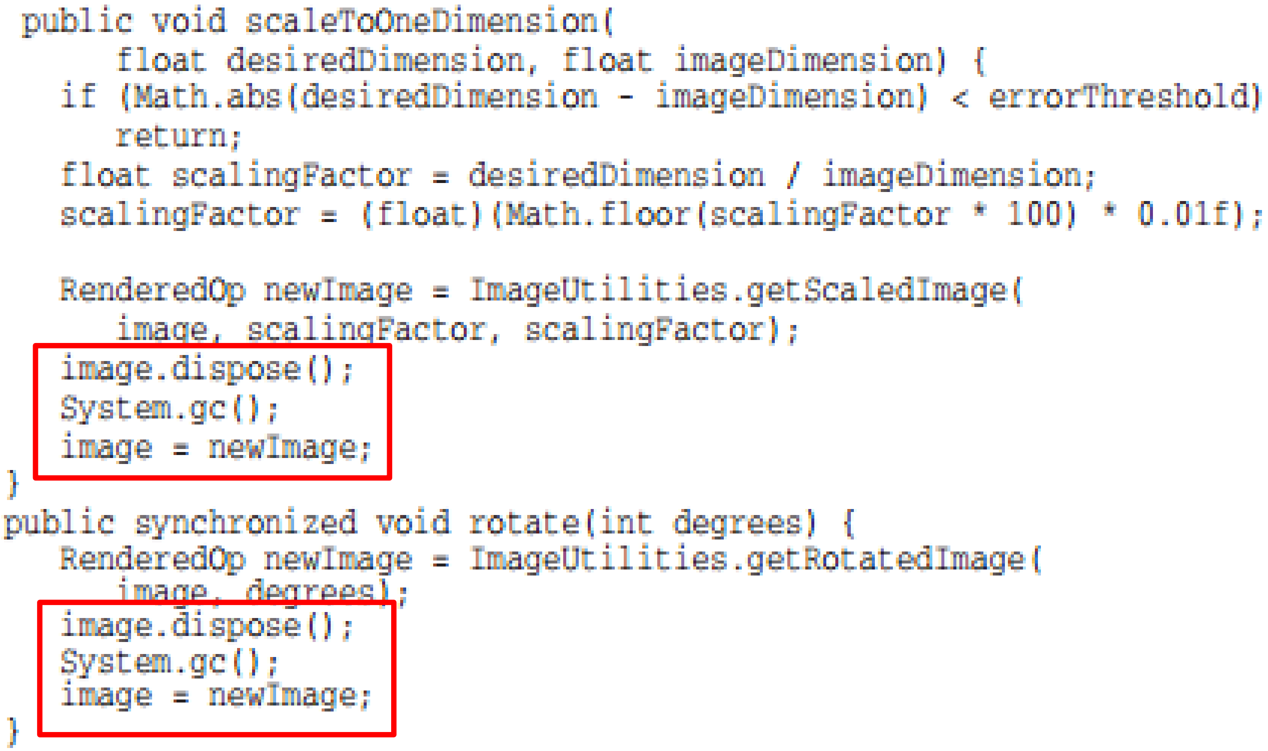
- Clarifies rather than obfuscates
- Makes programs read naturally
- Encourages best coding practices
  - Ease of reading and understanding is more important than ease of writing/”clever coding”
- Promotes intention-revealing code
  - Names

Some Clean Code Heuristics

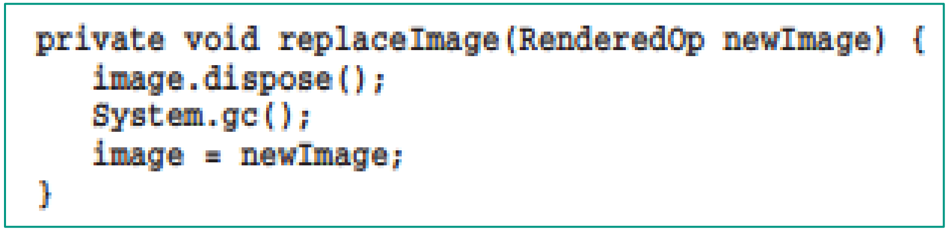


- Avoid duplication – Don't Repeat Yourself (DRY)
- Use explanatory variables
- Replace magic numbers with named constants
- Encapsulate conditionals
- Avoid negative conditionals
- Encapsulate boundary conditions
- Function names should say what they do
- Functions should do one thing

No Duplication (DRY: Don’t Repeat Yourself) (1)



```
 public void scaleToOneDimension(
      float desiredDimension, float imageDimension) {
   if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
      return;
   float scalingFactor = desiredDimension / imageDimension;
   scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);

   RenderedOp newImage = ImageUtilities.getScaledImage(
      image, scalingFactor, scalingFactor);
   image.dispose();
   System.gc();
   image = newImage;
}
public synchronized void rotate(int degrees) {
   RenderedOp newImage = ImageUtilities.getRotatedImage(
      image, degrees);
   image.dispose();
   System.gc();
   image = newImage;
}
```



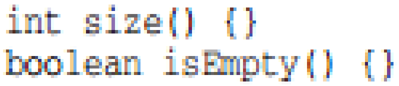
```
private void replaceImage(RenderedOp newImage) {
   image.dispose();
   System.gc();
   image = newImage;
}
```

- Refactor code, e.g. abstract common code block into separate function (method)

Algebra: Factoring

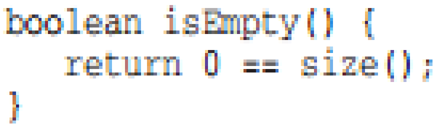
- A * B + A * C = A * (B + C)

No Duplication (DRY: Don’t Repeat Yourself) (2)



```
int size() {}
boolean isEmpty() {}
```

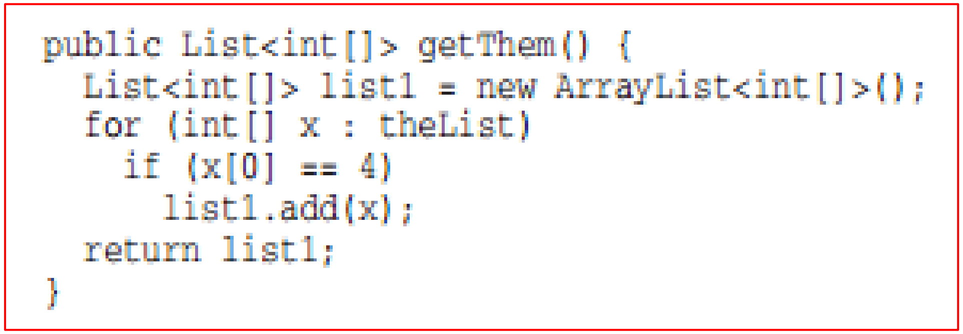
- Two separate implementations: duplicates semantics of empty collection



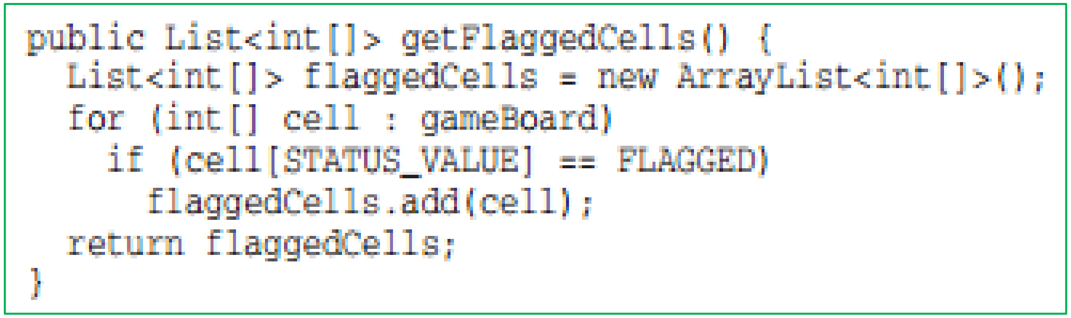
```
boolean isEmpty() {
   return 0 == size();
}
```

- Define isEmpty as a function of size(): makes dependency explicit

Replace magic numbers with named constants



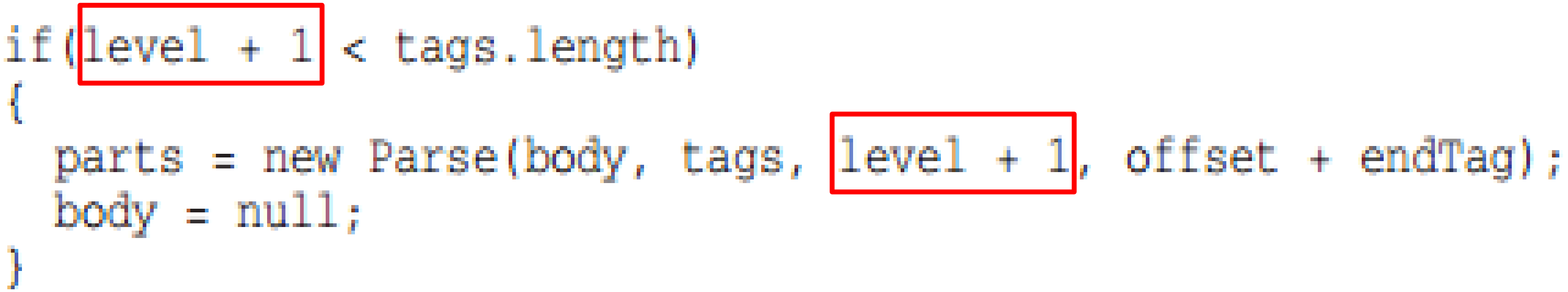
```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```



```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

- Use meaningful constant names instead of constant literals
- Use meaningful variable names instead of generic names

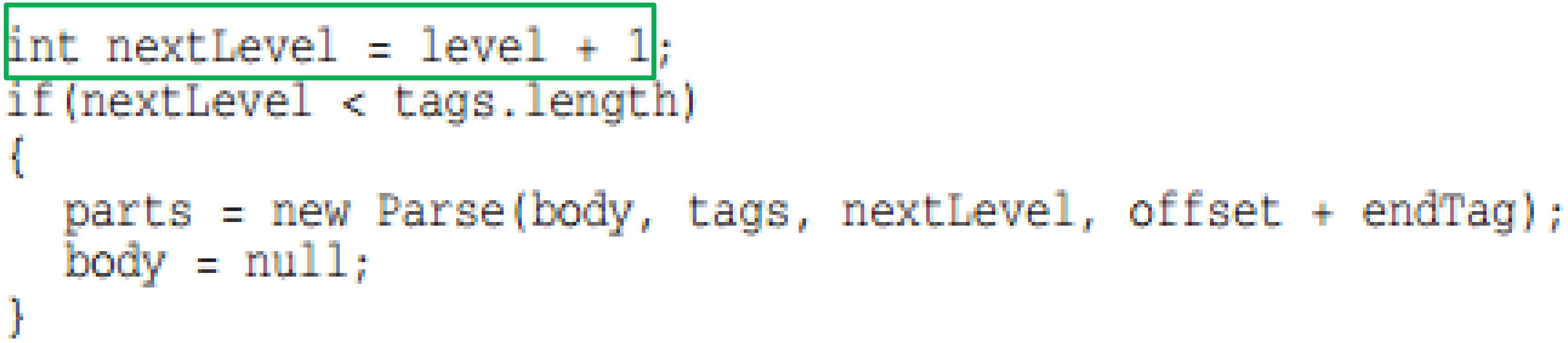
Encapsulate Expressions in Boundary Conditions



```
if(level + 1 < tags.length)
{
  parts = new Parse(body, tags, level + 1, offset + endTag);
  body = null;
}
```

Level + 1 occurs twice;

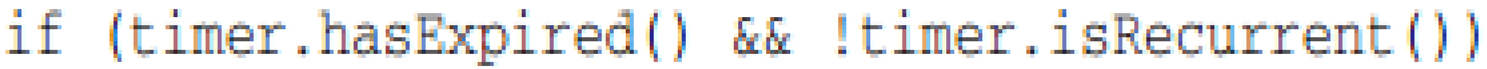
Encapsulate (and DRY):



```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

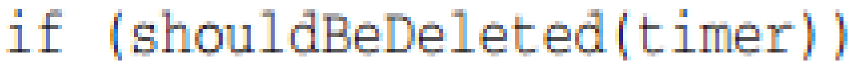
Encapsulate Conditionals

- Instead of:



```
if (timer.hasExpired() && !timer.isRecurrent())
```

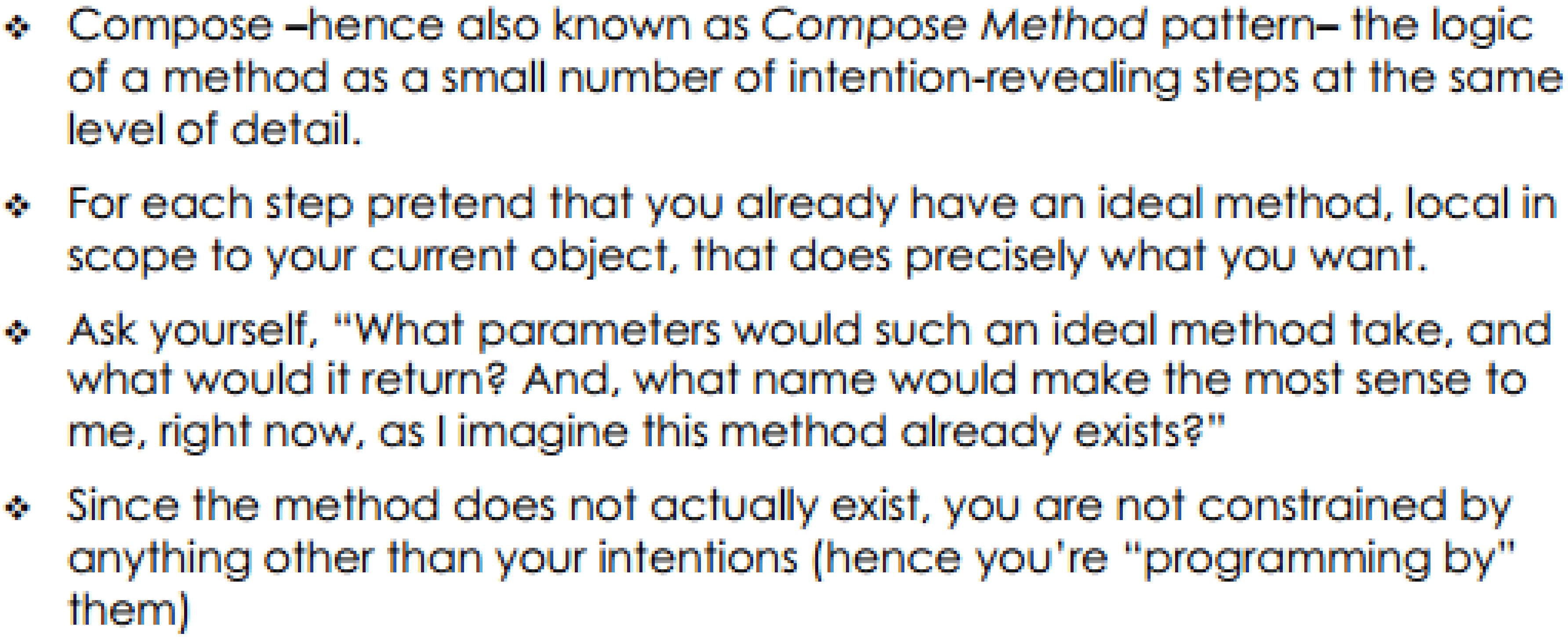
- prefer



```
if (shouldBeDeleted(timer))
```

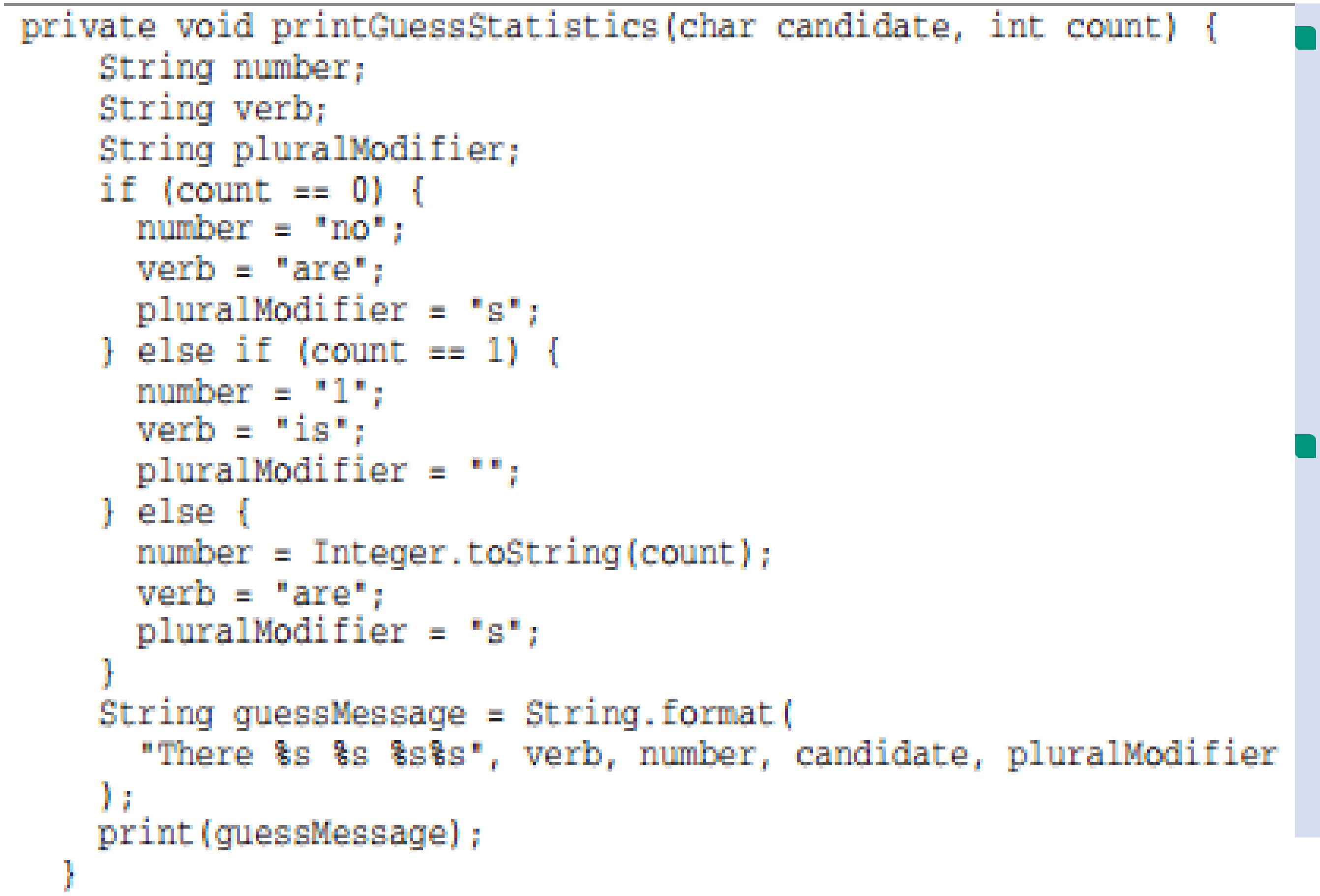
- Reference: Martin, *Clean Code* (Canvas > Pages > Reading Material – Software Engineering)

Programming by Intention



- Compose –hence also known as Compose *Method* pattern– the logic of a method as a small number of intention-revealing steps at the same level of detail.
- For each step pretend that you already have an ideal method, local in scope to your current object, that does precisely what you want.
- Ask yourself, "What parameters would such an ideal method take, and what would it return? And, what name would make the most sense to me, right now, as I imagine this method already exists?"
- Since the method does not actually exist, you are not constrained by anything other than your intentions (hence you're "programming by" them)

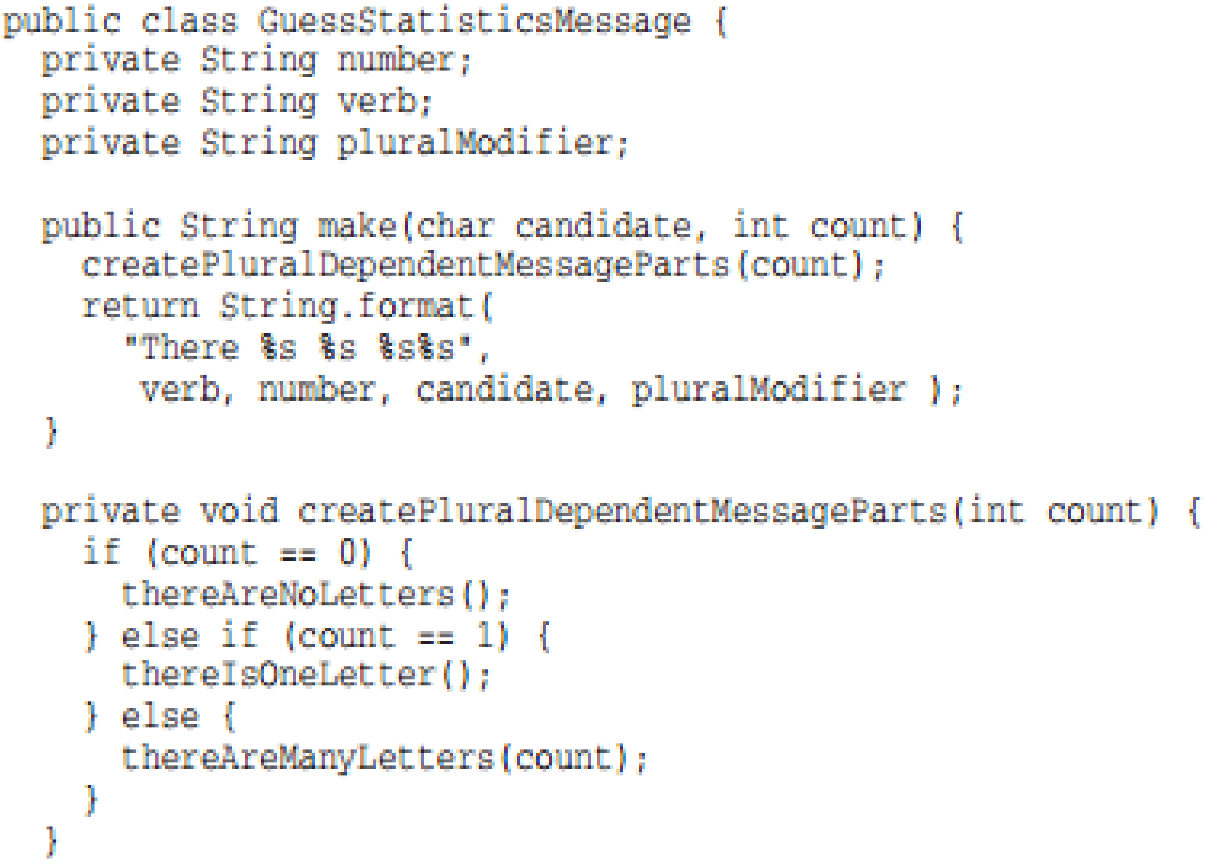
Example: GuessStatisticsMessage



```
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);
 }
```

- Break up long method by abstracting pieces into separate functions (methods)
- Makes code understandable at a higher level of abstraction

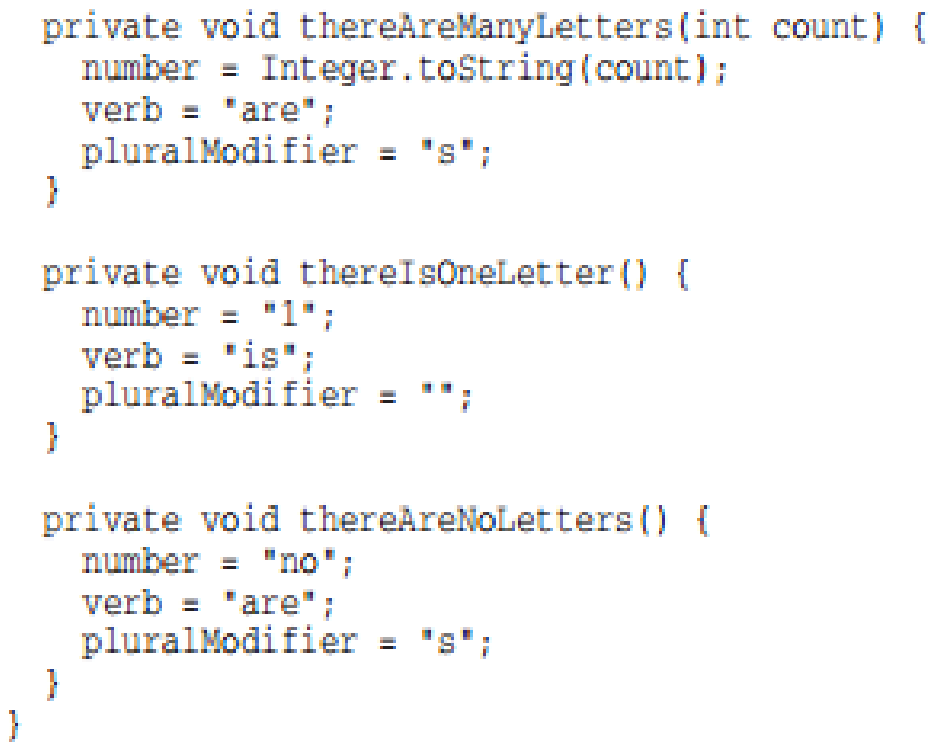
Example: GuessStatisticsMessage (2)



```java
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);
    }
  }
```

- Clear context for variables: attributes of new class
- New method names capture key semantics in application-specific terms

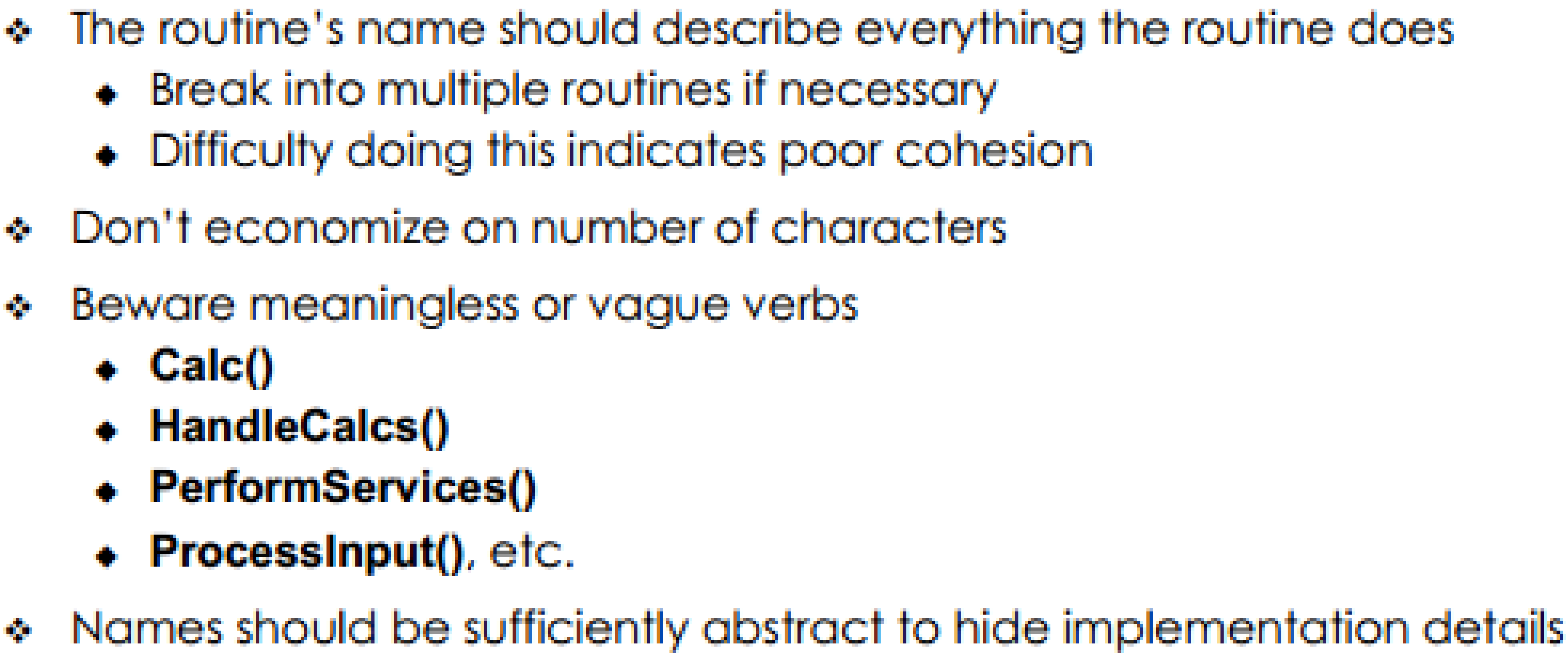
Example: GuessStatisticsMessage (3)



```
  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";
  }
}
```

- Definition of new abstractions

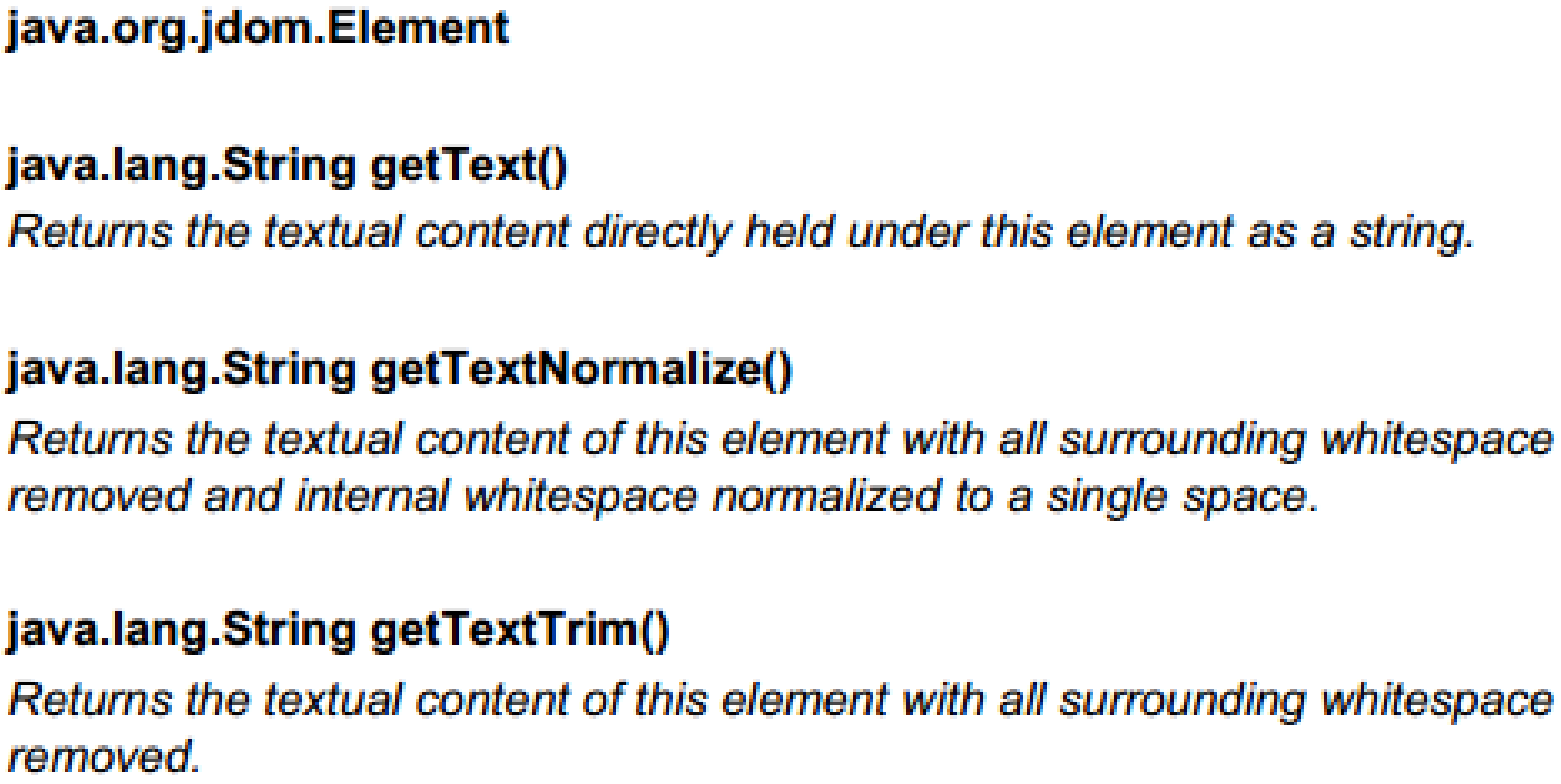
Routine/Function/Procedure/Method Names



- The routine's name should describe everything the routine does
  - Break into multiple routines if necessary
  - Difficulty doing this indicates poor cohesion
- Don't economize on number of characters
- Beware meaningless or vague verbs
  - **Calc()**
  - **HandleCalcs()**
  - **PerformServices()**
  - **ProcessInput()**, etc.
- Names should be sufficiently abstract to hide implementation details

- Aim for high cohesion and loose coupling

Different Behaviors → Different Methods



java.org.jdom.Element

java.lang.String getText()

Returns the textual content directly held under this element as a string.

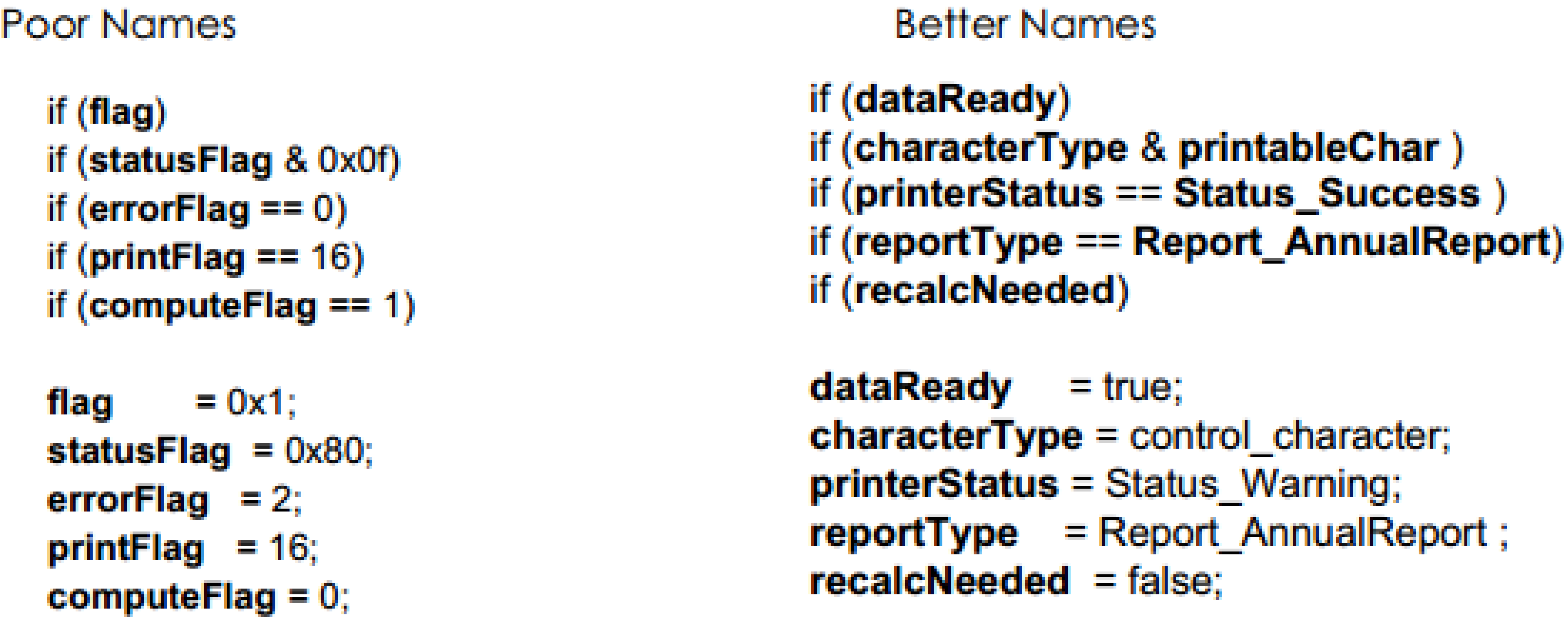
java.lang.String getTextNormalize()

Returns the textual content of this element with all surrounding whitespace removed and internal whitespace normalized to a single space.

java.lang.String getTextTrim()

Returns the textual content of this element with all surrounding whitespace removed.

Variable Names



Poor Names

```
if (flag)
if (statusFlag & 0x0f)
if (errorFlag == 0)
if (printFlag == 16)
if (computeFlag == 1)

flag        = 0x1;
statusFlag  = 0x80;
errorFlag   = 2;
printFlag   = 16;
computeFlag = 0;
```

Better Names

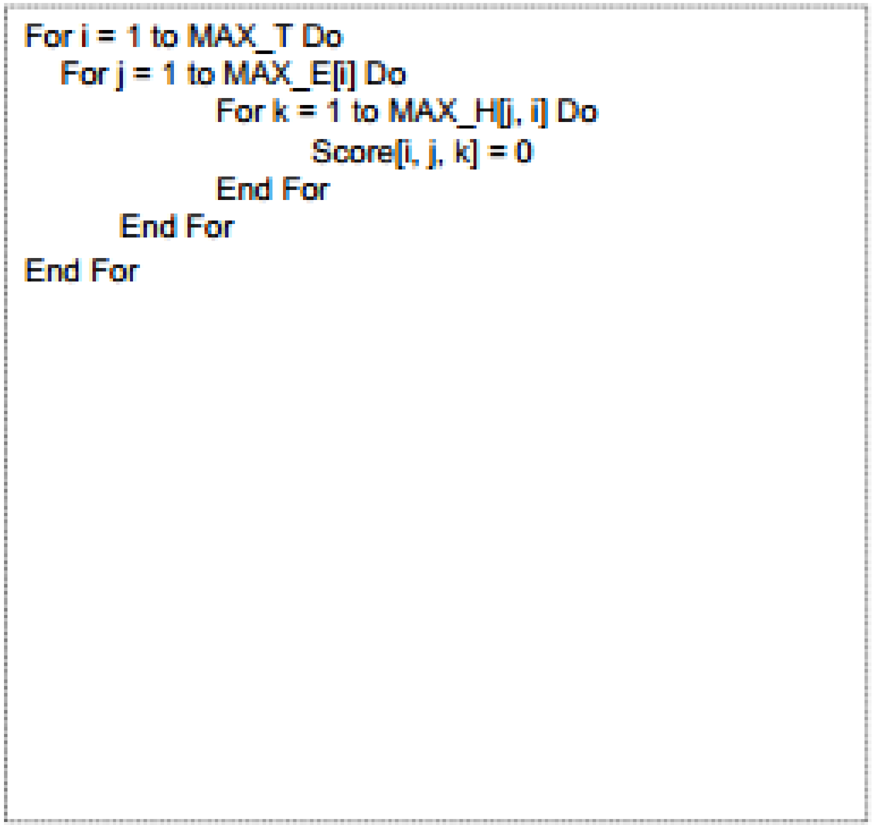
```
if (dataReady)
if (characterType & printableChar )
if (printerStatus == Status_Success )
if (reportType == Report_AnnualReport)
if (recalcNeeded)

dataReady     = true;
characterType = control_character;
printerStatus = Status_Warning;
reportType    = Report_AnnualReport ;
recalcNeeded  = false;
```

Managing Complexity

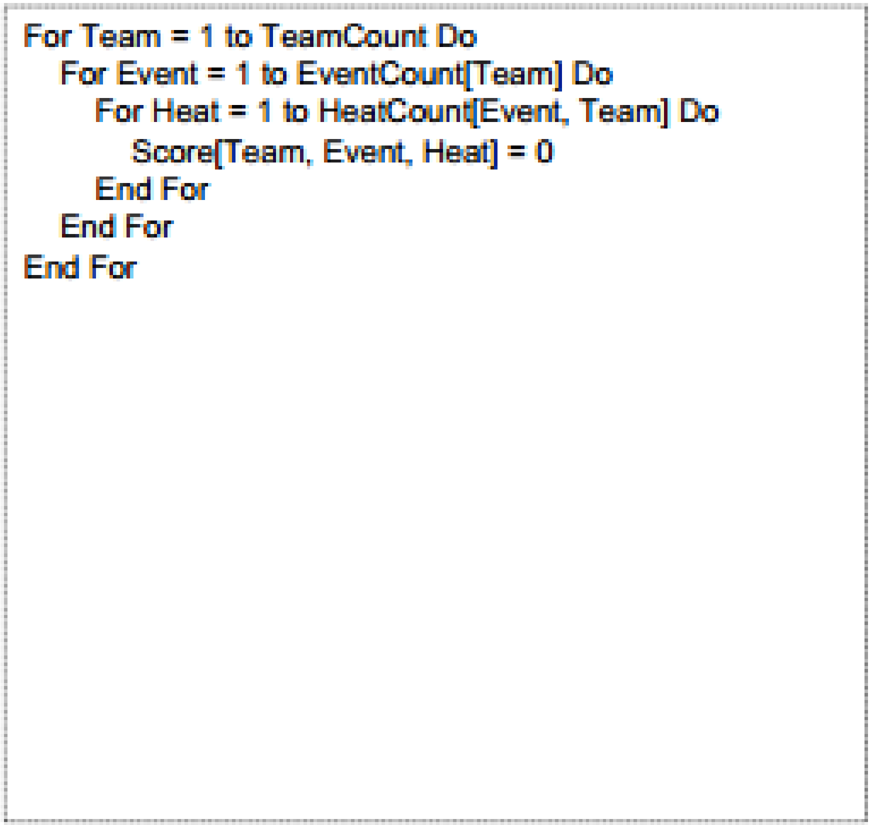
- High Cohesion
  - Components of a composite entity (class, object)
  - Have high cohesion
  - If the components tend to occur together in most contexts

- Loose Coupling
  - Components are loosely coupled
  - **If** there are few dependencies between the components

Loop Indices



```
For i = 1 to MAX_T Do
  For j = 1 to MAX_E[i] Do
          For k = 1 to MAX_H[j, i] Do
               Score[i, j, k] = 0
          End For
     End For
End For
```



```
For Team = 1 to TeamCount Do
  For Event = 1 to EventCount[Team] Do
    For Heat = 1 to HeatCount[Event, Team] Do
      Score[Team, Event, Heat] = 0
    End For
  End For
End For
```

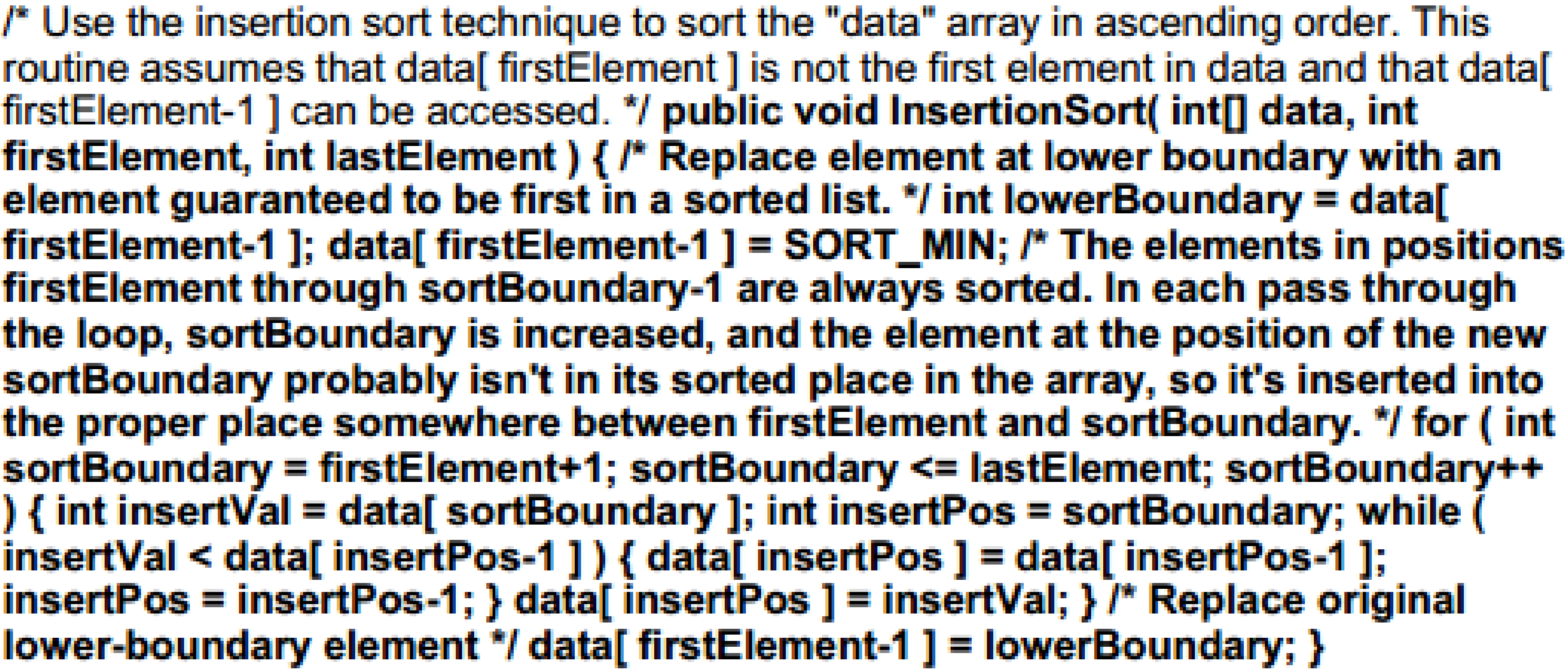
- Names of loop indices/bounds convey meaning of structural significance of index
  - In simple loops, single-letter loop index may be acceptable.

Exercise: Naming Guidelines

- Which of the naming guidelines provided would provide biggest improvement in your code?

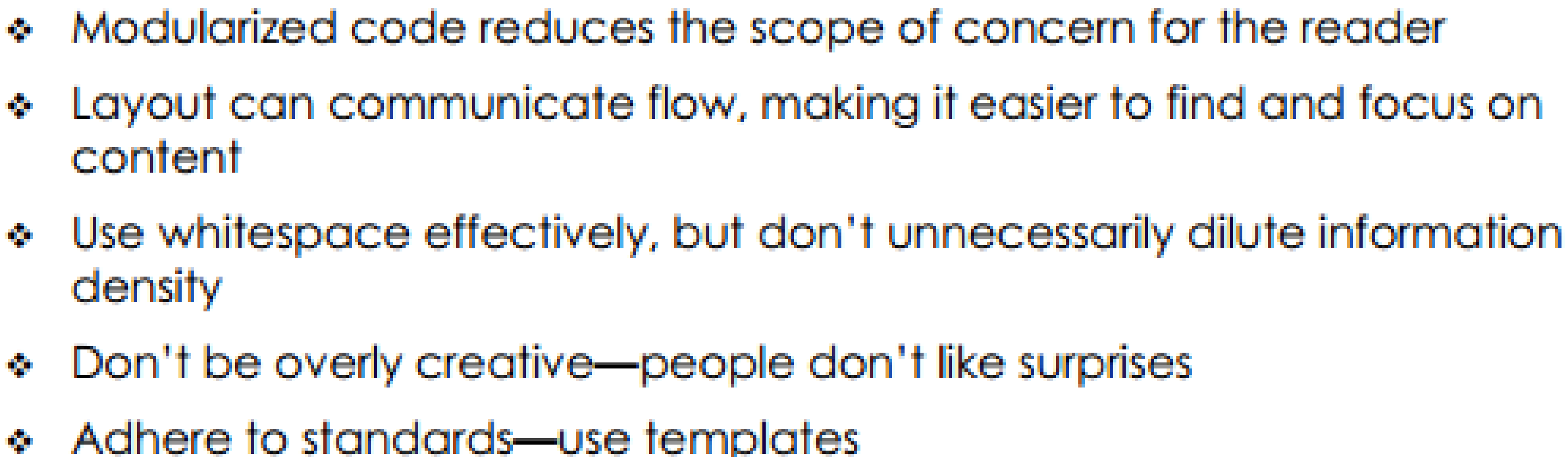
- Discuss with the person(s) sitting next to you

Layout Affects Readability



```
/* Use the insertion sort technique to sort the "data" array in ascending order. This
routine assumes that data[ firstElement ] is not the first element in data and that data[
firstElement-1 ] can be accessed. */ public void InsertionSort( int[] data, int
firstElement, int lastElement ) { /* Replace element at lower boundary with an
element guaranteed to be first in a sorted list. */ int lowerBoundary = data[
firstElement-1 ]; data[ firstElement-1 ] = SORT_MIN; /* The elements in positions
firstElement through sortBoundary-1 are always sorted. In each pass through
the loop, sortBoundary is increased, and the element at the position of the new
sortBoundary probably isn't in its sorted place in the array, so it's inserted into
the proper place somewhere between firstElement and sortBoundary. */ for ( int
sortBoundary = firstElement+1; sortBoundary <= lastElement; sortBoundary++
) { int insertVal = data[ sortBoundary ]; int insertPos = sortBoundary; while (
insertVal < data[ insertPos-1 ] ) { data[ insertPos ] = data[ insertPos-1 ];
insertPos = insertPos-1; } data[ insertPos ] = insertVal; } /* Replace original
lower-boundary element */ data[ firstElement-1 ] = lowerBoundary; }
```

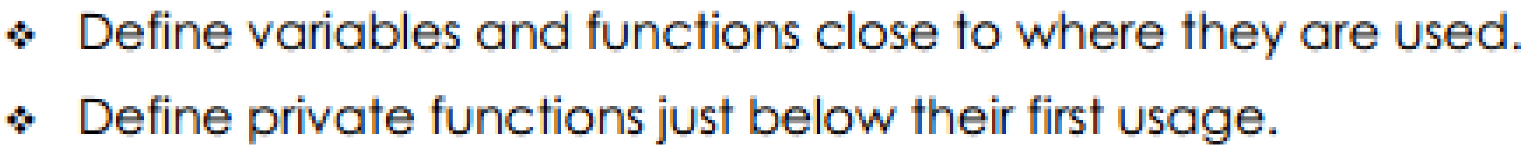
Layout Affects Readability



- Modularized code reduces the scope of concern for the reader
- Layout can communicate flow, making it easier to find and focus on content
- Use whitespace effectively, but don't unnecessarily dilute information density
- Don't be overly creative—people don't like surprises
- Adhere to standards—use templates

- Coding standards/style guides typically address use of white space (indentation), placement of parenthesis, …

Vertical Separation

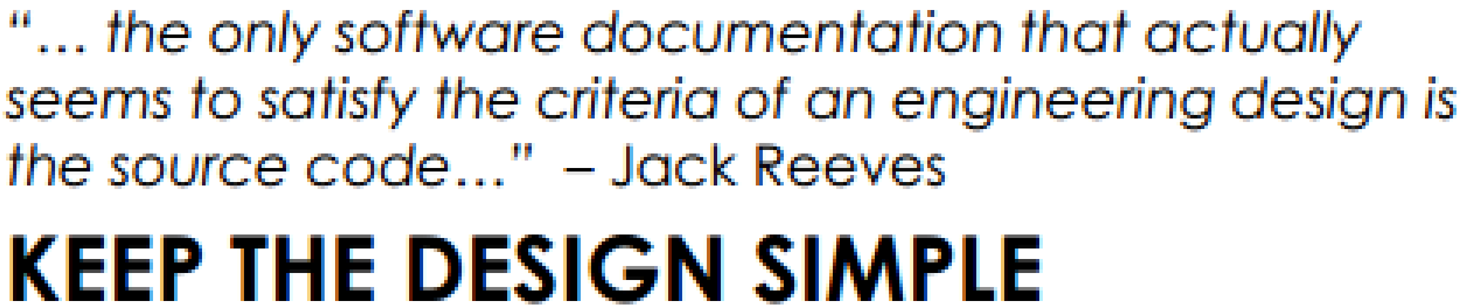


- Define variables and functions close to where they are used.
- Define private functions just below their first usage.

Minimize

- demands on reader’s memory
- Time spent browsing through the code

Simple Design



"… the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code…" – Jack Reeves

KEEP THE DESIGN SIMPLE

Useful Resources

- Robert Martin, *Clean Code* (2009)
  - Chapter 12: Emergence
- Corey Haines,
  Understanding the 4 Rules of Simple Design, 2014
  - Section on *4 Rules of Simple Design*
  - Check out link iterative nature of the 4 rules

- Both resources posted on
  Canvas > Pages > Reading Material – Software Engineering

What is Simple Design?

- Form a group with people next to you

- Write down some criteria for or attributes of Simple Design

Simple Design Criteria

- No super long lines
- Consistent indentation
- Good abstractions, well-named functions
- Consistent style
- Easy to make changes
- Unique variables
- Concise comments
- Visual separation of code blocks
- Good function documentation
- Good repo organization

What is Simple Design?

- Write down some criteria for or attributes of Simple Design

- Form a group with people next to you

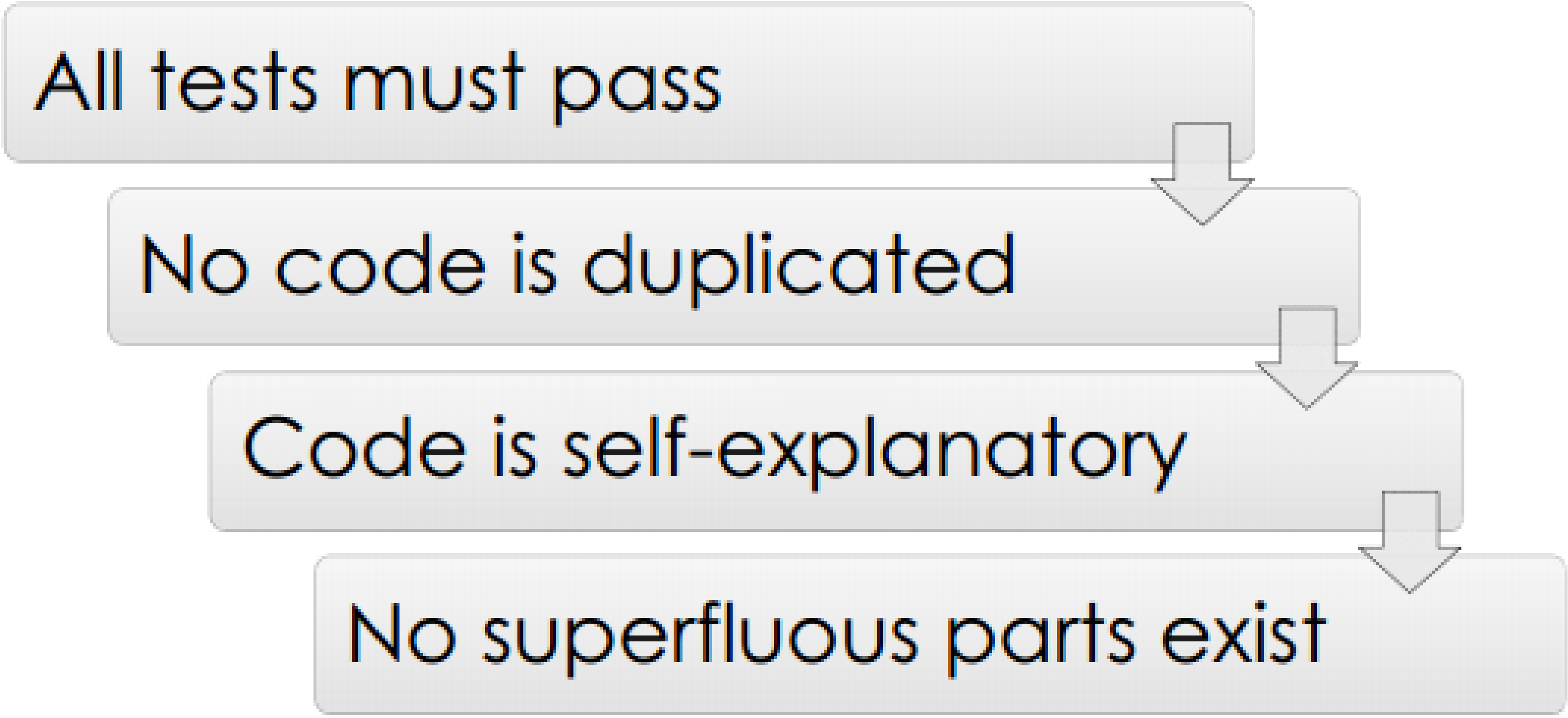
- Create a table with four columns:
  - All tests must pass
  - No code is duplicated
  - Code is self-explanatory
  - No superfluous parts exist

- Sort your collected criteria into these columns

Four Rules of Simple Design

- All tests must pass
  - Tests are written before code to be tested
  - If all tests pass then code is “correct”
- Code contains no duplication (DRY)
  - Duplicated code/concepts suggest abstraction
- Code is self-explanatory
  - Code reveals intent
  - Well-chosen names critical
    - Often reflect application domain concepts
- No superfluous parts
  - Minimal number of classes and methods
  - No extraneous/legacy parts

Rules of Simple Design



All tests must pass

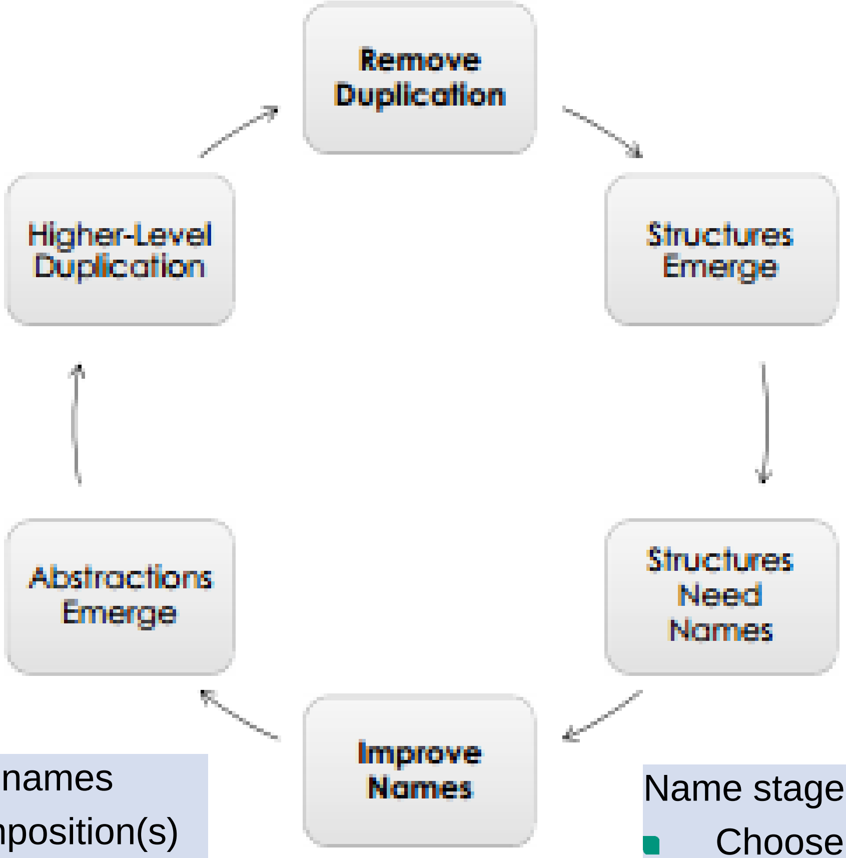
No code is duplicated

Code is self-explanatory

No superfluous parts exist

- Note: some authors reverse step 2 and step 3
- Consensus view (next slide) renders the issue moot

Simple Design Feedback Cycle



Finding meaningful names

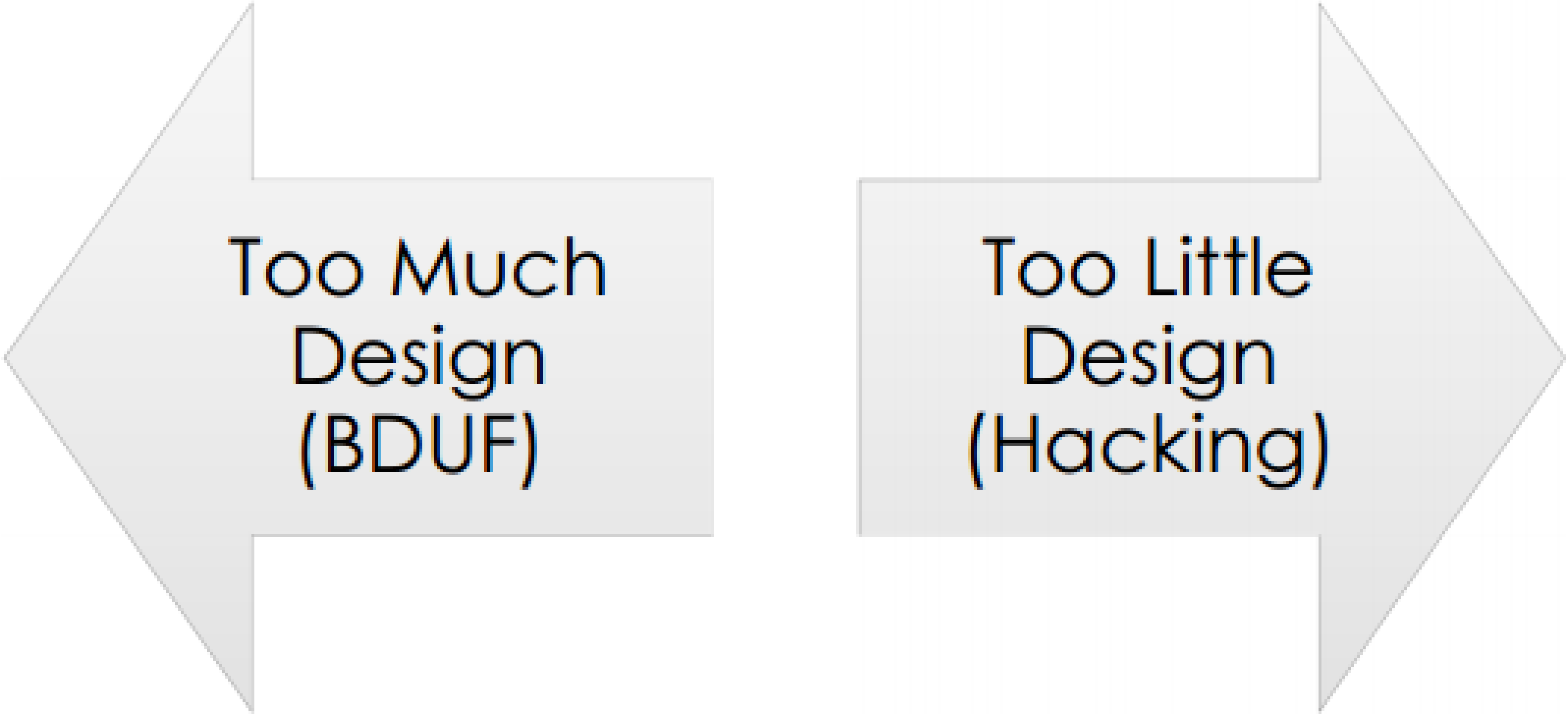
- Involves decomposition(s)
- Exposes new abstractions

Name stages

- Choose any name
- Find accurate name
- Find precise name
- Find meaningful name(s)

Cf. iterative nature of the 4 rules

The Design Dilemma

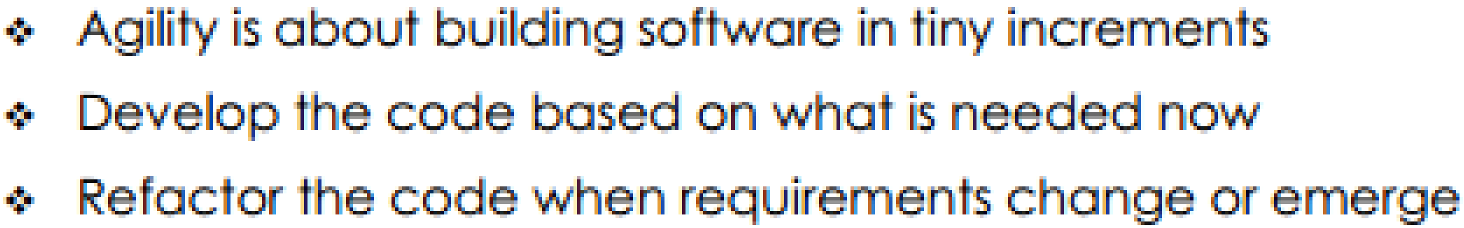


Too Much Design (BDUF)

Too Little Design (Hacking)

- BDUF: Big Design Up Front
  - Frowned upon by Agile folk
- See also RAC4:
  Waterman (2015) Grounded theory of agile architectures

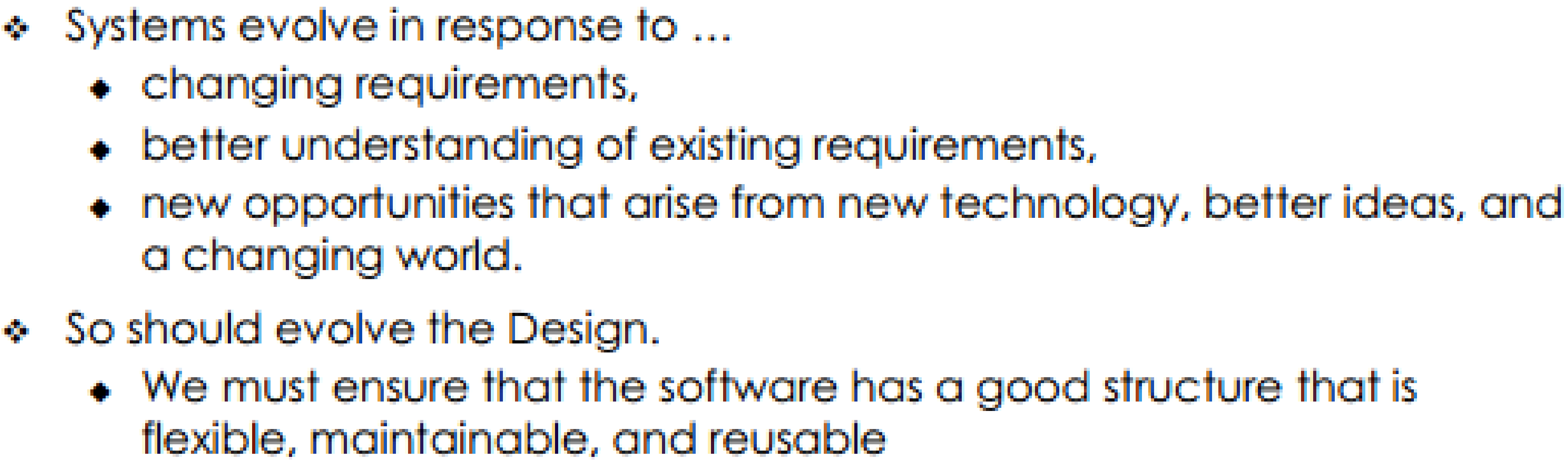
Emerging Design



- Agility is about building software in tiny increments
- Develop the code based on what is needed now
- Refactor the code when requirements change or emerge

- Emerging design: design (pattern) discovery
- Design patterns: records of other people’s discoveries

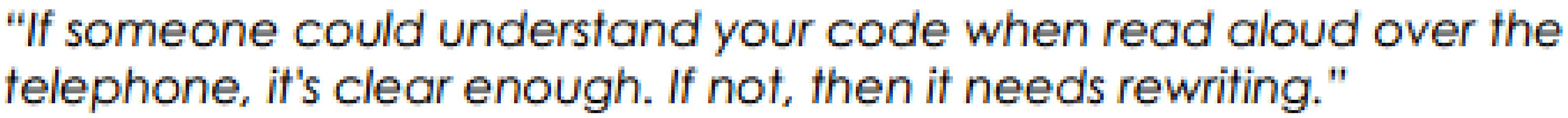
Emergent Design



- Systems evolve in response to …
  - changing requirements,
  - better understanding of existing requirements,
  - new opportunities that arise from new technology, better ideas, and a changing world.
- So should evolve the Design.
  - We must ensure that the software has a good structure that is flexible, maintainable, and reusable

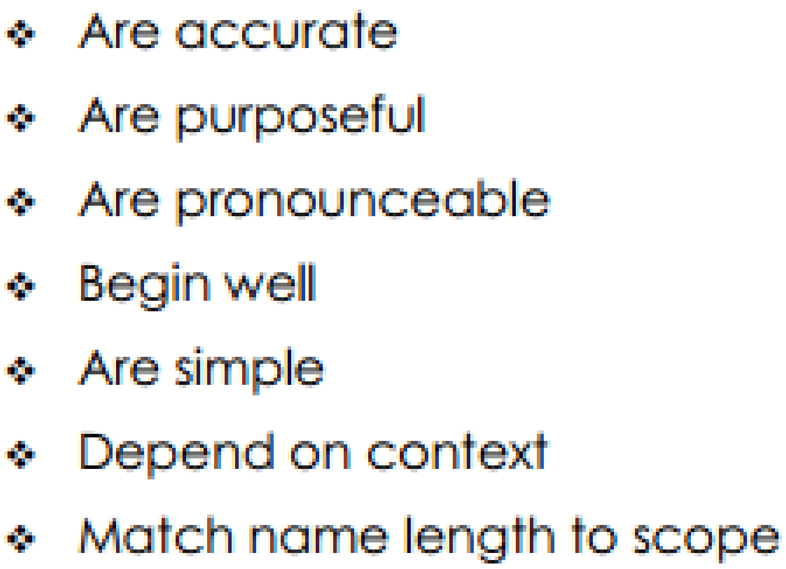
- Discussion: Agile vs. Architecture
  - Debate is on-going
  - See also Kruchten links on Canvas > Pages > Reading Material – Software Eng.

The Telephone Test



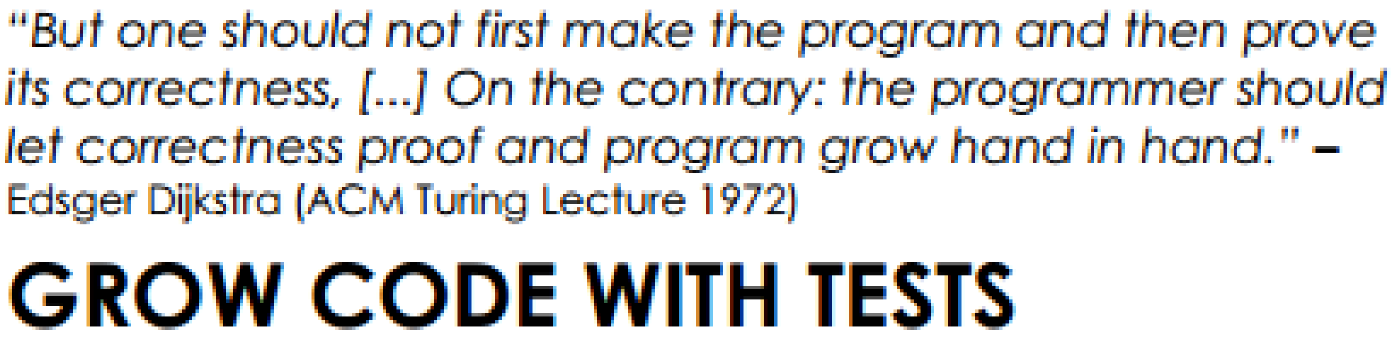
"If someone could understand your code when read aloud over the telephone, it's clear enough. If not, then it needs rewriting."

Really Meaningful Names



- Are accurate
- Are purposeful
- Are pronounceable
- Begin well
- Are simple
- Depend on context
- Match name length to scope

Grow Code with Tests

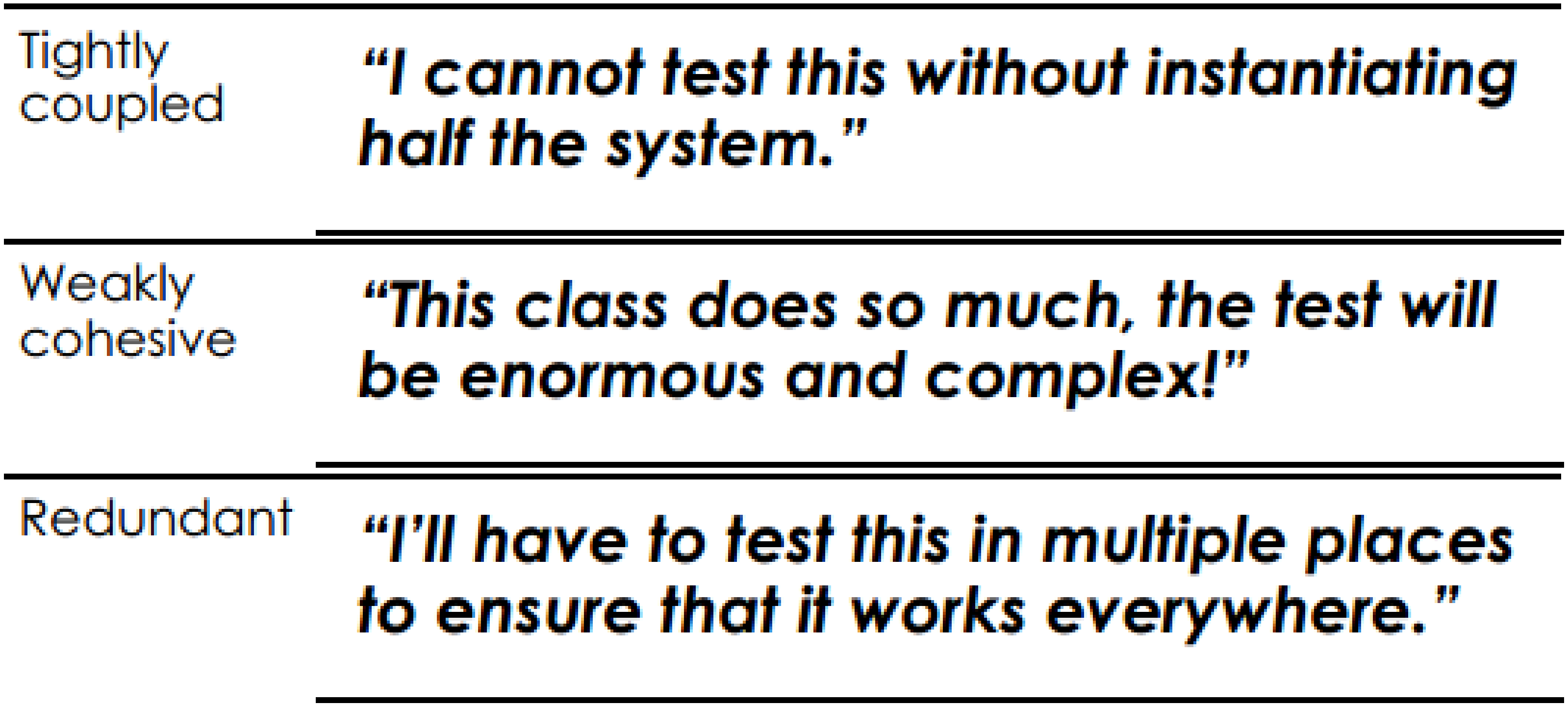


"But one should not first make the program and then prove its correctness, [...] On the contrary: the programmer should let correctness proof and program grow hand in hand." – Edsger Dijkstra (ACM Turing Lecture 1972)

GROW CODE WITH TESTS

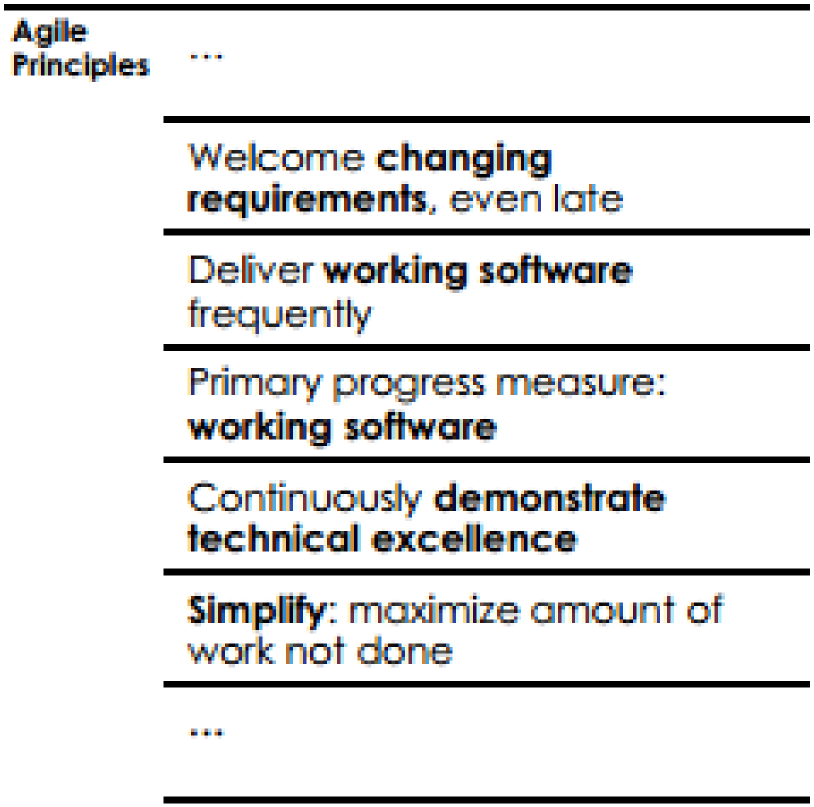
- An old idea: develop program and proof hand-in-hand
- Now: develop tests and programs hand-in-hand
- … and the old idea is coming back as well

Hard-to-Test Code is …

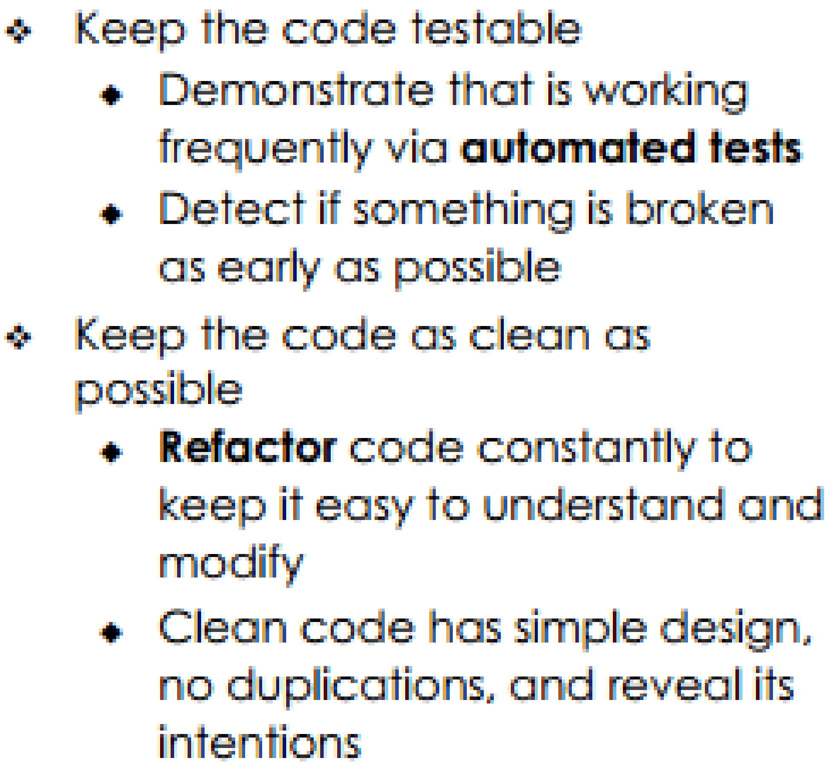


Tightly coupled	*“I cannot test this without instantiating half the system.”*
Weakly cohesive	*“This class does so much, the test will be enormous and complex!”*
Redundant	*“I’ll have to test this in multiple places to ensure that it works everywhere.”*

Unexpected Changes and Working Software



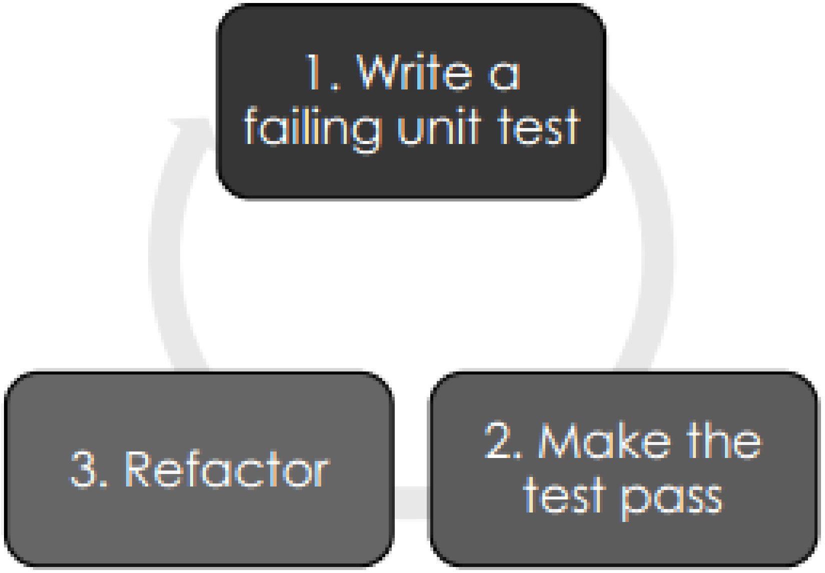
Agile Principles	…
	Welcome **changing requirements**, even late
	Deliver **working software** frequently
	Primary progress measure: **working software**
	Continuously **demonstrate technical excellence**
	Simplify: maximize amount of work not done
	…



- Keep the code testable
  - Demonstrate that is working frequently via **automated tests**
  - Detect if something is broken as early as possible
- Keep the code as clean as possible
  - **Refactor** code constantly to keep it easy to understand and modify
  - Clean code has simple design, no duplications, and reveal its intentions

Test-Driven Development (TDD)

"We only write new code when we have a test that doesn't work"



1. Write a failing unit test

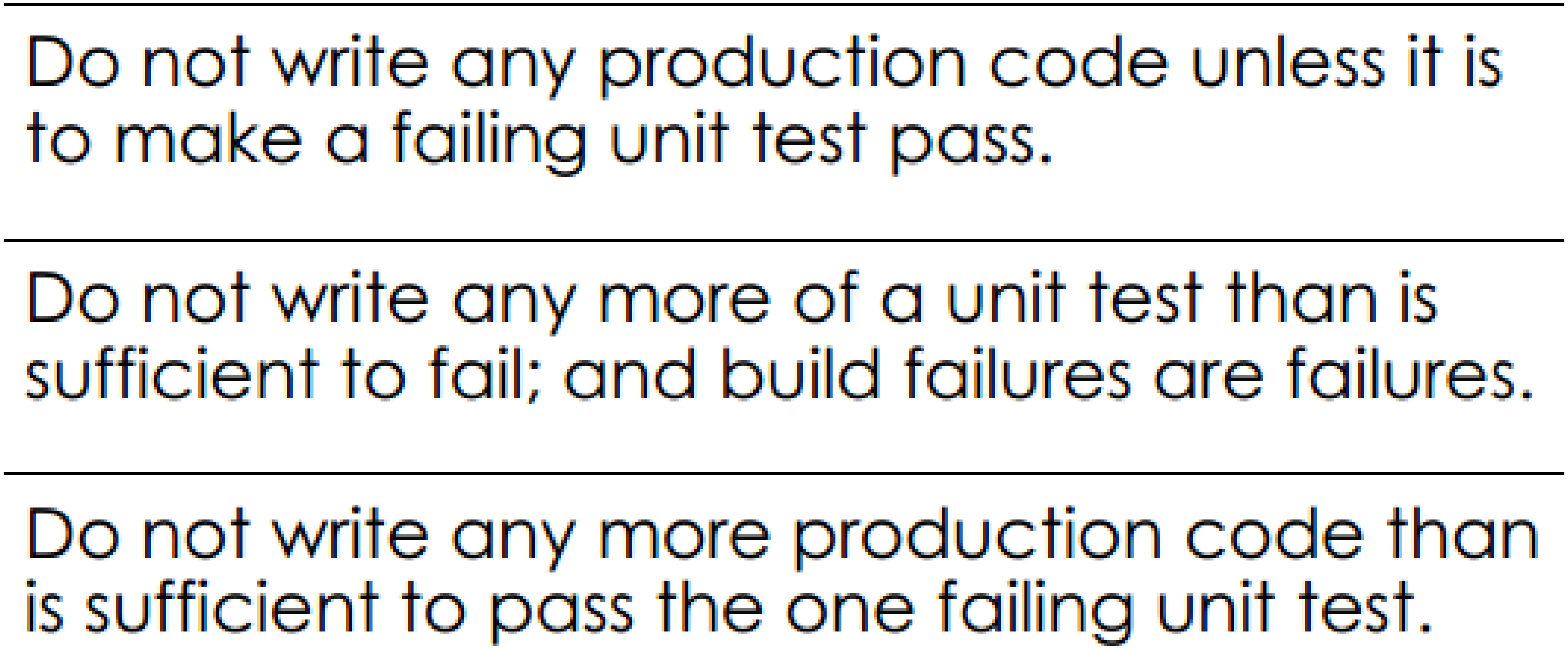
2. Make the test pass

3. Refactor

"TDD is primarily a design technique with a side effect of ensuring that your source code is thoroughly unit tested"
– Scott W. Ambler

- Turns testing into a design activity
  - Consumption awareness: test case code represents how you would like to access the functionality
  - Lead to programming by intention
- Provides continuous feedback
  - *"does it work?", "is it well structured?"*

Three Laws of TDD



Do not write any production code unless it is to make a failing unit test pass.

Do not write any more of a unit test than is sufficient to fail; and build failures are failures.

Do not write any more production code than is sufficient to pass the one failing unit test.

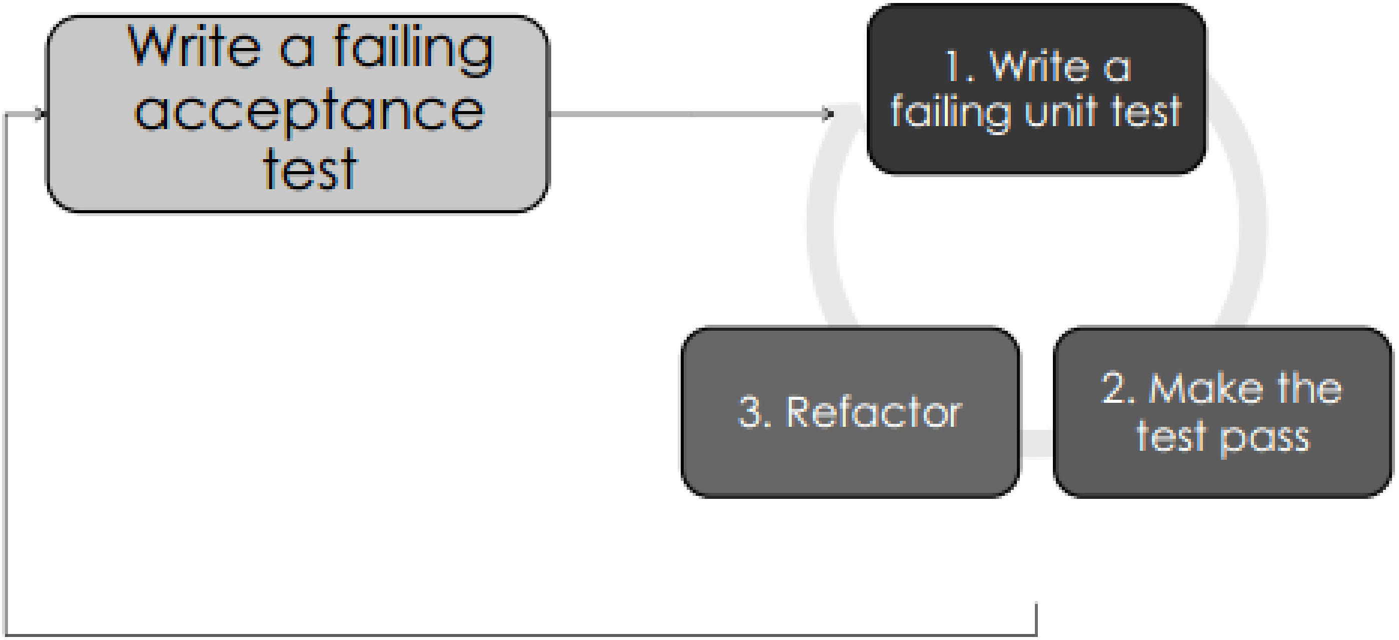
Desirable Test Characteristics (F.I.R.S.T.)

- Fast
  - tests will run frequently
  - Small and simple: test one concept at a time
- Independent
  - No dependencies between tests
  - Tests can run in any order
  - Simplifies failure analysis (debugging)
- Repeatable
  - Tests can run at any time, in any order
- Self-Validating
  - Test either pass or fail (Boolean result)
- Timely
  - Write the tests when you need them
  - In TDD: write test first, then code

Clean Tests

- Test code is code, too.
- Test code requires same care as production code
  - Readable
  - Understandable
  - Self-explanatory; clear intent
- Dirty test code
  - Same, or worse than, no tests
  - Ill-structured tests increasingly harder to change and grow

Bigger Picture: Acceptance TDD



- The code for passing an acceptance test
  (for a single *acceptance criterion*)
  is built incrementally based on several unit tests
- Acceptance TDD also known as Behavior Driven Dev (BDD)