university
of
tyumen

school
of advanced
studies

Information Technology - intermediate

Lecture 6
Sound programming

Fabio Grazioso - *April 2018*

low level - high level languages

low level language

- low level (machine language)
  - reading memory location
  - writing memory location
  - operations on binary numbers
- PROs
  - fast
  - small memory footprint
- CONs
  - machine-dependent
  - difficult

low level language

- low level (assembly)
- similar to machine language
- mnemonic acronyms
- translation from menonic to machine codes = assembly

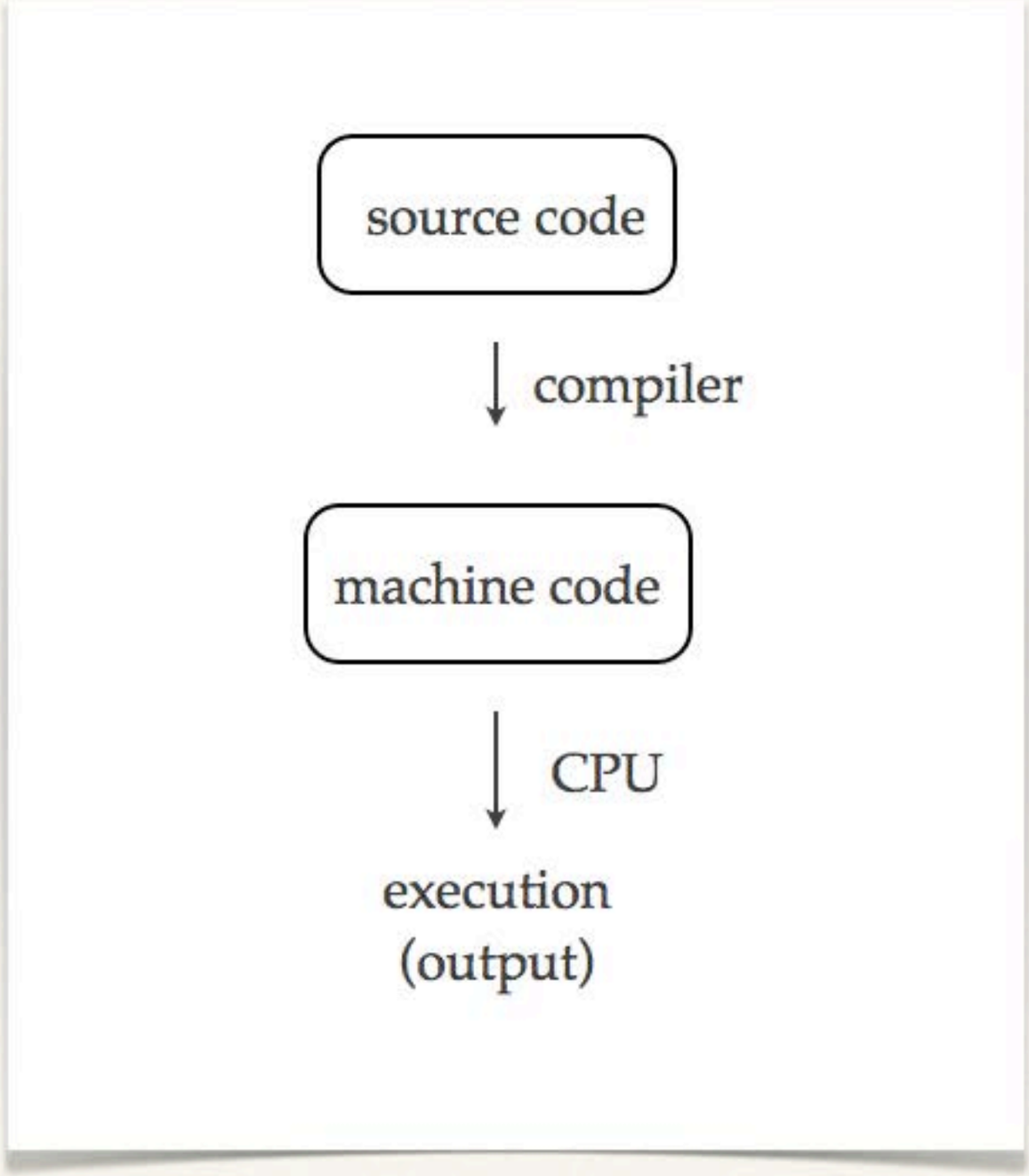
high level language

- instrunctions closer to human language
- PROs
  - easier to learn
  - easier to write (more complex instructions)
  - machine-independent
- CONs
  - translation to machine language is needed => slower
  - more memory usage

compiled and interpreted languages

compiled

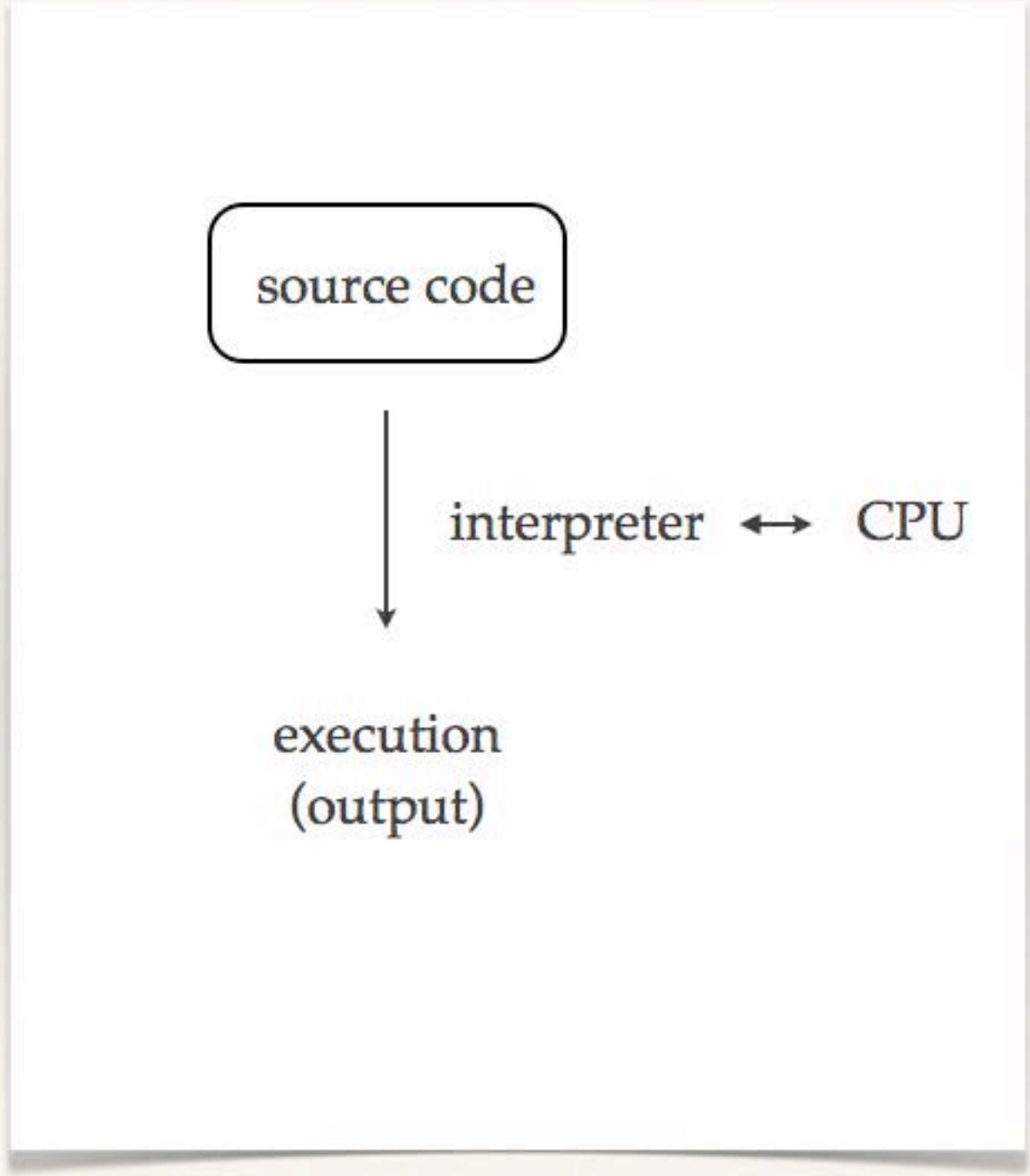
- PROs
  - faster
    - it is compiled only once
- CONs
  - needs an intermediate step (machine code)
  - executes all the instructions at once



Fox p25

interpreted

- PROs
  - no intermediate step
    - execution step-by-step
- CONs
  - it is slower
  - needs to "compile" each time



Fox p95

Procedural Programming vs Object-Oriented Programming

Procedural Programming

- Each statement in the language tells the computer to do something: Get some input, add these numbers, divide by six, display that output.
- A program in a procedural language is a **list of instructions**.
- For very small programs, no other *organizing principle* (often called a *paradigm*) is needed.
- The programmer creates the list of instructions, and the computer carries them out.

Lafore p10

Structured Programming

- When programs become larger, a single list of instructions becomes unwieldy.
- We can **break down** the program into smaller units.
- Function are a way to make programs more comprehensible to their human creators. (The term function is used in C++ and C. In other languages the same concept may be referred to as a subroutine, a subprogram, or a procedure.)
- A procedural program is divided into **functions**, and (ideally, at least) each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.
- The idea of breaking a program into functions can be further extended by grouping a number of functions together into a larger entity called a **module** (which is often a **file**).
- Dividing a program into functions and modules is one of the cornerstones of structured programming, the somewhat loosely defined discipline that influenced programming organization for several decades before the advent of object-oriented programming.

Lafore p10

Structured Programming

- Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by
- making extensive use of the **structured control flow constructs** of selection (if/then/else) and **repetition block structures** (while and for), and subroutines
- in contrast to using simple tests and jumps such as the **go to statement**, which can lead to "*spaghetti code*" that is potentially difficult to follow and maintain.

Wikipedia
(Structured programming)

Object-oriented Programming

- object-oriented programming is not primarily concerned with the details of program operation. Instead, it deals with the **overall organization of the program**.
- Most individual program statements in C++ are similar to statements in procedural languages, and many are identical to statements in C. Indeed, an entire member function in a C++ program may be very similar to a procedural function in C.
- It is only when you look at the larger context that you can determine whether a statement or a function is part of a procedural C program or an object-oriented C++ program.

Lafore p15

Characteristics of Object-Oriented Languages

- An **object** has the same relationship to a **class** that a variable has to a data type. An object is said to be an instance of a class, in the same way my 1954 Chevrolet is an instance of a vehicle.
- The data items within a class are called data members (or sometimes member data). There can be any number of data members in a class, just as there can be any number of data items in a structure. The data member some data follows the keyword private, so it can be accessed from within the class, but not from outside.

- Member Functions
- Member functions are functions that are included within a class. In some object-oriented languages member functions are called methods; some writers use this term in C++ as well.
- However, when member functions are small, it is common to compress their definitions this way to save space.
- Functions Are Public, Data Is Private
- Usually the data within a class is private and the functions are public. This is a result of the way classes are used. The data is hidden so it will be safe from accidental manipulation, while the functions that operate on the data are public so they can be accessed from outside the class. However, there is no rule that says data must be private and functions public; in some circumstances you may find you'll need to use private functions and public data.


Lafore p217

Data hiding

- The body of the class contains two unfamiliar keywords: private and public. A key feature of object-oriented programming is data hiding.
- Data hiding, means hiding data from parts of the program that don't need to access it. More specifically, one class's data is hidden from other classes. Data hiding is designed to protect well-intentioned programmers from honest mistakes.

Lafore p217

Examples

- Defining the Class
- Here's the definition (sometimes called a specifier) for the class smallobj, copied from the SMALLOBJ listing:

```
class smallobj                  //define a class
    {
    private:
       int somedata;       //class data
    public:
       void setdata(int d)     //member function to set data
    { somedata = d; }
void showdata()                 //member function to display data
{ cout << "\nData is " << somedata; }


  };
```

Lafore p218

Examples

Using the Class

- Now that the class is defined, let's see how main() makes use of it. We'll see how objects are defined, and, once defined, how their member functions are accessed.

Defining Objects

- The first statement in main()

```
smallobj s1, s2;
```

- defines two objects, s1 and s2, of class smallobj. Remember that the definition of the class smallobj does not create any objects. It only describes how they will look when they are created, just as a structure definition describes how a structure will look but doesn't create any structure variables. It is objects that participate in program operations. Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory.
- Defining objects in this way means creating them. This is also called instantiating them. The term instantiating arises because an instance of the class is created. An object is an instance (that is, a specific example) of a class. Objects are sometimes called instance variables.

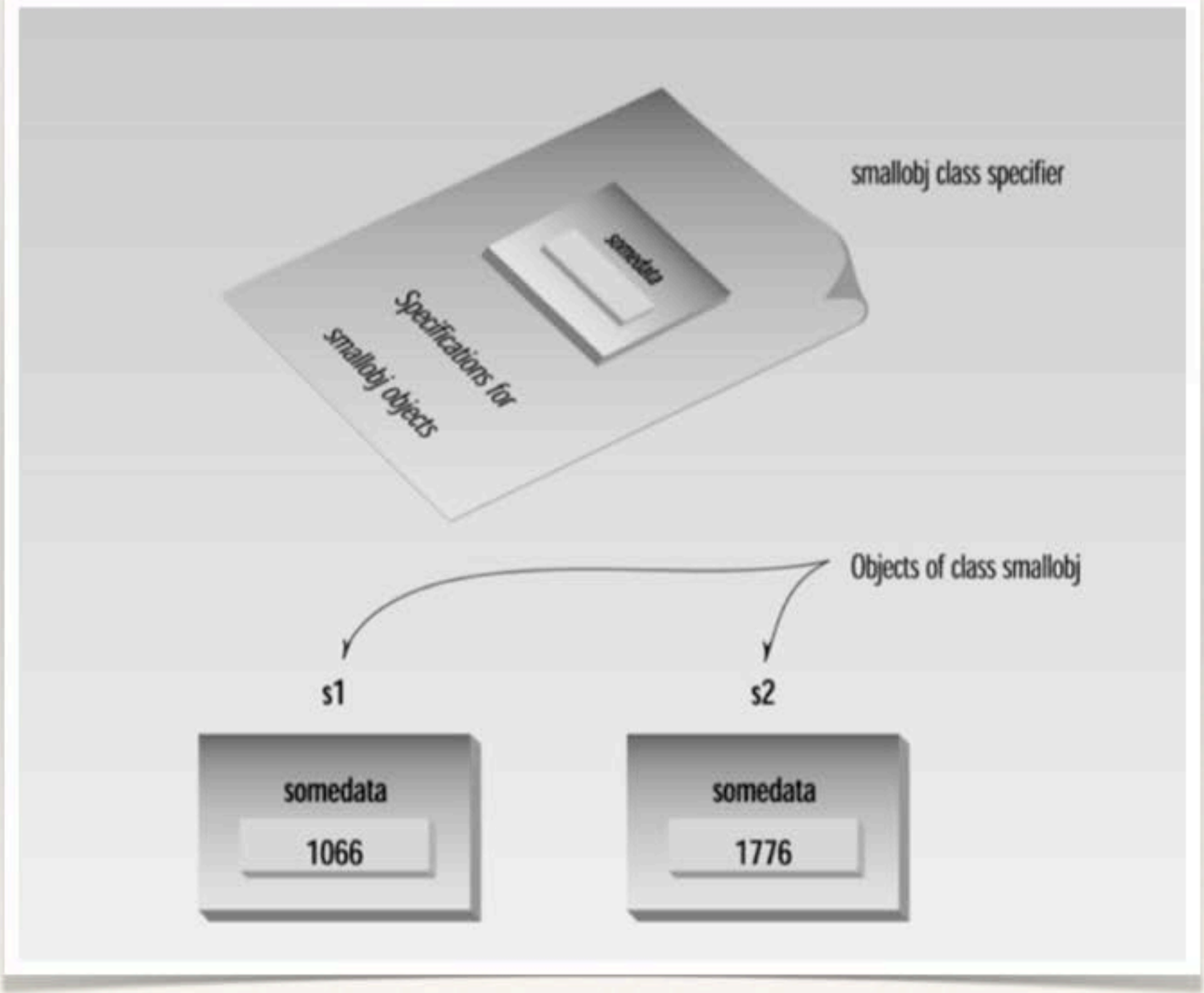
Calling Member Functions

- The next two statements in main() call the member function setdata():

```
s1.setdata(1066);
s2.setdata(1776);
```

Lafore p217

Examples



Lafore p222

Good practice

A possible list of rules

How should be a “good code”

- Good code is well-organized. Data and operations in classes fit together. There aren't extraneous dependencies between classes. It does not look like "spaghetti."
- Good code comments explain why things are done not what is done. The code itself explains what is done. The need for comments should be minimal.
- Good code uses meaningful naming conventions for all but the most transient of objects. the name of something is informative about when and how to use the object.
- Good code is well-tested. Tests serve as an executable specification of the code and examples of its use.
- Good code is not "clever". It does things in straightforward, obvious ways.
- Good code is developed in small, easy to read units of computation. These units are reused throughout the code.

Excerpts from Martin - Clean Code: A Handbook of Agile Software Craftsmanship (2008)

Use Pronounceable Names

Humans are good at words. A significant part of our brains is dedicated to the concept of words. And words are, by definition, pronounceable. It would be a shame not to take

advantage of that huge portion of our brains that has evolved to deal with spoken lan-

guage. So make your names pronounceable.

If you can't pronounce it, you can't discuss it without sounding like an idiot. "Well, over here on the bee cee arr three cee enn tee we have a pee ess zee kyew int, see?" This matters because programming is a social activity.

A company I know has `genymdhms` (generation date, year, month, day, hour, minute, and second) so they walked around saying "gen why emm dee aich emm ess". I have an annoying habit of pronouncing everything as written, so I started saying "gen-yah-mudda- hims." It later was being called this by a host of designers and analysts, and we still sounded silly. But we were in on the joke, so it was fun. Fun or not, we were tolerating poor naming. New developers had to have the variables explained to them, and then they spoke about it in silly made-up words instead of using proper English terms. Compare

```
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";

/* ... */

};
```

Intelligent conversation is now possible: "Hey, Mikey, take a look at this record! The gen- eration timestamp is set to tomorrow's date! How can that be?"

Use Searchable Names

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

One might easily grep for `MAX_CLASSES_PER_STUDENT`, but the number 7 could be more troublesome. Searches may turn up the digit as part of file names, other constant defini- tions, and in various expressions where the value is used with different intent. It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer's search.

Likewise, the name `e` is a poor choice for any variable for which a programmer might need to search. It is the most common letter in the English language and likely to show up in every passage of text in every program. In this regard, longer names trump shorter names, and any searchable name trumps a constant in code.

My personal preference is that single-letter names can ONLY be used as local vari- ables inside short methods. *The length of a name should correspond to the size of its scope*. If a variable or constant might be seen or used in multiple places in a body of code,

it is imperative to give it a search-friendly name. Once again compare:

```
for (int j=0; j<34; j++) { s += (t[j]*4)/5;
}
```

to

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
int realTaskDays = taskEstimate[j] * realDaysPerIdealDay; int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
sum += realTaskWeeks;
}
```

Note that `sum`, above, is not a particularly useful name but at least is searchable. The intentionally named code makes for a longer function, but consider how much easier it will be to find `WORK_DAYS_PER_WEEK` than to find all the places where 5 was used and filter the list down to just the instances with the intended meaning.

Martin p22

Reading Code from Top to Bottom: *The Stepdown Rule*

We want the code to read like a top-down narrative.[5] We want every function to be fol- lowed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions. I call this *The Step- down Rule*.

> To say this differently, we want to be able to read the program as though it were a set of *TO* paragraphs, each of which is describing the current level of abstraction and refer- encing subsequent *TO* paragraphs at the next level down.
>
> *To include the setups and teardowns, we include setups, then we include the test page con- tent, and then we include the teardowns.*
>
> *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
>
> *To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.*
>
> *To search the parent. . .*

It turns out to be very difficult for programmers to learn to follow this rule and write functions that stay at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do "one thing." Making the code read like a top-down set of *TO* paragraphs is an effective technique for keeping the abstraction level consistent.

Take a look at Listing 3-7 at the end of this chapter. It shows the whole `testableHtml` function refactored according to the principles described here. Notice how each function introduces the next, and each function remains at a consistent level of abstraction.

Martin p37

Use Descriptive Names

In Listing 3-7 I changed the name of our example function from `testableHtml` to `SetupTeardownIncluder.render`. This is a far better name because it better describes what the function does. I also gave each of the private methods an equally descriptive name such as `isTestable` or `includeSetupAndTeardownPages`. It is hard to overestimate the value of good names. Remember Ward's principle: "*You know you are working on clean code when each routine turns out to be pretty much what you expected.*" Half the battle to achieving that principle is choosing good names for small functions that do one thing. The smaller and more focused a function is, the easier it is to choose a descriptive name.

Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment. Use a naming convention that allows multiple words to be easily read in the function names, and then make use of those multiple words to give the function a name that says what it does.

Don't be afraid to spend time choosing a name. Indeed, you should try several differ- ent names and read the code with each in place. Modern IDEs like Eclipse or IntelliJ make it trivial to change names. Use one of those IDEs and experiment with different names until you find one that is as descriptive as you can make it.

Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.

Be consistent in your names. Use the same phrases, nouns, and verbs in the function names you choose for your modules. Consider, for example, the names `includeSetup-AndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage`, and `includeSetupPage`. The similar phraseology in those names allows the sequence to tell a story. Indeed, if I showed you just the sequence above, you'd ask yourself: "What happened to `includeTeardownPages`, `includeSuiteTeardownPage`, and `includeTeardownPage`?" How's that for being "*. . . pretty much what you expected.*"

Structured Programming

Some programmers follow Edsger Dijkstra's rules of structured programming. Dijkstra said that every function, and every block within a function, should have one entry and one exit. Following these rules means that there should only be one `return` statement in a function, no `break` or `continue` statements in a loop, and never, *ever,* any `goto` statements.

While we are sympathetic to the goals and disciplines of structured programming, those rules serve little benefit when functions are very small. It is only in larger functions that such rules provide significant benefit.

So if you keep your functions small, then the occasional multiple `return`, `break`, or `continue` statement does no harm and can sometimes even be more expressive than the single-entry, single-exit rule. On the other hand, `goto` only makes sense in large functions, so it should be avoided.

Martin p48

Comments Do Not Make Up for Bad Code

One of the more common motivations for writing comments is bad code. We write a module and we know it is confusing and disorganized. We know it's a mess.

So we say to ourselves, "Ooh, I'd better comment that!"

No! You'd better clean it!

Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

Martin p55

Explain Yourself in Code

There are certainly times when code makes a poor vehicle for explanation. Unfortunately, many programmers have taken this to mean that code is seldom, if ever, a good means for explanation. This is patently false. Which would you rather see? This:

```
// Check to see if the employee is eligible for full benefits if
((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))
```

Or this?

```
if (employee.isEligibleForFullBenefits())
```

It takes only a few seconds of thought to explain most of your intent in code. In many cases it's simply a matter of creating a function that says the same thing as the comment you want to write.

Martin p55

Algorithms

Sort algorithm

Search algorithm