SWEN 6301 Software Construction

Module 5: Software Construction

Ahmed Tamrawi

Be the first to **CRACK** this code to claim a **bonus**!

https://gist.github.com/atamrawi/be5f2c23641f00c2cba41f0b6c6e7f62

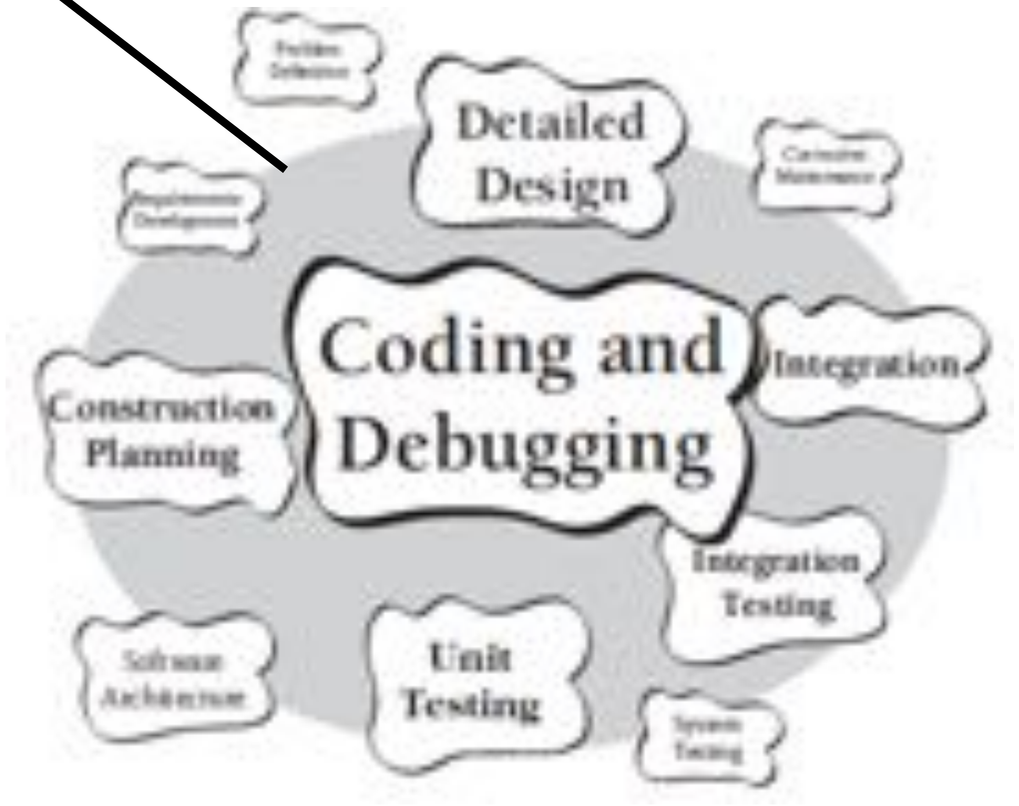
Software Construction

SWEN 6301 Software Construction Definition

Software construction is the process of **creating** and **evolving** software source code that results on *extensible*, *maintainable*, *robust*, and *secure* software

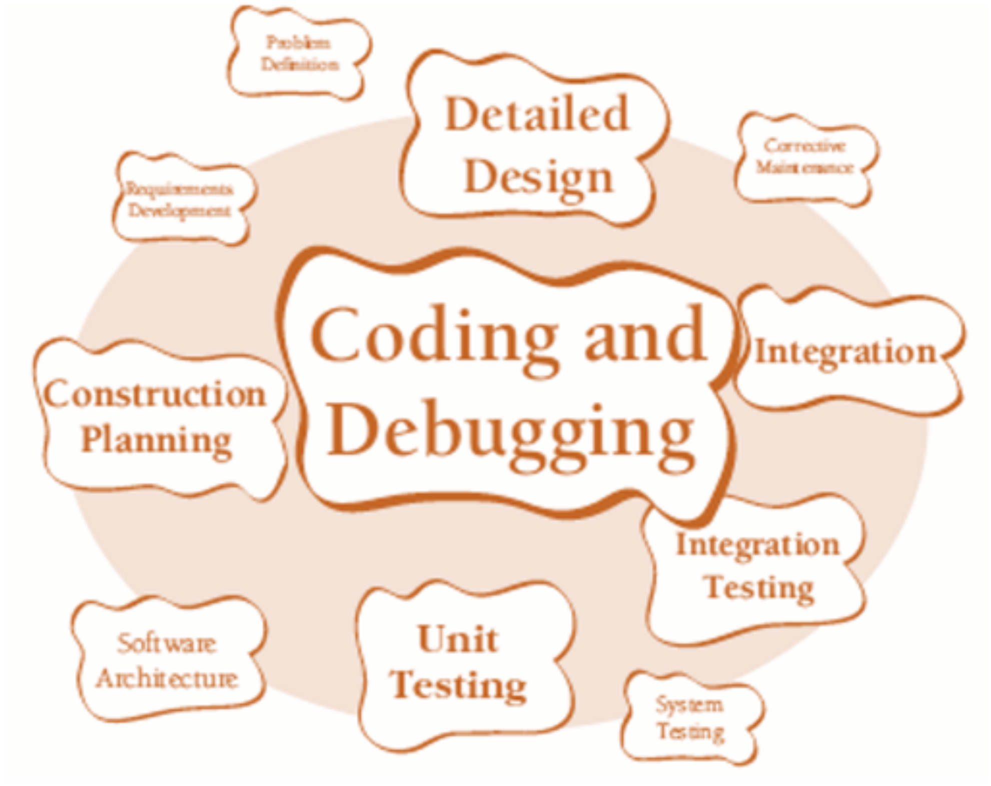
What Is Software Construction?

Construction Activities



Coding implies the mechanical translation of a preexisting design into a computer language; **construction** is not at all mechanical and involves substantial creativity and judgment.

Construction focuses on **coding and debugging** but also includes detailed design, unit testing, integration testing, and other activities.



High-level View of Construction Activities

- Verifying that the groundwork has been laid so that construction can proceed successfully
- Determining how your code will be tested
- Designing and writing classes and routines
- Creating and naming variables and named constants
- Selecting control structures and organizing blocks of statements
- Unit testing, integration testing, and debugging your own code
- Reviewing other team members' low-level designs and code and having them review yours
- Polishing code by carefully formatting and commenting it
- Integrating software components that were created separately
- Tuning code to make it faster and use fewer resources

Specific Tasks of Construction Activities

Why is Software Construction Important?

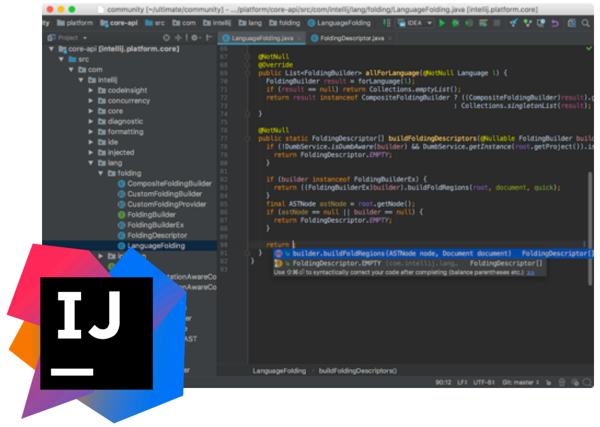
- Construction is a large part of software development
- Construction is the central activity in software development
- With a focus on construction, the individual programmer’s productivity can improve enormously
- Construction’s product, the source code, is often the only accurate description of the software
- Construction is the only activity that’s guaranteed to be done

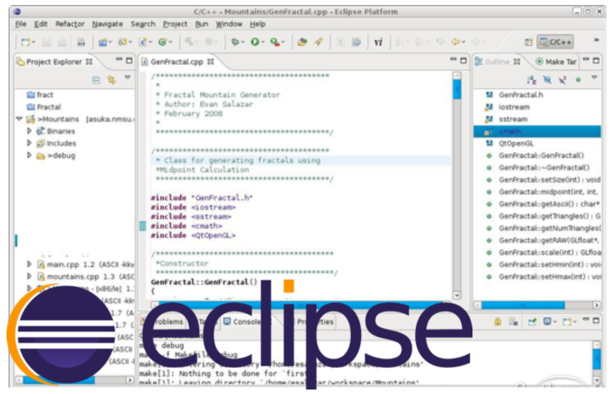
SUMMARY

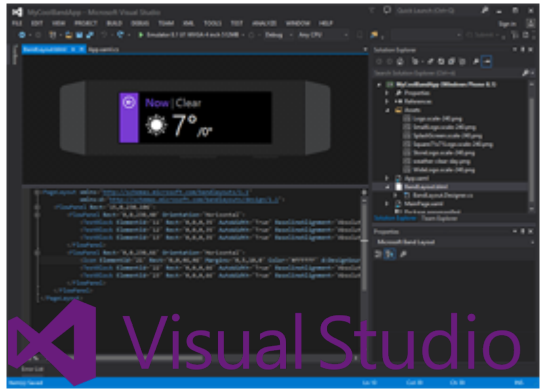
- Software construction is the central activity in software development; construction is the only activity that's guaranteed to happen on every project.
- The main activities in construction are detailed design, coding, debugging, integration, and developer testing (unit testing and integration testing).
- Other common terms for construction are "coding" and "programming."
- The quality of the construction substantially affects the quality of the software.
- Your understanding of how to do construction determines how good a programmer you are.

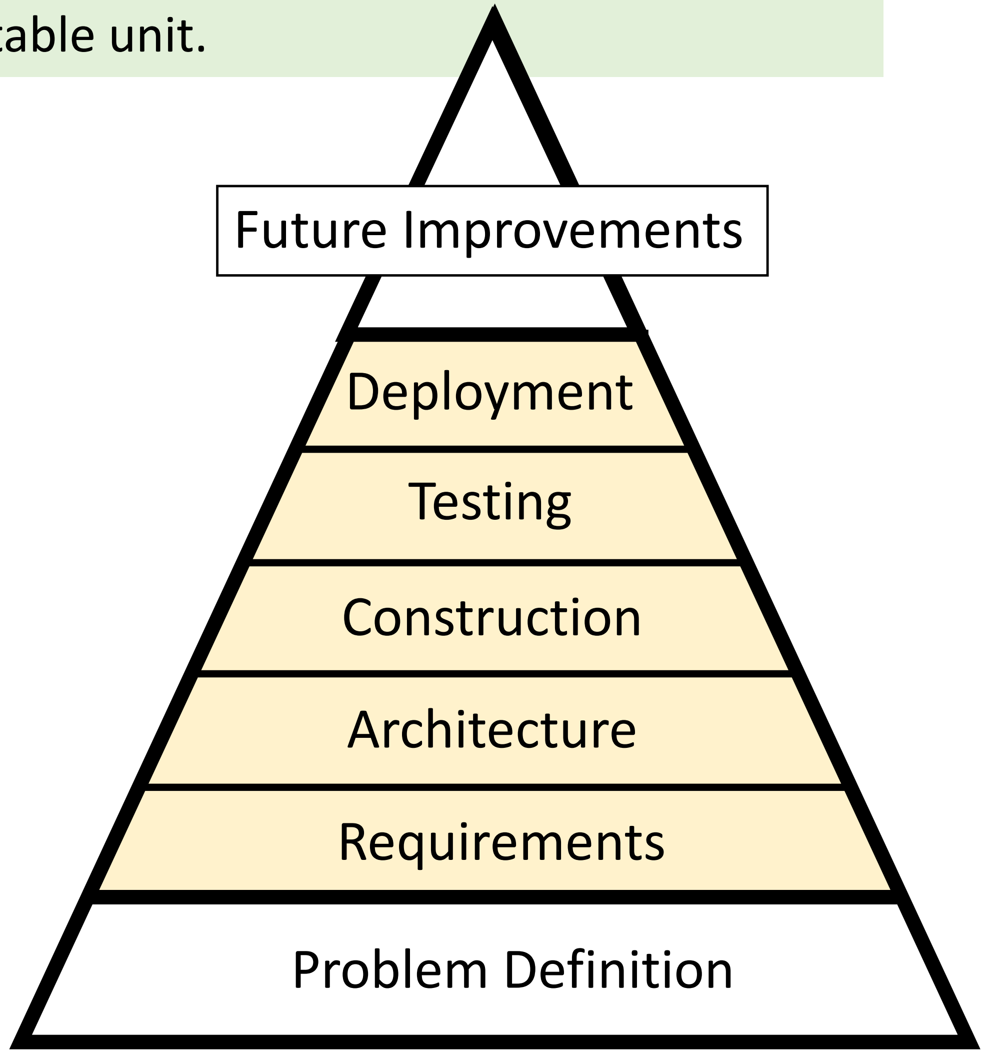
Construction

Software is usually designed and created (coded/written/programmed) in integrated development environments (IDE) like Eclipse, Xcode or Microsoft Visual Studio that can simplify the process and compile the program to an executable unit.









Key Construction Decisions

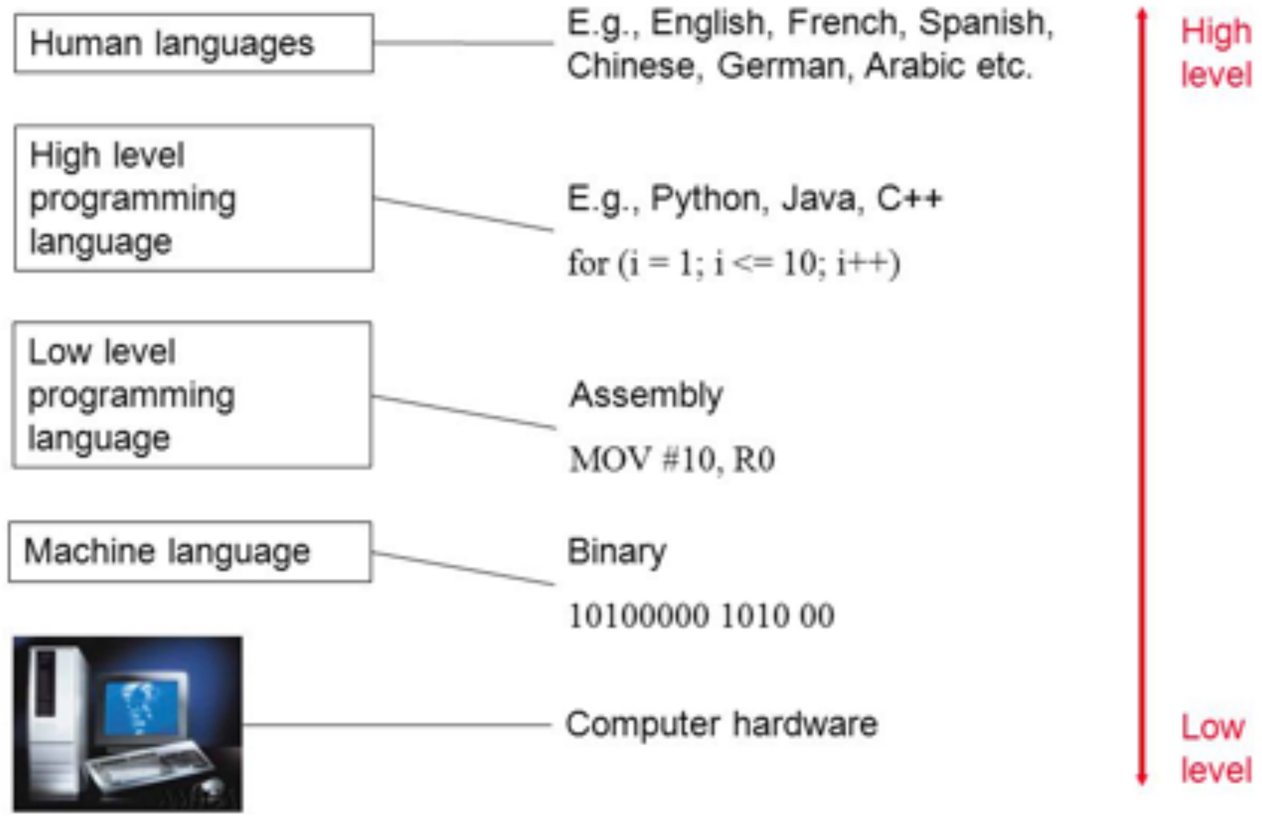
- Choice of Programming Language
- Programming Conventions
- Your Location on the Technology Wave
- Selection of Major Construction Practices

Choice of Programming Language

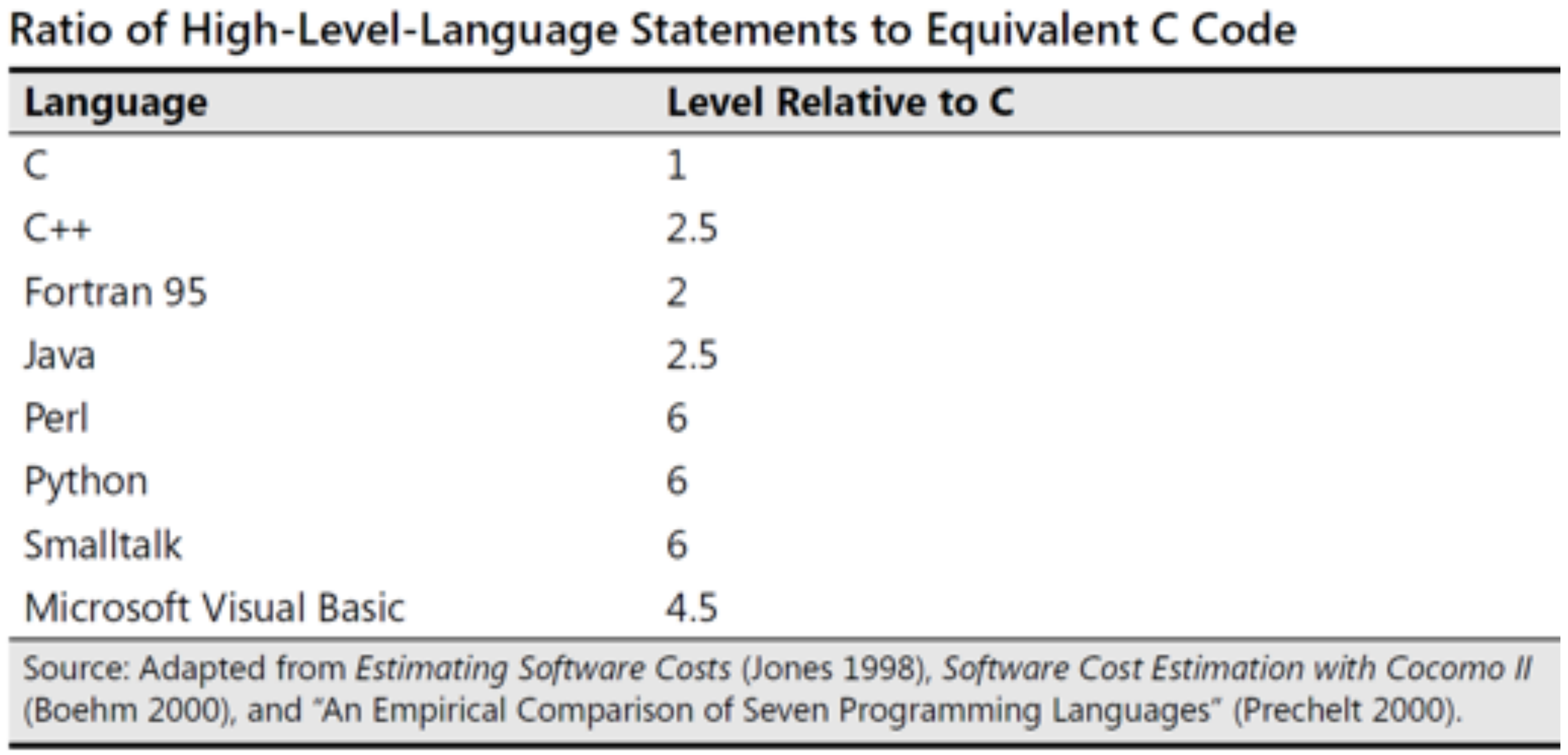
Studies have shown that the programming-language choice affects **productivity** and **code quality** in several ways

Programmers are **more productive** using a familiar language than an unfamiliar one

Programmers working with **high-level languages** achieve better productivity and quality than those working with *lower-level languages*.


Human languages
E.g., English, French, Spanish, Chinese, German, Arabic etc.
High level programming language
E.g., Python, Java, C++
for (i = 1; i <= 10; i++)
Low level programming language
Assembly
MOV #10, R0
Machine language
Binary
10100000 1010 00
Computer hardware
High level
Low level

Choice of Programming Language

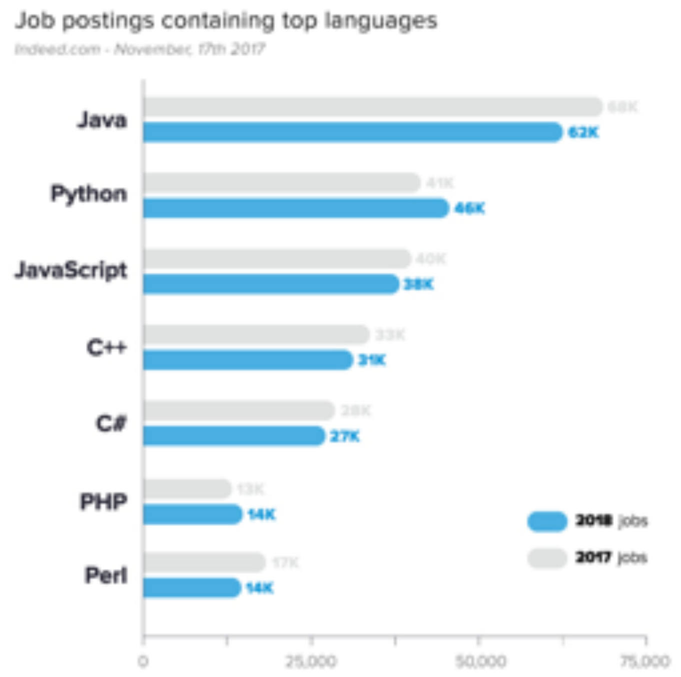


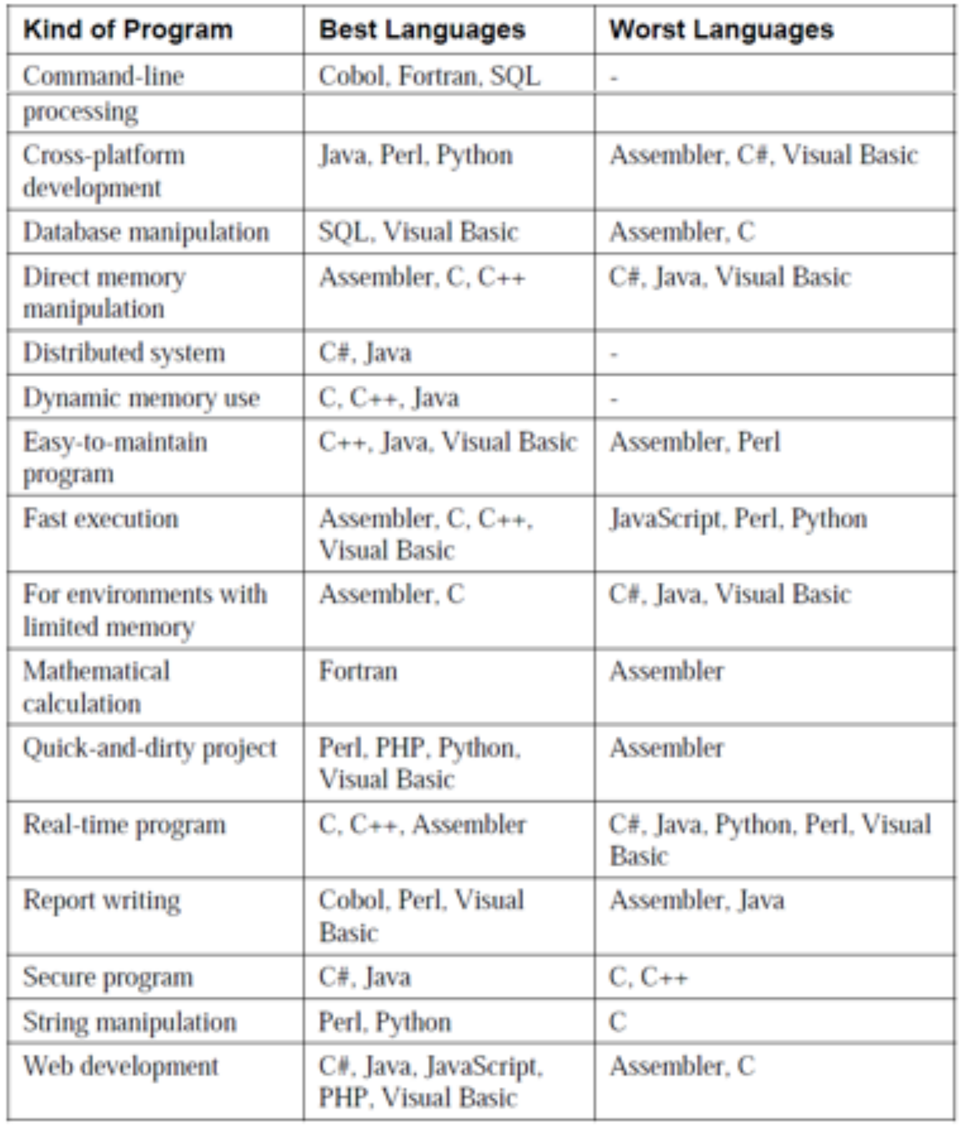
Ratio of High-Level-Language Statements to Equivalent C Code

Language	Level Relative to C
C	1
C++	2.5
Fortran 95	2
Java	2.5
Perl	6
Python	6
Smalltalk	6
Microsoft Visual Basic	4.5

Source: Adapted from *Estimating Software Costs* (Jones 1998), *Software Cost Estimation with Cocomo II* (Boehm 2000), and "An Empirical Comparison of Seven Programming Languages" (Prechelt 2000).

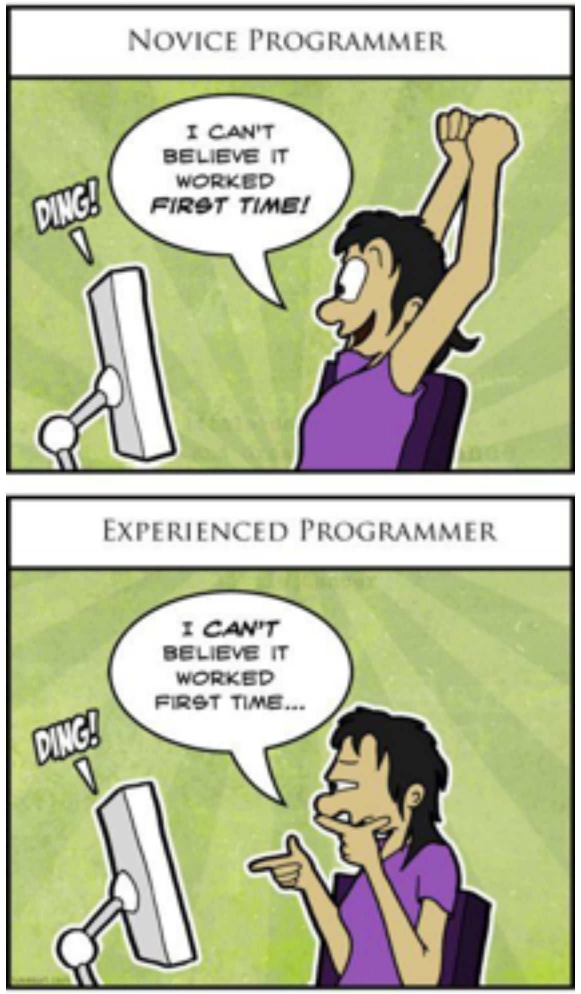
Choice of Programming Language





Kind of Program	Best Languages	Worst Languages
Command-line processing	Cobol, Fortran, SQL	-
Cross-platform development	Java, Perl, Python	Assembler, C#, Visual Basic
Database manipulation	SQL, Visual Basic	Assembler, C
Direct memory manipulation	Assembler, C, C++	C#, Java, Visual Basic
Distributed system	C#, Java	-
Dynamic memory use	C, C++, Java	-
Easy-to-maintain program	C++, Java, Visual Basic	Assembler, Perl
Fast execution	Assembler, C, C++, Visual Basic	JavaScript, Perl, Python
For environments with limited memory	Assembler, C	C#, Java, Visual Basic
Mathematical calculation	Fortran	Assembler
Quick-and-dirty project	Perl, PHP, Python, Visual Basic	Assembler
Real-time program	C, C++, Assembler	C#, Java, Python, Perl, Visual Basic
Report writing	Cobol, Perl, Visual Basic	Assembler, Java
Secure program	C#, Java	C, C++
String manipulation	Perl, Python	C
Web development	C#, Java, JavaScript, PHP, Visual Basic	Assembler, C

Some languages simply don't support certain kinds of programs, and those have not been listed as "worst" languages. For example, Perl is not listed as a "worst language" for mathematical calculations.


NOVICE PROGRAMMER
I CAN'T BELIEVE IT WORKED FIRST TIME!
DING!
EXPERIENCED PROGRAMMER
I CAN'T BELIEVE IT WORKED FIRST TIME...
DING!

Mother Tongues

Tracing the roots of computer languages through the ages

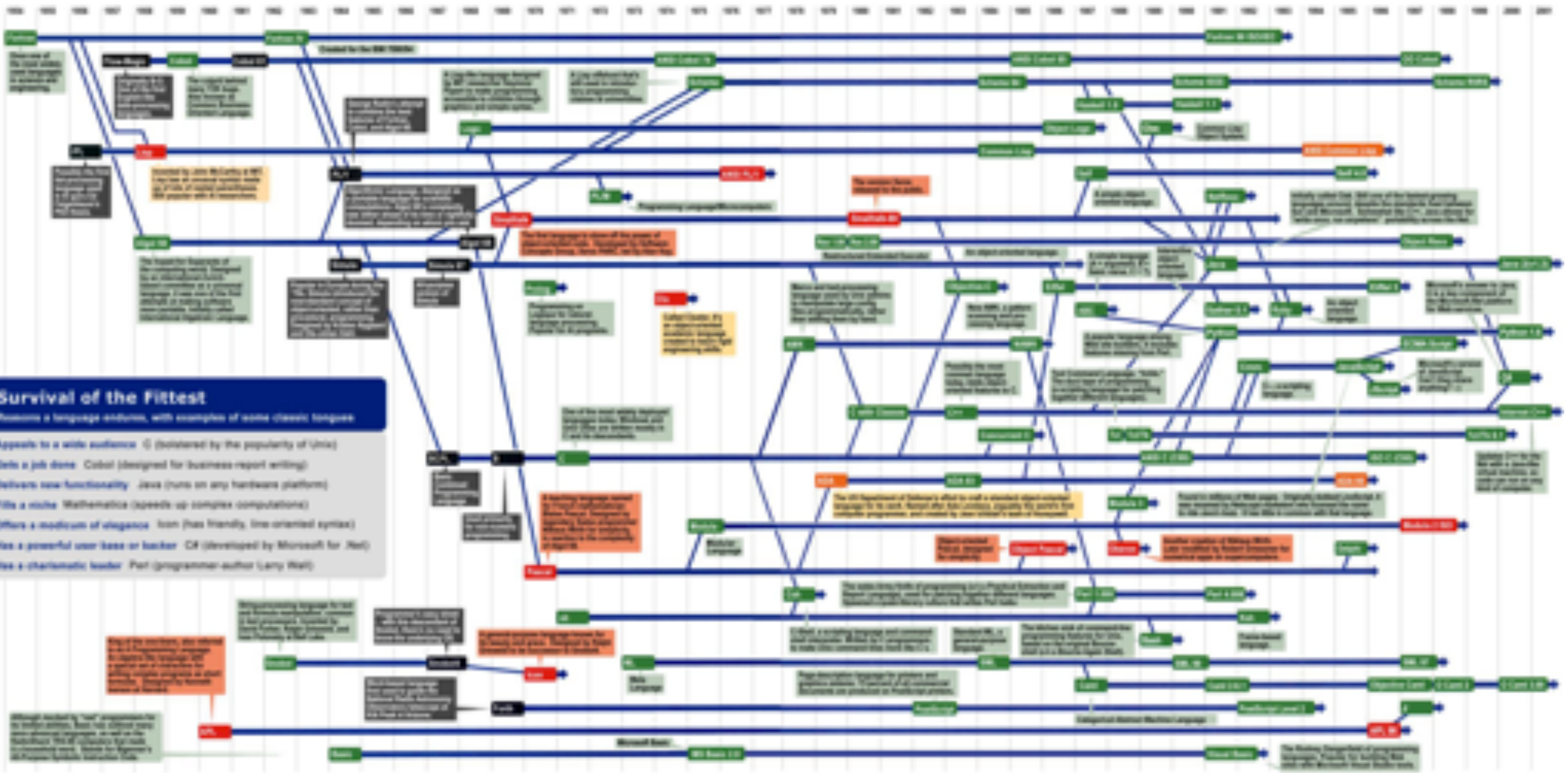
Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at HTTP://www.informatik.uni-freiburg.de/Java/misc/lang_list.html. - Michael Mendeno

Key

1954	**Year Introduced**
🟩	**Active:** thousands of users
🟧	**Protected:** taught at universities; compilers available
🟥	**Endangered:** usage dropping off
⬛	**Extinct:** no known active users or up-to-date compilers
➡	**Lineage continues**



Survival of the Fittest

Reasons a language endures, with examples of some classic tongues

Appeals to a wide audience C (bolstered by the popularity of Unix)

Gets a job done Cobol (designed for business report writing)

Delivers new functionality Java (runs on any hardware platform)

Fills a niche Mathematica (speeds up complex computations)

Offers a modicum of elegance Icon (has friendly, line-oriented syntax)

Has a powerful user base or backer C# (developed by Microsoft for .Net)

Has a charismatic leader Perl (programmer-author Larry Wall)

Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

Programming Conventions

In high-quality software, you can see a relationship between the conceptual integrity of the architecture and its low-level implementation.

That's the point of construction guidelines for variable names, class names, routine names, formatting conventions, and commenting conventions.

Without a unifying discipline, your creation will be a jumble of sloppy variations in style. Such variations tax your brain—and only for the sake of understanding coding-style differences that are essentially arbitrary.

checkstyle

http://checkstyle.sourceforge.net/

eclipse-cs

ECLIPSE-CHECKSTYLE INTEGRATION

http://checkstyle.org/eclipse-cs/

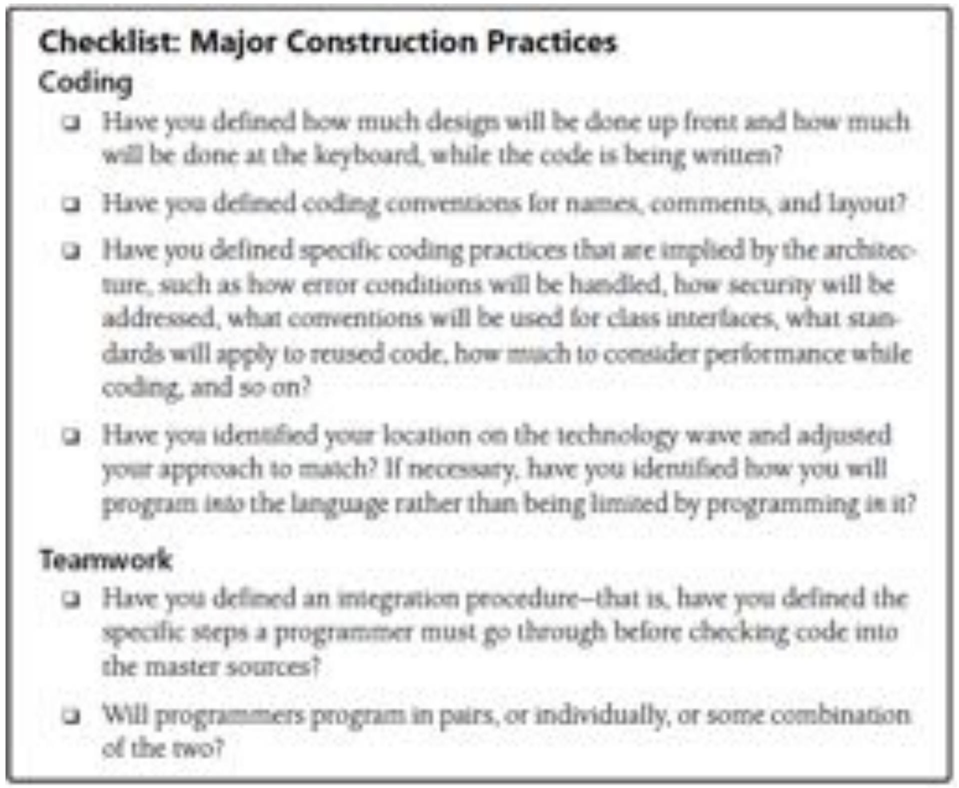
Before construction begins, spell out the programming conventions you'll use. Coding convention details are at such a level of precision that they're nearly impossible to retrofit into software after it's written.

Programming into a Language

Programmers who program **“in”** a language limit their thoughts to constructs that the language directly supports. If the language tools are primitive, the programmer’s thoughts will also be primitive.

Programmers who program **“into”** a language first decide what thoughts they want to express, and then they determine how to express those thoughts using the tools provided by their specific language.

Selection of Major Construction Practices



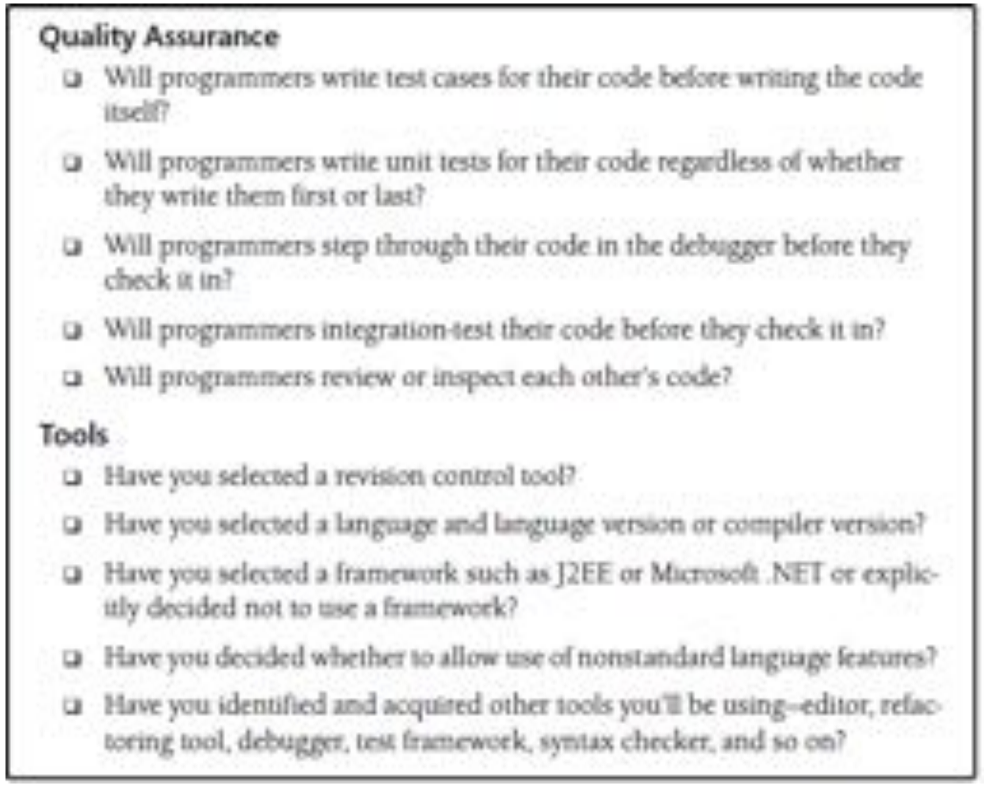
Checklist: Major Construction Practices

Coding

- ❑ Have you defined how much design will be done up front and how much will be done at the keyboard, while the code is being written?
- ❑ Have you defined coding conventions for names, comments, and layout?
- ❑ Have you defined specific coding practices that are implied by the architecture, such as how error conditions will be handled, how security will be addressed, what conventions will be used for class interfaces, what standards will apply to reused code, how much to consider performance while coding, and so on?
- ❑ Have you identified your location on the technology wave and adjusted your approach to match? If necessary, have you identified how you will program into the language rather than being limited by programming in it?

Teamwork

- ❑ Have you defined an integration procedure—that is, have you defined the specific steps a programmer must go through before checking code into the master sources?
- ❑ Will programmers program in pairs, or individually, or some combination of the two?



Quality Assurance

- ❑ Will programmers write test cases for their code before writing the code itself?
- ❑ Will programmers write unit tests for their code regardless of whether they write them first or last?
- ❑ Will programmers step through their code in the debugger before they check it in?
- ❑ Will programmers integration-test their code before they check it in?
- ❑ Will programmers review or inspect each other's code?

Tools

- ❑ Have you selected a revision control tool?
- ❑ Have you selected a language and language version or compiler version?
- ❑ Have you selected a framework such as J2EE or Microsoft .NET or explicitly decided not to use a framework?
- ❑ Have you decided whether to allow use of nonstandard language features?
- ❑ Have you identified and acquired other tools you'll be using—editor, refactoring tool, debugger, test framework, syntax checker, and so on?

SUMMARY

- Every programming language has strengths and weaknesses. Be aware of the specific strengths and weaknesses of the language you’re using.
- Establish programming conventions before you begin programming. It’s nearly impossible to change code to match them later.
- More construction practices exist than you can use on any single project. Consciously choose the practices that are best suited to your project.
- Ask yourself whether the programming practices you’re using are a response to the programming language you’re using or controlled by it. Remember to program into the language, rather than programming in it.
- Your position on the technology wave determines what approaches will be effective— or even possible. Identify where you are on the technology wave, and adjust your plans and expectations accordingly

Software Construction
a very complicated task

Creating High-Quality Code

Design in Construction | Working Classes | High Quality Routines | Defensive Programming

Design in Construction

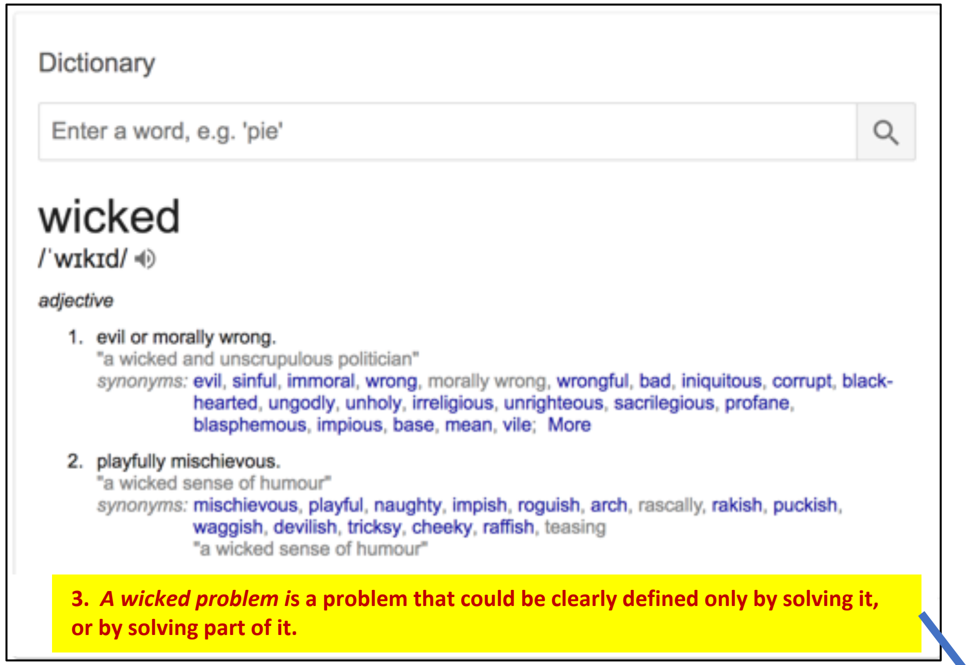
Software Design

The conception, invention, or contrivance of a scheme for turning a specification for computer software into operational software. Its the activity that links requirements to coding and debugging

I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?

A **good** top-level design provides a structure that can safely contain multiple lower-level designs

Design Challenges: *Design is a Wicked Problem*



Dictionary

Enter a word, e.g. 'pie'

wicked

/ˈwɪkɪd/

adjective

1. evil or morally wrong.
   "a wicked and unscrupulous politician"
   synonyms: evil, sinful, immoral, wrong, morally wrong, wrongful, bad, iniquitous, corrupt, black-hearted, ungodly, unholy, irreligious, unrighteous, sacrilegious, profane, blasphemous, impious, base, mean, vile; More
2. playfully mischievous.
   "a wicked sense of humour"
   synonyms: mischievous, playful, naughty, impish, roguish, arch, rascally, rakish, puckish, waggish, devilish, tricksy, cheeky, raffish, teasing
   "a wicked sense of humour"

3. *A wicked problem* is a problem that could be clearly defined only by solving it, or by solving part of it.

Horst Rittel and Melvin Webber 1973

WICKED PROBLEMS
MALCOLM BROWN ELI
Deliberately un-deliberate
a meditation on methods
EDUCAUSE LEARNING INITIATIVE
500 years
Diverge Converge
DESIGN THINKING
MOOC
CORE
unlike others
complex ambiguous
Need something really new
WICKED PROBLEMS
Can't write a check
Require disruptive
definition itself is wicked problem
2. IDEO
Evolutionary 10%
70%
10%
Part of extended curriculum
Disruption helps INSIDE
CONCLUSIONS for WICKED PROBLEMS
1. EXERCISE
Draw a sound 30 sec.
Alex Osterwalder IDEO
John Cleese Creativity
"Not a talent way of operating"
why all SAME?
OUT OF BOX requires OUT OF BOX ENVIRONMENTS
In constraint - we do what we know.
EFFICIENCY → In Box Ideas
CLOSED MODE
Time bound
Focused
OPEN MODE
Relaxed
Curious
* Need CREATIVE THINKING "free & fragile"
* LEADERSHIP is KEY to ENCOURAGING CREATIVITY
* CREATIVE THINKING → LEARNING which improves CORE.

The paradox implies that you have to solve the problem once in order to clearly define it and then solve it again to create a solution that works.

Design Challenges: *Design Is a Wicked Problem*

The event is presented as an example of elementary forced resonance, with the wind providing an external periodic frequency that matched the natural structural frequency, ***even though the real cause*** *of the bridge's failure was* ***aeroelastic flutter****, not resonance. A contributing factor was its solid sides, not allowing wind to pass through the bridge's deck. Thus, its design allowed the bridge to catch the wind and sway, which ultimately took it down.*

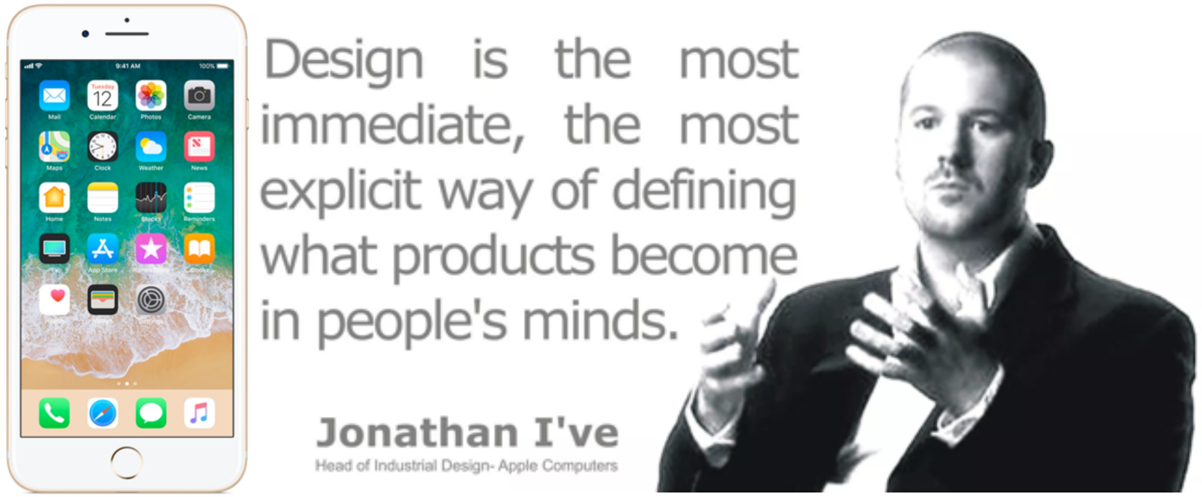
The Tacoma Narrows bridge—an example of a wicked problem

Until the bridge collapsed, its engineers didn't know that aerodynamics needed to be considered to such an extent.

Only by building the bridge (solving the problem) could they learn about the additional consideration in the problem that allowed them to build another bridge that still stands.

Design Challenges: *Design Is a Sloppy Process*

The finished software design should look well organized and clean, but the process used to develop the design isn't nearly as tidy as the end result.


Design is the most immediate, the most explicit way of defining what products become in people's minds.

Jonathan I've

Head of Industrial Design- Apple Computers

Design Challenges: *Design Is About Tradeoffs and Priorities*

A key part of the designer's job is to **weigh** competing design characteristics and strike a **balance** among those characteristics

Design Challenges: *Design Involves Restrictions*

The point of design is partly to create possibilities and partly to restrict possibilities

The constraints of limited resources for constructing buildings force simplifications of the solution that ultimately improve the solution.

Design Challenges: *Design Is Nondeterministic*

Design Challenges: *Design Is a Heuristic Process*

*Because design is nondeterministic, design techniques tend to be **heuristics**—"rules of thumb" or "things to try that sometimes work"—rather than repeatable processes that are guaranteed to produce predictable results*

Design Challenges: *Design Is Emergent*

DESIGN IS
MERGENT
Designs don't spring fully formed directly from someone's brain. They **evolve and improve** through design reviews, informal discussions, experience writing the code itself, and experience revising the code.

Key Design Concepts

Managing Complexity

Desirable Characteristics

Levels of Design

Managing Complexity

Accidental and Essential Difficulties

Software development is made difficult because of two different classes of problems: the **essential** and the **accidental**

Fred Brooks's landmark paper, "No Silver Bullets: Essence and Accidents of Software Engineering" (1987).

The properties that a thing must have in order to be that thing

The properties a thing happens to have and don't really bear on whether the thing is what it is

Managing Complexity

Importance of Managing Complexity

The only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, a ratio of 1 to 10^9 , or nine orders of magnitude *(Dijkstra 1989)*

No one's skull is really big enough to contain a modern computer program *(Dijkstra 1972)*

The goal is to **minimize** the amount of a program you have to think about at any one time.

Dividing the system into subsystems

Break a complicated problem into simple pieces

More independent the subsystems

Keeping routines short

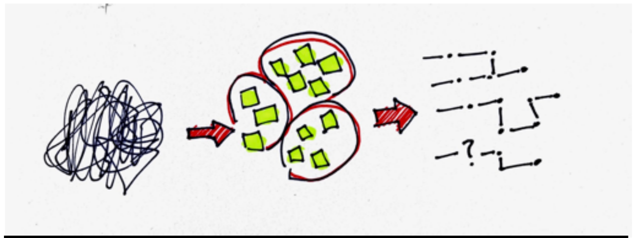
Managing Complexity

How to Attack Complexity

Minimize the amount of **essential** complexity that anyone's brain has to deal with at any one time

Keep **accidental** complexity from needlessly proliferating

Desirable Characteristics of a Design

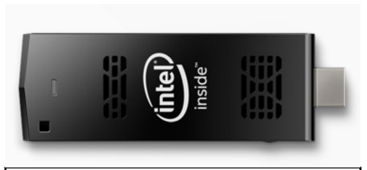


Minimal Complexity

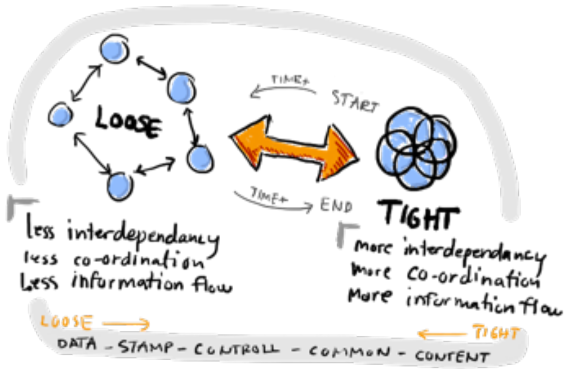
Ease of Maintenance

Extensibility

Reusability

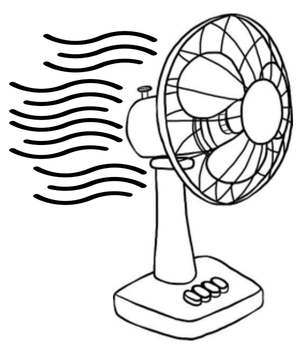


Portability

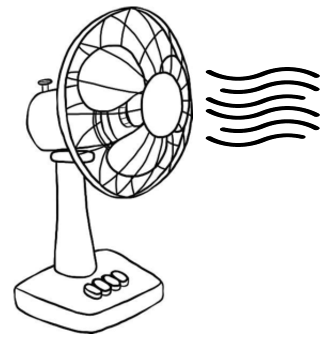


Loose Coupling

Leanness



High Fan-In



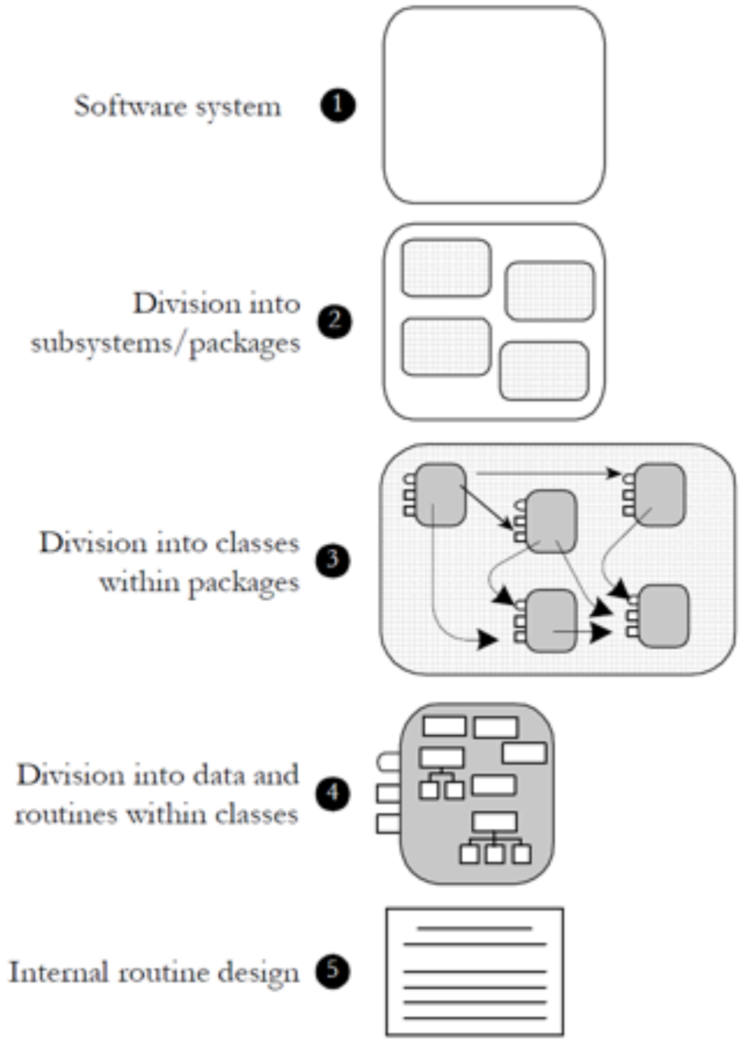
Low-to-Medium Fan-Out

Stratification

Standard Techniques

Levels of Design

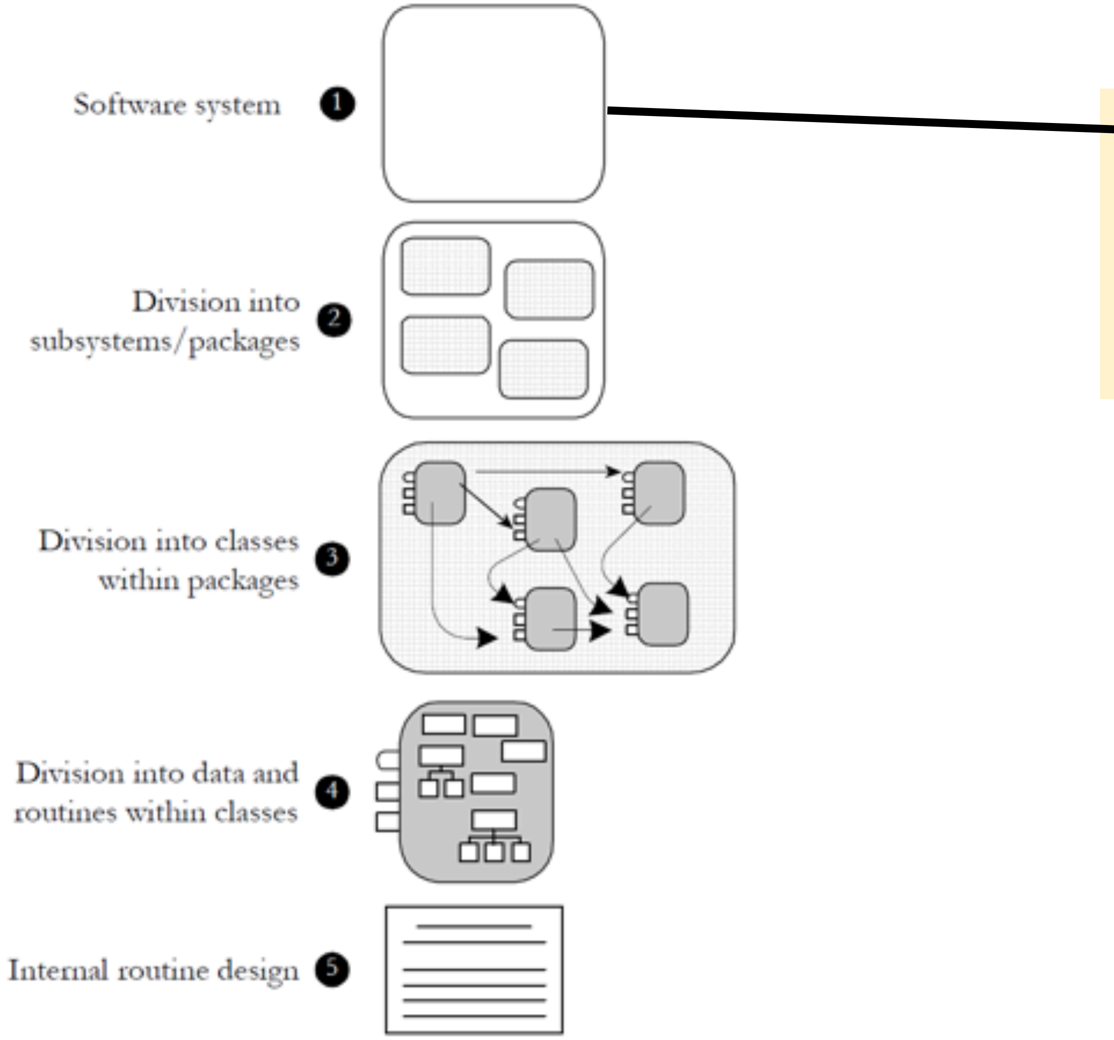
Design is **needed** at several different levels of detail in a software system. Some design techniques apply at all levels, and some apply at only one or two.



Software system 1
Division into subsystems/packages 2
Division into classes within packages 3
Division into data and routines within classes 4
Internal routine design 5

The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

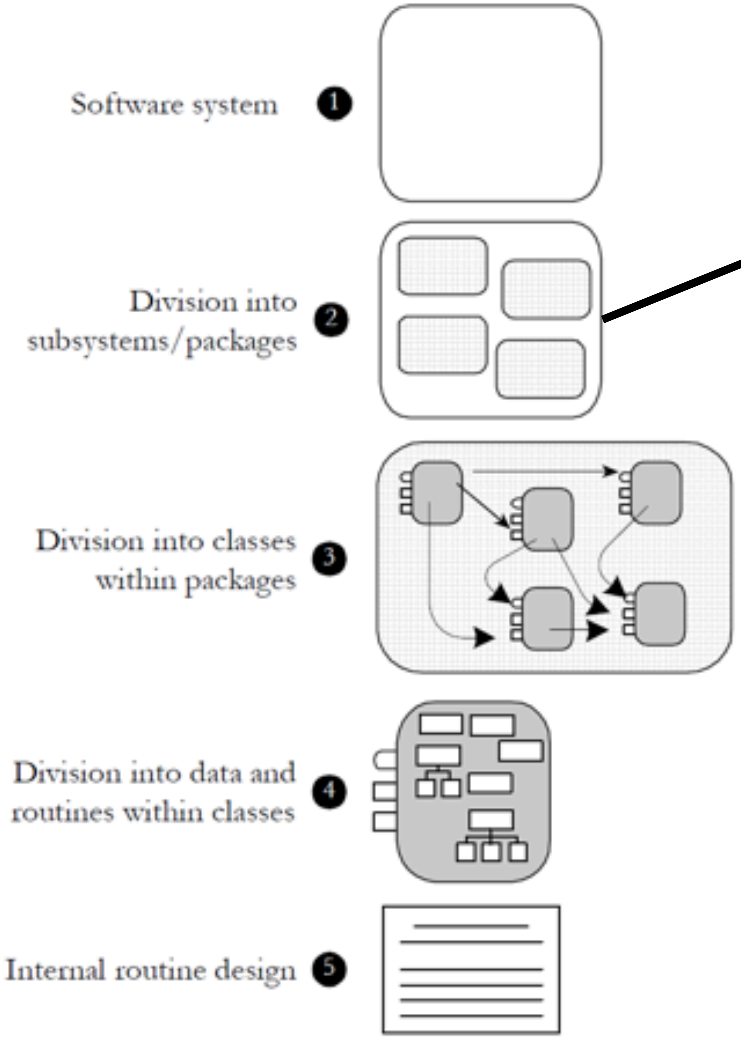
Levels of Design


Software system
1
Division into subsystems/packages
2
Division into classes within packages
3
Division into data and routines within classes
4
Internal routine design
5

The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

❶ **Software System**
The first level is the entire system. Some programmers jump right from the system level into designing classes, but it's usually beneficial to think through higher level combinations of classes, such as subsystems or packages.

Levels of Design



Software system 1
Division into subsystems/packages 2
Division into classes within packages 3
Division into data and routines within classes 4
Internal routine design 5

The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

❷ Division into Subsystems or Packages
The major design activity at this level is deciding how to partition the program into major subsystems and defining how each subsystem is allowed to use each other subsystem.

Within each subsystem, different methods of design might be used—choosing the approach that best fits each part of the system.

Common Subsystems

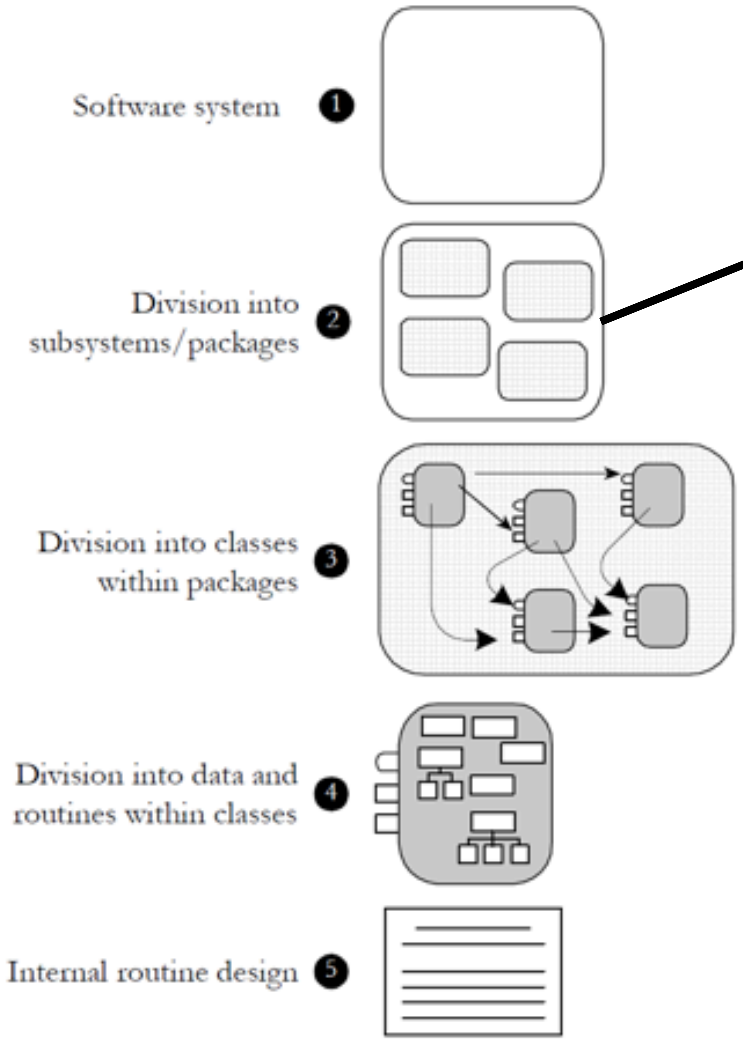
User interface
May use several subordinate subsystems or classes for the GUI interface, command line interface, menu operations, window management, help system, and so forth

Business Rules
the laws, regulations, policies, and procedures that you encode into a computer system

Database Access
centralize database operations in one place and reduce the chance of errors in working with the data.

System dependencies
If you're developing a program for Microsoft Windows, why limit yourself to the Windows environment? Isolate the Windows calls in a Windows-interface subsystem. If you later want to move your program to Mac OS or Linux, all you'll have to change is the interface subsystem

Levels of Design



Software system ❶
Division into subsystems/packages ❷
Division into classes within packages ❸
Division into data and routines within classes ❹
Internal routine design ❺

The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

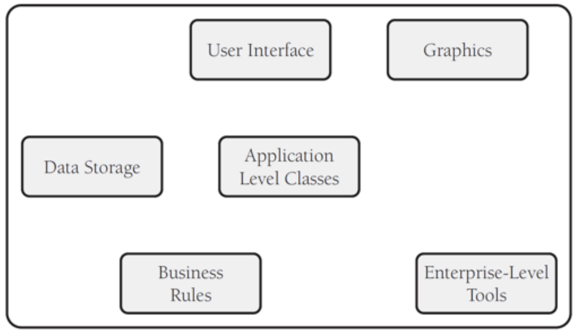
❷ Division into Subsystems or Packages
The major design activity at this level is deciding how to partition the program into major subsystems and defining how each subsystem is allowed to use each other subsystem.

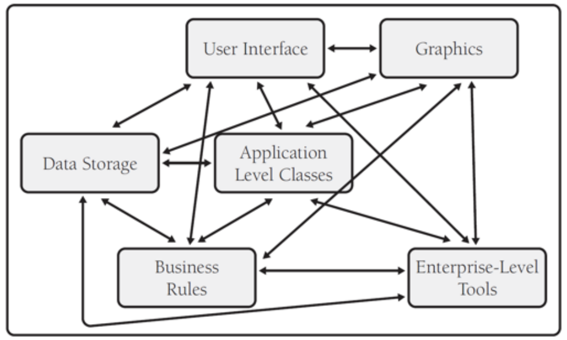
Within each subsystem, different methods of design might be used—choosing the approach that best fits each part of the system.

IMPORTANT RULE *How the various subsystems can communicate?*

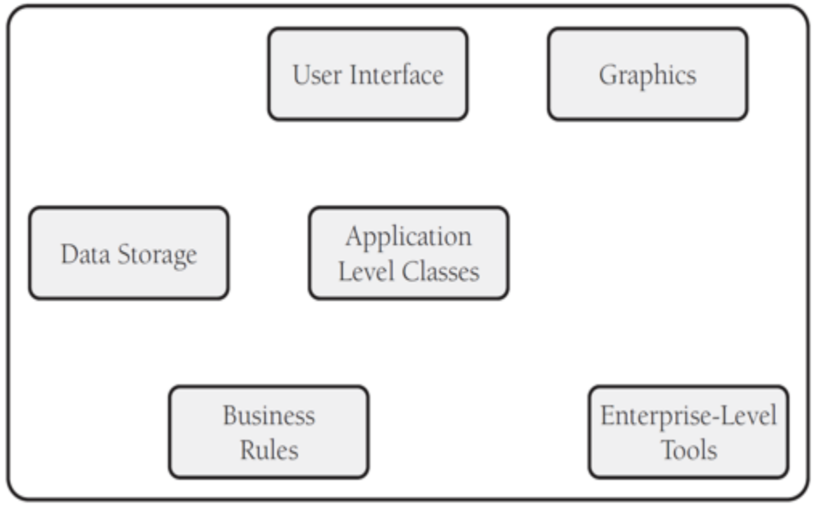
If all subsystems can communicate with all other subsystems, **you lose the benefit of separating them at all**.

Make each subsystem meaningful by restricting communications.

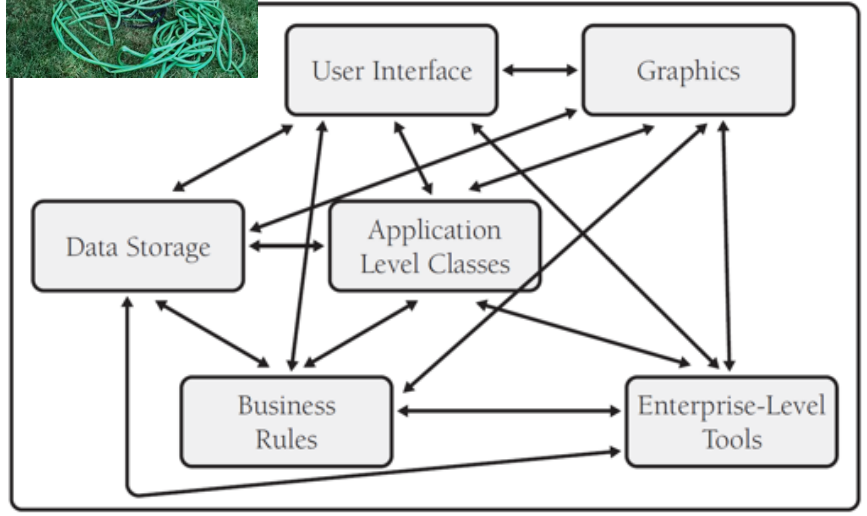




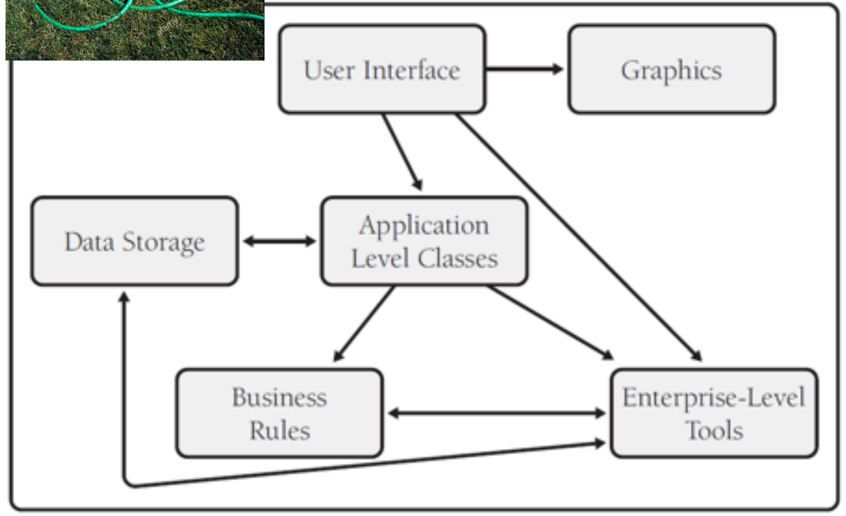
Levels of Design



Suppose for example that you define a system with six subsystems

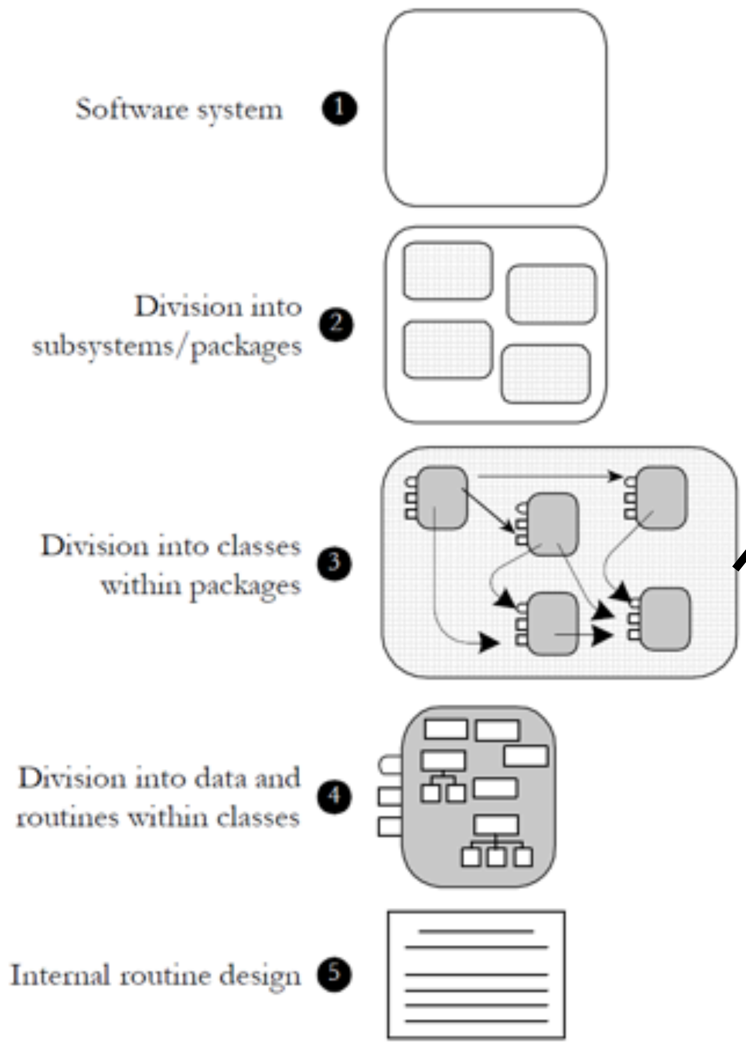


- How many different parts of the system does a developer need to understand at least a little bit to change something in the graphics subsystem?
- What happens when you try to use the business rules in another system?
- What happens when you want to put a new user interface on the system, perhaps a command-line UI for test purposes?
- What happens when you want to put data storage on a remote machine?



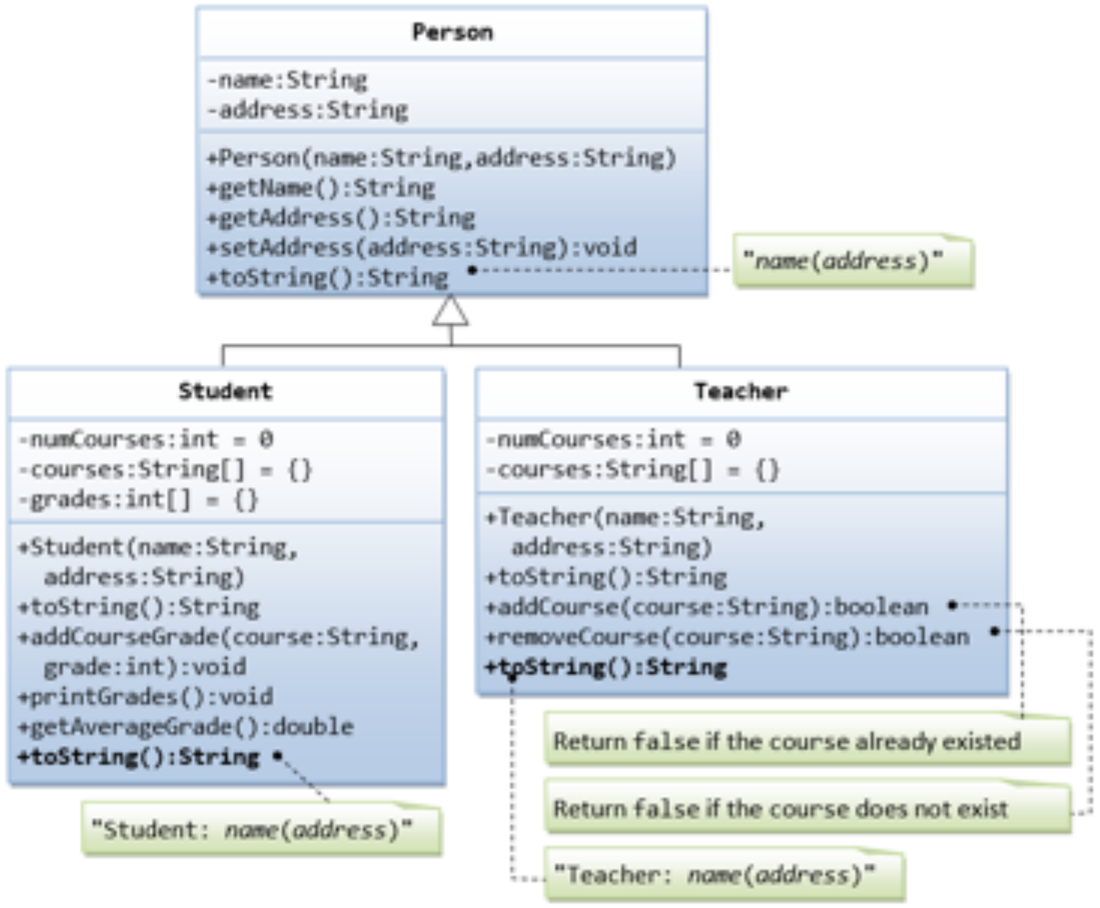
- Allow communication between subsystems only on a "**need to know**" basis—and it had better be a good reason.
- If in doubt, **it's easier to restrict communication** early and relax it later than it is to relax it early and then try to tighten it up after you've coded several hundred inter-subsystem calls.
- The **simplest** relationship is to have one subsystem call routines in another.
- A **more involved** relationship is to have one subsystem contain classes from another.
- The **most involved** relationship is to have classes in one subsystem inherit from classes in another

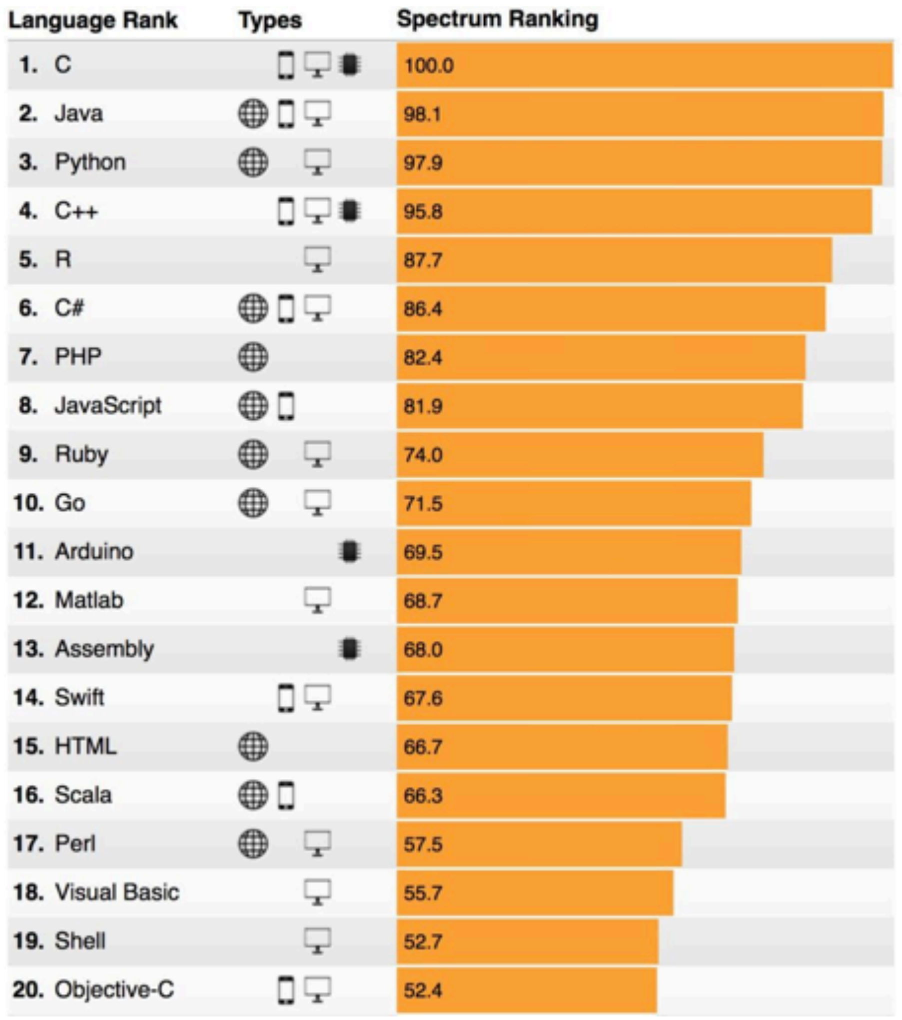
Levels of Design


Software system ❶
Division into subsystems/packages ❷
Division into classes within packages ❸
Division into data and routines within classes ❹
Internal routine design ❺

The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

❸ Division into Classes within Packages
Design at this level includes identifying all classes in the system.


Person
-name:String
-address:String
+Person(name:String,address:String)
+getName():String
+getAddress():String
+setAddress(address:String):void
+toString():String
"name(address)"
Student
-numCourses:int = 0
-courses:String[] = {}
-grades:int[] = {}
+Student(name:String, address:String)
+toString():String
+addCourseGrade(course:String, grade:int):void
+printGrades():void
+getAverageGrade():double
+toString():String
"Student: name(address)"
Teacher
-numCourses:int = 0
-courses:String[] = {}
+Teacher(name:String, address:String)
+toString():String
+addCourse(course:String):boolean
+removeCourse(course:String):boolean
+toString():String
Return false if the course already existed
Return false if the course does not exist
"Teacher: name(address)"

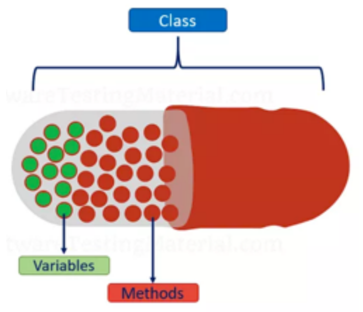


Language Rank	Types	Spectrum Ranking
1. C	📱 🖥 ▪	100.0
2. Java	🌐 📱 🖥	98.1
3. Python	🌐 🖥	97.9
4. C++	📱 🖥 ▪	95.8
5. R	🖥	87.7
6. C#	🌐 📱 🖥	86.4
7. PHP	🌐	82.4
8. JavaScript	🌐 📱	81.9
9. Ruby	🌐 🖥	74.0
10. Go	🌐 🖥	71.5
11. Arduino	▪	69.5
12. Matlab	🖥	68.7
13. Assembly	▪	68.0
14. Swift	📱 🖥	67.6
15. HTML	🌐	66.7
16. Scala	🌐 📱	66.3
17. Perl	🌐 🖥	57.5
18. Visual Basic	🖥	55.7
19. Shell	🖥	52.7
20. Objective-C	📱 🖥	52.4

https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016

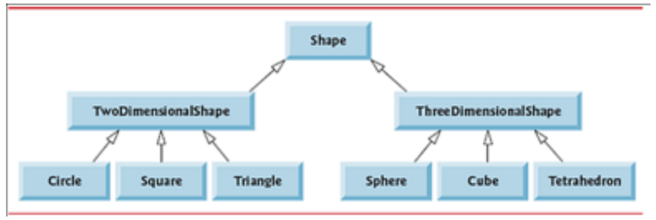
These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Encapsulation



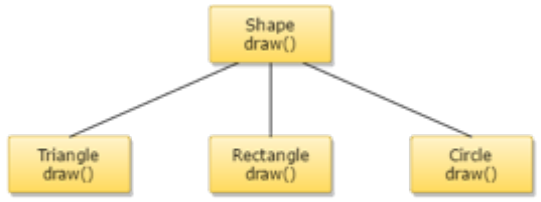
Objects

Inheritance

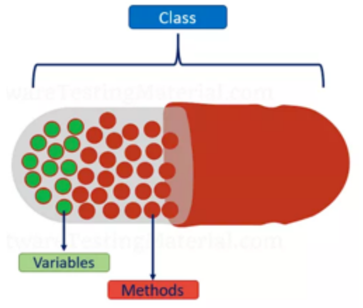


OOP Key Technologies

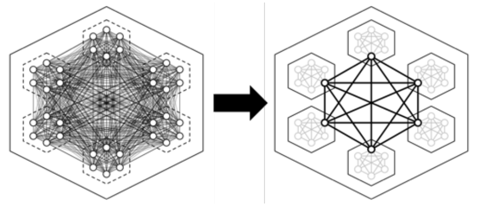
Polymorphism



Classes



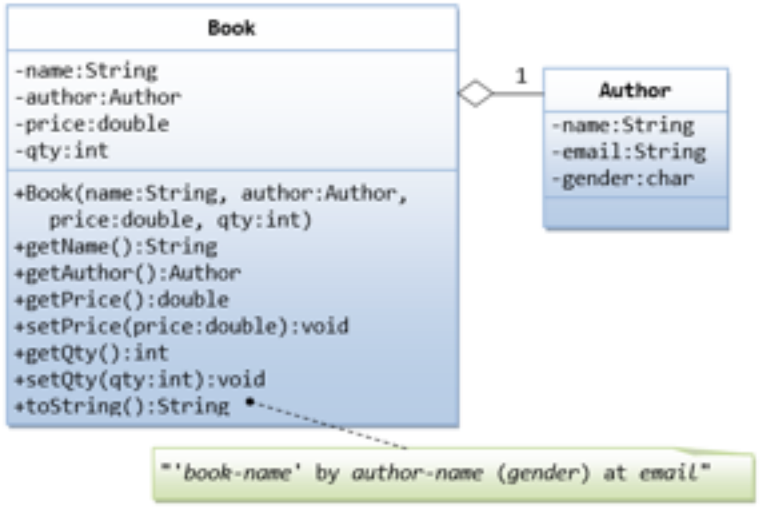
Abstraction



These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

A **class** is the *static* thing you look at in the program listing while an **object** (*instantiation of a class*) is any specific entity that exists in your program at run time.

Book Class



Instances of "Book" Class (Objects)

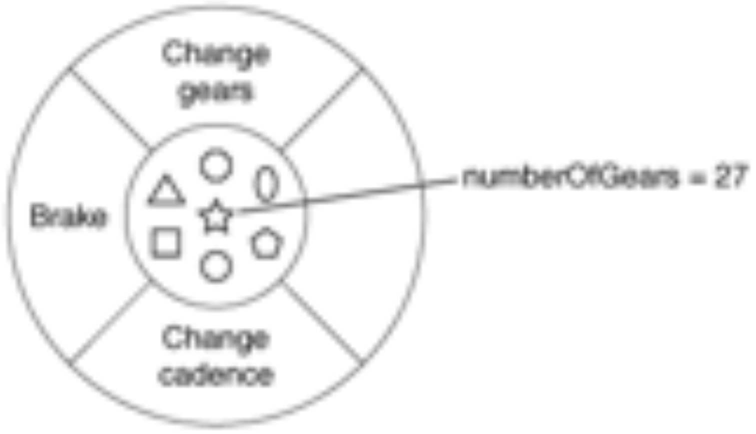
These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Real-world objects share two characteristics

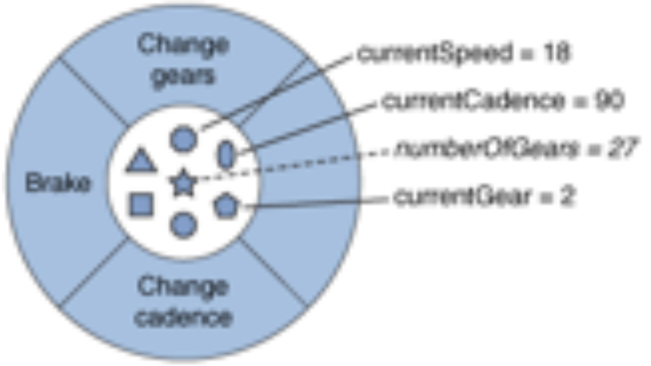
State *attribute/field*

Behavior *method/function*

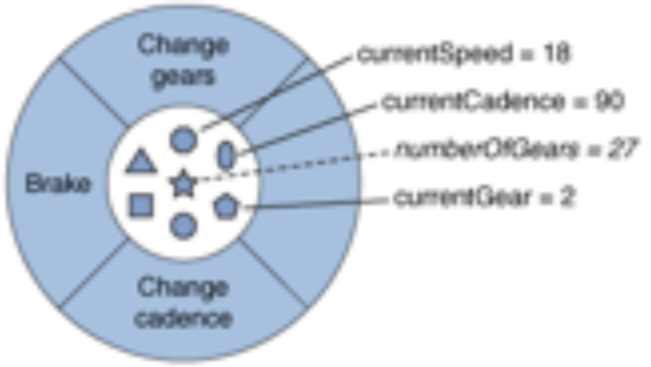
```java
class Bike {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        this.cadence = newValue;
    }

    void changeGear(int newValue) {
        this.gear = newValue;
    }

    void speedUp(int increment) {
        this.speed += increment;
    }

    void applyBrakes(int decrement) {
        this.speed -= decrement;
    }

    void printStates() {
        System.out.println(
            "cadence: " + this.cadence
          + " speed: " + this.speed
          + " gear: " + this.gear
        );
    }
}
```



Bike Class



My Bike



Foo Bike

These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

```java
class Bike {
   int cadence = 0;
   int speed = 0;
   int gear = 1;

   void changeCadence(int newValue) {
      this.cadence = newValue;
   }

   void changeGear(int newValue) {
      this.gear = newValue;
   }

   void speedUp(int increment) {
      this.speed += increment;
   }

   void applyBrakes(int decrement) {
      this.speed -= decrement;
   }

   void printStates() {
      System.out.println(
           "cadence: " + this.cadence
         + " speed: " + this.speed
         + " gear: " + this.gear
      );
   }
}
```

Bike.java

```java
class BikeDemo {

   public static void main(String[] args) {

      // create two different Bike Objects
      Bike bike1 = new Bike();
      Bike bike2 = new Bike();

      // perform operations on bike1
      bike1.changeCadence(50);
      bike1.speedUp(10);
      bike1.changeGear(2);
      bike1.printStates();

      // perform operations on bike2
      bike2.changeCadence(50);
      bike2.speedUp(10);
      bike2.changeGear(2);
      bike2.changeCadence(40);
      bike2.speedUp(10);
      bike2.changeGear(3);
      bike2.printStates();
   }
}
```

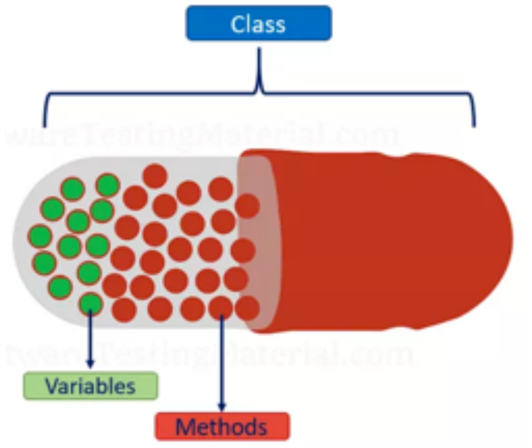
BikeDemo.java

What is the output?

These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

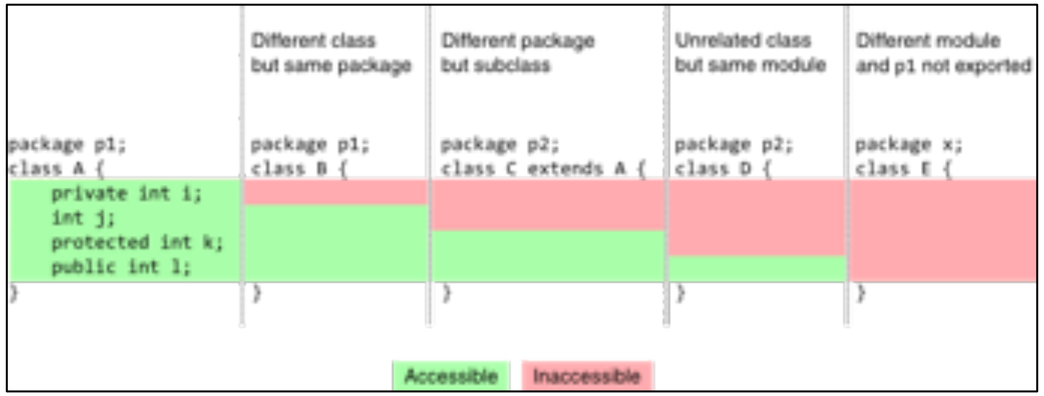
Encapsulation

The process of wrapping code and data together into a single unit



Data/Information Hiding

The variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class



```
package p1;
class A {
    private int i;
    int j;
    protected int k;
    public int l;
}
```

Different class but same package

```
package p1;
class B {
}
```

Different package but subclass

```
package p2;
class C extends A {
}
```

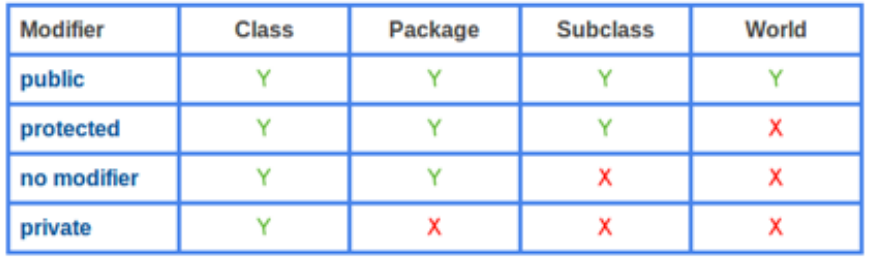
Unrelated class but same module

```
package p2;
class D {
}
```

Different module and p1 not exported

```
package x;
class E {
}
```

🟩 Accessible 🟥 Inaccessible



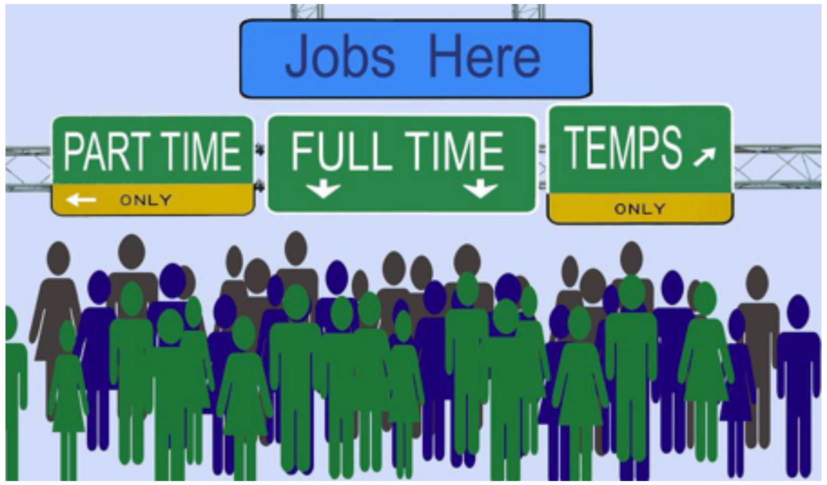
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	X
no modifier	Y	Y	X	X
private	Y	X	X	X

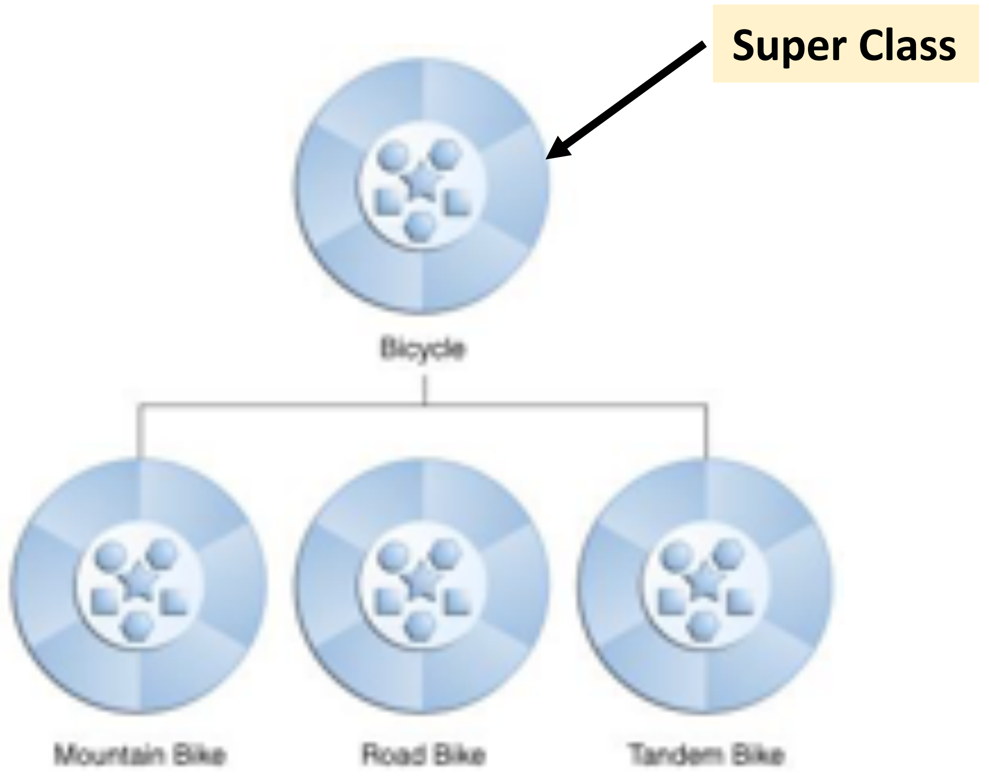
These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Inheritance

Different kinds of objects often have a certain amount in common with each other

Object-oriented programming allows classes to **inherit commonly used state and behavior** from other classes and let you focus on the features that make a specific class unique





```
class MountainBike extends Bicycle {
    // new fields/methods specific to
    // Mountain bike go here
}
```

```
class TandemBike extends Bicycle {
    // new fields/methods specific to
    // Tandem bike go here
}
```

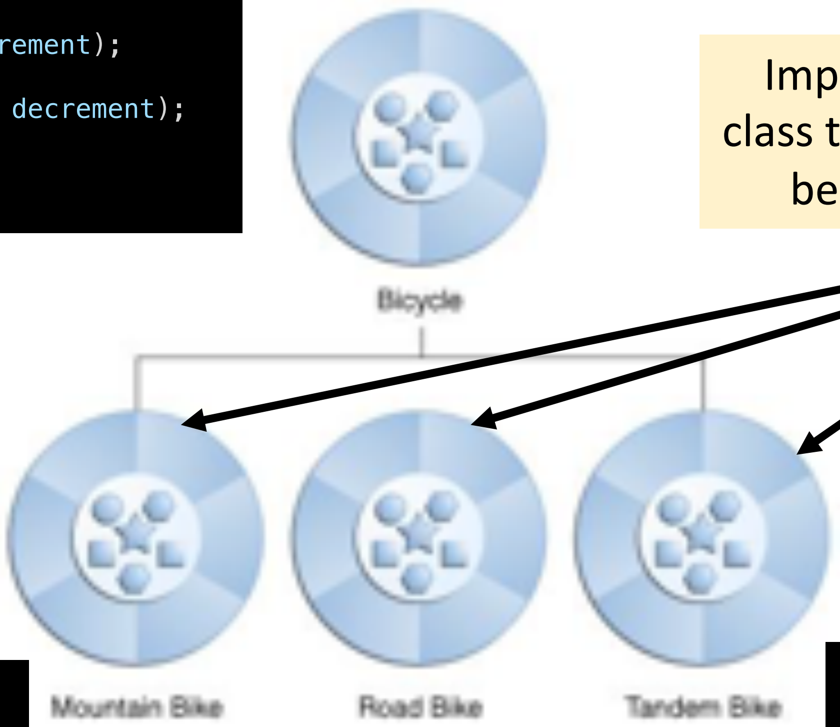
```
class RoadBike extends Bicycle {
    // new fields/methods specific to
    // Road bike go here
}
```

These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Interfaces

Define class instances interaction with the outside world through the methods that they expose

```
interface Bicycle {

   void changeCadence(int newValue);

   void changeGear(int newValue);

   void speedUp(int increment);

   void applyBrakes(int decrement);

   void printStates();
}
```



Implementing an interface allows a class to become more formal about the behavior it promises to provide.

Interfaces **form a contract between the class and the outside world**, and this contract is **enforced at build time by the compiler**.

If a class claims to implement an interface, all methods defined by that interface must appear in its source code.

```
class MountainBike implements Bicycle {
   ...
}
```

```
class RoadBike implements Bicycle {
 ...
}
```

```
class TandemBike implements Bicycle {
 ...
}
```

LET'S RECAP...
Object
Oriented
Programming

Interfaces

Define class instances interaction with the outside world through the methods that they expose

Signing into Facebook and not talking to anyone
Just stare at peoples statuses like
something is wrong with these people

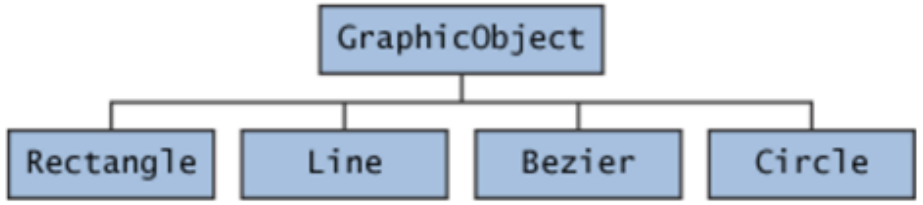
```
public interface SomethingIsWrong {

   void foo(int value) {
      System.out.println("Something is wrong!");
   }
}
```

These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Abstract Classes/Methods

*An **abstract class** is a class that is declared abstract and cannot be instantiated but can be sub-classed. An **abstract method** is a method that is declared without an implementation*

```java
abstract class GraphicObject {
   int x, y;

   void moveTo(int newX, int newY) {
      // Some code here
   }

   abstract void draw();
   abstract void resize();
}
```



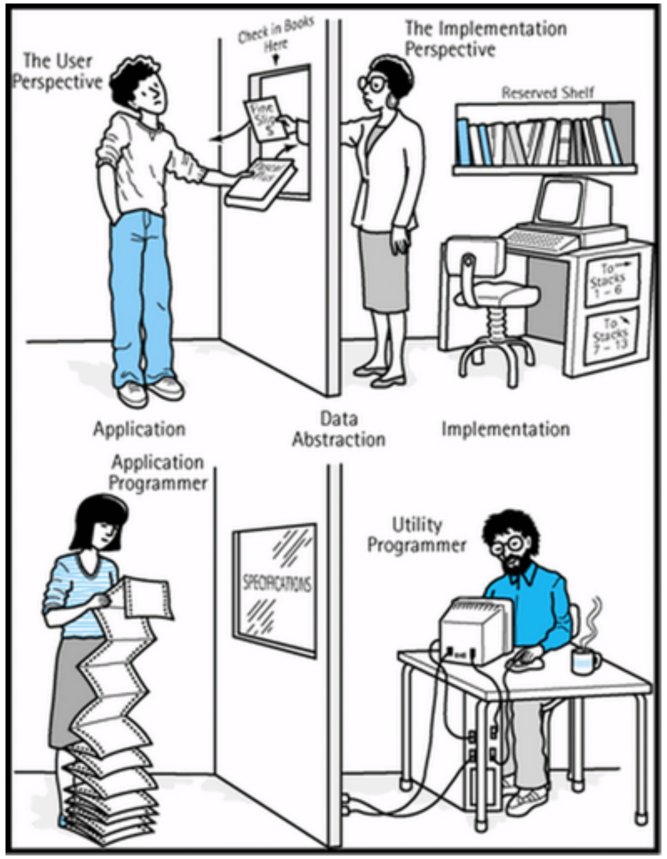
```java
class Rectangle extends GraphicObject {

   void draw() {
      // some implementation here
   }

   void resize() {
      // some implementation here
   }
}
```

```java
class Circle extends GraphicObject {

   void draw() {
      // some implementation here
   }

   void resize() {
      // some implementation here
   }
}
```

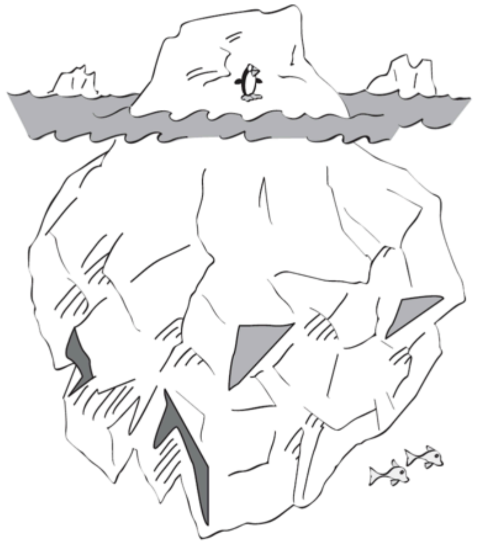
These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

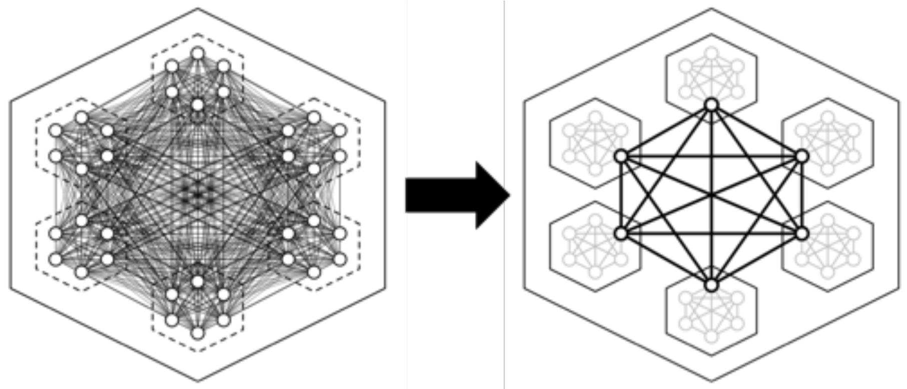
Abstraction

The process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, user will have the information on ***what*** *the object does instead of* ***how*** *it does it.*



In Java, Abstraction is achieved using **abstract classes**, and **interfaces**

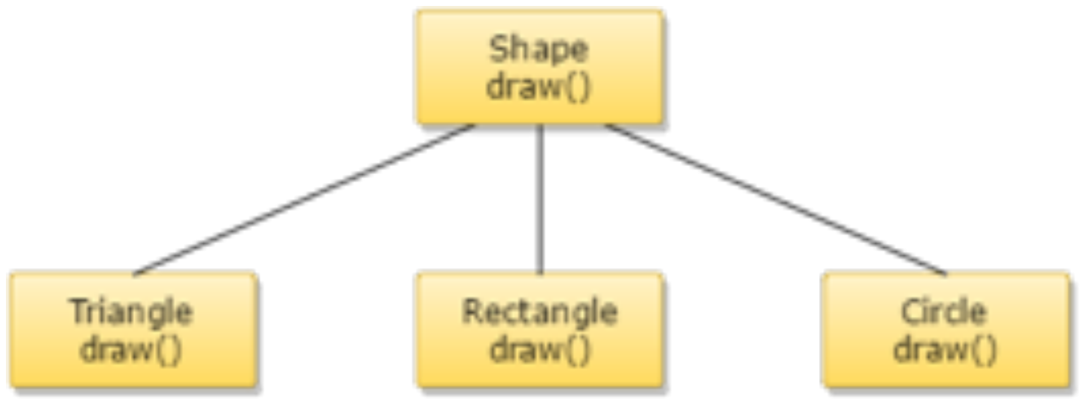


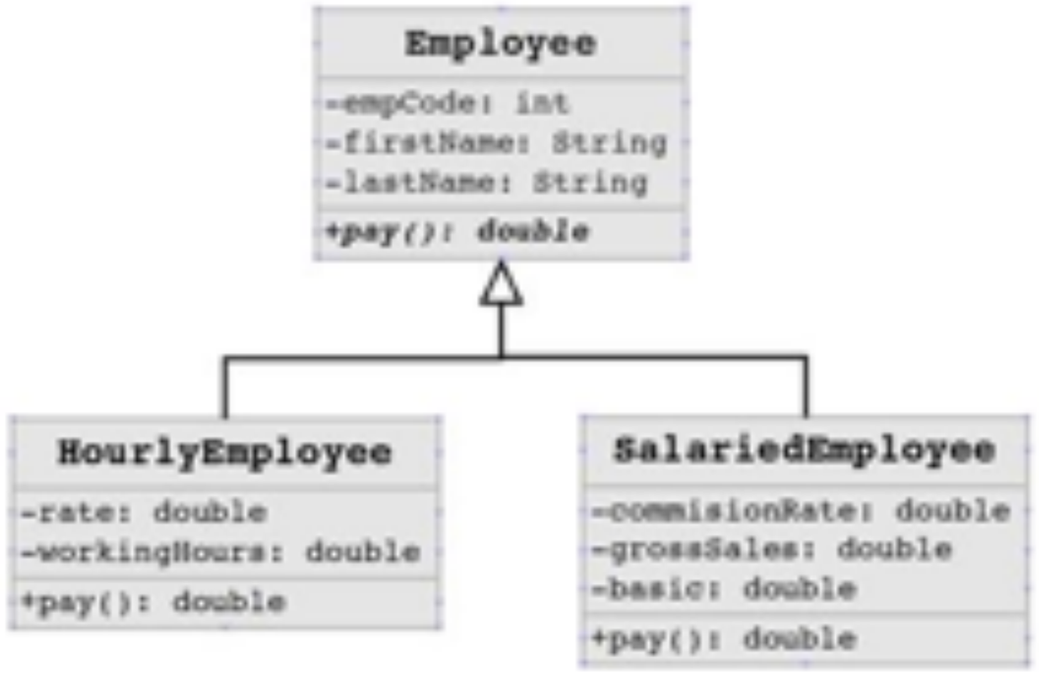


These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Polymorphism

The ability of an object to take on many forms

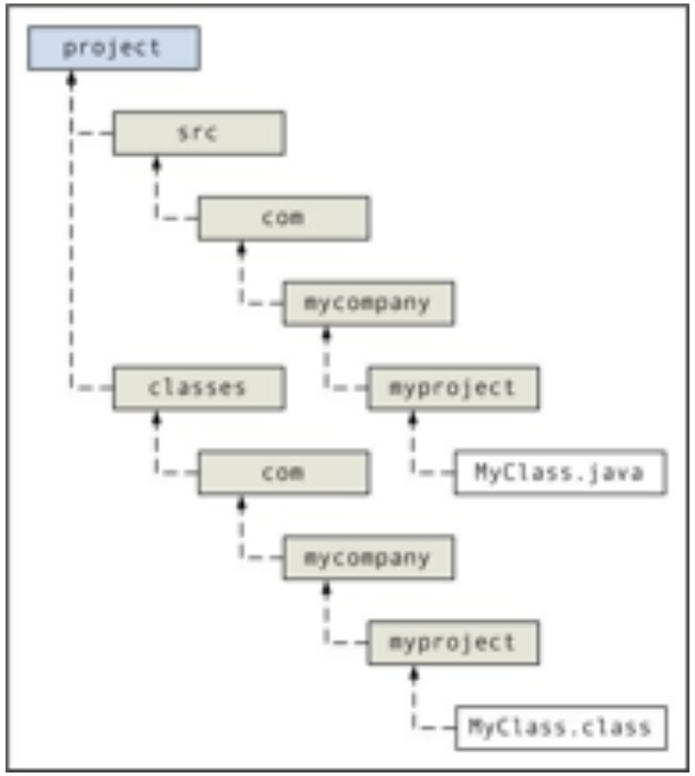


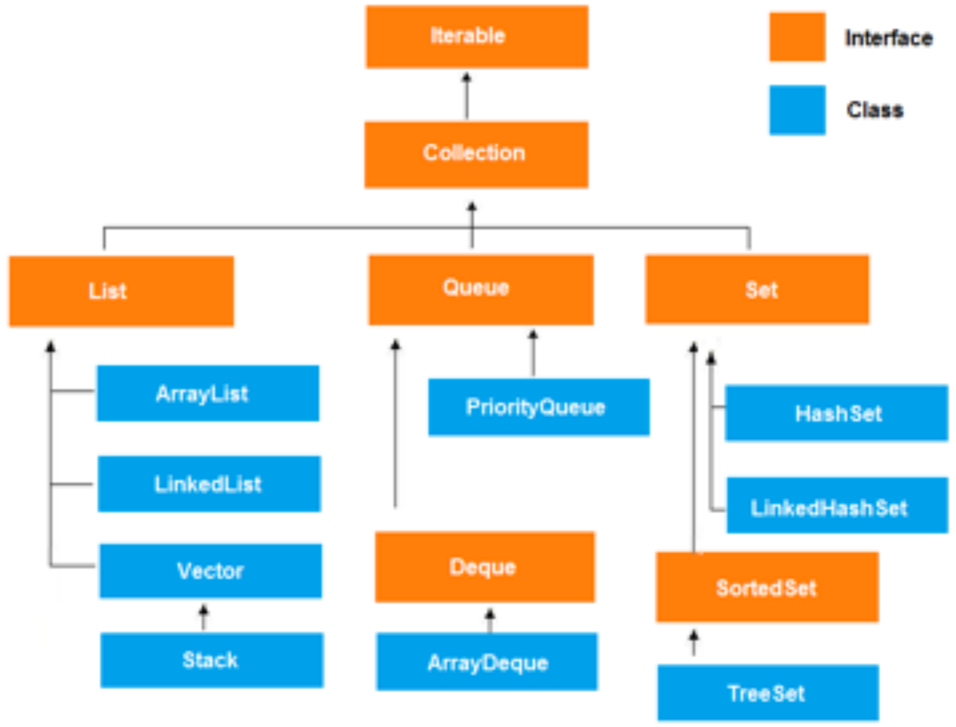


These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Packages

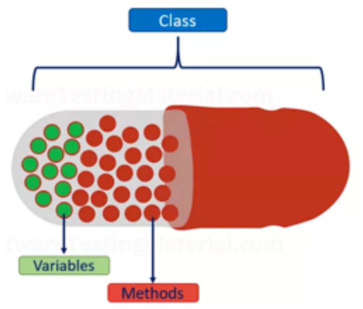
Namespaces that organize a set of related classes and interfaces





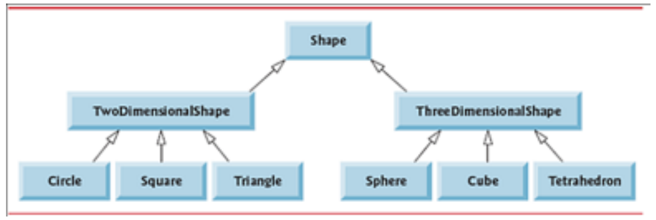
OOP Key Technologies

Encapsulation

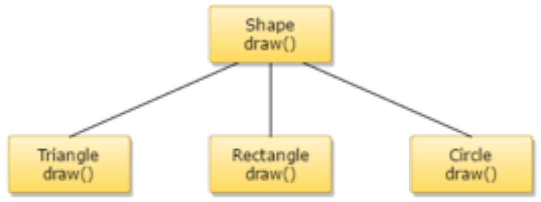


Objects

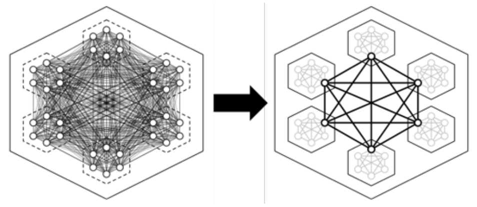
Inheritance



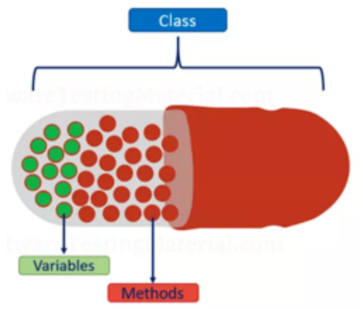
Polymorphism



Abstraction

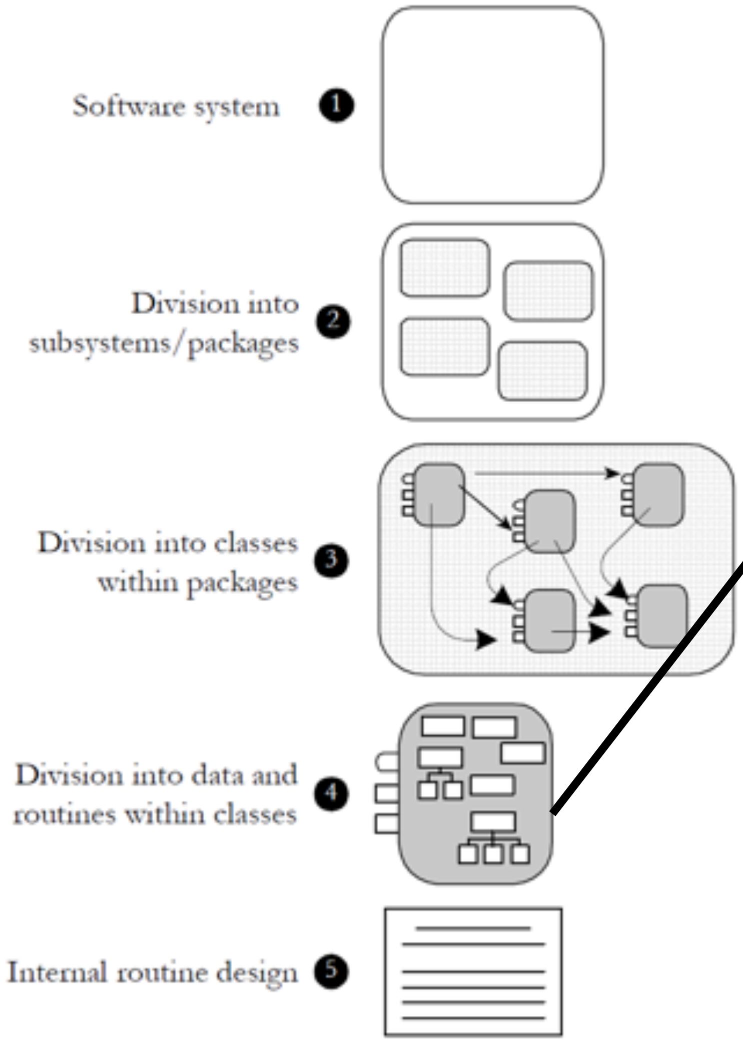


Classes



These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Levels of Design


Software system
1
Division into subsystems/packages
2
Division into classes within packages
3
Division into data and routines within classes
4
Internal routine design
5

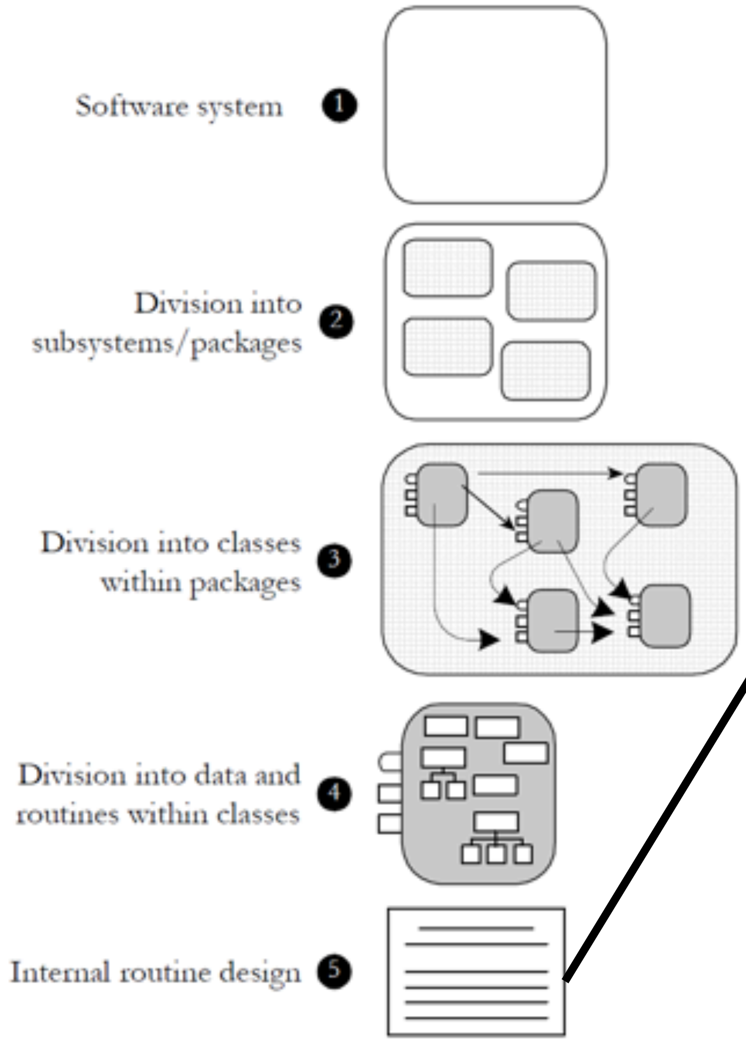
The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

4 Division into Data & Routines within Classes

When you examine the details of the routines inside a class, you can see that many routines are simple boxes but a few are composed of hierarchically organized routines, which require still more design.

The act of fully defining the class's routines often results in a better understanding of the class's interface, and that causes corresponding changes to the interface—that is, changes back at Level 3.

Levels of Design



Software system
1
Division into subsystems/packages
2
Division into classes within packages
3
Division into data and routines within classes
4
Internal routine design
5

The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

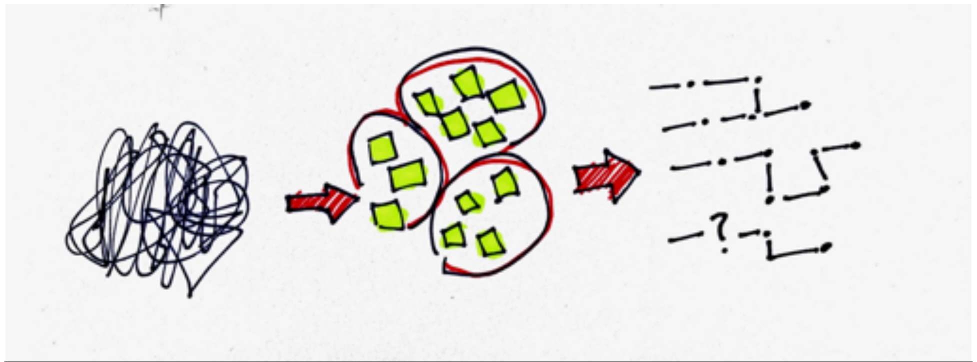
5 Internal Routine Design

Design at the routine level consists of laying out the detailed functionality of the individual routines. The design consists of activities such as writing pseudo-code, looking up algorithms in reference books, deciding how to organize the paragraphs of code in a routine, and writing programming-language code.

Internal routine design is typically left to the individual programmer working on an individual routine.

Design Building Blocks: **Heuristics**

Because design is nondeterministic, skillful application of an effective set of heuristics is the core activity in good software design



GOAL: Minimal Complexity

Design Heuristics: **Find Real-World Objects**

Identify the objects and their attributes (methods and data)

Computer programs are usually based on real-world entities

Identifying the objects' attributes is no more complicated than identifying the objects themselves. Each object has characteristics that are relevant to the computer program.

Determine what can be done to each object

What are the operations performed on each object?

Determine what each object is allowed to do to other objects

The two generic things objects can do to each other are containment and inheritance.

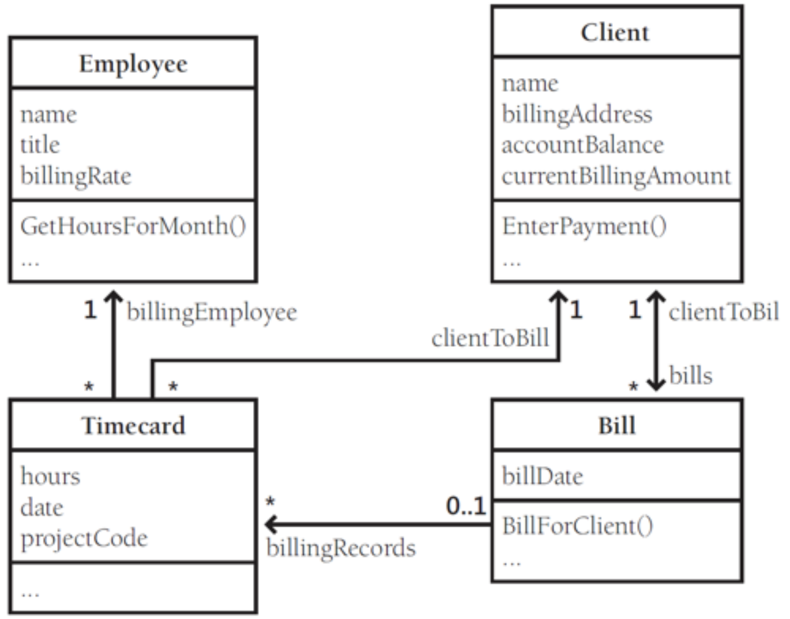
Determine the parts of each object that will be visible to other objects

The visibility of the parts of an object should be determined. This decision has to be made for both fields and methods

Define each object's public interface

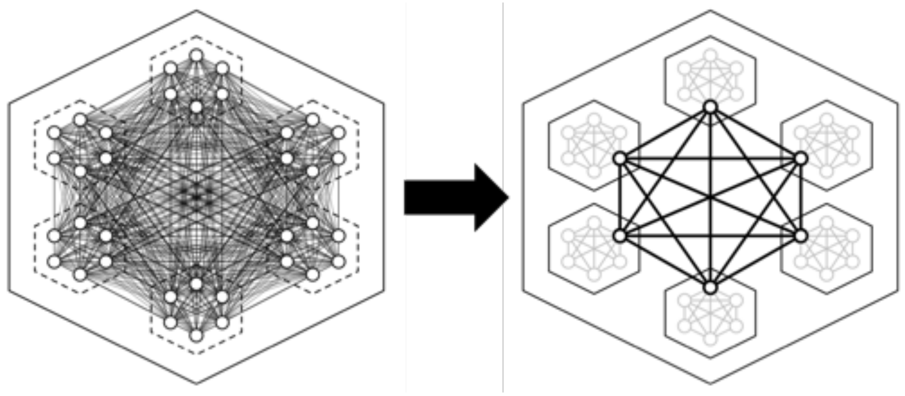
Define the formal, syntactic, programming-language level interfaces to each object.

The data and methods the object exposes to every other object is called the object's "**public interface**." The parts of the object that it exposes to derived objects via inheritance is called the object's "**protected interface**."



Design Heuristics: **Form Consistent Abstractions**

From a complexity point of view, the principal benefit of abstraction is that it allows you to ignore irrelevant details



Good programmers create abstractions at the routine-interface level, class-interface level, and package-interface level

Design Heuristics: **Encapsulate Implementation Details**

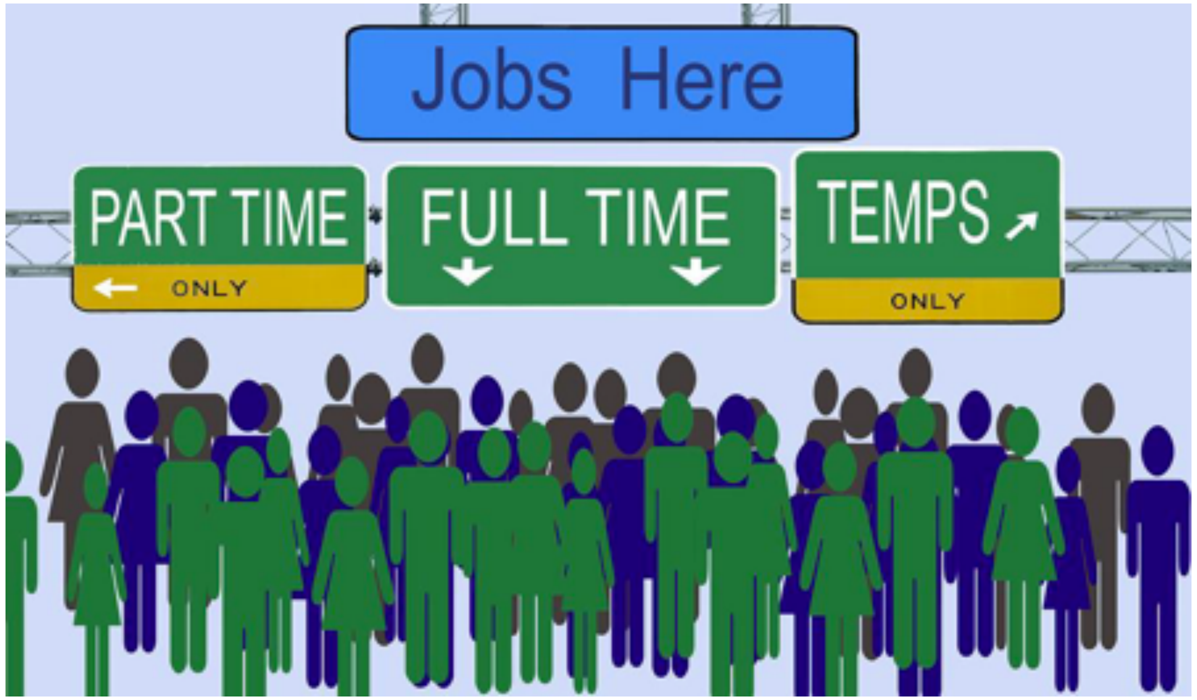
Encapsulation picks up where abstraction leaves off. It helps managing complexity by forbidding you to look at the complexity.

Encapsulation says that, not only are you allowed to take a simpler view of a complex concept, you are not allowed to look at any of the details of the complex concept. What you see is what you get—it's all you get!

Design Heuristics: **Inherit**

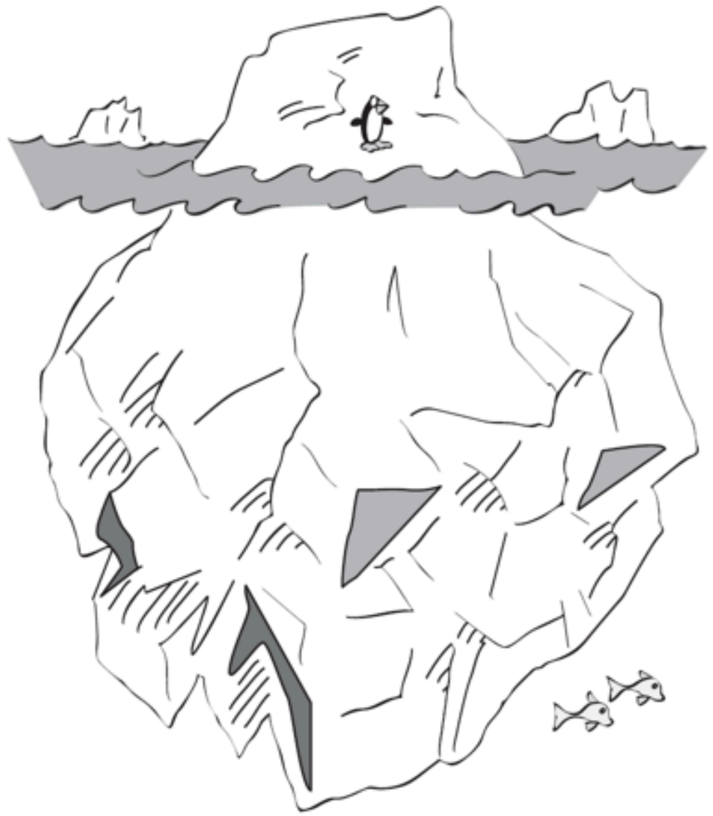
Inheritance is one of object-oriented programming's most powerful tools. It can provide great benefits when used well, and it can do great damage when used naively.

Object-oriented programming allows classes to **inherit commonly used state and behavior** from other classes and let you focus on the features that make a specific class unique

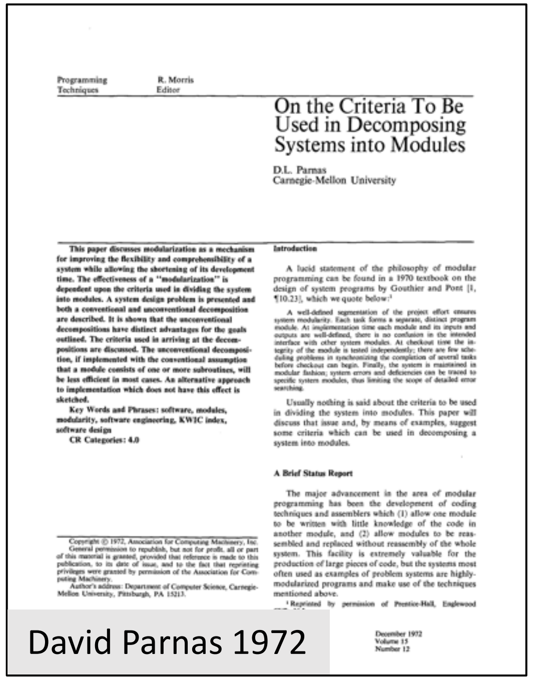


Design Heuristics: **Hide Secrets (Information Hiding)**

Information hiding gives rise to the concepts of encapsulation and modularity and it is associated with the concept of abstraction.



A good class interface is like the tip of an iceberg, leaving most of the class unexposed.



Programming Techniques   R. Morris Editor

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design

CR Categories: 4.0

Copyright © 1972, Association for Computing Machinery, Inc.
General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.
Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

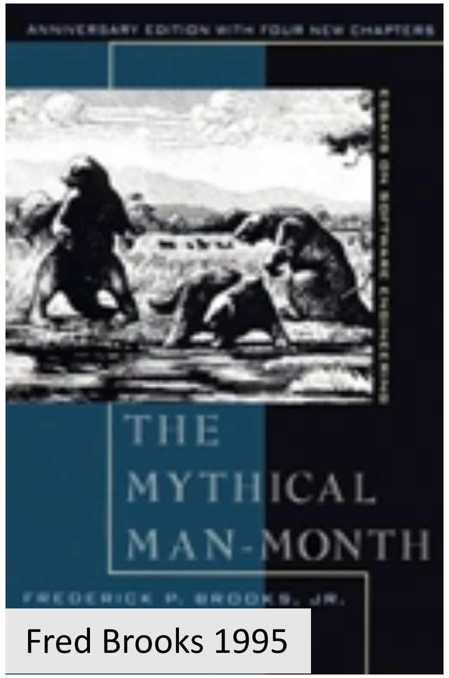
A Brief Status Report

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

[1] Reprinted by permission of Prentice-Hall, Englewood

December 1972
Volume 15
Number 12

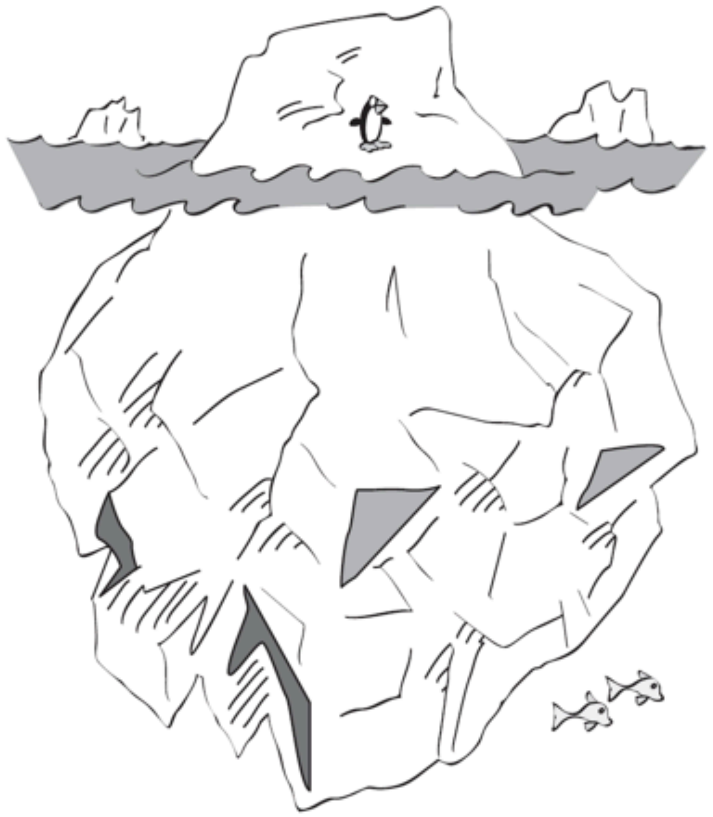
David Parnas 1972


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

Fred Brooks 1995

Information hiding is a particularly powerful heuristic for Software's Primary Technical Imperative because, beginning with its name and throughout its details, it emphasizes hiding complexity

Design Heuristics: **Hide Secrets (Information Hiding)**


A good class interface is like the tip of an iceberg, leaving most of the class unexposed.

Secrets and the Right to Privacy

In information hiding, each class (or package or routine) is characterized by the design or construction decisions that it hides from all other classes. The secret might be an area that's likely to change, the format of a file, the way a data type is implemented, or an area that needs to be walled off from the rest of the program so that errors in that area cause as little damage as possible.

Design Heuristics: **Hide Secrets (Information Hiding)**

Two Categories of Secrets

Hiding complexity so that your brain doesn't have to deal with it unless you're specifically concerned with it

Hiding sources of change so that when change occurs, the effects are localized

Barriers to Information Hiding

Excessive distribution of information

Circular dependencies

Class data mistaken for global data

Perceived performance penalties

Value of Information Hiding

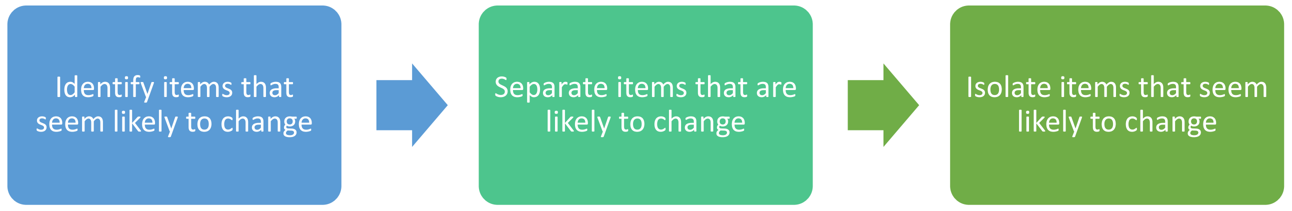
Information hiding is a theoretical techniques that has indisputably proven its value in practice, which has been true for a long time

Large programs that use information hiding were found years ago to be easier to modify—by a factor of 4—than programs that don't

Information hiding is part of the foundation of both structured and object-oriented design.

Design Heuristics: **Identify Areas Likely to Change**

Accommodating changes is one of the most challenging aspects of good program design. The goal is to isolate unstable areas so that the effect of a change will be limited to one routine, class, or package


Identify items that seem likely to change
Separate items that are likely to change
Isolate items that seem likely to change

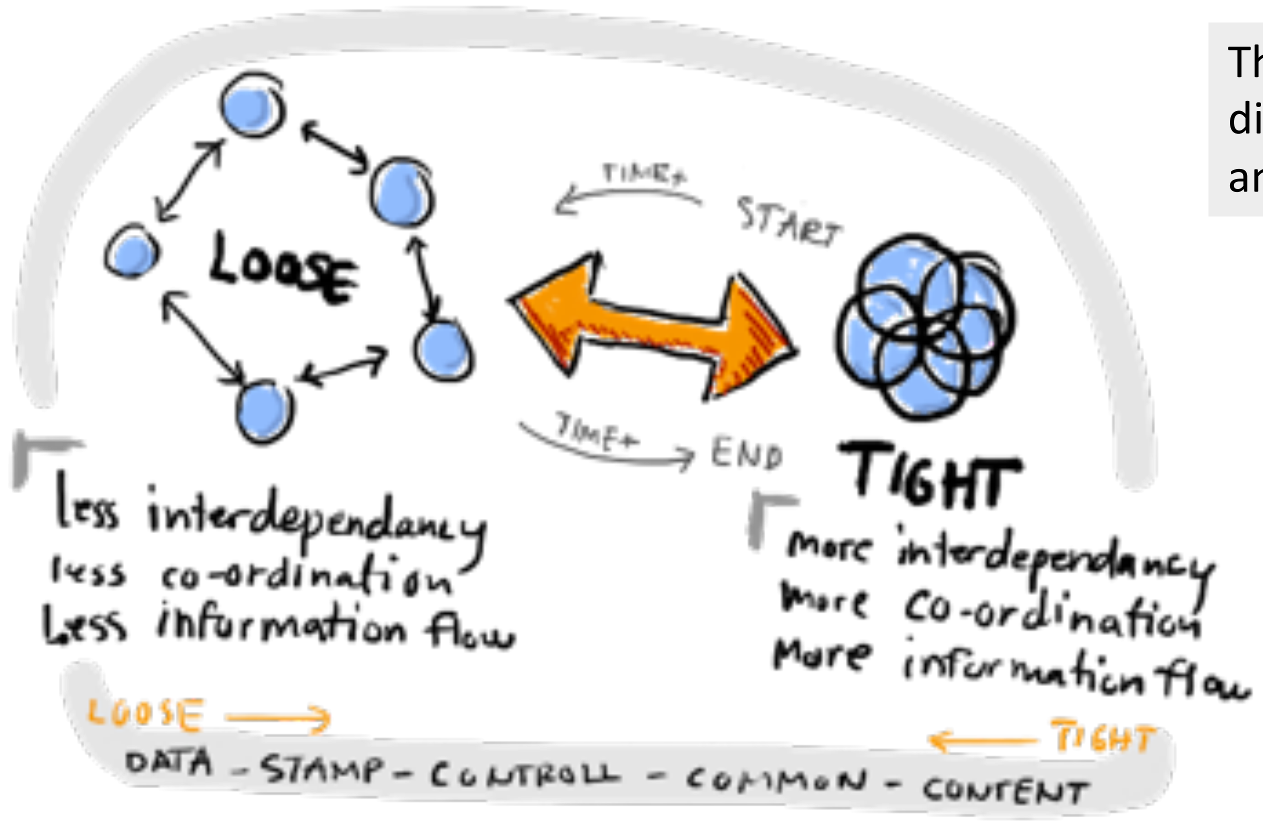
- Business rules
- Hardware dependencies
- Input and output
- Nonstandard language features
- Difficult design and construction areas
- Status variables
- Data-size constraints

Areas Likely to Change

A good technique for identifying areas likely to change is first to identify the minimal subset of the program that might be of use to the user. The subset makes up the core of the system and is unlikely to change.

Design Heuristics: **Keep Coupling Loose**

Coupling describes how tightly a class or routine is related to other classes or routines.



The goal is to create classes and routines with small, direct, visible, and flexible relations to other classes and routines, which is known as "loose coupling.

Classes and **routines** are first and foremost intellectual tools for **reducing complexity**. If they're not making your job simpler, they're not doing their jobs.

Coupling Criteria

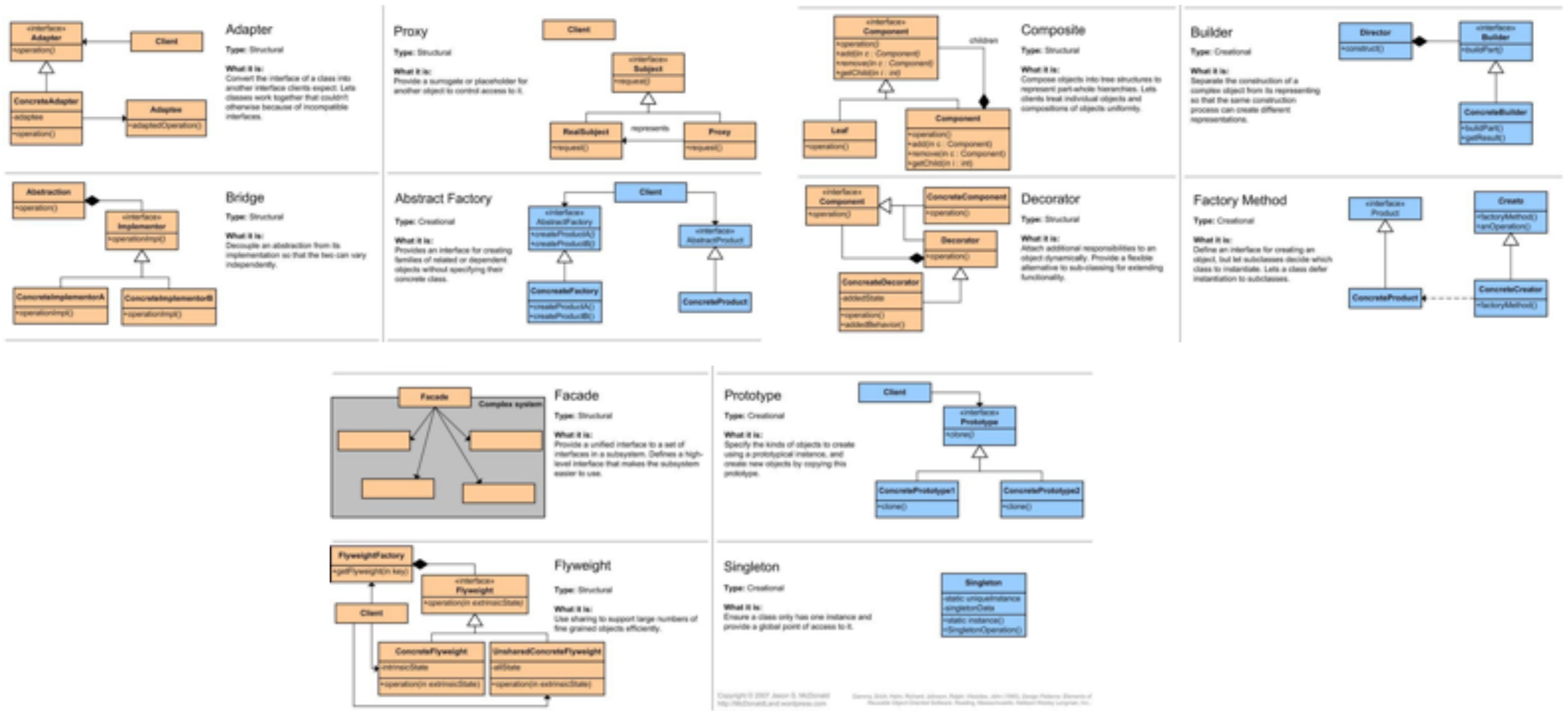
- Size
- Visibility
- Flexibility

Kinds of Coupling

- *Simple-data-parameter coupling*
- *Simple-object coupling*
- *Object-parameter coupling*
- *Semantic coupling*

Design Heuristics: **Look for Common Design Patterns**

Design patterns provide the cores of ready-made solutions that can be used to solve many of software's most common problems



Adapter
Type: Structural
What it is: Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Proxy
Type: Structural
What it is: Provide a surrogate or placeholder for another object to control access to it.

Composite
Type: Structural
What it is: Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Builder
Type: Creational
What it is: Separate the construction of a complex object from its representing so that the same construction process can create different representations.

Bridge
Type: Structural
What it is: Decouple an abstraction from its implementation so that the two can vary independently.

Abstract Factory
Type: Creational
What it is: Provides an interface for creating families of related or dependent objects without specifying their concrete class.

Decorator
Type: Structural
What it is: Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Factory Method
Type: Creational
What it is: Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

Facade
Type: Structural
What it is: Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

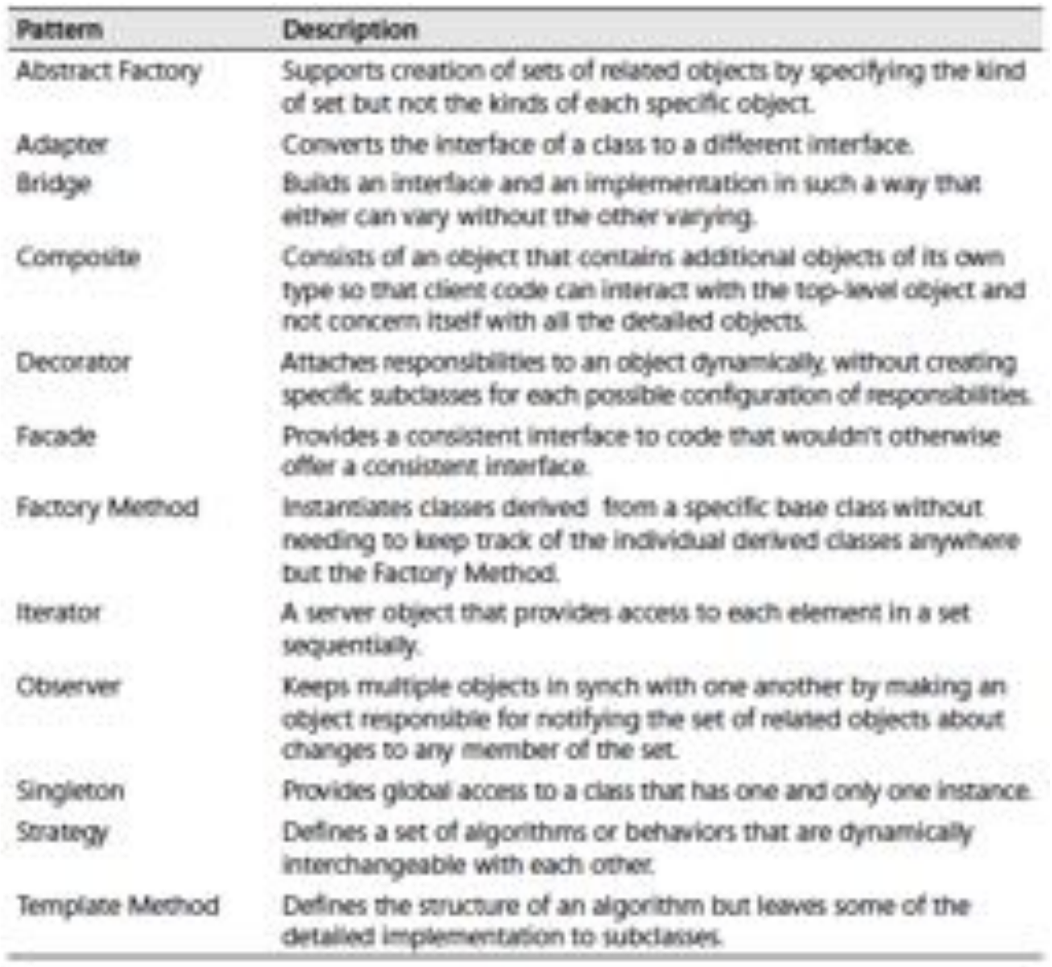
Prototype
Type: Creational
What it is: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Flyweight
Type: Structural
What it is: Use sharing to support large numbers of fine grained objects efficiently.

Singleton
Type: Creational
What it is: Ensure a class only has one instance and provide a global point of access to it.

Design Heuristics: **Look for Common Design Patterns**

Design patterns provide the cores of ready-made solutions that can be used to solve many of software's most common problems



Pattern	Description
Abstract Factory	Supports creation of sets of related objects by specifying the kind of set but not the kinds of each specific object.
Adapter	Converts the interface of a class to a different interface.
Bridge	Builds an interface and an implementation in such a way that either can vary without the other varying.
Composite	Consists of an object that contains additional objects of its own type so that client code can interact with the top-level object and not concern itself with all the detailed objects.
Decorator	Attaches responsibilities to an object dynamically, without creating specific subclasses for each possible configuration of responsibilities.
Facade	Provides a consistent interface to code that wouldn't otherwise offer a consistent interface.
Factory Method	Instantiates classes derived from a specific base class without needing to keep track of the individual derived classes anywhere but the Factory Method.
Iterator	A server object that provides access to each element in a set sequentially.
Observer	Keeps multiple objects in synch with one another by making an object responsible for notifying the set of related objects about changes to any member of the set.
Singleton	Provides global access to a class that has one and only one instance.
Strategy	Defines a set of algorithms or behaviors that are dynamically interchangeable with each other.
Template Method	Defines the structure of an algorithm but leaves some of the detailed implementation to subclasses.

Reduce complexity by providing ready-made abstractions

Reduce errors by institutionalizing details of common solutions

Provide heuristic value by suggesting design alternatives

Streamline communication by moving the design dialog to a higher level

One potential trap with patterns is **force-fitting code to use a pattern**. In some cases, shifting code slightly to conform to a well-recognized pattern will ***improve*** understandability of the code. But if the code has to be shifted too far, forcing it to look like a standard pattern can sometimes ***increase complexity***.

Another potential trap with patterns is **feature-itis**: using a pattern because of a desire to try out a pattern rather than because the pattern is an appropriate design solution.

Design Heuristics: **Other Heuristics**

Aim for Strong Cohesion

Cohesion refers to how closely all the routines in a class or all the code in a routine support a central purpose—how focused the class is

Build Hierarchies

Hierarchies are a useful tool for reducing complexity because they allow you to focus on only the level of detail you're currently concerned with.

Formalize Class Contracts

Contracts are useful for managing complexity because, at least in theory, the object can safely ignore any noncontractual behavior.

Assign Responsibilities

Asking what each object should be responsible for

Design for Test

A thought process that can yield interesting design insights is to ask what the system will look like if you design it to facilitate testing.

Avoid Failure

The high-profile security lapses of various well-known systems the past few years make it hard to avoid security vulnerabilities but careful considerations should be taken to known failures.

Choose Binding Time Consciously

Binding time refers to the time a specific value is bound to a variable. Code that binds early tends to be simpler, but it also tends to be less flexible.

Make Central Points of Control

The Principle of One Right Place—there should be One Right Place to look for any nontrivial piece of code, and One Right Place to make a likely maintenance change"

Consider Using Brute Force

A brute-force solution that works is better than an elegant solution that doesn't work

Draw a Diagram

You actually want to leave out most of the 1000 words because one point of using a picture is that a picture can represent the problem at a higher level of abstraction

Keep Your Design Modular

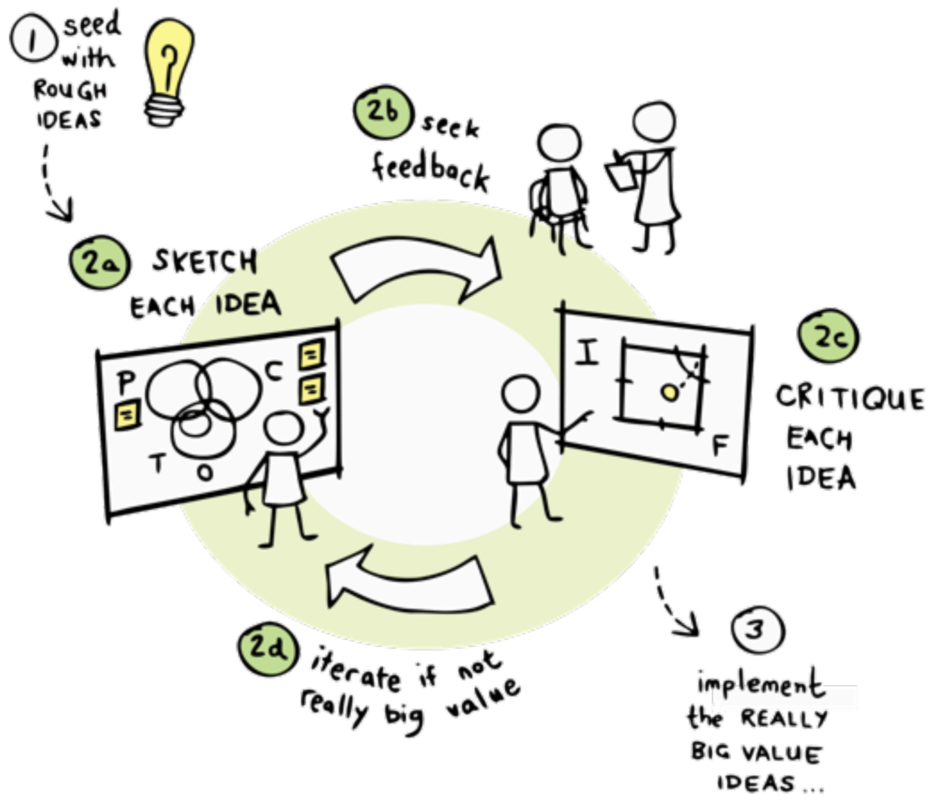
Modularity's goal is to make each routine or class like a "black box": You know what goes in, and you know what comes out, but you don't know what happens inside.

Design Practices

Heuristics related to design attributes—what you want the completed design to look like.

Design Practices: **Iterate**

Design is an iterative process. You don't usually go from point A only to point B; you go from point A to point B and back to point A



As you cycle through **candidate designs** and **try different approaches**, you'll look at both **high-level** and **low-level views**.

The big picture you get from working with high-level issues will help you to put the low-level details in perspective. The details you get from working with low-level issues will provide a foundation in solid reality for the high-level decisions.

Design Practices: **Divide and Conquer**

As Edsger Dijkstra pointed out, no one's skull is big enough to contain all the details of a complex program, and that applies just as well to design

Incremental refinement is a powerful tool for managing complexity.

Divide the program into different areas of concern, then tackle each of those areas individually. If you run into a dead end in one of the areas, iterate!

Design Practices: **Top-Down and Bottom-Up**

Top-down design begins at a high level of abstraction. You define base classes or other nonspecific design elements. As you develop the design, you increase the level of detail, identifying derived classes, collaborating classes, and other detailed design elements.

Bottom-up design starts with specifics and works toward generalities. It typically begins by identifying concrete objects and then generalizes aggregations of objects and base classes from those specifics

Bottom Up
Top Down

Design Practices: **Experimental Prototyping**

You can't fully define the design problem until you've at least partially solved it.

Prototyping means writing the absolute minimum amount of throwaway code that's needed to answer a specific design question.

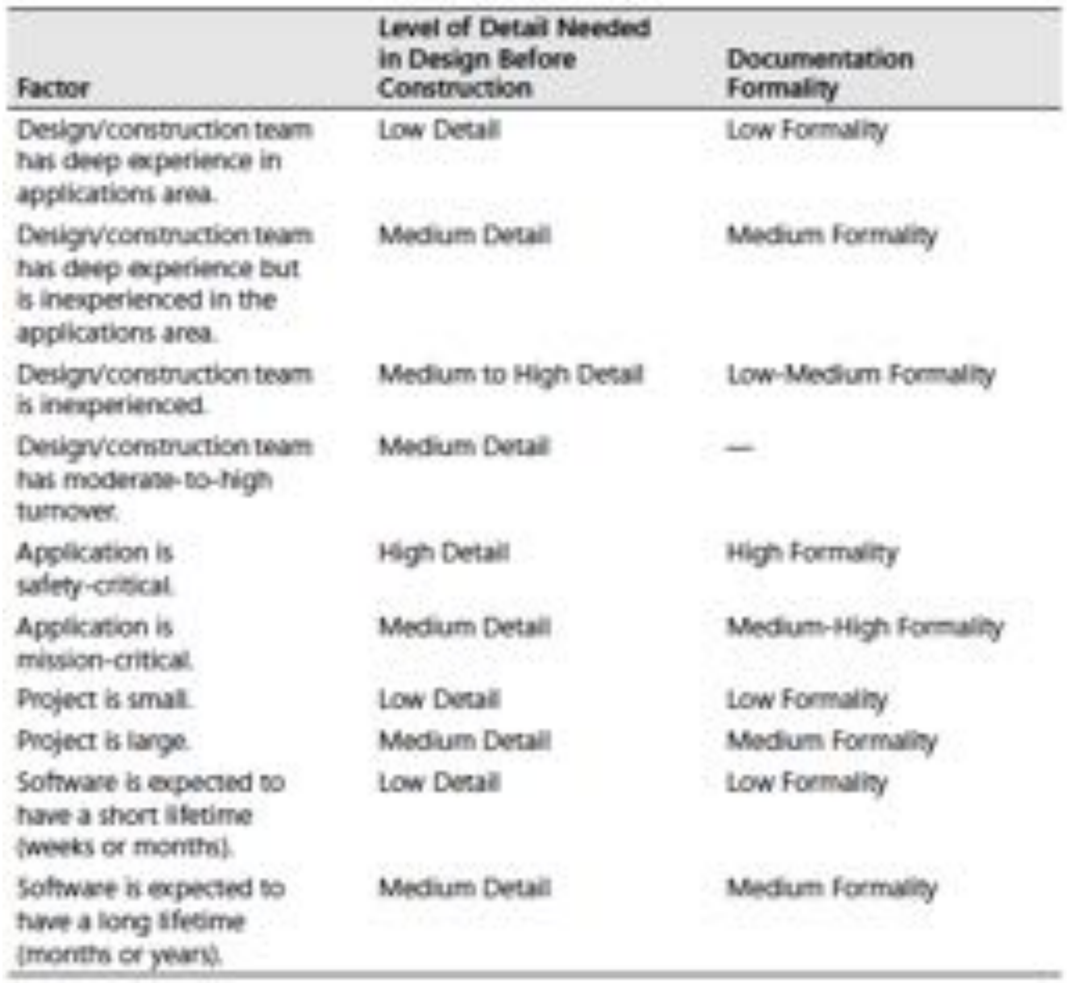
A risk of prototyping arises when developers do not treat the code as **throwaway code**.

Design Practices: **Collaborative Design**

In design, two heads are often better than one, whether those two heads are organized formally or informally

Design Practices: **How Much Design Is Enough?**

Sometimes only the barest sketch of an architecture is mapped out before coding begins. Other times, teams create designs at such a level of detail that coding becomes a mostly mechanical exercise.



Factor	Level of Detail Needed in Design Before Construction	Documentation Formality
Design/construction team has deep experience in applications area.	Low Detail	Low Formality
Design/construction team has deep experience but is inexperienced in the applications area.	Medium Detail	Medium Formality
Design/construction team is inexperienced.	Medium to High Detail	Low-Medium Formality
Design/construction team has moderate-to-high turnover.	Medium Detail	—
Application is safety-critical.	High Detail	High Formality
Application is mission-critical.	Medium Detail	Medium-High Formality
Project is small.	Low Detail	Low Formality
Project is large.	Medium Detail	Medium Formality
Software is expected to have a short lifetime (weeks or months).	Low Detail	Low Formality
Software is expected to have a long lifetime (months or years).	Medium Detail	Medium Formality

Design Practices: **Capturing Your Design Work**

- Insert design documentation into the code itself
- Capture design discussions and decisions on a Wiki
- Write e-mail summaries
- Use a digital camera
- Save design flip charts
- Use CRC (Class, Responsibility, Collaborator) cards
- Create UML diagrams at appropriate levels of detail

SUMMARY

- Software’s Primary Technical Imperative is managing complexity . This is greatly aided by a design focus on simplicity.
- Simplicity is achieved in two general ways: minimizing the amount of essential complexity that anyone’s brain has to deal with at any one time, and keeping accidental complexity from proliferating needlessly.
- Design is heuristic. Dogmatic adherence to any single methodology hurts creativity and hurts your programs
- Good design is iterative; the more design possibilities you try, the better your final design will be.
- Information hiding is a particularly valuable concept. Asking “What should I hide?” settles many difficult design issues.
- Lots of useful, interesting information on design is available outside this book. The perspectives presented here are just the tip of the iceberg.

Working Classes

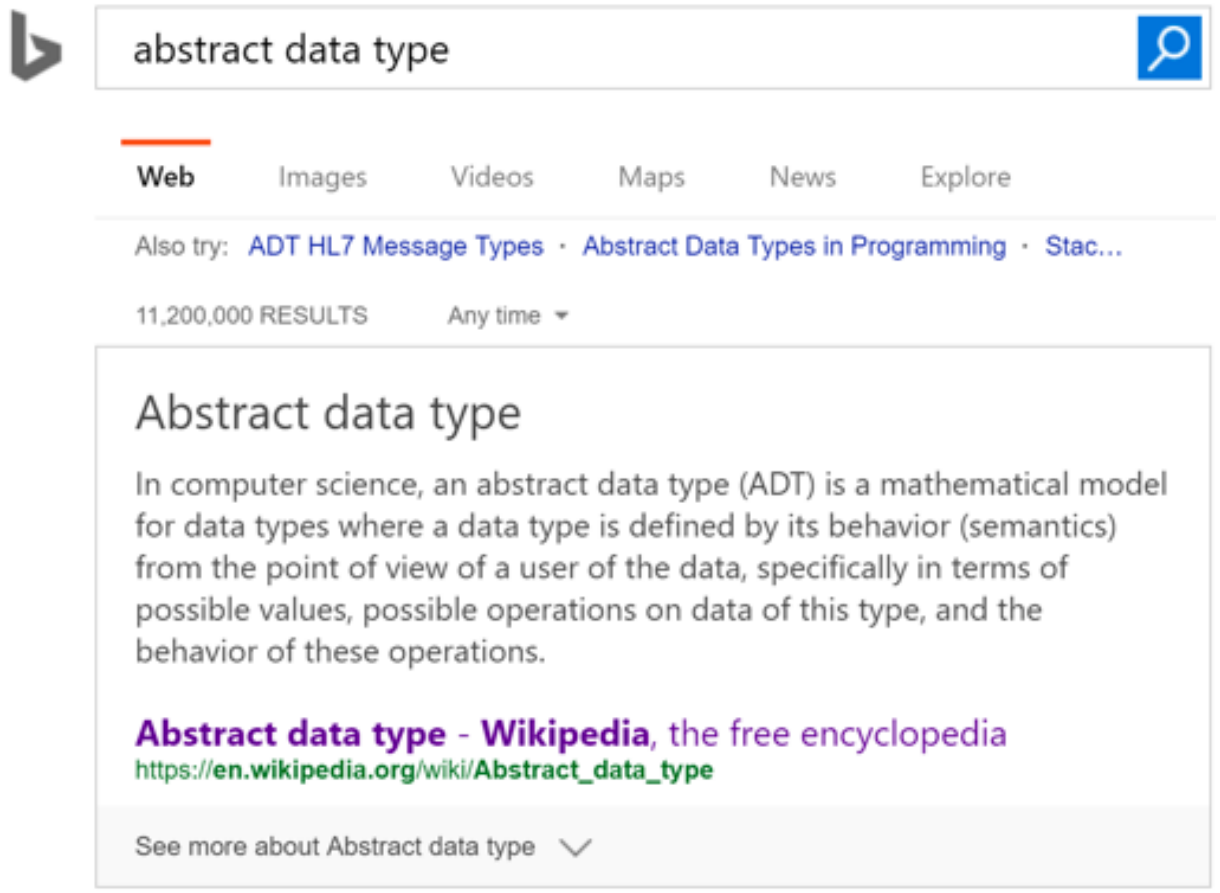
*In the dawn of computing, programmers thought about programming in terms of **statements**.*
*Throughout the 1970s and 1980s, programmers began thinking about programs in terms of **routines**.*
*In the twenty-first century, programmers think about programming in terms of **classes**.*

A class is a collection of data and routines that share a **cohesive**, **well-defined responsibility**. A class might also be a collection of routines that provides a **cohesive set of services even if no common data is involved**

Maximizes the portion of a program that you can safely ignore while working on another section of code

Class Foundations: **Abstract Data Types (ADTs)**

An abstract data type is a collection of data and operations that work on that data.



abstract data type

Web Images Videos Maps News Explore

Also try: ADT HL7 Message Types · Abstract Data Types in Programming · Stac...

11,200,000 RESULTS Any time

Abstract data type

In computer science, an abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

Abstract data type - **Wikipedia**, the free encyclopedia
https://en.wikipedia.org/wiki/Abstract_data_type

See more about Abstract data type

Understanding ADTs is essential to understanding object-oriented programming.

An ADT might be a graphics window with all the operations that affect it, a file and file operations, an insurance-rates table and the operations on it, or something else

Tap into the power of being able to work in the problem domain rather than at the low-level implementation domain!

Instead of inserting a node into a linked list, you can add a cell to a spreadsheet, a new type of window to a list of window types, or another passenger car to a train simulation

Class Foundations: **Abstract Data Types (ADTs)**

Suppose you're writing a program to control text output to the screen using a variety of typefaces, point sizes, and font attributes (such as bold and italic)

Using ADT

A group of font routines bundled with the data—the typeface names, point sizes, and font attributes—they operate on.

```
currentFont.SetSizeInPoints( sizeInPoints )
currentFont.SetSizeInPixels( sizeInPixels )
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace( faceName )
```

Not Using ADT

Ad hoc approach to manipulating fonts. For example, if you need to change to a 12-point font size, which happens to be 16 pixel high

```
currentFont.size = 16
currentFont.size = PointsToPixels( 12 )
currentFont.sizeInPixels = PointsToPixels( 12 )
currentFont.sizeInPixels = PointsToPixels( 12 )
currentFont.attribute = currentFont.attribute or 0x02
currentFont.attribute = currentFont.attribute or BOLD
currentFont.bold = True
```

Class Foundations: **Abstract Data Types (ADTs)**

You can hide implementation details

Changes don't affect the whole program

You can make the interface more informative

It's easier to improve performance

The program is more obviously correct

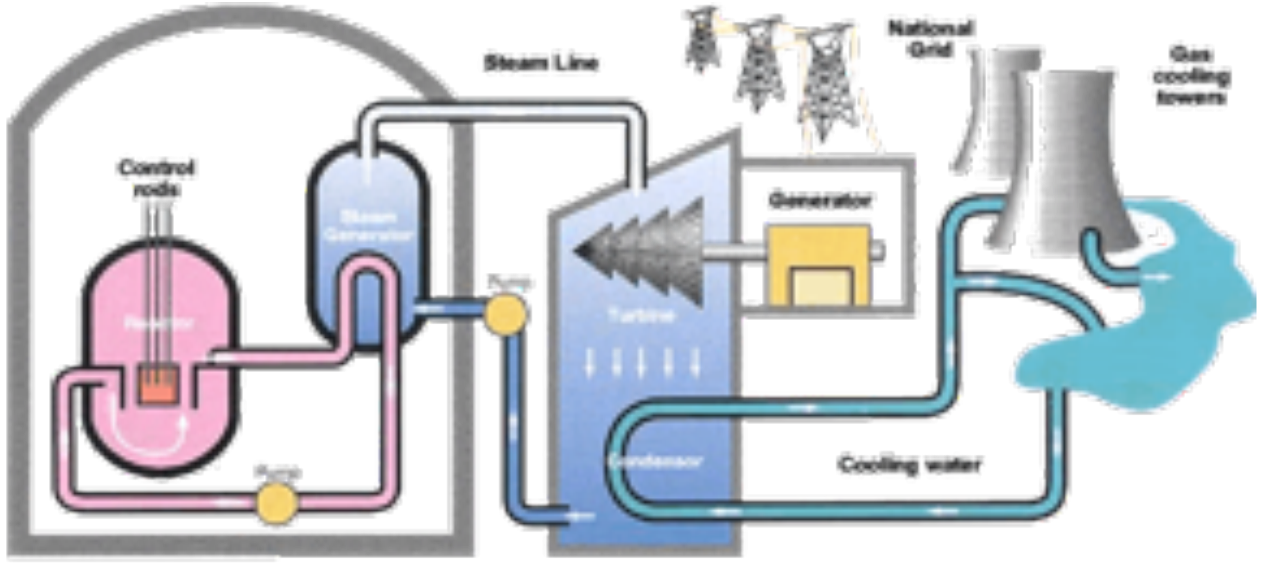
The program becomes more self-documenting

You don't have to pass data all over your program

You're able to work with real-world entities rather than with low-level implementation structures

Class Foundations: **Abstract Data Types (ADTs)**

Suppose you're writing software that controls the cooling system for a nuclear reactor. You can treat the cooling system as an abstract data type.

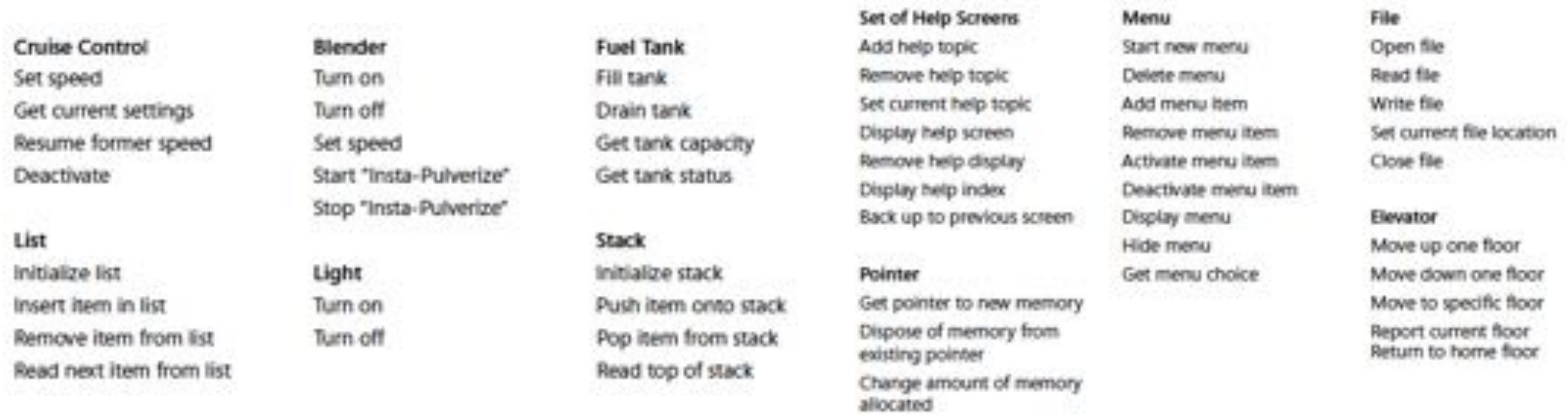


```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate( rate )
coolingSystem.OpenValve( valveNumber )
coolingSystem.CloseValve( valveNumber )
```

The specific environment would determine the code written to implement each of these operations.

The rest of the program could deal with the cooling system through these functions and wouldn't have to worry about internal details of data-structure implementations, data-structure limitations, changes, and so on.

Class Foundations: **Abstract Data Types (ADTs)**



Cruise Control
- Set speed
- Get current settings
- Resume former speed
- Deactivate

List
- Initialize list
- Insert item in list
- Remove item from list
- Read next item from list

Blender
- Turn on
- Turn off
- Set speed
- Start "Insta-Pulverize"
- Stop "Insta-Pulverize"

Light
- Turn on
- Turn off

Fuel Tank
- Fill tank
- Drain tank
- Get tank capacity
- Get tank status

Stack
- Initialize stack
- Push item onto stack
- Pop item from stack
- Read top of stack

Set of Help Screens
- Add help topic
- Remove help topic
- Set current help topic
- Display help screen
- Remove help display
- Display help index
- Back up to previous screen

Pointer
- Get pointer to new memory
- Dispose of memory from existing pointer
- Change amount of memory allocated

Menu
- Start new menu
- Delete menu
- Add menu item
- Remove menu item
- Activate menu item
- Deactivate menu item
- Display menu
- Hide menu
- Get menu choice

File
- Open file
- Read file
- Write file
- Set current file location
- Close file

Elevator
- Move up one floor
- Move down one floor
- Move to specific floor
- Report current floor
- Return to home floor

Build or use typical low-level data types as ADTs, not as low-level data types

Treat common objects such as files as ADTs

Treat even simple items as ADTs

Refer to an ADT independently of the medium it's stored on

Class Foundations: **Abstract Data Types (ADTs)**

Handling Multiple Instances of Data with ADTs in Non-Object-Oriented Environments

```
SetCurrentFontSize( sizeInPoints )
SetCurrentFontBoldOn()
SetCurrentFontBoldOff()
SetCurrentFontItalicOn()
SetCurrentFontItalicOff()
SetCurrentFontTypeFace( faceName )

CreateFont( fontId )
DeleteFont( fontId )
SetCurrentFont( fontId )
```

Option 1: Explicitly identify instances each time you use ADT services.

Option 2: Explicitly provide the data used by the ADT services.

Option 3: Use implicit instances

Good Class Interfaces

The first and probably most important step in creating a high-quality class is creating a good interface.

Creating a good abstraction for the interface to represent and ensuring that the details remain hidden behind the abstraction.

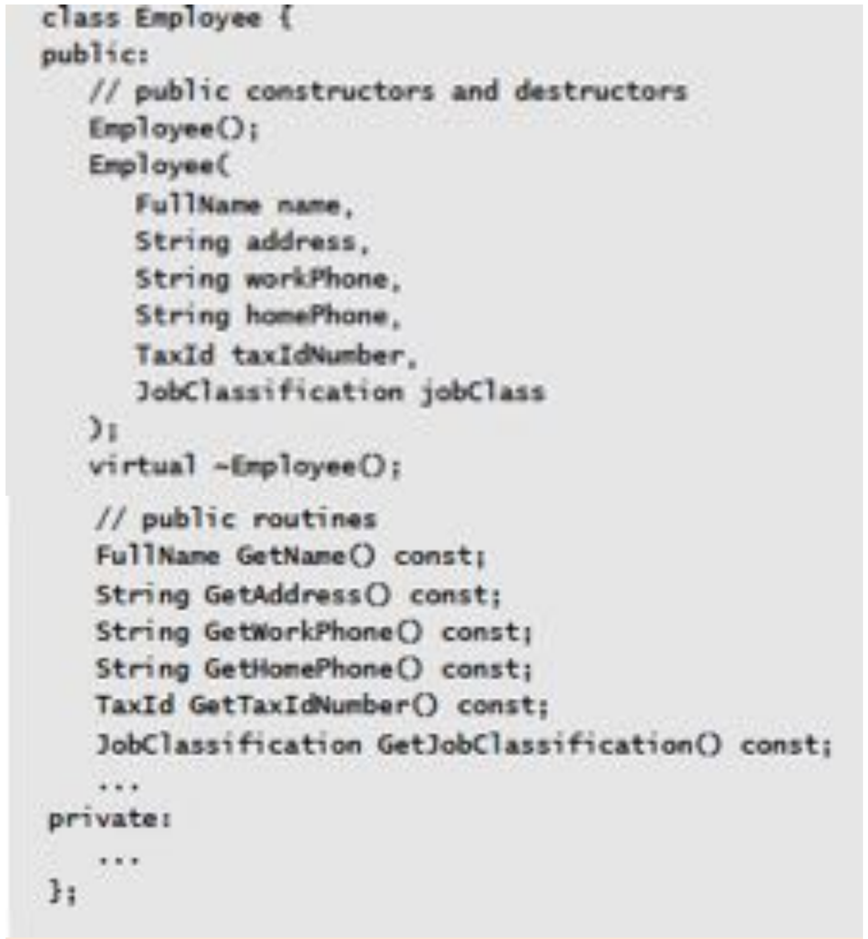
Good Abstraction

Good Encapsulation

Good Class Interfaces: **Good Abstraction**

A class interface provides an abstraction of the implementation that's hidden behind the interface

It would contain data describing the employee's name, address, phone number, and so on. It would offer services to initialize and use an employee

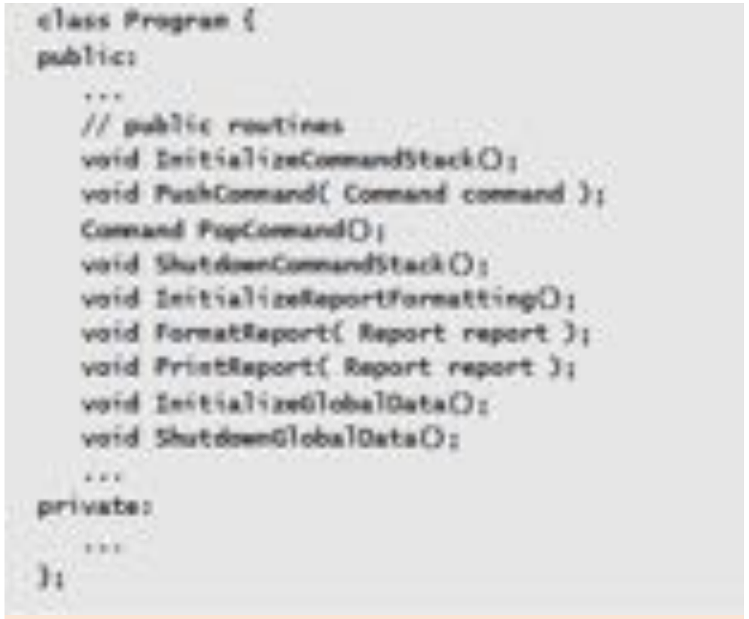


```
class Employee {
public:
   // public constructors and destructors
   Employee();
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   virtual ~Employee();
   // public routines
   FullName GetName() const;
   String GetAddress() const;
   String GetWorkPhone() const;
   String GetHomePhone() const;
   TaxId GetTaxIdNumber() const;
   JobClassification GetJobClassification() const;
   ...
private:
   ...
};
```

Internally, this class might have additional routines and data to support these services, but users of the class don't need to know anything about them, so it is great.

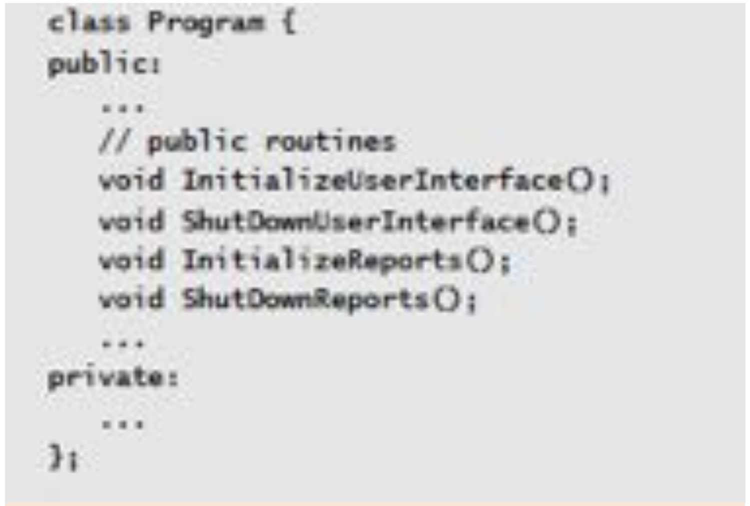
Good Class Interfaces: **Good Abstraction**

A class interface provides an abstraction of the implementation that's hidden behind the interface



```
class Program {
public:
   ...
   // public routines
   void InitializeCommandStack();
   void PushCommand( Command command );
   Command PopCommand();
   void ShutdownCommandStack();
   void InitializeReportFormatting();
   void FormatReport( Report report );
   void PrintReport( Report report );
   void InitializeGlobalData();
   void ShutdownGlobalData();
   ...
private:
   ...
};
```

It's hard to see any connection among the command stack and report routines or the global data. The class interface doesn't present a consistent abstraction. The routines should be reorganized into more focused classes, each of which provides a better abstraction in its interface.



```
class Program {
public:
   ...
   // public routines
   void InitializeUserInterface();
   void ShutDownUserInterface();
   void InitializeReports();
   void ShutDownReports();
   ...
private:
   ...
};
```

The cleanup of this interface assumes that some of the original routines were moved to other, more appropriate classes and some were converted to private routines used by InitializeUserInterface() and the other routines.

Good Class Interfaces: **Good Abstraction**

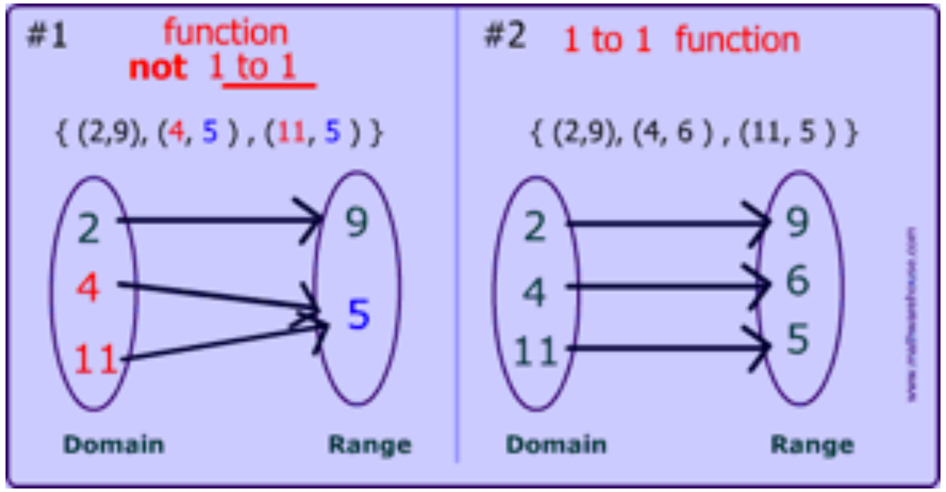
The pursuit of good abstract interfaces gives rise to several guidelines for creating class interfaces.

Present a consistent level of abstraction in the class interface

One Class = One ADT

Each class should implement one and only one ADT.

If you find a class implementing more than one ADT, or if you can't determine what ADT the class implements, it's time to reorganize the class into one or ore well defined ADTs.

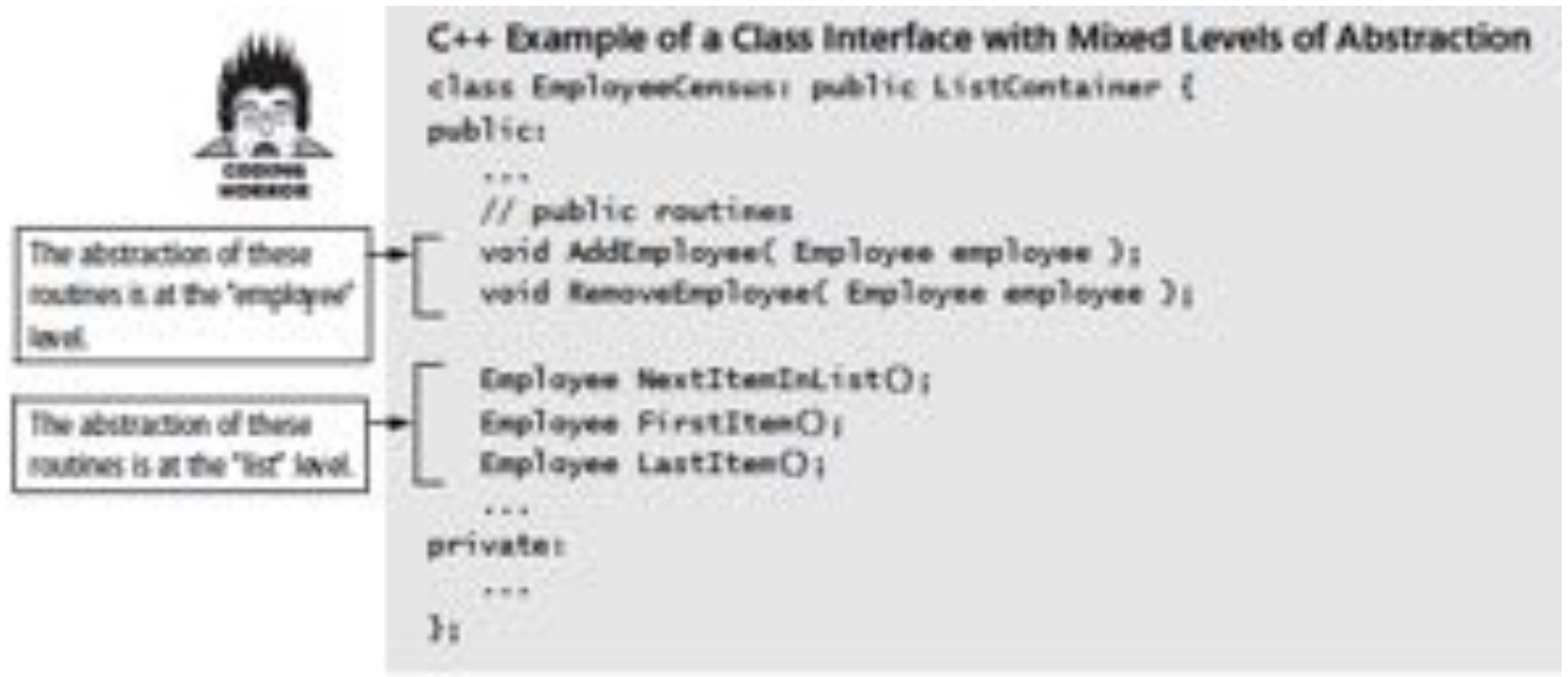


Good Class Interfaces: **Good Abstraction**

The pursuit of good abstract interfaces gives rise to several guidelines for creating class interfaces.

Present a consistent level of abstraction in the class interface

One Class = One ADT


CODING HORROR

The abstraction of these routines is at the "employee" level.

The abstraction of these routines is at the "list" level.

C++ Example of a Class Interface with Mixed Levels of Abstraction

```
class EmployeeCensus: public ListContainer {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );

   Employee NextItemInList();
   Employee FirstItem();
   Employee LastItem();
   ...
private:
   ...
};
```

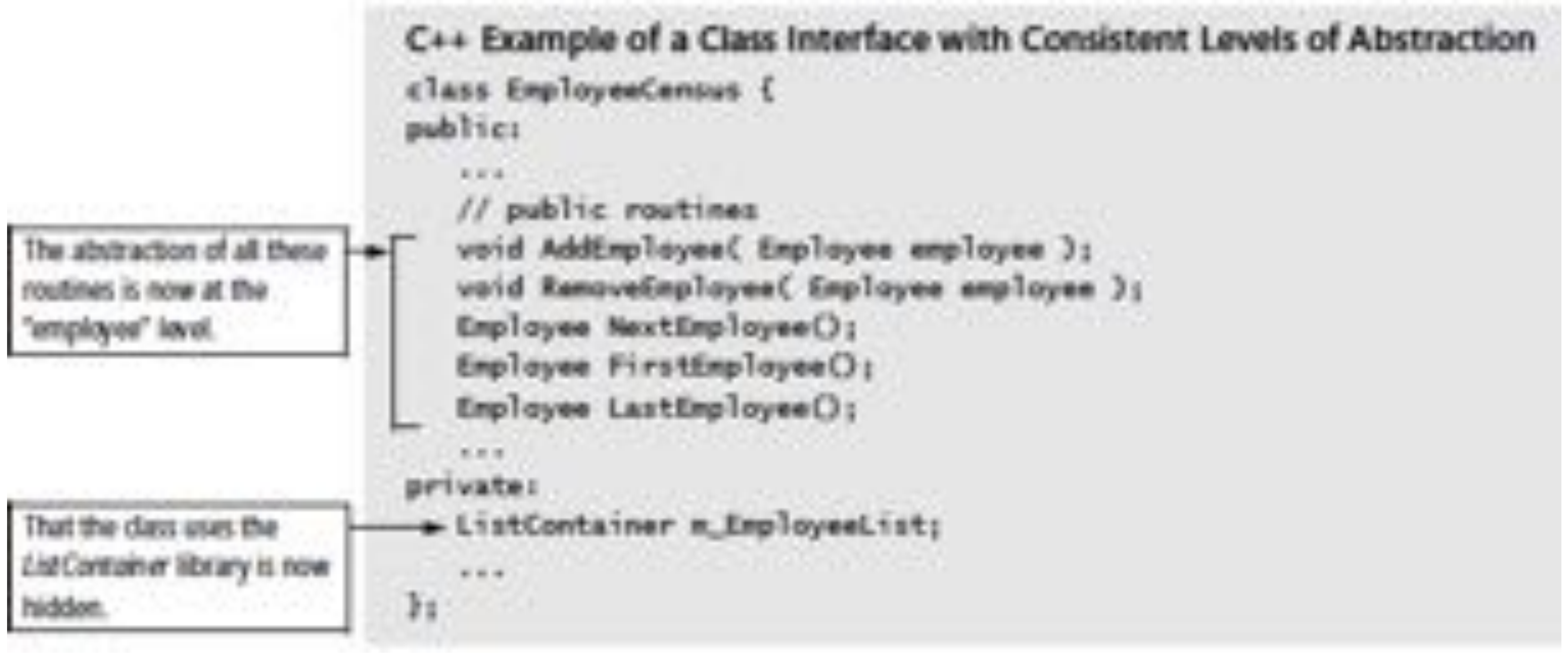
Ask yourself whether the fact that a container class is used should be part of the abstraction. Usually that's an implementation detail that should be hidden from the rest of the program.

Good Class Interfaces: **Good Abstraction**

The pursuit of good abstract interfaces gives rise to several guidelines for creating class interfaces.

Present a consistent level of abstraction in the class interface

One Class = One ADT



C++ Example of a Class Interface with Consistent Levels of Abstraction

```
class EmployeeCensus {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );
   Employee NextEmployee();
   Employee FirstEmployee();
   Employee LastEmployee();
   ...
private:
   ListContainer m_EmployeeList;
   ...
};
```

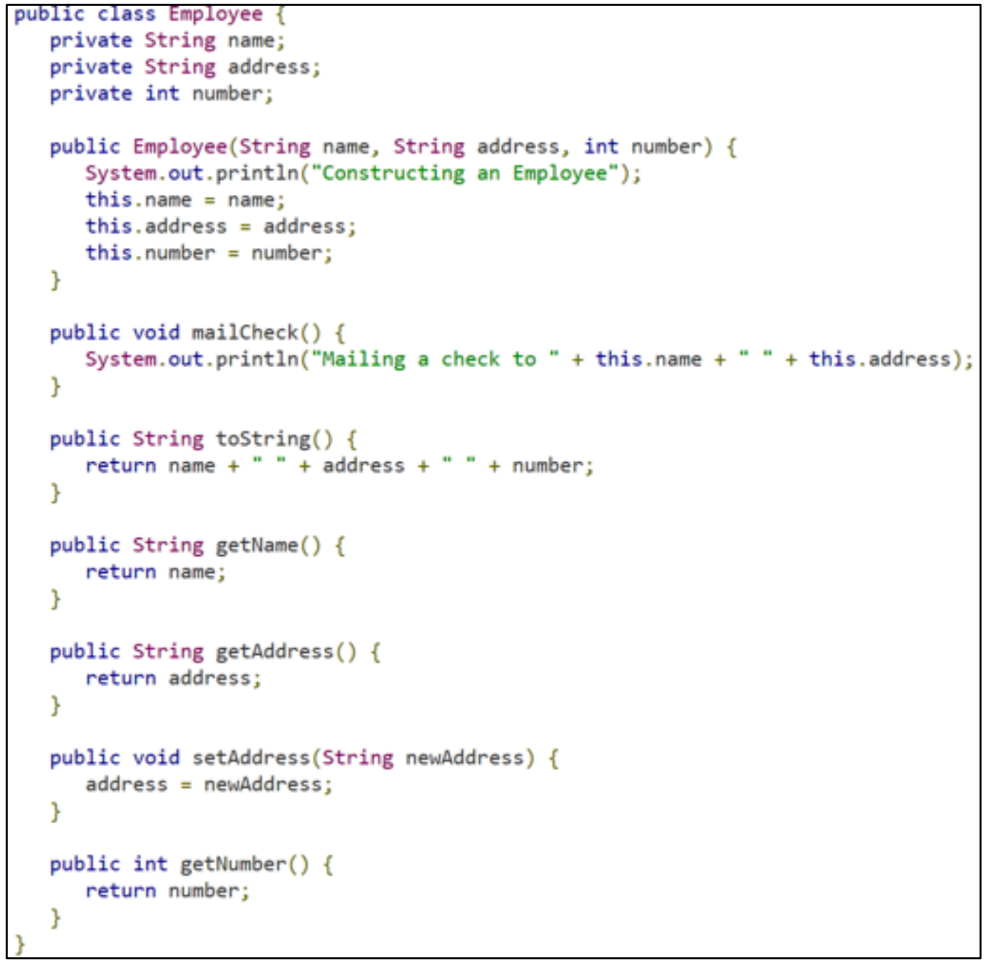
The abstraction of all these routines is now at the "employee" level.

That the class uses the *ListContainer* library is now hidden.

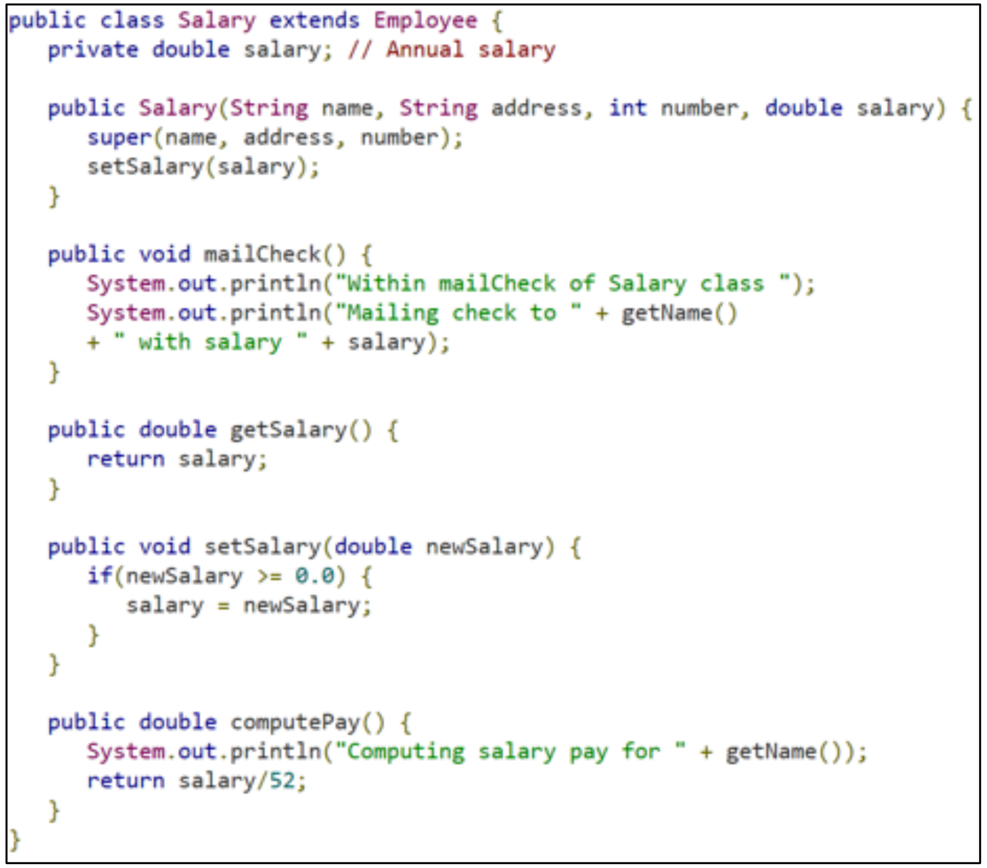
Programmers might argue that inheriting from ListContainer is convenient because it supports polymorphism, allowing an external search or sort function that takes a ListContainer object.

That argument fails the main test for inheritance, which is, "Is inheritance used only for "is a" relationships?“ To inherit from ListContainer would mean that EmployeeCensus "is a“ ListContainer, which obviously false

Good Class Interfaces: **Good Abstraction**



```java
public class Employee {
   private String name;
   private String address;
   private int number;

   public Employee(String name, String address, int number) {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }

   public void mailCheck() {
      System.out.println("Mailing a check to " + this.name + " " + this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
   }

   public String getName() {
      return name;
   }

   public String getAddress() {
      return address;
   }

   public void setAddress(String newAddress) {
      address = newAddress;
   }

   public int getNumber() {
      return number;
   }
}
```



```java
public class Salary extends Employee {
   private double salary; // Annual salary

   public Salary(String name, String address, int number, double salary) {
      super(name, address, number);
      setSalary(salary);
   }

   public void mailCheck() {
      System.out.println("Within mailCheck of Salary class ");
      System.out.println("Mailing check to " + getName()
      + " with salary " + salary);
   }

   public double getSalary() {
      return salary;
   }

   public void setSalary(double newSalary) {
      if(newSalary >= 0.0) {
         salary = newSalary;
      }
   }

   public double computePay() {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

Good Class Interfaces: **Good Abstraction**

The pursuit of good abstract interfaces gives rise to several guidelines for creating class interfaces.

Present a consistent level of abstraction in the class interface

Be sure you understand what abstraction the class is implementing

Provide services in pairs with their opposites

Move unrelated information to another class

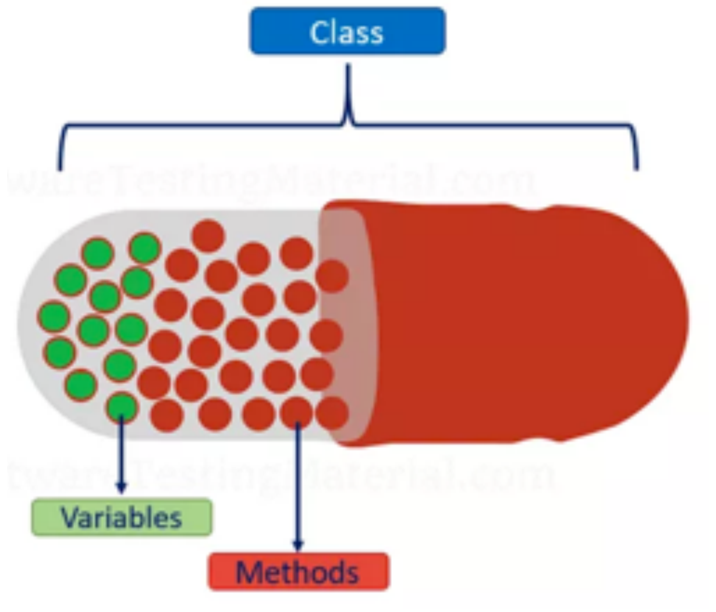
Make interfaces programmatic rather than semantic when possible

Beware of erosion of the interface's abstraction under modification

Don't add public members that are inconsistent with the interface abstraction

Consider abstraction and cohesion together

Good Class Interfaces: **Good Encapsulation**



Abstraction helps to *manage complexity* by providing models that allow you to ignore implementation details. **Encapsulation** is the enforcer that *prevents you from looking at the details* even if you want to

Without encapsulation, abstraction tends to break down

Good Class Interfaces: **Good Encapsulation**

Minimize accessibility of classes and members

If you're wondering whether a specific routine should be public, private, or protected, one school of thought is that **you should favor the strictest level of privacy that's workable**

Meyers 1998, Bloch 2001

If exposing the routine is consistent with the abstraction, it's probably fine to expose it. If you're not sure, **hiding more is generally better than hiding less.**

Good Class Interfaces: **Good Encapsulation**

Don’t expose member data in public

Exposing member data is a violation of encapsulation and limits your control over the abstraction

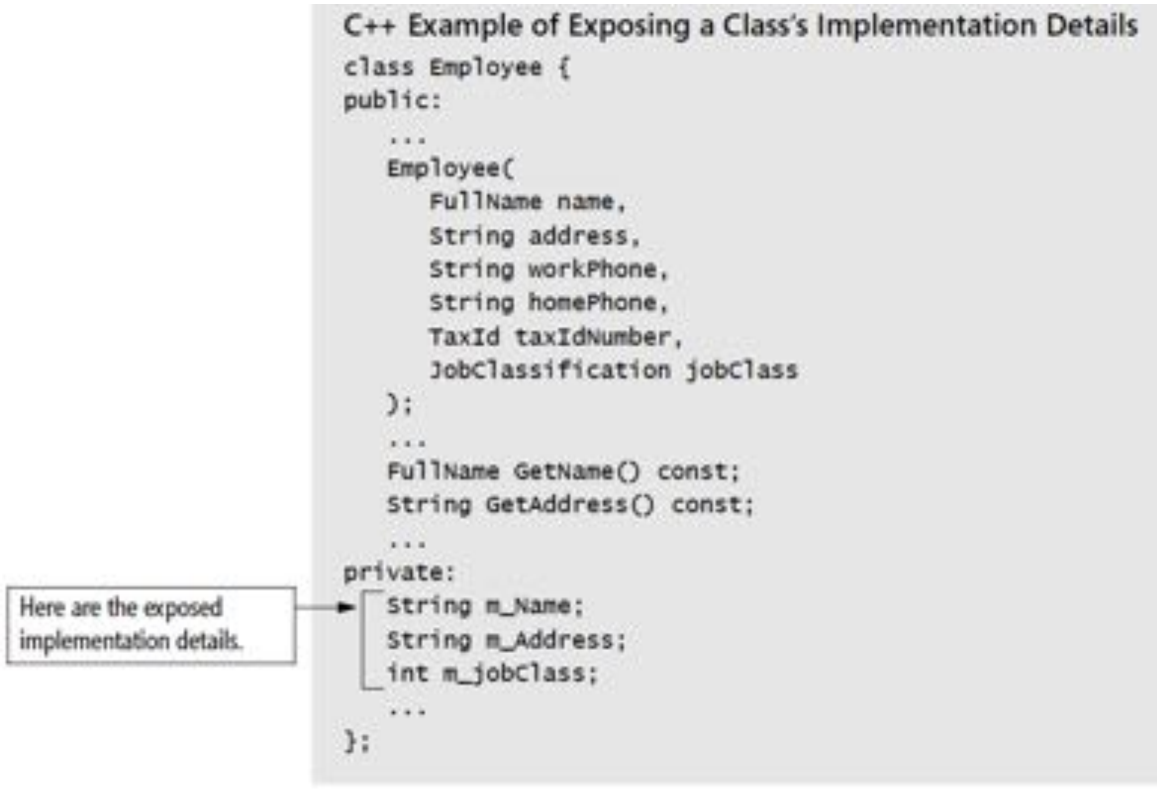
```
float x;
float y;
float z;
```

```
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

Good Class Interfaces: **Good Encapsulation**

Avoid putting private implementation details into a class's interface

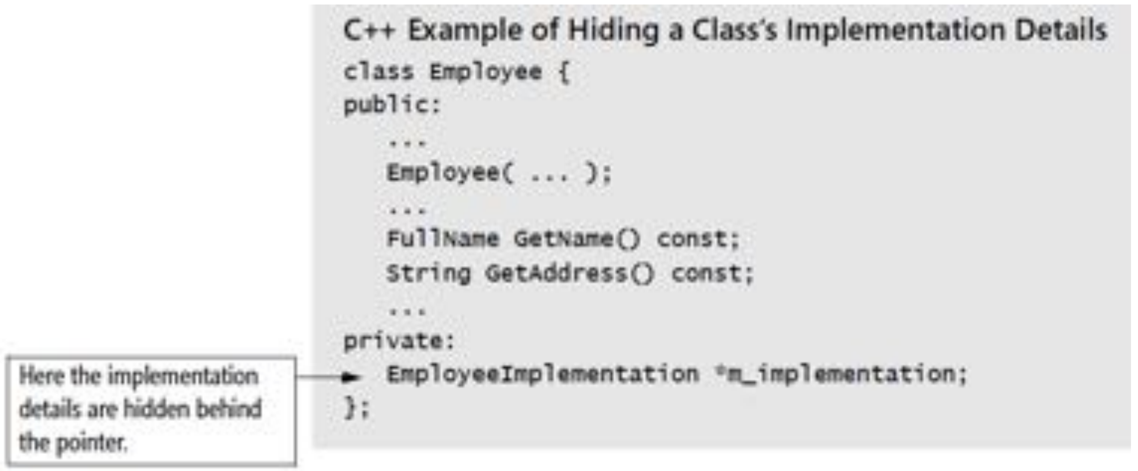
With true encapsulation, programmers would not be able to see implementation details at all



C++ Example of Exposing a Class's Implementation Details

```
class Employee {
public:
   ...
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:
   String m_Name;
   String m_Address;
   int m_jobClass;
   ...
};
```

Here are the exposed implementation details.



C++ Example of Hiding a Class's Implementation Details

```
class Employee {
public:
   ...
   Employee( ... );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:
   EmployeeImplementation *m_implementation;
};
```

Here the implementation details are hidden behind the pointer.

Good Class Interfaces: **Good Encapsulation**

Don’t make assumptions about the class’s users

A class should be designed and implemented to adhere to the contract implied by the class interface. It shouldn’t make any assumptions about how that interface will or won’t be used,

```
// initialize x, y, and z to 1.0 because DerivedClass blows
// up if they're initialized to 0.0
```

Good Class Interfaces: **Good Encapsulation**

Favor read-time convenience to write-time convenience

Code is read far more times than it's written, even during initial development

Favoring a technique that speeds write-time convenience at the expense of read-time convenience is a false economy.

Good Class Interfaces: **Good Encapsulation**

Be very, very wary of semantic violations of encapsulation

The difficulty of semantic encapsulation compared to syntactic encapsulation is similar.

Not calling Class A’s InitializeOperations() routine because you know that Class A’s PerformFirstOperation() routine calls it automatically.

Not calling the database.Connect() routine before you call employee.Retrieve(database) because you know that the employee.Retrieve() function will connect to the database if there isn’t already a connection.

Not calling Class A’s Terminate() routine because you know that Class A’s PerformFinalOperation() routine has already called it.

Using a pointer or reference to ObjectB created by ObjectA even after ObjectA has gone out of scope, because you know that ObjectA keeps ObjectB in static storage and ObjectB will still be valid.

Using Class B’s MAXIMUM_ELEMENTS constant instead of using ClassA.MAXIMUM_ELEMENTS , because you know that they’re both equal to the same value.

Good Class Interfaces: **Good Encapsulation**

Watch for coupling that's too tight

In general, the looser the connection, the better

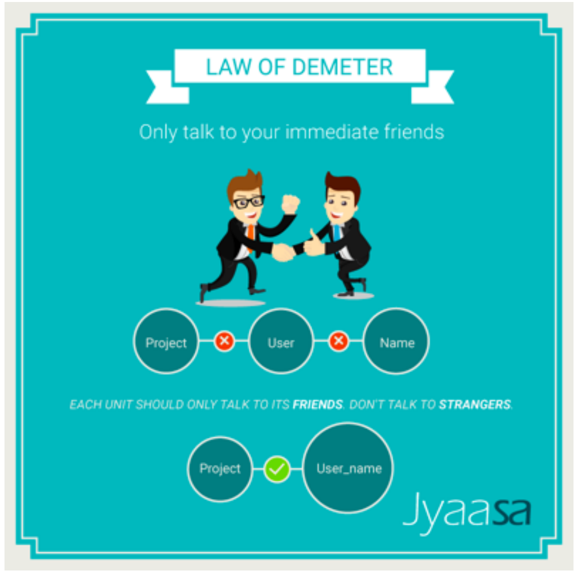
Minimize accessibility of classes and members.

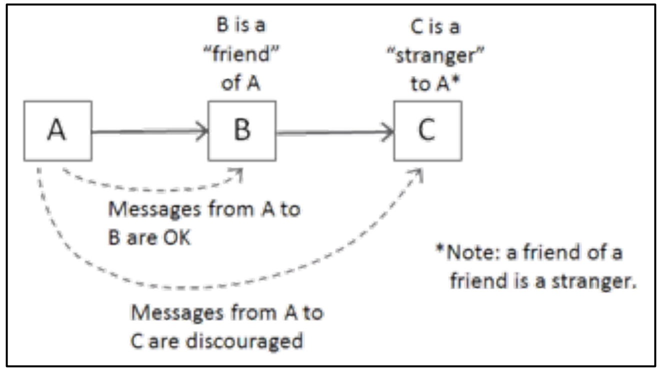
Make data private rather than protected in a base class to make derived classes less tightly coupled to the base class.

Avoid exposing member data in a class's public interface

Be wary of semantic violations of encapsulation

Observe the "Law of Demeter"


LAW OF DEMETER
Only talk to your immediate friends
Project
User
Name
EACH UNIT SHOULD ONLY TALK TO ITS FRIENDS. DON'T TALK TO STRANGERS.
Project
User_name
Jyaasa


B is a "friend" of A
C is a "stranger" to A*
A
B
C
Messages from A to B are OK
Messages from A to C are discouraged
*Note: a friend of a friend is a stranger.

Design and Implementation Issues

Defining good class interfaces goes a long way toward creating a high-quality program.

Design and Implementation Issues

Containment ("has a" Relationships)

Containment is the simple idea that a class contains a primitive data element or object. Inheritance is more popular than containment, not because it's better.

Implement "has a" through containment

An employee "has a" name, "has a" phone number, "has a" tax ID. You can usually accomplish this by making the name, phone number, and tax ID member data of the Employee class.

Implement "has a" through private inheritance as a last resort

In some instances you might find that you can't achieve containment through making one object a member of another

Be critical of classes that contain more than about seven data members

The number "7±2" has been found to be a number of discrete items a person can remember while performing other tasks

Design and Implementation Issues

Inheritance ("is a" Relationships)

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes.

For each member routine, will the routine be visible to derived classes? Will it have a default implementation? Will the default implementation be overridable?

For each data member (including variables, named constants, enumerations, and so on), will the data member be visible to derived classes?

Implement "is a" through public inheritance

When a programmer decides to create a new class by inheriting from an existing class, that programmer is saying that the new class "is a" more specialized version of the older class.

If the derived class isn't going to adhere completely to the same interface contract defined by the base class, inheritance is not the right implementation technique. Consider containment or making a change further up the inheritance hierarchy.

Design and Implementation Issues

Inheritance (“is a” Relationships)

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes.

Design and document for inheritance or prohibit it

Inheritance adds complexity to a program, and, as such, it’s a dangerous technique

If a class isn’t designed to be inherited from, make its members non-virtual in C++, final in Java, or non-overridable in Microsoft Visual Basic so that you can’t inherit from it.

Design and Implementation Issues

Inheritance (“is a” Relationships)

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes.

Adhere to the Liskov Substitution Principle (LSP)

Barbara Liskov argued that you shouldn’t inherit from a base class unless the derived class truly “is a” more specific version of the base class

Subclasses must be usable through the base class interface without the need for the user to know the difference. In other words, all the routines defined in the base class should mean the same thing when they’re used in each of the derived classes.

If you have a base class of **Account** and derived classes of **CheckingAccount**, **SavingsAccount**, and **AutoLoanAccount**, a programmer should be able to invoke any of the routines derived from **Account** on any of **Account**'s subtypes without caring about which subtype a specific account object is the derived classes.

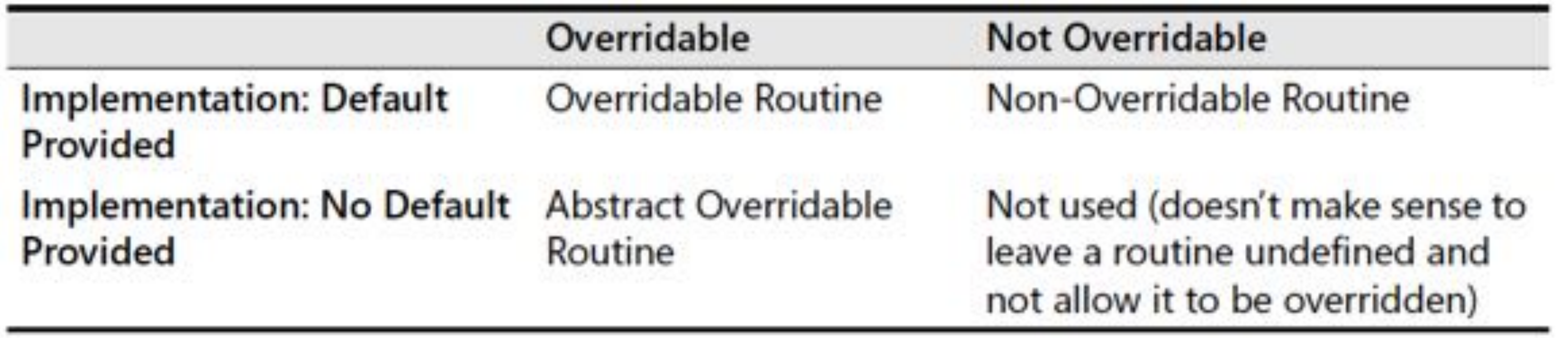
Design and Implementation Issues

Inheritance (“is a” Relationships)

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes.

Be sure to inherit only what you want to inherit

A derived class can inherit member routine interfaces, implementations, or both.



	Overridable	Not Overridable
Implementation: Default Provided	Overridable Routine	Non-Overridable Routine
Implementation: No Default Provided	Abstract Overridable Routine	Not used (doesn't make sense to leave a routine undefined and not allow it to be overridden)

An abstract overridable routine means that the derived class inherits the routine’s interface but not its implementation.

An overridable routine means that the derived class inherits the routine’s interface and a default implementation and it is allowed to override the default implementation

A non-overridable routine means that the derived class inherits the routine’s interface and its default implementation and it is not allowed to override the routine’s implementation.

Design and Implementation Issues

Inheritance ("is a" Relationships)

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes.

Don't "override" a non-overridable member function

Move common interfaces, data, and behavior as high as possible in the inheritance tree

Be suspicious of base classes of which there is only one derived class

Be suspicious of classes that override a routine and do nothing inside the derived routine

Avoid deep inheritance trees

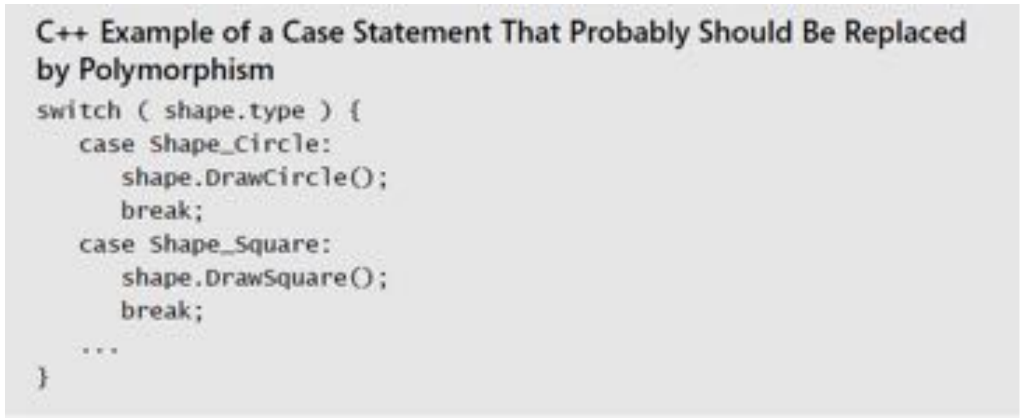
Make all data private, not protected

Design and Implementation Issues

Inheritance (“is a” Relationships)

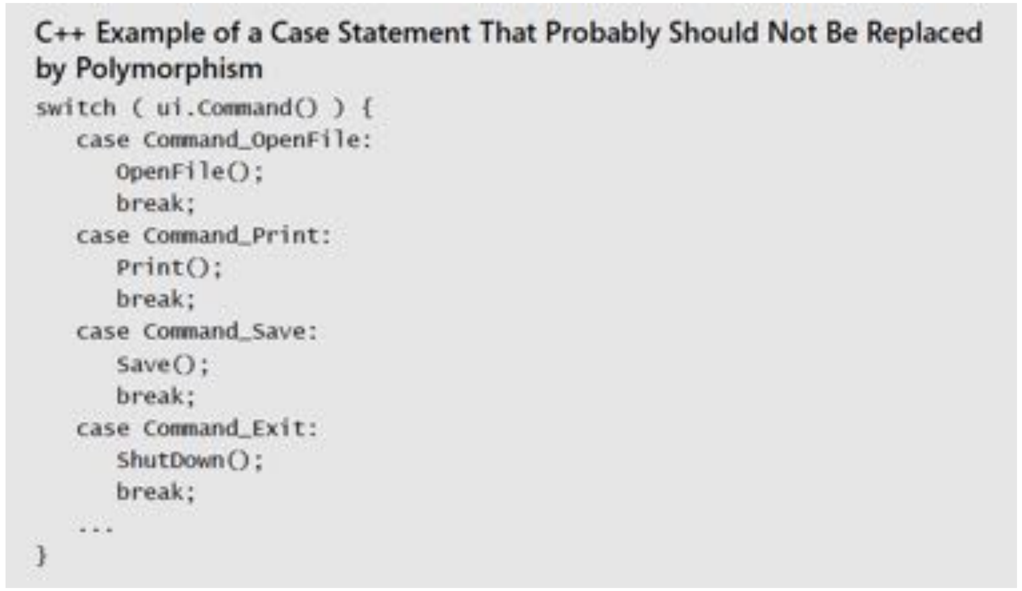
Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes.

Prefer polymorphism to extensive type checking



C++ Example of a Case Statement That Probably Should Be Replaced by Polymorphism

```
switch ( shape.type ) {
   case Shape_Circle:
      shape.DrawCircle();
      break;
   case Shape_Square:
      shape.DrawSquare();
      break;
   ...
}
```



C++ Example of a Case Statement That Probably Should Not Be Replaced by Polymorphism

```
switch ( ui.Command() ) {
   case Command_OpenFile:
      OpenFile();
      break;
   case Command_Print:
      Print();
      break;
   case Command_Save:
      Save();
      break;
   case Command_Exit:
      ShutDown();
      break;
   ...
}
```

Design and Implementation Issues

Inheritance (“is a” Relationships)

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes.

Why Are There So Many Rules for Inheritance?

If multiple classes share common data but not behavior, create a common object that those classes can contain.

If multiple classes share common behavior but not data, derive them from a common base class that defines the common routines

If multiple classes share common data and behavior, inherit from a common base class that defines the common data and routines.

Inherit when you want the base class to control your interface; contain when you want to control your interface.

Design and Implementation Issues

Member Functions and Data

Keep the number of routines in a class as small as possible

Minimize indirect routine calls to other classes

Initialize all member data in all constructors, if possible

Prefer deep copies to shallow copies until proven otherwise

A deep copy of an object is a member-wise copy of the object's member data; a shallow copy typically just points to or refers to a single reference copy

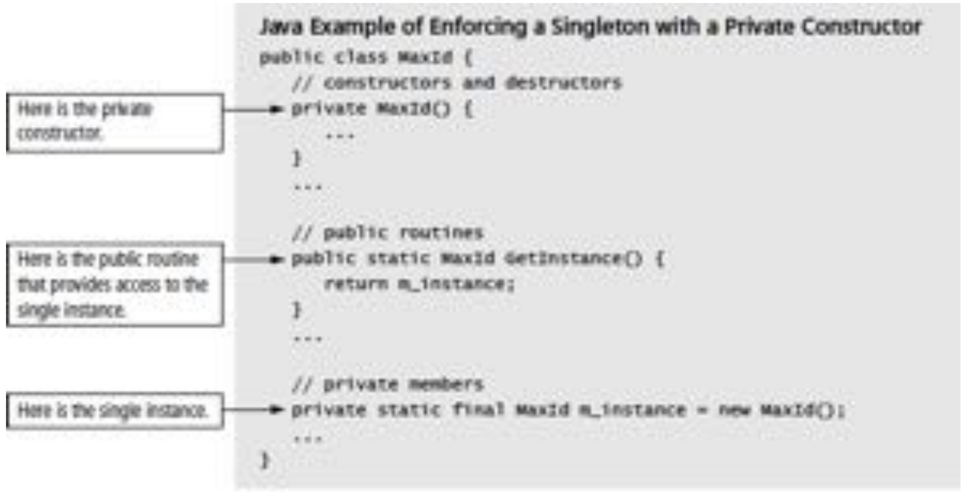
Disallow implicitly generated member functions and operators you don't want

In general, minimize the extent to which a class collaborates with other classes

Minimize the number of different routines called by a class

One study found that the number of faults in a class was statistically correlated with the total number of routines that were called from within a class

Enforce the singleton property by using a private constructor



Java Example of Enforcing a Singleton with a Private Constructor

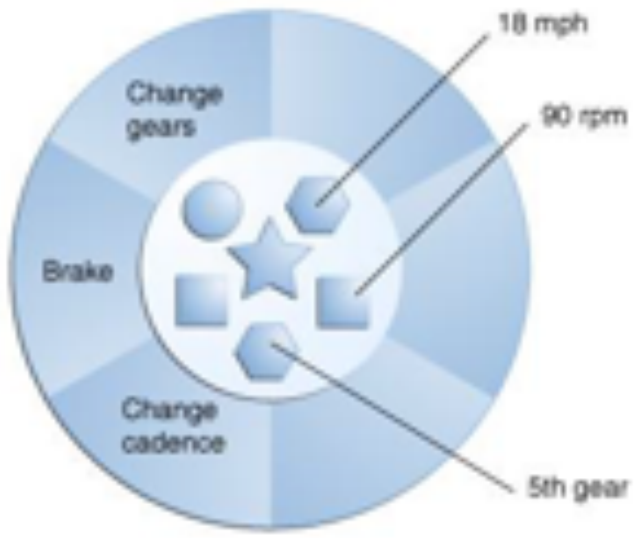
```
public class MaxId {
   // constructors and destructors
   private MaxId() {
      ...
   }
   ...

   // public routines
   public static MaxId GetInstance() {
      return m_instance;
   }
   ...

   // private members
   private static final MaxId m_instance = new MaxId();
   ...
}
```

Here is the private constructor.
Here is the public routine that provides access to the single instance.
Here is the single instance.

Reasons to Create a Class

Model real-world objects

Create a class for each real-world object type that your program models

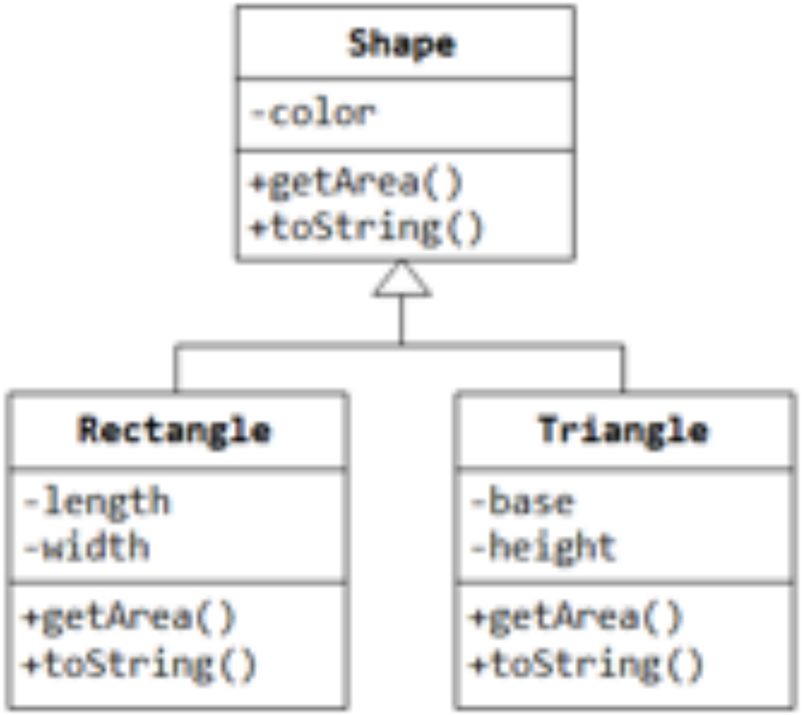


Reasons to Create a Class

Model abstract objects

An object that isn't a concrete, real-world object but that provides an abstraction of other concrete objects.

For example, the classic *Shape* object. *Rectangle* and *Triangle* really exist, but *Shape* is an abstraction of other specific shapes.



Reasons to Create a Class

Reduce complexity

Create a class to hide information so that you won't need to think about it, no need to know about its internal workings. Also, to minimize code size and improve maintainability

Reasons to Create a Class

Isolate complexity

Complexity in all forms—complicated algorithms, large data sets, intricate communications protocols, and so on—is prone to errors

If an error does occur, it will be easier to find if it isn't spread through the code but is localized within a class

Reasons to Create a Class

Hide implementation details

Limit effects of changes

Isolate areas that are likely to change so that the effects of changes are limited to the scope of a single class or a few classes

Hide global data

If you need to use global data, you can hide its implementation details behind a class interface. Working with global data through access routines provides several benefits compared to working with global data directly.

Streamline parameter passing

If you're passing a parameter among several routines, that might indicate a need to factor those routines into a class that share the parameter as object data

Make central points of control

It's a good idea to control each task in one place

Facilitate reusable code

Code put into well-factored classes can be reused in other programs more easily than the same code embedded in one larger class

Classes to Avoid

Avoid creating god classes, all-knowing and all-powerful

If a class spends its time retrieving data from other classes using *Get()* and *Set()* routines (that is, digging into their business and telling them what to do), ask whether that functionality might better be organized into those other classes rather than into the god class (Riel 1996).

Eliminate irrelevant classes

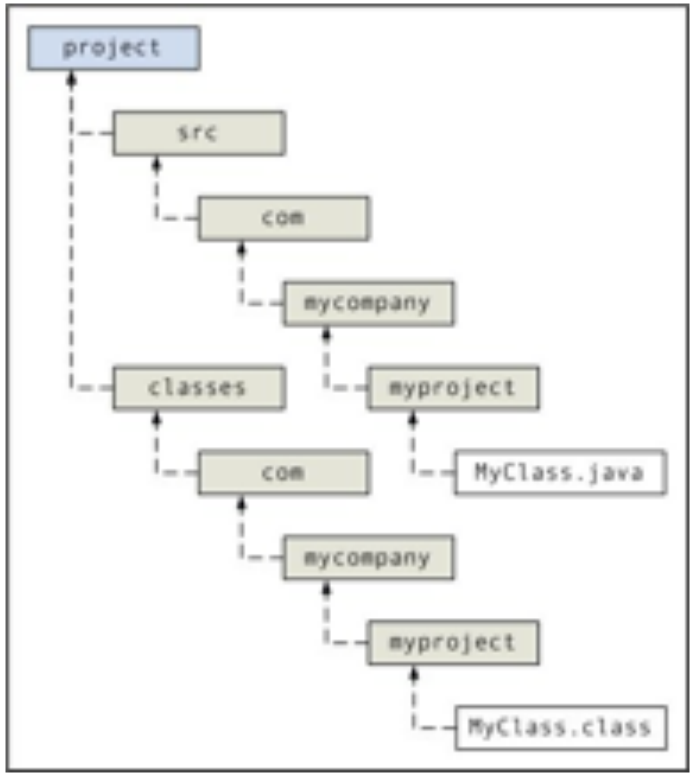
If a class consists only of data but no behavior, ask yourself whether it's really a class and consider demoting it so that its member data just becomes attributes of one or more other classes.

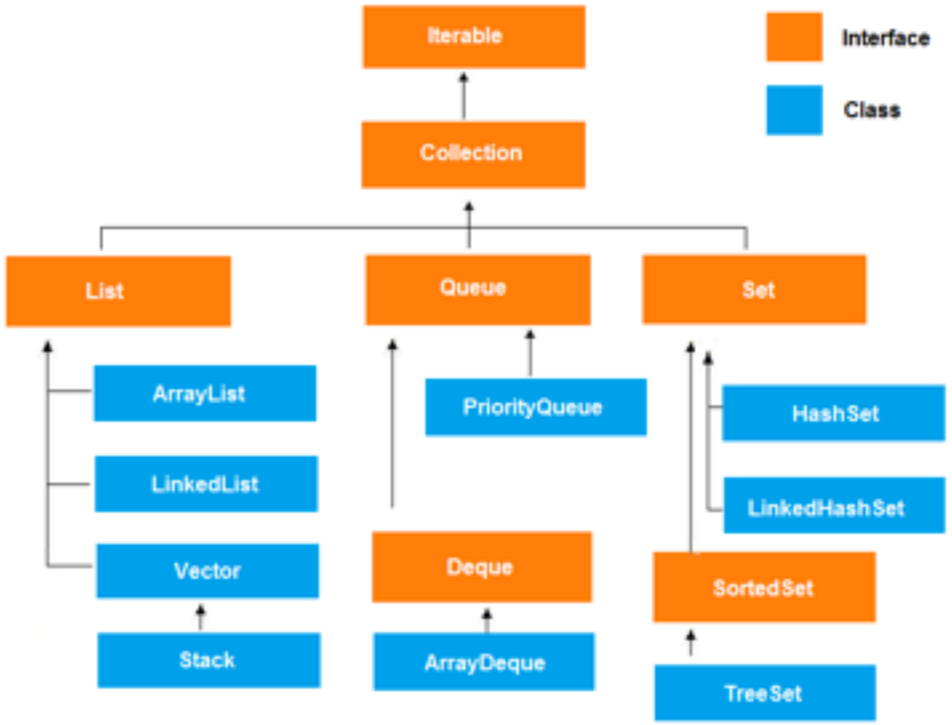
Avoid classes named after verbs

A class that has only behavior but no data is generally not really a class. Consider turning a class like *DatabaseInitialization()* or *StringBuilder()* into a routine on some other class.

Beyond Classes: **Packages**

Classes are currently the best way for programmers to achieve modularity. But modularity is a big topic, and it extends beyond classes.





What are the **right** and **wrong** things?

```java
//SportsCar class (SportsCar.java)
public class SportsCar extends Engine {

     public Car myCar = new Car(); //myCar attribute: Object initialization of Car

     public void activateSportMode(){ //Implementation }
     public void activateComfortMode(){ //Implementation }

     //Main function: required to run a Java program
     public static void main(String[] args){

          SportsCar sc = new SportsCar(); //Object initialization of SportsCar
          sc.myCar.brand = "Ferrari";
          sc.myCar.numberOfGears = 6;
          sc.engineMaxSpeed = 315;
          sc.enginePower = 552;
          sc.activateSportMode();
     }
}
```

```java
//Car class (Car.java)
public class Car {

     public String brand;
     public int productionYear;
     public int numberOfGears;
     public int numberOfSeats;

     public void start(){ //Implementation }
     public void stop(){ //Implementation }
     public void changeGear(){ //Implementation }
}
```

```java
//Engine class (Engine.java)
public class Engine {

     public int enginePower;
     public int engineMaxSpeed;
     private String engineType;
}
```

SUMMARY

- Class interfaces should provide a consistent abstraction. Many problems arise from violating this single principle.
- A class interface should hide something—a system interface, a design decision, or an implementation detail.
- Containment is usually preferable to inheritance unless you’re modeling an “is a” relationship.
- Inheritance is a useful tool, but it adds complexity, which is counter to Software’s Primary Technical Imperative of managing complexity.
- Classes are your primary tool for managing complexity. Give their design as much attention as needed to accomplish that objective.

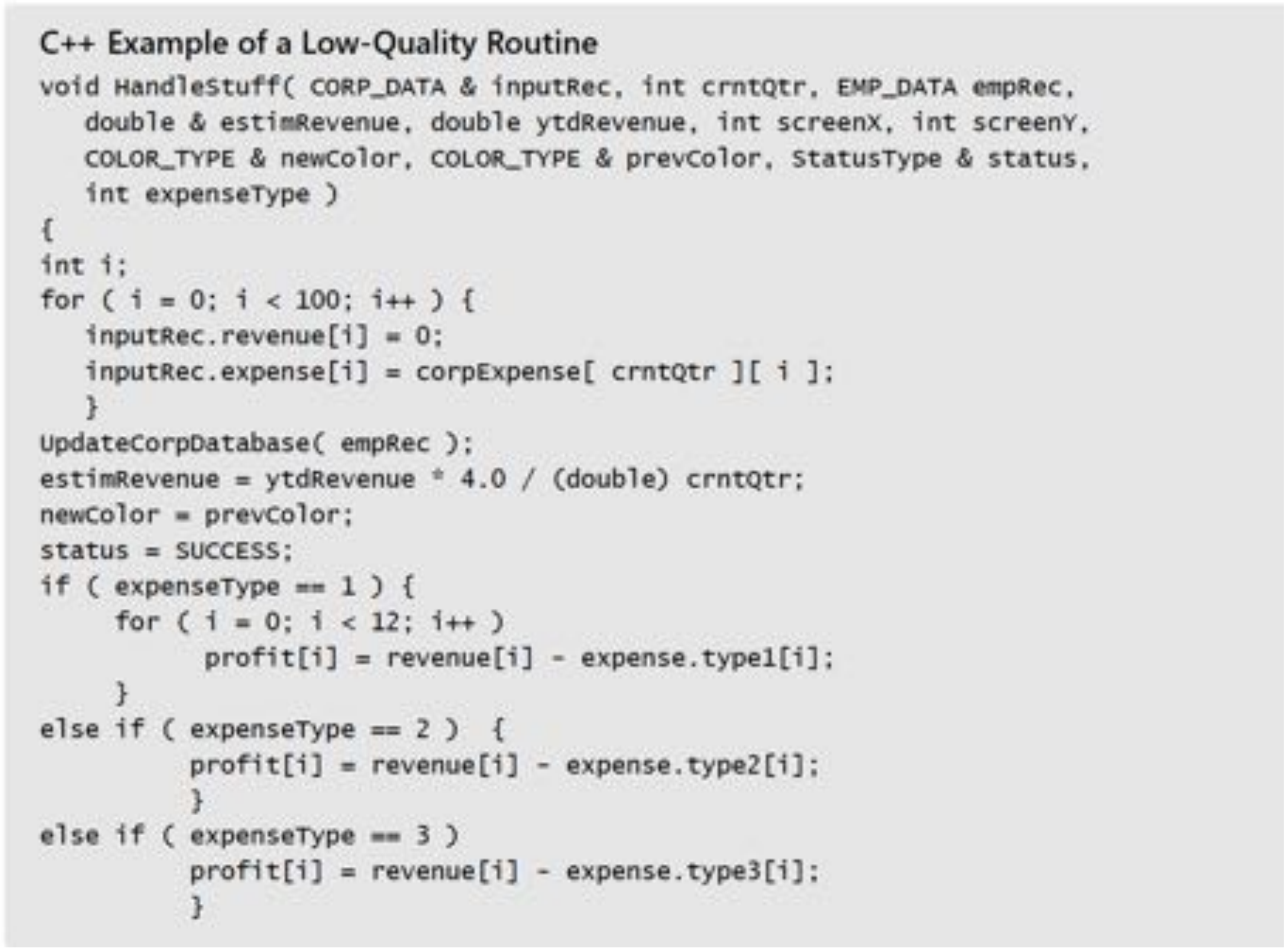
High Quality Routines

What is a routine?

A **routine** is an individual method or procedure invocable for a single purpose. Examples include a function in C++, a method in Java, a function or sub procedure in Microsoft Visual Basic

What is a high-quality routine? **That’s a harder question**.

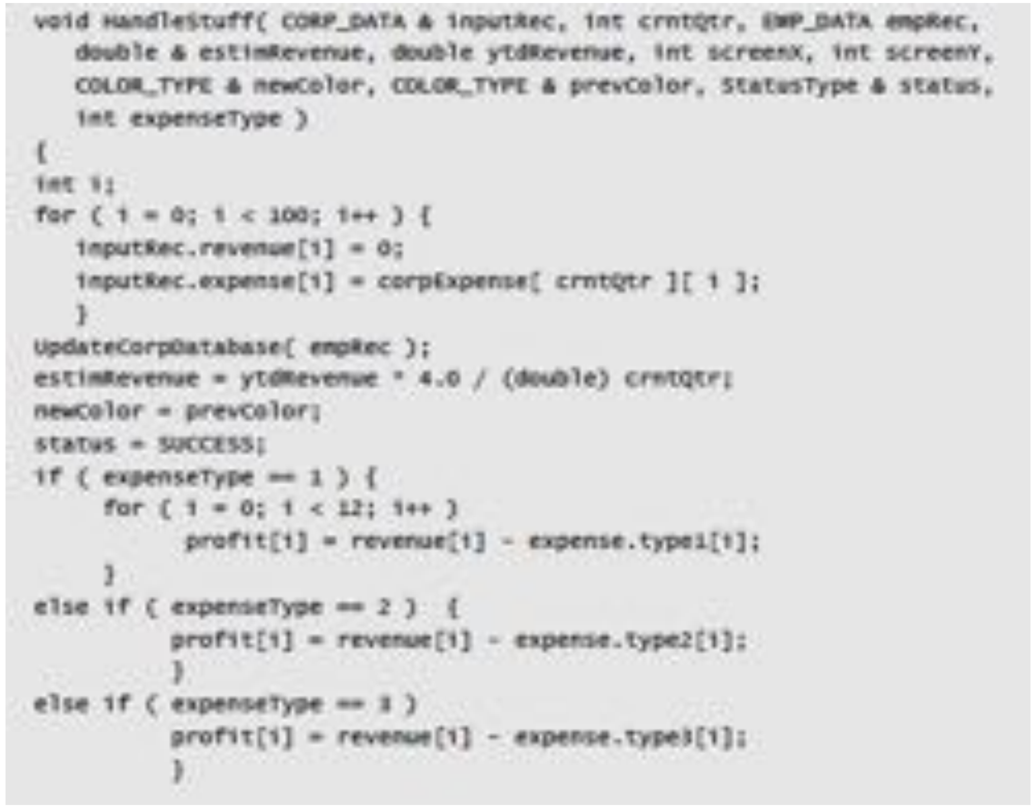
What is a high-quality routine?



C++ Example of a Low-Quality Routine

```
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 a high-quality routine?



```
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];
          }
```

The routine has a bad name. `HandleStuff()` tells you nothing about what the routine does.

The routine isn't documented.

The routine has a bad layout. The physical organization of the code on the page gives few hints about its logical organization.

The routine's input variable, `inputRec`, is changed. If it's an input variable, its value should not be modified

The routine reads and writes global variables—it reads from `corpExpense` and writes to `profit` . It should communicate with other routines more directly than by reading and writing global variables.

The routine doesn't have a single purpose. It initializes some variables, writes to a database, does some calculations—none of which seem to be related to each other in any way. A routine should have a single, clearly defined purpose.

The routine doesn't defend itself against bad data. If `crntQtr` equals 0 , the expression `ytdRevenue` * 4.0 / (double) `crntQtr` causes a divide-by-zero error.

The routine uses several magic numbers: 100 , 4.0, 12 , 2 , and 3 .

Some of the routine's parameters are unused: `screenX` and `screenY` are not referenced within the routine.

The routine has too many parameters. The upper limit for an understandable number of parameters is about 7

The routine's parameters are poorly ordered and are not documented

One of the routine's parameters is passed incorrectly: `prevColor` is labeled as a reference parameter (&) even though it isn't assigned a value within the routine.

Valid Reasons to Create a Routine

Reduce Complexity

The single most important reason to create a routine is to reduce a program's complexity. Create a routine to hide information so that you won't need to think about it.

Other reasons to create routines: minimizing code size and improving maintainability and correctness

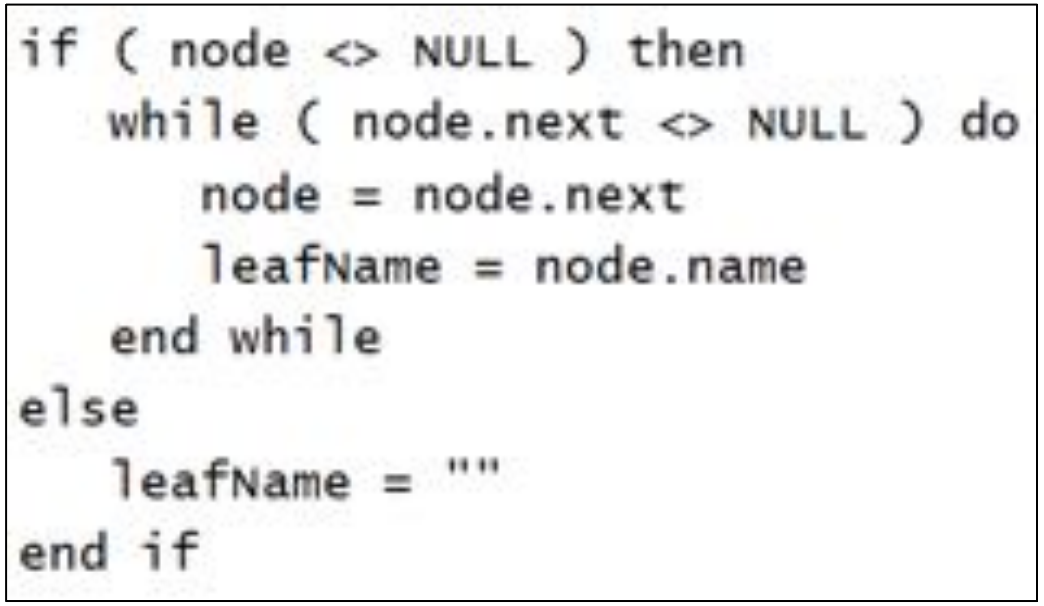
But without the abstractive power of routines, complex programs would be impossible to manage.

An indication that a routine needs to be broken out is loop deep nesting or a conditional

Valid Reasons to Create a Routine

Introduce an intermediate, understandable abstraction

Putting a section of code into a well-named routine is one of the best ways to document its purpose



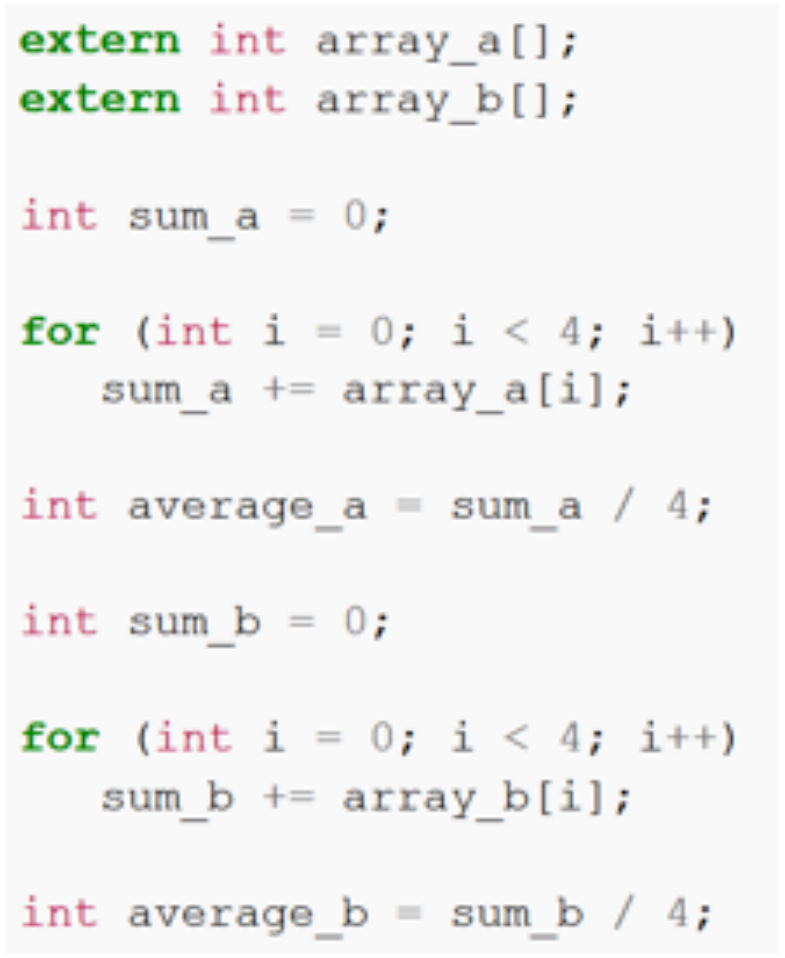
```
if ( node <> NULL ) then
   while ( node.next <> NULL ) do
      node = node.next
      leafName = node.name
   end while
else
   leafName = ""
end if
```

```
leafName = GetLeafName( node )
```

Valid Reasons to Create a Routine

Avoid duplicate code

Undoubtedly the most popular reason for creating a routine is to avoid duplicate code.



```c
extern int array_a[];
extern int array_b[];

int sum_a = 0;

for (int i = 0; i < 4; i++)
   sum_a += array_a[i];

int average_a = sum_a / 4;

int sum_b = 0;

for (int i = 0; i < 4; i++)
   sum_b += array_b[i];

int average_b = sum_b / 4;
```

Valid Reasons to Create a Routine

Hide Sequences

It's a good idea to hide the order in which events happen to be processed

For example, a sequence might be found when you have two lines of code that read the top of a stack and decrement a *stackTop*variable.

Put those two lines of code into a *PopStack()* routine to hide the assumption about the order in which the two operations must be performed

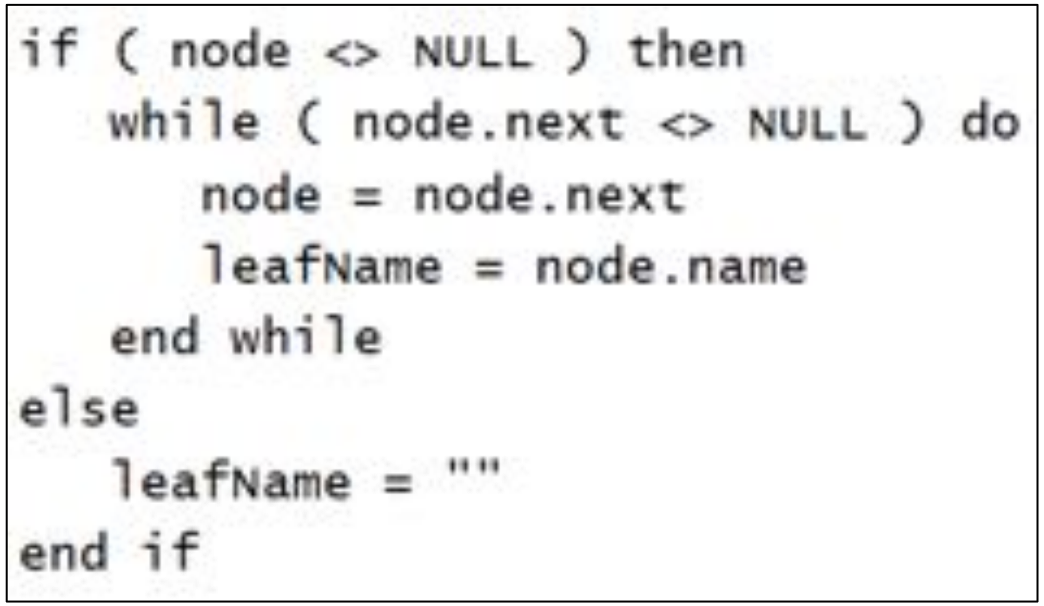
Hiding that assumption will be better than baking it into code from one end of the system to the other.

Valid Reasons to Create a Routine

Hide Pointer Operations

Pointer operations tend to be hard to read and error prone. By isolating them in routines, you can concentrate on the intent of the operation rather than on the mechanics of pointer manipulation

if the operations are done in only one place, you can be more certain that the code is correct. If you find a better data type than pointers, you can change the program without traumatizing the code that would have used the pointers.



```
if ( node <> NULL ) then
   while ( node.next <> NULL ) do
      node = node.next
      leafName = node.name
   end while
else
   leafName = ""
end if
```

Valid Reasons to Create a Routine

Improve portability

Use of routines isolates nonportable capabilities, explicitly identifying and isolating future portability work.

Nonportable capabilities include nonstandard language features, hardware dependencies, operating-system dependencies, and so on.

Valid Reasons to Create a Routine

Simplify Complicated Boolean Tests

Understanding complicated boolean tests in detail is rarely necessary for understanding program flow.

Putting such a test into a function makes the code more readable because (1) the details of the test are out of the way and (2) a descriptive function name summarizes the purpose of the test.

Giving the test a function of its own emphasizes its significance. It encourages extra effort to make the details of the test readable inside its function.

Valid Reasons to Create a Routine

Improve Performance

You can optimize the code in one place instead of in several places.

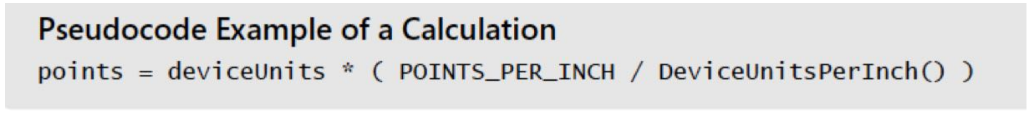
Centralizing code into a routine means that a single optimization benefits all the code that uses that routine, whether it uses it directly or indirectly.

Having code in one place makes it practical to recode the routine with a more efficient algorithm or in a faster, more efficient language.

Operations That Seem Too Simple to Put Into Routines

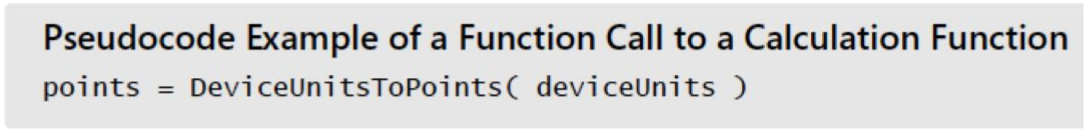
Constructing a whole routine to contain two or three lines of code might seem like overkill, but experience shows how helpful a good small routine can be.

Small routines offer several advantages. One is that they improve readability.



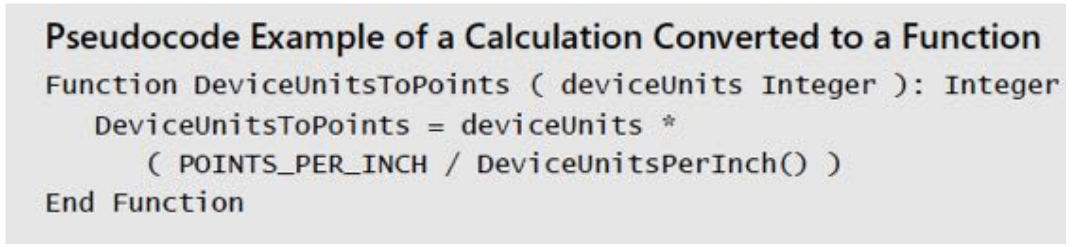
Pseudocode Example of a Calculation

```
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```



Pseudocode Example of a Function Call to a Calculation Function

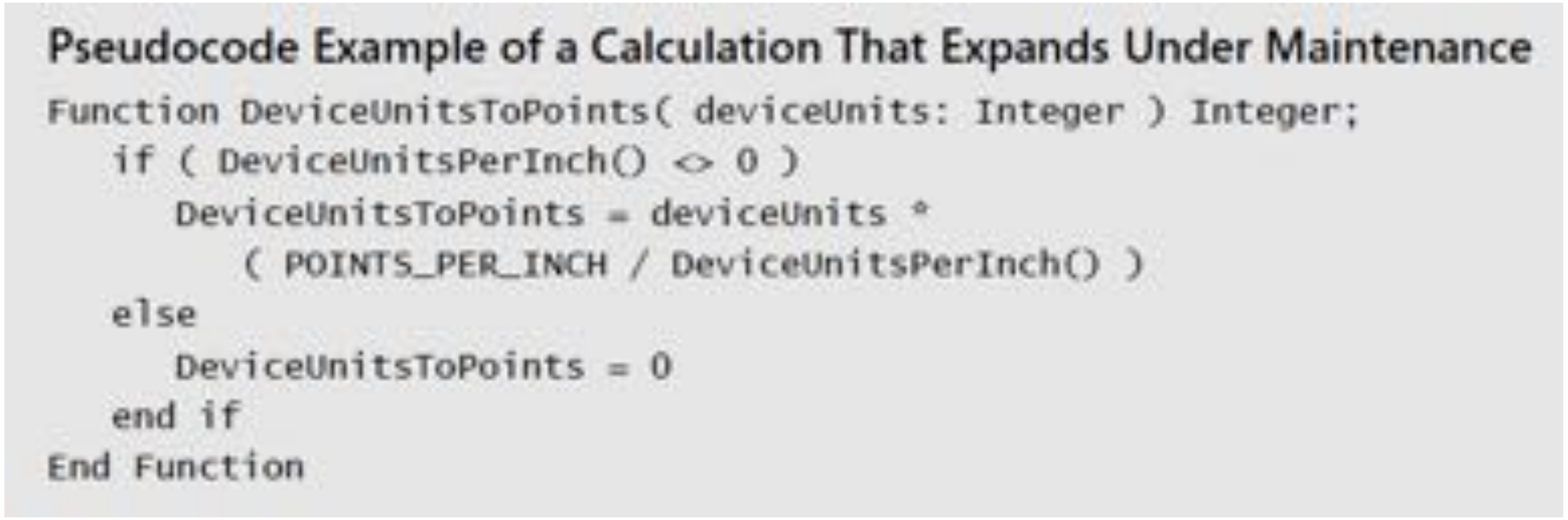
```
points = DeviceUnitsToPoints( deviceUnits )
```



Pseudocode Example of a Calculation Converted to a Function

```
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
   DeviceUnitsToPoints = deviceUnits *
      ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

Operations That Seem Too Simple to Put Into Routines



Pseudocode Example of a Calculation That Expands Under Maintenance

```
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;
   if ( DeviceUnitsPerInch() <> 0 )
      DeviceUnitsToPoints = deviceUnits *
         ( POINTS_PER_INCH / DeviceUnitsPerInch() )
   else
      DeviceUnitsToPoints = 0
   end if
End Function
```

If that original line of code had still been in a dozen places, the test would have been repeated a dozen times, for a total of 36 new lines of code. A simple routine reduced the 36 new lines to 3.

Design at the Routine Level

Cohesion

how closely the operations in a routine are related

Coupling

the relationships between functions

Design at the Routine Level: **Cohesion**

Some programmers prefer the term "strength"; how strongly related are the operations in a routine

`Cosine()`

A function like *Cosine()* is perfectly cohesive because the whole routine is dedicated to performing one function.

`CosineAndTan()`

A function like CosineAndTan() has lower cohesion because it tries to do more than one thing. The goal is to have each routine do one thing well and not do anything else.

One study of 450 routines found that 50 percent of the highly cohesive routines were fault free, whereas only 18 percent of routines with low cohesion were fault free
(Card, Church, and Agresti1986)

Another study of a different 450 routines (which is just an unusual coincidence) found that routines with the highest coupling-to-cohesion ratios had 7 times as many errors as those with the lowest coupling-to-cohesion ratios and were 20 times as costly to fix
(Selby and Basili1991)

Design at the Routine Level: *Desired* **Cohesion**

Functional Cohesion

Functional cohesion is the strongest and best kind of cohesion, occurring when a routine performs one and only one operation

- Compute Cosine of Angle
- Verify Alphabetic Syntax
- Read Transaction Record
- Determine Customer Mortgage Repayment
- Compute Point of Impact of Missile
- Calculate Net Employee Salary
- Assign Seat to Airline Customer

Design at the Routine Level: *Acceptable* **Cohesion**

Sequential Cohesion

Sequential cohesion exists when a routine contains operations that must be performed in a specific order, that share data from step to step, and that don't make up a complete function when done together.

For example, given a birth date, calculates an employee's age and time to retirement.

If the routine calculates the age and then uses that result to calculate the employee's time to retirement, it has sequential cohesion.

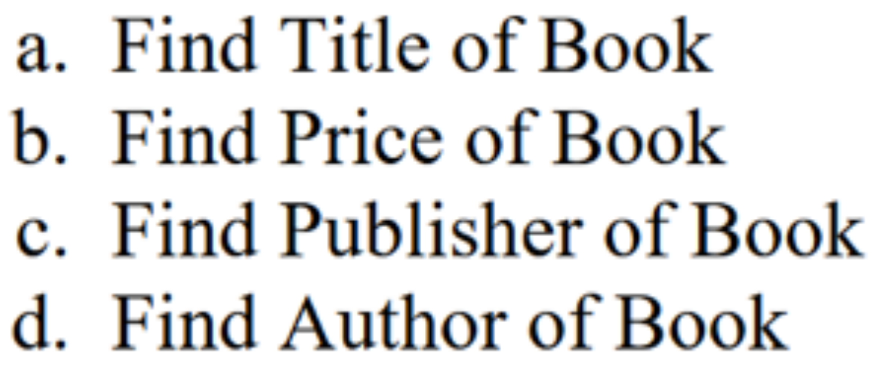
Design at the Routine Level: *Acceptable* **Cohesion**

Communicational Cohesion

Communicational cohesion occurs when operations in a routine make use of the same data and aren't related in any other way.

For example, suppose you wrote a function to query a database to get the name and office number for an employee in your company.

It may make sense for your application, but the only common point between the two operations is that the data comes from the same employee record.



a. Find Title of Book
b. Find Price of Book
c. Find Publisher of Book
d. Find Author of Book

Design at the Routine Level: *Acceptable* **Cohesion**

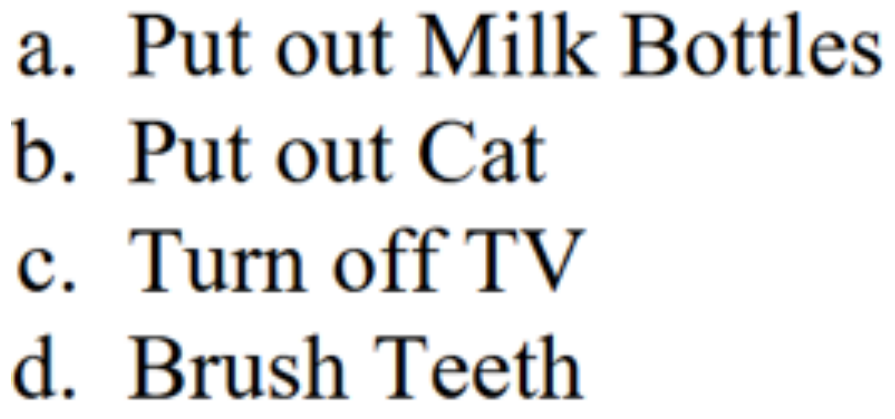
Temporal Cohesion

Temporal cohesion occurs when operations are combined into a routine because they are all done at the same time.

Some programmers consider temporal cohesion to be unacceptable because it’s sometimes associated with bad programming practices such as having a mixture of dissimilar code in a Startup() routine.

To avoid this problem, think of temporal routines as organizers of other events.

have the *temporally cohesive* routine call other routines to perform specific activities rather than performing the operations directly itself. **But this raises the issue of *choosing a name that describes the routine at the right level of abstraction***



a. Put out Milk Bottles
b. Put out Cat
c. Turn off TV
d. Brush Teeth

It will be clear that the point of the routine is to orchestrate activities rather than to do them directly.

Design at the Routine Level: *Unacceptable* **Cohesion**

Procedural Cohesion

Procedural cohesion occurs when operations in a routine are done in a specified order.

The routine has procedural cohesion because it puts a set of operations in a specified order and the operations don't need to be combined for any other reason.

To achieve better cohesion, put the separate operations into their own routines.

- Clean Utensils from Previous Meal
- Prepare Chicken for Roasting
- Make Phone Call
- Take Shower
- Chop Vegetables
- Set Table

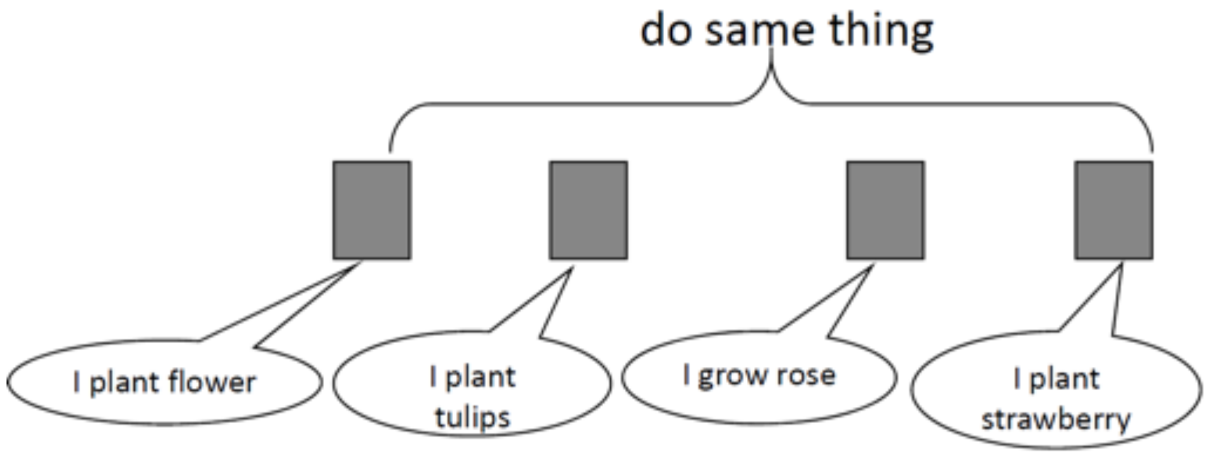
Design at the Routine Level: *Unacceptable* **Cohesion**

Logical Cohesion

Logical cohesion occurs when several operations are stuffed into the same routine and one of the operations is selected by a control flag that's passed in.

The control flow or "logic" of the routine is the only thing that ties the operations together—they're all in a big if statement or case statement together.

```
public void sample( int flag ) {
   switch ( flag ) {
      case ON:
         // bunch of on stuff
         break;
      case OFF:
         // bunch of off stuff
         break;
      case CLOSE:
         // bunch of close stuff
         break;
      case COLOR:
         // bunch of color stuff
         break;
   }
}
```



Design at the Routine Level: *Unacceptable* **Cohesion**

Logical Cohesion

Logical cohesion occurs when several operations are stuffed into the same routine and one of the operations is selected by a control flag that's passed in.

It's usually all right, to create a logically cohesive routine if its code consists solely of a series of if or case statements and calls to other routines.

if the routine's only function is to dispatch commands and it doesn't do any of the processing itself, that's usually a good design.

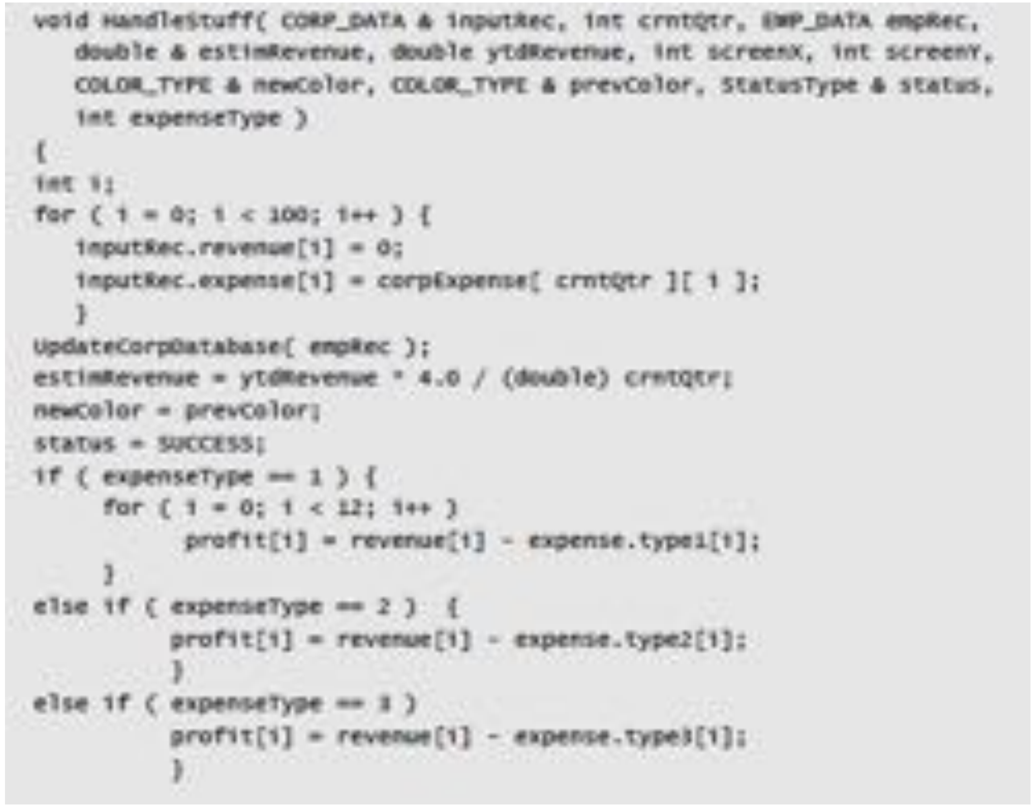
*The technical term for this kind of routine is "**event handler**"* An event handler is often used in interactive environments such as the Windows and Linux GUI environments.

Design at the Routine Level: *Unacceptable* **Cohesion**

Coincidental Cohesion

Coincidental cohesion occurs when the operations in a routine have no discernible relationship to each other

It’s hard to convert coincidental cohesion to any better kind of cohesion—you usually need to do a deeper redesign and reimplementation



```
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];
          }
```

- Fix Car
- Bake Cake
- Walk Dog
- Fill our Astronaut-Application Form
- Get out of Bed
- Go the the Movies

Design at the Routine Level: *Bad* **Coupling**

Tight Coupling

Large dependence on the structure of one module by another.

Design at the Routine Level: *Good* **Coupling**

Loose Coupling

Modules with loose coupling are more independent and easier to maintain

Design at the Routine Level: *Worst* **Coupling**

Content Coupling

A module changes another module's data

Design at the Routine Level: *Not Worst* **Coupling**

Common Coupling

This occurs when all modules reference the same global data structure

Design at the Routine Level: *Not Worst* **Coupling**

External Coupling

Modules communicate through an external medium, such as files

Design at the Routine Level: *Acceptable* **Coupling**

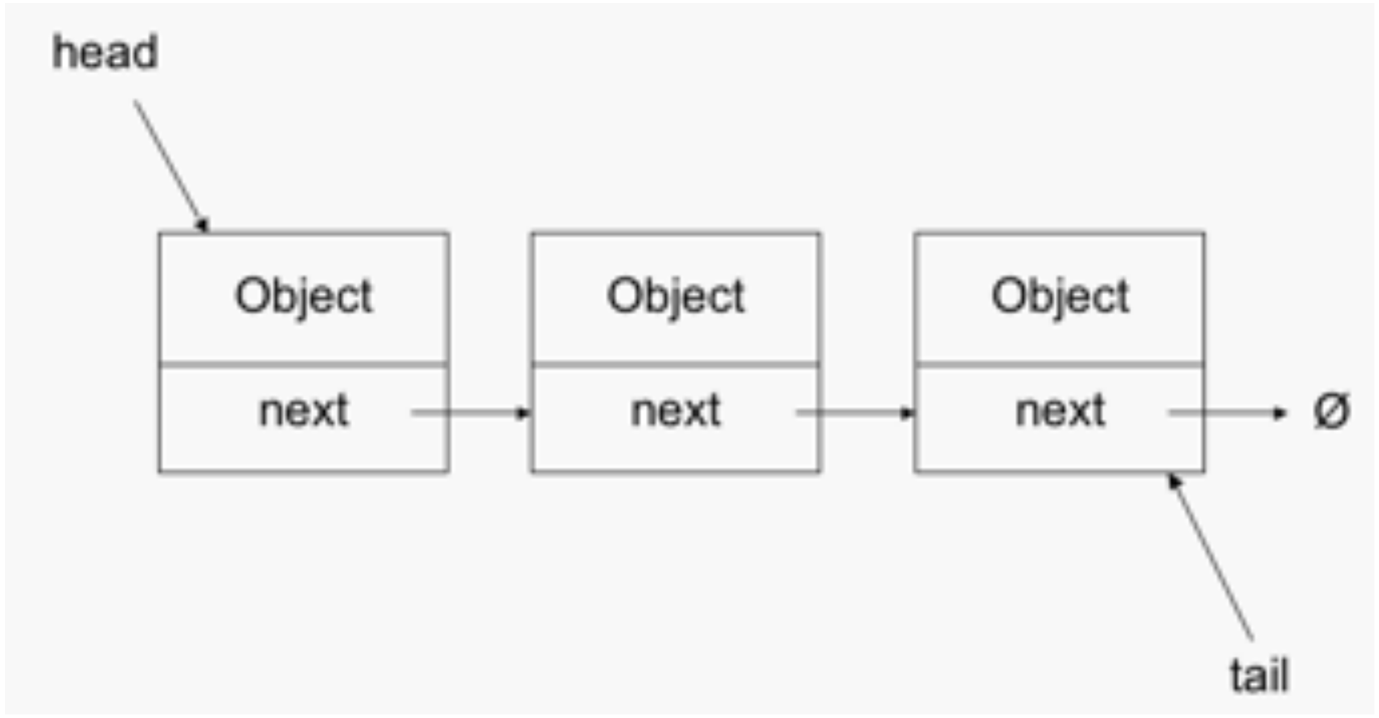
Control Coupling

Two modules exhibit control coupling if one (``module A'') passes to the other (``module B'') a piece of information that is intended to control the internal logic of the other.

Design at the Routine Level: *Acceptable* **Coupling**

Stamp Coupling

Two modules (``A'' and ``B'') exhibit stamp coupling if one passes directly to the other a ``composite'' piece of data-that is, a piece of data with meaningful internal structure -such as a record (or structure), array, or (pointer to) a list or tree.



Design at the Routine Level: *Ideal* **Coupling**

Modules *A* and *B* have the lowest possible level of coupling -no coupling at all -if they have no direct communication and are also not ``tied together'' by shared access to the same global data area or external device.

it implies that A and B be implemented, tested, and maintained (almost) completely independently; neither will affect the behavior of the other

Good Routine Names

A good name for a routine clearly describes everything the routine does

Describe everything the routine does

describe all the outputs and side effects. If a routine computes report totals and opens an output file, ComputeReportTotals() is not an adequate name for the routine. ComputeReportTotalsAndOpenOutputFile() is an adequate name but is too long and silly.

Avoid meaningless, vague, or wishy washy verbs

Some verbs are elastic, stretched to cover just about any meaning. Routine names like HandleCalculation() , PerformServices() , OutputUser() , ProcessInput() , and DealWithOutput() don't tell you what the routines do.

Don't differentiate routine names solely by number

Make names of routines as long as necessary

To name a function, use a description of the return value

describe all the outputs and side effects. If a routine computes report totals and opens an output file, ComputeReportTotals() is not an adequate name for the routine. ComputeReportTotalsAndOpenOutputFile() is an adequate name but is too long and silly.

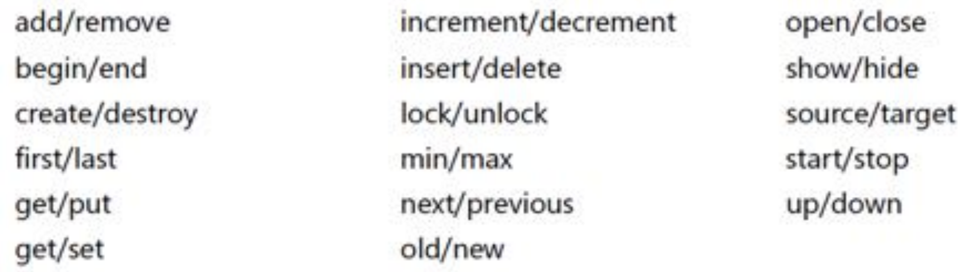
To name a procedure, use a strong verb followed by an object

A procedure with functional cohesion usually performs an operation on an object. The name should reflect what the procedure does, and an operation on an object implies a verb-plus object name.

Establish conventions for common operations

In some systems, it's important to distinguish among different kinds of operations. A naming convention is often the easiest and most reliable way of indicating these distinctions

Use opposites precisely



add/remove
begin/end
create/destroy
first/last
get/put
get/set
increment/decrement
insert/delete
lock/unlock
min/max
next/previous
old/new
open/close
show/hide
source/target
start/stop
up/down

How Long Can a Routine Be?

The theoretical best maximum length is often described as one screen or one or two pages of program listing, approximately **50 to 150 lines**. In this spirit, IBM once limited routines to 50 lines, and TRW limited them to two pages (McCabe 1976)

A large percentage of routines in object-oriented programs will be accessor routines, which will be very short. From time to time, a complex algorithm will lead to a longer routine, and in those circumstances, the routine should be allowed to grow organically up to 100–200 lines (A line is a non comment, nonblank line of source code).

How to Use Routine Parameters?

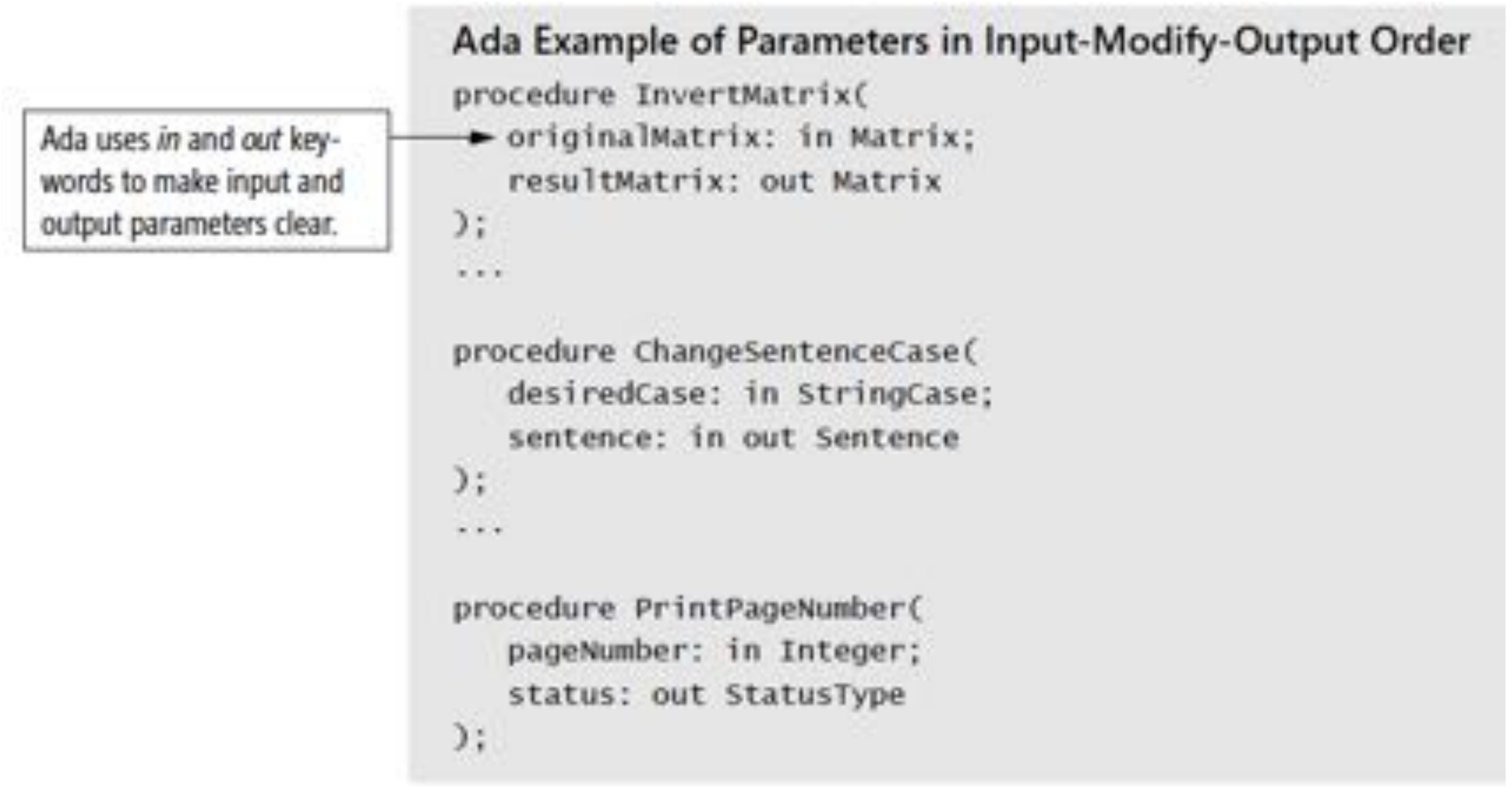
Interfaces between routines are some of the most error-prone areas of a program

One often-cited study by Basiliand Perricone (1984) found that **39 percent** of all errors were internal interface errors—errors in communication between routines.

How to Use Routine Parameters?

Put parameters in input-modify-output order

Instead of ordering parameters randomly or alphabetically, list the parameters that are input-only first, input-and-output second, and output-only third



Ada Example of Parameters in Input-Modify-Output Order

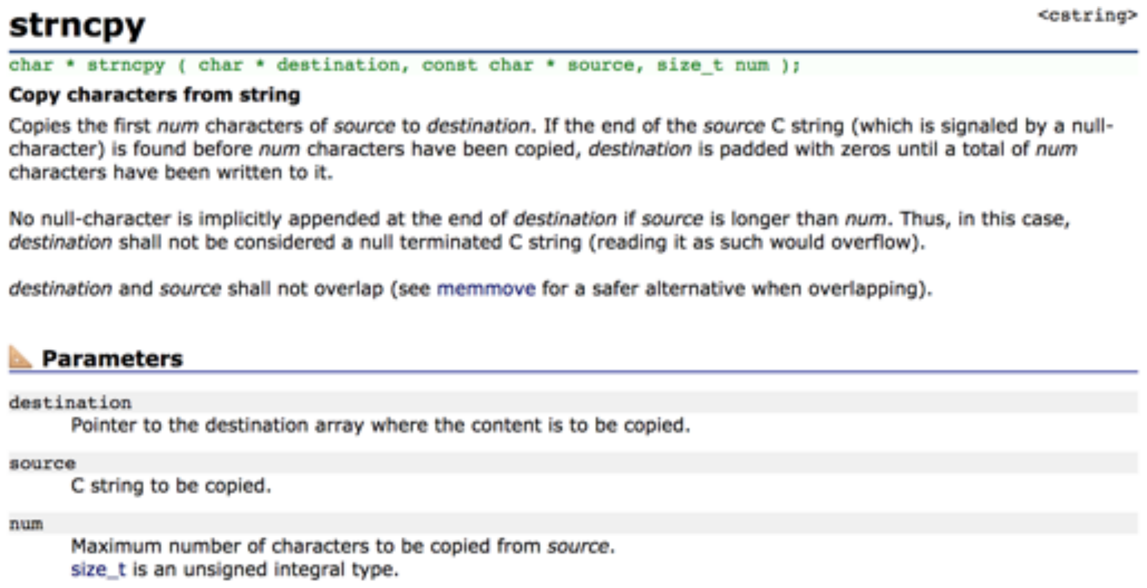
Ada uses *in* and *out* keywords to make input and output parameters clear.

```
procedure InvertMatrix(
   originalMatrix: in Matrix;
   resultMatrix: out Matrix
);
...

procedure ChangeSentenceCase(
   desiredCase: in StringCase;
   sentence: in out Sentence
);
...

procedure PrintPageNumber(
   pageNumber: in Integer;
   status: out StatusType
);
```

How to Use Routine Parameters?

If several routines use similar parameters, put the similar parameters in a consistent order

The order of routine parameters can be a mnemonic, and inconsistent order can make parameters hard to remember.



strncpy <cstring>

`char * strncpy ( char * destination, const char * source, size_t num );`

Copy characters from string

Copies the first *num* characters of *source* to *destination*. If the end of the *source* C string (which is signaled by a null-character) is found before *num* characters have been copied, *destination* is padded with zeros until a total of *num* characters have been written to it.

No null-character is implicitly appended at the end of *destination* if *source* is longer than *num*. Thus, in this case, *destination* shall not be considered a null terminated C string (reading it as such would overflow).

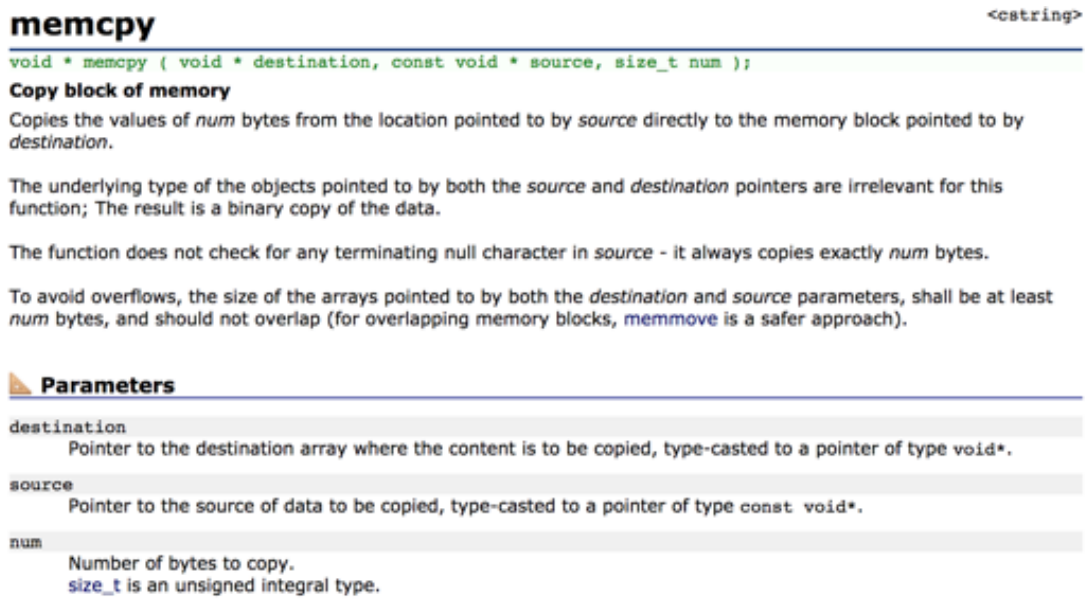
destination and *source* shall not overlap (see memmove for a safer alternative when overlapping).

Parameters

`destination`
Pointer to the destination array where the content is to be copied.

`source`
C string to be copied.

`num`
Maximum number of characters to be copied from *source*.
size_t is an unsigned integral type.



memcpy <cstring>

`void * memcpy ( void * destination, const void * source, size_t num );`

Copy block of memory

Copies the values of *num* bytes from the location pointed to by *source* directly to the memory block pointed to by *destination*.

The underlying type of the objects pointed to by both the *source* and *destination* pointers are irrelevant for this function; The result is a binary copy of the data.

The function does not check for any terminating null character in *source* - it always copies exactly *num* bytes.

To avoid overflows, the size of the arrays pointed to by both the *destination* and *source* parameters, shall be at least *num* bytes, and should not overlap (for overlapping memory blocks, memmove is a safer approach).

Parameters

`destination`
Pointer to the destination array where the content is to be copied, type-casted to a pointer of type `void*`.

`source`
Pointer to the source of data to be copied, type-casted to a pointer of type `const void*`.

`num`
Number of bytes to copy.
size_t is an unsigned integral type.

How to Use Routine Parameters?

Use all the parameters

If you pass a parameter to a routine, use it. If you aren't using it, remove the parameter from the routine interface.

Unused parameters are correlated with an increased error rate. In one study, 46 percent of routines with no unused variables had no errors, and only 17 to 29 percent of routines with more than one unreferenced variable had no errors (Card, Church, and Agresti1986).

How to Use Routine Parameters?

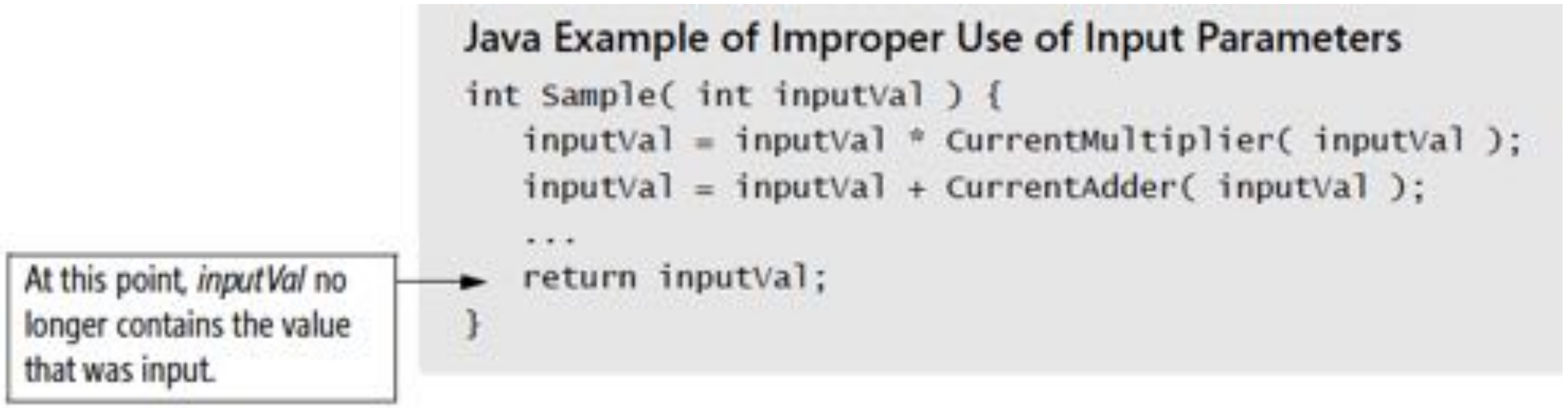
Put status or error variables last

By convention, status variables and variables that indicate an error has occurred go last in the parameter list. They are incidental to the main purpose of the routine, and they are output-only parameters, so it’s a sensible convention.

How to Use Routine Parameters?

Don’t use routine parameters as working variables

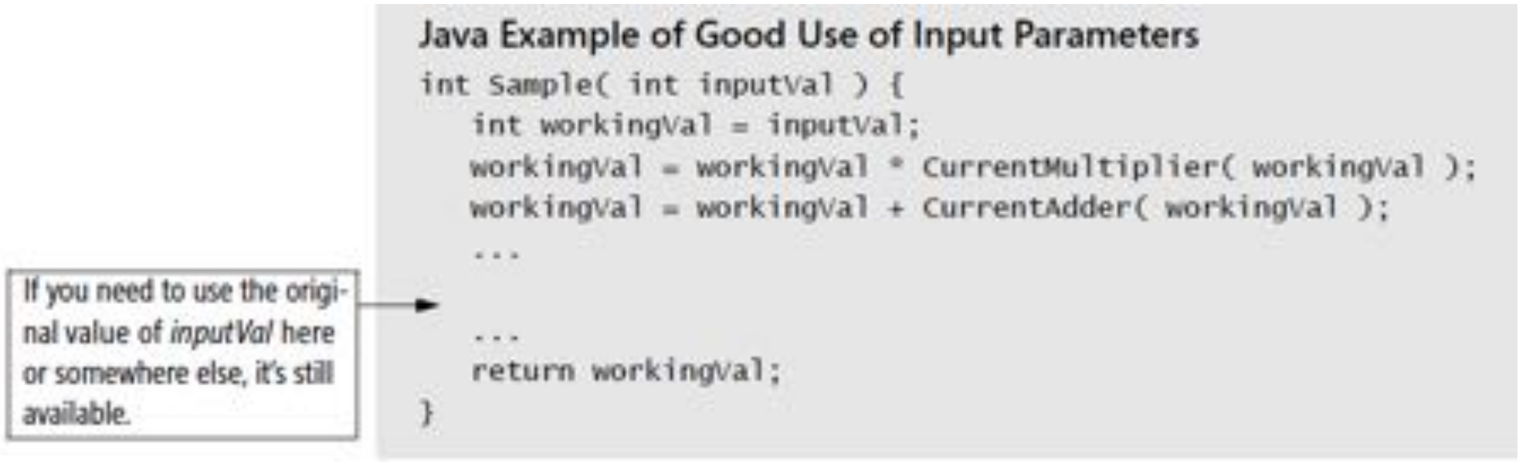
It’s dangerous to use the parameters passed to a routine as working variables. Use local variables instead.



Java Example of Improper Use of Input Parameters

```java
int Sample( int inputVal ) {
   inputVal = inputVal * CurrentMultiplier( inputVal );
   inputVal = inputVal + CurrentAdder( inputVal );
   ...
   return inputVal;
}
```

At this point, *inputVal* no longer contains the value that was input.



Java Example of Good Use of Input Parameters

```java
int Sample( int inputVal ) {
   int workingVal = inputVal;
   workingVal = workingVal * CurrentMultiplier( workingVal );
   workingVal = workingVal + CurrentAdder( workingVal );
   ...

   ...
   return workingVal;
}
```

If you need to use the original value of *inputVal* here or somewhere else, it’s still available.

How to Use Routine Parameters?

Document interface assumptions about parameters

If you assume the data being passed to your routine has certain characteristics, document the assumptions as you make them. Even better than commenting your assumptions, use assertions to put them into code

Whether parameters are input-only, modified, or output-only

Units of numeric parameters (inches, feet, meters, and so on)

Meanings of status codes and error values if enumerated types aren't used

Ranges of expected values

Specific values that should never appear

How to Use Routine Parameters?

Limit the number of a routine’s parameters to about seven

Seven is a magic number for people’s comprehension

If you find yourself consistently passing more than a few arguments, the **coupling among your routines is too tight**. Design the routine or group of routines to reduce the coupling. *If you are passing the same data to many different routines, group the routines into a class and treat the frequently used data as class data*.

How to Use Routine Parameters?

Make sure actual parameters match formal parameters

Formal parameters, also known as "dummy parameters," are the variables declared in a routine definition. Actual parameters are the variables, constants, or expressions used in the actual routine calls.

A common mistake is to put the wrong type of variable in a routine call

SUMMARY

- The most important reason for creating a routine is to improve the intellectual manageability of a program, and you can create a routine for many other good reasons. Saving space is a minor reason; improved readability, reliability, and modifiability are better reasons.
- Sometimes the operation that most benefits from being put into a routine of its own is a simple one.
- You can classify routines into various kinds of cohesion, but you can make most routines functionally cohesive, which is best.
- The name of a routine is an indication of its quality. If the name is bad and it's accurate, the routine might be poorly designed. If the name is bad and it's inaccurate, it's not telling you what the program does. Either way, a bad name means that the program needs to be changed.

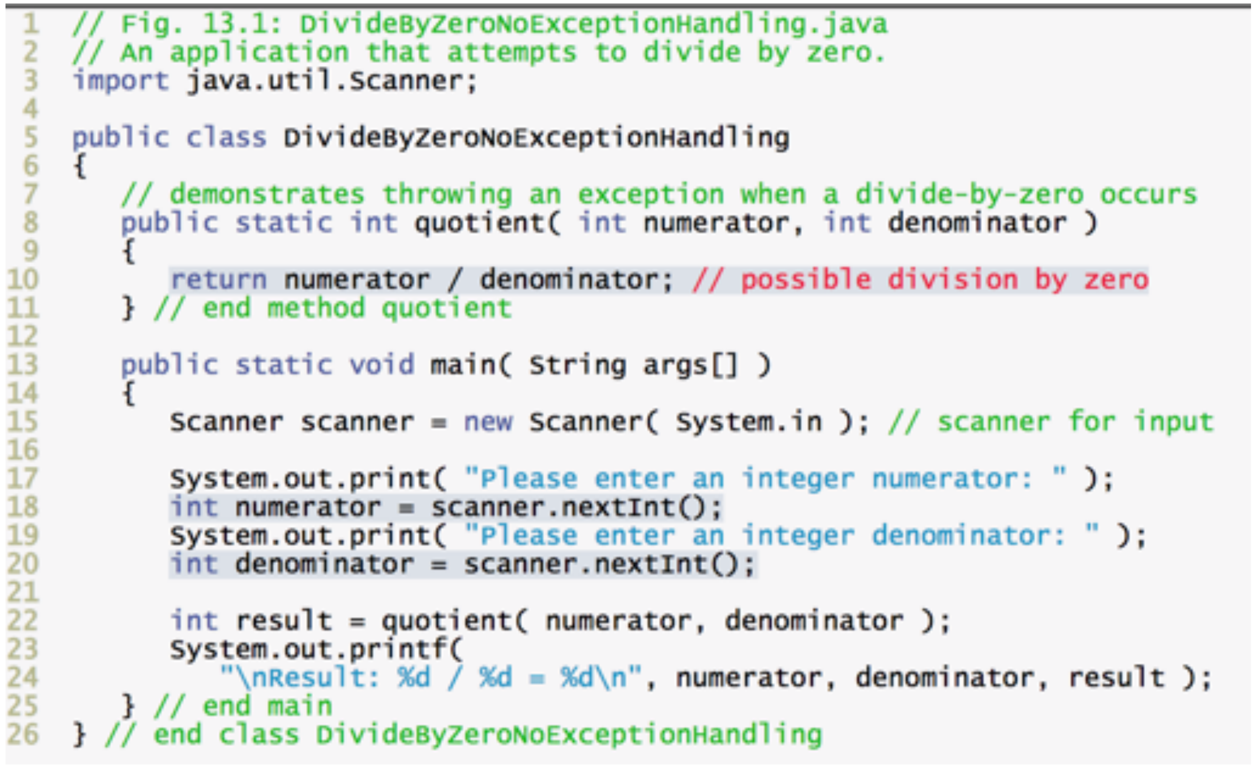
Defensive Programming

The idea is based on *defensive driving*. In defensive driving, you adopt the mind-set that you’re never sure what the other drivers are going to do. That way, you make sure that if they do something dangerous you won’t be hurt. You take responsibility for protecting yourself even when it might be the other driver’s fault.

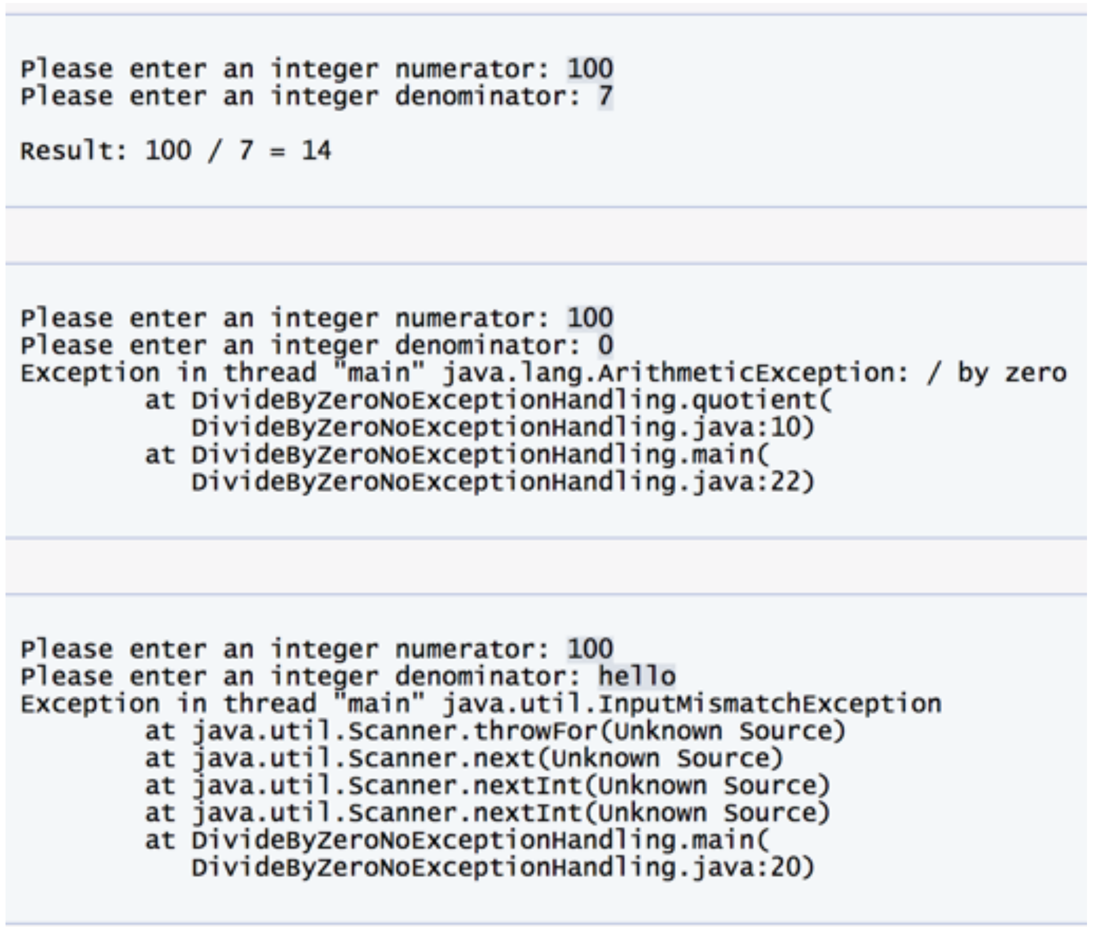
Defensive Programming

Part of the Interstate-90 floating bridge in Seattle sank during a storm because the flotation tanks were left uncovered, they filled with water, and the bridge became too heavy to float. During construction, protecting yourself against the small stuff matters more than you might think.

Defensive Programming



```java
 1  // Fig. 13.1: DivideByZeroNoExceptionHandling.java
 2  // An application that attempts to divide by zero.
 3  import java.util.Scanner;
 4
 5  public class DivideByZeroNoExceptionHandling
 6  {
 7     // demonstrates throwing an exception when a divide-by-zero occurs
 8     public static int quotient( int numerator, int denominator )
 9     {
10        return numerator / denominator; // possible division by zero
11     } // end method quotient
12
13     public static void main( String args[] )
14     {
15        Scanner scanner = new Scanner( System.in ); // scanner for input
16
17        System.out.print( "Please enter an integer numerator: " );
18        int numerator = scanner.nextInt();
19        System.out.print( "Please enter an integer denominator: " );
20        int denominator = scanner.nextInt();
21
22        int result = quotient( numerator, denominator );
23        System.out.printf(
24           "\nResult: %d / %d = %d\n", numerator, denominator, result );
25     } // end main
26  } // end class DivideByZeroNoExceptionHandling
```

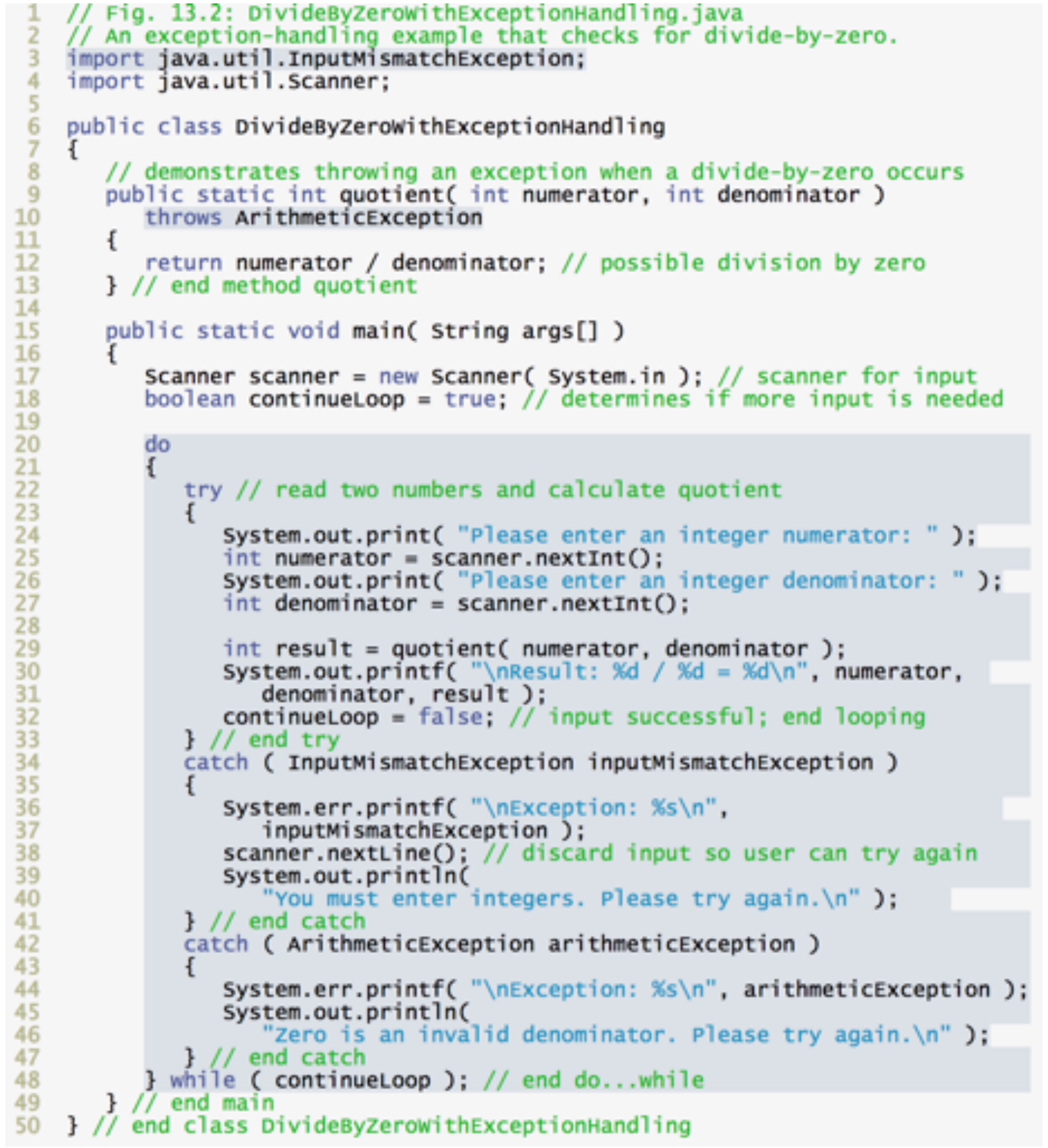


```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

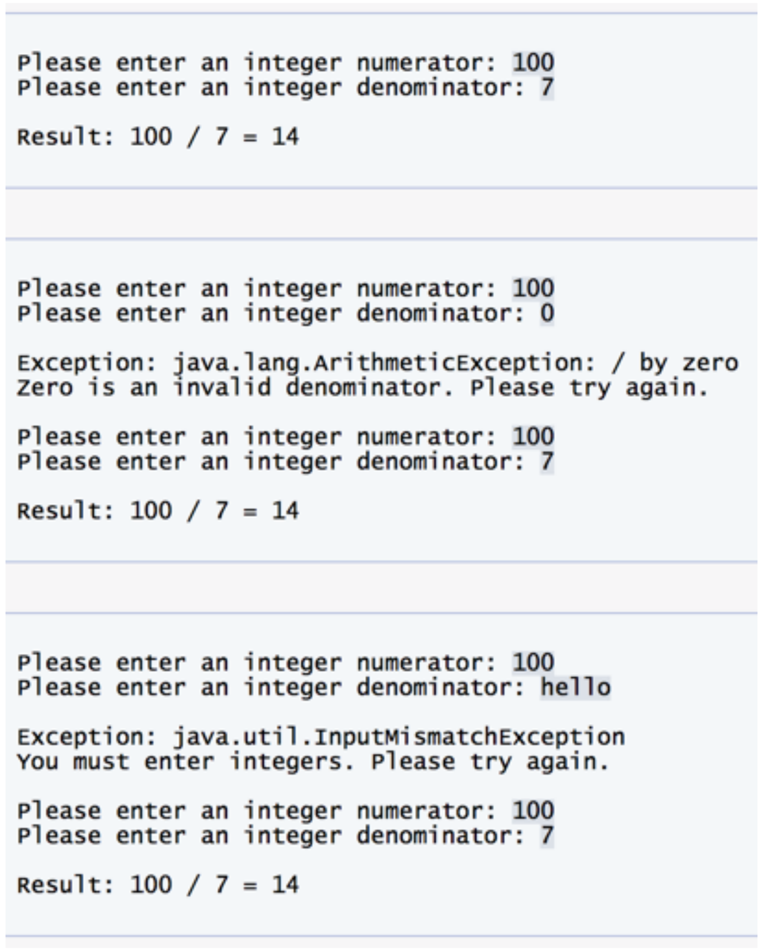
```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:10)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:22)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:20)
```

Defensive Programming



```java
// Fig. 13.2: DivideByZeroWithExceptionHandling.java
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean continueLoop = true; // determines if more input is needed

      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print( "Please enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Please enter an integer denominator: " );
            int denominator = scanner.nextInt();

            int result = quotient( numerator, denominator );
            System.out.printf( "\nResult: %d / %d = %d\n", numerator,
               denominator, result );
            continueLoop = false; // input successful; end looping
         } // end try
         catch ( InputMismatchException inputMismatchException )
         {
            System.err.printf( "\nException: %s\n",
               inputMismatchException );
            scanner.nextLine(); // discard input so user can try again
            System.out.println(
               "You must enter integers. Please try again.\n" );
         } // end catch
         catch ( ArithmeticException arithmeticException )
         {
            System.err.printf( "\nException: %s\n", arithmeticException );
            System.out.println(
               "Zero is an invalid denominator. Please try again.\n" );
         } // end catch
      } while ( continueLoop ); // end do...while
   } // end main
} // end class DivideByZeroWithExceptionHandling
```



```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Protecting Your Program from Invalid Inputs

In school you might have heard the expression, “Garbage in, garbage out.” That expression is essentially software development’s version of caveat emptor: let the user beware.

For production software, garbage in, garbage out isn’t good enough. A good program never puts out garbage, regardless of what it takes in.

Check the values of all data from external sources

Check the values of all routine input parameters

Decide how to handle bad inputs

Assertions

An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs

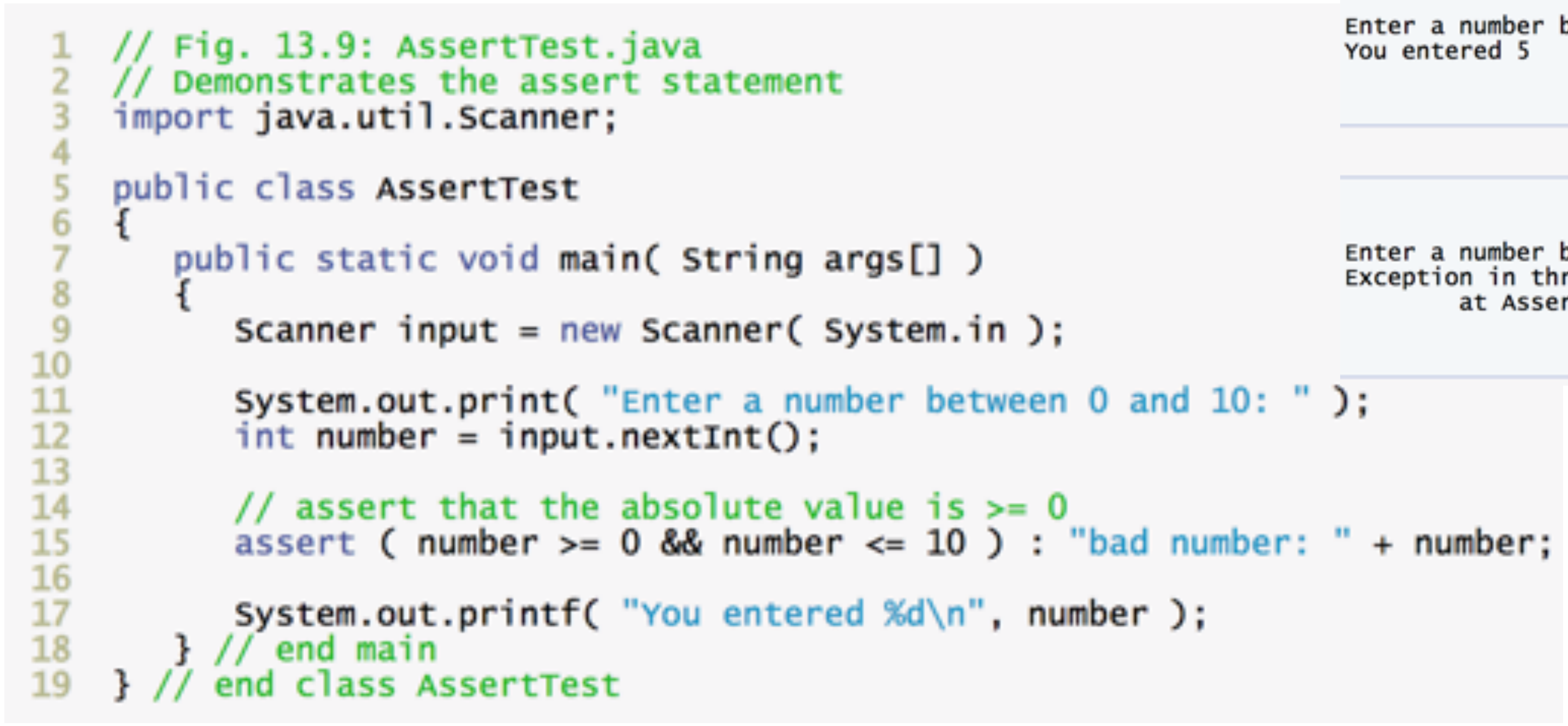
When an assertion is true, that means everything is operating as expected. When it's false, that means it has detected an unexpected error in the code.

Assertions are especially useful in large, complicated programs and in high-reliability programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on.

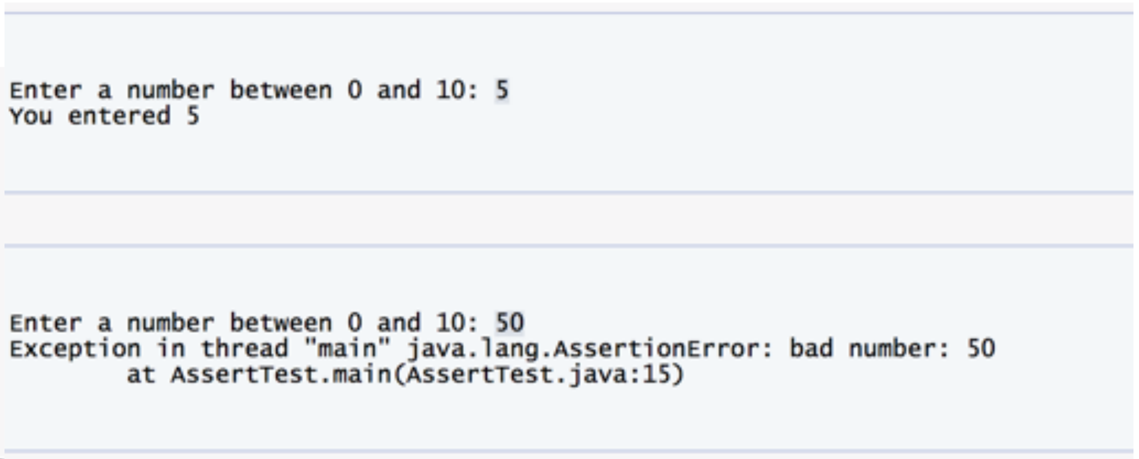
Assertions

An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs

An assertion usually takes two arguments: a **Boolean expression** that describes the assumption that's supposed to be true, and a **message** to display if it isn't.



```java
// Fig. 13.9: AssertTest.java
// Demonstrates the assert statement
import java.util.Scanner;

public class AssertTest
{
   public static void main( String args[] )
   {
      Scanner input = new Scanner( System.in );

      System.out.print( "Enter a number between 0 and 10: " );
      int number = input.nextInt();

      // assert that the absolute value is >= 0
      assert ( number >= 0 && number <= 10 ) : "bad number: " + number;

      System.out.printf( "You entered %d\n", number );
   } // end main
} // end class AssertTest
```



```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:15)
```

Assertions

An assertion is code that’s used during development—usually a routine or macro—that allows a program to check itself as it runs

You use assertions primarily for debugging and identifying logic errors in an application. They are comment-like code

You must explicitly enable assertions when executing a program, because they reduce performance and are unnecessary for the program’s user.

Users should not encounter any Assertion Errors through normal execution of a properly written program. Such errors should only indicate bugs in the implementation. E.g., **Debug mode vs. Release mode**

Assertions

An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs

- That an input parameter's value falls within its expected range (or an output parameter's value does)
- That a file or stream is open (or closed) when a routine begins executing (or when it ends executing)
- That a file or stream is at the beginning (or end) when a routine begins executing (or when it ends executing)
- That a file or stream is open for read-only, write-only, or both read and write
- That the value of an input-only variable is not changed by a routine
- That a pointer is non-null
- That an array or other container passed into a routine can contain at least X number of data elements
- That a table has been initialized to contain real values
- That a container is empty (or full) when a routine begins executing (or when it finishes)
- That the results from a highly optimized, complicated routine match the results from a slower but clearly written routine

Assertions: **Guidelines for Using Assertions**

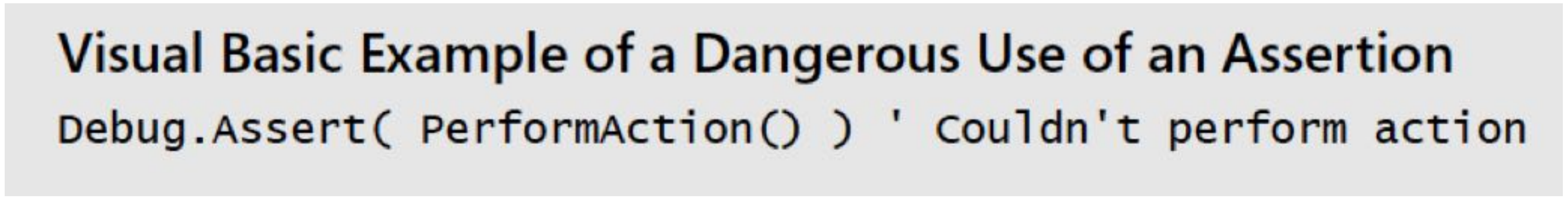
Use error-handling code for conditions you expect to occur; use assertions for conditions that should never occur

Assertions check for conditions that should never occur. Error-handling code checks for off-nominal circumstances that might not occur very often, but that have been anticipated by the programmer who wrote the code and that need to be handled by the production code. Error handling typically checks for bad input data; assertions check for bugs in the code.

Assertions: **Guidelines for Using Assertions**

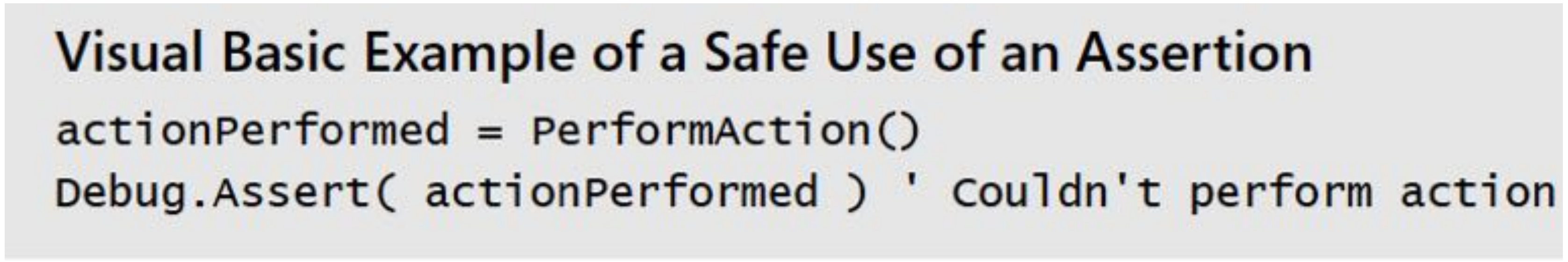
Avoid putting executable code into assertions

Putting code into an assertion raises the possibility that the compiler will eliminate the code when you turn off the assertions.



Visual Basic Example of a Dangerous Use of an Assertion

```
Debug.Assert( PerformAction() ) ' Couldn't perform action
```



Visual Basic Example of a Safe Use of an Assertion

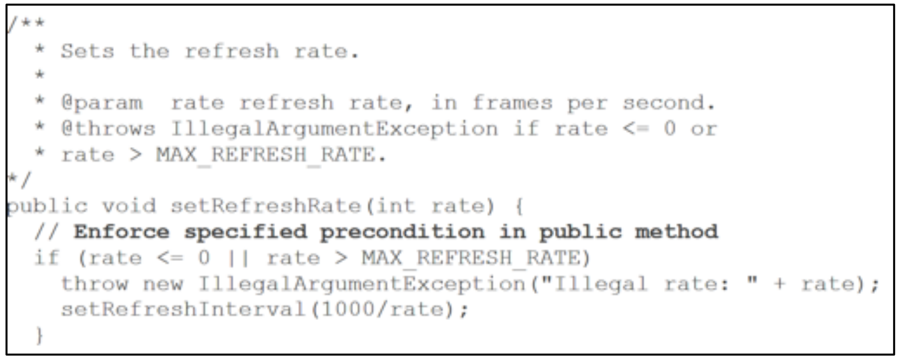
```
actionPerformed = PerformAction()
Debug.Assert( actionPerformed ) ' Couldn't perform action
```

Assertions: **Guidelines for Using Assertions**

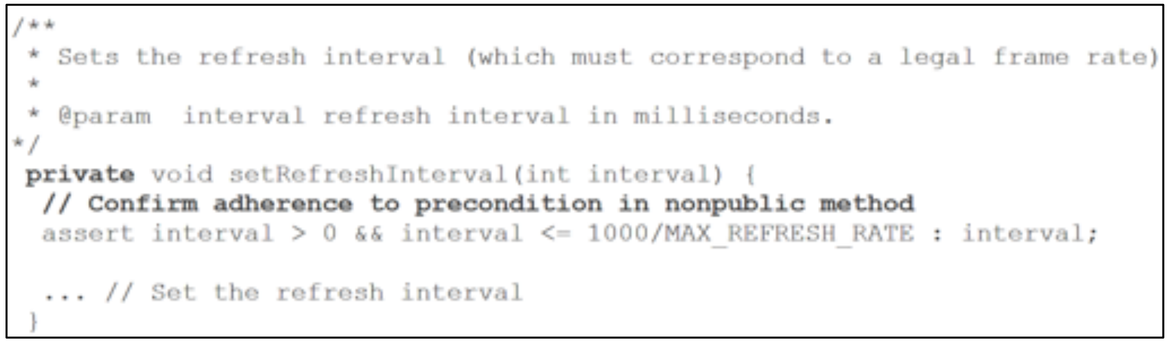
Do not use assertions for argument checking in public methods

Argument checking is typically part of the published specifications (or contract) of a method, and these specifications must be obeyed whether assertions are enabled or disabled

Erroneous arguments should result in an appropriate runtime exception (such as IllegalArgumentException, IndexOutOfBoundsException, or NullPointerException)



```
/**
  * Sets the refresh rate.
  *
  * @param  rate refresh rate, in frames per second.
  * @throws IllegalArgumentException if rate <= 0 or
  * rate > MAX_REFRESH_RATE.
*/
public void setRefreshRate(int rate) {
  // Enforce specified precondition in public method
  if (rate <= 0 || rate > MAX_REFRESH_RATE)
    throw new IllegalArgumentException("Illegal rate: " + rate);
    setRefreshInterval(1000/rate);
  }
```

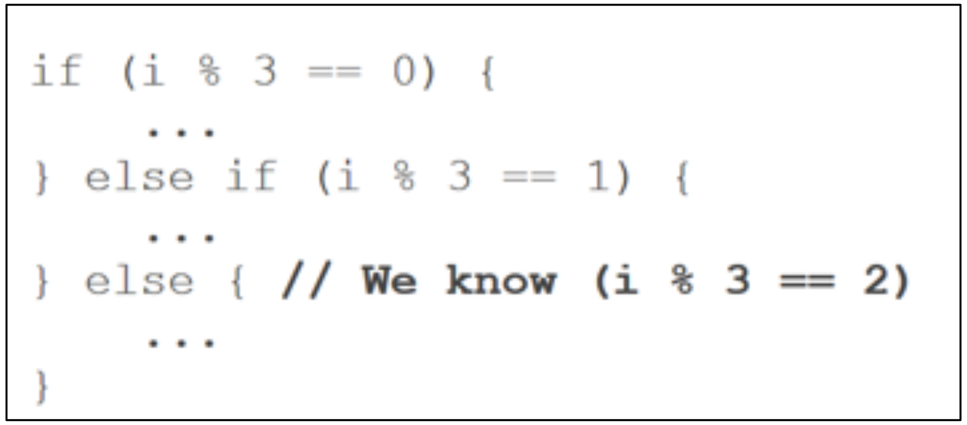


```
/**
 * Sets the refresh interval (which must correspond to a legal frame rate)
 *
 * @param  interval refresh interval in milliseconds.
*/
 private void setRefreshInterval(int interval) {
  // Confirm adherence to precondition in nonpublic method
  assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;

  ... // Set the refresh interval
 }
```

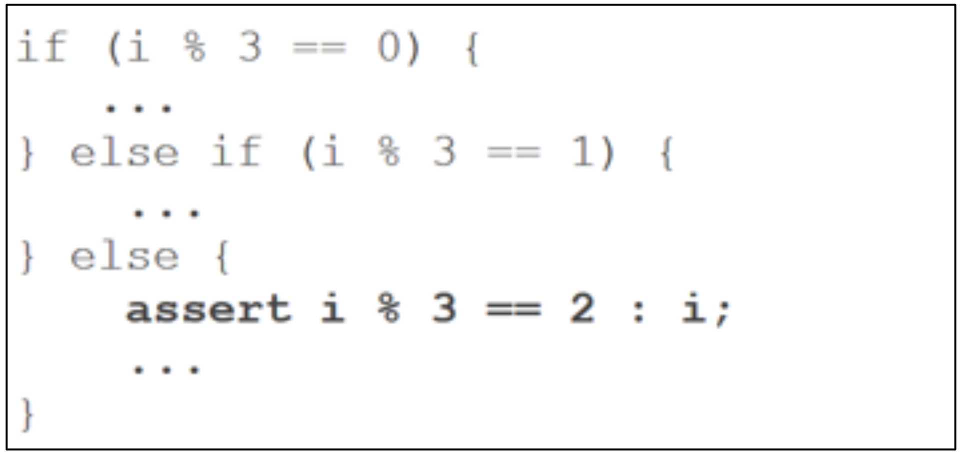
Assertions: **Guidelines for Using Assertions**

Use Assertions for Internal Invariants

An invariant is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution. For example, a loop invariant is a condition that is true at the beginning and end of every execution of a loop.



```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // We know (i % 3 == 2)
    ...
}
```



```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}
```

Assertions: **Guidelines for Using Assertions**

Use Assertions for Internal Invariants

An invariant is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution. For example, a loop invariant is a condition that is true at the beginning and end of every execution of a loop.

Assumption: the suit variable will have one of only four values. To test this assumption, you should add the following default case:

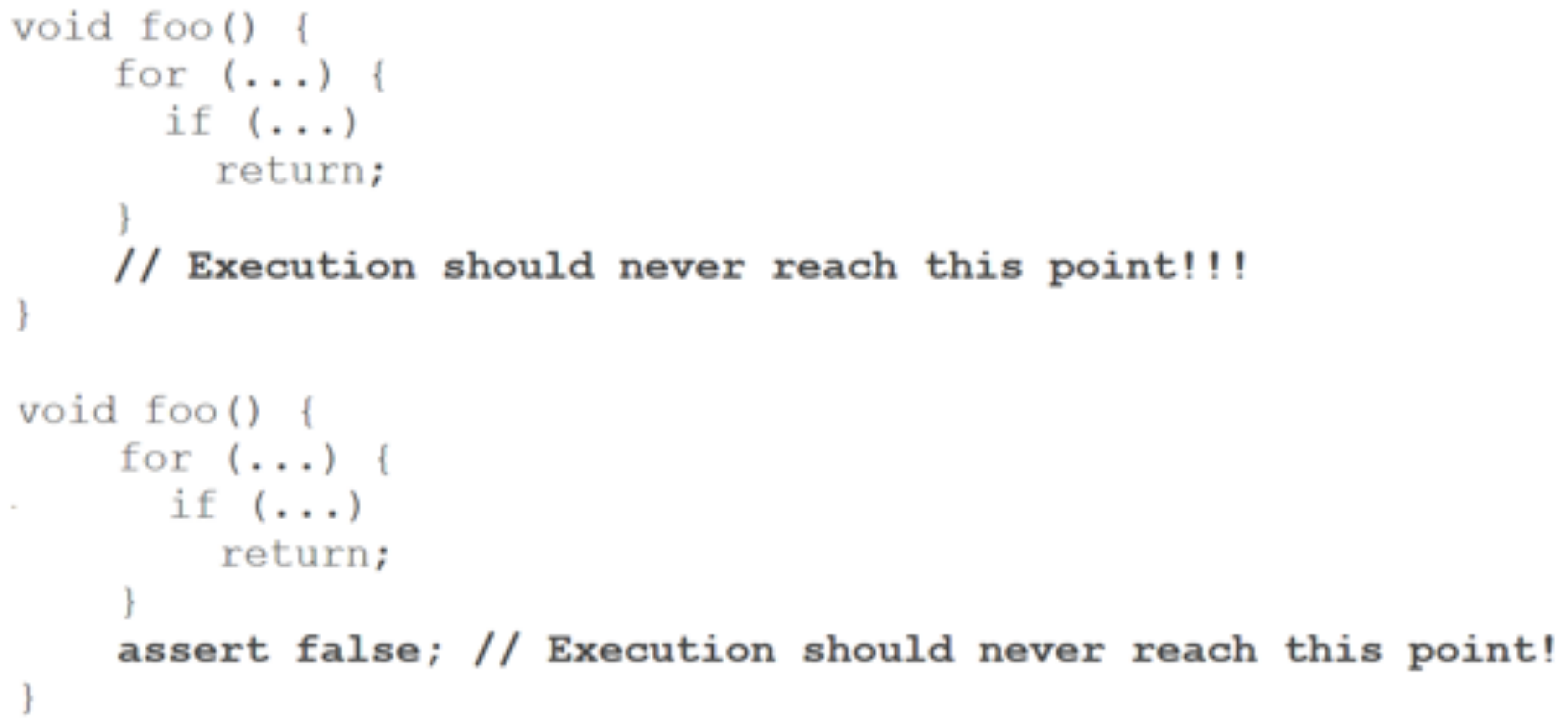
```
default:
    assert false : suit;
```

```
switch(suit) {
  case Suit.CLUBS:
    ...
  break;

  case Suit.DIAMONDS:
    ...
  break;

  case Suit.HEARTS:
    ...
    break;

  case Suit.SPADES:
      ...
}
```

Assertions: **Guidelines for Using Assertions**

Use Assertions for Control Flow Invariants

place an assertion at any location you assume will not be reached



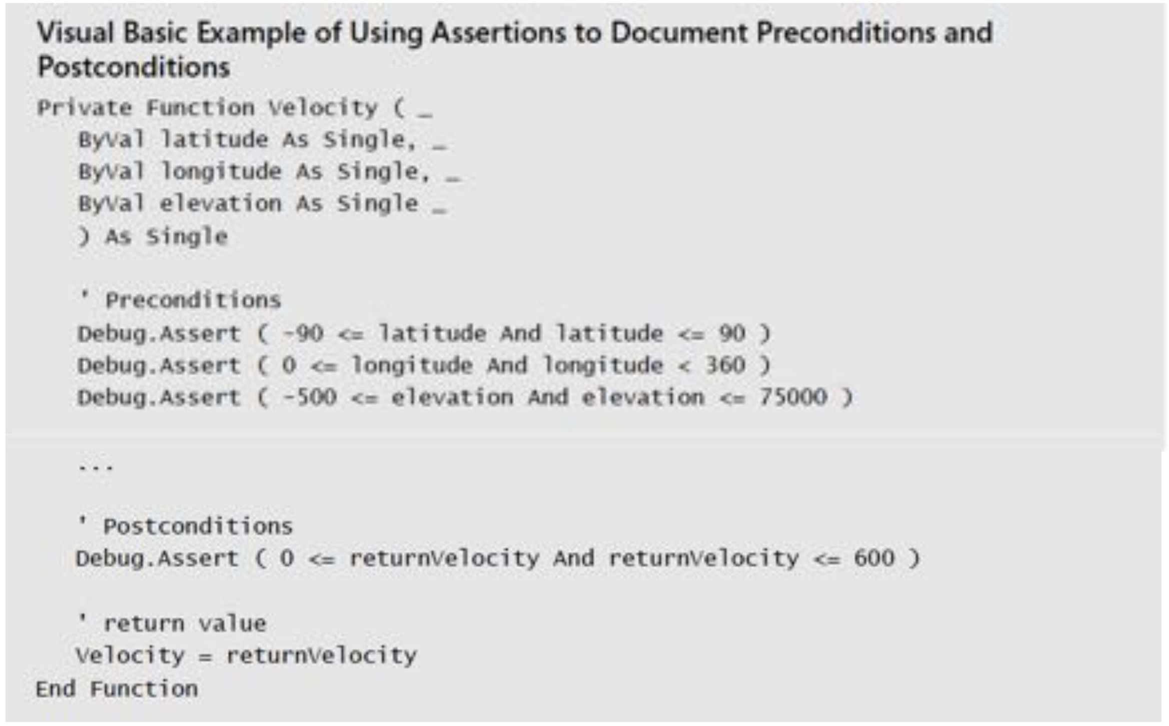
```
void foo() {
     for (...) {
       if (...)
         return;
     }
     // Execution should never reach this point!!!
}

void foo() {
     for (...) {
       if (...)
         return;
     }
     assert false; // Execution should never reach this point!
}
```

Assertions: **Guidelines for Using Assertions**

Use assertions to document and verify preconditions and postconditions

Preconditions are the properties that the client code of a routine or class promises will be true before it calls the routine or instantiates the object. *Preconditions are the client code's obligations to the code it calls*.

Postconditions are the properties that the routine or class promises will be true when it concludes executing. *Postconditions are the routine's or class's obligations to the code that uses it*.



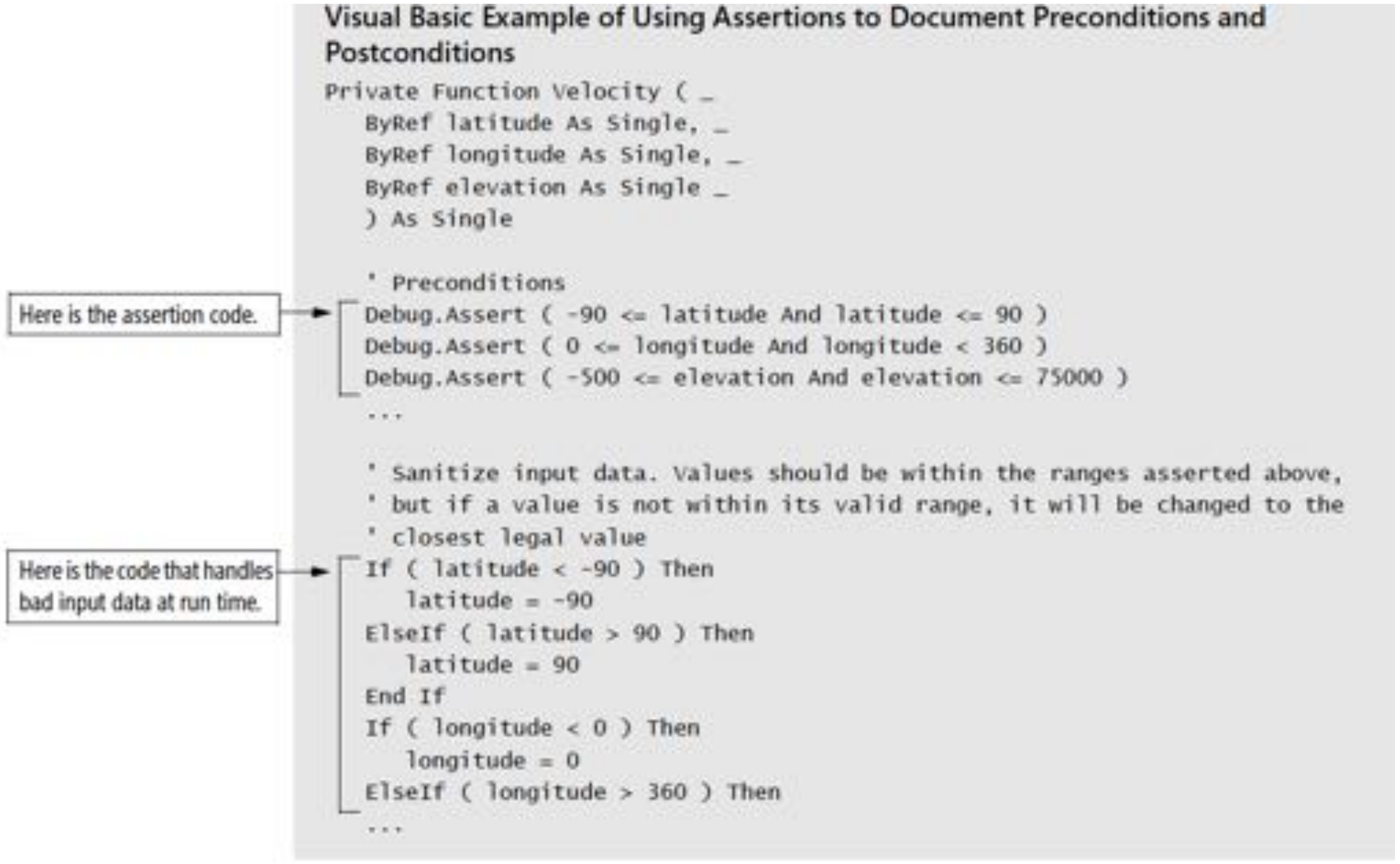
Visual Basic Example of Using Assertions to Document Preconditions and Postconditions

```
Private Function Velocity ( _
   ByVal latitude As Single, _
   ByVal longitude As Single, _
   ByVal elevation As Single _
   ) As Single

   ' Preconditions
   Debug.Assert ( -90 <= latitude And latitude <= 90 )
   Debug.Assert ( 0 <= longitude And longitude < 360 )
   Debug.Assert ( -500 <= elevation And elevation <= 75000 )

   ...

   ' Postconditions
   Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )

   ' return value
   Velocity = returnVelocity
End Function
```

If the variables *latitude*, *longitude*, and *elevation* were coming from an external source, invalid values should be checked and handled by error-handling code rather than by assertions.

Assertions: **Guidelines for Using Assertions**

For highly robust code, assert and then handle the error anyway



Visual Basic Example of Using Assertions to Document Preconditions and Postconditions

```
Private Function Velocity ( _
   ByRef latitude As Single, _
   ByRef longitude As Single, _
   ByRef elevation As Single _
   ) As Single

   ' Preconditions
   Debug.Assert ( -90 <= latitude And latitude <= 90 )
   Debug.Assert ( 0 <= longitude And longitude < 360 )
   Debug.Assert ( -500 <= elevation And elevation <= 75000 )
   ...

   ' Sanitize input data. Values should be within the ranges asserted above,
   ' but if a value is not within its valid range, it will be changed to the
   ' closest legal value
   If ( latitude < -90 ) Then
      latitude = -90
   ElseIf ( latitude > 90 ) Then
      latitude = 90
   End If
   If ( longitude < 0 ) Then
      longitude = 0
   ElseIf ( longitude > 360 ) Then
   ...
```

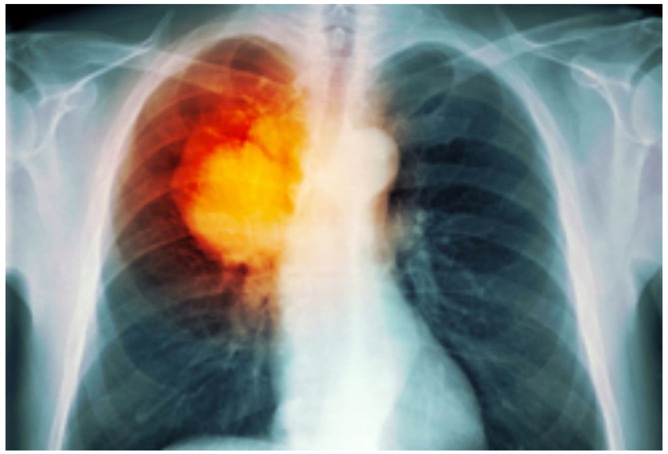
Here is the assertion code.
Here is the code that handles bad input data at run time.

Error-Handling Techniques

Return a neutral value

Sometimes the best response to bad data is to continue operating and simply return a value that’s known to be harmless.

- A numeric computation might return 0.
- A string operation might return an empty string, or a pointer operation might return an empty pointer.
- A drawing routine that gets a bad input value for color in a video game might use the default background or foreground color.



Error-Handling Techniques

Substitute the next piece of valid data

When processing a stream of data, some circumstances call for simply returning the next valid data.

If you’re reading records from a database and encounter a corrupted record, you might simply continue reading until you find a valid record.

If you’re taking readings from a thermometer 100 times per second and you don’t get a valid reading one time, you might simply wait another 1/100th of a second and take the next reading.

Error-Handling Techniques

Return the same answer as the previous time

Error-Handling Techniques

Substitute the closest legal value

In some cases, you might choose to return the closest legal value. This is often a reasonable approach when taking readings from a calibrated instrument

The thermometer might be calibrated between 0 and 100 degrees Celsius, for example. If you detect a reading less than 0, you can substitute 0, which is the closest legal value.

Cars use this approach to error handling whenever going back. Since a speedometer doesn’t show negative speeds, when it simply shows a speed of 0—the closest legal value.

Error-Handling Techniques

Log a warning message to a file

When bad data is detected, you might choose to log a warning message to a file and then continue on.

This approach can be used in conjunction with other techniques like substituting the closest legal value or substituting the next piece of valid data.

If you use a log, consider whether you can safely make it publicly available or whether you need to encrypt it or protect it some other way.

Error-Handling Techniques

Return an Error Code

You could decide that only certain parts of a system will handle errors. Other parts will not handle errors locally; they will simply report that an error has been detected and trust that some other routine higher up in the calling hierarchy will handle the error.

- Set the value of a status variable
- Return status as the function's return value
- Throw an exception by using the language's built-in exception mechanism

Call an error-processing routine/object

Centralize error handling in a global error-handling routine or error-handling object.

Error-Handling Techniques

Display an error message wherever the error is encountered

This approach minimizes error-handling overhead; however, it does have the potential to spread user interface messages through the entire application-how to separate UI. ***Tight coupling***

Beware of telling a potential attacker of the system too much. Attackers sometimes use error messages to discover how to attack a system.

Error-Handling Techniques

Shutdown

Some systems shut down whenever they detect an error. This approach is useful in safety-critical applications.

Error-Handling Techniques: **Correctness vs. Robustness**

Correctness means never returning an inaccurate result;
returning no result is better than returning an inaccurate result.

Robustness means always trying to do something that will allow the software to keep operating, even if that leads to results that are inaccurate sometimes.

Safety-critical applications tend to favor correctness to robustness. It is better to return no result than to return a wrong result. e.g. the radiation machine

Consumer applications tend to favor robustness to correctness. Any result whatsoever is usually better than the software shutting down.

Exceptions

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

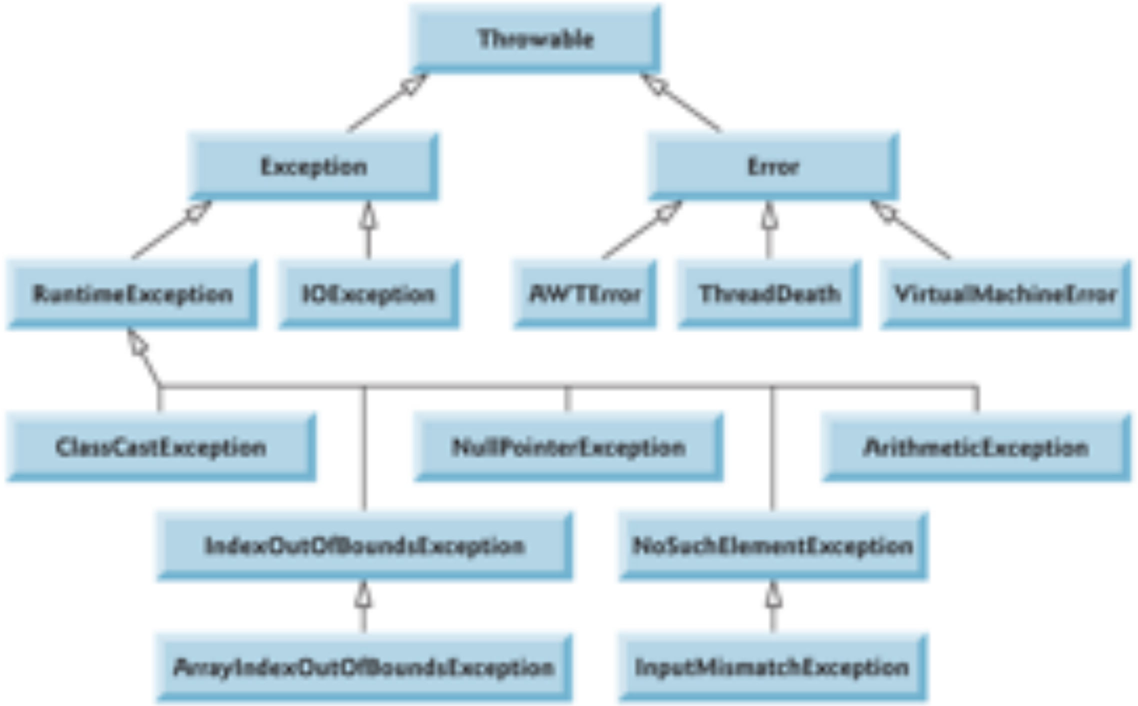
If code in one routine encounters an unexpected condition that it doesn’t know how to handle, it throws an exception, essentially throwing up its hands and yelling, “**I don’t know what to do about this—I sure hope somebody else knows how to handle it!**”

Code that has no sense of the context of an error can return control to other parts of the system that might have a better ability to interpret the error and do something useful about it.

Exceptions

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

Exception Attribute	C++	Java	Visual Basic
Try-catch support	yes	yes	yes
Try-catch-finally support	no	yes	yes
What can be thrown	*Exception* object or object derived from *Exception* class; object pointer; object reference; data type like string or int	*Exception* object or object derived from *Exception* class	*Exception* object or object derived from *Exception* class
Effect of uncaught exception	Invokes *std::unexpected()*, which by default invokes *std::terminate()*, which by default invokes *abort()*	Terminates thread of execution if exception is a "checked exception"; no effect if exception is a "runtime exception"	Terminates program
Exceptions thrown must be defined in class interface	No	Yes	No
Exceptions caught must be defined in class interface	No	Yes	No



Exceptions

Use exceptions to notify other parts of the program about errors that should not be ignored

The benefit of exceptions is their ability to signal error conditions in such a way that they cannot be ignored (Meyers 1996)

Other approaches to handling errors create the possibility that an error condition can propagate through a code base undetected. Exceptions eliminate that possibility.

Exceptions

Throw an exception only for conditions that are truly exceptional

Exceptions should be reserved for conditions that are truly exceptional—in other words, for conditions that cannot be addressed by other coding practices

Exceptions represent a tradeoff between a powerful way to handle unexpected conditions on the one hand and increased complexity on the other.

Exceptions

Don’t use an exception to pass the buck

If an error condition can be handled locally, handle it locally. Don’t throw an uncaught exception in a section of code if you can handle the error locally.

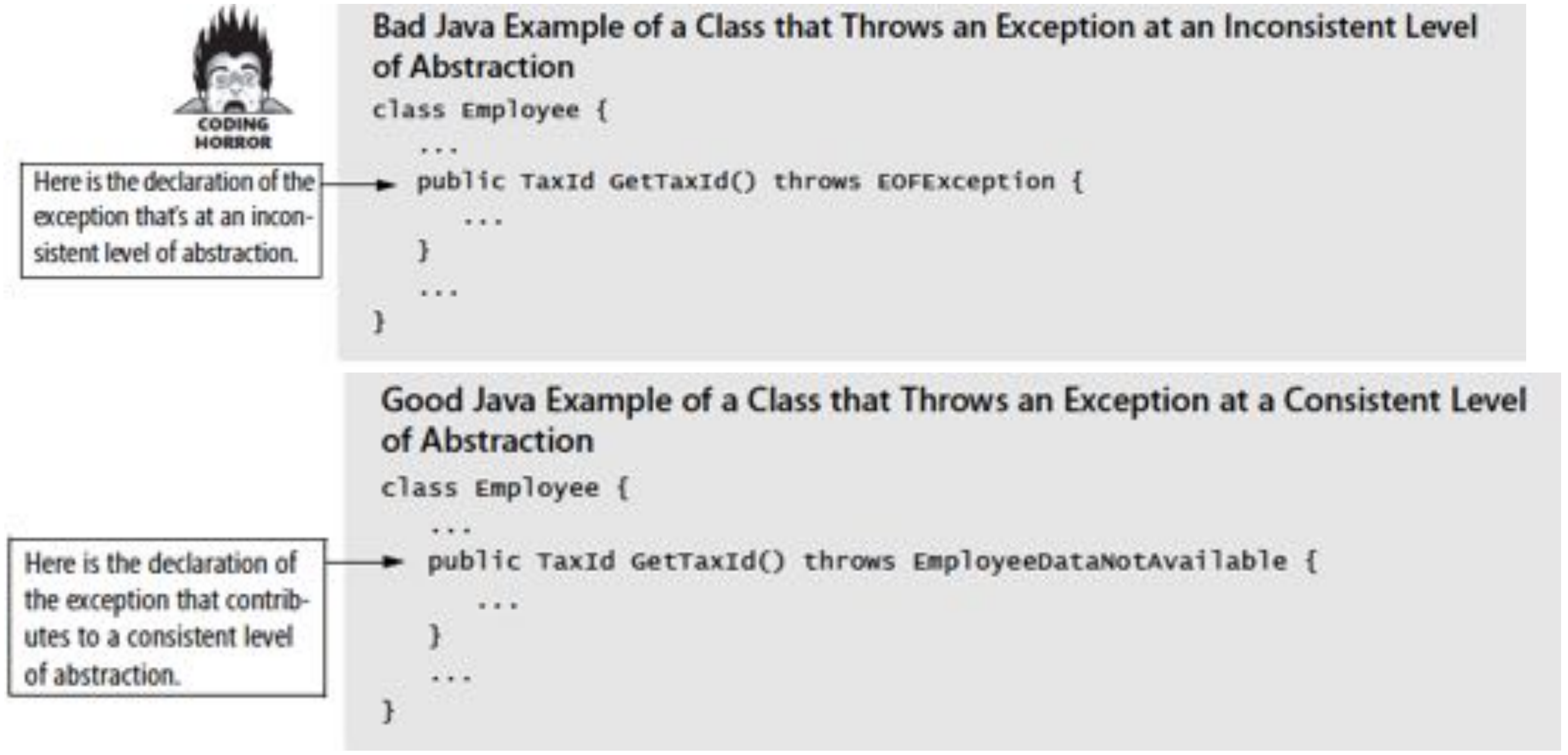
Avoid throwing exceptions in constructors and destructors unless you catch them in the same place

The rules for how exceptions are processed become very complicated very quickly when exceptions are thrown in constructors and destructors.

Exceptions

Throw exceptions at the right level of abstraction

A routine should present a consistent abstraction in its interface, and so should a class. The exceptions thrown are part of the routine interface, just like specific data types are.



CODING HORROR

Bad Java Example of a Class that Throws an Exception at an Inconsistent Level of Abstraction

Here is the declaration of the exception that's at an inconsistent level of abstraction.

```
class Employee {
   ...
   public TaxId GetTaxId() throws EOFException {
      ...
   }
   ...
}
```

Good Java Example of a Class that Throws an Exception at a Consistent Level of Abstraction

Here is the declaration of the exception that contributes to a consistent level of abstraction.

```
class Employee {
   ...
   public TaxId GetTaxId() throws EmployeeDataNotAvailable {
      ...
   }
   ...
}
```

Exceptions

Include in the exception message all information that led to the exception

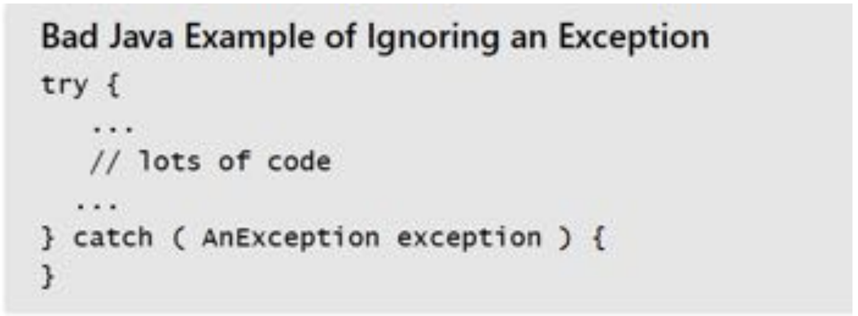
Be sure the message contains the information needed to understand why the exception was thrown.

If the exception was thrown because of an array index error, be sure the exception message includes the upper and lower array limits and the value of the illegal index.

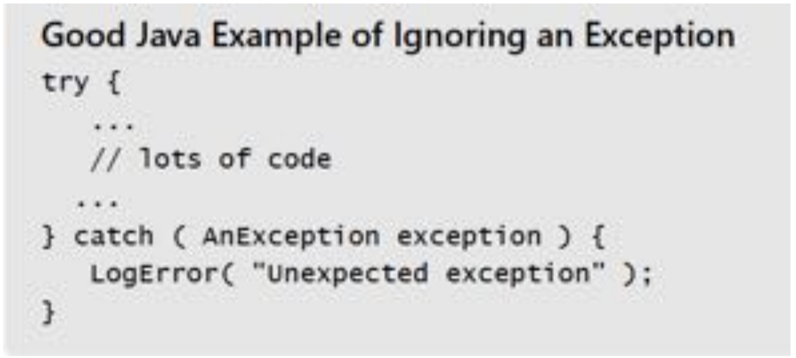
Exceptions

Avoid empty catch blocks

Either the code within the try block is wrong because it raises an exception for no reason, or the code within the catch block is wrong because it doesn't handle a valid exception.


Bad Java Example of Ignoring an Exception

```
try {
   ...
   // lots of code
   ...
} catch ( AnException exception ) {
}
```


Good Java Example of Ignoring an Exception

```
try {
   ...
   // lots of code
   ...
} catch ( AnException exception ) {
   LogError( "Unexpected exception" );
}
```

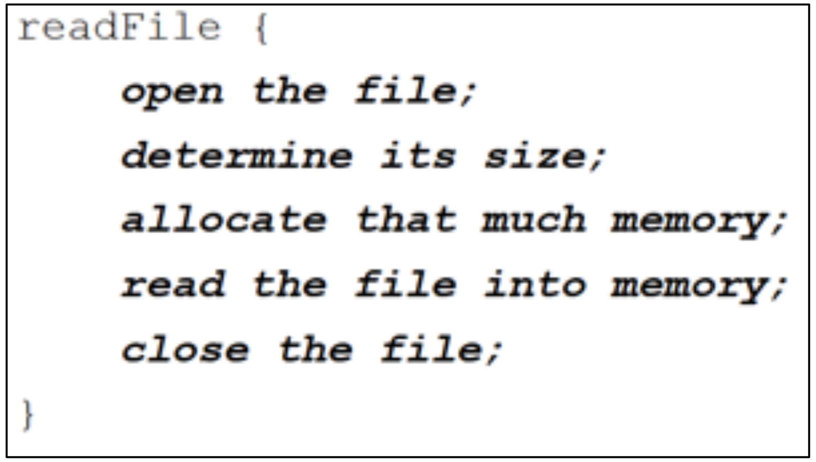
Know the exceptions your library code throws

If you're working in a language that doesn't require a routine or class to define the exceptions it throws, be sure you know what exceptions are thrown by any library code you use.

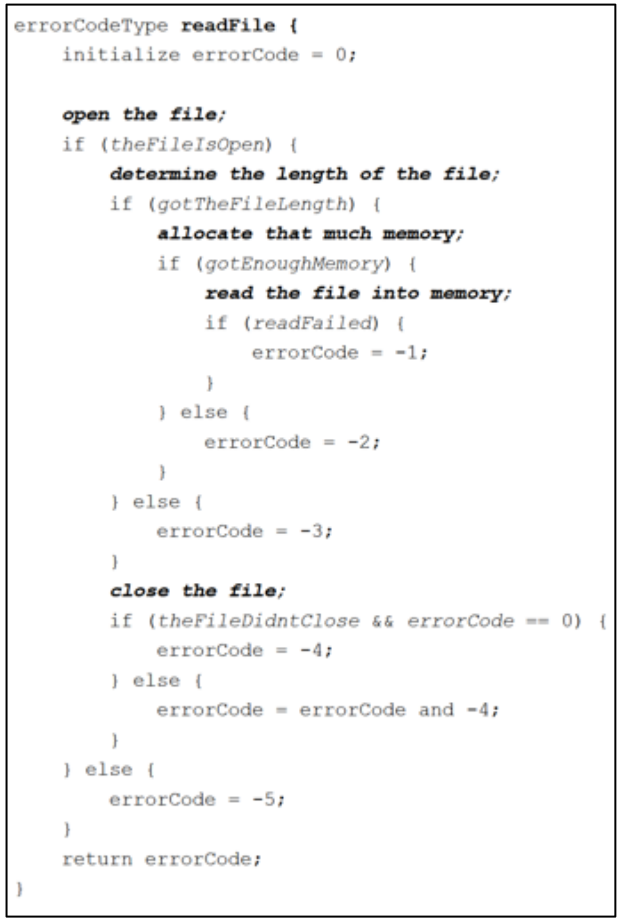
Exceptions

Consider building a centralized exception reporter

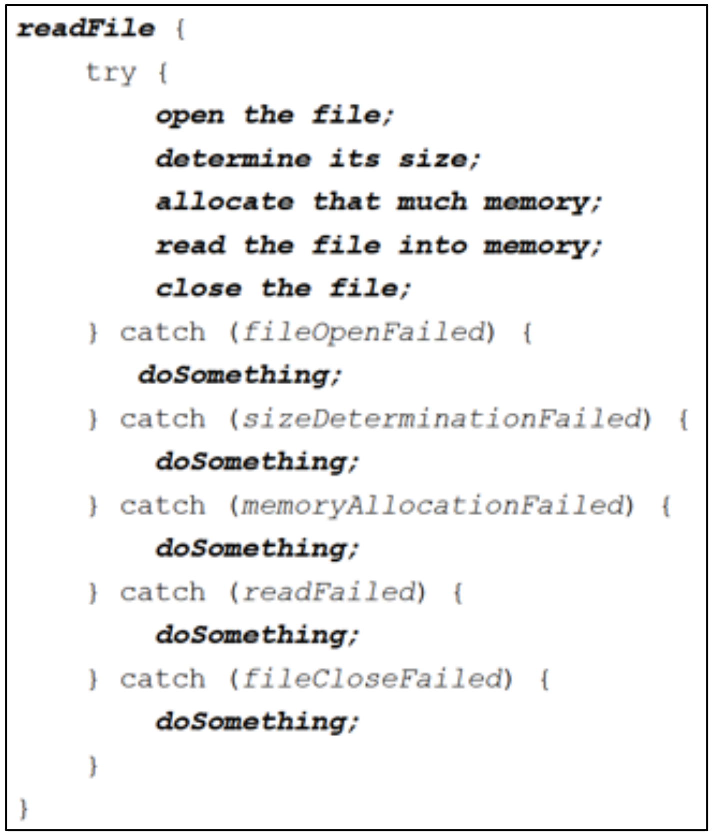
Exceptions provide the means to separate the details of what to do when something out of the ordinary. Error detection, reporting, and handling often lead to confusing spaghetti code



```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```



```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

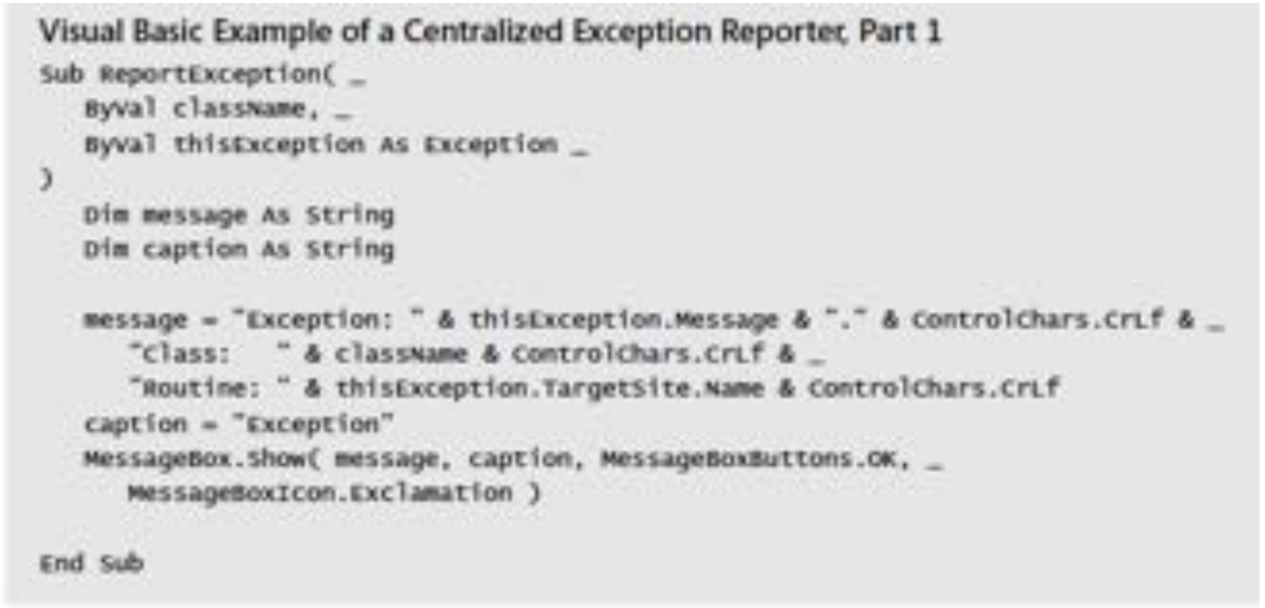


```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Exceptions

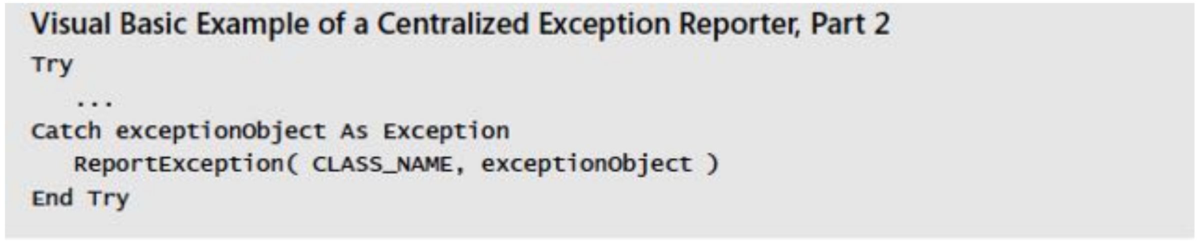
Consider building a centralized exception reporter

Exceptions provide the means to separate the details of what to do when something out of the ordinary. Error detection, reporting, and handling often lead to confusing spaghetti code



Visual Basic Example of a Centralized Exception Reporter, Part 1

```
Sub ReportException( _
   ByVal className, _
   ByVal thisException As Exception _
)
   Dim message As String
   Dim caption As String

   message = "Exception: " & thisException.Message & "." & ControlChars.CrLf & _
      "Class:   " & className & ControlChars.CrLf & _
      "Routine: " & thisException.TargetSite.Name & ControlChars.CrLf
   caption = "Exception"
   MessageBox.Show( message, caption, MessageBoxButtons.OK, _
      MessageBoxIcon.Exclamation )

End Sub
```

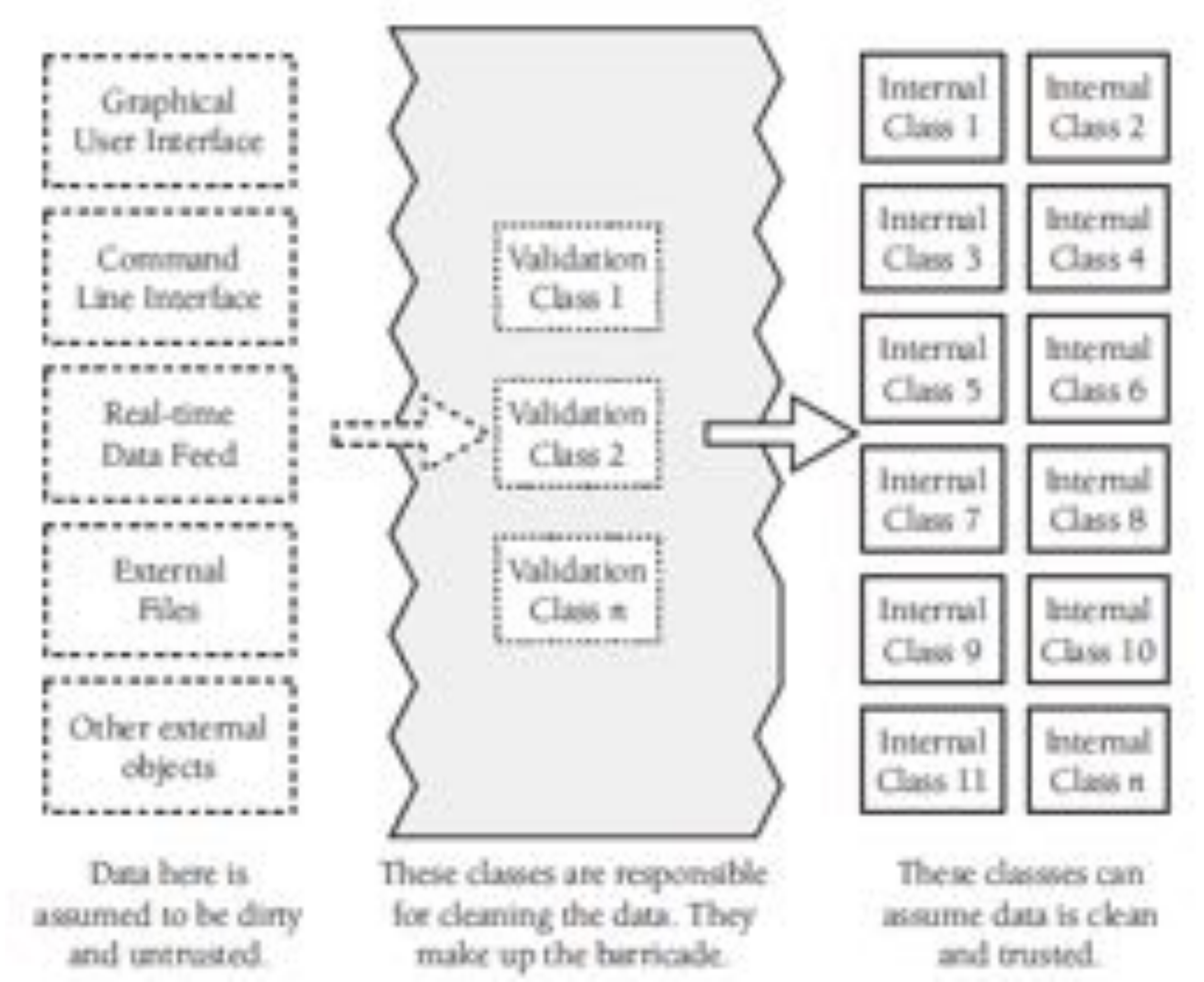


Visual Basic Example of a Centralized Exception Reporter, Part 2

```
Try
   ...
Catch exceptionObject As Exception
   ReportException( CLASS_NAME, exceptionObject )
End Try
```

Barricade Your Program to Contain the Damage Caused by Errors

Barricades are a damage-containment strategy. The reason is similar to that for having isolated compartments in the hull of a ship.



Debugging Aids

Don’t Automatically Apply Production Constraints to the Development Version

A common programmer blind spot is the assumption that limitations of the production software apply to the development version

Be willing to trade speed and resource usage during development in exchange for built-in tools that can make development go more smoothly.

Introduce Debugging Aids Early

The earlier you introduce debugging aids, the more they’ll help

Debugging Aids

Use Offensive Programming

Exceptional cases should be handled in a way that makes them obvious during development and recoverable when production code is running

- Make sure assert/abort the program. Don't allow programmers to get into the habit of just hitting the Enter key to bypass a known problem. Make the problem painful enough that it will be fixed.
- Completely fill any memory allocated so that you can detect memory allocation errors.
- Completely fill any files or streams allocated to flush out any file-format errors.
- Be sure the code in each case statement's default or else clause fails hard (aborts the program) or is otherwise impossible to overlook.
- Fill an object with junk data just before it's deleted.
- Set up the program to e-mail error log files to yourself so that you can see the kinds of errors that are occurring in the released software, if that's appropriate for the kind of software you're developing.

Debugging Aids

Plan to Remove Debugging Aids

If you're writing code for your own use, it might be fine to leave all the debugging code in the program.
If you're writing code for commercial use, the performance penalty in size and speed can be prohibitive.

Use version-control tools and build tools like ant and make

Use a built-in preprocessor

Write your own preprocessor

Use debugging stubs

Determining How Much Defensive Programming to Leave in Production Code

- Leave in code that checks for important errors
- Leave in code that helps the program crash gracefully
- Log errors for your technical support personnel
- Make sure that error messages you leave in are friendly

Being Defensive About Defensive Programming

Think about where you need to be defensive, and set your defensive programming priorities accordingly

SUMMARY

- Production code should handle errors in a more sophisticated way than “garbage in, garbage out.”
- Defensive-programming techniques make errors easier to find, easier to fix, and less damaging to production code.
- Assertions can help detect errors early, especially in large systems, high-reliability systems, and fast-changing code bases.
- The decision about how to handle bad inputs is a key error-handling decision and a key high-level design decision.
- Exceptions provide a means of handling errors that operates in a different dimension from the normal flow of the code. They are a valuable addition to the programmer’s intellectual toolbox when used with care, and they should be weighed against other error-processing techniques
- Constraints that apply to the production system do not necessarily apply to the development version. You can use that to your advantage, adding code to the development version that helps to flush out errors quickly.

つづく