SWEN 6301 Software Construction

Lecture 3: Creating High-Quality Code

Ahmed Tamrawi


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

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

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 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

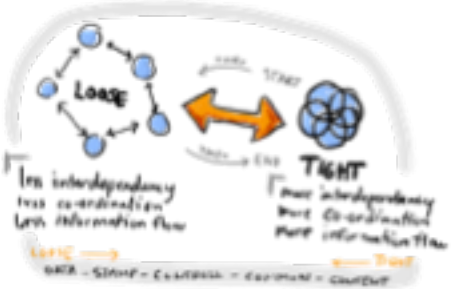
Minimal Complexity

Ease of Maintenance

Extensibility

Reusability

Portability



Loose Coupling

Leanness

High Fan-In

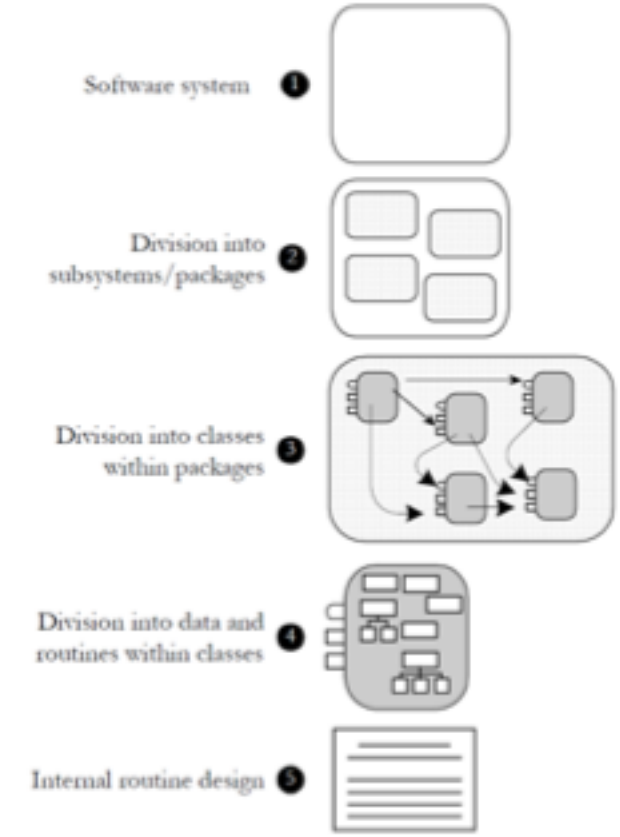
Low-to-Medium Fan-Out

Stratification

Standard Techniques

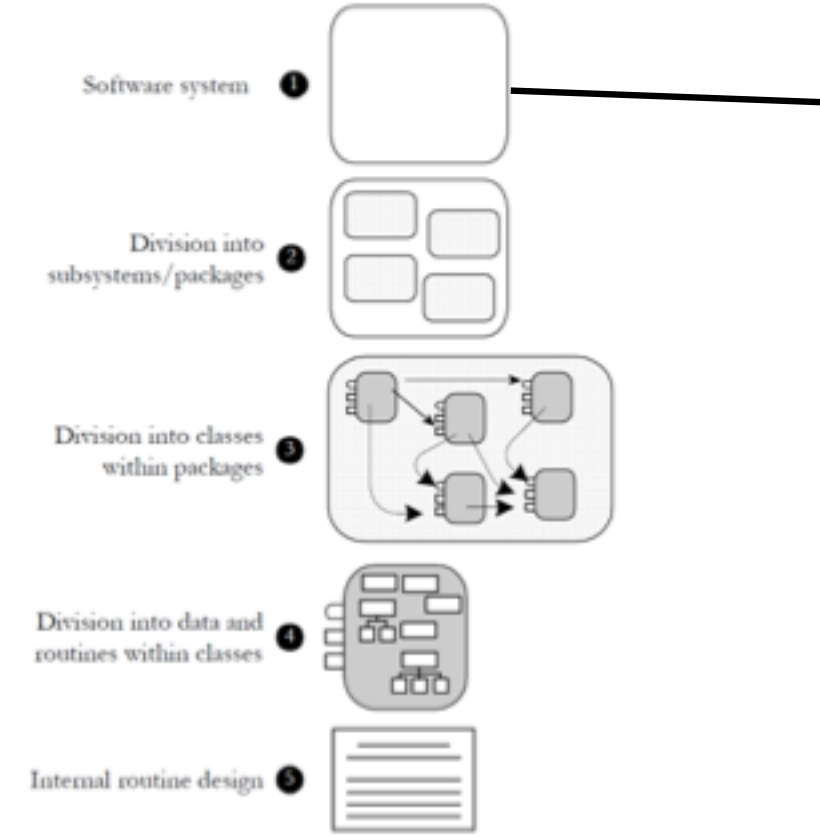
Levels of Design

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.



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

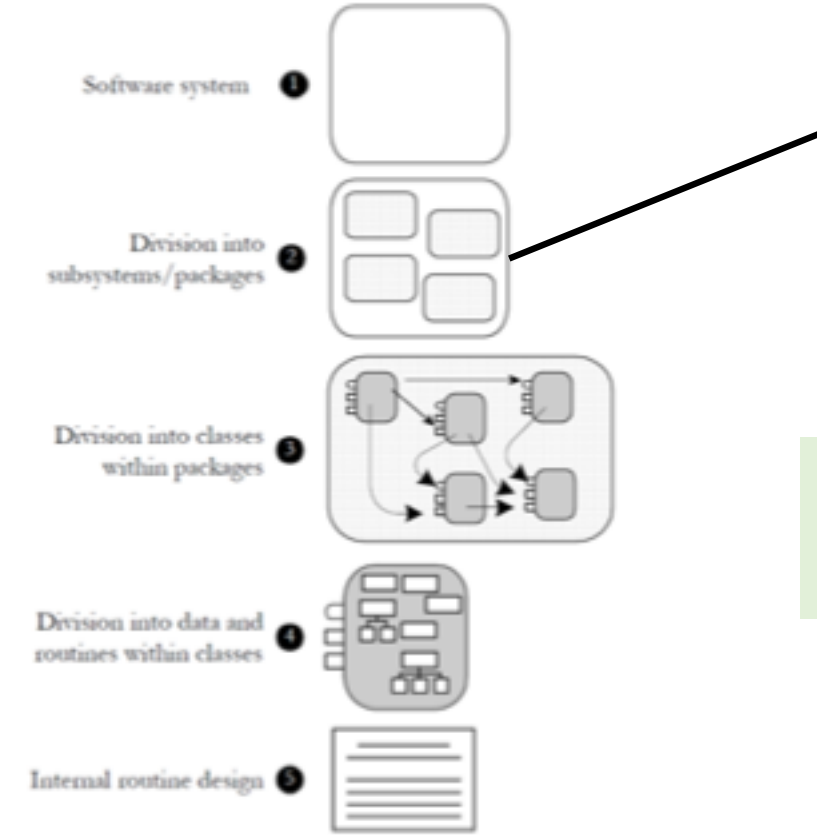


The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

1 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



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

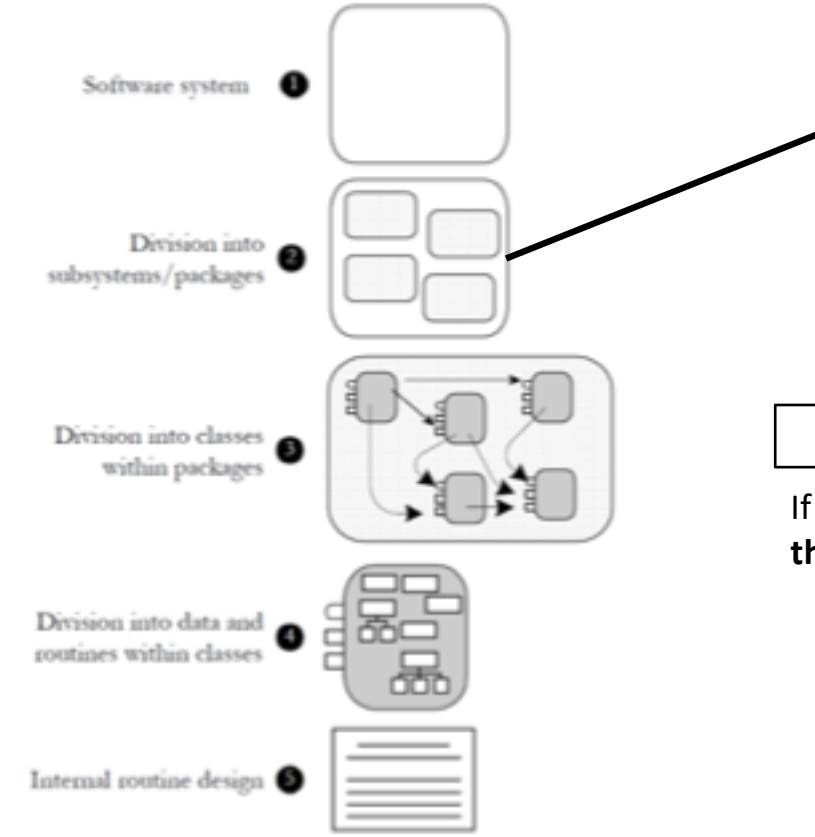
System dependencies

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

Database Access

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

Levels of Design



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

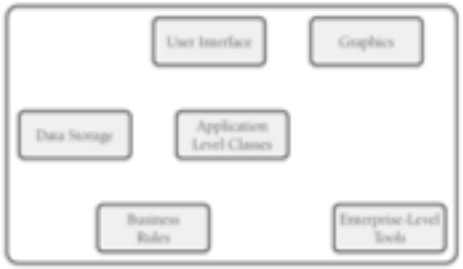
The major design activity at this level is deciding how to partition the program into major subsystems and defining how each subsystem is allowed to use each other subsystem.

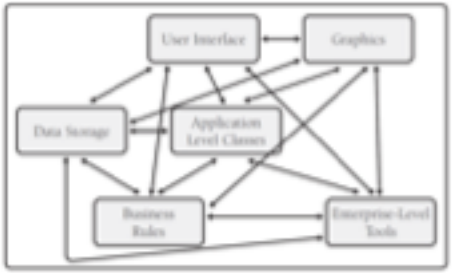
Within each subsystem, different methods of design might be used—choosing the approach that best fits each part of the system.

IMPORTANT RULE *How the various subsystems can communicate?*

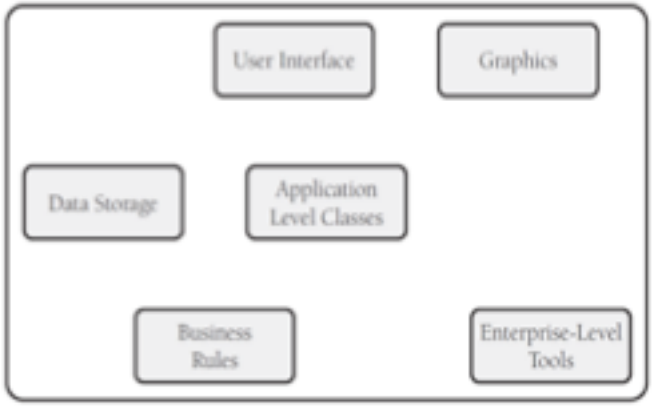
If all subsystems can communicate with all other subsystems, **you lose the benefit of separating them at all**.

Make each subsystem meaningful by restricting communications.

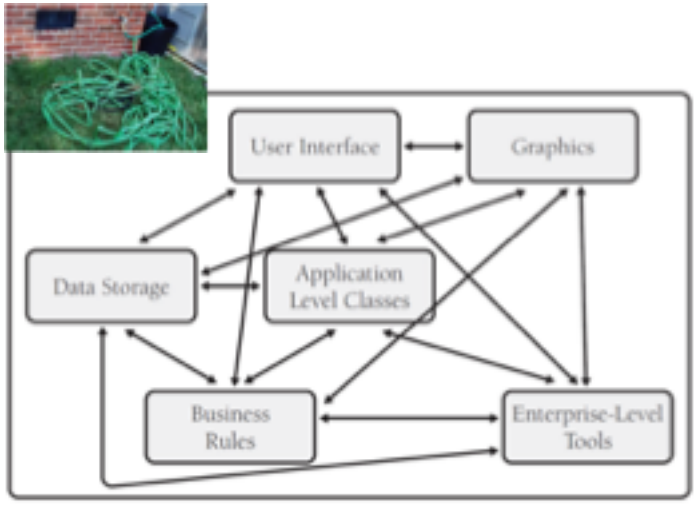




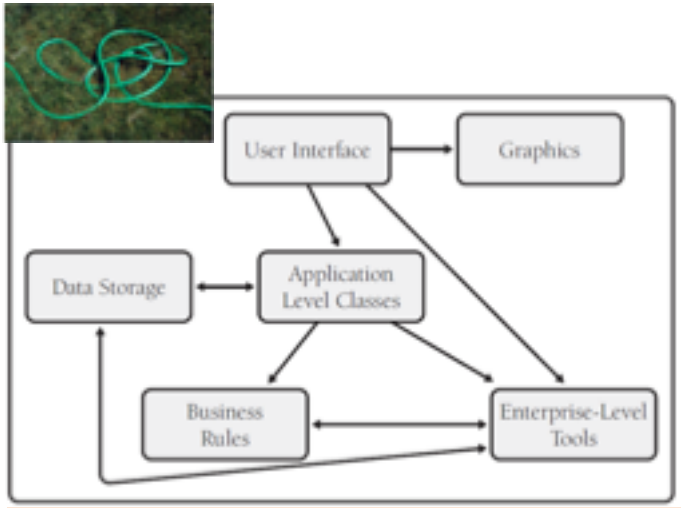
Levels of Design



Suppose for example that you define a system with six subsystems

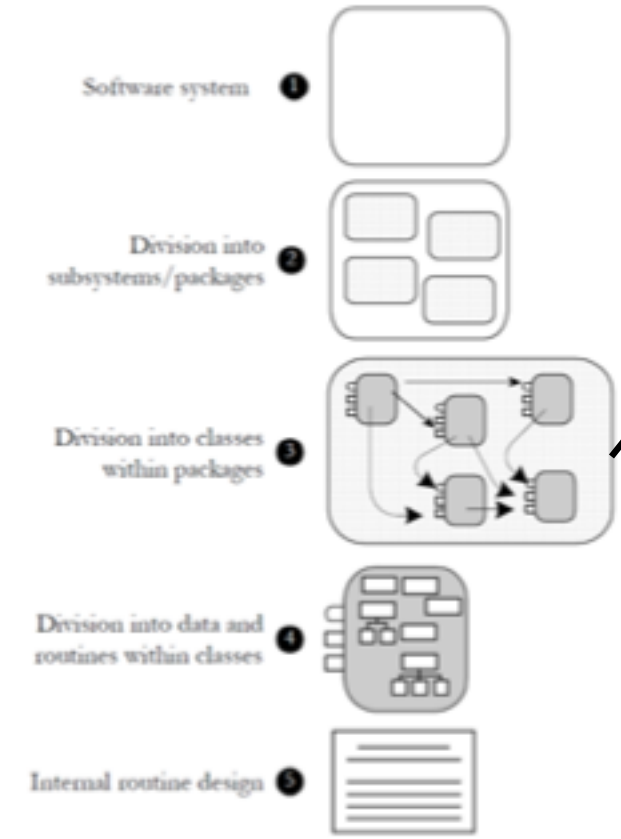


- How many different parts of the system does a developer need to understand at least a little bit to change something in the graphics subsystem?
- What happens when you try to use the business rules in another system?
- What happens when you want to put a new user interface on the system, perhaps a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



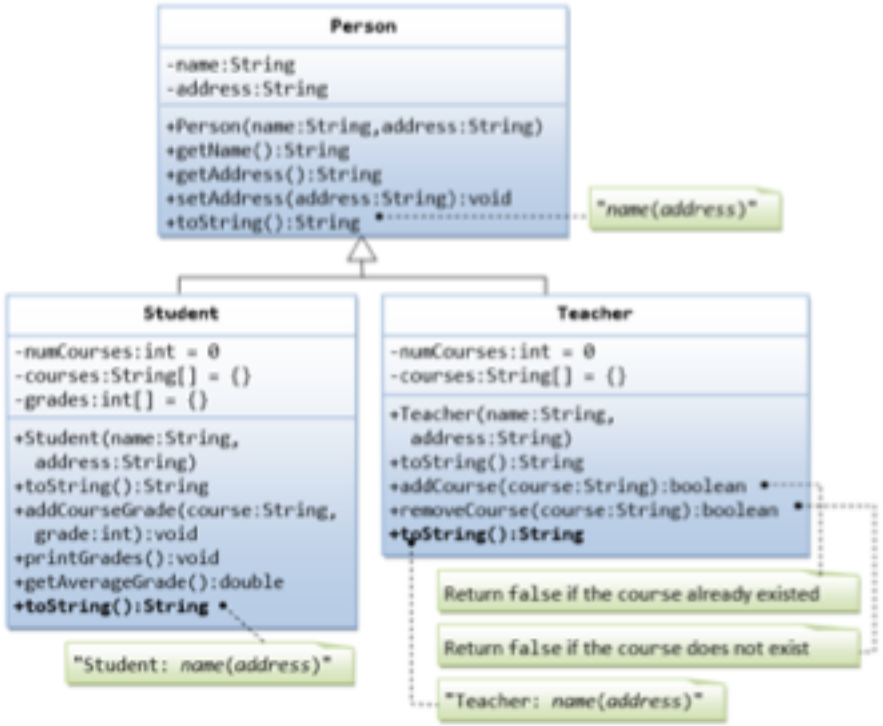
- Allow communication between subsystems only on a "**need to know**" basis—and it had better be a good reason.
- If in doubt, **it's easier to restrict communication** early and relax it later than it is to relax it early and then try to tighten it up after you've coded several hundred inter-subsystem 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



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.

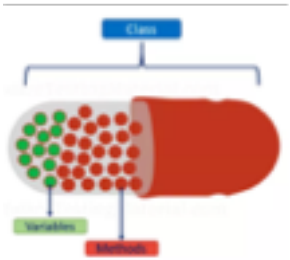


Language Rank	Types	Spectrum Ranking
1. C		100.0
2. Java		98.1
3. Python		97.9
4. C++		95.8
5. R		87.7
6. C#		86.4
7. PHP		82.4
8. JavaScript		81.9
9. Ruby		74.0
10. Go		71.5
11. Arduino		69.5
12. Matlab		68.7
13. Assembly		68.0
14. Swift		67.6
15. HTML		66.7
16. Scala		66.3
17. Perl		57.5
18. Visual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



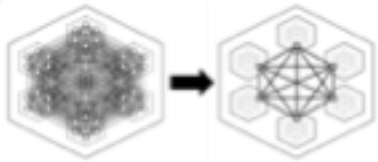
Objects

Inheritance

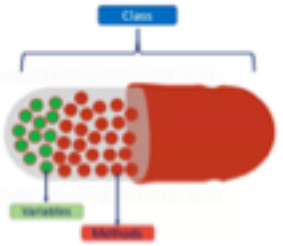
OOP Key Technologies

Polymorphism

Abstraction



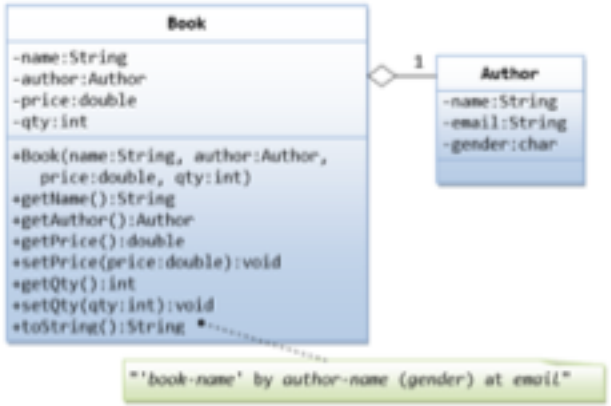
Classes



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)

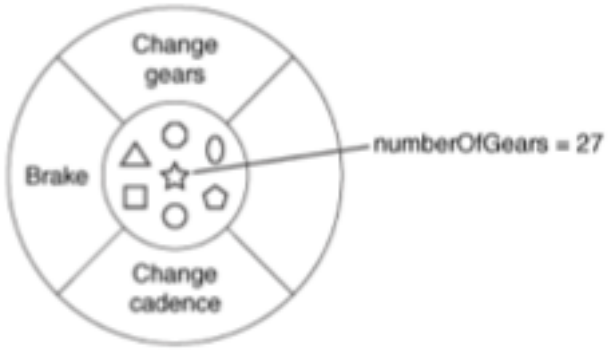
These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Real-world objects share two characteristics

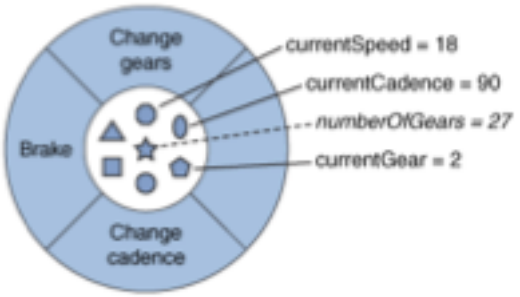
State *attribute/field*

Behavior *method/function*

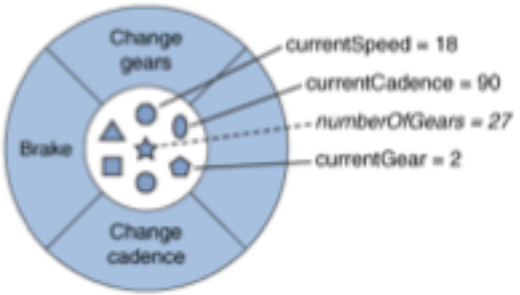
```
1  class Bike {
2     int cadence = 0;
3     int speed = 0;
4     int gear = 1;
5
6     void changeCadence(int newValue) {
7        this.cadence = newValue;
8     }
9
10    void changeGear(int newValue) {
11       this.gear = newValue;
12    }
13
14    void speedUp(int increment) {
15       this.speed += increment;
16    }
17
18    void applyBrakes(int decrement) {
19       this.speed -= decrement;
20    }
21
22    void printStates() {
23       System.out.println(
24            "cadence: " + this.cadence
25          + " speed: " + this.speed
26          + " gear: " + this.gear
27       );
28    }
29 }
```



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();
   }
}
```

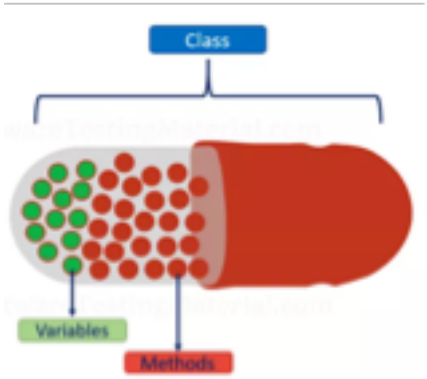
BikeDemo.java

What is the output?

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

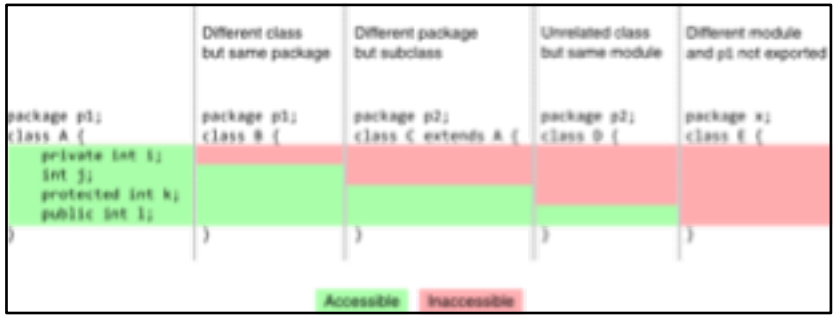
Encapsulation

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



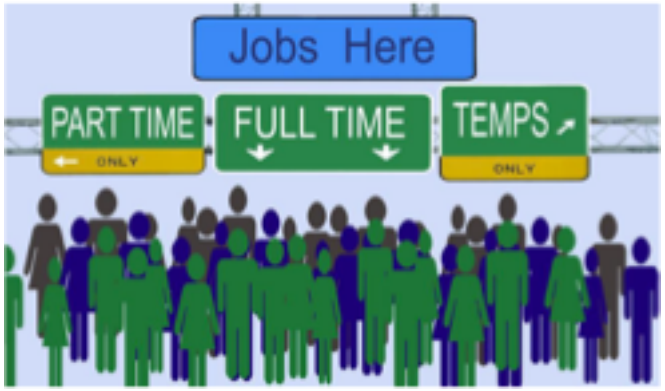
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	X
no modifier	Y	Y	X	X
private	Y	X	X	X

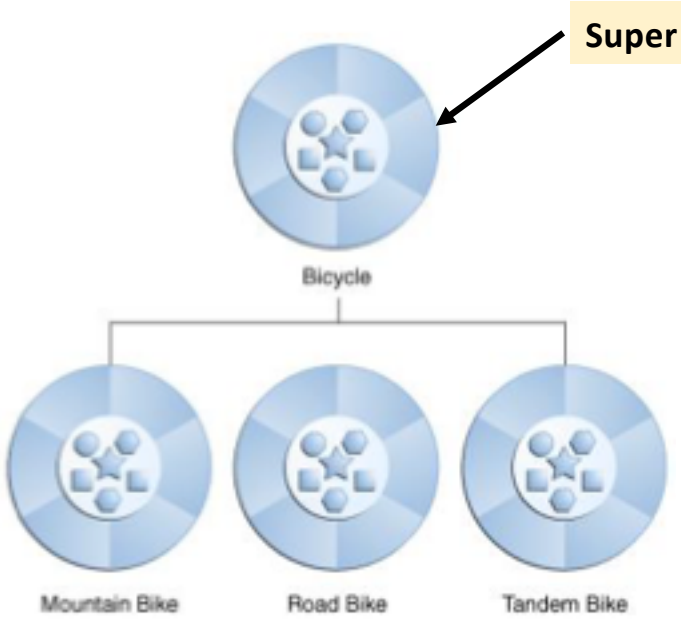
These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

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

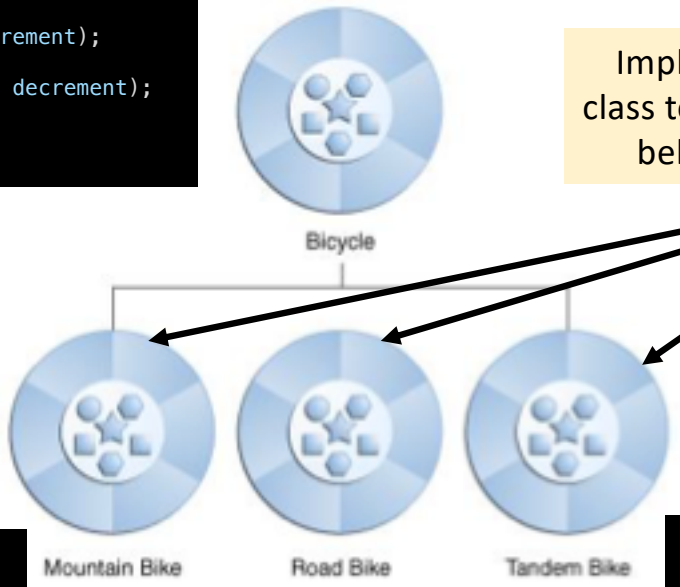
Define class instances interaction with the outside world through the methods that they expose

```java
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.




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

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

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

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

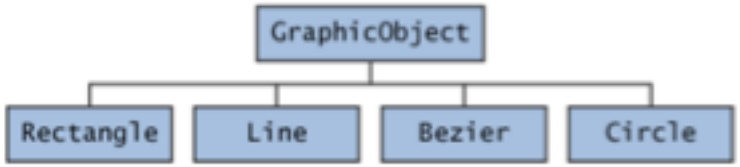

```java
public interface SomethingIsWrong {

   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*

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

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

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




```
class Rectangle extends GraphicObject {

   void draw() {
      // some implementation here
   }

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

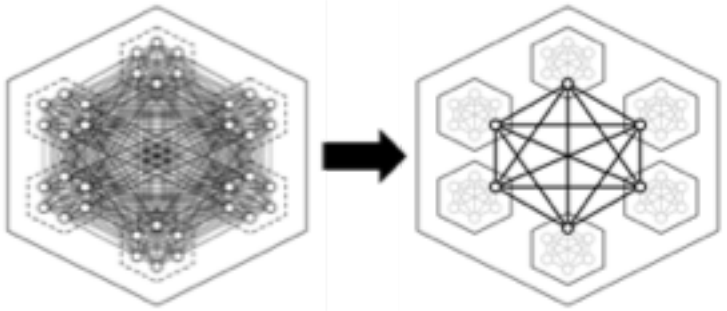
```
class Circle extends GraphicObject {

   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.*

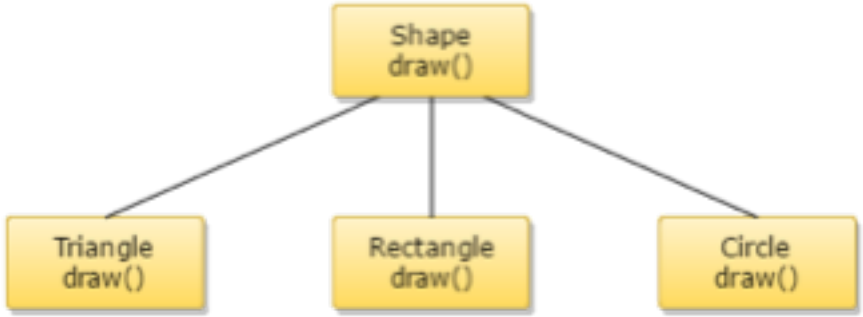
In Java, Abstraction is achieved using
Abstract classes, and **Interfaces**

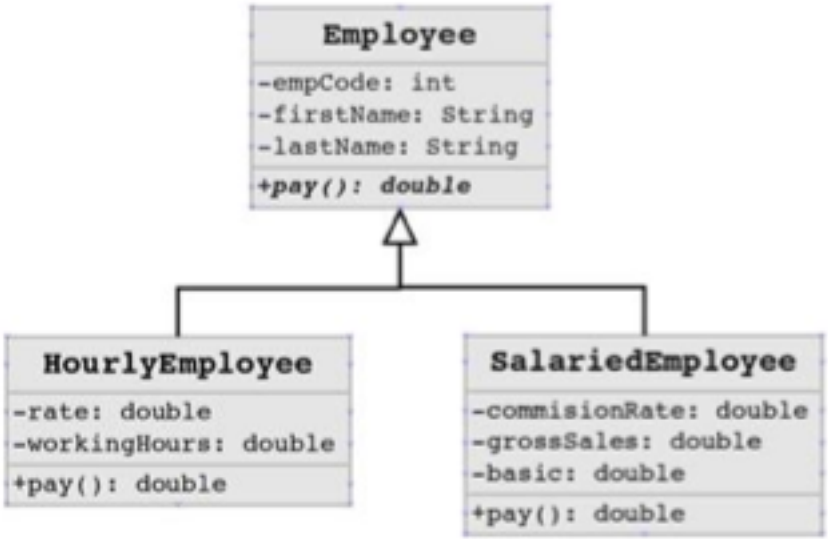


These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Polymorphism

The ability of an object to take on many forms

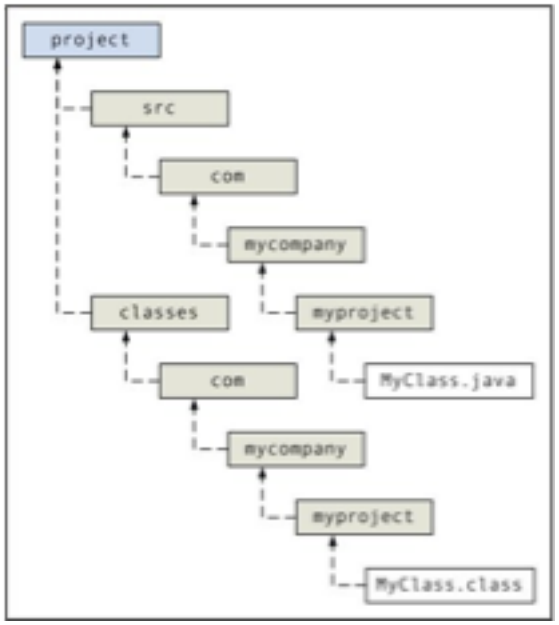


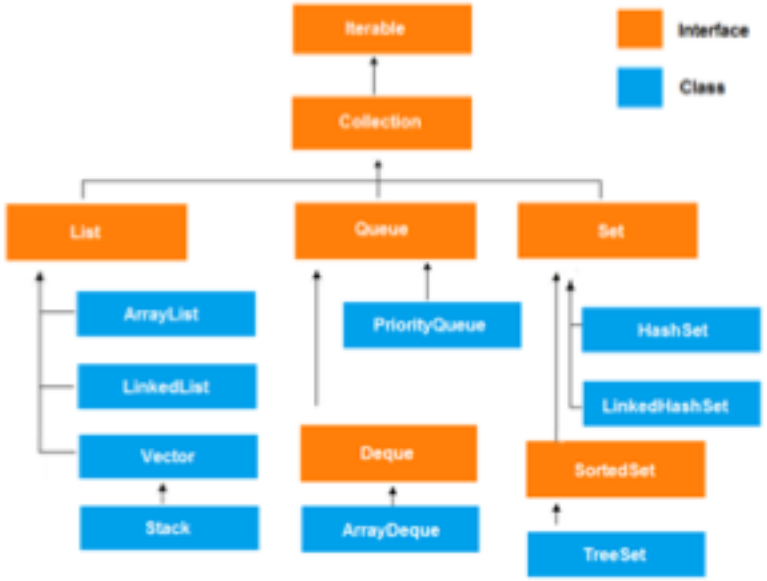


These slides are based on the material presented in https://docs.oracle.com/javase/tutorial/java/concepts/

Packages

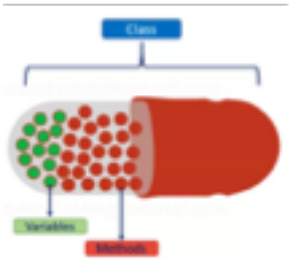
Namespaces that organize a set of related classes and interfaces





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

Encapsulation



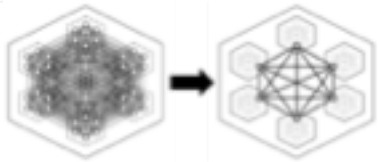
Objects

Inheritance

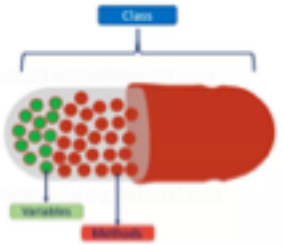
OOP Key Technologies

Polymorphism

Abstraction

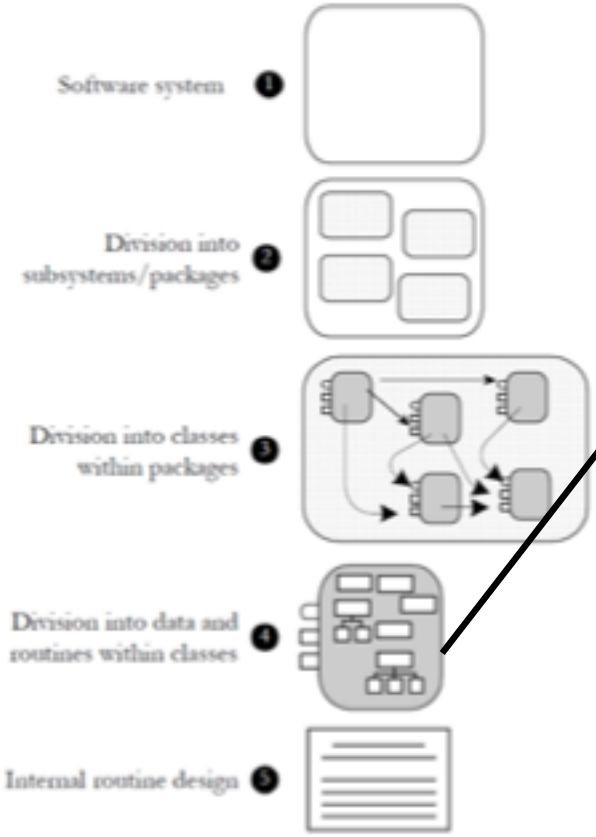


Classes



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

Levels of Design



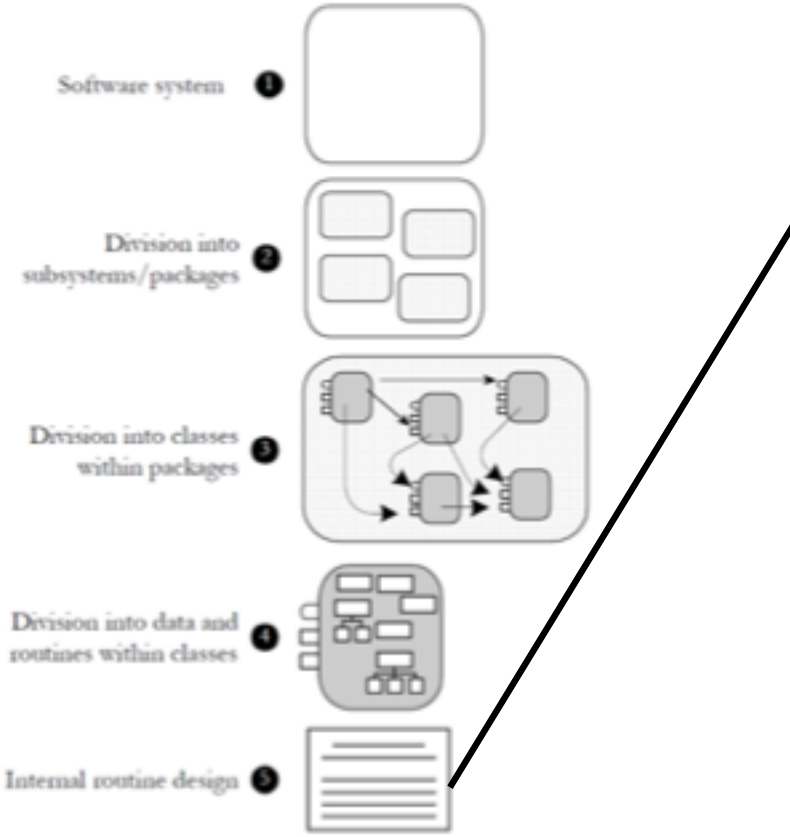
The levels of design in a program. The system (1) is first organized into subsystems (2). The subsystems are further divided into classes (3), and the classes are divided into routines and data (4). The inside of each routine is also designed (5).

❹ 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



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).

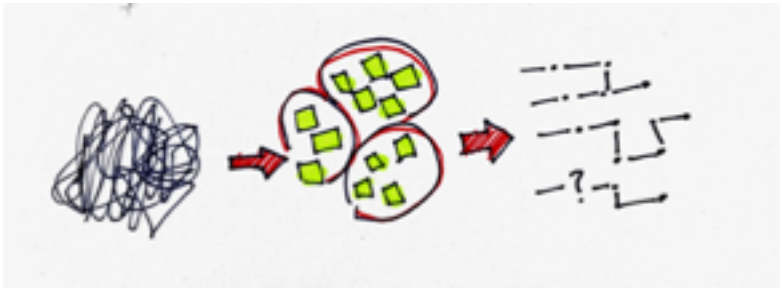
5 Internal Routine Design

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**

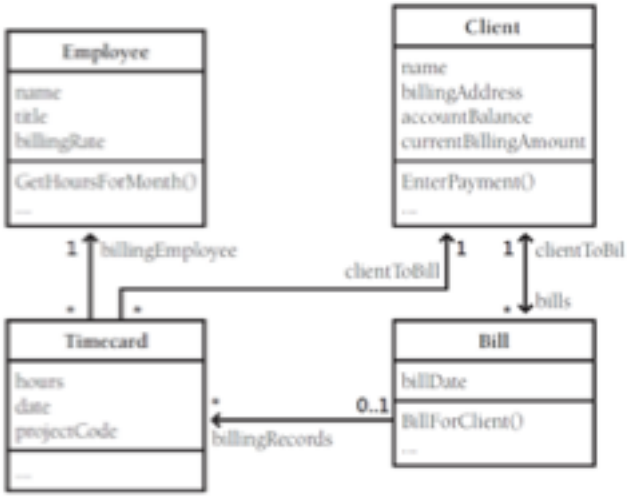
Identify the objects and their attributes (methods and data)

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**."

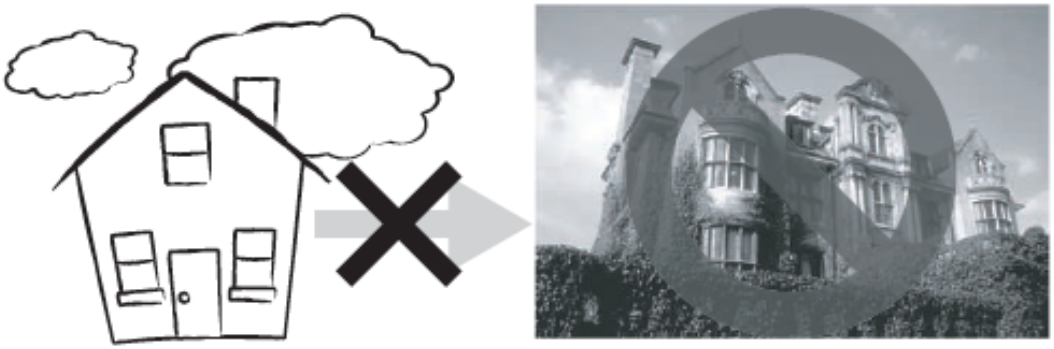
Design Heuristics: **Form Consistent Abstractions**

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.

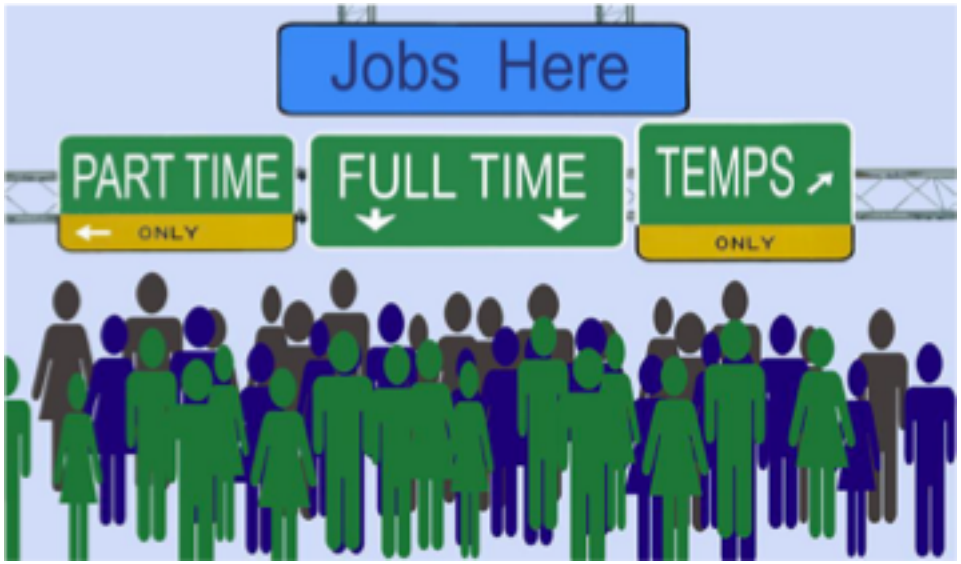


Encapsulation says that, not only are you allowed to take a simpler view of a complex concept, you are not allowed to look at any of the details of the complex concept. What you see is what you get—it's all you get!

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

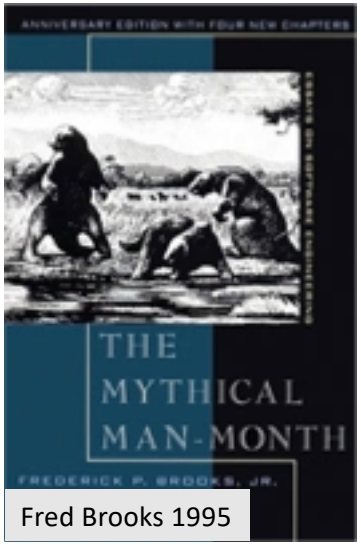


Design Heuristics: **Hide Secrets (Information Hiding)**

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.

David Parnas 1972



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)**

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

An Example of Information Hiding

Information hiding is useful at all levels of design, from the use of named constants instead of literals, to creation of data types, to class design, routine design, and subsystem design.

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

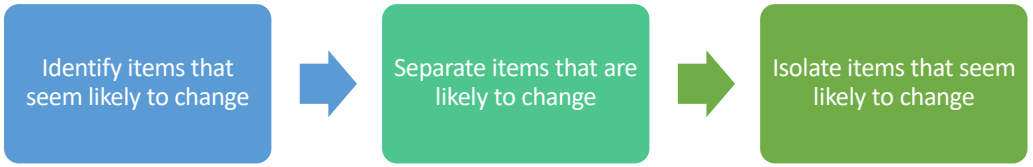
Information hiding is atheoretical techniques that has indisputably proven its value in practice, which has been true for a long time

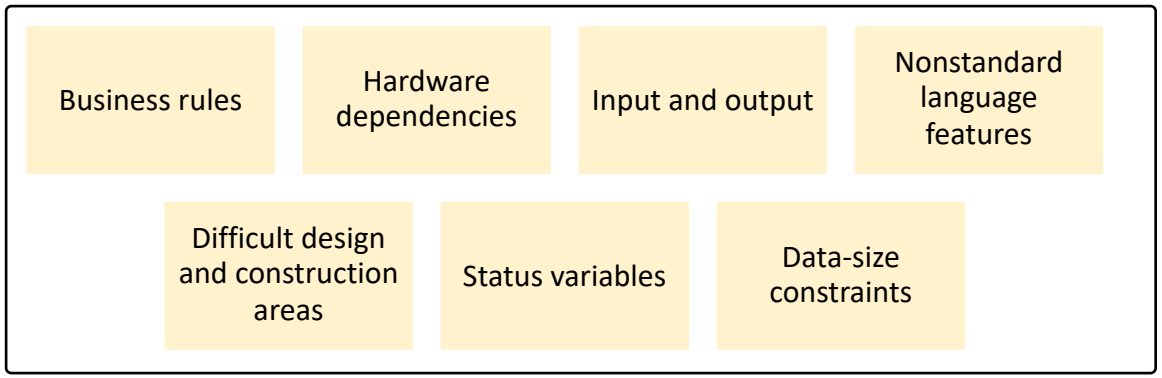
Large programs that use information hiding were found years ago to be easier to modify—by a factor of 4—than programs that don't

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



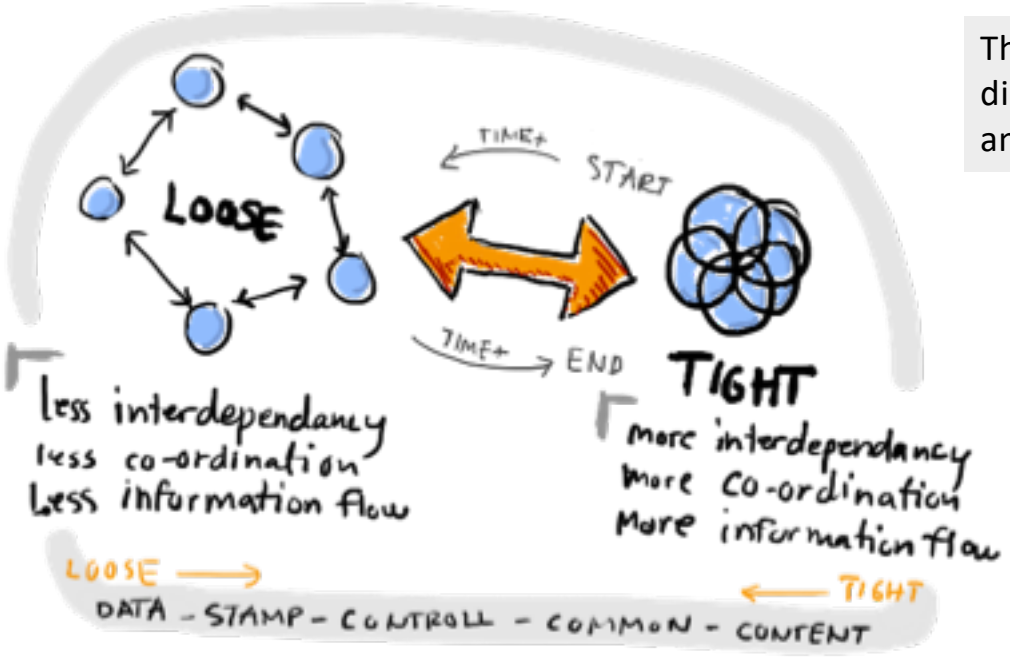


Areas Likely to Change

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.



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.

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.

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

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

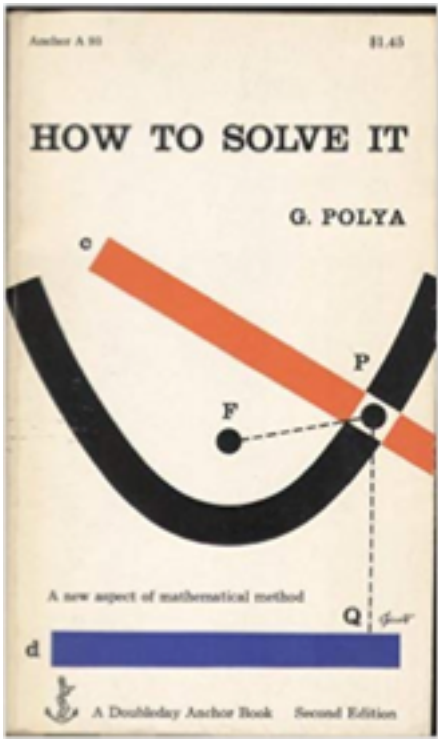
Draw a Diagram

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.

Guidelines for Using Heuristics



G. Polya developed an approach to problem solving in mathematics that's also useful in solving problems in software design

1. ***Understanding the Problem.*** You have to *understand* the problem.

   What is the unknown? What are the data? What is the condition? Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?

   Draw a figure. Introduce suitable notation. Separate the various parts of the condition. Can you write them down?

2. ***Devising a Plan.*** Find the connection between the data and the unknown. You might be obliged to consider auxiliary problems if you can't find an intermediate connection. You should eventually come up with a *plan* of the solution.

   Have you seen the problem before? Or have you seen the same problem in a slightly different form? *Do you know a related problem?* Do you know a theorem that could be useful?

   Look at the unknown! And try to think of a familiar problem having the same or a similar unknown. *Here is a problem related to yours and solved before. Can you use it?* Can you use its result? Can you use its method? Should you introduce some auxiliary element in order to make its use possible?

   Can you restate the problem? Can you restate it still differently? Go back to definitions.

   If you cannot solve the proposed problem, try to solve some related problem first. Can you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Can you solve a part of the problem? Keep only a part of the condition, drop the other part; how far is the unknown then determined, how can it vary? Can you derive something useful from the data? Can you think of other data appropriate for determining the unknown? Can you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other?

   Did you use all the data? Did you use the whole condition? Have you taken into account all essential notions involved in the problem?

3. ***Carrying out the Plan.*** *Carry out* your plan.

   Carrying out your plan of the solution, *check each step*. Can you see clearly that the step is correct? Can you prove that it's correct?

4. ***Looking Back.*** *Examine* the solution.

   Can you *check the result*? Can you check the argument? Can you derive the result differently? Can you see it at a glance?

   Can you use the result, or the method, for some other problem?

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

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

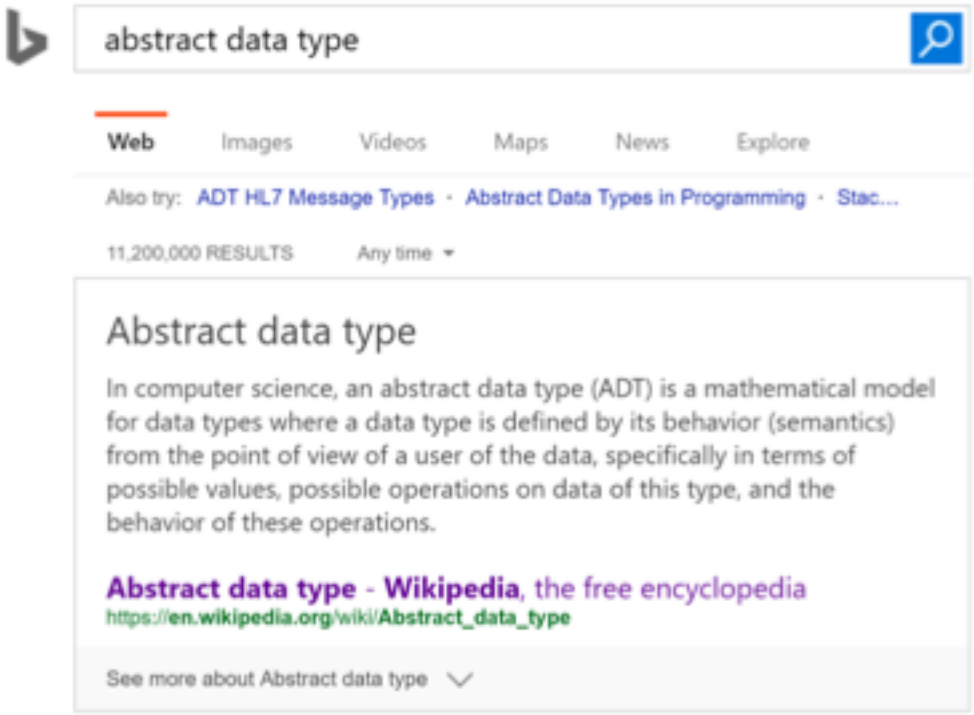
*In the dawn of computing, programmers thought about programming in terms of **statements**.*
*Throughout the 1970s and 1980s, programmers began thinking about programs in terms of **routines**.*
*In the twenty-first century, programmers think about programming in terms of **classes**.*

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.



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

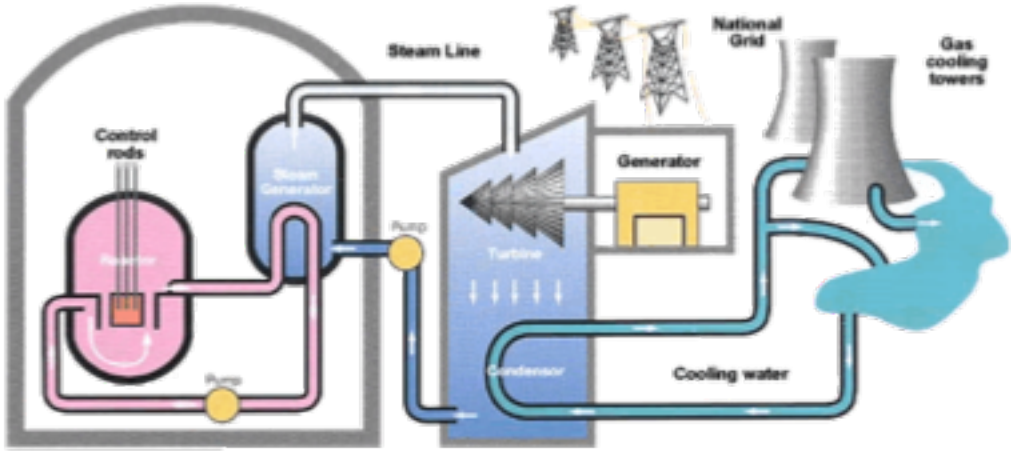
```
currentFont.size = 16
currentFont.size = PointsToPixels( 12 )
currentFont.sizeInPixels = PointsToPixels( 12 )
currentFont.sizeInPixels = PointsToPixels( 12 )
currentFont.attribute = currentFont.attribute or 0x02
currentFont.attribute = currentFont.attribute or BOLD
currentFont.bold = True
```

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

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

Abstract data types form the foundation for the concept of classes. In languages that support classes, you can implement each abstract data type as its own class.

Classes usually involve the additional concepts of inheritance and polymorphism. One way of thinking of a class is as an abstract data type plus inheritance and polymorphism.

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**

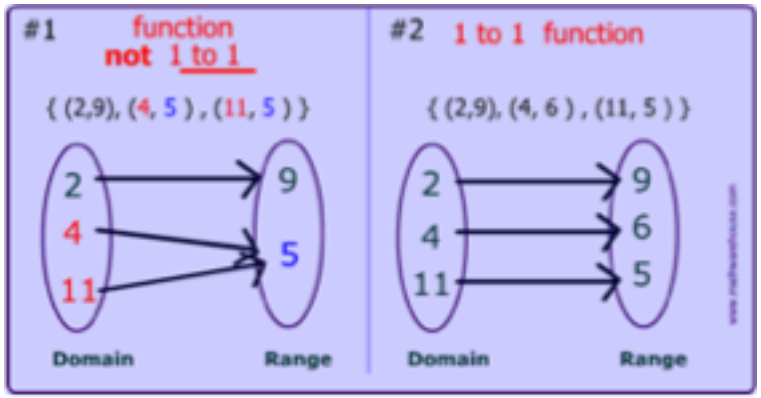
The pursuit of good abstract interfaces gives rise to several guidelines for creating class interfaces.

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

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

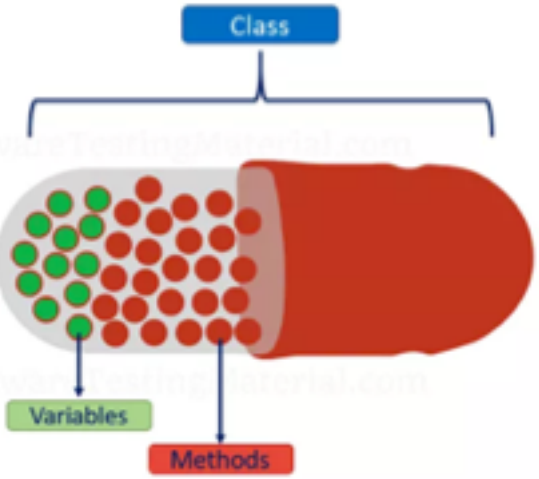
Make interfaces programmatic rather than semantic when possible

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

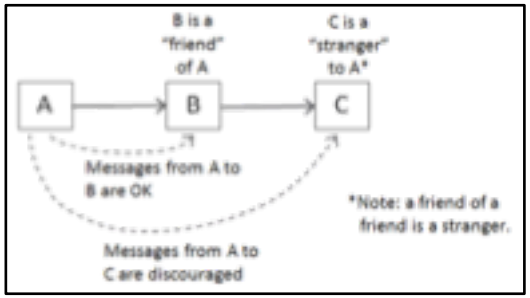
Minimize accessibility of classes and members.

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"



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

Here is the private constructor.

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

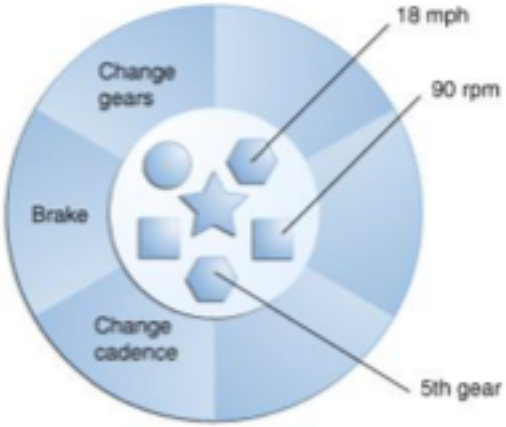
Here is the single instance.

```
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();
   ...
}
```

Reasons to Create a Class

Model real-world objects

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

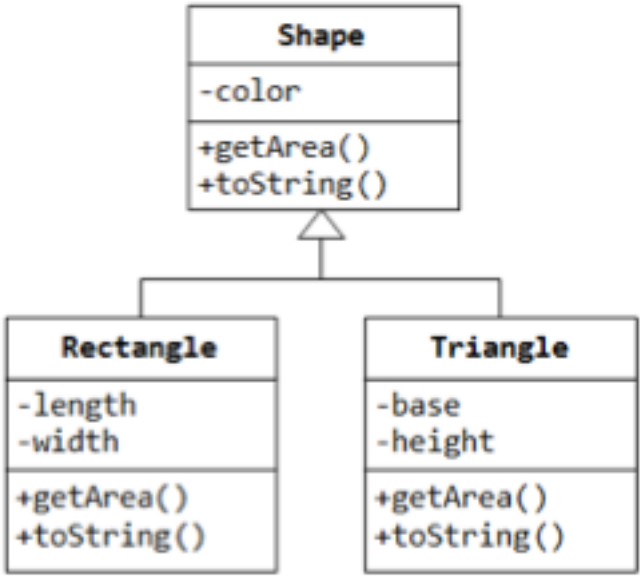


Reasons to Create a Class

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

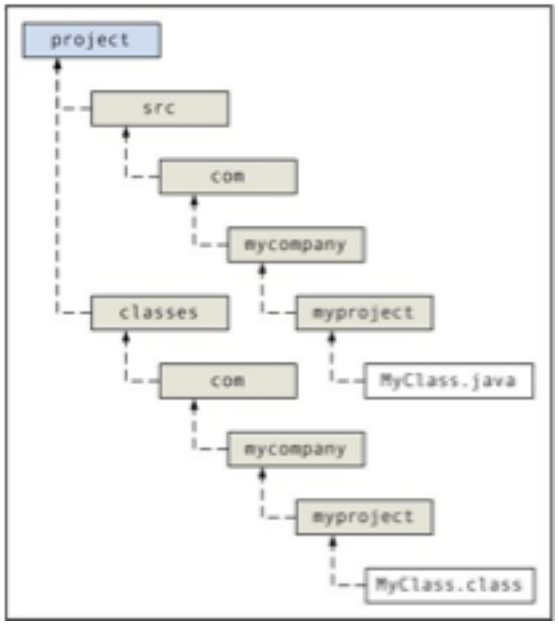
If a class consists only of data but no behavior, ask yourself whether it's really a class and consider demoting it so that its member data just becomes attributes of one or more other classes.

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;
}
```

CHECKLIST: Class Quality

Abstract Data Types

- ❑ Have you thought of the classes in your program as abstract data types and evaluated their interfaces from that point of view?

Abstraction

- ❑ Does the class have a central purpose?
- ❑ Is the class well named, and does its name describe its central purpose?
- ❑ Does the class's interface present a consistent abstraction?
- ❑ Does the class's interface make obvious how you should use the class?
- ❑ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
- ❑ Are the class's services complete enough that other classes don't have to meddle with its internal data?
- ❑ Has unrelated information been moved out of the class?
- ❑ Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
- ❑ Are you preserving the integrity of the class's interface as you modify the class?

Encapsulation

- ❑ Does the class minimize accessibility to its members?
- ❑ Does the class avoid exposing member data?
- ❑ Does the class hide its implementation details from other classes as much as the programming language permits?
- ❑ Does the class avoid making assumptions about its users, including its derived classes?
- ❑ Is the class independent of other classes? Is it loosely coupled?

Inheritance

- ❑ Is inheritance used only to model "is a" relationships—that is, do derived classes adhere to the Liskov Substitution Principle?
- ❑ Does the class documentation describe the inheritance strategy?
- ❑ Do derived classes avoid "overriding" non-overridable routines?
- ❑ Are common interfaces, data, and behavior as high as possible in the inheritance tree?
- ❑ Are inheritance trees fairly shallow?
- ❑ Are all data members in the base class private rather than protected?

Other Implementation Issues

- ❑ Does the class contain about seven data members or fewer?
- ❑ Does the class minimize direct and indirect routine calls to other classes?
- ❑ Does the class collaborate with other classes only to the extent absolutely necessary?
- ❑ Is all member data initialized in the constructor?
- ❑ Is the class designed to be used as deep copies rather than shallow copies unless there's a measured reason to create shallow copies?

Language-Specific Issues

- ❑ Have you investigated the language-specific issues for classes in your specific programming language?

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.

つづく