Program Design and Analysis 6

Weeks of coding can save you hours of planning.
—Anonymous

- → **Program development, including design and testing**
- → **Object-oriented program design**
- → **Relationships between classes**
- → **Program analysis**
- → **Efficiency**

Students of introductory computer science typically see themselves as programmers. They no sooner have a new programming project in their heads than they're at the computer, typing madly to get some code up and running. (Is this you?)

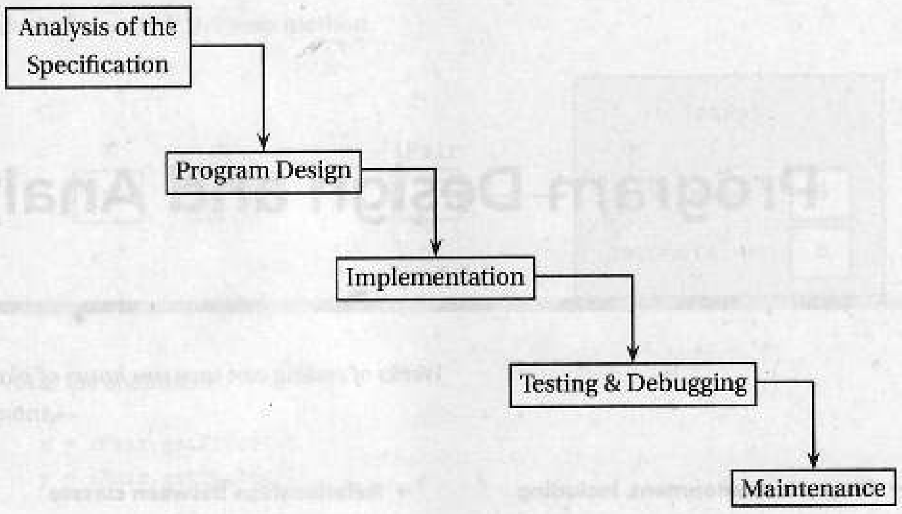
To succeed as a programmer, however, you have to combine the practical skills of a software engineer with the analytical mindset of a computer scientist. A software engineer oversees the life cycle of software development: initiation of the project, analysis of the specification, and design of the program, as well as implementation, testing, and maintenance of the final product. A computer scientist (among other things!) analyzes the implementation, correctness, and efficiency of algorithms. All these topics are tested on the APCS exam.

THE SOFTWARE DEVELOPMENT LIFE CYCLE

The Waterfall Model

The waterfall model of software development came about in the 1960s in order to bring structure and efficiency into the process of creating large programs.

Each step in the process flows into the next: The picture resembles a waterfall.



Program Specification

The *specification* is an explicit written description of the project. Typically it is based on a customer's (or a teacher's!) requirements. The first step in writing a program is to analyze the specification, make sure you understand it, and clarify with the customer anything that is unclear.

Program Design

Even for a small-scale program a good design can save programming time and enhance the reliability of the final program. The design is a fairly detailed plan for solving the problem outlined in the specification. It should include all objects that will be used in the solution, the data structures that will implement them, plus a detailed list of the tasks to be performed by the program.

A good design provides a fairly detailed overall plan at a glance, without including the minutiae of Java code.

Program Implementation

Program implementation is the coding phase. Design and implementation are discussed in more detail on p. 224.

Testing and Debugging

TEST DATA

Not every possible input value can be tested, so a programmer should be diligent in selecting a representative set of *test data*. Typical values in each part of a domain of the program should be selected, as well as endpoint values and out-of-range values. If only positive input is required, your test data should include a negative value just to check that your program handles it appropriately.

➡ Example

A program must be written to insert a value into its correct position in this sorted list:

```
2  5  9
```

Test data should include

- A value less than 2
- A value between 2 and 5
- A value between 5 and 9
- A value greater than 9
- 2, 5, and 9
- A negative value

TYPES OF ERRORS (BUGS)

- A *compile-time error* occurs during compilation of the program. The compiler is unable to translate the program into bytecode and prints an appropriate error message. A *syntax error* is a compile-time error caused by violating the rules of the programming language. Some examples are omitting semicolons or braces, using undeclared identifiers, using keywords inappropriately, having parameters that don't match in type and number, and invoking a method for an object whose class definition doesn't contain that method.
- A *run-time error* occurs during execution of the program. The Java run-time environment *throws an exception*, which means that it stops execution and prints an error message. Typical causes of run-time errors include attempting to divide an integer by zero, using an array index that is out of bounds, attempting to open a file that cannot be found, and so on. An error that causes a program to run forever ("infinite loop") can also be regarded as a run-time error. (See also Errors and Exceptions, p. 85.)
- An *intent* or *logic error* is one that fails to carry out the specification of the program. The program compiles and runs but does not do the job. These are sometimes the hardest types of errors to fix.

ROBUSTNESS

Always assume that any user of your program is not as smart as you are. You must therefore aim to write a *robust* program, namely one that

- Won't give inaccurate answers for some input data.
- Won't crash if the input data are invalid.
- Won't allow execution to proceed if invalid data are entered.

Examples of bad input data include out-of-range numbers, characters instead of numerical data, and a response of "maybe" when "yes" or "no" was asked for.

Note that bad input data that invalidates a computation won't be detected by Java. Your program should include code that catches the error, allows the error to be fixed, and allows program execution to resume.

Program Maintenance

Program maintenance involves upgrading the code as circumstances change. New features may be added. New programmers may come on board. To make their task easier, the original program must have clear and precise documentation.

OBJECT-ORIENTED PROGRAM DESIGN

Object-oriented programming has been the dominant programming methodology since the mid 1990s. It uses an approach that blurs the lines of the waterfall model. Analysis of the problem, development of the design, and pieces of the implementation all overlap and influence one another.

Here are the steps in object-oriented design:

- Identify classes to be written.
- Identify behaviors (i.e., methods) for each class.
- Determine the relationships between classes.
- Write the interface (public method headers) for each class.
- Implement the methods.

Identifying Classes

Identify the objects in the program by picking out the nouns in the program specification. Ignore pronouns and nouns that refer to the user. Select those nouns that seem suitable as classes, the "big-picture" nouns that describe the major objects in the application. Some of the other nouns may end up as attributes of the classes.

Many applications have similar object types: a low-level basic component; a collection of low-level components; a controlling object that puts everything together; and a display object that could be a GUI (graphical user interface) but doesn't have to be.

➡ Example 1

Write a program that maintains an inventory of stock items for a small store.

Nouns to consider: inventory, item, store.

Basic Object:	`StockItem`
Collection:	`Inventory` (a list of `StockItems`)
Controller:	`Store` (has an `Inventory`, uses a `StoreDisplay`)
Display:	`StoreDisplay` (could be a GUI)

➡ Example 2

Write a program that simulates a game of bingo. There should be at least two players, each of whom has a bingo card, and a caller who calls the numbers.

Nouns to consider: game, players, bingo card, caller.

Basic Objects:	`BingoCard`, `Caller`
Collection:	`Players` (each has a `BingoCard`)
Controller:	`GameMaster` (sets up the `Players` and `Caller`)
Display:	`BingoDisplay` (shows each player's card and displays winners, etc.)

➡ Example 3

Write a program that creates random bridge deals and displays them in a specified format. (The specification defines a "deal" as consisting of four hands. It also describes a deck of cards, and shows how each card should be displayed.)

Nouns to consider: deal, hand, format, deck, card.

Basic Object:	`Card`
Collection:	`Deck` (has an array of `Cards`)
	`Hand` (has an array of `Cards`)
	`Deal` (has an array of `Hands`)
	`Dealer` (has a `Deck`, or several `Decks`)
Controller:	`Formatter` (has a `Deal` and a `TableDisplay`)
Display:	`TableDisplay` (could be a GUI)

Identifying Behaviors

Find all verbs in the program description that help lead to the solution of the programming task. These are likely behaviors that will probably become the methods of the classes. Now decide which methods belong in which classes. Recall that the process of bundling a group of methods and data fields into a class is called *encapsulation*.

Think carefully about who should do what. Do not ask a basic object to perform operations for the group. For example, a `StockItem` should keep track of its own details (price, description, how many on the shelf, etc.) but should not be required to search for another item. A `Card` should know its value and suit but should not be responsible for keeping track of how many cards are left in a deck. A `Caller` in a bingo game should be responsible for keeping track of the numbers called so far and for producing the next number but not for checking whether a player has bingo: That is the job of an individual player (element of `Players`) and his `BingoCard`.

You will also need to decide which data fields each class will need and which data structures should store them. For example, if an object represents a list of items, consider an array or `ArrayList` as the data structure.

Determining Relationships Between Classes

INHERITANCE RELATIONSHIPS

Look for classes with common behaviors. This will help identify *inheritance relationships*. Recall the *is-a* relationship—if `object1` *is-a* `object2`, then `object2` is a candidate for a superclass.

COMPOSITION RELATIONSHIPS

Composition relationships are defined by the *has-a* relationship. For example, a `Nurse` *has-a* `Uniform`. Typically, if two classes have a composition relationship, one of them contains an instance variable whose type is the other class.

Note that a wrapper class always implements a *has-a* relationship with any objects that it wraps.

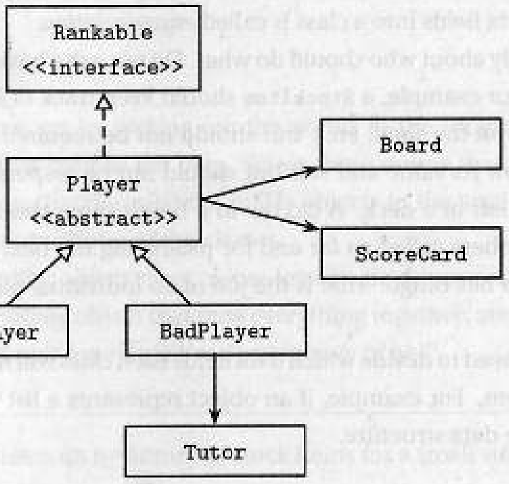
UML Diagrams

An excellent way to keep track of the relationships between classes and show the inheritance hierarchy in your programs is with a UML (Unified Modeling Language) diagram. This is a standard graphical scheme used by object-oriented programmers. Although it is not part of the AP subset, on the AP exam you may be expected to interpret simple UML diagrams and inheritance hierarchies.

Here is a simplified version of the UML rules:

- Represent classes with rectangles.
- Use angle brackets with the word "abstract" or "interface" to indicate either an abstract class or interface.
- Show the *is-a* relationship between classes with an open up-arrow.
- Show the *is-a* relationship that involves an interface with an open, dotted up-arrow.
- Show the *has-a* relationship with a down arrow or sideways arrow (indicates composition).

➡ Example



From this diagram you can see at a glance that `GoodPlayer` and `BadPlayer` are subclasses of an abstract class `Player`, and that each `Player` implements the `Rankable` interface. Every `Player` has a `Board` and a `ScoreCard`, while only the `BadPlayer` has a `Tutor`.

Implementing Classes

BOTTOM-UP DEVELOPMENT

For each method in a class, list all of the other classes needed to implement that particular method. These classes are called *collaborators*. A class that has no collaborators is *independent*.

To implement the classes, often an incremental, *bottom-up* approach is used. This means that independent classes are fully implemented and tested before being incorporated into the overall project. Typically, these are the basic objects of the program, like `StockItem`, `Card`, and `BingoCard`. Unrelated classes in a programming project can be implemented by different programmers.

Note that a class can be tested using a dummy `Tester` class that will be discarded when the methods of the class are working. Constructors, then methods, should be added, and tested, one at a time. A *driver class* that contains a `main` method can be used to test the program as you go. The purpose of the driver is to test the class fully before incorporating it as an object in a new class.

When each of the independent classes is working, classes that depend on just one other class are implemented and tested, and so on. This may lead to a working, bare bones version of the project. New features and enhancements can be added later.

Design flaws can be corrected at each stage of development. Remember, a design is never set in stone: It simply guides the implementation.

TOP-DOWN DEVELOPMENT

In a top-down design, the programmer starts with an overview of the program, selecting the highest-level controlling object and the tasks needed. During development of the program, subsidiary classes may be added to simplify existing classes.

Implementing Methods

PROCEDURAL ABSTRACTION

A good programmer avoids chunks of repeated code wherever possible. To this end, if several methods in a class require the same task, like a search or a swap, you should use *helper methods*. The `reduce` method in the `Rational` class on p. 129 is an example of such a method. Also, wherever possible you should enhance the readability of your code by using helper methods to break long methods into smaller tasks. The use of helper methods within a class is known as *procedural abstraction* and is an example of top-down development within a class. This process of breaking a long method into a sequence of smaller tasks is sometimes called *stepwise refinement*.

INFORMATION HIDING

Instance variables and helper methods are generally declared as `private`, which prevents client classes from accessing them. This strategy is called *information hiding*.

STUB METHOD

Sometimes it makes more sense in the development of a class to test a calling method before testing a method it invokes. A *stub* is a dummy method that stands in for a method until the actual method has been written and tested. A stub typically has an output statement to show that it was called in the correct place, or it may return some reasonable values if necessary.

ALGORITHM

An *algorithm* is a precise step-by-step procedure that solves a problem or achieves a goal. Don't write any code for an algorithm in a method until the steps are completely clear to you.

➡ Example 1

A program must test the validity of a four-digit code number that a person will enter to be able to use a photocopy machine. The number is valid if the fourth digit equals the remainder when the sum of the first three digits is divided by seven.

Classes in the program may include an `IDNumber`, the four-digit code; `Display`, which would handle input and output; and `IDMain`, the driver for the program. The data structure used to implement an `IDNumber` could be an instance variable of type `int`, or an instance variable of type `String`, or four instance variables of type `int`—one per digit, and so on.

A top-down design for the program that tests the validity of the number is reflected in the steps of the `main` method of `IDMain`:

```
Create Display
Read in IDNumber
Check validity
Print message
```

Each method in this design is tested before the next method is added to `main`. If the display will be handled in a GUI (graphical user interface), stepwise refinement of the design might look like this:

```
Create Display
    Construct a Display
    Create window panels
    Set up text fields
    Add panels and fields to window
Read in IDNumber
    Prompt and read

Check validity of IDNumber
  Check input
     Check characters
     Check range
  Separate into digits
  Check validity property

Print message
  Write number
  State if valid
```

NOTE

1. The `IDNumber` class, which contains the four-digit code, is responsible for the following operations:

   Split value into separate digits
   Check condition for validity

   The `Display` class, which contains objects to read and display, must also contain an `IDNumber` object. It is responsible for the following operations:

   Set up display
   Read in code number
   Display validity message

   Creating these two classes with their data fields (instance variables) and operations (methods) is an example of encapsulation.

2. The `Display` method `readCodeNumber` needs private helper methods to check the input: `checkCharacters` and `checkRange`. This is an example of procedural abstraction (use of helper methods) and information hiding (making them private).
3. Initially the programmer had just an `IDNumber` class and a driver class. The `Display` class was added as a refinement, when it was realized that handling the input and message display was separate from checking the validity of the `IDNumber`. This is an example of top-down development (adding an auxiliary class to clarify the code).
4. The `IDNumber` class contains no data fields that are objects. It is therefore an independent class. The `Display` class, which contains an `IDNumber` data member, has a composition relationship with `IDNumber` (`Display` *has-a* `IDNumber`).
5. When testing the final program, the programmer should be sure to include each of the following as a user-entered code number: a valid four-digit number, an invalid four-digit number, an n-digit number, where $n \neq 4$, and a "number" that contains a nondigit character. A robust program should be able to deal with all these cases.

➡ Example 2

A program must create a teacher's grade book. The program should maintain a class list of students for any number of classes in the teacher's schedule. A menu should be provided that allows the teacher to

- Create a new class of students.
- Enter a set of scores for any class.
- Correct any data that's been entered.
- Display the record of any student.
- Calculate the final average and grade for all students in a class.
- Print a class list, with or without grades.
- Add a student, delete a student, or transfer a student to another class.
- Save all the data in a file.

IDENTIFYING CLASSES

Use nouns in the specification to identify possible classes.

Use the nouns in the specification as a starting point for identifying classes in the program. The nouns are: program, teacher, grade book, class list, class, student, schedule, menu, set of scores, data, record, average, grade, and file.

Eliminate each of the following:

program	(Always eliminate "program" when used in this context.)
teacher	(Eliminate, because he or she is the user.)
schedule	(This will be reflected in the name of the external file for each class, e.g., `apcs_period3.dat`.)
data, record	(These are synonymous with student name, scores, grades, etc., and will be covered by these features.)
class	(This is synonymous with class list.)

The following seem to be excellent candidates for classes: `GradeBook`, `ClassList`, `Student`, and `FileHandler`. Other possibilities are `Menu`, `ScoreList`, and a `GUI_Display`.

On further thought: Basic independent objects are `Student`, `Menu`, `Score`, and `FileHandler`. Group objects are `ClassList` (collection of students), `ScoreList` (collection of scores), and

AllClasses (collection of ClassLists). The controlling class is the GradeBook. A Display class is essential for many of the grade book operations, like showing a class list or displaying information for a single student.

RELATIONSHIPS BETWEEN CLASSES

There are no inheritance relationships. There are many composition relationships between objects, however. The GradeBook *has-a* Menu, the ClassList *has-a* Student (several, in fact!), a Student *has-a* name, average, grade, list_of_scores, etc. The programmer must decide whether to code these attributes as classes or data fields.

Use verbs in the specification to identify possible methods.

IDENTIFYING BEHAVIORS

Use the verbs in the specification to identify required operations in the program. The verbs are: maintain <list>, provide <menu>, allow <user>, create <list>, enter <scores>, correct <data>, display <record>, calculate <average>, calculate <grade>, print <list>, add <student>, delete <student>, transfer <student>, and save <data>.

You must make some design decisions about which class is responsible for which behavior. For example, will a ClassList display the record of a single Student, or will a Student display his or her own record? Who will enter scores—the GradeBook, a ClassList, or a Student? Is it desirable for a Student to enter scores of other Students? Probably not!

DECISIONS

Here are some preliminary decisions. The GradeBook will provideMenu. The menu selection will send execution to the relevant object.

The ClassList will maintain an updated list of each class. It will have these public methods: addStudent, deleteStudent, transferStudent, createNewClass, printClassList, printScores, and updateList. A good candidate for a helper method in this class is search for a given student.

Each Student will have complete personal and grade information. Public methods will include setName, getName, enterScore, correctData, findAverage, getAverage, getGrade, and displayRecord.

Saving and retrieving information is crucial to this program. The FileHandler will take care of openFileForReading, openFileForWriting, closeFiles, loadClass, and saveClass. The FileHandler class should be written and tested right at the beginning, using a small dummy class list.

Score, ScoreList, and Student are easy classes to implement. When these are working, the programmer can go on to ClassList. Finally the Display GUI class, which will have the GradeBook, can be developed. This is an example of bottom-up development.

Example 3

A program simulates a game of Battleships, which is a game between two players, each of whom has a grid where ships are placed. Each player has five ships:

```
battleship o o o o o
cruiser    o o o o
submarine  o o o
destroyer  o o
frigate    o
```

The grids of the players' fleets may look like this. Any two adjacent squares that are taken must belong to the same ship, i.e., different ships shouldn't "touch."

	0	1	2	3	4	5	6	7
0								o
1	o	o	o	o	o			
2								
3								o
4					o		o	
5	o				o			
6		o			o			
7			o		o			

Player 1

	0	1	2	3	4	5	6	7
0	o			o	o			
1		o						o
2			o					o
3				o				o
4	o				o			
5								
6	o	o	o	o				
7								

Player 2

Each player's grid is hidden from the other player. Players alternate "shooting" at each other's ships by calling out a position, a row and column number. A player must make an honest response, "hit" or "miss." If it's a hit, a player gets another turn. If the whole ship has been hit, the owner must say something like, "You sank my cruiser." Each player must keep track of hits and misses. The first player to sink his opponent's fleet is the winner.

IDENTIFYING CLASSES

The nouns in the specification are program, game, players, grid, ship, battleship, cruiser, submarine, destroyer, frigate, square, position, opponent, row, column, turn, hits, misses, fleet, winner.

Eliminate each of the following:

program	Always eliminate.
row, col	These are parts of a given position or square, more suitable as instance variables for a position or square object.
hits, misses	These are simply marked positions and probably don't need their own class.
turn	Taking a turn is an action and will be described by a method rather than a class.
opponent	This is another word for player.

The following seem to be good candidates for classes: `Player`, `Grid`, `Position`, `Ship`, `Battleship`, `Cruiser`, `Submarine`, `Destroyer`, and `Frigate`. Additionally, it seems there should be a `GameManager` and `Display`.

RELATIONSHIP BETWEEN CLASSES

This program provides two examples of inheritance relationships. Each of the five ships *is-a* `Ship`, and shares common features, like `isHit`, `isSunk`, and array of positions. However, each has a unique name, length, and position in the grid. This means that `Ship` is a good candidate for an abstract class with abstract methods like `getLength`, `getName`, and `getPositions`, which depend on the kind of ship.

The second inheritance relationship is between the grids. There are two types of grids for each player: his own `FleetGrid` (the current state of his own ships) and his opponent's

HitGrid, which keeps track of his hits and misses. Each of these grids *is-a* Grid. A grid is a candidate for an interface, with a list of methods like getAdjacentNeighbors, getRightNeighbor, etc. Each of FleetGrid and HitGrid would implement Grid.

There are several composition relationships in this program. A Player *has-a* HitGrid and a FleetGrid and also has five ships. The GameManager has each of the two Player objects and also *has-a* Display. The Display has each of the grids.

IDENTIFYING BEHAVIORS

Use the verbs to identify key methods in the program: simulate <game>, place <ships>, shoot <at position>, call out <position>, respond <hit or miss>, sink <ship>, inform that <ship was sunk>, keep track of <hits or misses>, sink <opponent's fleet>, win <game>.

You need to decide who will do what. There's no definitive way of implementing the program, but it seems clear that the GameManager should run the game and declare the winner. Should the GameManager also be in charge of announcing if a ship is sunk? It makes sense because the game manager can see both players' grids. Each player should keep track of his calls, so that he can make an intelligent next call and also respond "hit" or "miss." Will each player have a display? Or will the Display have both players? You have to set it up so that a player can't see his opponent's FleetGrid, but he can see his own and also a grid showing the state of the calls he has made. Should each player have a list of his ships, so he can keep track of the state of his fleet? And what about each ship in the fleet? Should a ship have a list of its positions, and should it keep track of if it's hit or sunk?

Saving and retrieving updated information is crucial to this program. It seems a bit overwhelming. Where should you start? The Ship classes are low-level classes, independent of the players and grids. Start with these and test that you can get accurate information about each ship. In your driver program create an ArrayList<Ship>. Have a loop that prints information about each ship. Polymorphism will take care of getting the correct information about each ship.

Now try the Grid classes. This is a complicated program where each small piece should be coded and tested with simple output. For example, a Grid can be displayed with a two-dimensional array of 0's and 1's to show the positions of ships. Other symbols can be used to show what's been hit and what's been sunk.

When everything is working with the grids, you could add a Display class that has Grid variables and a display method.

Try a Player. Give him a list of ships, two grids and a Display.

Then create a GameManager. Give her two Player variables and be sure she has a playGame method.

The program development shown above is an example of bottom-up development.

Vocabulary Summary

Know these terms for the AP exam:

Vocabulary	Meaning
software development	Writing a program
object-oriented program	Uses interacting objects
program specification	Description of a task
program design	A written plan, an overview of the solution
program implementation	The code
test data	Input to test the program
program maintenance	Keeping the program working and up to date
top-down development	Implement main classes first, subsidiary classes later
independent class	Doesn't use other classes of the program in its code
bottom-up development	Implement lowest level, independent classes first
driver class	Used to test other classes; contains `main` method
inheritance relationship	*is-a* relationship between classes
composition relationship	*has-a* relationship between classes
inheritance hierarchy	Inheritance relationship shown in a tree-like diagram
UML diagram	Tree-like representation of relationship between classes
data structure	Java construct for storing a data field (e.g., array)
encapsulation	Combining data fields and methods in a class
information hiding	Using `private` to restrict access
stepwise refinement	Breaking methods into smaller methods
procedural abstraction	Using helper methods
algorithm	Step-by-step process that solves a problem
stub method	Dummy method called by another method being tested
debugging	Fixing errors
robust program	Screens out bad input
compile-time error	Usually a syntax error; prevents program from compiling
syntax error	Bad language usage (e.g., missing brace)
run-time error	Occurs during execution (e.g., `int` division by 0)
exception	Run-time error thrown by Java method
logic error	Program runs but does the wrong thing

PROGRAM ANALYSIS

Program Correctness

Testing that a program works does not prove that the program is correct. After all, you can hardly expect to test programs for every conceivable set of input data. Computer scientists have developed mathematical techniques to prove correctness in certain cases, but these are beyond the scope of the APCS course. Nevertheless, you are expected to be able to make assertions about the state of a program at various points during its execution.

Assertions

An *assertion* is a precise statement about a program at any given point. The idea is that if an assertion is proved to be true, then the program is working correctly at that point.

An informal step on the way to writing correct algorithms is to be able to make different kinds of assertions about your code.

PRECONDITION

The *precondition* for any piece of code, whether it is a method, loop, or block, is a statement of what is true immediately before execution of that code.

POSTCONDITION

The *postcondition* for a piece of code is a statement of what is true immediately after execution of that code.

Efficiency

An efficient algorithm is one that is economical in the use of

- CPU time. This refers to the number of machine operations required to carry out the algorithm (arithmetic operations, comparisons, data movements, etc.).
- Memory. This refers to the number and complexity of the variables used.

Some factors that affect run-time efficiency include unnecessary tests, excessive movement of data elements, and redundant computations, especially in loops.

Always aim for early detection of output conditions: Your sorting algorithm should halt when the list is sorted; your search should stop if the key element has been found.

In discussing efficiency of an algorithm, we refer to the *best case, worst case,* and *average case.* The best case is a configuration of the data that causes the algorithm to run in the least possible amount of time. The worst case is a configuration that leads to the greatest possible run time. Typical configurations (i.e., not specially chosen data) give the average case. It is possible that best, worst, and average cases don't differ much in their run times.

For example, suppose that a list of distinct random numbers must be searched for a given key value. The algorithm used is a sequential search starting at the beginning of the list. In the best case, the key will be found in the first position examined. In the worst case, it will be in the last position or not in the list at all. On average, the key will be somewhere in the middle of the list.

Chapter Summary

There's a lot of vocabulary that you are expected to know in this chapter. Learn the words!

Never make assumptions about a program specification, and always write a design before starting to write code. Even if you don't do this for your own programs, these are the answers you will be expected to give on the AP exam. You are certain to get questions about program design. Know the procedures and terminology involved in developing an object-oriented program.

Be sure you understand what is meant by best case, worst case, and average case for an algorithm. There will be questions about efficiency on the AP exam.

By now you should know what a precondition and postcondition are.

MULTIPLE-CHOICE QUESTIONS ON PROGRAM DESIGN AND ANALYSIS

1. A program that reads in a five-digit identification number is to be written. The specification does not state whether zero can be entered as a first digit. The programmer should
   (A) write the code to accept zero as a first digit since zero is a valid digit.
   (B) write the code to reject zero as a first digit since five-digit integers do not start with zero.
   (C) eliminate zero as a possibility for any of the digits.
   (D) treat the identification number as a four-digit number if the user enters a number starting with zero.
   (E) check with the writer of the specification whether zero is acceptable as a first digit.

2. Refer to the following three program descriptions:

   I Test whether there exists at least one three-digit integer whose value equals the sum of the squares of its digits.

   II Read in a three-digit code number and check if it is valid according to some given formula.

   III Passwords consist of three digits and three capital letters in any order. Read in a password, and check if there are any repeated characters.

   For which of the preceding program descriptions would a `ThreeDigitNumber` class be suitable?
   (A) I only
   (B) II only
   (C) III only
   (D) I and II only
   (E) I, II, and III

3. Top-down programming is illustrated by which of the following?
   (A) Writing a program from top to bottom in Java
   (B) Writing an essay describing how the program will work, without including any Java code
   (C) Using driver programs to test all methods in the order that they're called in the program
   (D) Writing and testing the lowest level methods first and then combining them to form appropriate abstract operations
   (E) Writing the program in terms of the operations to be performed and then refining these operations by adding more detail

4. Which of the following should influence your choice of a particular algorithm?

    I The run time of the algorithm
    II The memory requirements of the algorithm
    III The ease with which the logic of the algorithm can be understood

    (A) I only
    (B) III only
    (C) I and III only
    (D) I and II only
    (E) I, II, and III

5. A list of numbers is stored in a sorted array. It is required that the list be maintained in sorted order. This requirement leads to inefficient execution for which of the following processes?

    I Summing the five smallest numbers in the list
    II Finding the maximum value in the list
    III Inserting and deleting numbers

    (A) I only
    (B) III only
    (C) II and III only
    (D) I and III only
    (E) I, II, and III

6. Which of the following is *not* necessarily a feature of a robust program?

    (A) Does not allow execution to proceed with invalid data
    (B) Uses algorithms that give correct answers for extreme data values
    (C) Will run on any computer without modification
    (D) Will not allow division by zero
    (E) Will anticipate the types of errors that users of the program may make

7. A certain freight company charges its customers for shipping overseas according to this scale:

    \$80 per ton for a weight of 10 tons or less
    \$40 per ton for each additional ton over 10 tons but
        not exceeding 25 tons
    \$30 per ton for each additional ton over 25 tons

    For example, to ship a weight of 12 tons will cost 10(80) + 2(40) = \$880. To ship 26 tons will cost 10(80) + 15(40) + 1(30) = \$1430.

    A method takes as parameter an integer that represents a valid shipping weight and outputs the charge for the shipment. Which of the following is the smallest set of input values for shipping weights that will adequately test this method?

    (A) 10, 25
    (B) 5, 15, 30
    (C) 5, 10, 15, 25, 30
    (D) 0, 5, 10, 15, 25, 30
    (E) 5, 10, 15, 20, 25, 30

8. A code segment calculates the mean of values stored in integers `n1`, `n2`, `n3`, and `n4` and stores the result in `average`, which is of type `double`. What kind of error is caused with this statement?

```
double average = n1 + n2 + n3 + n4 / (double) 4;
```

(A) Logic
(B) Run-time
(C) Overflow
(D) Syntax
(E) Type mismatch

9. A program evaluates binary arithmetic expressions that are read from an input file. All of the operands are integers, and the only operators are +, -, *, and /. In writing the program, the programmer forgot to include a test that checks whether the right-hand operand in a division expression equals zero. When will this oversight be detected by the computer?
(A) At compile time
(B) While editing the program
(C) As soon as the data from the input file is read
(D) During evaluation of the expressions
(E) When at least one incorrect value for the expressions is output

10. Which best describes the precondition of a method? It is an assertion that
(A) describes precisely the conditions that must be true at the time the method is called.
(B) initializes the parameters of the method.
(C) describes the effect of the method on its postcondition.
(D) explains what the method does.
(E) states what the initial values of the local variables in the method must be.

11. Consider the following code fragment:

```
/** Precondition: a1, a2, a3 contain 3 distinct integers.
 *  Postcondition: max contains the largest of a1, a2, a3.
 */
//first set max equal to larger of a1 and a2
if (a1 > a2)
    max = a1;
else
    max = a2;
//set max equal to larger of max and a3
if (max < a3)
    max = a3;
```

For this algorithm, which of the following initial setups for a1, a2, and a3 will cause
(1) the least number of computer operations (best case) and
(2) the greatest number of computer operations (worst case)?

(A) (1) largest value in a1 or a2 (2) largest value in a3
(B) (1) largest value in a2 or a3 (2) largest value in a1
(C) (1) smallest value in a1 (2) largest value in a2
(D) (1) largest value in a2 (2) smallest value in a3
(E) (1) smallest value in a1 or a2 (2) largest value in a3

12. Refer to the following code segment.

```
/** Compute the mean of integers 1 .. N.
 *  N is an integer >= 1 and has been initialized.
 */
int k = 1;
double mean, sum = 1.0;
while (k < N)
{
    /* loop body */
}
mean = sum / N;
```

What is the precondition for the while loop?

(A) `k ≥ N,  sum = 1.0`
(B) `sum = 1 + 2 + 3 + ...  + k`
(C) `k < N,  sum = 1.0`
(D) `N ≥ 1,  k = 1,  sum = 1.0`
(E) `mean = sum / N`

13. The sequence of Fibonacci numbers is 1, 1, 2, 3, 5, 8, 13, 21, ... The first two Fibonacci numbers are each 1. Each subsequent number is obtained by adding the previous two. Consider this method:

```
/** Precondition: n >= 1.
 *  Postcondition: The nth Fibonacci number has been returned.
 */
public static int fib(int n)
{
    int prev = 1, next = 1, sum = 1;
    for (int i = 3; i <= n; i++)
    {
        /* assertion */
        sum = next + prev;
        prev = next;
        next = sum;
    }
    return sum;
}
```

Which of the following is a correct `/* assertion */` about the loop variable `i`?

(A) $1 \le i \le n$

(B) $0 \le i \le n$

(C) $3 \le i \le n$

(D) $3 < i \le n$

(E) $3 < i < n+1$

14. Refer to the following method.

```
/** Precondition: a and b are initialized integers.
 */
public static int mystery(int a, int b)
{
    int total = 0, count = 1;
    while (count <= b)
    {
        total += a;
        count++;
    }
    return total;
}
```

What is the postcondition for method `mystery`?

(A) total $= a + b$

(B) total $= a^b$

(C) total $= b^a$

(D) total $= a * b$

(E) total $= a/b$

15. A program is to be written that prints an invoice for a small store. A copy of the invoice will be given to the customer and will display

- A list of items purchased.
- The quantity, unit price, and total price for each item.
- The amount due.

Three candidate classes for this program are `Invoice`, `Item`, and `ItemList`, where an `Item` is a single item purchased and `ItemList` is the list of all items purchased. Which class is a reasonable choice to be responsible for the `amountDue` method, which returns the amount the customer must pay?

I `Item`

II `ItemList`

III `Invoice`

(A) I only
(B) III only
(C) I and II only
(D) II and III only
(E) I, II, and III

16. Which is a *false* statement about classes in object-oriented program design?

(A) If a class `C1` has an instance variable whose type is another class, `C2`, then `C1` *has-a* `C2`.
(B) If a class `C1` is associated with another class, `C2`, then `C1` depends on `C2` for its implementation.
(C) If classes `C1` and `C2` are related such that `C1` *is-a* `C2`, then `C2` *has-a* `C1`.
(D) If class `C1` is independent, then none of its methods will have parameters that are objects of other classes.
(E) Classes that have common methods do not necessarily define an inheritance relationship.

17. A Java program maintains a large database of vehicles and parts for a car dealership. Some of the classes in the program are `Vehicle`, `Car`, `Truck`, `Tire`, `Circle`, `SteeringWheel`, and `AirBag`. The declarations below show the relationships between classes. Which is a poor choice?

```
(A) public class Vehicle
    {   ...
        private Tire[] tires;
        private SteeringWheel sw;
        ...
    }
(B) public class Tire extends Circle
    {   ...
        //inherits methods that compute circumference
        //and center point
    }
(C) public class Car extends Vehicle
    {   ...
        //inherits private Tire[] tires from Vehicle class
        //inherits private SteeringWheel sw from Vehicle class
        ...
    }
(D) public class Tire
    {   ...
        private String rating;      //speed rating of tire
        private Circle boundary;
    }
(E) public class SteeringWheel
    {   ...
        private AirBag ab;  //AirBag is stored in SteeringWheel
        private Circle boundary;
    }
```

18. A Java programmer has completed a preliminary design for a large program. The programmer has developed a list of classes, determined the methods for each class, established the relationships between classes, and written an outline for each class. Which class(es) should be implemented first?

(A) Any superclasses
(B) Any subclasses
(C) All collaborator classes (classes that will be used to implement other classes)
(D) The class that represents the dominant object in the program
(E) All independent classes (classes that have no references to other classes)

Use the program description below for Questions 19–21.

A program is to be written that simulates bumper cars in a video game. The cars move on a square grid and are located on grid points (x, y), where x and y are integers between -20 and 20. A bumper car moves in a random direction, either left, right, up, or down. If it reaches a boundary (i.e., x or y is ± 20), then it reverses direction. If it is about to collide with another bumper car, it reverses direction. Your program should be able to add bumper cars and run the simulation. One step of the simulation allows each car in the grid to move. After a bumper car has reversed direction twice, its turn is over and the next car gets to move.

19. To identify classes in the program, the nouns in the specification are listed:

    program, bumper car, grid, grid point, integer, direction, boundary, simulation

    How many nouns in the list should immediately be discarded because they are unsuitable as classes for the program?

    (A) 0
    (B) 1
    (C) 2
    (D) 3
    (E) 4

A programmer decides to include the following classes in the program. Refer to them for Questions 20 and 21.

- `Simulation` will run the simulation.
- `Display` will show the state of the game.
- `BumperCar` will know its identification number, position in the grid, and current direction when moving.
- `GridPoint` will be a position in the grid. It will be represented by two integer fields, `x_coord` and `y_coord`.
- `Grid` will keep track of all bumper cars in the game, the number of cars, and their positions in the grid. It will update the grid each time a car moves. It will be implemented with a two-dimensional array of `BumperCar`.

20. Which operation should not be the responsibility of the `GridPoint` class?

(A)	`isEmpty`	returns false if grid point contains a `BumperCar`, true otherwise
(B)	`atBoundary`	returns true if *x* or *y* coordinate = ±20, false otherwise
(C)	`left`	if not at left boundary, change grid point to 1 unit left of current point
(D)	`up`	if not at top of grid, change grid point to 1 unit above current point
(E)	`get_x`	return *x*-coordinate of this point

21. Which method is not suitable for the `BumperCar` class?

```
(A) public boolean atBoundary()
    //Returns true if BumperCar at boundary, false otherwise.

(B) public void selectRandomDirection()
    //Select random direction (up, down, left, or right)
    // at start of turn.

(C) public void reverseDirection()
    //Move to grid position that is in direction opposite to
    // current direction.

(D) public void move()
    //Take turn to move. Stop move after two changes
    // of direction.

(E) public void update()
    //Modify Grid to reflect new position after each stage
    // of move.
```

ANSWER KEY

1. **E**	8. **A**	15. **D**
2. **D**	9. **D**	16. **C**
3. **E**	10. **A**	17. **B**
4. **E**	11. **A**	18. **E**
5. **B**	12. **D**	19. **C**
6. **C**	13. **C**	20. **A**
7. **C**	14. **D**	21. **E**

ANSWERS EXPLAINED

1. **(E)** A programmer should never make unilateral decisions about a program specification. When in doubt, check with the person who wrote the specification.

2. **(D)** In I and II a three-digit number is the object being manipulated. For III, however, the object is a six-character string, suggesting a class other than a `ThreeDigitNumber`.

3. **(E)** Top-down programming consists of listing the methods for the main object and then using stepwise refinement to break each method into a list of subtasks. Eliminate choices A, C, and D: Top-down programming refers to the design and planning stage and does not involve any actual writing of code. Choice B is closer to the mark, but "top-down" implies a list of operations, not an essay describing the methods.

4. **(E)** All three considerations are valid when choosing an algorithm. III is especially important if your code will be part of a larger project created by several programmers. Yet even if you are the sole writer of a piece of software, be aware that your code may one day need to be modified by others.

5. **(B)** A process that causes excessive data movement is inefficient. Inserting an element into its correct (sorted) position involves moving elements to create a slot for this element. In the worst case, the new element must be inserted into the first slot, which involves moving every element up one slot. Similarly, deleting an element involves moving elements down a slot to close the "gap." In the worst case, where the first element is deleted, all elements in the array will need to be moved. Summing the five smallest elements in the list means summing the first five elements. This requires no testing of elements and no excessive data movement, so it is efficient. Finding the maximum value in a sorted list is very fast—just select the element at the appropriate end of the list.

6. **(C)** "Robustness" implies the ability to handle all data input by the user and to give correct answers even for extreme values of data. A program that is not robust may well run on another computer without modification, and a robust program may need modification before it can run on another computer.

7. **(C)** Eliminate choice D because 0 is an invalid weight, and you may infer from the method description that invalid data have already been screened out. Eliminate choice E because it tests two values in the range 10–25. (This is not wrong, but choice C is better.) Eliminate choice A since it tests only the endpoint values. Eliminate B because it tests *no* endpoint values.

8. **(A)** The statement is syntactically correct, but as written it will not find the mean of the integers. The bug is therefore an intent or logic error. To execute as intended, the statement needs parentheses:

```
double average = (n1 + n2 + n3 + n4) / (double) 4;
```

9. **(D)** The error that occurs is a run-time error caused by an attempt to divide by zero (`ArithmeticException`). Don't be fooled by choice C. Simply reading an expression `8/0` from the input file won't cause the error. Note that if the operands were of type `double`, the correct answer would be E. In this case, dividing by zero does not cause an exception; it gives an answer of `Infinity`. Only on inspecting the output would it be clear that something was wrong.

10. **(A)** A precondition does not concern itself with the action of the method, the local variables, the algorithm, or the postcondition. Nor does it initialize the parameters. It simply asserts what must be true directly before execution of the method.

11. **(A)** The best case causes the fewest computer operations, and the worst case leads to the maximum number of operations. In the given algorithm, the initial test `if (a1 > a2)` and the assignment to `max` will occur irrespective of which value is the largest. The second test, `if (max < a3)`, will also always occur. The final statement, `max = a3`, will occur only if the largest value is in `a3`; thus, this represents the worst case. So the best case must have the biggest value in `a1` or `a2`.

12. **(D)** The precondition is an assertion about the variables in the loop just before the loop is executed. Variables `N`, `k`, and `sum` have all been initialized to the values shown in choice D. Choice C is wrong because `k` may equal `N`. Choice A is wrong because `k` may be less than `N`. Choice E is wrong because `mean` is not defined until the loop has been exited. Choice B is wrong because it omits the assertions about `N` and `k`.

13. **(C)** Eliminate choices A and B, since `i` is initialized to `3` in the `for` loop. Choices D and E are wrong because `i` is equal to `3` the first time through the loop.

14. **(D)** The quantity `a` is being added to `total` `b` times, which means that at the end of execution `total = a*b`.

15. **(D)** It makes sense for an `Item` to be responsible for its name, unit price, quantity, and total price. It is *not* reasonable for it to be responsible for other `Items`. Since an `ItemList`, however, will contain information for all the `Items` purchased, it is reasonable to have it also compute the total `amountDue`. It makes just as much sense to give an `Invoice` the responsibility for displaying information for the items purchased, as well as providing a final total, `amountDue`.

16. **(C)** The *is-a* relationship defines inheritance, while the *has-a* relationship defines association. These types of relationship are mutually exclusive. For example, a graduate student *is-a* student. It doesn't make sense to say a student *has-a* graduate student!

17. **(B)** Even though it's convenient for a `Tire` object to inherit `Circle` methods, an inheritance relationship between a `Tire` and a `Circle` is incorrect: It is false to say that a `Tire` *is-a* `Circle`. A `Tire` is a car part, while a `Circle` is a geometric shape. Notice that there is an *association* relationship between a `Tire` and a `Circle`: A `Tire` *has-a* `Circle` as its boundary.

18. **(E)** Independent classes do not have relationships with other classes and can therefore be more easily coded and tested.

19. **(C)** The word "program" is never included when it's used in this context. The word "integer" describes the type of coordinates *x* and *y* and has no further use in the specification. While words like "direction," "boundary," and "simulation" may later be removed from consideration as classes, it is not unreasonable to keep them as candidates while you ponder the design.

20. **(A)** A `GridPoint` object knows only its *x* and *y* coordinates. It has no information about whether a `BumperCar` is at that point. Notice that operations in all of the other choices depend on the *x* and *y* coordinates of a `GridPoint` object. An `isEmpty` method should be the responsibility of the `Grid` class that keeps track of the status of each position in the grid.

21. **(E)** A `BumperCar` is responsible for itself—keeping track of its own position, selecting an initial direction, making a move, and reversing direction. It is not, however, responsible for maintaining and updating the grid. That should be done by the `Grid` class.