
PROGRAM DESIGN

INTRODUCTION

Although there are many design techniques that we could discuss we shall keep things simple and talk about only two generic techniques based on two major ideas in programming: procedures, and objects. Until object-oriented methods became popular, the best way to attack the complex problem of creating a computer program was to use a form of problem analysis based on divide-and-conquer. This is the method called Top-Down Stepwise Refinement (TDSR). In this method, the problem is broken down into a time-ordered sequence of sub-tasks, which can themselves be broken down further if necessary. This approach is definitely procedure-oriented, i.e. it concentrates on what has to be done next in order to achieve the goal.

The object-oriented approach follows a different emphasis. Here the problem is analyzed in terms of what objects are involved, what kinds of interactions they have, and only as a last step, determining whether these interactions are procedural or not. The figure on page 27 shows, pictorially what these solutions look like for a simple problem, that of estimating the cost of painting a room.

TOP-DOWN STEPWISE REFINEMENT

The method has four stages:

- 1) Problem statement
- 2) Problem description
- 3) TDSR diagram
- 4) Source code

We will use a simple example to motivate the use of these four stages in designing a program. The example concerns the estimation of the cost of painting a room.

PROBLEM STATEMENT

Every computer application has a goal that can be summarized in a single sentence, or perhaps a small number of sentences. This we will call a problem statement. For our example, the problem statement is:

Estimate the cost of painting a room

PROBLEM DESCRIPTION

A successful computer application stems from a successful problem analysis. You must first understand the scope of the problem in order to design a program to carry out the task in the problem statement. A problem description should address at least the following:

- What input does the program have to accept? Where does the input come from? What are the limitations (if any) on the format of the input?
- Does the problem have natural parts, or stages?
- Are there any parts of the problem that can be ignored or simplified by making a reasonable assumption?
- Does the problem need access to data files or data bases?
- Does the problem produce output data that needs to be preserved between program runs?
- Does the problem process lists, sequences or other compound data?
- Does the problem involve repetitions of tasks or sub-tasks?
- Does the problem involve choices made by the user?
- What output does the program have to produce? Where should this output appear?
- Are library routines, such as math functions or graphics procedures, needed?

For our example, the problem description is:

This program estimates the cost of painting the four walls and the ceiling of a room. Any doors or windows can be ignored. The height of the room can be assumed as a constant (8 ft.) but the program should input the length and width, in feet. The same paint is used to cover walls and ceiling. The cost of the paint is constant (\$12.50 per gallon). The coverage of the paint is to be input, in sq. ft. per gallon. The program should display the final cost of painting.

Notice that this description says nothing about the language to be used for the programming, and should be a problem analysis, not a program analysis.

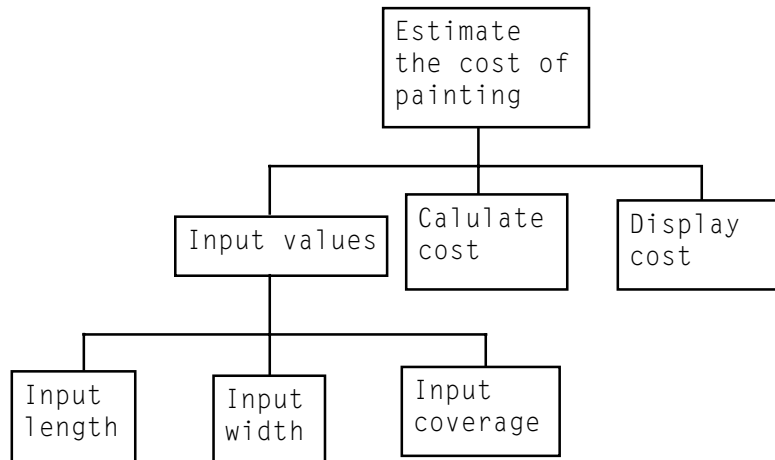
TOP-DOWN STEPWISE REFINEMENT

This methodology uses an age-old technique of divide and conquer. The aim is to analyze the problem statement, using the problem description in terms of a sequence of sub-tasks. If the problem is viewed as an input-output processor, then the first natural breakdown is into three stages: input, calculation and output. Many programs (but not all) fit this mold.

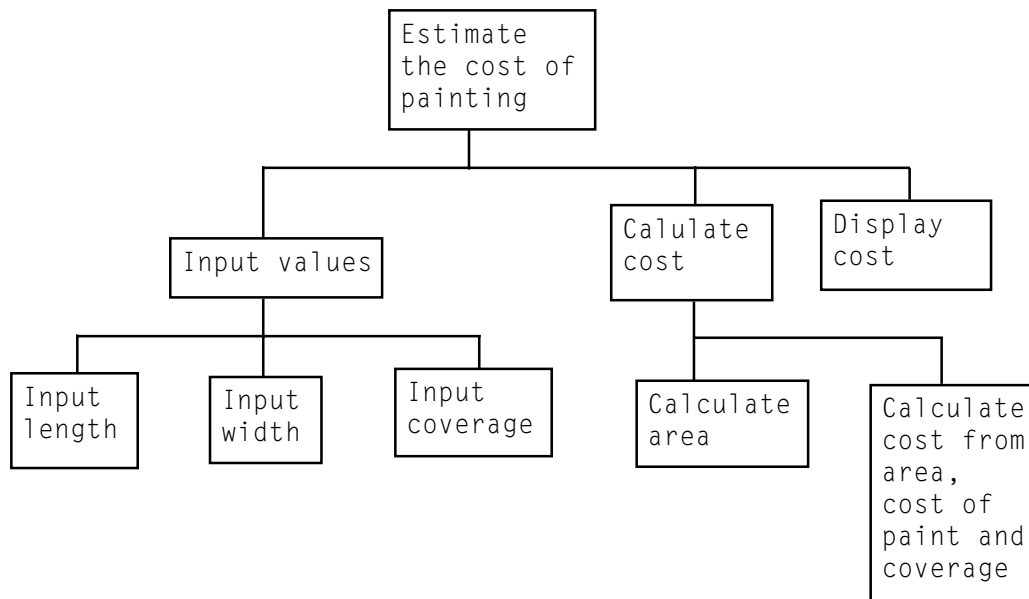
The TDSR diagram is a hierarchical chart where each block represents a program step. The successive levels in the diagram show how each step can be broken down further. Each level is a refinement of the problem analysis. In our example, the first level is simply the problem statement:

Estimate the cost of painting

The next level is a refinement in terms of the three stages: input, calculation, output:



At this stage, it is clear that some further analysis of the problem is necessary. It may be a good idea to refine the problem description as you go, but the distinction between problem analysis and program analysis should always be maintained. The values to be input are length and width of the room (height will be constant) and the paint coverage. The next refinement is thus:



The display task does not need further refinement.

The aim of the TDSR diagram should be to reach a sufficiently low level to make coding (writing programming language statements) as easy as possible. Mostly, each of the lowest level boxes end up as one statement, or perhaps a small number.

Program coding

The first decision to make is program structure in terms of the number and size of the subprograms to use. Too few subprograms and subprograms that are too long make a program that is difficult to read. On the other hand, too many subprograms and subprograms that are too small also tends to make a program that is over fussy and confusing. There is no general solution or method that will guarantee a good program. All we can say is that a one line subprogram is probably too small, and a 100 line subprogram is too large. However, we can use the TDSR diagram to guide our choices of the number and size of subprograms. A good rule of thumb is that there should be at *least* as many subprograms as the second-level boxes. In our example, this is three. We could also add two subprograms for the sub-tasks in the calculation. The first attempt thus gives this structure (NOTE that this is not a complete program, it just shows the intended program structure):

```

SUBPROGRAM inputValues()
    ...

SUBPROGRAM calculateArea()
    ...

SUBPROGRAM calculateCostFromArea()
    ...

SUBPROGRAM calculateCost()
    calculateArea();
    calculateCostFromArea();

SUBPROGRAM displayCost()
    ...

MAINPROGRAM()
    inputValues();
    calculateCost();
    displayCost();

```

It is now time to choose representations for the values to be used in the program. In our example, all measurements and costs can be floating point numbers; the question is where are they to be declared? One solution is to make everything global, but this is not considered to be a good solution:

See Source code Example #5

The reason this is not a good solution is that global variables can be modified by any function, not just the ones that the problem analysis intends. This problem is called indiscriminate access. The solution is to make variables local to the function that needs to access the variables, and to pass values through parameters where other functions need those values. The reworked solution is then:

See Source code Example #6

Notice we were unable to get rid of all the global variables because inputValues needs to set three values, so the function return mechanism cannot be used. There is a solution to this problem as well, but since it involves the use of pointers, we shall not show it here. It would be possible to declare the three variables as local in function main, but there would then need to be three input functions, one for each variable:

See Source code Example #7

Notice also that the constant values have been left as global definitions. Since no function can alter their values, it is not necessary to make them local to any particular function.

OBJECT-ORIENTED DESIGN

An simple, yet effective way to get started in object-oriented design follows the same kind of initial thinking as TDSR, but branches off in the ways in which the analysis proceeds. It has five steps:

- 1) problem statement
- 2) problem description
- 3) object analysis
- 4) object relationships
- 5) object interaction

The first two steps are identical to TDSR. Thus for our painting example, we have:

PROBLEM STATEMENT

For our example, the problem statement is:

Estimate the cost of painting a room

PROBLEM DESCRIPTION

For our example, the problem description is:

This program estimates the cost of painting the four walls and the ceiling of a room. Any doors or windows can be ignored. The height of the room can be assumed as a constant (8 ft.) but the program should input the length and width, in feet. The same paint is used to cover walls and ceiling. The cost of the paint is constant (\$12.50 per gallon). The coverage of the paint is to be input, in sq. ft. per gallon. The program should display the final cost of painting.

Now, however, the methods diverge. The next step is to find all the nouns in the description—they become objects, and all the verbs—they become procedural interactions among the objects. Again, as with TDSR, this is a problem analysis, and says nothing about the language for writing the program. If we do this we get:

OBJECT ANALYSIS

We show the nouns first:

This program estimates the cost of painting the four walls and the ceiling of a room. Any doors or windows can be ignored. The height of the room can be assumed as a constant (8 ft.) but the program should input the length and width, in feet. The same paint is used to cover walls and ceiling. The cost of the paint is constant (\$12.50 per gallon). The coverage of the paint is to be input, in sq. ft. per gallon. The program should display the final cost of painting.

Notice that there is no need to distinguish irrelevant objects: the program, doors and windows, constant, and the measurement units, sq. ft., gallon, \$.

Now the verbs:

This program estimates the cost of painting the four walls and the ceiling of a room. Any doors or windows can be ignored. The height of the room can be assumed as a constant (8 ft.) but the program should input the length and width, in feet. The same paint is used to cover walls and ceiling. The cost of the paint is constant (\$12.50 per gallon). The coverage of the paint is to be input, in sq. ft. per gallon. The program should display the final cost of painting.

Again, notice that irrelevant verbs, such as “can be assumed” can be ignored, and can be omitted from the analysis.

OBJECT RELATIONSHIPS

Each noun then becomes an object, which is an instance of some class of such objects. The object-oriented languages all have a mechanism for defining the attributes of an object without specifying what values these may take on for any one object. The class mechanism, whether in C++, Java, Object Pascal or Smalltalk is a way of defining these attributes. Since most object-based languages only have two main ways of defining attributes, then we must look for these kinds of object interactions. So, objects can be related in two ways:

- 1) An object can contain another; we call this the HAS-A relation. In our example, a room HAS-A ceiling. HAS-A is also useful for properties like size, shape, color, cost etc. So, a ceiling HAS-A height, and the paint HAS-A cost.
- 2) An object can be similar to another, but different in some ways; we call this the IS-A relation. Our example is too simple to have an example of this, but if, for instance, we had two kinds of paint—acrylic and enamel, we might say that acrylic paint IS-A (kind of) paint, and enamel paint IS-A paint. This inheritance relationship gives object languages a great deal of their power. Notice, however, that those languages that have inheritance are useful because they follow the kind of problem analysis we are presenting here; they are not arbitrary features of these languages.

For our example, we get the following analysis:

```

Estimator HAS-A CostOfPainting
Room HAS-A Walls
Room HAS-A Ceiling
Room HAS-A Height
Room HAS-A Width
Room HAS-A Length
Estimator HAS-A Paint
Paint HAS-A CostOfPaint
Paint HAS-A Coverage

```

It is almost always necessary in object-oriented programs to make an object for the whole task, sometimes called the application. Here this is the Estimator object. With a little thinking we can eliminate the Walls and Ceiling objects since their properties (length, width, height) are those of the Room anyway. We also need to add the fact that the Estimator HAS-A Room, a link which is not explicit in the description (we should go back to add this, but here we will not). If we make these changes, and gather properties together under their common objects, we get:

```

Estimator
Room
Paint
CostOfPainting

Room
Width
Length
Height

Paint
CostOfPaint
Coverage

```

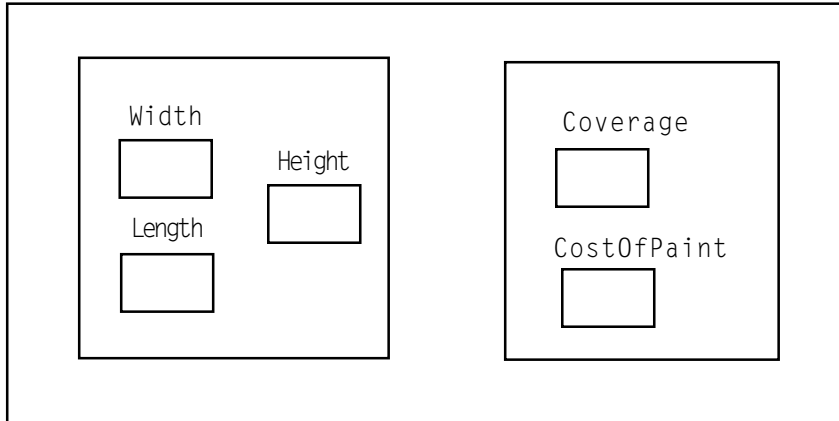
These are ready to turn into class definitions in whatever object language you care to use. The examples are easier to understand in a real object-oriented language, i.e. C++ and Java. However, the technique can still be used with languages that are not object-oriented. The basic requirements are only two:

- A A mechanism for encapsulation of data variables to make compound variables, and
- B A mechanism for passing a reference to such a variable to a subprogram.

See Source code Example #8

Note the use of the keyword `const` for the constants, as specified in the problem description, and the choice of base types (here they are all floating-point numbers) for measurements and money amounts. This diagram shows these relationships in terms of the objects created by the application.

Estimator



OBJECT INTERACTION

This is where methods are added to the classes already defined. A method is a procedure that an object carries out to achieve some task. The tasks are described by the verbs in the description. There are:

```

estimate
display (cost of painting)
input (length, width, coverage)
  
```

Each task has to be assigned to a class, and this assignment is not always obvious. The best plan is to determine the input-output requirements for each task and this should then allow you to place the method in the most appropriate class. Since the object-oriented approach says that all data should be private to an object, it is very often necessary to add accessor methods to a class so that other methods can acquire data locked up inside other objects. This kind of thinking leads to this analysis:

The estimate task needs length, width and height of the room, the cost of paint and its coverage. It outputs the cost of painting the room. This clearly should go in the Estimator as far as the output goes, but there will have to be accessor methods in Room and Paint in order to retrieve the measurements and costs.

The display task only needs the cost of painting, so it clearly should go in the Estimator class.

The input task has to take input from the user and produce output in two different objects: the room and the paint. Again this is best put in the Estimator, which has access to both contained objects, but there could also be two different methods with the same name in each of the two classes Room and Paint. We will give the second solution here to show how methods can be moved around to suit the coding and the application itself. Both solutions are good:

See Source code Example #9

A 'run' method has been added to the Estimator class as the entry point of the program. Now that the structure of the program matches the structure of the problem, it only remains to flesh out the skeleton provided by the class mechanism. All we need to do here is to define the two Estimator methods estimate and run. The hope is that they will be easy to write since the object analysis should

have been detailed enough that each method is easy to write. If it is not, it can be necessary to resort to a TDSR approach where a method can call other methods in the same class to do its job. The same principles described in the TDSR approach can be applied in this case. Luckily we will not need this in our simple example. A top level “main” function is also needed.

See Source code Example #10

SAMPLE QUESTIONS AND ANSWERS

These sample design questions are given without explanation. However, by following the detailed examples in this chapter you should be able to follow the solutions. Each problem is given as an informal description, followed by the analysis, either top-down stepwise refinement, or object-oriented, or both.

PROBLEM 1

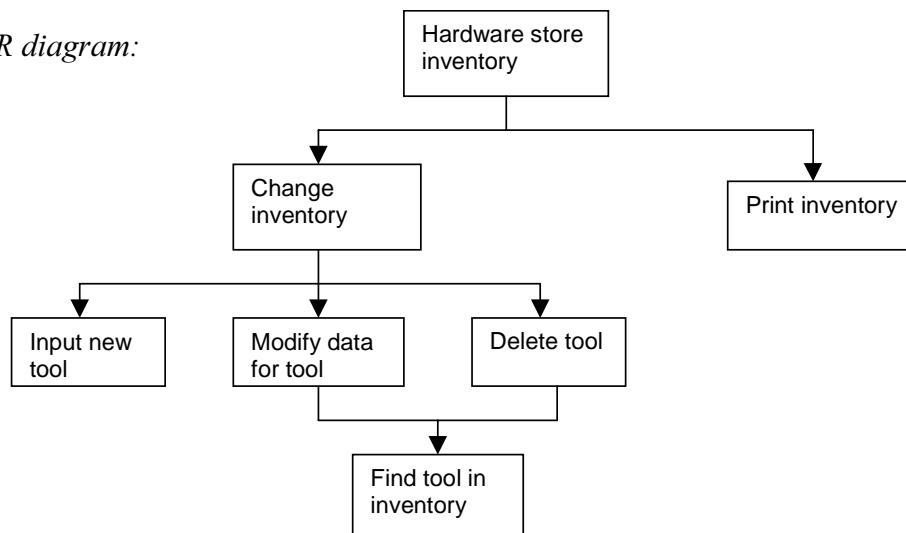
You are the owner of a hardware store, and you need to keep an inventory of what tools you have, how many of each kind, and what they cost. The program should handle up to 100 different kinds of tool, and allow you to input data for each kind of tool, delete information for a tool you no longer have, and update any information regarding any tool. Each kind of tool has an identification number and a name. e.g. Power saws are number 56, and Crescent wrenches are 83.

ANSWER TO PROBLEM 1: TOP-DOWN STEPWISE REFINEMENT

Problem statement: Hardware store inventory

Problem Description: The inventory program will maintain a database containing an inventory of tools. Each type of tool has a name, which is a string of up to 32 characters (with embedded spaces); an identification number, a positive integer with maximum value 100; the quantity on hand, which is a positive integer with maximum value 10000; and the unit cost which is a positive floating-point number with maximum value 10000.00 (dollars). The program will be able to input commands from the keyboard that will add a new tool, delete a tool which is no longer required, and be able to modify any single piece of data for any tool in the inventory. It will also report any and all changes to the inventory to the screen, and be able to print a complete listing of the total inventory on a printer.

TDSR diagram:



[Note the sub-task for finding a tool in the inventory is shared by the two tasks above it]

ANSWER TO PROBLEM 1: OBJECT-ORIENTED ANALYSIS

Problem statement: Hardware store inventory

Problem Description: The inventory program will maintain a database of an inventory tools. Each type of tool has a name, which is a string of up to 32 characters (with embedded spaces); an identification number, an positive integer with maximum value 100; the quantity on hand, which is a positive integer with maximum value 10000; and the unit cost which is a positive floating-point number with maximum value 10000.00 (dollars). The program will be able to input commands from the keyboard that will add a new tool, delete a tool which is no longer required, and be able to modify any single piece of data for any tool in the inventory. It will also report any and all changes to the inventory to the screen, and be able to print a complete listing of the total inventory on a printer.

Noun/object analysis: The inventory program will maintain a database containing an inventory of tools. Each type of tool has a name, which is a string of up to 32 characters (with embedded spaces); an identification number, a positive integer with maximum value 100; the quantity on hand, which is a positive integer with maximum value 10000; and the unit cost which is a positive floating-point number with maximum value 10000.00 (dollars). The program will be able to input commands from the keyboard that will add a new tool, delete a tool which is no longer required, and be able to modify any single piece of data for any tool in the inventory. It will also report any and all changes to the inventory to the screen, and be able to print a complete listing of the total inventory on a printer.

Objects: program HAS-A database
 database HAS-A (set of) tools
 tool HAS-A name
 tool HAS-A identification number
 tool HAS-A unit cost
 tool HAS-A quantity on hand

Verb/method analysis: The inventory program will maintain a database of an inventory tools. Each type of tool has a name, which is a string of up to 32 characters (with embedded spaces); an identification number, an positive integer with maximum value 100; the quantity on hand, which is a positive integer with maximum value 10000; and the unit cost which is a positive floating-point number with maximum value 10000.00 (dollars). The program will be able to input commands from the keyboard that will add a new tool, delete a tool which is no longer required, and be able to modify any single piece of data for any tool in the inventory. It will also report any and all changes to the inventory to the screen, and be able to print a complete listing of the total inventory on a printer.

Methods: maintain

input
add
delete
modify
report
print

Class structure:

```

Inventory
    Database database
Database
    Tool tool (set of)
Tool
    PosInteger idName
    String name
    PosInteger quantity
    Floating unitCost
  
```

Add methods:

```

Inventory
    Database database
    maintain()
    input()
Database
    Tool tools (set of)
    add()
    delete()
    modify()
    print()
Tool
    PosInteger idName
    String name
    PosInteger quantity
    Floating unitCost
    report()
  
```

[Note the choice of putting report in the Tool class, so that it can report any changes made to piece of data for the tool]

PROBLEM 2

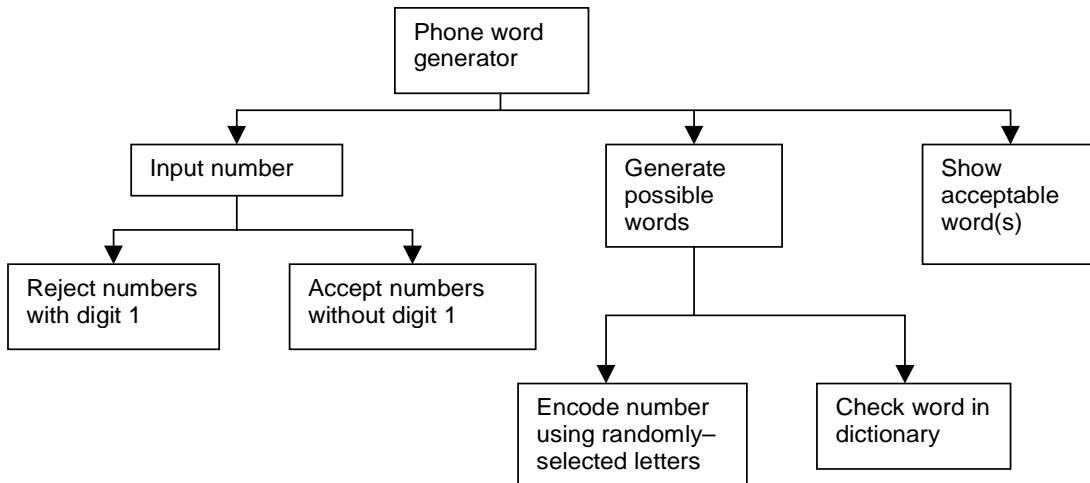
Businesses often use words corresponding to their 7-digit telephone number, according to the standard telephone key-pad. A program is needed that generates all possible words or sequences of words from a given telephone number. You can assume that an on-line dictionary is available to check the validity of words generated by the program.

ANSWER TO PROBLEM 2: TOP-DOWN STEPWISE REFINEMENT

Problem statement: Phone word generator

Problem description: The program will input a 7-digit telephone number from the keyboard as a character string of digits, and find a word or words that correspond to the number using the standard telephone key-pad encoding: 2/ABC, 3/DEF, 4/GHI, 5/JKL, 6/MNO, 7/PRS, 8/TUV, 9/WXY (Note 1 has no corresponding letters. Numbers with a 1 will be rejected). An on-line dictionary will be available to check whether generated words are acceptable. The acceptable words will be displayed on the screen.

TDSR diagram:



PROBLEM 3

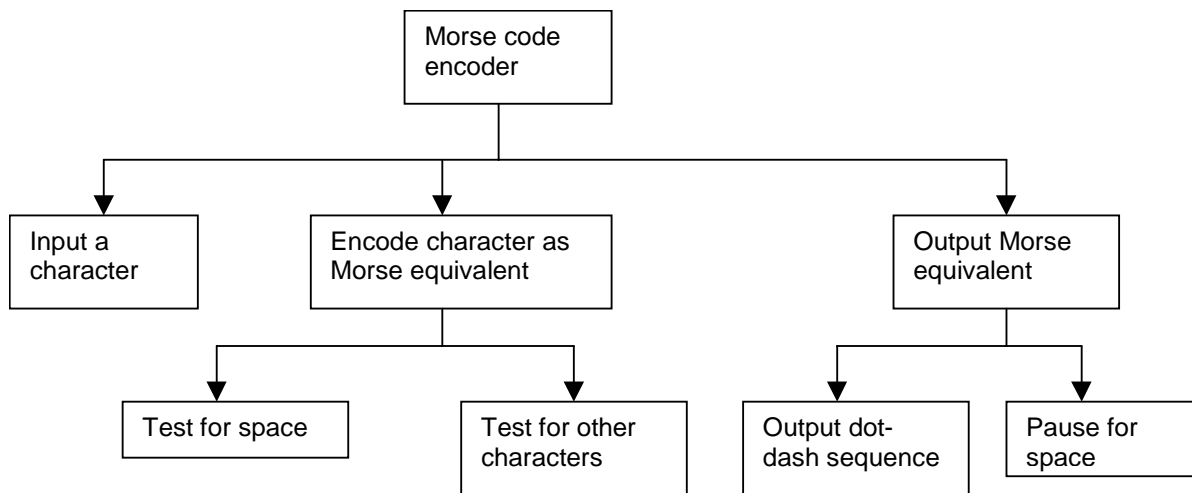
Morse code assigns a series of dots and dashes to each letter of the alphabet, the digits, and a few punctuation characters (period, comma, colon and semi-colon). Spaces between words are signified by a pause in the sequence of dots and dashes. A program is needed that reads a stream of characters and outputs the corresponding Morse code. The program should do this as fast as possible; it cannot wait for the whole message to be read in to start the encoding.

ANSWER TO PROBLEM 3: TOP-DOWN STEPWISE REFINEMENT

Problem statement: Morse code encoder

Problem description: The program will input a stream of characters from the keyboard and encode each one as the corresponding standard Morse code sequence of dots (the period) and dashes (the underscore). Spaces input from the keyboard signify spaces between words, and will produce a corresponding pause in the output. The program will operate in real time, so that the Morse code output will be produced as soon as possible after the characters are input.

TDSR diagram:



PROBLEM 4

A program is needed that can simulate the flow of traffic through any intersection of roads. The roads can be of any number of lanes, and have traffic lights or not. The simplest intersection is a T-junction onto a main road; the most complex might be an interchange between three highways (a “spaghetti junction”). Vehicles can be cars (including light trucks and vans) and heavy trucks. Each vehicle needs to be described by its position, speed, acceleration and intended path through the intersection. The simulation should be able to answer questions like “What is the maximum possible traffic flow through the intersection?” and “Where are the bottlenecks in terms of local flow?”.

ANSWER TO PROBLEM 1: OBJECT-ORIENTED ANALYSIS

Problem statement: Traffic flow simulator

Problem description: The program will simulate traffic flow through an intersection. Traffic consists of vehicles which can be of two kinds: cars (including light trucks) and heavy trucks. A vehicle is described at any point in time by its position, speed, acceleration, and intended path through the intersection. The intersection may have any number of roads, each of which can have any number of lanes, and optional traffic lights at the intersection. The simulation can be initialized with any number of vehicles at any part of the simulation given a typical scenario. The possible scenarios will be morning rush hour, evening rush hour, daytime, and nighttime. The simulation will run by updating vehicle parameters at one-second intervals. At each interval the simulation will report the traffic volume (vehicles per second) along any road into or out of the intersection. Queries may be made to report the maximum allowable flow through the intersection for any two roads (one in and one out), and to identify possible local bottlenecks. Other queries may be added in the future.

Noun/object analysis: The program will simulate traffic flow through an intersection. Traffic consists of vehicles which can be of two kinds: cars (including light trucks) and heavy trucks. A vehicle is described at any point in time by its position, speed, acceleration, and intended path through the intersection. The intersection may have any number of roads, each of which can have any number of lanes, and optional traffic lights at the intersection. The simulation can be initialized with any number of vehicles at any part of the simulation given a typical scenario. The possible

scenarios will be morning rush hour, evening rush hour, daytime, and nighttime. The simulation will run by updating vehicle parameters at one-second intervals. At each interval the simulation will report the traffic volume (vehicles per second) along any road into or out of the intersection. Queries may be made to report the maximum allowable flow through the intersection for any two roads (one in and one out), and to identify possible local bottlenecks. Other queries may be added in the future.

Objects: simulation HAS-A intersection
 intersection HAS-A (set of) road
 road HAS-A number of lanes
 road HAS-A traffic light
 road HAS-A (set of) vehicle
 road HAS-A direction (in or out)
 road HAS-A volume
 car IS-A vehicle
 truck IS-A vehicle
 vehicle HAS-A position
 vehicle HAS-A speed
 vehicle HAS-A acceleration
 vehicle HAS-A path
 simulation HAS-A scenario
 rushAM IS-A scenario
 rushPM IS-A scenario
 daytime IS-A scenario
 nighttime IS-A scenario
 simulation HAS-A (set of) query
 maxFlow IS-A query
 bottleneck IS-A query

Verb/method analysis: The program will simulate traffic flow through an intersection. Traffic consists of vehicles which can be of two kinds: cars (including light trucks) and heavy trucks. A vehicle is described at any point in time by its position, speed, acceleration, and intended path through the intersection. The intersection may have any number of roads, each of which can have any number of lanes, and optional traffic lights at the intersection. The simulation can be initialized with any number of vehicles at any part of the simulation given a typical scenario. The possible scenarios will be morning rush hour, evening rush hour, daytime, and nighttime. The simulation will run by updating vehicle parameters at one-second intervals. At each interval the simulation will report the traffic volume (vehicles per second) along any road into or out of the intersection. Queries may be made to report the maximum allowable flow through the intersection for any two roads (one in and one out), and to identify possible local bottlenecks. Other queries may be added in the future.

Methods: run
 initialize
 update
 report
 makeQuery
 addQuery

Class structure:

Simulation
 Intersection intersection
 Scenario scenario
 Query queries (set of)

Intersection
 Road roads (set of)

Road
 Float volume
 Vehicle vehicles (set of)
 PosInteger numLanes
 Boolean trafficLight
 PosInteger direction (in=1 or out=2)

Vehicle
 Float speed
 Float acceleration
 Position position (lane number within road)
 Path path (in road and out road)

Car inherits from Vehicle
 Truck inherits from Vehicle

Position
 Road road (reference)
 PosInteger laneNumber

Path
 Road inRoad (reference)
 Road outRoad (reference)

Scenario
 (no contained objects)

RushAM inherits from Scenario
 RushPM inherits from Scenario
 Daytime inherits from Scenario
 Nighttime inherits from Scenario

Query
 String description

Add methods:

Simulation
 Intersection intersection
 Scenario scenarios (set of)
 Query queries (set of)
 run()
 addQuery()
 makeQuery()

Scenario

(no contained objects)

initialize()

update()

report()