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 analysed in terms of what objects are involved, what kinds of interactions they have, and only as a last step, determining whether these interactiosn 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:

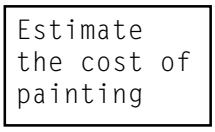
> *This program estimates the cost of painting the four walls and the ceiling of a room. Any doors or windows can be ignored. The height of the room can be assumed as a constant (8 ft.) but the program should input the length and width, in feet. The same paint is used to cover walls and ceiling. The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

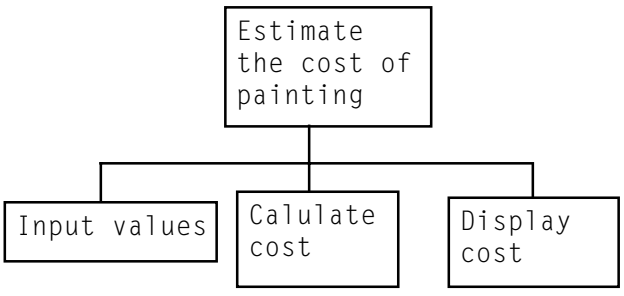
TOP-DOWN STEPWISE REFINEMENT

This methodology uses an age-old technique of divide and conquer. The aim is to analyse 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.

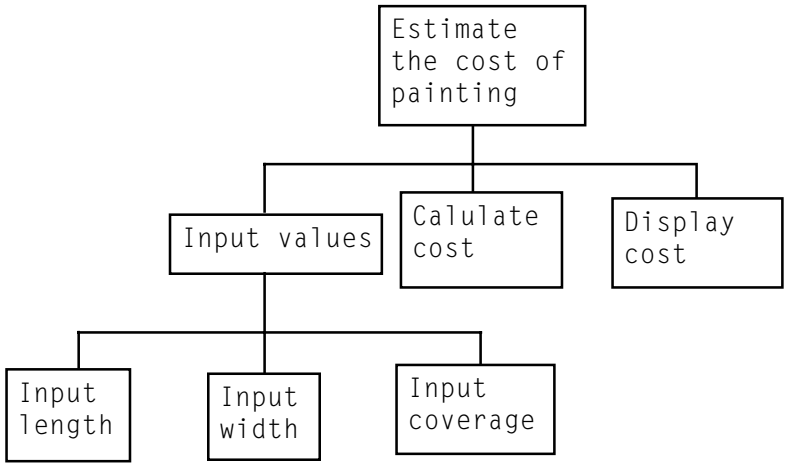
The TDSR diagram is a hierarchical chart where each block represents a program step. The successive levels in



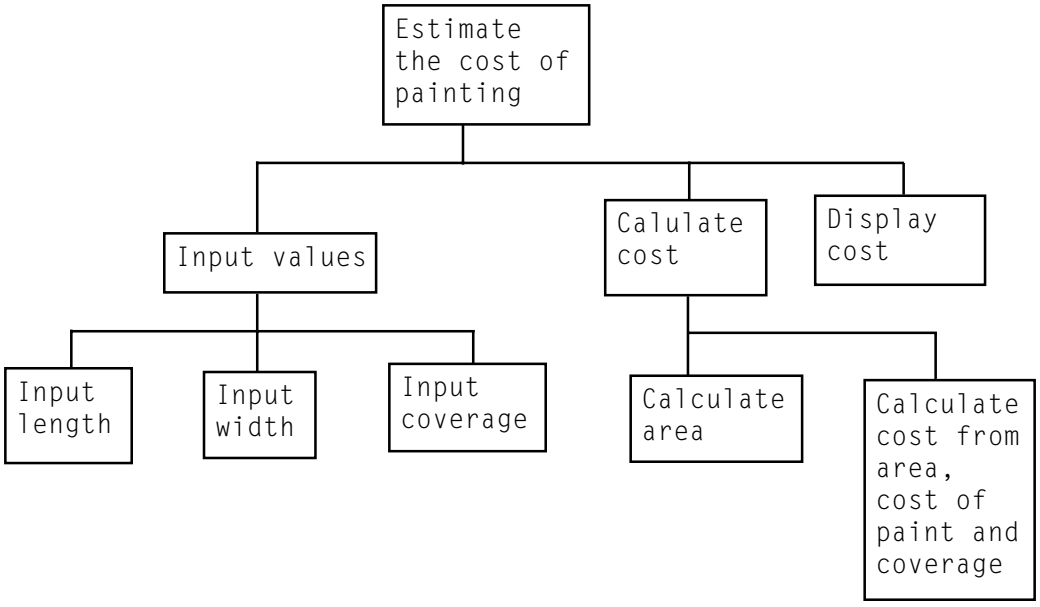
the diagram show how each step can be broken down further. Each level is a refinement of the problem analysis. In



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 bemaintained. 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:



Thinking through the calculation reveals that it is really in two stages: calculating the area of the walls and the ceiling, and then using that area to give the total cost:



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 functions to use. Too few functions and functions that are too long make a program that is difficult to read. On the other hand, too many functions and functions 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 function is probably too small, and a 100 line function is too large. However, we can use the TDSR diagram to guide our choices of the number and size of functions. A good rule of thumb is that there should be at least as many functions as the second-

level boxes. In our example, this is three. We could also add two functions 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):

```
void inputValues() {
...
}
void calculateArea() {
...
}

void calculateCostFromArea() {
...
}

void calculateCost() {
   calculateArea();
   calculateCostFromArea();
}
void displayCost() {
...
}
int main() {
   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:

```
/**********************************
 The paint estimator
**********************************/

#include <stdio.h>
const float paintCost = 12.5;
const float height = 8.0;

float costOfPainting, length, width,
      coverage, area; /* global variables */

void inputValues() {
   printf("Type in length, width, coverage: ");
   scanf("%f%f%f", &length, &width, &coverage);
}
void calculateArea() {
   area = 2 * (length + width) * height + length * width;
}

void calculateCostFromArea() {
   costOfPainting = area / coverage * paintCost;
}
```

```
void calculateCost() {
   calculateArea();
   calculateCostFromArea();
}
void displayCost() {
   printf("Cost of painting a room is %f\n",
           costOfPainting);
}
int main() {
   inputValues();
   calculateCost();
   displayCost();
}
```

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:

```
/**********************************
 The paint estimator
**********************************/

#include <stdio.h>
const float paintCost = 12.5;
const float height = 8.0;

float costOfPainting, length, width, coverage;

void inputValues() {
   printf("Type in length, width, coverage: ");
   scanf("%f%f%f", &length, &width, &coverage);
}
float calculateArea() {
   return 2 * (length + width) * height + length * width;
}

float calculateCostFromArea() {
   return area / coverage * paintCost;
}

void calculateCost(float l, float w, float c) {
   float area;

   area = calculateArea(l, w);
   calculateCostFromArea(area, c);
}
void displayCost(float l, float w, float c, float cp) {
   printf("Cost of painting a room %f by %f with\n
           paint of coverage %f is %f\n",
           l, w, c, cp);
}
int main() {
   float costOfPainting;
```

```
    inputValues();
    costOfPainting = calculateCost(length, width, coverage);
    displayCost(length, width, coverage, costOfPainting);
}
```

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:

```
int main() {
    float costOfPainting, length, width, coverage;

    length = inputLength();
    width = inputWidth();
    coverage = inputCOverage();
    calculateCost(length, width, coverage);
    displayCost(length, width, coverage, costOfPainting);
}
```

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 this 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. Here we give a C++ interpretation:

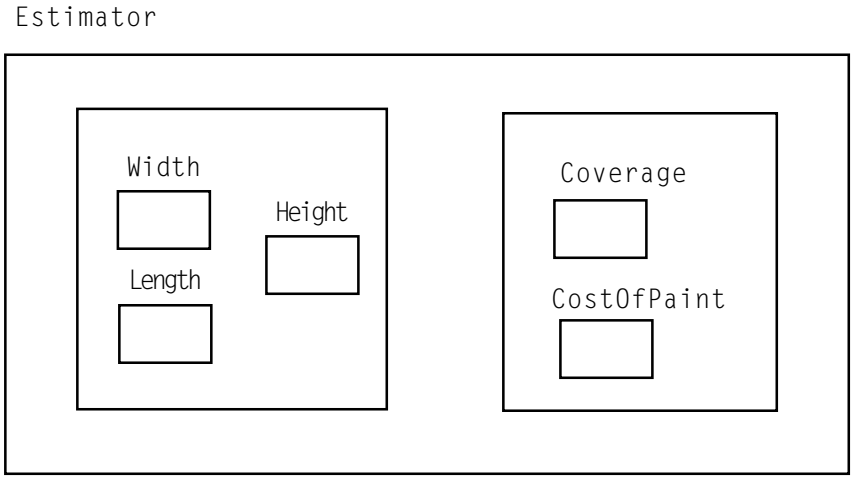
```
class Room {
private:
 float Width, Length;   // in ft.
 const float Height;    // in ft.
};

class Paint {
private:
 const float CostOfPaint;// in $/gallon
 float Coverage;        // in sq.ft./gallon
};

class Estimator {
private:
 Room room;
 Paint  paint;
 float CostOfPainting;  // in $
};
```

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 realtionshipts in terms of the objects created by the application.



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 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:

```
class Room {
private:
 float Width, Length;   // in ft.
 const float Height;    // in ft.
public:
 Room() : Height(8.0) {}      // a constructor to set the constant
 float getWidth() { return Width; }       // accessor method
 float getLength() { return Length; }     // accessor method
 float getHeight() { return Height; }     // accessor method
```

```
 void input() {
    cout << "Type width and length: ";     // input method prompts
    cin >> Width >> Height;                // the user
 }
};

class Paint {
private:
 const float CostOfPaint;// in $/gallon
 float Coverage;        // in sq.ft./gallon
public:
 Paint() : CostOfPaint(12.5) {}     // a constructor to set the
      // constant
 float getCoverage() { return Coverage; }       // accessor method
 float getCostOfPaint() { return CostOfPaint; } // accessor method
 void input() {                                 // input method
    cout << "Type coverage: ";
    cin >> Coverage;
}
};

class Estimator {
private:
 Room room;
 Paint  paint;
 float CostOfPainting;  // in $
public:
 Estimator() {}   // default constructor
 void estimate();
 void display() { cout << "Cost is " << CostOfPainting; }
 void run();
};
```

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:

```
void Estimator:estimate() {
 float wallArea =
    2 * (room.getWidth() + room.getLength()) * room.getHeight();
 float ceilingArea = room.getWidth() * room.getLength();
 CostOfPainting = (wallArea + ceilingArea) / paint.getCoverage() *
 paint.getCost();
}
```