6. Template and generic programming

Generic and Template

- Generic is a form of polymorphism. It provides a way of parameterized class or a function by use of a type, like as normal parameter or a function. Parameterized provides a way of defining an abstract algorithm without identifying specific value. In generic, a variable can be defined as type parameter although its property may not be known. In description of a class, that unknown parameter is matched and used. In C++ generic programming is implemented by template. So templates are used to create a family of classes or functions.
- Templates are used to classes in which different types of variable can be create in future. When an object of specific type is defined for actual use, the template definition for that class is substituted with the required data type. The templates are parameters that can be replaced by specific data type at the time of actual use of a class/function, the template are also called parameterized class/function.

…

Function Template

- A generic function defines set of operation that will be applied to various types of data. A generic function has the type of data that it will operate, open and passed to it as a parameter. A generic function is created with the keyword template and the general form of a template function definition is:

```
template <class type>
return_type function-name(parameter-list)
{
\\body of the function
}
```

type is a place holder name for the data type used by the function. This name can be used within the function definition.

…

```
//Example Function Template
#include<iostream.h>
#include<conio.h>
template <class T>
void swap(T &a, T &b)
{T temp;
temp=a;
a=b;
b=temp;}
void main()
{int i=10,j=20;
cout<<"Original
   value:\t"<<i<<"\t"<<j<<endl;
swap(i,j);
cout<<"Swapped value:\t"
   <<i<<"\t"<<j<<endl; getch();}
```

```
//Example Function with two generic types
   (multiple parameters)
#include<iostream.h>
#include<conio.h>
template <class T1, class T2>
void function ( T1 x, T2 y)
{cout<<x<<","<<y<<endl;}
void main()
{function(10,12.34);
function("HI",23L); getch();}
```

- In the above program, the template encapsulates two place holder types T1 and T2 separated by commas (,). The compiler replaces the place holder with the int, double and char, long respectively.

…

Note: typedef

- A keyword **typedef** is used to define a new type in terms of an old one
- Syntax: typedef float D;

  Makes a new type D a synonym for float

Class Template

- In generic classes, the actual type of data being manipulated will be specified as a parameter when object of class are created.

Syntax:

```
template <class Type>
class classname
{
//body of class };
```

…

```
//Example Class Template
#include<iostream.h>
#include<conio.h>
template <class T>
class Test
{T a;
public:
Test(T x)
{a=x;}
void show()
{cout<<"a="<<a<<endl;}};
```

```
void main()
{Test <int> test1(10); /*creating
   class from template class*/
Test <float> test2(12.12);
test1.show(); test2.show();
getch();}
```

Output:

```
a=10
a=12.12
```

For declaration outside the class

```
template <class T>
Test<T> :: Test(T x)
{ ............
............ }
```

…

```
/*Example: WAP using template
   to add 2 integers, two of float
   and one integer and 2 float
   respectively. Display final
   result in float.(class template
   with multiple parameters)*/
#include<iostream.h>
#include<conio.h>
template<class T1, class T2>
class Add
{T1 a;
T2 b;
public:
Add( T1 x, T2 y)
{a=x; b=y;}
```

```
void sum()
{cout<<"Sum of
   Numbers="<<(a+b)<<endl;}};
void main()
{Add<int, int> obj(12,23);
Add<float, float>
   obj1(12.1,32.3);
Add<int, float> obj2(12,2.2);
obj.sum(); obj1.sum();
obj2.sum(); getch();}
```

Output:

```
Sum of Numbers=35
Sum of Numbers=44.4
Sum of Numbers=14.2
```

…

Advantages of Class Templates:

a) One class template can handle different types of parameter

b) Compiler generates classes for only the types. If the template is instantiated for int type, compiler generates only an int version.

c) Template reduces the effort on coding for different data types to single set of code.

d) Testing and debugging effort is reduced.

Standard Template Library (STL)

- In programming, data structure is most important aspect of program design. So, object oriented programming provides the use of template library to use predefined data structures. These data includes class for vectors, list, sets, queue, stack, priority queue etc. The implementation of standard is vary in each techniques by which program becomes short. STL provides generic algorithm in which parameterized classes can be constructed. The implementation of generic algorithm in STL uses the ability not only to create a template container class but also to make template definition of individual function.

…

Component of STL

The main components of STL are:

i) Container

A container is an object that actually stores in memory. The STL container is implemented by the template class and it can store different types of data. The containers found in the STL are lists, maps, queues, sets, stacks, vectors etc. and are grouped into three categories (*container class*) as:

- Sequence Container

  stored elements are in sequence. Each element is related with each other by its position. Such as vector, list etc

- Associated Container

  elements should occur directly from the set or collection. The data or elements are stored in a fixed data structures. These containers are very slow. Such as set, map etc

…

- Derived Container

  STL provides three derived containers namely stack, queue and priority queue. These are created from other sequence containers.

ii) Algorithm

An algorithm is a procedure that is ued to process the data contained in the container. The STL includes different types of algorithm to provide support to task such as initializing, searching, copying, sorting, merging, etc. Algorithms are implemented by template function.

iii) Iterators

An iterator is an object (like pointer) that points to an element in a container. Iterators connect algorithm with containers and play a key role in a manipulation of data stored in the containers.

…

```
//STL example
#include<iostream>
#include<algorithm>
using namespace std;
 void printArray(int arr[], int len)
{ for(int i=0; i < len; i++)
cout << " " << arr[i]; cout << endl;}
void main()
{ int a[] = {5, 7, 2, 1, 4, 3, 6};
   printArray(a,7);
sort(a,a+7);
printArray(a, 7);
reverse(a, a+7);
printArray(a, 7);  cin.get(); }
```

Output:

```
5 7 2 1 4 3 6
1 2 3 4 5 6 7
7 6 5 4 3 2 1
```

STL Example(2)…

```
#include<iostream>
#include<conio.h>
#include<vector>
#include<algorithm>
using namespace std;
void main(){
vector<int> a;
ostream_iterator<int> out(cout," ");
for(int i=0;i!=5;a.push_back(i++)){}
copy(a.begin(),a.end(),out);
cout<<endl;
random_shuffle(a.begin(),a.end());
copy(a.begin(),a.end(),out);
cout<<endl;
sort(a.begin(),a.end());
copy(a.begin(),a.end(),out);
cout<<endl;
getch();}
```

Output:

```
0 1 2 3 4
3 0 4 2 1
0 1 2 3 4
```

Let's explain STL Example(2)…

1. We need the header files vector and algorithm
2. We declare a `vector` of `int` with no items.
3. We use the member function `push_back()` to add items to the *back* of the vector.
4. We declare `out` to be of type `ostream_iterator<int>` – this is an ostream (output) iterator
5. The 2nd argument in the `ostream_iterator` constructor is a string to place between successive values on the output stream.
6. The `ostream_iterator` allows us to write *to* the stream, but not read *from* it.
7. The iterator only supports `operator++`. Once we have passed a value, we cannot write to that position in the stream again.
8. The `copy` function (a generic algorithm) copies items from the vector to the output stream.
9. The `random_shuffle` function (a generic algorithm) randomizes the vector.
10. The `sort` function (a generic algorithm) then sorts the vector in place.
11. Both `sort` and `random_shuffle` take iterators of type `RandomAccessIterator` as arguments,

…

Namespace

- When you include a new-style header in your program, the contents of that header are contained in the std namespace. The namespace is simply a declarative region. The purpose of a namespace is to localise the names of identifiers to avoid name collision. Traditionally, the names of library functions and other such items were simply placed into the global namespace (as they are in C). However, the contents of new-style headers are place in the std namespace. Using the statement,

using namespace std;

- brings the std namespace into visibility. After this statement has been compiled, there is no difference working with an old-style header and a new-style one.

178

```
#include<iostream.h>
#include<conio.h>
void main(){
        int a,b; float d;
try{
        cout<<"Enter first numerator:";          cin>>a;
        cout<<"Enter second de-numerator:";
cin>>b;
        if(b==0) throw b;
        d=a/b;
cout<<"Result="<<d;
        }
catch( int e){
cout<<"Exception occur because denumerator = 0";}
        getch();}
```

…

Exception handling

- Exception handling is a subsystem of C++ that allows you to handle errors that occur at run time in a structured and controlled manner. With C++ exception handling, your program can automatically invoke an error handling routine when an error occurs. The principle advantage of exception handling is that it automates much of the error handling code that previously had to be coded 'by hand' in any large program. The proper use of exception handling helps you to create resilient code.
- C++ exception handling is built upon three keywords: **try, throw and catch**. In the most general terms, program statements that you want to monitor for exceptions are contained in a try block. If an exception (i.e. an error) occurs within the try block, it is thrown (using throw). The exception is caught, using catch, and processed.

179

```
#include<iostream> //exception handling for multiple catches
#include<conio.h>
int main(){    float  Op1, Op2, Result; char Opr;
try {
cout <<"To proceed, enter a number, an operator, and a
number:\n";
cin >> Op1 >> Opr >> Op2;
if(Opr != '+'&& Opr != '-'&& Opr != '*'&& Opr != '/')
throw Opr;
switch(Opr)           {
case'+': Result = Op1 + Op2; break;
case'-': Result = Op1 - Op2;break;
case'*': Result = Op1 * Op2; break;
case'/': Result = Op1 / Op2; break;}
cout <<"\n"<< Op1 <<" "<< Opr <<" "<< Op2 <<" = "<< Result; }
catch(const char n) {  cout <<"\nOperation Error: "<< n <<" is not a
valid operator"; }
getch();  return 0;   }
```

….

```
// Exception handling example
#include<iostream>
void main( )
{   cout << "Start\n";
 try {  // start a try block
 cout << "Inside try block\n";
 throw 10;    // throw an error
   }
 catch( int i) {  // catch an error
 cout << "Caught One! Number is: ";
 cout << i << "\n";   }
 cout << "end";   cin.get(); }
```

Output:

```
start
Inside try block
Caught One! Number is: 10
end
```

180

7. Object Oriented Design

- Important aspect of OOP is the creation of a universe of largely autonomous interacting agents (ie, OOP deals with decomposition of problem into a number of interacting agents called objects and their interactions to solve the problems)

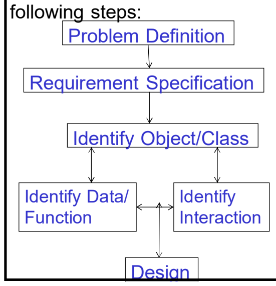
But how does one come up with such a system?

- By using a design technique called responsibility driven design (RDD) however some other techniques are there, RDD is one of the better technique because it facilitates the transition from analysis (OOA), design (OOD) to programming in simple way

183

Object Oriented Analysis and Design contain two different parts:

1. Object Oriented Analysis (OOA): method of analysis that examines the problems in terms of classes & objects and their interactions with the help of the following steps:



2. Object Oriented Design(OOD): A method of design that leads to an OO decomposition with the help of different notations to express logical, physical, static and dynamic model of a system. For this various methods are there, among them responsibility-driven design(RDD) is one.

7. Object Oriented Design

Responsibility-driven design(RDD):

A design technique used in OOD which is developed by Rebecca Wirfs-Brock and is driven by the determination and delegation of responsibilities . ie, design process begin with an analysis of behavior (responsibility) of a system then transit towards the identification of components, representation/design of components, implementation of components, integration of components and ends with maintenance & evoluation.

…

Responsibility implies Non-Interference:

- An OO approach is like a community of interacting indivisuals, each having assigned responsibility where the major task is determining the specific responsibilities for each individuals to solve the problem
- Responsibility implies a degree of independence or noninterference
- By specification and delegation, practitioners of OOD have developed a technique, which is called Responcibility-Driven Design (RDD).
- Major benefits of OO approach can be seen when reusing a subsystem from one projects to the other.

…

Programming in small and Programming in large

Main difference between small and large programming is based on its scale of creation & application. However in:

Programming in small

- Code developed by single programmer perhaps by a very small collection of programmers. A single individual can understand all expected the project from top to bottom or begin to end.
- The major problem in the software development process is the design and development of algorithms for dealing with the problem.

Programming in large

- The software system is developed by large team of programmers. Individual involved in the specification or design of the system may differ from those involved in the coding of individual components, which may differ as well from those involved in the integration of the various components in the final products. No single individual can be considered responsible for entire project.

…

- The major problem in the software development process is the management of details and the communication of information between diverse portions of the project.

Components and Behaviors

- **Components:** A component is simply an abstract entity that perform tasks i.e. fulfill some responsibilities. A component may ultimately be turned into a function, a structure or class, or a collection of other components. Component shows two important characteristics:

1. A component must have a small well-defined set of responsibilities.
2. A component should interact with other components to the minimal extent possible.

- The development of software is simplified by the identification and development of software component.

…

- **Behaviors:** how an object acts and reacts in terms of its state changes and message passing; the outwardly visible and testable activity of an object.

Role of Behavior in OOP

- Old design techniques were concetrating on data structure, and sequence of function call, with formal specification and has limited application that could not satisfied growing human requirements
- Behavior is something that can be described almost from the moment an idea is conceived, to both the programmers and client.
- So the design process of OOP in RDD begin with an analysis of behavior because the behavior of the system is usually understood long before any other aspect.
- Therefore Responsibility Driven Design (RDD) gives more emphasis on behavior at all level of software development cycle

…

Case Study in RDD

Lets us take an illustration to develop the Interactive Intelligent Kitchen Helper (IIKH) to demonstrate RDD technique.

- The IIKH is a pc-based application that will replace the index card system of recipes found in the average kitchen. Not only simple maintaining database of recipes, the kitchen helper assists in the planning of meals for an extended period, say a week. The user of IIKH can sit down at a terminal, browse the database of recipes, and interactively create series menus. The IIKH will automatically scale the recipes to any number of servings and will print out menus of the entire week, for a particular day, or for a particular meal. And it will print an integrated grocery list of all the items needed for the entire period.

…

Working through Scenarios

- The first task is to refine the specification. Initial specification are almost ambiguous and unclear on anything expect the most general points. There are several goals for this step.
- One objective is to get a better handle on the "look and feel" of the eventual product. This information can be carried out then be carried back to the client to see if it is in agreement with the original conception. It is likely, that the specifications for the final application will change during the creation of the software system, and it is important that the design be developed to easily accommodate change and that potential changes are noted as early as possible. At this point very high level decisions can be made concerning the structure of the eventual software system. In short, the activities to perform can be mapped into component.

…

Identification of components

- In OO approach software development process begins with **identification of components** with their responsibilities and collaborations.

What/Who Cycle

- The **identification of the components** takes place during the process of imagining the execution of a working system. Often this proceeds as a cycle of what/who questions. First, the programming team identifies what activity needs to be performed. Next, this is immediately followed by answering the question of who performs the action. In this manner, designing a software system is much like organizing a collection of people, such as a club. Any activity that is performed must be assigned as a responsibility to some component. If any action is to take place, there must be an agent assigned to perform it. Just as in running of a club any action to be performed must be assigned to some individual, in organizing an object-oriented program all actions must be the responsibility of some component.
- Answers of What/Who questions can be analyzed in effective way by using **CRC Cards** which give components a physical representation

…

CRC Cards [Component (class), Responsibility and Collaborator]

It is a collection of standard index card that contain three sections:

- A class (component) that represents a collection of similar object
- A responsibility is something that a class knows
- A collaborator is a class interacts with other class to fulfill its responsibilities
- Used to analyze the scenario (component, responsibility and interaction between the components) of a system to be developed

CRC card layout:

Class	
Responsibility	Collaborator

Example:

Customer	
Place order Knows names Knows address Knows customers number Knows order history	Order

…

Case example:

- Let us take IIKH case, at first, the team get answers for what/who questions for that system and decided that when the system begins, the user should be presented with an attractive information window that contain greeting also. The responsibility for displaying this window assigned to a component called **Greeter**. From this window user can selects one of several actions through a specified manner. Initially, the team identified five actions:

1. Casually browse the database of existing recipes
2. Add a new recipe to the database.
3. Edit an existing recipe.
4. Review an existing plan for several meals.
5. Create a new plan of meals.

…

- These activities seem to divide themselves into two groups. The first three are associates with the recipe database and the last two are associates with menu plans. As a result, the team decides to create other component corresponding to these, two responsibilities.
- The responsibility of the **recipe database** component is to maintain a collection of recipes. Further, the recipe component database must facilitate browsing the library of existing recipes, editing the recipes, and including new recipes in the database
- Each recipe will be identified with specific **recipe** component. Once a recipe is selected, control is passed to the associated recipe object. It consists of a list of ingredients and the steps needed to transform the ingredients into the final product.

…

The recipe component must also perform other activities such as it will display the recipe interactively on terminal screen. The user may be given the ability to change either the lists of ingredients or the instruction portions. Alternatively, the user may request a printed copy of the recipe. All of these actions are the responsibility of recipe components.

- Returning to the greeter will start a different scenario. This leads to the description of the **Plan Manager** component which permits the user to select a sequence of dates for planning and edit an existing plan, for this control is passed to the date object.
- The **Date** component holds a sequence of meals for a sequence of dates. User can edit specific meals, annotate information about dates ("Bob's Birthday", ``Christmas Dinner", and so on) and print out grocery list for entire set of meals.
- The **Meal** component holds information about a single meal. Allows user to interact with the recipe database to select individual

…

recipes for meals, sets number of people to be present at meal, recipes are automatically scaled. Can produce grocery list for entire meal, by combining grocery lists from individual scaled recipes.

Analysis through CRC Cards:

Greeter	
- displaying informative initial message - offer user choice option - pass control to either recipe database or plan manager	- database manager - plan manager

Recipe Database Manager	
- maintain the database of recipe - permits browsing of the database - permits editing of recipe - permits to add new recipes	-Recipe, -Meal

The Plan Manager	
- permits selecting sequence of planning - permits editing existing plan - associates with date objects	The Date Component

…

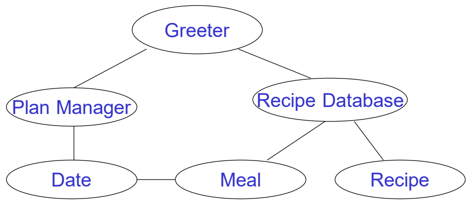
The Date Component	
- editing specific meals - Information about date, example birthday, dinner, etc. - permits list for entire set of meals	The Meal Component

The Meal Component	
- allow user to select individual recipe from database - automatically scales recipe to user-supplied number of serving - produce list of meal	Recipe database

Recipe Component	
- list of ingredients and transformation - edit these recipe -display itself on the output device	Greeter

…

Having walked through the various scenarios, the team eventually decides everything can be accomplished using six components for the IIKH system. Let's summarize the scenarios:



…

Interaction Diagram

- Description from ordinary figure may describe the static relationship between components; it is not very good for describing their dynamic interaction during the execution of scenario. A better for this purpose is an interaction diagram. Figure below shows the beginning of an interaction diagram for the interactive kitchen helper. In the diagram, time moves forward from top to the bottom. Each component is represented by a labeled vertical line. A component sending a message to another component is represented by a horizontal arrow from one line to another.
- Similarly, a component returning control and perhaps a result value back to the caller is represented by an arrow. The commentary on the right sides of the figure explains more fully the interaction taking place.

…

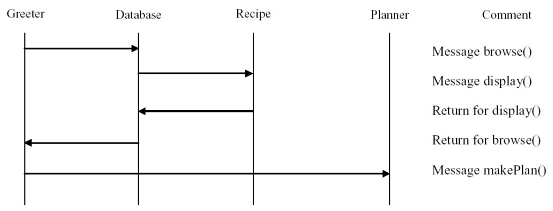


Figure: An Example of interaction diagram.

…

Software Components:

Simply component is an abstract entity that can perform task however there are many aspects to this term:

1. Behavior and State

Components are characterized their behavior i.e. what they can do and may also hold certain information. So component can be viewed as a pair consisting of behavior and state.

- The behavior of a component is the set of actions it can perform. The complete description of all the behavior for a component is sometimes called the protocol.
- The state of component represents all the information held within it.

It is not necessary that all components maintain state information. However most components will consists of a combination of behavior and state.

…

2. Instances and classes

Behavior of each component is the same; it is only the state that differs between individual components. The class is used to describe a set of objects with similar behavior. An individual representative of a class is known as instances. Behavior is associated with a class, not with an individual whereas state is a property of an individual.

3. Coupling and Cohesion

Cohesion is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating in a single component tasks that are related in some manner.

Coupling describes the relationship between software components. It is the desirable to reduce the amount of coupling as much as possible, since connection between software components inhibit ease of development, modification, or reuse.

…

Coupling is increased when one software component must access data values- the state-held by another component. Such situation should almost always be avoided in favor of moving a task into the list of responsibilities of the component that holds the necessary data.

4. Interface and Implementation

- Characterizing a software components by its behavior make possible for one programmer to know how to use a components developed by another programmer, without needing to know how the components is implemented. The purposeful omission of implementation details behind a simple interface known as information hiding. The component encapsulates the behavior, showing only how the component can be used, not the detail action it performed. Thus there exists two different views' of software system. The interface view describes what a software component can perform.

…

- The implementation view describes how a component goes about completing a task. For the successful reuse of general purpose software component in multiple projects, there must be minimal and well-understand interconnections between the various portion of the system. These ideas are adopted by David Parnas in a pair of rules, known as **Parnas's Principle**.
- The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide no other information.
- The developer of a software component must provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provide with no other information.

…

Formalizing the interface

- Formalize the interface is the process of outlook design of the components. The first step in this process is to formalize the patterns and channels of communication.
- Decision should be taken in order to implement each component by the use of general structure of class. A component with only one behavior and no internal state may be made into function. Component with many task are probably more easily implemented as classes. Names are given to each of the responsibilities identified on the CRC card for each component and these will be mapped on the functions or procedure names.
- Along with the names, the types of arguments to be passed to function are identified. The information maintained within the component itself should be described. If a component requires some data to perform a specific task, the source of the data either through argument or global value, or maintained internally by the component, must be clearly identified.

…

Date	
Responsibility	Collaborators
Maintain information about specific date Date (Year, month, day) – create new date	Plan Manager
Display & Edit() – display date information in window allowing user to edit entries BuildGrocery (List &) – add items from all means to grocery list.	Meal

Fig. Revised CRC card for the Date Component.

Coming up with Names

The selection of useful names is extremely important, name create the vocabulary with which the eventual design will be formulated. Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem. Often a considerable amount of time is spent finding just the right set of terms to describe the tasks performed and the object is manipulated.

…

The following general guidelines have been suggested:

- Use pronounceable noun.
- Use capitalization to mark the beginning of new word within a name, such as “CardReader” or “Card_Reader” rather than the less readable “cardreader”.
- Examine abbreviations carefully. An abbreviation that is clear to one person may be confusing to the next.
- Avoid names with several interpretations.
- Avoid digits within name. They are easy to misread as letters.
- Name functions and variables that yield Boolean values to so they describe clearly the interpretation of a true or false value.
- Take extra care in the selection of names for operations that are costly and infrequently used. By doing so, errors caused by using the wrong functions can be avoided.

…

Design and representation of components

- At this point, the design team can be divided into groups, each responsible for one or more software components. Now the important task is to transform the description of a component into a software system implementation. The major portion of this process is designing the data structures that will be used by subsystem to maintain the state information required to fulfill the assigned responsibilities.
- The selection of data structures is an important task, central to software design process, once they have been chosen the code used by the fulfillment of a responsibility is often almost evident. But the data structures should be carefully match to the task at hand. A wrong choice can result in complex and inefficient programs, while an intelligent choice can result in just the opposite. At this point, the description of behavior must be transformed into algorithms. These descriptions should then be matched against the expectations of each component listed as a collaborator, to ensure that expectations are fulfilled and necessary data items are available to carry out each process.

…

Implementation of components

- Once the design of each software subsystem is laid out, the next step is to implements each components with desired behavior. The task at this step is to implement the desired activities in a computer language.
- Often in the implementation of one component, it will become clear that certain information or actions might be assigned to yet another component that will act “behind the scene”, with little or no visibility to users of the software abstraction. Such components are sometimes known as facilitator.
- An important part of analysis and coding at this point is characterizing and documenting the necessary pre-conditions a software require to complete a task and verifying that the software component will perform correctly when presented with legal input values. This is establishing the correctness aspect of the algorithm used in the implementation of a component.

…

Integration of components

- Once software sub-system have been individually designed and tested, they can be integrated into a final products starting from a single base element are slowly added to the system and tested using stubs-simple dummy routines with no behavior or with very limited behavior. Testing of an individual component is often referred to as unit testing.
- Next, one or the other of the stubs can be replaced by more complete code. Further testing can be performed until it appears that the system is working as desired which is sometimes referred as integrating testing.
- The application is finally complete when all stubs have been replaced with working components. The ability to test components in isolation is greatly facilitated by the conscious design goal of reducing connections between components, since this reduces the need for extensive stubbing.

…

- Testing during integration can involve the discovery of errors, which then result in changes to some of the components. Following these changes the components should be once again tested in isolation before an attempt to reintegrate the software. Re-executing previously developed test cases following a change to a software component is sometimes referred to as regression testing.

Maintenance & Evolution

The term software maintenance describes the activities subsequent to the delivery of the initial working version of a software system. A wide variety of activities fall into this category:

- Errors discovered in the delivered products must be corrected.
- Requirement may change, perhaps as a result if government regulation or standardization among similar product.

…

- Hardware may change.
- User exception may change. User may expect greater functionality, lower cost and easier use.
- Better documentation may be required by user.

A good design recognizes the inevitability of changes and plans an accommodation for them from the very beginning.

Assignments

27. What are the differences between programming in small and programming in large? Explain.
28. What is meant by CRC card? Explain with example.
29. What do you understand by software component? Explain integration of component, with an example scenario to support your answer.
30. What are generic and templates? Create a template class stack to show push and pop operation on stack.
31. Differentiate between Interface and implementation.
32. Discuss the role of behavior in object oriented programming.
33. What are the basic steps involved in Responsibility Driven Design (RDD). How does it help in object oriented design? Support your answer with an example.
34. Write short notes on: a) STL b)Exception handling, also give example of each

214

More notations with example

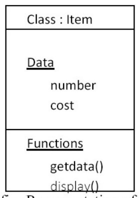
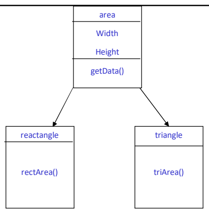


fig: Representation of Class

```
# include <iostream>
class item
{int number;
float cost;
 public:
void getdata(int a, float b)
{number=a;
cost=b;}
void display(void)
{cout << number << "\n";
cout << cost << "\n";}};
void main ( )
{item x;
   x.getdata(101, 145.5);
   x.display();
cin.get();}
```

215

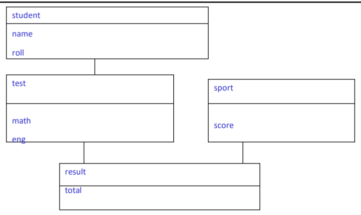
…



```
#include <iostream>
class area
{protected:
     float width, height;
  public:
     void getData(float a, float b)
     {width = a; height = b;}};
class rectangle: public area
{   public:
      void rectArea()
      {cout <<
"Total area of reactangle: " <<
(width * height) << endl;}};
class triangle: public area
{   public:
      void triArea()
      {cout <<
"Total area of triangle: " <<
(0.5*width * height) << endl;}};
void main() {rectangle rect;
   triangle tri;
   rect.getData(5.5,7.5);
   tri.getData(5,7);
   rect.rectArea(); tri.triArea ();}
```

216

…



Assume necessary functions yourself.

(for solution see the example of hybrid inheritance)

Assignment:

35. Solve the questions of semester end examinations (spring 2019 of object oriented programming in C++) and submit.

217