18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

Unit 4

Generic - Templates: Introduction

- Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.
- A template is a blueprint or formula for creating a generic class or a function that can operate on different types of data.
- Templates are widely used to implement the Standard Template Library (STL) to define containers, iterators and algorithms are examples of generic programming and have been developed using template concept
- **Templates can be represented in two ways:**

  1. **Function templates**
  2. **Class templates**

Function templates

- In Function template, a single function works with different data types.
- But a standard function works only with one set of data types.
- i.e. Generic functions template define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, add() function is implemented using a generic function, it can be implemented to add int, float or double type values.
- Whereas in Function overloading the same add() function redefined to add int, float or double type data type values. With this function overloading, the code size is not reduced, but in Function template only one function code definition can be implemented to add int, float or double type values.
- **Syntax of Function Template**

```
template < class Ttype >

return_type function_name (parameter_list)

{  // body of function.

}
```

- Where Ttype: It is a placeholder name for a data type used by the function. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.
- class: A class keyword is used to specify a generic type in a template declaration.
- **Function Templates with Multiple Parameters**
    - We can use more than one generic type in the template function by using the comma to separate the list.
    - Template function can accept any number of arguments of a different type.
    - **Syntax**

      Template <class T1, class T2,.....>

      return_type function_name (arguments of type T1, T2....)

      {

       // body of function.

      }

Example programs Function templates

Example1: Function template to add int, float, double data values

```
#include <iostream>
using namespace std;

template <class t>
t add (t a, t b)
{ return a+b; }

int main()
{
   cout<<"Integer sum is "<<add(5, 6)<<endl;
   cout<<"float sum is "<<add(5.3, 2.6)<<endl;
   return 0;
}
```

Output:

```
Integer sum is 11
float sum is 7.9
```

Example2: Function template with Multiple Parameters

```
#include <iostream>
```

```
#include <string>
using namespace std;

template<class T1,class T2>
void fun(T1 b ,T2 c)
{
    cout << "Value of b is : " <<b<< endl;
    cout << "Value of c is : " <<c<< endl;
}
int main()
{
   fun(20,30.5);
   fun(12.34,1234)
   fun(1999,"EBG");
   return 0;
}
```

Output:

```
Value of b is : 20
Value of c is : 30.5
Value of b is : 12.34
Value of c is : 1234
Value of b is : 1999
Value of c is : EBG
```

Overloaded Function Template:

- Like normal functions, Template functions can also be overloaded,
- The template function has the same name but different number or type of arguments.

Example3: Function template for overloaded Function

```
#include <string>
using namespace std;

template <class T>
 void display(T num)
   { cout<<num<<endl; }

template<class T1,class T2>
void display(T1 num1 ,T2 num2)
{   cout << num1<<"\t"<<num2;  }

template<class T1,class T2, class T3>

int main()
{
   display(20);     display(12.34,1234);     return 0;   }
```

Rules to select suitable template

- When the compiler encounters an overloaded template function, it uses the following rules to select a suitable template.
- If an exact match is found using an ordinary function, the compiler calls it.
- If an exact match is not found, it looks for a function template from which an exact match can be generated, if found, it calls for the same.
- If no match is found, ordinary overloading resolution for the function is tried.
- If no match is still found, an error is generated.
- If more than one match is found, then an ambiguity error is generated.

Class Templates

- A class template specify how individual classes can be constructed that supports similar operations for different data types.
- Therefore, templates enable programmers to create abstract classes that define the behavior of the class without actually knowing what data type will be handled by the class operations.
- The template class is prefixed with template <class Type> that tells the compiler that a template is being declared.
- The Syntax for declaring a class template is

  ```
  template <class Type1, class Type2, ……>
  class class_name
  {  //body of the class  };
  ```
- The Syntax for creating an object of the template class is

  ```
  Class_name <Type> objectname(arguments..);
  ```
- The process of creating a specific class from a template is called instantiation.
- An instantiated object of a template class is called a specialization.

Example programs for Class and Function templates

```
#include <iostream>   using namespace std;
template <class T>
class Calculator
```

```
{     private:      T num1, num2;
  public:
    Calculator(T n1, T n2)
      {          num1 = n1;             num2 = n2;       }
    void displayResult() {
        cout << "Numbers: " << num1 << " and " << num2 << "." << endl;
        cout << num1 << " + " << num2 << " = " << add() << endl;
        cout << num1 << " - " << num2 << " = " << subtract() << endl;
        cout << num1 << " * " << num2 << " = " << multiply() << endl;
        cout << num1 << " / " << num2 << " = " << divide() << endl;
    }
    T add() { return num1 + num2; }
    T subtract() { return num1 - num2; }
    T multiply() { return num1 * num2; }
    T divide() { return num1 / num2; }
};
int main() {
    Calculator <int> obj1(2, 1);
    Calculator  <float> obj2(2.4, 1.2);
    cout << "Int results:" << endl;       obj1.displayResult();
    cout << endl  << "Float results:" << endl;      obj2.displayResult();
    return 0;
}
```

Output:

```
Int results:
Numbers: 2 and 1.
2 + 1 = 3
2 - 1 = 1
```

```
2 * 1 = 2

2 / 1 = 2


Float results:

Numbers: 2.4 and 1.2.

2.4 + 1.2 = 3.6

2.4 - 1.2 = 1.2

2.4 * 1.2 = 2.88

2.4 / 1.2 = 2
```

Uses of Class Templates

- Remove code duplication
- Generic callback
- Re-use source code as opposed to inheritance and composition, which provides a way to reuse object code

Difference between class Template and Function Template

Class Template	**Function Template**
A class template is used when we want to create a class that is parameterised by a type, several functions defined in the same class template work on different data type.	Function template is used when we want to create a function that can operate on many different types. One function operates on many different data type.
A class template can have default template parameters	Function templates can't have default template parameters
A class template can be partially specialized while classes can	Function templates can't be partially specialized
Write Example template class Program	Write Example template function Program

Exception

- Exceptions are runtime anomalies or unusual conditions such as divide by zero, accessing array out its bound, running out of memory or disk space, overflow and underflow that a program may encounter during execution.

- Exceptions can be categorized as synchronous and asynchronous.
- Synchronous Exceptions like divide by zero, array index out of bound etc can be controlled by the program.
- Asynchronous Exceptions like an interrupt from the keyboard, hardware malfunction and disk failure are caused by events that are beyond the control of the program.

Exceptional Handling: try and catch

- Exceptional handling is used to detect and report an exceptional conditions, so that appropriate action can be taken to deal with it.
- It is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.
- Exceptional handling comprises two blocks 1.Try block 2. Catch block
- Try: this block is a group of statements that may generate an exception. When an exception is generated, it is thrown using the keyword throw. The throw keyword invokes the handling routine.
- Catch Block: this block catches the exception thrown from the try block and handles it in the way specified by the statements in the catch block.
- Syntax for try and catch :

```
try
{……………..
throw exception;   // protected code
……………….
}
catch (type arg)
{………………..
………………. // code to handle any exception
}
```

Example program : try, catch

```
#include <iostream>
using namespace std;

int main()
{ float a,b;
cout<<"Enter any two integer values";
cin>>a>>b;
try
{ if(b!=0)
  { cout<<"Result(a/b)="<<a/b<<endl; }
  else
  { throw b; }

}
catch(float ex)
{
cout<<"Exception caught:Divide By Zero \n";
}
}
```

Exceptional Handling: Multilevel exceptional - multiple catch

- It is possible to associate more than one catch statement with a single try block.
- This is usually done when a program segment has more than one condition to throw as an exception.
- In such cases, when an exception is thrown, the exception handlers are searched to find an appropriate match.
- Then, the first matched catch block is executed. After execution, the program control goes to the first statement after the last catch block for that try block. This means that all other catch blocks are ignored.
- However, if no match is found, then the program is terminated using the default abort ().
- **Syntax for multiple catch**

```
try
{   //try block   }
catch(data type1 arg)
{   //catch block1   }
catch(data type2 arg)
{   //catch block2   }
………………
……………..
catch(data typeN arg)
{    //catch blockN    }
```

<u>Example Program for multiple catch</u>

```
#include <iostream>
using namespace std;
void test(int x) {
   try
  {
      if (x > 0)
         throw x;
      else if (x < 0)
         throw  'N';
     else
         throw "zero";
```

```
    }

   catch (int x)
   {    cout << "Catch a integer and that integer is positive:" << x<<endl;     }
   catch (char x)
   {    cout << "Catch a integer and that integer is negative:" << x<<endl;     }

   catch (const char* x)
   {      cout << "Catch a integer and that integer is:" << x;      }
}


int main() {
    cout << "Testing multiple catches\n";
    test(10); test(-100);
    test(0);

   return 0;
}
```

Output:

```
Testing multiple catches
Catch a integer and that integer is positive:10
Catch a integer and that integer is negative:N
Catch a integer and that integer is:zero
```

Catch All Exceptions or Default Exception Handling

- The catch-all clause, catch (...) matches exceptions of any type.
- If present, it has to be the last catch clause in the handler-seq.
- Catch-all block may be used to catch every possible exception thrown from the try block.

Simple C++ Program for Catch All or Default Exception Handling

```
#include <iostream>
using namespace std;

int main() {
   int var = 0;

   try {
         if (var == 0)
         {          throw var;        }
   }
   catch (float ex) {
       // Code executed when exception Catch with float type
```

```
    cout << "Float Exception catch : Value :" << ex;
  }
  catch (...) {
    // Code executed when exception Catch : for default
    cout << "Default Exception Catch";
  }

  return 0;
}
```

Output : Default Exception Catch

(Note : since no float type exception, default exception is executed)

Rethrowing Exception:

- In some situation, the exception handler in the catch block may decide to rethrow the exception without processing it.
- If a catch block cannot handle the particular exception it has caught, it can rethrow the exception.
- Syntax to rethrow the exception is **throw;**
- A throw statement without any exception explicitly mentioned causes the current exception to be thrown to the next try catch block.

Example program for Rethrowing Exception

```
#include <iostream>  using namespace std;
int var = 0;
void display()
{    try {
         if (var == 0)
         {          throw var;        }    //  throw exception type int
   }
   catch (int ex) {             throw;     }
}
int main() {
         try {         display();      }
    catch (int ex) {
           cout << "Integer Exception catch - Value :" << ex; }
        return 0;
}
```

Output : Integer Exception catch - Value :0

Restricting the Exceptions that can be Thrown

- To restrict a function to throw only certain specified exceptions, throw list clause is used in the function defintion.
- Syntax:

```
return_type function_name(arg-list) throw(type-list)
{
      // ...
 }
```

Example program for Restricting the Exceptions that can be Thrown

```
#include <iostream>   using namespace std;
int a;
void display(int a)throw(int, char, float)
{ try {
        if (a == 1)     {   throw a;   }
        if (a==  0 )   {   throw 'c';   }
        if (a== -1)   {   throw 1.1;   }
   }
   catch (int ex) { cout << "int Exception catch : Value :" << ex; }
   catch (char ex) {cout << "CharException catch : Value :" << ex; }
   catch (float ex) { cout << "float Exception catch : Value :" << ex; }
}

int main() {

      cout << "enter Value (1 or 0 or -1)" ;
      cin>> a;
      display(a);
      return 0;
}
```

Handling uncaught exception (Stack unwinding)

- A call stack, function stack, execution stack, run time stack, or a control stack is used to keep a track point to which the active function will return after completing its execution.
- If a Function doesn't handle the exception, the program stack unwinds the stack and control returns to main(). But there's no exception handler here either in main, then main() terminates. At this point, we just terminated our application!

Example program for Handling uncaught exception (Stack unwinding)

```
#include <iostream>          using namespace std;
#include <cmath> // for sqrt() function
double mySqrt(double x)
{
   // If the user entered a negative number, this is an error condition
   if (x < 0.0)
       throw "Can not take sqrt of negative number"; // throw exception of type const char*
      return sqrt(x);
}
```

```
int main()
{
   cout << "Enter a number: ";
   double x;
   cin >> x;
   // Look, no exception handler!
   cout << "The sqrt of " << x << " is " << mySqrt(x) << '\n';

   return 0;
}
```

Output 1:

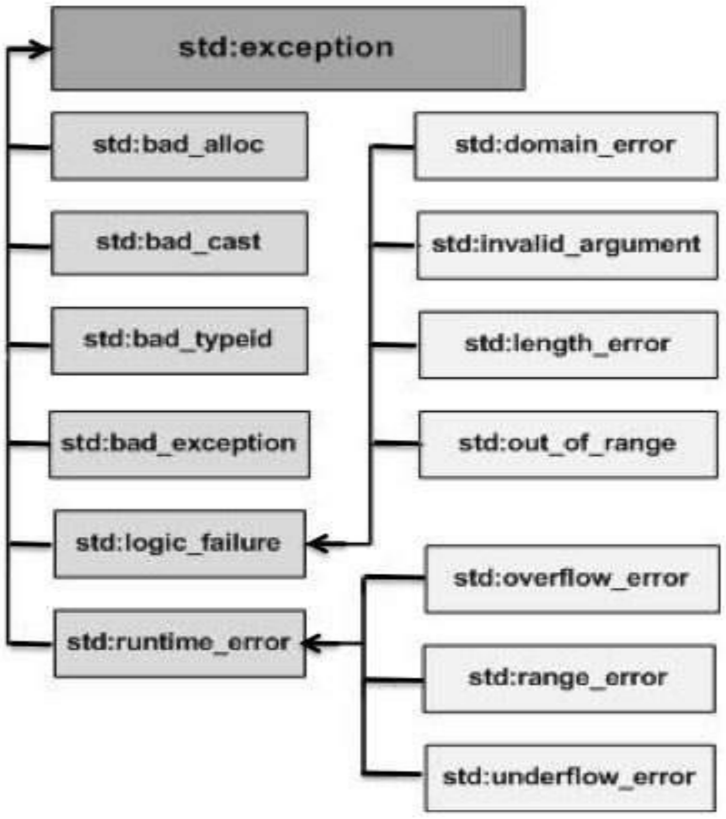
```
Enter a number:  -5
terminate called after throwing an instance of 'char const*'
```

Output 2:

```
Enter a number: 5
The sqrt of 5 is 2.23607
```

Standard Exception

- C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown below –



Description of each exception mentioned in the above hierarchy are

S.No	Exception	Description
1	std::exception	An exception and parent class of all the standard C++ exceptions.
2	std::bad_alloc	This can be thrown by new.
3	std::bad_cast	This can be thrown by dynamic_cast.
4	std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program.
5	std::bad_typeid	This can be thrown by typeid.
6	std::logic_error	An exception that theoretically can be detected by reading the code.
7	std::domain_error	This is an exception thrown when a mathematically invalid domain is used.
8	std::invalid_argument	This is thrown due to invalid arguments.
9	std::length_error	This is thrown when a too big std::string is created.
10	std::out_of_range	This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
11	std::runtime_error	An exception that theoretically cannot be detected by reading the code.
12	std::overflow_error	This is thrown if a mathematical overflow occurs.
13	std::range_error	This is occurred when you try to store a value which is out of range.
14	std::underflow_error	This is thrown if a mathematical underflow occurs.

- The exception class has a public virtual function called what() that returns a string that explains the error.

<u>Example program for Standard Exception</u>

```
#include <iostream>
#include <exception>   using namespace std;
int main()
{
   try
   { int *arr = new int[1000000000000000000000000000000000000000000];
```

```
        cout<<"memory allocated"; }

        catch (exception & e)    // to catch std exception only
        { cout <<"memory allocation failed"<<e.what(); }

        cout<<" exit main";
       }
```

Output:

```
memory allocation failed  bad_alloc
exit main
```

Exception Handling - User defined Exception

C++ allows programmers to define their own exceptions by inheriting and overriding exception class functionality.

Example Program:

```
#include <iostream>
#include <exception>  using namespace std;
struct MyException : public exception    // inherting class exception
{   const char * what () const throw ()
     {   return "C++ Exception";    }
};

int main() {
   try {
      throw MyException();    //throwing customized exception
   }
     catch(MyException & e)
    {      cout << "MyException caught" << endl;
           cout << e.what() << endl;
     }
cout<<"\n Existing main()";  return 0;
}
```

Exceptional Handling: throw and throws

S.NO	throw	throws
1	throw keyword is used to explicitly throw an exception within a method or block of code	The throws keyword is used to declare which exceptions can be thrown from a method
2	Internally throw is implemented as it is allowed to throw only single exception at a time i.e we cannot throw multiple	On other hand we can declare multiple exceptions with throws keyword that could get thrown by the function where throws

	exception with throw keyword.	keyword is used.
3	Supported by C++	Both throw and throws are supported by java

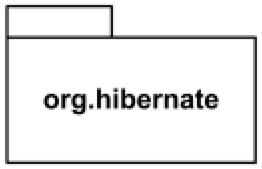
Both throw and throws are concepts of exception handling. The throws keyword is used to declare which exceptions can be thrown from a method, while the throw keyword is used to explicitly throw an exception within a method or block of code.

Exceptional Handling: finally

C++ does not support 'finally' blocks.

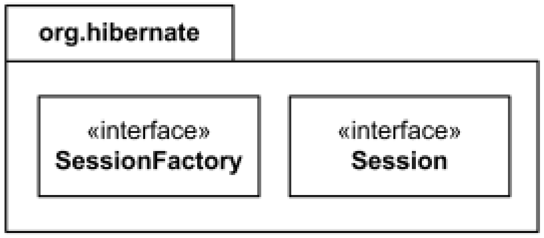
Dynamic Modeling: Package Diagram

- Package diagram is a type of **UML structure diagram** which shows packages and dependencies between the packages.
- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- A package is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain subordinate packages as well as other kinds of model elements. All kinds of UML model elements can be organized into packages.
- A package is rendered as a **tabbed folder** - a rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.

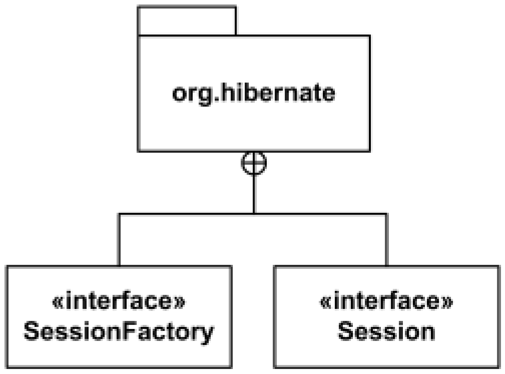


Package org.hibernate

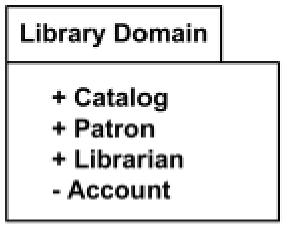
- The members of the package may be shown within the boundaries of the package. In this case the name of the package should be placed on the tab.



- A diagram showing a package with content is allowed to show only a subset of the contained elements according to some criterion.
- Members of the package may be shown outside of the package by branching lines from the package to the members. A plus sign (+) within a circle is drawn at the end attached to the namespace (package). This notation for packages is semantically equivalent to composition (which is shown using solid diamond.)



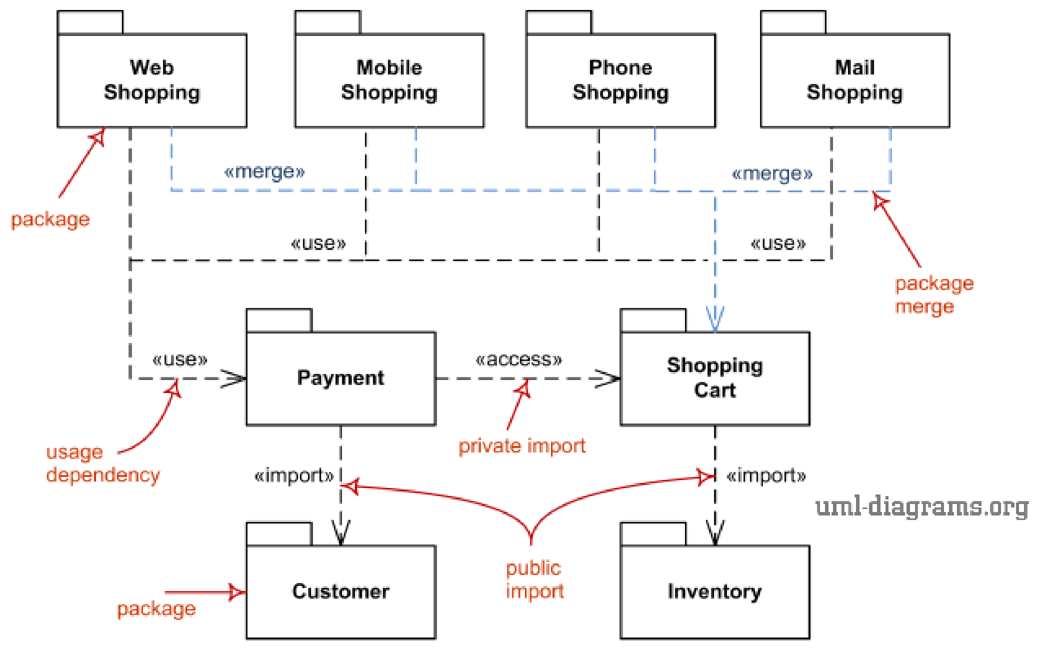
- Owned and imported elements may have a visibility that determines whether they are available outside the package.
- If an element that is owned by a package has visibility, it could be only public or private visibility. Protected or package visibility is not allowed. The visibility of a package element may be indicated by preceding the name of the element by a visibility symbol ("+" for public and "-" for private).



- Package Diagram - Dependency Notation : There are two sub-types involved in dependency.
- They are <<import>> & <<access>>.
- <<import>> - one package imports the functionality of other package.
- <<access>> - one package requires help from functions of other package.

Example Package Diagram for online shopping

- Some major elements of the package diagram are shown on the drawing below. Web Shopping, Mobile Shopping, Phone Shopping, and Mail Shopping packages merge Shopping Cart package. The same 4 packages use Payment package. Both Payment and Shopping Cart packages import other packages.



Implementation Diagram

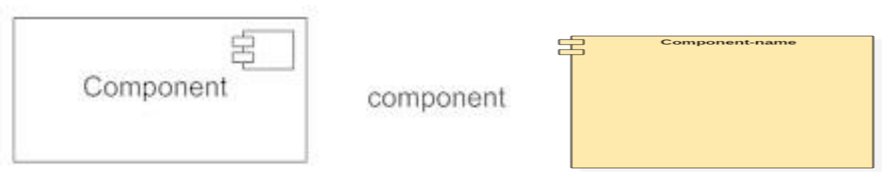
- It shows the implementation phase of systems development, such as the source code structure and the run-time implementation structure.
- There are two types of implementation diagrams:
    - **component diagrams** show the structure of the code itself,
    - and **deployment diagrams** show the structure of the runtime system

UML Component Diagram

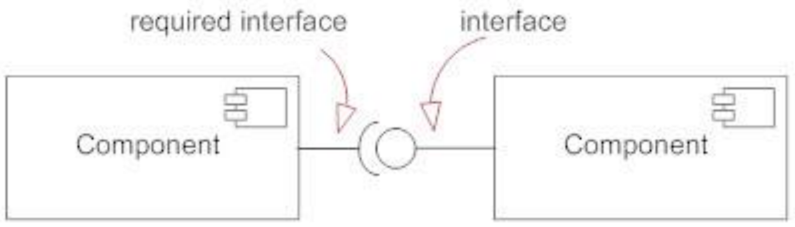
- Component diagram is a type of **UML structure diagram**
- Component diagrams are used to model the physical aspects of a system.
- It model the physical components(such as source code, executable program, user interface, libraries,dll's) in a design.
- Component diagrams can also be described as a static implementation view of a system.
- Static implementation represents the organization of the components at a particular moment.
- A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.
- Purpose of Component Diagrams
    - Visualize the components of a system.
    - Construct executables by using forward and reverse engineering.
    - Describe the organization and relationships of the components.
- Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.
- Before drawing a component diagram, the following artifacts are to be identified clearly

1. Files used in the system.
2. Libraries and other artifacts relevant to the application.
3. Relationships among the artifacts.

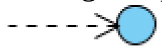
- **Component diagram Notations**
  1. A component: A component is a replaceable and executable piece of a system whose implementation details are hidden. It is represented as a rectangle with a smaller rectangle in the upper right corner with tabs or the word written above the name of the component to help distinguish it from a class.



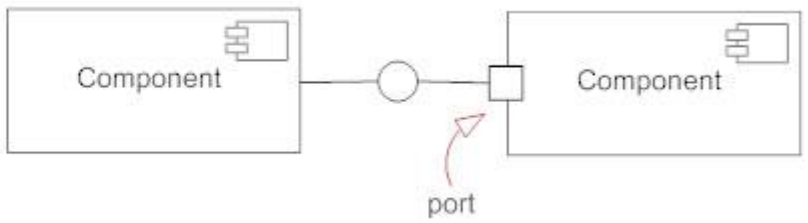
  2. Interface: The Interface is a named set of public features. There are two types of Interfaces in Component Diagram: 1. Provided interfaces 2. Required interfaces. An interface (semi-circle on a stick) describes a group of operations used (required) or created by components. A full circle represents an interface created or provided by the component.



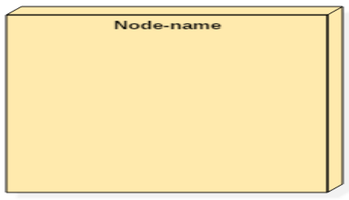
  3. Dependencies: Draw dependencies among components using dashed arrows.



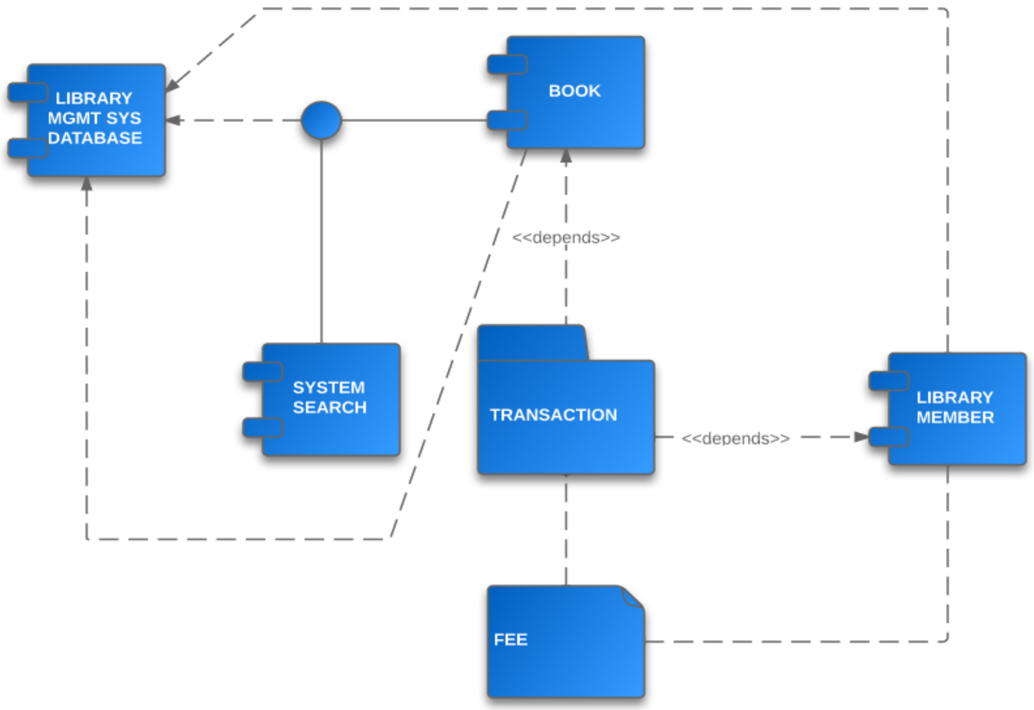
  4. Port: Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.



  5. Node: Represents hardware or software objects, which are of a higher level than components.



Example Component diagram for a library management system



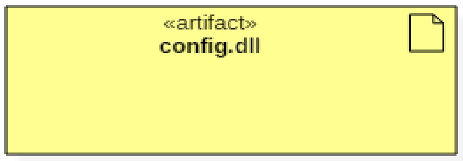
UML Deployment Diagram

- Deployment diagram is a type of **UML structure diagram**
- Deployment diagram represents or describe the static deployment view of a system.
- It specifies the physical hardware on which the software system will execute.
- It also determines how the software is deployed on the underlying hardware.
- It maps software pieces of a system to the device that are going to execute it.
- In most cases, component diagrams are used in conjunction with deployment diagrams to show how physical modules of code are distributed on various h/w platform.
- In many cases, component and deployment can be combined. Where Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.

- Deployment diagram show the configuration of run-time processing elements and the software components, processes, and objects that live in them.
- Software component instances represent run-time manifestations of code units.
- Before drawing a deployment diagram, the following artifacts should be identified.
    - Nodes
    - Relationships among nodes

Deployment diagram elements:

- A variety of shapes make up deployment diagrams.
- Artifact: A product developed by the software, symbolized by a rectangle with the name and the word "artifact" enclosed by double arrows.



- Communication path: A straight line that represents communication between two device nodes.
- Package: A file-shaped box that groups together all the device nodes to encapsulate the entire deployment.
- Component: An entity required to execute a stereotype function. Take a look at this guide for UML component notation.
- Node: A hardware or software object, shown by a three-dimensional box.

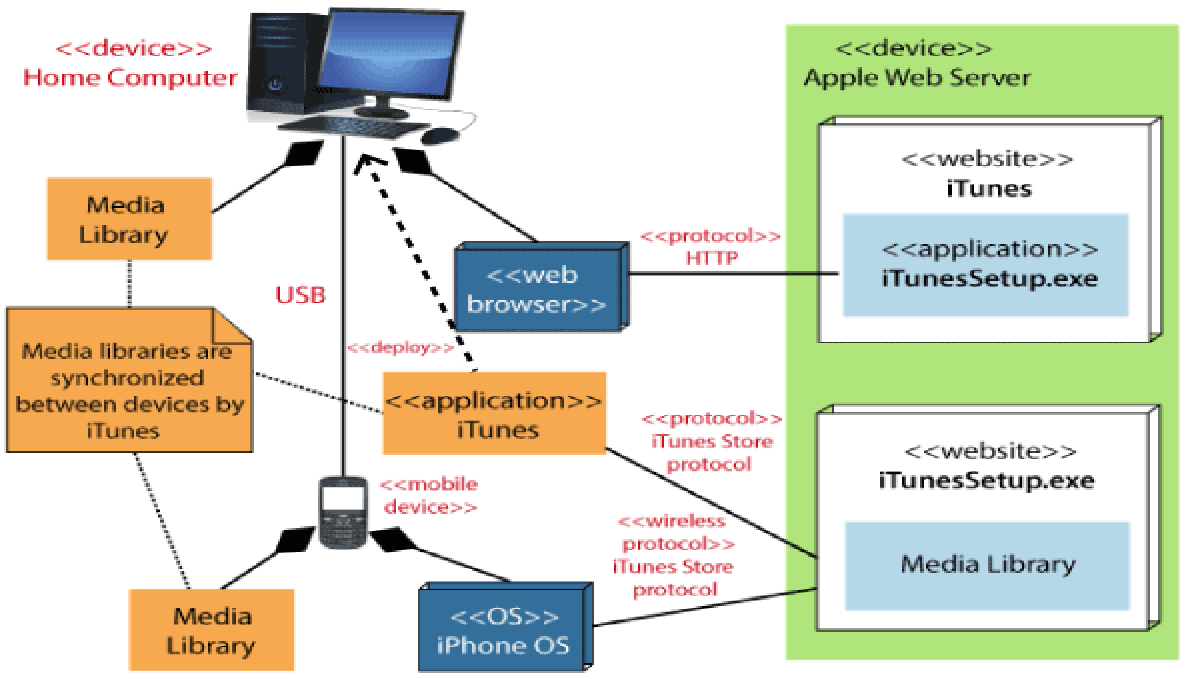
Example deployment diagram for the Apple iTunes application

- The iTunes setup can be downloaded from the iTunes website, and also it can be installed on the home computer.

- Once the installation and the registration are done, iTunes application can easily interconnect with the Apple iTunes store.

- Users can purchase and download music, video, TV serials, etc. and cache it in the media library.

- Devices like Apple iPod Touch and Apple iPhone can update its own media library from the computer with iTunes with the help of USB or simply by downloading media directly from the Apple iTunes store using wireless protocols, for example; Wi-Fi, 3G, or EDGE.


<<device>>
Home Computer
Media Library
USB
<<deploy>>
<<web browser>>
<<protocol>> HTTP
<<device>>
Apple Web Server
<<website>>
iTunes
<<application>>
iTunesSetup.exe
Media libraries are synchronized between devices by iTunes
<<application>>
iTunes
<<protocol>> iTunes Store protocol
<<mobile device>>
<<wireless protocol>> iTunes Store protocol
<<website>>
iTunesSetup.exe
Media Library
Media Library
<<OS>>
iPhone OS