Slide 3

OBJECT ORIENTED METHODOLOGY

Importance of Object Orientation

Higher level of abstraction

- The object-oriented approach supports abstraction at the object level.
- Since objects encapsulate both data (attributes) and functions (methods), they work at a higher level of abstraction. The development can proceed at the object level and ignore the rest of the system for as long as necessary. This makes designing, coding, testing, and maintaining the system much simpler.

Seamless transition among different phases of software development.

•The traditional approach to software development requires different styles and methodologies for each step of the process.

•Moving from one phase to another requires a complex transition of perspective between models that almost can be in different worlds.

•The object-oriented approach, on the other hand, essentially uses the same language to talk about analysis, design, programming, and database design. This seamless approach reduces the level of complexity and redundancy and makes for clearer, more robust system development.

Encouragement of good programming techniques.
In a properly designed system, the classes will be grouped into subsystems but remain independent; therefore, changing one class has no impact on other classes.

Promotion of reusability.
Objects are reusable because they are modeled directly out of a real-world problem domain. Each object stands by itself or within a small circle of peers (other
objects). Within this framework, the class does not concern itself with the rest of the system or how it is going to be used within a particular

The Rational Unified Process

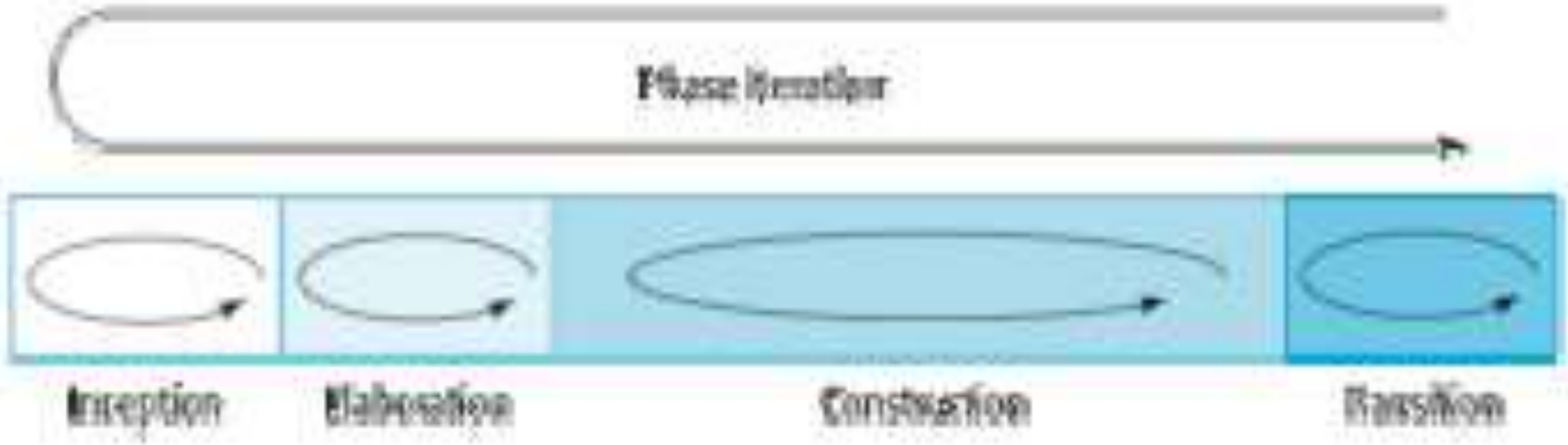
- This is an Object Oriented Methodology derived from the work on the UML and associated process.
- Brings together aspects of the Waterfall model, Incremental Development and the Reuse-Oriented Software Engineering.
- Normally described from 3 perspectives
  - A dynamic perspective that shows phases over time;
  - A static perspective that shows process activities;
  - A practive perspective that suggests good practice

- RUP establishes a unifying and unitary framework around the Booch, Rumbaugh, and Jacobson works by utilizing the unified modeling language (UML) to describe, model, and document the software development process.
- The main motivation here is to combine the best practices, processes, methodologies, and *guidelines* along with UML notations and diagrams for better understanding object-oriented concepts and system development

RUP Approach

- Essential Principles:
  - Attack major risks early and continuously
  - Ensure that you deliver value to your customer
  - Stay focused on executable software
  - Accommodate change early in the project
  - Baseline an executable architecture early on
  - Build your system with components
  - Work together as one team
  - Make quality a way of life, not an afterthought

Phases in the Rational Unified Process



RUP phases

- Inception
  - Establish the business case for the system.
- Elaboration
  - Develop an understanding of the problem domain and the system architecture.
- Construction
  - System design, programming and testing.
- Transition
  - Deploy the system in its operating environment

In-phase iteration

- In-phase iteration
  - Each phase is iterative with results developed incrementally.
- Cross-phase iteration
  - As shown by the loop in the RUP model, the whole set of phases may be enacted incrementally.

RUP Work Flows

A Workflow is a sequence of tasks in a business process. Workflows are the paths that describe how something is done. RUP workflows include

- –Business modeling
- –Requirements management
- –Analysis and design
- –Implementation
- –Deployment
- –Test
- –Project management
- –Change management
- –Environment

Workflows in the Rational Unified Process

- Business modeling - The business processes are modeled using business use cases.
- Requirements - Actors who interact with the system are identified and use cases are developed to model the system requirements.
- Analysis and design - A design model is created and documented using architectural models, component models, object models and sequence models.
- Implementation - The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.

- Testing - Testing is an iterative process that is carried out in conjunction
  with implementation. System testing follows the completion of the implementation.
- Deployment - A product release is created, distributed to users and installed in their workplace.
- Configuration and change management - This supporting workflow manage changes to the system.
- Project management - This supporting workflow manages the system development
- Environment - This workflow is concerned with making appropriate software tools available to the software development team.

UML

- UML stands for "Unified Modeling Language"
- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design

UML

UML is a modeling language for visualising, specifying, constructing and documenting the artifacts of software systems.

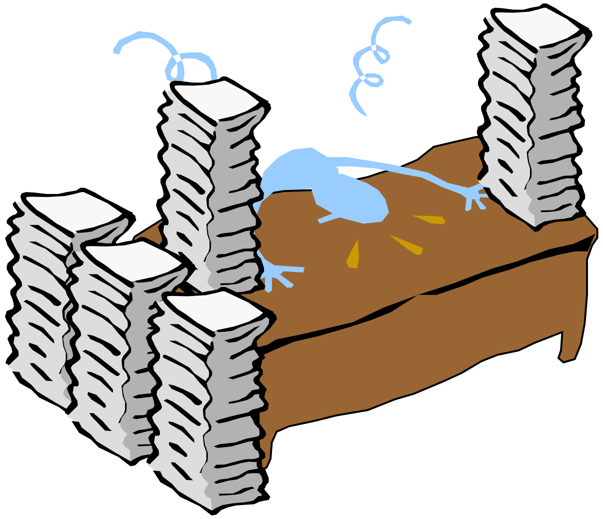
Visualising

More on UML...

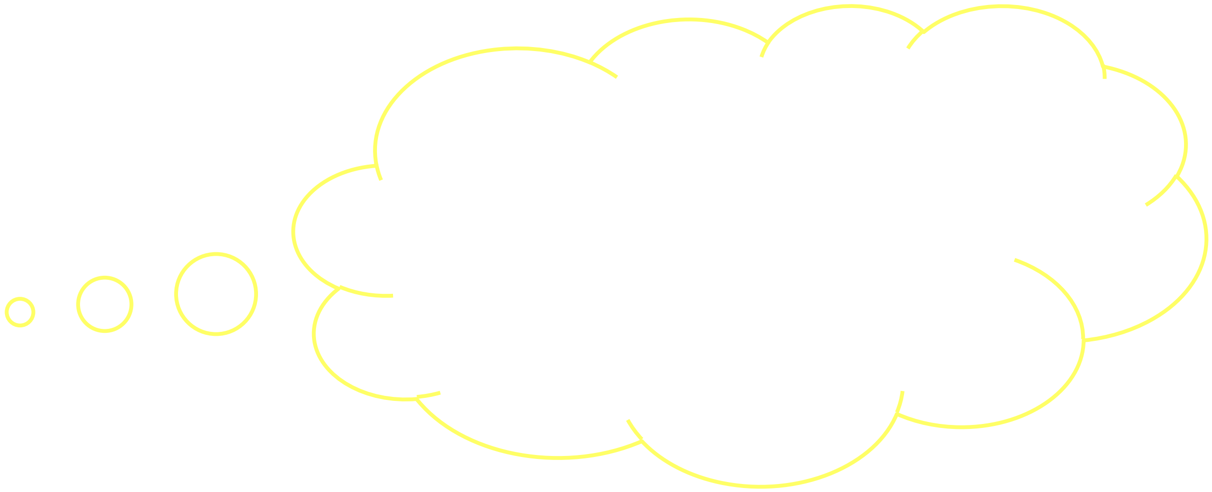
Specifying - *UML provides the means to model precisely, unambiguously and completely, the system in question.*

Constructing

More on UML…



Documenting - *every software project involves a lot of documentation - from the inception phase to the deliverables.*



UML Diagrams

- Class Diagram – *the most common diagram found in OOAD, shows a set of classes, interfaces, collaborations and their relationships. Models the static view of the system.*
- Object Diagram – *a snapshot of a class diagram; models the instances of things contained in a class diagram.*
- Use Case Diagram – *shows a set of "Use Cases" (sets of functionality performed by the system), the "actors" (typically, people/systems that interact with this system[problem-domain]) and their relationships. Models WHAT the system is expected to do.*

UML Diagrams

- Sequence Diagram - *models the flow of control by time-ordering; depicts the interaction between various objects by of messages passed, with a temporal dimension to it.*
- Collaboration Diagram - *models the interaction between objects, without the temporal dimension; merely depicts the messages passed between objects.*
- Statechart Diagram - *shows the different state machines and the events that leads to each of these state machines. Statechart diagrams show the flow of control from state to state.*

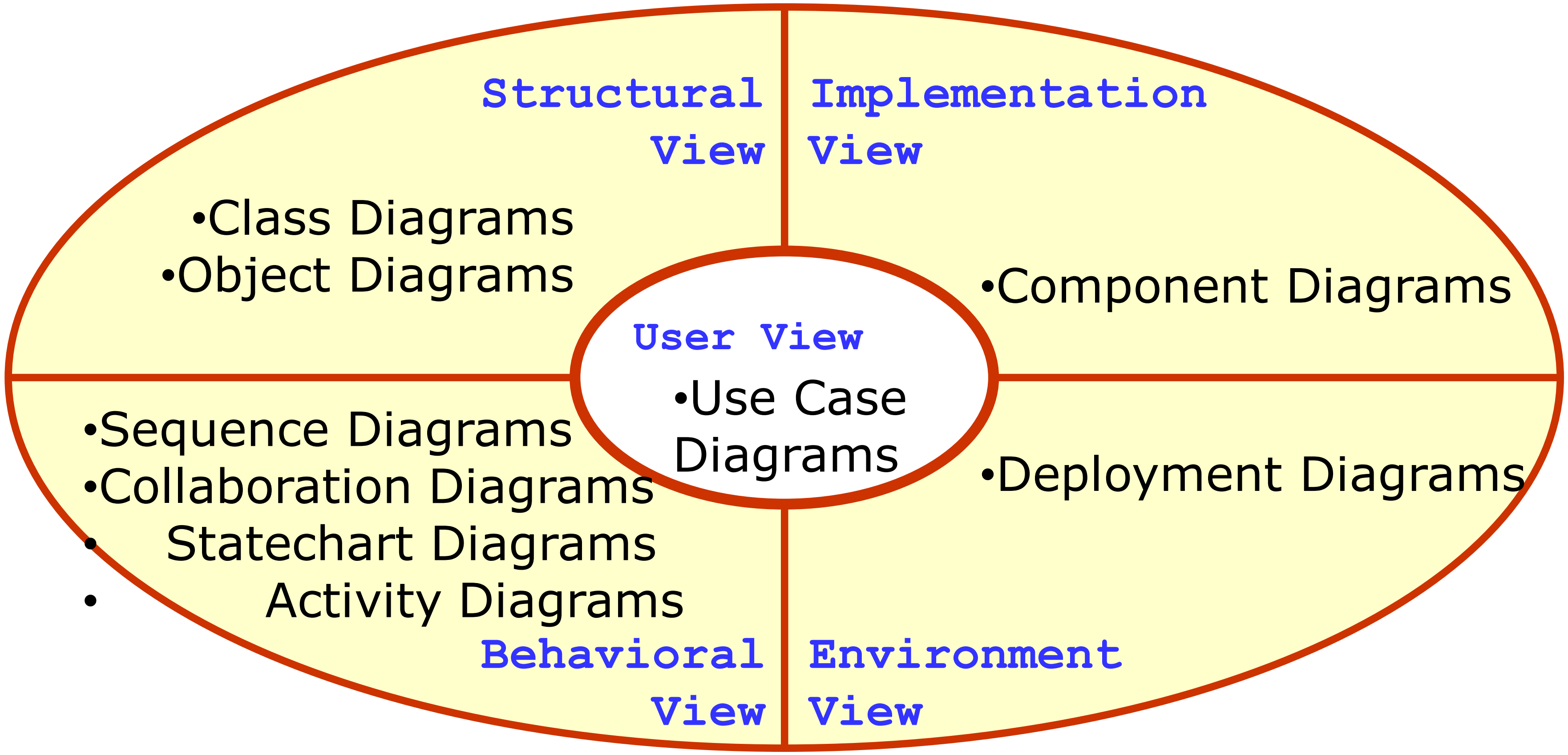
UML Diagrams

- Activity Diagram – *shows the flow from activity to activity; an "activity" is an ongoing non-atomic execution within a state machine.*
- Component Diagram – *shows the physical packaging of software in terms of components and the dependencies between them.*
- Deployment Diagram – *shows the configuration of the processing nodes at run-time and the components that live on them.*

UML Diagrams

- The UML includes specifications for nine key diagrams used to document various perspectives of a software solution from project inception to installation and maintenance.
- One way to organize the UML diagrams is by using views. A view is a collection of diagrams that describe a similar aspect of the project.

Dimensions


Structural View
•Class Diagrams
•Object Diagrams
Implementation View
•Component Diagrams
User View
•Use Case Diagrams
•Sequence Diagrams
•Collaboration Diagrams
• Statechart Diagrams
• Activity Diagrams
Behavioral View
•Deployment Diagrams
Environment View

Object Oriented Programming in C++

STRUCTURE OF C++ PROGRAM

- Include files
- Class declaration
- Class functions, definition
- **Main function program**

```
# include<iostream>
using namespace std
class person
{
char name[30];
int age;
public:
void getdata(void);
void display(void);
};
void person :: getdata ( void
)
{
cout<<"enter name";
cin>>name;
cout<<"enter age";
cin>>age;
}
```

```
void display()
{
cout<<"\n name:"<<name;
cout<<"\n age:"<<age;
}
int main( )
{
person p;
p.getdata();
p.display();
return(0);
}
```

CLASS

- Class is a group of objects that share common properties and relationships .In C++, a class is a new data type that contains member variables and member functions that operates on the variables. A class is defined with the keyword class. It allows the data to be hidden, if necessary from external use. When we defining a class, we are creating a new abstract data type that can be treated like any other built in data type.

Generally a class specification has two parts:-

a) Class declaration

b) Class function definition

The class declaration describes the type and scope of its members. The class function definition describes how the class functions are implemented.

Syntax:-

```
class class-name
{
private:
variable declarations;
function declaration ;
protected:
variable declarations;
function declaration ;

public:
variable declarations;
function declaration;
};
```

The members that have been declared as private can be accessed only from with in the class.
On the other hand , public members can be accessed from outside the class .
The data hiding is the key feature of oops. The use of keywords private is optional, by default the members of a class are private.
The variables declared inside the class are known as data members and the functions are known as members mid the functions.
 Only the member functions can have access to the private data members and private functions. However, the public members can be accessed from the outside the class. The binding of data and functions together into a single class type variable is referred to as encapsulation.

CREATING OBJECTS

Once a class has been declared we can create variables of that type by using the class name.

Example:

item x;

creates a variables x of type item. In C++, the class variables are known as objects. Therefore x is called an object of type item.

item x, y ,z also possible.

```
class item
{
-----------
-----------
-----------
}x ,y ,z;
```

would create the objects x ,y ,z of type item.

ACCESSING CLASS MEMBER

The private data of a class can be accessed only through the member functions of that class

Syntax:

object name.function-name(actual arguments);

```
class xyz
{
Int x;
Int y;
public:
int z;
};
---------
----------
xyz p;
p. x =0; error . x is private
p, z=10; ok ,z is public
```

DEFINING MEMBER FUNCTION

Member can be defined in two places

- Outside the class definition
- Inside the class function

OUTSIDE THE CLASS DEFINAT1ON

- Member function that are declared inside a class have to be defined separately outside the class.
- Syntax:

```
return type class-name::function-
   name(argument declaration )
{
function-body
}
```

INSIDE THE CLASS DEF1NATION:

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class .

Example:

```
class item
{
Intnumber; float cost;
public:
void getdata (int a ,float b);
void putdata(void)
{
cout<<number<<endl; cout<<cost<<endl;
}
};
```

END.
Wairagu G.R