UNIT-III

CLASS & OBJECT DIAGRAMS

Class Diagrams:

TERMS AND CONCEPTS

A class diagram is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs.

Common Properties:

A class diagram is just a special kind of diagram and shares the same common properties as do all other diagrams name and graphical content that are a projection into a model. What distinguishes a class diagram from other kinds of diagrams is its particular content.

Contents:

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships. Like all other diagrams, class diagrams may contain notes and constraints.

Class diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes you'll want to place instances in your class diagrams as well, especially when you want to visualize the (possibly dynamic) type of an instance.

Common Uses :

You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a system the services the system should provide to its end users.

When you model the static design view of a system, you'll typically use class diagrams in one of three ways.

1. To model the vocabulary of a system

Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

2. To model simple collaborations

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. For example, when you re modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

3. To model a logical database schema

Think of a schema as the blueprint for the conceptual design of a database. In many domains, you'll want to store persistent information in a relational database or in an object-oriented database. You can model schemas for these databases using class diagrams.

COMMON MODELING TECHNIQUES

1.Modeling Simple Collaborations
To model a collaboration,

• Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

• For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things as well.

• Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.

• Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these in to concrete attributes and operations.

2. Modeling a Logical Database Schema

To model a schema,

• Identify those classes in your model whose state must transcend the lifetime of their applications.

• Create a class diagram that contains these classes. You can define your own set of stereotypes and tagged values to address database-specific details.

• Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their multiplicities that relate these classes.

• Watch for common patterns that complicate physical database design, such as cyclic associations and one-to-one associations. Where necessary, create intermediate abstractions to simplify your logical structure.

• Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.

• Where possible, use tools to help you transform your logical design into a physical design.

Forward and Reverse Engineering:

Forward engineering is the process of transforming a model into code through a mapping to an implementation language. Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language. In fact, this is a major reason why you need models in addition to code. Structural features, such as collaborations, and behavioral features, such as interactions, can be visualized clearly in the UML, but not so clearly from raw code.

To forward engineer a class diagram,

• Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

• Depending on the semantics of the languages you choose, you may want to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent), or you can develop idioms that transform these richer features into

the implementation language (which makes the mapping more complex).

• Use tagged values to guide implementation choices in your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.

• Use tools to generate code.

```
public abstract class EventHandler {
EventHandler successor;
private Integer currentEventID;
private String source;
EventHandler() {}
public void handleRequest() {}
}
```

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models. At the same time, reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.

To reverse engineer a class diagram,

•Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

• Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered. It is unreasonable to expect to reverse engineer a single concise model from a large body of code. You need to select portion of the code and build the model from the bottom.

• Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other

neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

• Manually add design information to the model to express the intent of the design that is missing or hidden in the code.

Object Diagrams:

Object diagrams model the instances of things contained in class diagrams. An object diagram shows a set of objects and their relationships at a point in time.

TERMS AND CONCEPTS

An object diagram is a diagram that shows a set of objects and their relationships at a point in time. Graphically, an object diagram is a collection of vertices and arcs.

Common Properties

An object diagram is a special kind of diagram and shares the same common properties as all other diagrams that is, a name and graphical contents that are a projection into a model. What distinguishes an object diagram from all other kinds of diagrams is its particular content.

Contents

Object diagrams commonly contain

•Objects

•Links

Like all other diagrams, object diagrams may contain notes and constraints.

Sometimes you'll want to place classes in your object diagrams as well, especially when you want to visualize the classes behind each instance.

Common Uses

You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams, but from the perspective of real or prototypical instances. This view primarily supports the functional requirements of a systemthat is, the services the system should provide to its end users. Object diagrams let you model static data structures.

When you model the static design view or static process view of a system, you typically use object diagrams in one way:

To model object structures

Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time. An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram. You use object diagrams to visualize, specify, construct, and document the existence of certain instances in your system, together with their relationships to one another.

COMMON MODELING TECHNIQUES

Modeling Object Structures

To model an object structure,

•Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

• Create a collaboration to describe a mechanism.

• For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things as well.

•Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.

•Expose the state and attribute values of each such object, as necessary, to understand the scenario.

•Similarly, expose the links among these objects, representing instances of associations among them.

FORWARD AND REVERSE ENGINEERING

Forward engineering (the creation of code from a model) an object diagram is

theoretically possible but pragmatically of limited value. In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you cannot exactly instantiate these objects from the outside.

Reverse engineering (the creation of a model from code) an object diagram can be useful. In fact, while you are debugging your system, this is something that you or your tools will do all the time. For example, if you are chasing down a dangling link, you'll want to literally or mentally draw an object diagram of the affected objects to see where, at a given moment in time, an object's state or its relationship to other objects is broken. To reverse engineer an object diagram,

•Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.

• Using a tool or simply walking through a scenario, stop execution at a certain moment in time.

• Identify the set of interesting objects that collaborate in that context and render them in an object diagram.

• As necessary to understand their semantics, expose these object's states.

• As necessary to understand their semantics, identify the links that exist among these objects.