UNIT-2

Introduction to UML: Importance of modeling, principles of modeling, object oriented modeling, conceptual model of the UML, Architecture, Software Development Life Cycle. **Structural Modeling**: Classes, Relationships, common Mechanisms, and diagrams. **Case Study:** Control System: Traffic Management.

1.1.Introduction to UML:

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

UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

OMG is continuously making efforts to create a truly industry standard.

- UML stands for **Unified Modelling Language**.
- UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- UML is a pictorial language used to make software blueprints.
- UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
- Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.
- The UML is process independent.
- UML is not a programming language, it is rather a visual language.
- UML diagrams are not only made for developers but also for business users, common people, and anybody interested to understand the system.

☛ It helps in designing and characterizing, especially those software systems that incorporate the concept of Object orientation. It describes the working of both the software and hardware systems.

☛ The UML was developed in 1994-95 by Grady Booch, Ivar Jacobson, and James Rumbaugh at the Rational Software. In 1997, it got adopted as a standard by the Object Management Group (OMG).

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

The artifacts of a software-intensive system.

1.Visualizing: The UML Is a Language for Visualizing

- Communicating conceptual models to others is prone to error unless everyone involved speaks thesame language.
- There are things about a software system you can't understand unless you build models.

2.Specifying: UML is a language for Specifying

- Specifying means building models that are precise, unambiguous and complete.
- The UML addresses the specification of all the important analysis, design and implementation decisions that must be made in developing and deploying a software system.

3.Constructing: UML is a language for Constructing

- The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages, such as JAVA, C++ or Visual Basic or even to Tables of database.

4.Documenting: The UML Is a Language for Documenting

- The UML addresses documentation of system architecture, requirements, tests, project planning, and release management.

1.2 Importance of modeling :

Model: UML is a modeling language

- A model is "a complete description of a system from a particular perspective." A model is a simplification of reality.
- A model provides the blueprints of a system.
- Model is important and mandatory.
- A model may be structural, emphasizing the organization of the system, or itmay be behavioral, emphasizing the dynamics of the system.

Why do we model?:

- Modeling is a central part of all the activities that lead to the development ofgood software.
- A model is a simplification at some level of abstraction .
- We build models to better understand the systems we are developing:
  - To help us visualize
  - To specify structure or behavioral
  - To provide template for building system
  - To document decisions we have made
- Analyse the problem-domain and Design the solution
  - simplify reality
  - capture requirements
  - visualize the system in its entirety
  - specify the structure and/or behaviour of the system

- Through Modeling achieves four aims:

- Helps you to **visualize a system** as you want it to be.
- Permits you to **specify the structure or behavior of a system.**
- Gives you a **template that guides you in constructing** a system.
- **Documents the decisions** you have made.

- You build models of complex systems because you cannot comprehend such asystem in its entirety.
- You build models to better understand the system you are developing.

1.3.principles of modelling:

UML is basically a modeling language; hence its principles will also be related to modeling concepts. Here are few basic principal of UML.

First: "The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped"

In other words , choose your models well. The right models will brilliantly illuminate the most wicked development problems. The wrong models will mislead you, causing you to focus on irrelevant issues.

Second: " Every model may be expressed at different levels of precision ".

Best approach to a given problem results in a best model. If the problem is complex mechanized level of approach & if the problem is simple decent approach is followed.

Third: "The best models are connected to reality."

The model built should have strong resemblance with the system.

Fourth: " No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models."

- If you constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you will need floor plans, elevations , electical plans, heating plans, and plumbing plans.

1.4.object oriented modelling:

In software , there are several ways to approaches a model. The two most common ways are

1. Algorithmic perspective
2. Object-Oriented perspective

1. Algorithmic perspective:

In this approach, the main building blocks of all software is the procedure or function .This view leads developers to focus on issues of control and decomposition of larger algorithms into smaller ones.

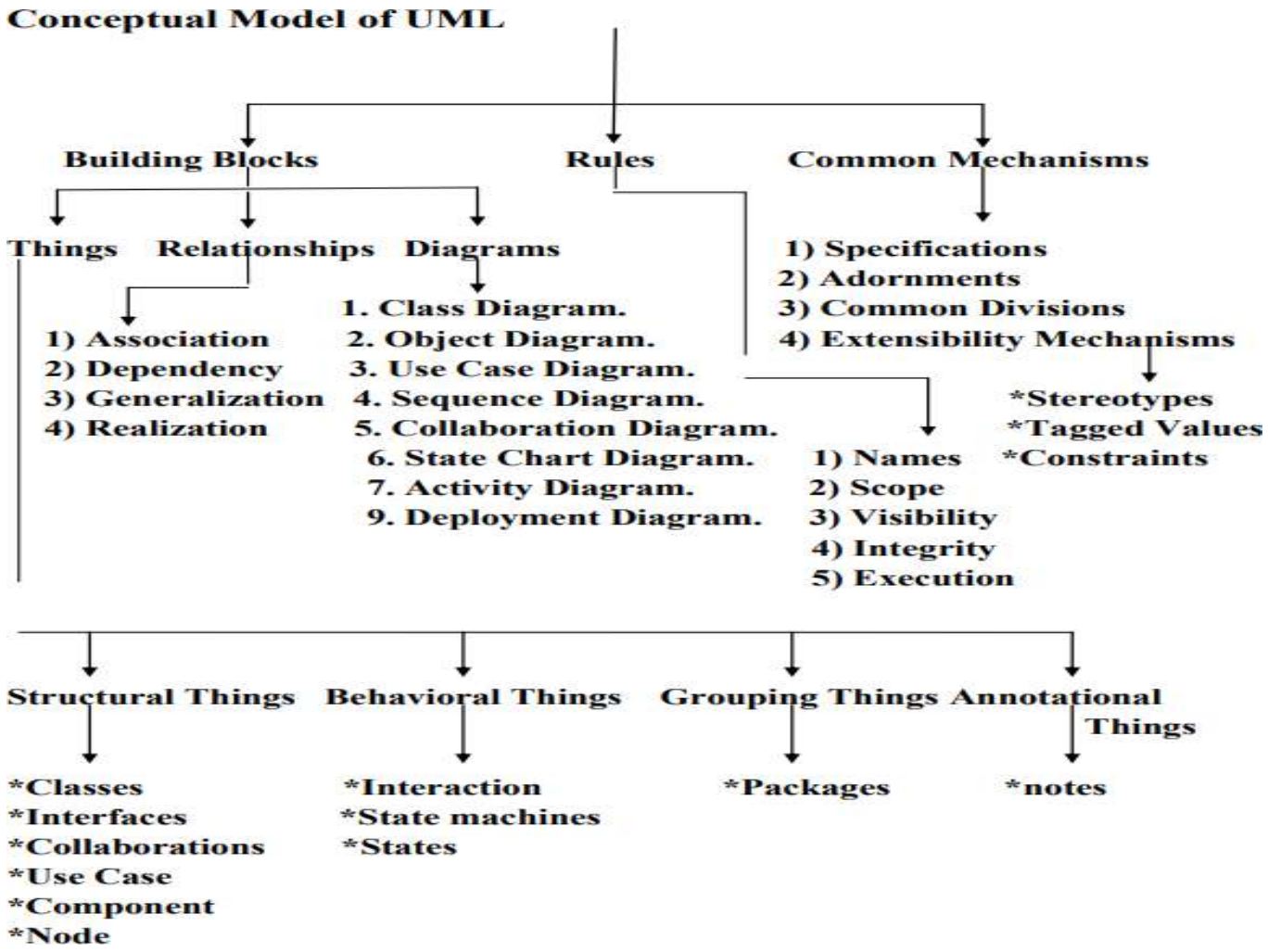
2.Object-Oriented perspective:

In this approach, the main building blocks of all software is the object or class. Simply put, an object is a thing. A class is a description of a set of common objects. Every object has identity, state and behavior.

For example, consider a simple a three-tier -architecture for a billing system, involving a user interface ,middleware, and a data base. In the user interface, you will find concrete objects, such as buttons, menus, and dialog boxes. In the database, you will find concrete objects ,such as tables. In the middle layer ,you will find objects such as transitions and business rules.

1.5.conceptual model of the UML:

- To understand the UML, a conceptual model of the language is need to formand it requires the three major elements:
- Building blocks of the UML.
- Rules that dictate how these building blocks may be put together.
- Common mechanisms that apply throughout the UML.



1.5.1. UML-Building Blocks:

UML is composed of three main building blocks, i.e., things, relationships, and diagrams. Building blocks generate one complete UML model diagram by rotating around several different blocks. It plays an essential role in developing UML diagrams.

The basic UML building blocks are enlisted below:

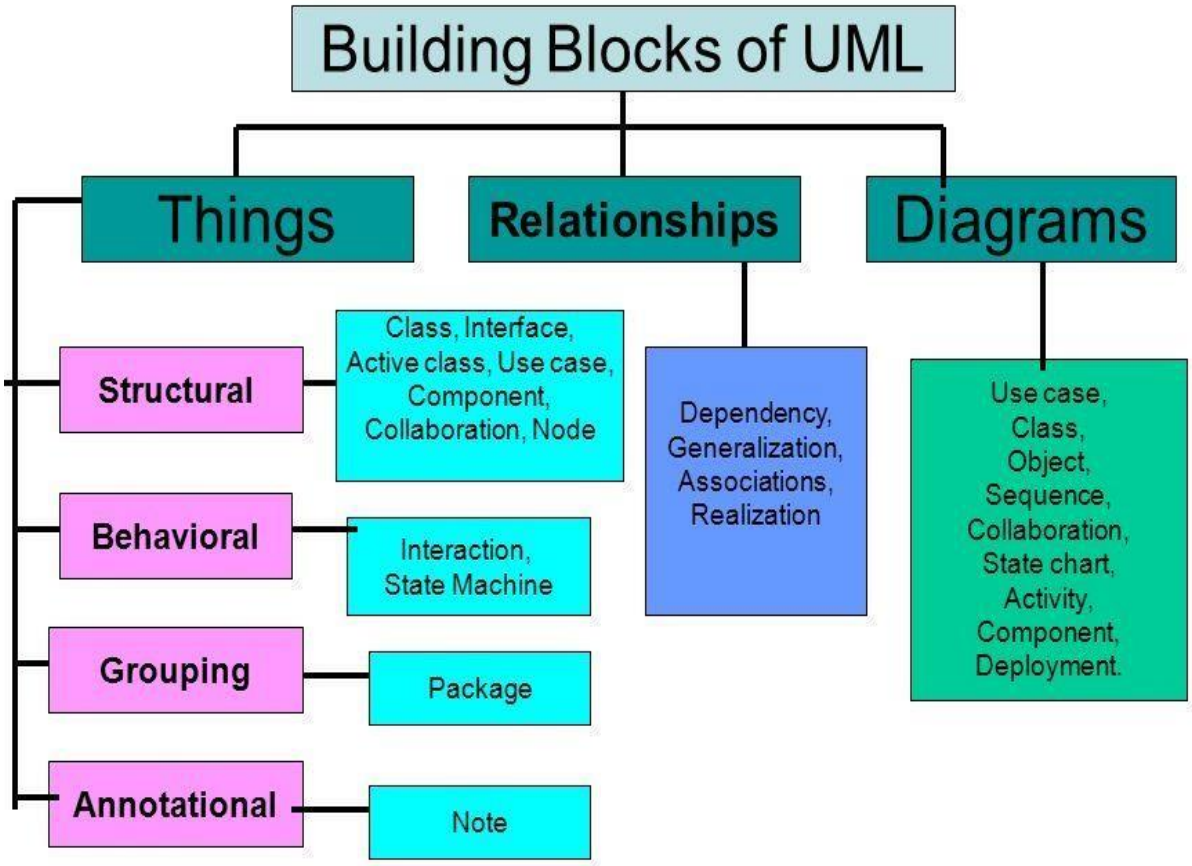
1. Things
2. Relationships
3. Diagrams

- The UML consists of three kinds of building blocks:

a) ***Things-*** These are the **abstractions** that are first-class citizens in a model.

b) ***Relationships-***Tie the above things together.

c) ***Diagrams-***Group interesting collections of things.



1.5.1.1 Things in the UML:

- There are 4 kinds of things in the UML
  1. Structural Things
  2. Behavioral Things
  3. Grouping Things
  4. Annotational Things

These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

1. Structural Things

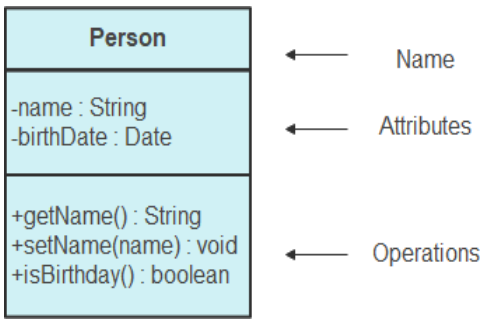
- Structural things are the nouns of UML models. These are mostly static parts of a model, representing elements that are either conceptual or physical.
- Collectively, the structural things are called classifiers.

There are 7 kinds of Structural things:

1. Class
2. Interface
3. Collaboration
4. Use case
5. Active class
6. Component
7. Node

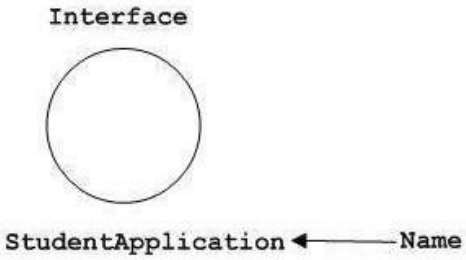
1.Class:-

- A class is a description of a set of objects that share the same attributes, operations, relationships and semantics.
- A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations.



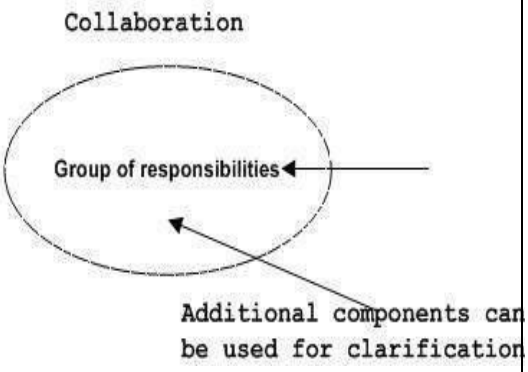
2.Interface:

- An interface is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element.
- An interface might represent the complete behavior of a class or component or only a part of that behavior.
- Interface is represented by a circle as shown in the following figure. It has a name which is generally written below the circle.



3.Collaboration:

- A collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements.
- Collaborations have structural, as well as behavioral, dimensions. A given class or object might participate in several collaborations.
- Graphically, a collaboration is rendered as

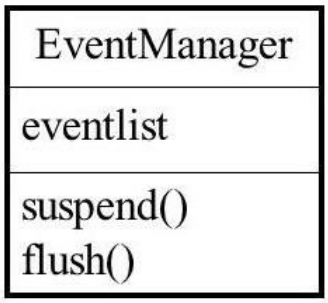


an ellipse with dashed lines, sometimes including only its name.

4.Use case:

- A use case is a description of sequences of actions that a system performs that yield observable results of value to a particular actor.
- A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name.

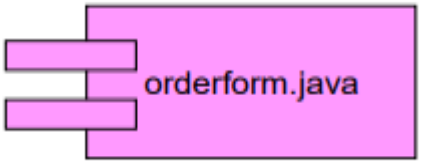
5.Active class:



- An active class is a class whose objects own one or more processes or threads and therefore can initiate control activity.
- An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements.
- Graphically, an active class is rendered as a class with dark line; it usually includes its name, attributes, and operations.

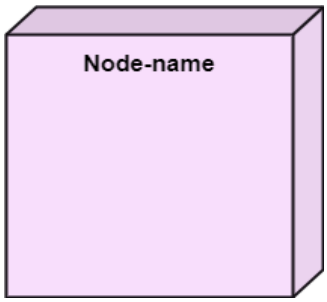
6.Component:

- A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.



Component

7.Node:



A physical element that exists at run time.

2. Behavioral Things

They are the verbs that encompass the dynamic parts of a model. It depicts the behavior of a system.

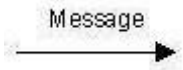
- There are two types of behavioral things

1. Interaction
2. State machine.

1.Interaction – Interaction is defined as a behavior that consists of a group of

messages exchanged among elements to accomplish a specific task.

Graphically, a message is rendered as a directed line.



2.State machine − State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

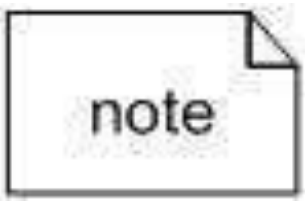
3.Grouping Things - Grouping things can be defined as a mechanism to group elements of a UML model
together. There is only one grouping thing available −

- **Package** − Package is the only one grouping thing available for gathering structural and behavioral things.

Structural things, behavioral things and even other grouping things that may be placed in a package.

Package

4. Annotational things : Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements.



- **Note** - It is the only one Annotational thing available. A note is used to render as a rectangle with a dog-eared corner.
- Annotational Things are the explanatory parts of the UML models.

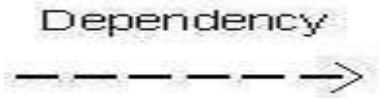
Relationships in the UML:

There are four kinds of relationships in the UML.

1. Dependency
2. Association
3. Generalization
4. Realization

- These relationships are the basic relational building blocks of the UML. Theseare used to write well-formed models.

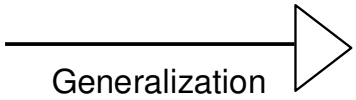
1. Dependency: First, a dependency is a semantic relationship between two model elements in which a change to one element (the independent one) may affect the semantics of the other element (the dependent one). Graphically, a



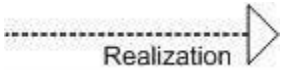
dependency is rendered as a dashed line, possibly directed.

2. Association : Second, an association is a structural relationship among classes that describes a set of links, a link being a connection among objects that are instances of the classes. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed

3. Generalization : Third, a generalization is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent). The child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent



4. Realization : Fourth, a realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two



places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them.

1.5.1.2 Diagrams in the UML:

- ✓ A diagram is the graphical presentation of a set of elements, most often renderedas a connected graph of vertices (things) and paths (relationships).

✓ You draw diagrams to visualize a system from different perspectives, so adiagram is a projection into a system.

UML includes nine kinds of diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Activity diagram
7. Statechart diagram
8. Component diagram
9. Deployment diagram

1.Class diagram:

- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- These diagrams are the most common diagram found in modeling object-oriented systems.
- Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

2.Object diagram:

- An object diagram shows a set of objects and their relationships.
- Object diagrams represent static snapshots of instances of the things found in class diagrams.
- These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

3. Use case diagram:

- A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships.
- Use case diagrams address the static use case view of a system.
- These diagrams are especially important in organizing and modeling the behaviors of a system.

4.Sequence diagram:

- Both sequence diagrams and Collaboration diagrams are kinds of interaction diagrams.
- An interaction diagram shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them.

- Interaction diagrams address the dynamic view of a system.
- A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages

5.Collaboration diagram:

- A Collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages.
- Sequence diagrams and Collaboration diagrams represent similar basic concepts, but each diagram emphasizes a different view of the concepts.
- Sequence diagrams emphasize temporal ordering, and Collaboration diagrams emphasize the data structure through which messages flow.

6.Activity diagram:

- An activity diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation.
- Activity diagrams address the dynamic view of a system.
- They are especially important in modeling the function of a system and emphasize the flow of control among objects.

7.Statechart diagram:

- A state diagram shows a state machine, consisting of states, transitions, events, and activities.
- A state diagrams shows the dynamic view of an object.
- They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of anobject, which is especially useful in modeling reactive systems

8.Component diagram:

- A component diagram is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors.
- Component diagrams address the static design implementation view of a system. They are important for building large systems from smaller parts.

9.Deployment diagram:

- A deployment diagram shows the configuration of run-time processing nodes and the components that live on them.
- Deployment diagrams address the static deployment view of an architecture. A node typically hosts one or more artifacts.

1.5.2. Rules of the UML :

- ✓ The UML's building blocks can't simply be thrown together in a random fashion.
- ✓ Like any language, the UML has a number of rules that specify what a well-formed model should look like.
- ✓ A well-formed model is one that is semantically self-consistent and in harmony with all its related models

The UML has semantic rules for:-

- Names- What you can call things, relationships and diagrams.
- Scope- The context that gives specific meaning to a name.
- Visibility- How those names can be seen and used by others.
- Integrity- How things properly and consistently relate to one another.
- Execution- What it means to run or simulate a dynamic model.

- Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times.
- For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

Elided- Certain elements are hidden to simplify the view.

Incomplete - Certain elements may be missing.

Inconsistent - The integrity of the model is not guaranteed.

- These less-than-well-formed models are unavoidable as the details of a system unfold and churn during the software development life cycle.

The rules of the UML encourage you but do not force you to address the most important analysis, design, and implementation questions that push such models to become well-formed over time.

1.5.3 Common Mechanisms

- The 4 common mechanisms that apply consistently throughout the language.

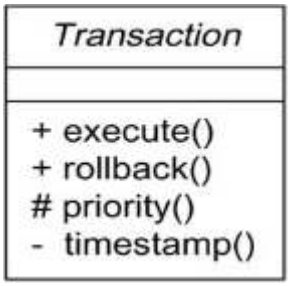
1. Specifications
2. Adornments
3. Common Divisions
4. Extensibility mechanisms

1.Specifications:-

- The UML is more than just a graphical language. Rather, behind every part of graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block.
- For example, behind a class icon is a specification that provides the full set of attributes, operations and behaviors.
- You use the UML's graphical notation to visualize a system; you use the UML's specification to state the system details.

2. Adornments:-

- Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element.
- For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems.
- The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations.



- A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations.
- Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation.
- For example a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.

3. Common Divisions:

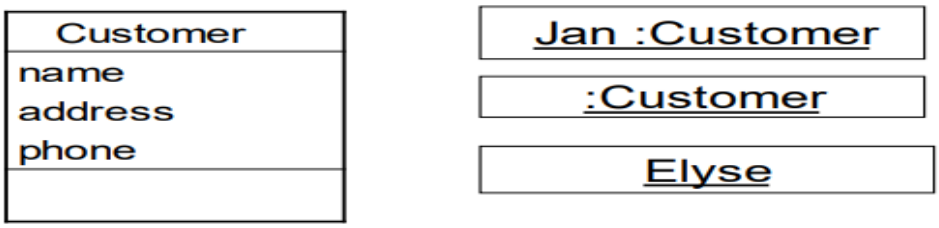
In modeling object-oriented systems, the world often gets divided in several ways.

1.Abstraction vs. manifestation

First, there is the division of class and object. A class is an abstraction; an objectis one concrete manifestation of that abstraction

2.Class vs. object

- Most UML building blocks have this kind of class/object distinction.

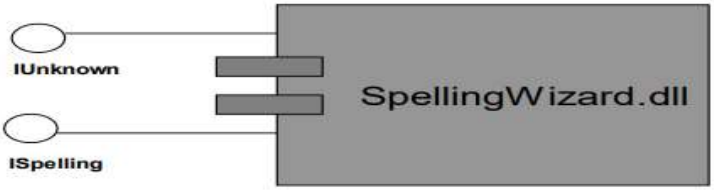


Classes and Objects

3.Interface vs. implementation

Second, there is the separation of interface and implementation.

- An interface declares a contract, and an implementation represents oneconcrete realization of that contract.
- An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics.



Interfaces and Implementations

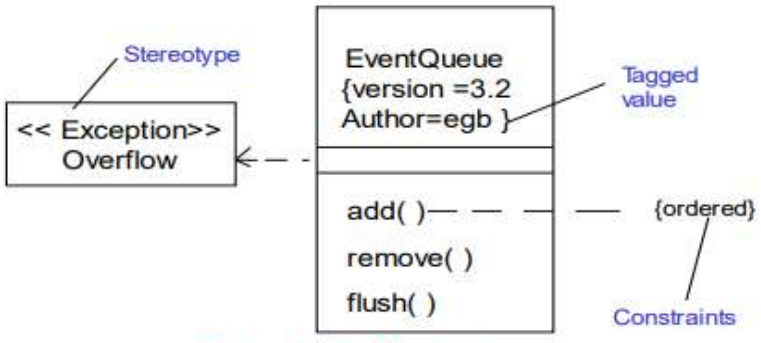
4. Extensibility Mechanisms:

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time.

The UML's extensibility mechanisms include:

- Stereotypes
- Tagged values
- Constraints

1.Stereotypes: Extend the vocabulary of the UML by creating new model elements derived from existing ones but that have specific properties suitable for your domain/problem.



Extensibility Mechanisms

2.Tagged values:-Properties for specifying key-value pairs of model elements, where keywords are attributes.

Extend the properties of a UML building block, allowing you to create new information in that elements specification.

3.Constraints:-

- Properties for specifying semantics or conditions that must be maintained as true for model elements.
- Extend the semantics of a UML building block, allowing you to add new rules, or modify existing ones.

Example :- you might want to constrain the Event Queue class so that all additions are done in order.

1.6 Architecture/ Modeling a System's Architecture:

- Any real world system is used by different users. The users can be developers,testers, business people, analysts and many more.
- So before designing a system the architecture is made with differentperspectives in mind. T
- he most important part is to visualize the system from different viewer.s perspective.
- The better we understand the better we make the system.

Architecture is the set of significant decisions about

- The organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified in the collaborations among those elements
- The composition of these structural and behavioral elements into progressivelylarger subsystems

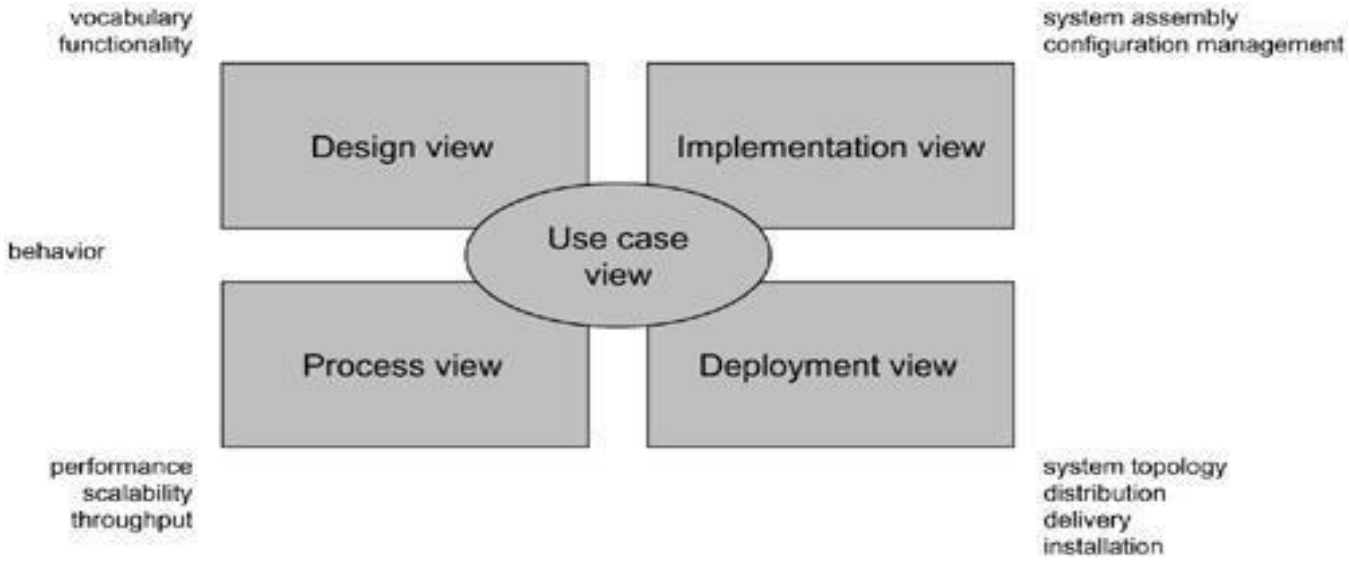
❖ Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

UML plays an important role in defining different perspectives of a

system. These perspectives are:

- Usecase View

- Design View
- Process View
- Implementation View
- Deployment View



The center is the Use Case view which connects all these four.

- ✓ **A Use case**: represents the functionality of the system, it Specify the **shape of the system's architecture** .So the other perspectives are connected with use case.
- ✓ **Design** of a system consists of classes, interfaces and collaboration. UML provides class diagram, object diagram to support this.
  - This view primarily **supports the functional requirements** of the system, meaning the **services that the system should provide to its end users.**
- ✓ **Implementation** defines the components assembled together to make a complete physical system. UML component diagram is used to support implementation perspective
  - This view primarily addresses the **configuration management of the system's releases**
- ✓ **Process** defines the flow of the system. So the same elements as used in Designare also used to support this perspective.
  - This view primarily addresses the performance, scalability, and throughput of the system.
- ✓ **Deployment** represents the physical nodes of the system that forms the hardware. UML deployment diagram is used to support this perspective.
  - This view primarily addresses the **distribution, delivery, and installation of the parts that make up the physical system.**

1.7.Software Development Life Cycle:

The UML is largely process-independent, meaning that it is not tied to any particular software development life cycle.

However, to get the most benefit from the UML, you should consider a process that is

- Use case driven
- Architecture-centric
- Iterative and incremental

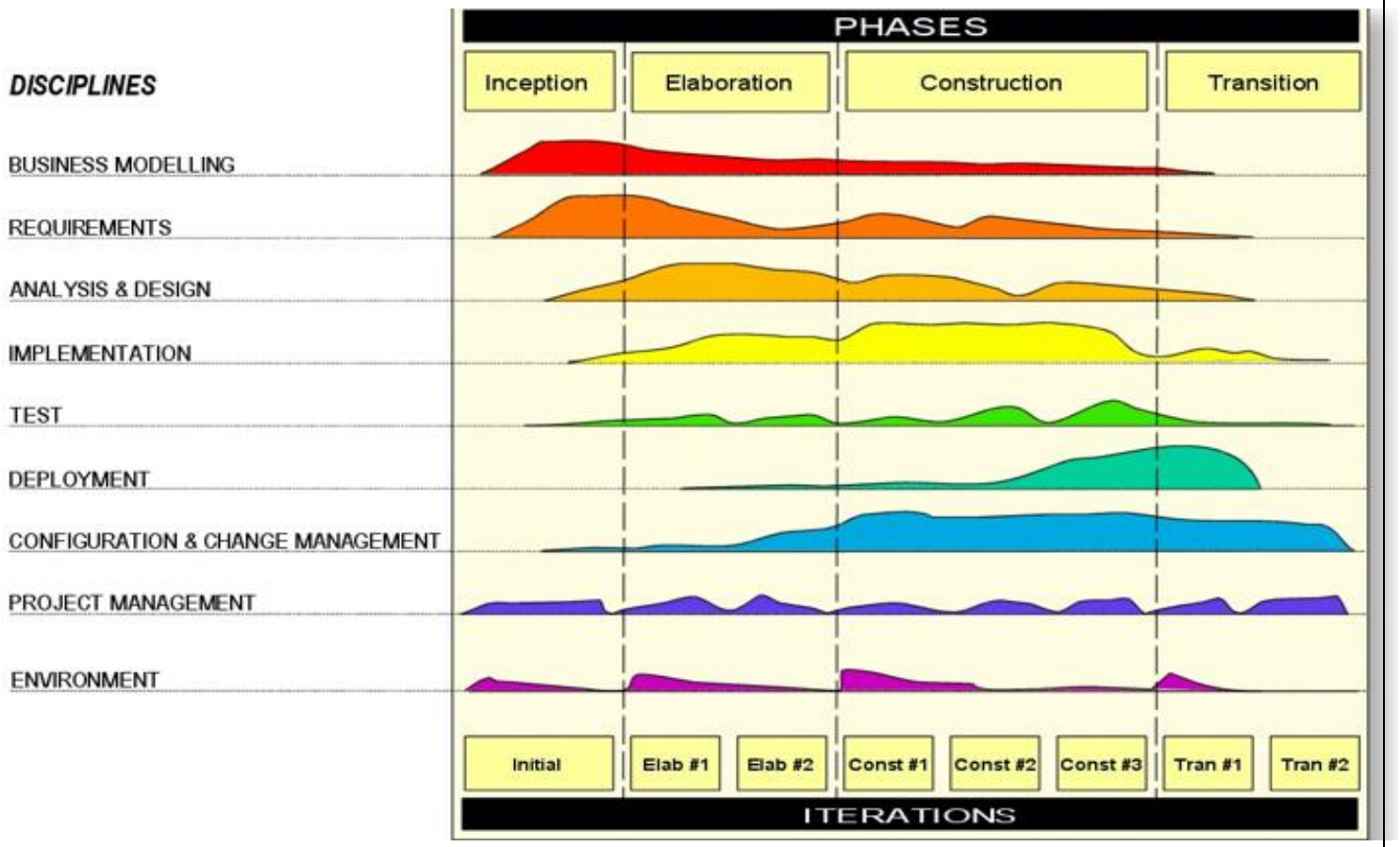
- ***Use case driven :*** means that use cases are used as a **primary artifact** for establishing the desired behavior of the system, for **verifying and validating the system's architecture, testing, communicating** among the stakeholders of the project.
- ***Architecture-centric*** means that a system's architecture is used as a **primary artifact** for **conceptualizing, constructing, managing, and evolving** the system under development.
- An ***iterative process*** is one that involves **managing a stream of executable releases**. An **iterative and incremental process is *risk-driven,*** meaning that each new release is focused on attacking and reducing the most significant risks to the success of the project.
  - This use case driven, architecture-centric, and iterative/incremental process can be broken into four **phases**. A ***phase*** is the span of time between two major milestones of the process.
  - There are four phases in the software development life cycle: inception, elaboration, construction, and transition

Inception is the first phase of the process, when the seed idea for the development is brought up to the point of being at least internally sufficiently well-founded to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are prioritized and baselined.

Construction is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community.

Transition is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated and features that didn't make an earlier release are added.



1.8.Basic Structural Modeling:

Contents:

1. Classes 2.Relationships

3.Common Mechanisms 4.Diagrams

Structural Modeling:

Structural modeling captures the static features of a system. They consist of the following −

- Classes diagrams
- Objects diagrams
- Component diagram
- Deployment diagrams
- Component diagram

Class diagram is the most widely used structural diagram.

1. Classes:

Terms and Concepts:

A **class** is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered

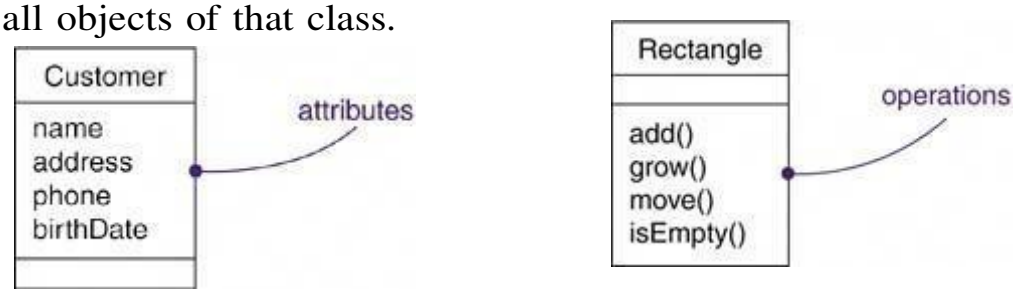
as arectangle.

Names

- Every class must have a name that distinguishes it from other classes.
- A name is a textual string. That name alone is known as a simple name; a qualified name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name.

Attributes:

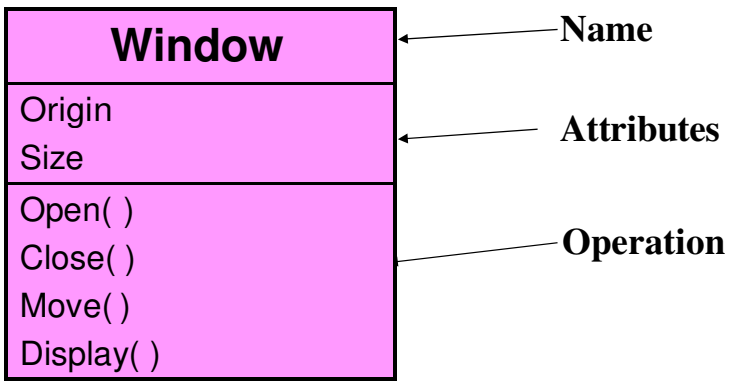
- An *attribute* is a named property of a class that describes a range of values that instances of the property may hold.
- A class may have any number of attributes or no attributes at all.
- An attribute represents some property of the thing you are modeling that is shared by all objects of that class.



Operations

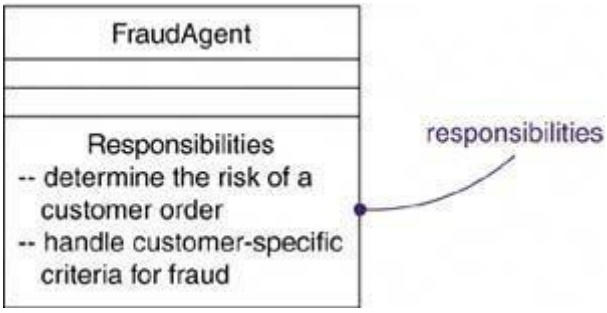
- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.
- an operation is an abstraction of something you can do to an object that is shared by all objects of that class.
- A class may have any number of operations or no operations at all.

class



Responsibilities

A *responsibility* is a contract or an obligation of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior.



2.Relationships

Terms and Concepts

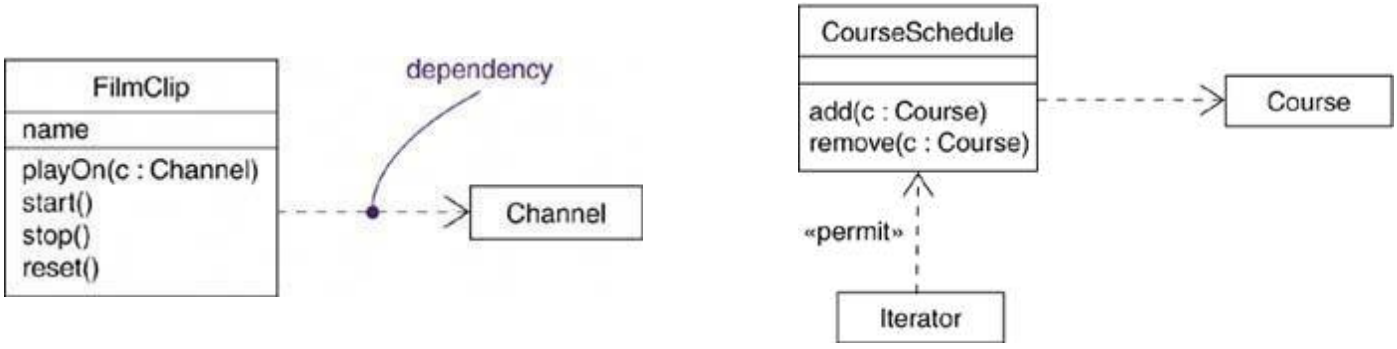
- A *relationship* is a connection among things.
- In object-oriented modeling, the three most important relationships are **dependencies, generalizations, and associations**.
- Graphically, a relationship is rendered as a path, with different kinds oflines used to distinguish the kinds of relationships.

Dependencies:

A *dependency* is a relationship that states that one thing (for example, class Window) uses the information and services of another thing (for example, classEvent), but not necessarily the reverse.

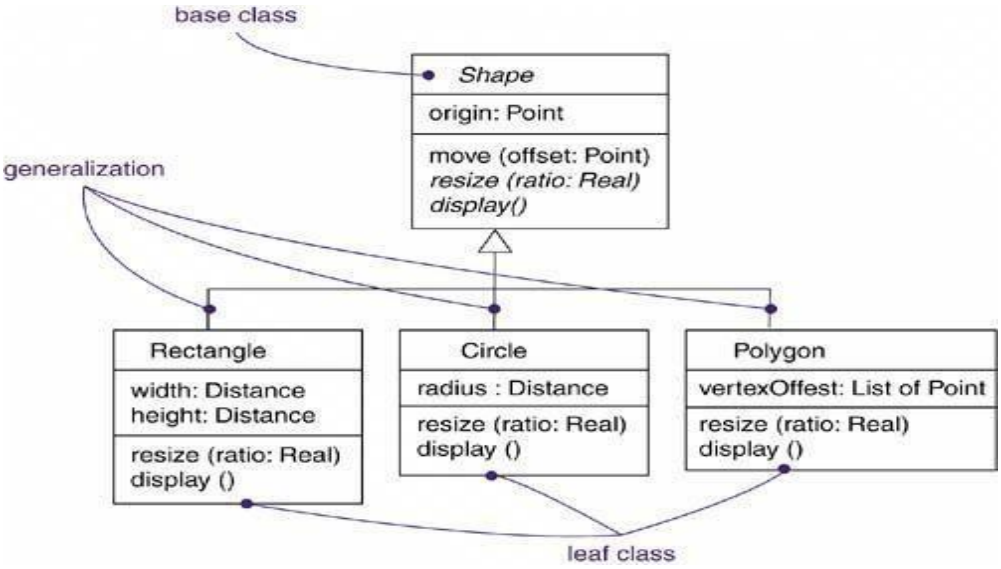
Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on.

Choose dependencies when you want to show one thing using another.



Generalizations:

A *generalization* is a relationship between a general kind of thing (called the superclass or parent) and a more specific kind of thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship.



Associations:

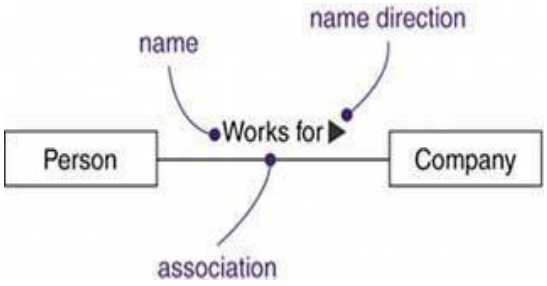
- An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another.
- Given an association connecting two classes, you can relate objects of one class to objects of the other class.
- An association that connects exactly two classes is called a binary association..
- Beyond this basic form, there are four adornments that apply to associations.

Name:

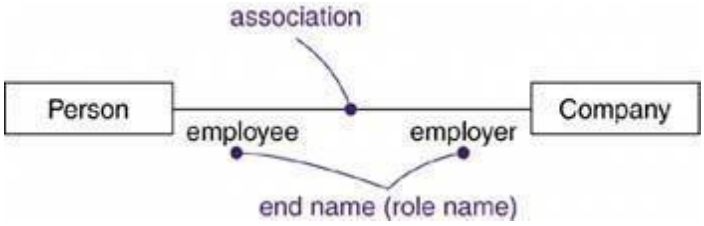
An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name.

Role:

- When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association.



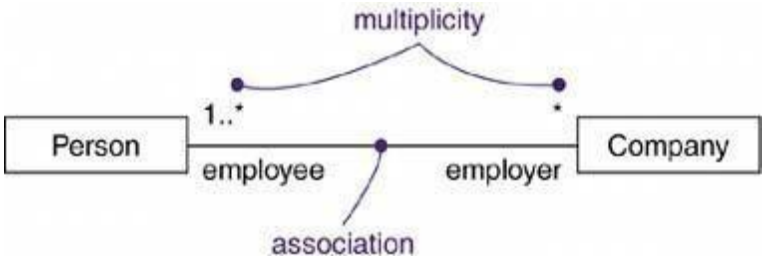
- You can explicitly name the role a class plays in an association.



Multiplicity:

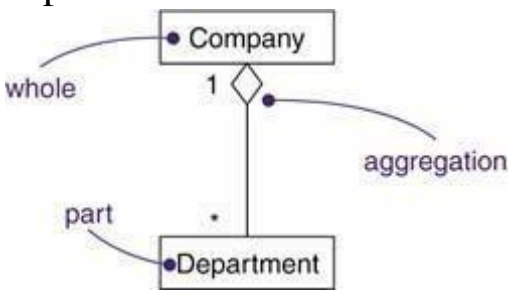
- An association represents a structural relationship among objects.
- In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association.
- This "how many" is called the multiplicity of an association's role.
- It represents a range of integers specifying the possible size of the set of related objects.

The number of objects must be in the given range. You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can give an integer range (such as 2..5). You can even state an exact number (for example, 3, which is equivalent to 3..3).



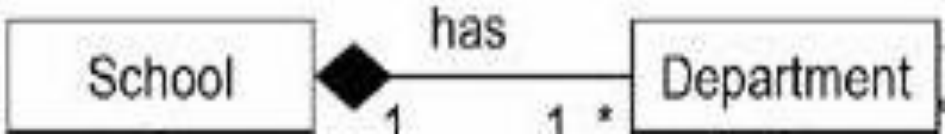
Aggregation:

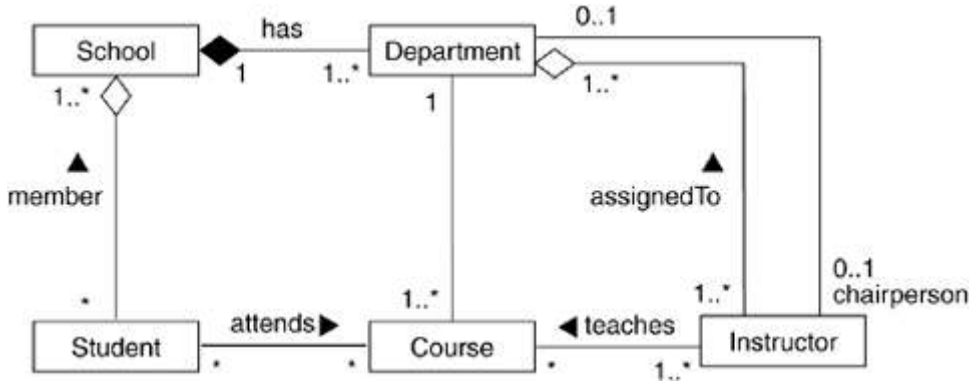
A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other.
Sometimes you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts"). This kind of relationship is called aggregation, which represents a "has-a" relationship



<u>Composition:-</u>

- Composition is a special form of aggregation within which the parts are inseparable from the whole.



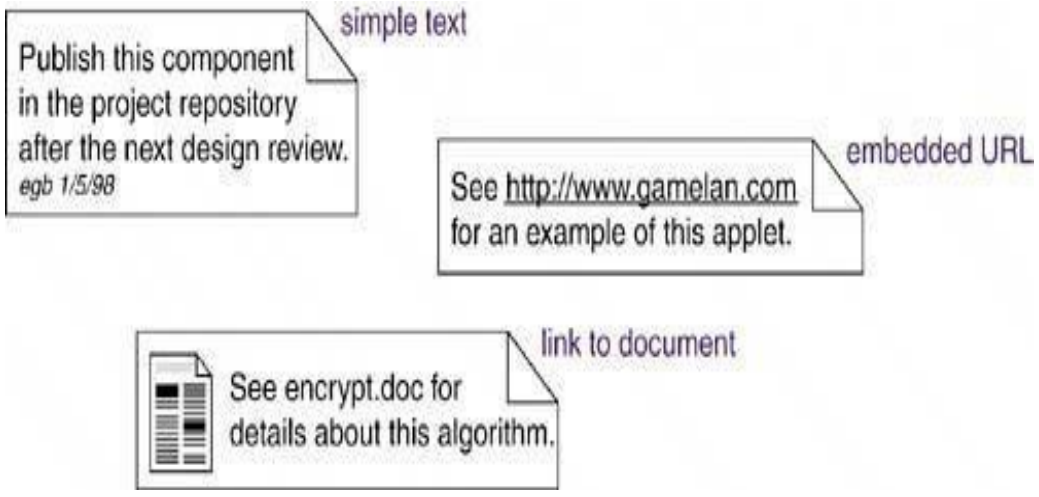


Modeling Structural Relationships

3. Common Mechanisms:

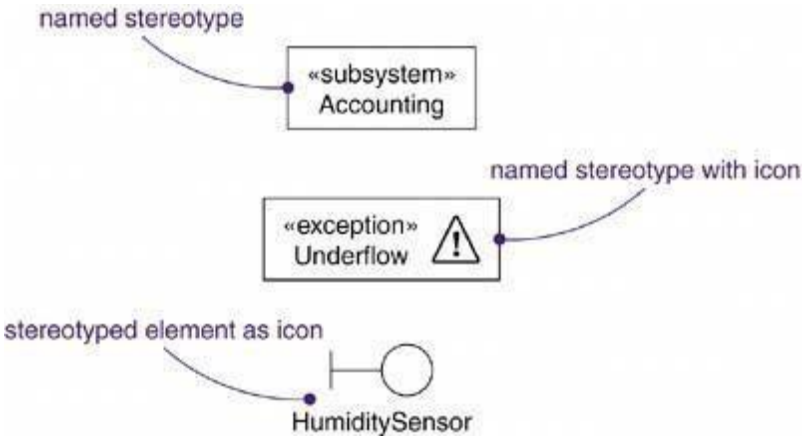
Terms and Concepts:

- A ***note*** is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.

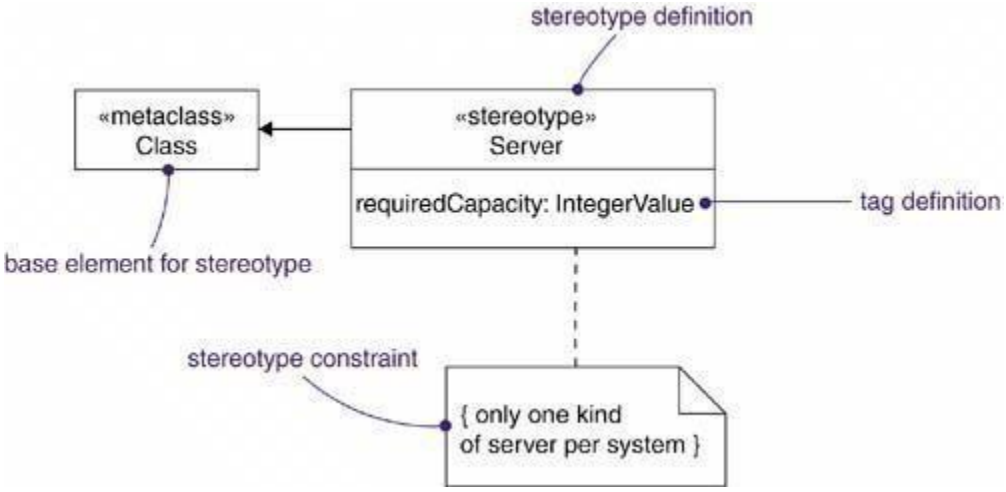


☞

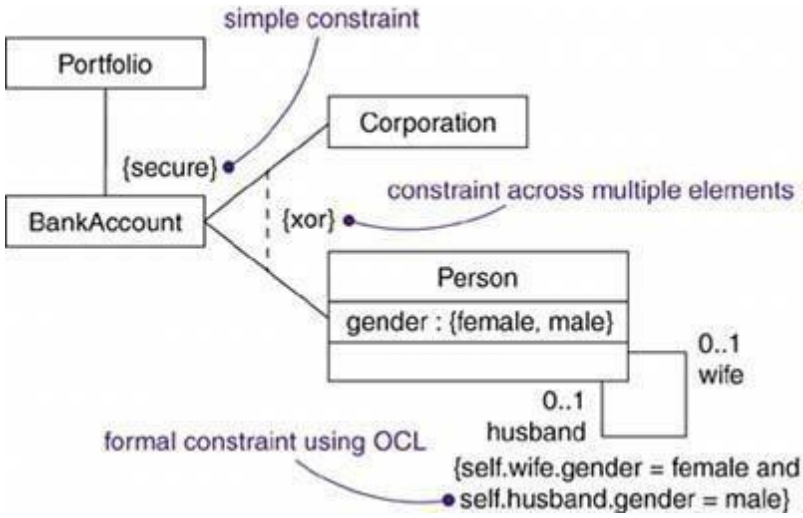
stereotype is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem. Graphically, a stereotype is rendered as a name enclosed by guillemets (French quotation marks of the form « »), placed above the name of another element.



- Optionally the stereotyped element may be rendered by using a new icon associated with that stereotype.
- A ***tagged value*** is a property of a stereotype, allowing you to create new information in an element bearing that stereotype. Graphically, a tagged value is rendered as a string of the form name = value within a note attachedto the object.
- classes have names, attributes, and operations; associations have names and two or more ends, each with its own properties; and so on. With stereotypes, you can add new things to the UML; with tagged values, you can add new properties to a stereotype.

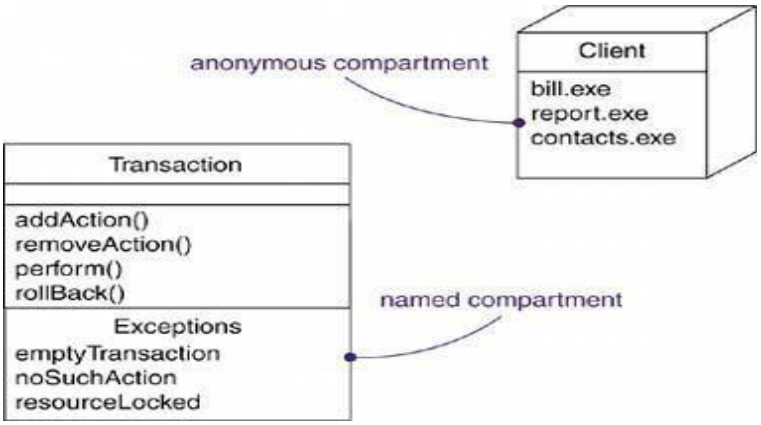


☞ A ***constraint*** is a textual specification of the semantics of a UML element, allowing you to add new rules or to modify existing ones. Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships. As an alternative, you can render a constraint in a note.



Other Adornments:

Adornments are textual or graphical items that are added to an element's basicnotation and are used to visualize details from the element's specification



4.Diagrams:

Terms and Concepts

A ***system*** is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints.

A ***subsystem*** is a grouping of elements, some of which constitute a specification ofthe behavior offered by the other contained elements.

A ***model*** is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system. In the context of architecture,

A ***view*** is a projection into the organization and structure of a system's model, focused on one aspect of that system.

A ***diagram*** is the graphical presentation of a set of elements, most often renderedas a connected graph of vertices (things) and arcs (relationships).

- The static parts of a system using one of the following diagrams:
    1. Class diagram
    2. Component diagram
    3. Object diagram
    4. Deployment diagram

- You'll often use five additional diagrams to view the dynamic parts of a system:
    1. Use case diagram
    2. Sequence diagram
    3. Collaboration diagram
    4. State diagram
    5. Activity diagram

Structural Diagrams:

- The UML's structural diagrams exist to visualize, specify, construct, anddocument the static aspects of a system.
- The UML's structural diagrams are roughly organized around the majorgroups of things you'll find when modeling a system.

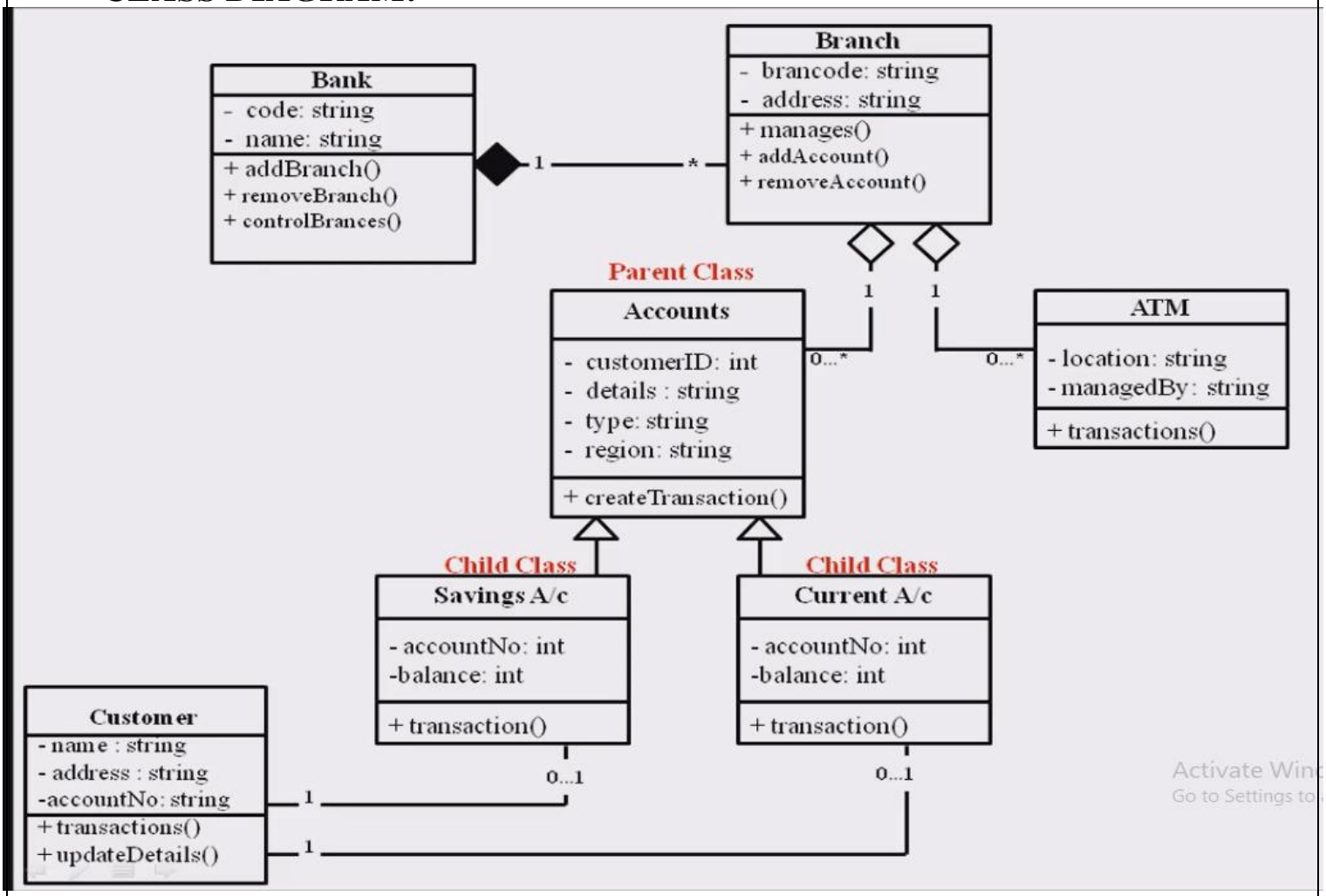
1.Class diagram	**Classes, interfaces, and collaborations**
2.Component diagram	**Components**
3.Object diagram	**Objects**
4.Deployment diagram	**Nodes**

Behavioral Diagrams:

- The UML's behavioral diagrams are used to visualize, specify, construct, anddocument the dynamic aspects of a system.
- The UML's behavioral diagrams are roughly organized around the majorways you can model the dynamics of a system.

1.Use case diagram	**Organizes the behaviors of the system**
2.Sequence diagram	**Focuses on the time ordering of messages**
3.Collaboration diagram	**Focuses on the structural organization ofobjects that send and receive messages**
4.State diagram	**Focuses on the changing state of a systemdriven by events**
5.Activity diagram	**Focuses on the flow of control fromactivity to activity**

EXAMPLE
CLASS DIAGRAM:



Case Study: Control System: Traffic Management.

1. Building systems so that their implementation is small is certainly an honorable task, but reality tells us that certain large problems demand large implementations.
2. To reduce development risk, projects usually involve a central organization that is responsible
   for systems architecture and integration; the remaining work may be subcontracted to other companies or to other in-house organizations.
3. Thus, the development team as a whole never assembles as one; it is typically distributed over space and—because of the personnel turnover common in large projects—over time.
4. Developers who are content with writing small, stand-alone, single-user, window-based tools may find the problems associated with building massive applications staggering.
5. Successfully automating such systems not only addresses the very real problems at hand but also leads to a number of tangible and intangible benefits, such as lower operational costs, greater safety, and increased functionality. Of course, the operative word here is successfully.
6. Building complex systems is plain hard work and requires the application of the best engineering practices we know, along with the creative insight of a few great designers.

1.Inception

2.Requirements for the Train Traffic Management System

3.Determining System Use Cases
4.Elobration
1.Analyzing System Functionality
2.Interaction Overview Diagram
3.Defining the TTMS Architecture
4.From Systems Engineering to Hardware and Software Engineering

3.construction:

1. Message Passing
2. Train Schedule Planning
3. Displaying Information
4. Sensor Data Acquisition
5. Release Management
6. System Architecture
7. Subsystem Specification

4.Post Transition

1.Inception

- Trains are an essential part of their transportation networks; tens of thousands of kilometers of track carry people and goods daily, both within cities and across national borders.
- In all fairness, trains do provide an important and economical means of transporting goods within the United States.
- Additionally, as major metropolitan centers grow more crowded, light rail transport is increasingly providing an attractive option for easing congestion and addressing the problems of pollution from internal combustion engines.
- The motivation for each of these systems is largely economic and social: Lower operating costs and more efficient use of resources are the goals, with improved safety as an integral by-product. In this section, we begin our analysis of the fictitious Train Traffic Management System (TTMS) by specifying its requirements and the system use cases that further describe the required functionality.

2. Requirements for the Train Traffic Management System :

- Experience with developing large systems has been that an initial statement of requirements is never complete, often vague, and always self-contradictory.
- This is a very large and highly complex system that in reality would not be specified by simple requirements. the requirements that follow will suffice for the purposes of our analysis and design effort.
- In the real world, a problem such as this could easily suffer from analysis paralysis because there would be many thousands of requirements, both functional and nonfunctional, with a myriad of constraints.
- Quite clearly, we would need to focus our efforts on the most critical elements and prototype candidate solutions within the operational context of the system under development.

The Train Traffic Management System has two primary functions:
1. train routing and 2. train systems monitoring.

- Related functions include traffic planning, failure prediction, train location tracking, traffic monitoring, collision avoidance, and maintenance logging. From these functions, define eight use cases, as shown in the following list.
- **Route Train**: Establish a train plan that defines the travel route for a particular train.
- **Plan Traffic**: Establish a traffic plan that provides guidance in the development of train plans for a time frame and geographic region.
- **Monitor Train Systems**: Monitor the onboard train systems for proper functioning.
- **Predict Failure**: Perform an analysis of train systems' condition to predict probabilities of failure relative to the train plan.
- **Track Train Location:** Monitor the location of trains using TTMS resources and the Navstar Global Positioning System (GPS).
- **Monitor Traffic:** Monitor all train traffic within a geographic region.
- **Avoid Collision:** Provide the means, both automatic and manual, to avoid train **collisions.**
- **Log Maintenance:** Provide the means to log maintenance performed on trains.
- In addition, we have nonfunctional requirements and constraints that impact the requirements specified by our use cases, as listed here. **Nonfunctional** requirements:
  ■ Safely transport passengers and cargo
  ■ Support train speeds up to 250 miles per hour
  ■ Interoperate with the traffic management systems of operators at the TTMS boundary
  ■ Ensure maximum reuse of and compatibility with existing equipment
  ■ Provide a system availability level of 99.99%
  ■ Provide complete functional redundancy of TTMS capabilities
  ■ Provide accuracy of train position within 10.0 yards
  ■ Provide accuracy of train speed within 1.5 miles per hour
  ■ Respond to operator inputs within 1.0 seconds
  ■ Have a designed-in capability to maintain and evolve the TTMS
- **Constraints:**
  ■ Meet national standards, both government and industry
  ■ Maximize use of commercial-off-the-shelf (COTS) hardware and **software Requirements defined**, at least at a very high level, we must turn our attention to understanding the users of the Train Traffic Management System.
  find that we have three types of people who interact with the system: Dispatcher, Train Engineer, and Maintainer.
- In addition, the Train Traffic Management System interfaces with one external system, Nav star GPS. These actors play the following roles within the TTMS.
  ■ Dispatcher establishes train routes and tracks the progress of individual trains.
  ■ Train Engineer monitors the condition of and operates the train.

■ Maintainer monitors the condition of and maintains train systems.
■ Nav star GPS provides geolocation services used to track trains

3.Determining System Use Cases

- Figure 9–1 shows the use case diagram for the Train Traffic Management System.
- the system functionality used by each of the actors.
- **«include» and «extend»** relationships used to organize relationships between several of the use cases.
- The functionality of the use case Monitor Train Systems is extended by the use case Predict Failure.
- During the course of monitoring systems, a failure prediction analysis (condition: {request Predict Failure}) can be requested for a particular system that is operating abnormally or may have been flagged with a yellow condition indicating a problem requiring investigation. This occurs at the Potential Failure extension point.
- The functionality of the Monitor Traffic use case is also extended, by that of the Avoid Collision use case.
- when monitoring train traffic, an actor has optional system capability to assist in the avoidance of a collision—at the Potential Collision extension point.
- This assistance can support both manual and automatic interventions.
- Monitor Traffic always includes the functionality of the Track Train Location use case to have a precise picture of the location of all train traffic.
- This is accomplished by using both TTMS resources and the Nav star GPS.
- It should be noted that these use case specifications focus on the boundary-level interaction between the users of the system and the Train Traffic Management System itself.
- This perspective is often referred to as a black-box view since the internal functioning of the system is not seen externally.
- **Use case name :** Route Train
- **Use case purpose:**
- The purpose of this use case is to establish a train plan that acts as a repository for all the pertinent information associated with the route of one particular train and the actions that take place along the way.
- **Point of contact: Katarina Bach**
- **Date modified: 9/5/06**
- **Preconditions:** A traffic plan exists for the time frame and geographic region (territory) relevant to the train plan being developed. **Postconditions:** A train plan has been developed for a particular train to detail its travel route.
- **Limitations:** Each train plan will have a unique ID within the system. Resources may not be committed for utilization by more than one train plan for a particular time frame.
- **Assumptions:** A train plan is accessible by dispatchers for inquiry and modification and accessible by train engineers for inquiry.

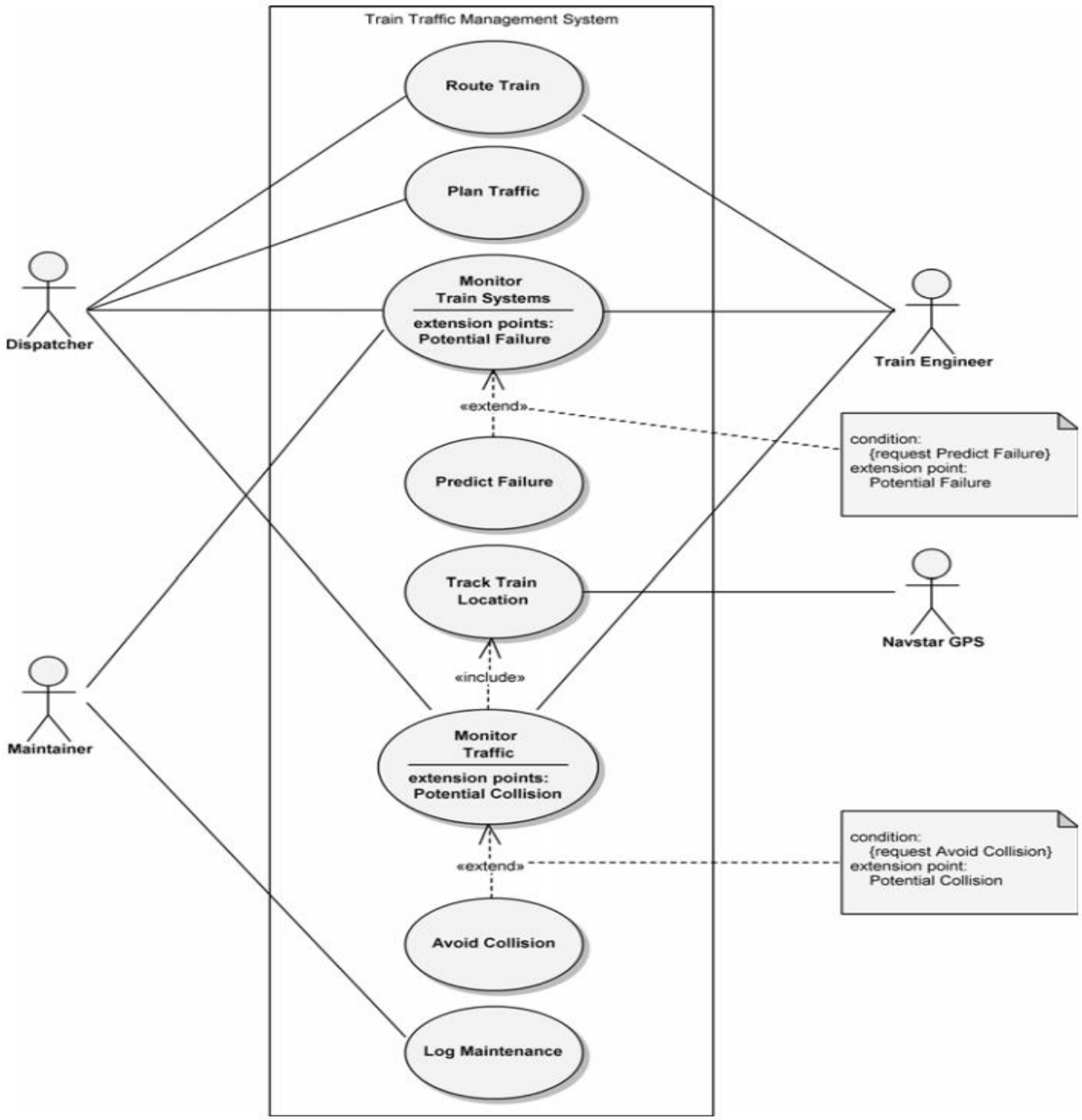


Figure 9–1 The Use Case Diagram for the Train Traffic Management System

Primary scenario:

- A. The Train Traffic Management System (TTMS) presents the dispatcher with a list of options.
- B. The dispatcher chooses to develop a new train plan.
- C. The TTMS presents the template for a train plan to the dispatcher.
- D. The dispatcher completes the train plan template, providing information about locomotive ID(s), train engineer(s), and waypoints with times.
- E. The dispatcher submits the completed train plan to the TTMS.
- F. The TTMS assigns a unique ID to the train plan and stores it. The
- TTMS makes the train plan accessible for inquiry and modification.

- G. This use case ends.

Alternate scenarios: Condition triggering an alternate scenario:

Condition 1: Develop a new train plan, based on an existing one.

- B1. The dispatcher chooses to develop a new train plan, based on an existing one.
- B2. The dispatcher provides search criteria for existing train plans.
- B3. The TTMS provides the search results to the dispatcher.
- B4. The dispatcher chooses an existing train plan.
- B5. The dispatcher completes the train plan.
- B6. The primary scenario is resumed at step E.
- Condition triggering an alternate scenario:
- Condition 2: Modify an existing train plan.
- B1. The dispatcher chooses to modify an existing train plan.
- B2. The dispatcher provides search criteria for existing train plans.
- B3. The TTMS provides the search results to the dispatcher.
- B4. The dispatcher chooses an existing train plan.
- B5. The dispatcher modifies the train plan.
- B6. The dispatcher submits the modified train plan to the TTMS.
- B7. The TTMS stores the modified train plan and makes it accessible for inquiry and modification.
- B8. This use case ends.
- Use case name: Monitor Train Systems Use case purpose:
- The purpose of this use case is to monitor the onboard train systems for proper functioning.
- Point of contact: Katarina Bach Date modified: 9/5/06
- Preconditions: The locomotive is operating
- Postconditions: Information concerning the functioning of onboard train systems has been provided.
- Limitations: None identified.

Assumptions: Monitoring of onboard train systems is provided when the locomotive is operating. Audible and visible indications of system problems, in addition to those via video display, are provided.

Primary scenario:

- A. The Train Traffic Management System (TTMS) presents the train engineer with a list of options.
- B. The train engineer chooses to monitor the onboard train systems.
- C. The TTMS presents the train engineer with the overview status information for the train systems.
- D. The train engineer reviews the overview system status information.
- E. This use case ends.
- **Assumptions:** Monitoring of onboard train systems is provided when the locomotive is operating. Audible and visible indications of system problems, in addition to those via video display, are provided.

- **Primary scenario**:
- A. The Train Traffic Management System (TTMS) presents the train engineer with a list of options.
- B. The train engineer chooses to monitor the onboard train systems.
- C. The TTMS presents the train engineer with the overview status information for the train systems.
- D. The train engineer reviews the overview system status information.
- E. This use case ends.
- Alternate scenarios:
- Condition triggering an alternate scenario:
- Condition 1: Request detailed monitoring of a system.
- E1. The train engineer chooses to perform detailed monitoring of a system that has a yellow condition.
- E2. The TTMS presents the train engineer with the detailed system status information for the selected system.
- E3. The train engineer reviews the detailed system status information. E4. The primary scenario is resumed at step B..
- **Extension point—Potential Failure:**
- **Condition** 2: Request a failure prediction analysis for a system.
- E3-1. The train engineer requests a failure prediction analysis for a system.
- E3-2. The TTMS performs a failure prediction analysis for the selected system.
- E3-3. The TTMS presents the train engineer with the failure prediction analysis for the system.
- E3-4. The train engineer reviews the failure prediction analysis.
- E3-5. The train engineer requests that the TTMS alert the maintainer of the system that might fail.
- E3-6. The TTMS alerts the maintainer of that system.
- E3-7. The maintainer requests the failure prediction analysis for review.
- E3-8. The TTMS presents the maintainer with the failure prediction analysis. E3-9. The maintainer reviews the analysis and determines that the yellow condition is not severe enough to warrant immediate action.
- E3-10. The maintainer requests that the TTMS inform the train engineer of this determination.
- E3-11. The TTMS provides the train engineer with the determination of the maintainer.
- E3-12. The train engineer chooses to perform detailed monitoring of the selected system.
- E3-13. The alternate scenario is resumed at step E3.

2.ELOBRATION:

- To developing the overall architecture framework for the Train Traffic Management System.
- begin by analyzing the required system functionality that leads us into the definition of the TTMS architecture.

- From there, transition from systems engineering to the disciplines of hardware and software engineering.
- conclude this section by describing the key abstractions and mechanisms of the TTMS.

1.Analyzing System Functionality
2.Interaction Overview Diagram
3.Defining the TTMS Architecture
4.From Systems Engineering to Hardware and Software Engineering

1.Analyzing System Functionality

- Let's begin by looking at Figure 9–2, which analyzes the primary scenario of the Route Train use case.
- This activity diagram is relatively straightforward and follows the course of the use case scenario.
- Here we see the interaction of the **Dispatcher actor** and the **Rail Operations Control System**, which we've designated as the primary command and control center for the TTMS, as the **Dispatcher** creates a new **Train Plan** object.
- In contrast to Figure 9–2, the activity diagram of Figure 9–3 is a bit more complicated because we've illustrated the first alternate scenario of the Monitor Train Systems use case, where the **Train Engineer** chooses to perform a detailed monitoring of the **Loco motive Analysis and Reporting** system, which has a yellow condition. Here we see that the constituent elements of the TTMS providing this capability are the **Onboard Display** system, the **Locomotive Analysis and Reporting** system, the **Energy Management** system, and the **Data Management** unit.
- the **On board Display** system is the interface between the **Train Engineer** and the TTMS.
- As such, it receives the **Train Engineer's** request to monitor the train systems and then requests the appropriate data from each of the other three systems.
- The overview level of status information is provided to the **Train Engineer** for review.
- At this point, the **Train Engineer** could remain at the overview level, which would end the primary scenario.
- In the alternate scenario, however, the **Train Engineer** requests a more detailed review from the **Locomotive Analysis and Reporting** system because it has presented a yellow condition indicating some type of problem that requires attention.
- In response, the **Onboard Display** system retrieves the detailed data from the system for presentation. After reviewing this information, the **Train Engineer** returns to monitoring the overview level of system status information.
- It is a matter of project convention whether we regard the activity diagram in Figure 9–3 as representing one (alternate) or two (primary and alternate) separate scenarios.
- The second alternate scenario we described earlier details the extension of the **Monitor Train Systems** use case functionality with that of the Predict Failure use case.
- This scenario could be appended to Figure 9–3 to provide a more complete picture of system capability by detailing the actions whereby the **Train Enginee**r requests a failure prediction analysis **(condition: {request Predict Failure})** be run on the problematic system.
- In fact, show this perspective in the Interaction Overview Diagram sidebar.

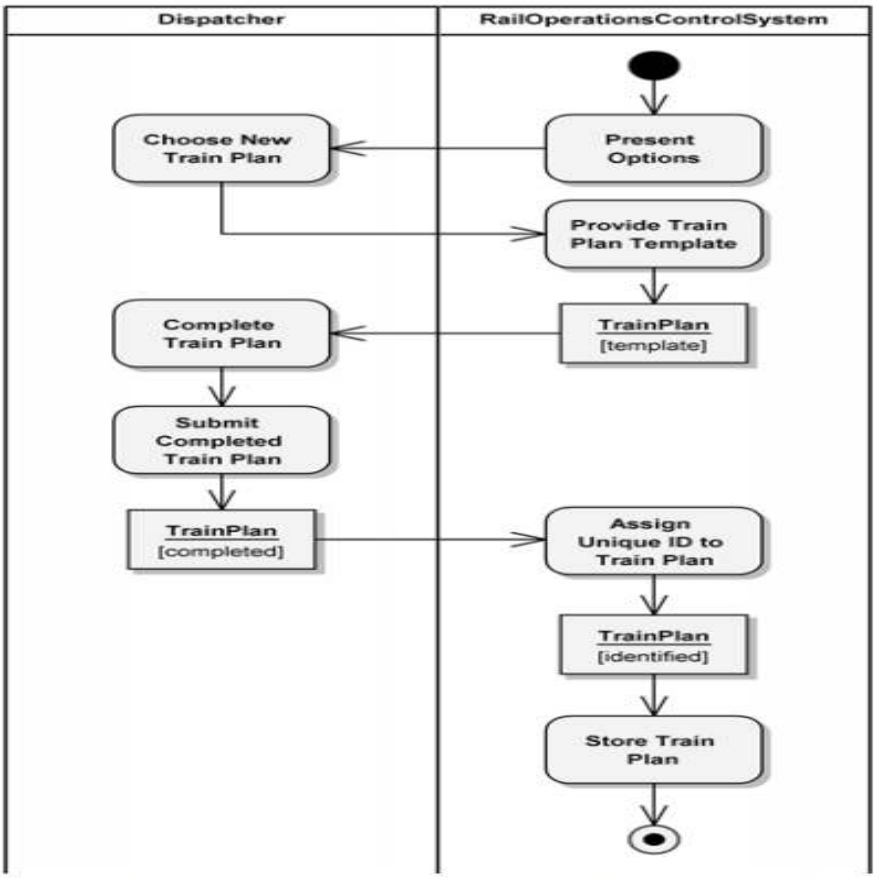


Figure 9–2 The `Route Train` Primary Scenario

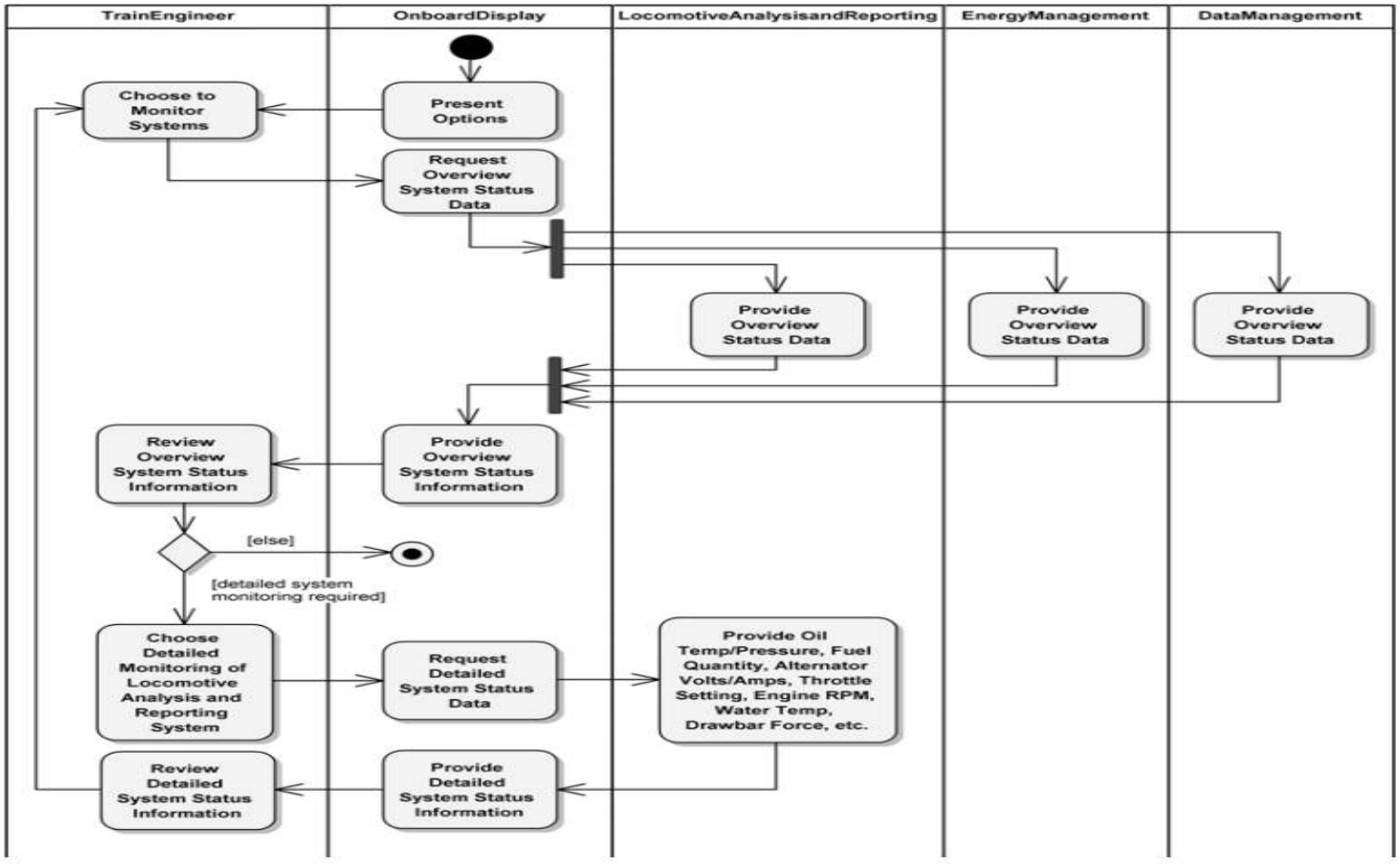


Figure 9–3 A `Monitor Train Systems` Alternate Scenario

2. **Interaction Overview Diagram** Another way to depict the Monitor Train Systems use case—with its primary and alternate scenarios—is by using an interaction overview diagram, as shown in Figure 9–4
3. True to its name, this diagram shows a higher-level overview of the complete Monitor Train Systems use case functionality.
4. This interaction overview diagram shows a flow among interaction occurrences, indicated by the three frames annotated with ref in their upper-left corners.
5. In place of the reference to interaction diagrams, we could show the actual interactions to provide the details for each of the scenarios: **Perform Overview System Status Monitoring, Perform Detailed System Status Monitoring, and Perform Failure Prediction Analysis.**

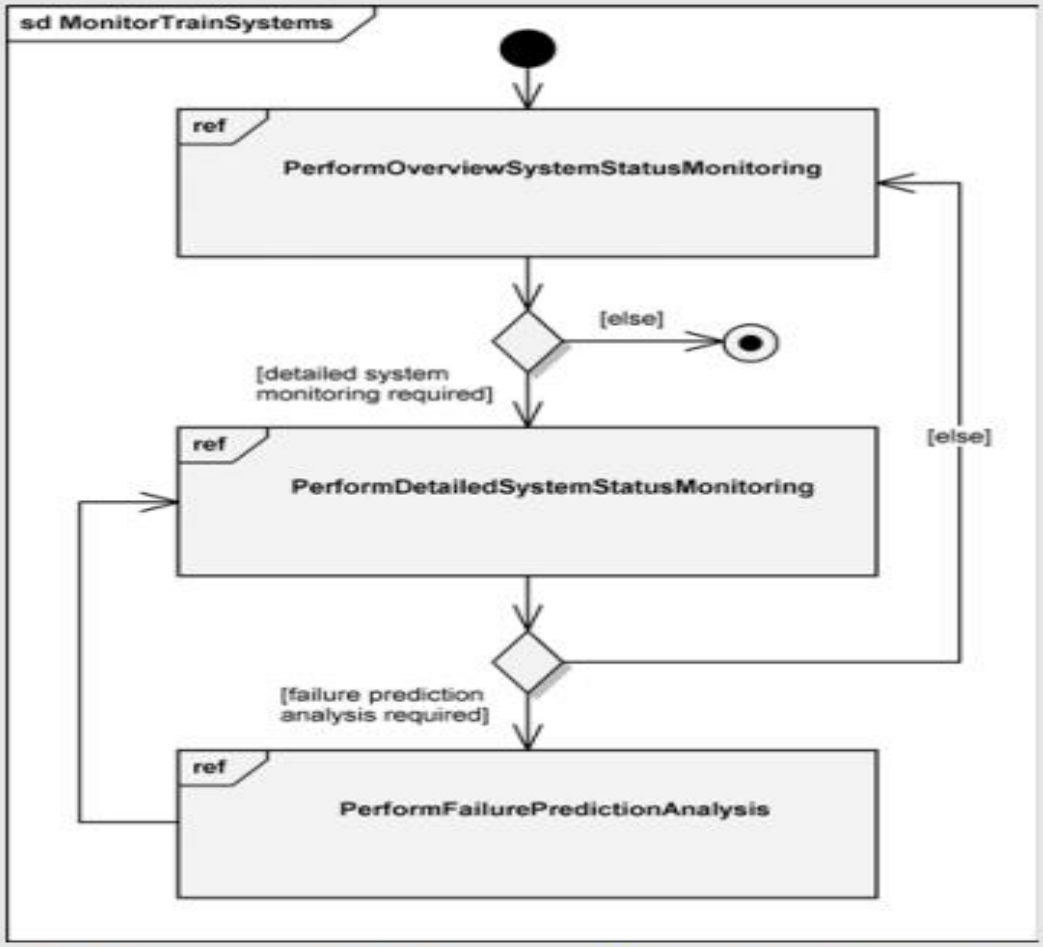


Figure 9–4 An Interaction Overview Diagram for Monitor Train Systems

3.Defining the TTMS Architecture

- A much more thorough analysis of the functionality required by all the use case scenarios, including the impact of the nonfunctional requirements and constraints, leads us to a block diagram for the Train Traffic Management System's major elements, as shown in Figure 9–5 The locomotive analysis and reporting system

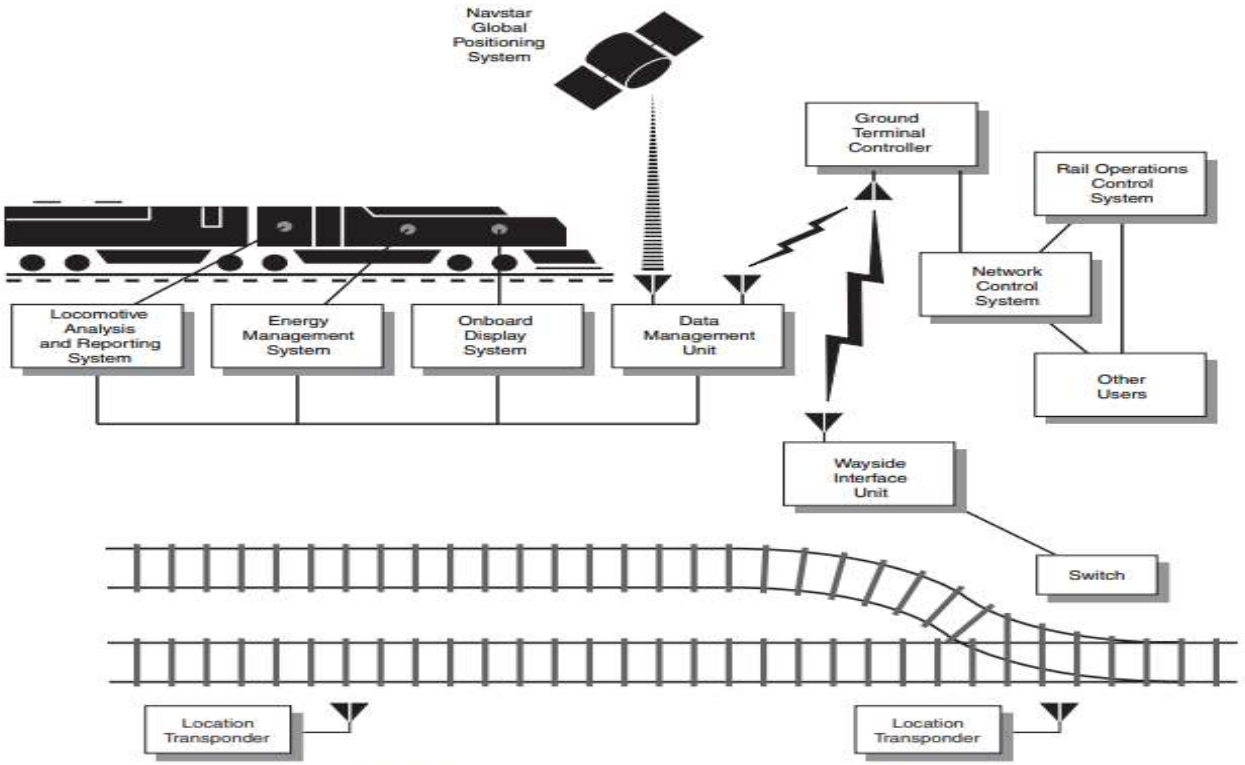


Figure 9–5 The Train Traffic Management System

4.From Systems Engineering to Hardware and Software Engineering

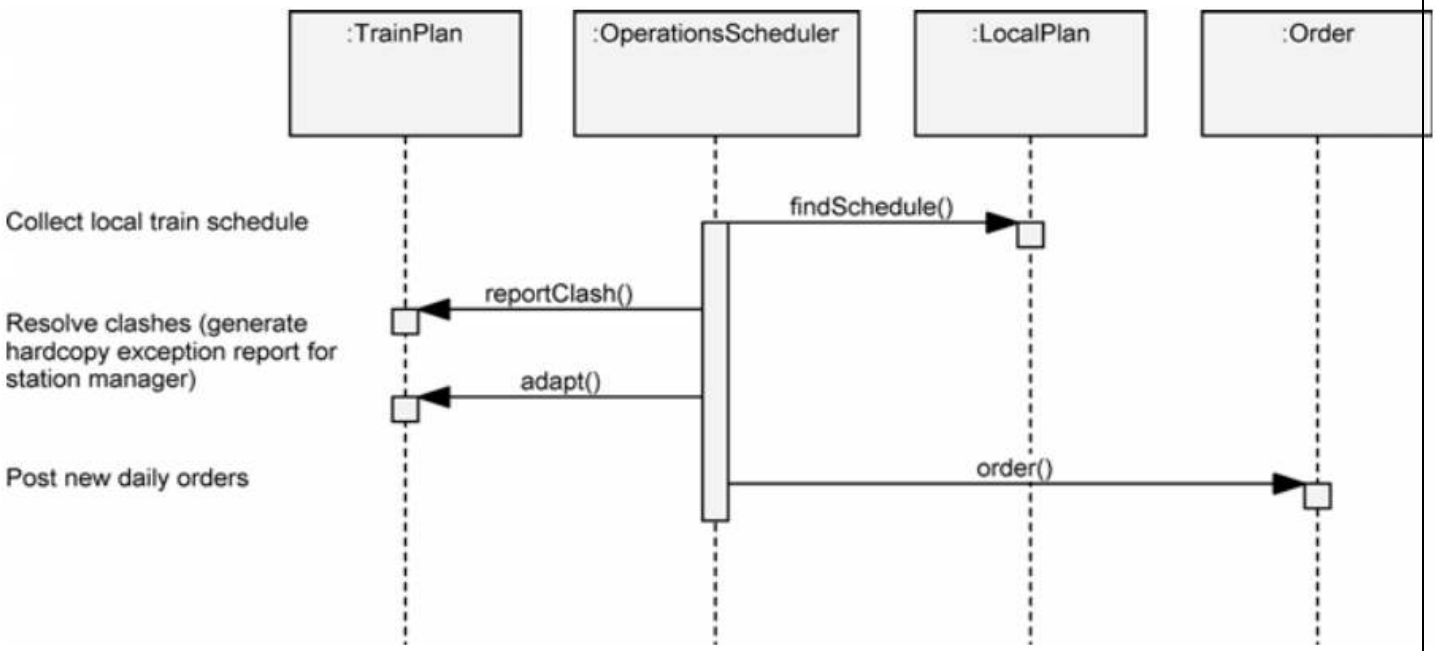


Figure 9–6 A Scenario for Processing Daily Train Orders

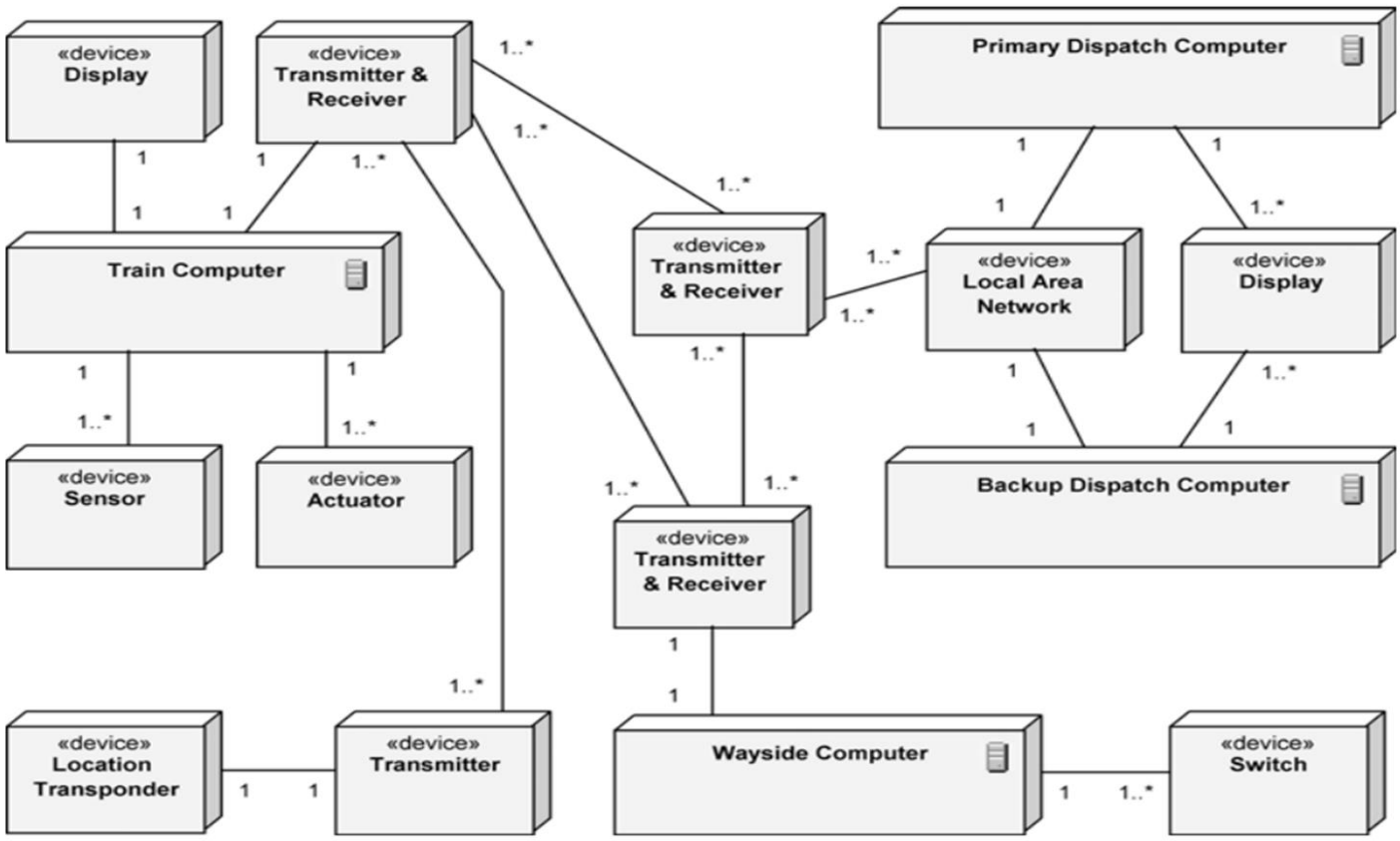


Figure 9–7 The Deployment Diagram for the Train Traffic Management System

Key Abstractions and Mechanisms

- A study of the requirements for the Train Traffic Management System suggests that we really have four different subproblems to solve:
- 1. Networking 2. Database 3. Human/machine interface 4. Real-time analog and digital device control
- Brief domain analysis across these four subproblem areas, we find that there are three common high-level key abstractions:

1. Trains: including locomotives and cars
2. Tracks: encompassing profile, grade, and wayside devices
3. Plans: including schedules, orders, clearances, authority, and crew assignments

- key mechanism for each of the four nearly independent subproblems:

1. Message passing 2. Train schedule planning 3. Displaying information 4. Sensor data acquisition

3.construction:

1. **Message Passing**
2. **Train Schedule Planning**
3. **Displaying Information**
4. **Sensor Data Acquisition**
5. **Release Management**
6. **System Architecture**
7. **Subsystem Specification**

4.Post Transition

1. Message Passing

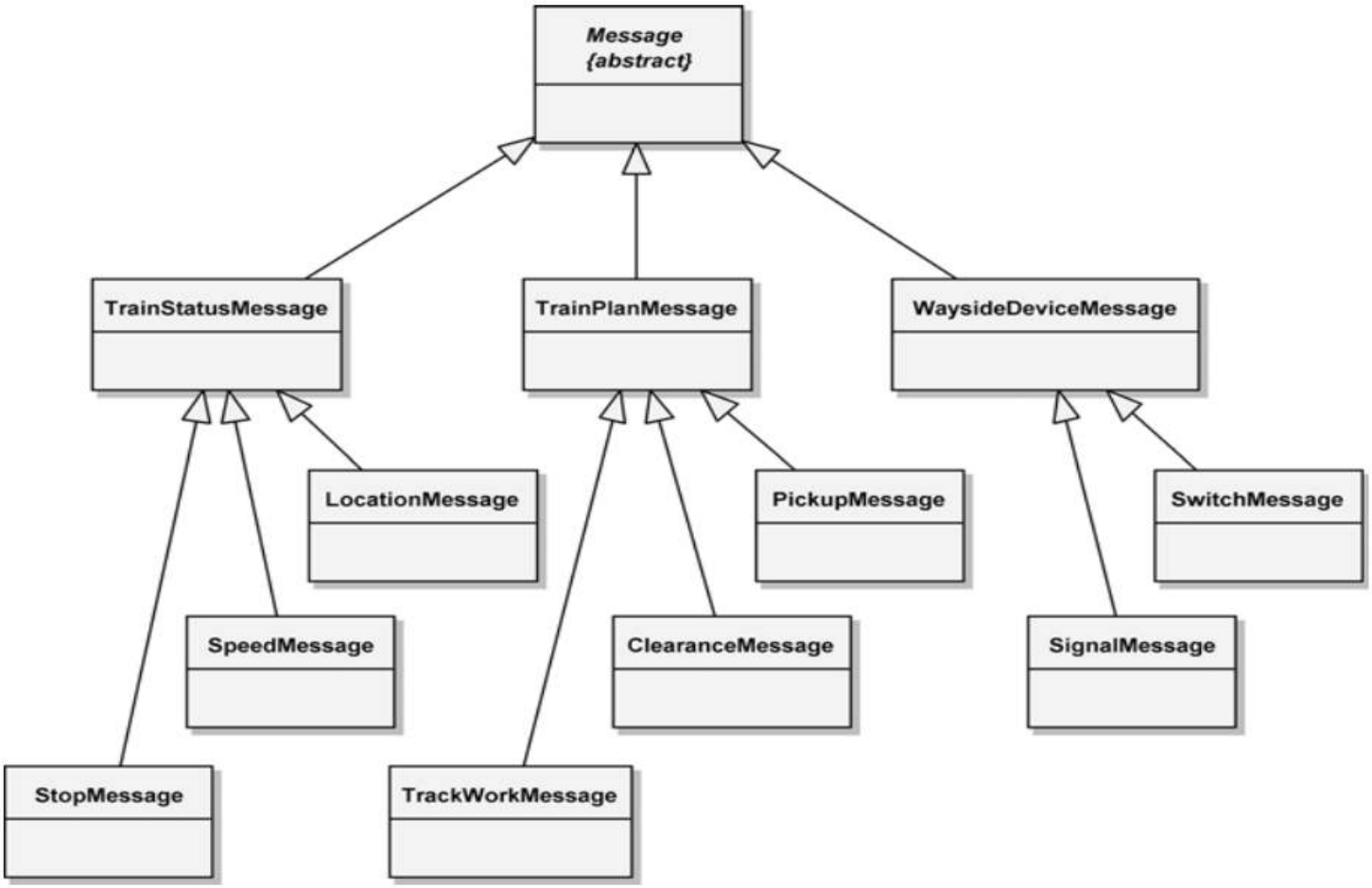


Figure 9–8 The Message Class Diagram

The class diagram in Figure 9–8 captures our design decisions regarding some of the most important messages in the Train Traffic Management System.

Note that all messages are ultimately instances of a generalized abstract class named Message, which encompasses the behavior common to all messages.

Three lower-level classes represent the major categories of messages, namely, Train **Status Message, Train Plan Message, and Way side Device Message.**

Each of these classes is further specialized. Indeed, final design might include dozens of such specialized classes, at which time the existence of these intermediate classes becomes even more important; without them, we would end up with many unrelated—and therefore difficult to maintain—components representing each distinct specialized class.

Once satisfied with this class structure, begin to design the message-passing mechanism itself. Here we have two competing goals for the mechanism:

It must provide for the reliable delivery of messages and yet do so at a high enough level of abstraction so that clients need not worry about how message delivery takes place.

Figure 9–9 provides a scenario that captures our design of the message-passing mechanism. As this diagram indicates, to send a message, a client first creates a new message m and then broadcasts it to its node's message manager, whose responsibility is to queue the message for eventual transmission.

Notice that our design uses four objects that are active (have their own thread of control), as indicated by the extra vertical lines within the object notation: Client, message Mgr : Queue, message Mgr' :Queue, and Receiver.
Notice also that the message manager receives the message to be broadcast as a parameter and then uses the services of a Transporter object to reduce the message to its canonical form and broadcast it across the network

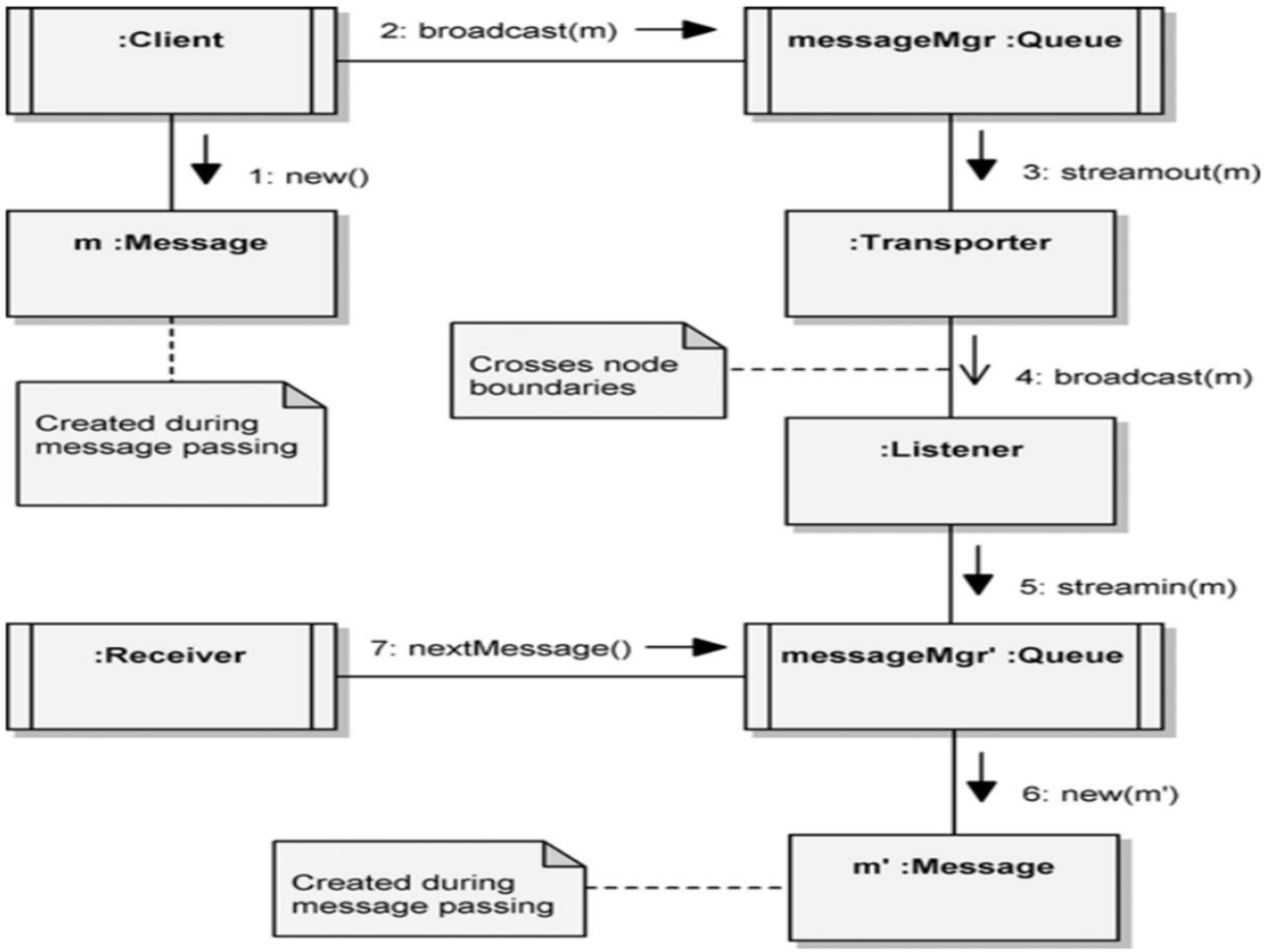


Figure 9–9 The Message-Passing Mechanism

2.Train Schedule Planning

- Figure 9–10 captures our strategic decisions regarding the structure of the Train Plan class. We use a class diagram to show the parts that compose a train plan

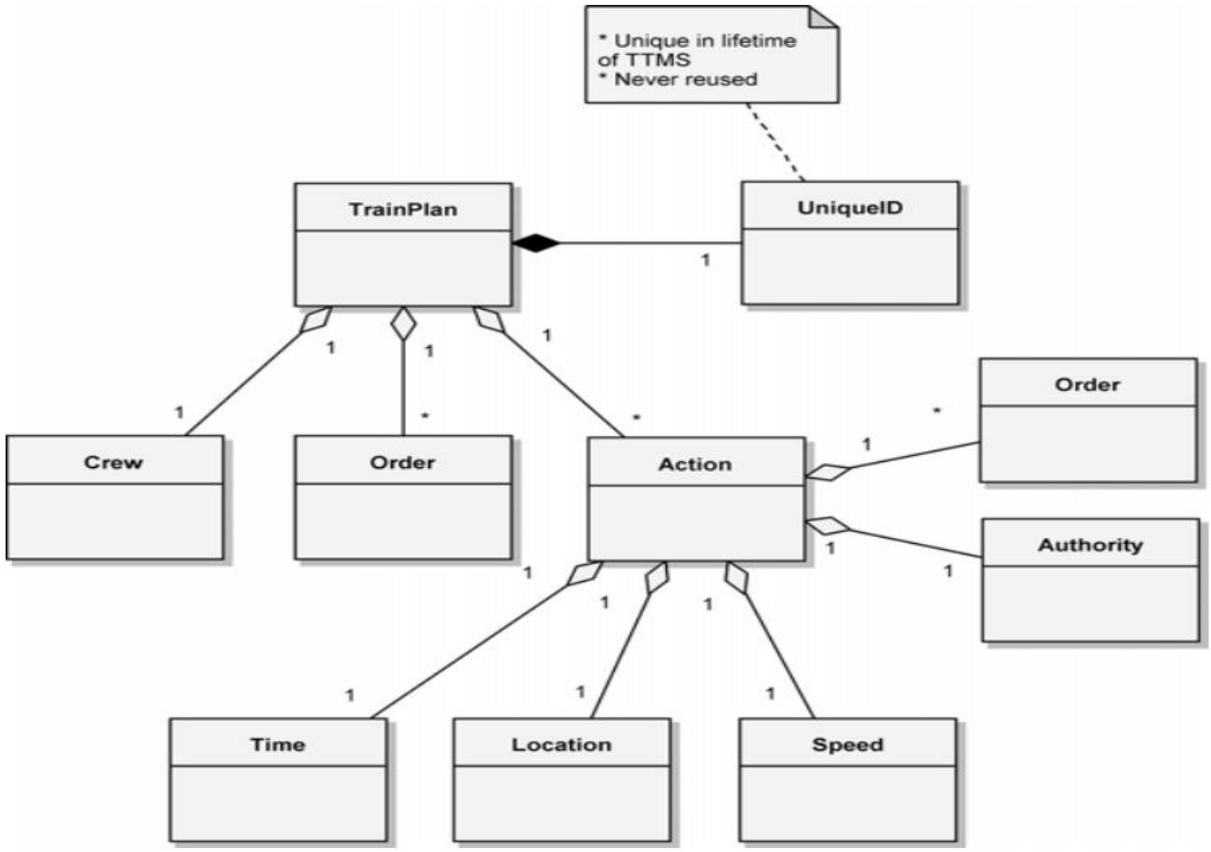


Figure 9–10 The `TrainPlan` Class Diagram

As the diagram in Figure 9–10 indicates, the **Train Plan** class has a **Unique Id**, whose purpose is to provide a number for uniquely identifying each Train Plan instance. Because of the complexity of the information here, classes in this diagram that might otherwise be considered an attribute of a class are really stand-alone classes. For example, the **UniqueID** class is not merely an identification number; it contains various attributes and operations necessary to meet stringent national and international regulations. Another example is that crews have restrictions placed on their work—they may work only at certain locations or must adhere to speed restrictions in certain locations at particular times.

shown in Figure 9–11. The primary version of each train plan resides in a centralized database at a dispatch center, with zero or more mirror-image copies scattered about the network. Whenever some client requests a copy of a particular train plan (via the **operation get(),** invoked with a value of **Unique Id** as an argument), the primary version is cloned and delivered to the client as a parameter, and the network location of the copy is recorded in the database

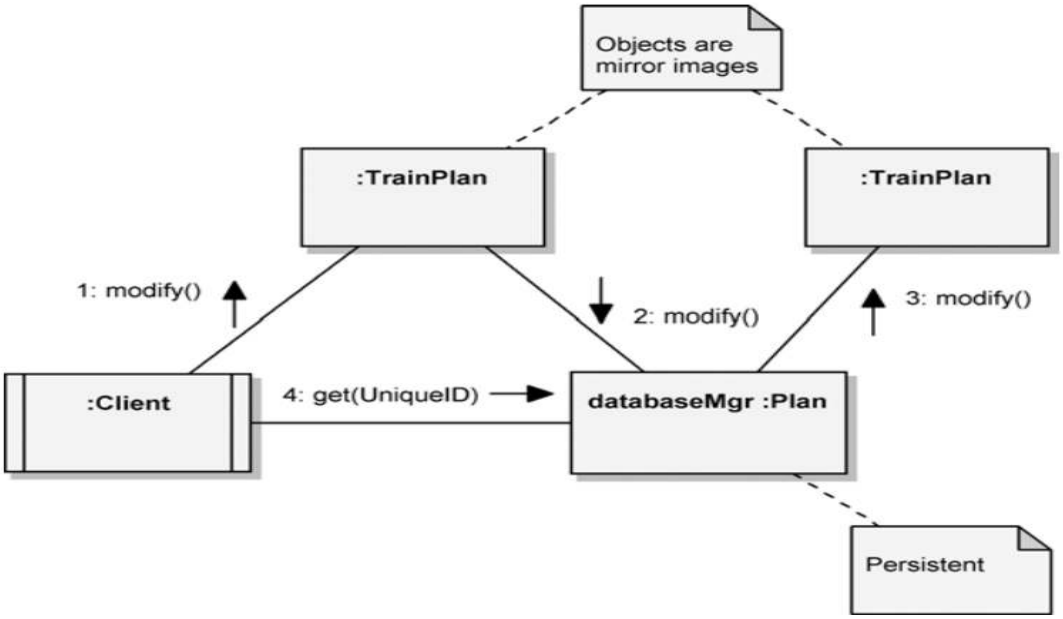


Figure 9–11 Train Schedule Planning

3.Displaying Information

Figure 9–12 illustrates this design, showing that the implementation of all displayable objects shares common class utilities.

These utilities in turn build on lower-level Windows interfaces, which are hidden from all of the higher-level classes. Pragmatically, interfaces such as the Windows API cannot easily be expressed in a single class.

Therefore, **diagram is a bit of a simplification:** It is more likely that our implementation will require a set of peer class utilities for the Windows API as well as for the train display utilities.

The principal advantage of this approach is that it limits the impact of any lower level changes resulting from hardware/software trade-offs.

For example, if we find that we need to replace our display hardware with more or less powerful devices, we need only reimplement the routines in the **Train Display Utility** class.

Without this collection of routines, low-level changes would require us to modify the implementation of every displayable object.

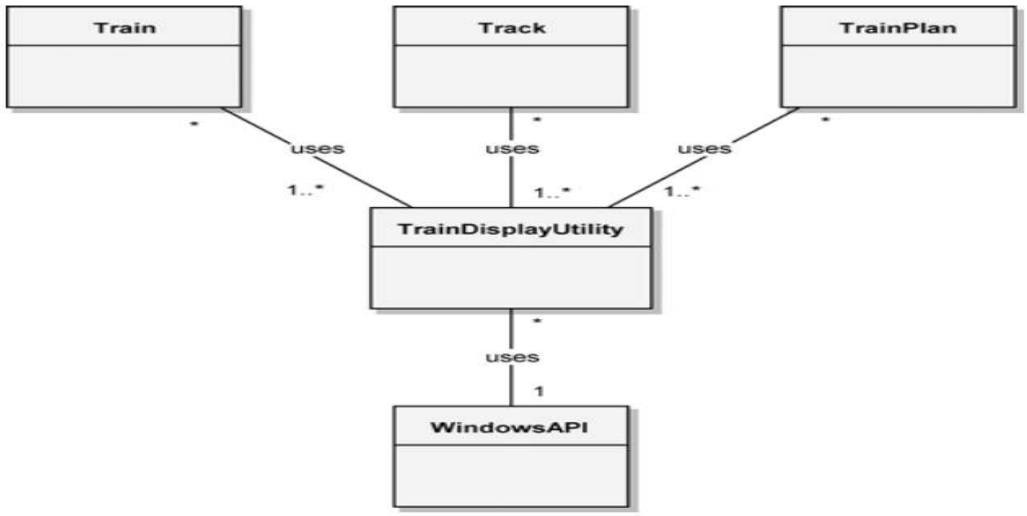


Figure 9–12 Class Utilities for Displaying

4 .Sensor Data Acquisition

- the Train Traffic Management System includes many different kinds of sensors. For example, sensors on each train monitor the oil temperature, fuel quantity, throttle setting, water temperature, drawbar load, so on.
- The kinds of values returned by the various sensors are all different, but the processing of different sensor data is all very much the same.
- Replicating this behavior for every kind of sensor not only is tedious and error prone but also usually results in redundant code.
- use an architecture that encompasses a hierarchy of sensor classes and a frame-based mechanism that periodically acquires data from these sensors.

5.Release Management

- using an incremental development approach, we will investigate the

employment of release management techniques and further analyze the system's architecture and the specification of its subsystems.

For example, the primary scenarios of three use cases: **Route Train, Monitor Train Systems, and Monitor Traffic.**

Once we successfully pass this milestone, we might then generate a stream of new releases, according to the following sequence.

1. Create a train plan based on an existing one; modify a train plan.
2. Request detailed monitoring of a system with a yellow condition; request a failure prediction analysis; request maintainer review of a failure prediction analysis.
2. Manually avoid a collision; request automated assistance in avoiding a collision; track train traffic using either TTMS resources or Navstar GPS.

6.System Architecture

The software design for very large systems must often commence before the target hardware is completed. Software design frequently takes far longer than hardware design, and in any case, trade-offs must be made against each along the way. This implies that hardware dependencies in the software must be isolated to the greatest extent possible, so that software design can proceed in the absence of a stable target environment.

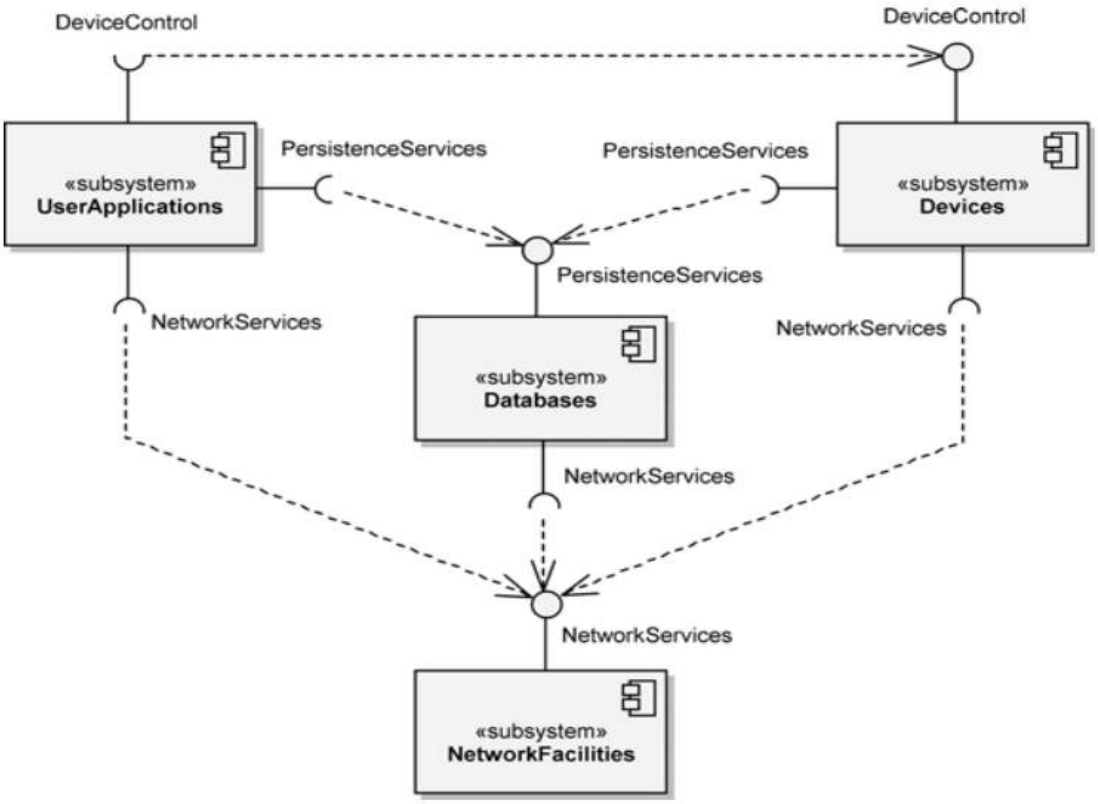


Figure 9–13 The Top-Level Component Diagram for the Train Traffic Management System

The software design for very large systems must often commence before the target hardware is completed. Software design frequently takes far longer than hardware design, and in any case, trade-offs must be made against each along the way.

This implies that hardware dependencies in the software must be isolated to the greatest extent possible, so that software design can proceed in the absence of a stable target environment.

The component diagram shown in Figure 9–13 represents our design decisions regarding the top-level system architecture of the Train Traffic Management System.

layered architecture that encompasses the functions of the four subproblems we identified earlier, namely, networking, database, the human/ machine interface, and real-time device control.

7.Subsystem Specification

- focus on the outside view of any of these subsystems, find that it has all the characteristics of an object. It has a unique, albeit static, identity; it embodies a significant amount of state; and it exhibits very complex behavior. Subsystems serve as the repositories of other subsystems and eventually classes; thus, they are best characterized by the resources they export through their provided interfaces, such as the Network Services provided by the Network Facilities subsystem shown in Figure 9–13.
- The component diagram in Figure 9–13 is merely a starting point for the specification of the TTMS subsystem architecture. These top-level subsystems must be further decomposed through multiple architectural levels of nested subsystems. Looking at the Network Facilities subsystem, decompose it into two other subsystems, a private Radio

Communication subsystem and a public Messages subsystem. The private subsystem hides the details of software control of the physical radio devices.

- The subsystem named Databases builds on the resources of the subsystem Network Facilities and implements the train plan mechanism we created earlier.
- name these nested subsystems **Train Plan Database** and **Track Database,** respectively. We also expect to have one private subsystem, Database Manager, whose purpose is to provide all the services common to the two domain-specific databases.
- In the Devices subsystem, we choose to group the software related to all wayside devices into one subsystem and the software associated with all onboard locomotive actuators and sensors into another. These two subsystems are available to clients of the **Devices** subsystem, and both are built on the resources of the **Train Plan Database** and Messages subsystems.
- Finally, choose to decompose the top level **User Applications** subsystem into several smaller ones, including the subsystems **Engineer Applications** and **Dispatcher Applications**, to reflect the different roles of the two main users of the Train Traffic Management System.
- The subsystem Engineer Applications includes resources that provide all the train-engineer/machine interaction specified in the requirements, including the functionality of the **locomotive** analysis and reporting system and the energy management system.
- Include the subsystem **Dispatcher Applications** to encompass the software that provides the functionality of all dispatcher/machine interactions.
- Both Engineer Applications and Dispatcher Applications share common private resources, as provided from the subsystem Displays.
- For example, consider the subsystem **Train Plan Database**. It builds on three other subsystems (Messages, Train Database, and Track Database) and has several important clients, namely, the four subsystems, Wayside Devices, Locomotive Devices, Engineer Applications, and Dispatcher Applications. The Train Plan Database embodies a relatively straightforward state, specifically, the state of all train plan.

4. POST TRANSITION

- For the Train Traffic Management System, we can envision a significant addition to our requirements, namely, payroll processing. Specifically, suppose that our analysis shows that train company payroll is currently being supported by a piece of hardware that is no longer being manufactured and that we are at great risk of losing our payroll processing capability because a single serious hardware failure would put our accounting system out of action forever.
- Our solution would be to add a new subsystem between the subsystems Train Plan Database and Dispatcher Applications because the knowledge base embodied by this expert system parallels the contents of the Train Plan Database; furthermore, the subsystem Dispatcher Applications is the sole client of this expert system. We would need to invent some new mechanisms to establish the manner in which advice is presented to the ultimate user. For example, we might use a blackboard architecture.

UNIT III: Class & Object Diagrams: Terms, concepts, modeling techniques for Class & Object Diagrams.

Common Modeling Techniques of classes:

1.Modeling the Vocabulary of a System

To model the vocabulary of a system,

- Identify those things that users or implementers use to describe the problem or solution.
- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

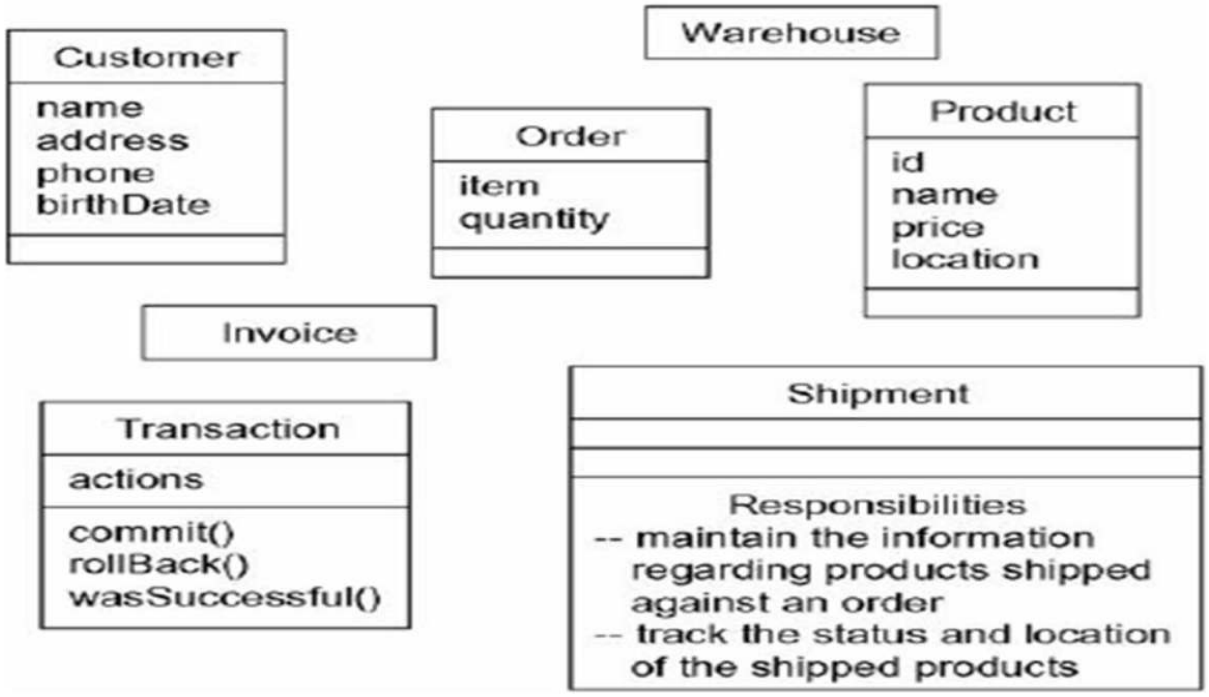
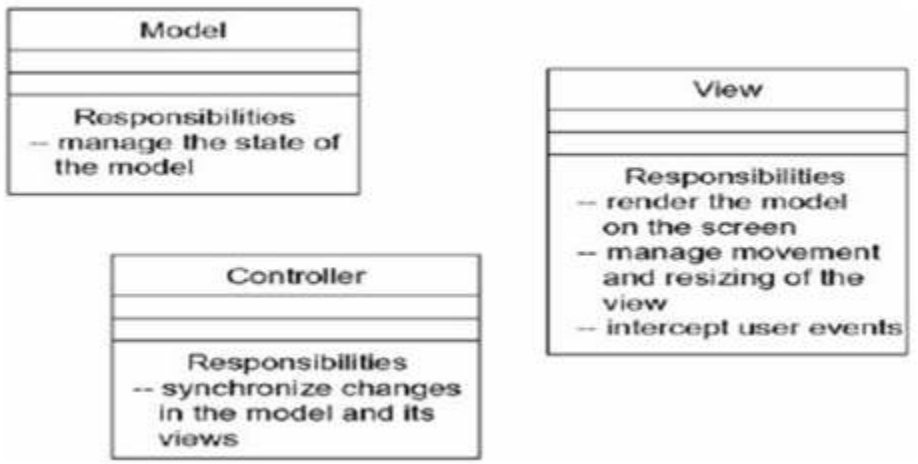


Figure shows a set of classes drawn from a retail system, including Customer, Order, and Product. This figure includes a few other related abstractions drawn from the vocabulary of the problem, such as Shipment (used to track orders), Invoice (used to bill orders), and Warehouse (where products are located prior to shipment). There is also one solution-related abstraction, Transaction, which applies to orders and shipments.

2. Modeling the Distribution of Responsibilities in a System

To model the distribution of responsibilities in a system,

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions.



For example, Figure shows a set of classes drawn from Smalltalk, showing the distribution of responsibilities among Model, View, and Controller classes. Notice how all these classes work together such that no one class does too much or too little.

1. Modeling Nonsoftware Things

To model nonsoftware things,

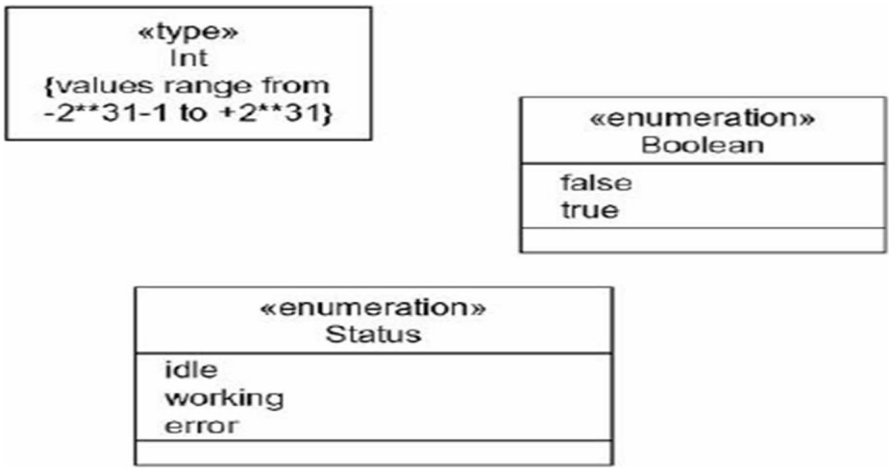
- Model the thing we are abstracting as a class.
- If we want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing we are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that we can further expand on its structure.

As Figure shows, it's perfectly normal to abstract humans (like Accounts Receivable Agent) and hardware (like Robot) as classes, because each represents a set of objects with a common structure and a common behavior.

2. Modeling Primitive Types

To model primitive types,

- Model the thing we are abstracting as a type or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If we need to specify the range of values associated with this type, use constraints.



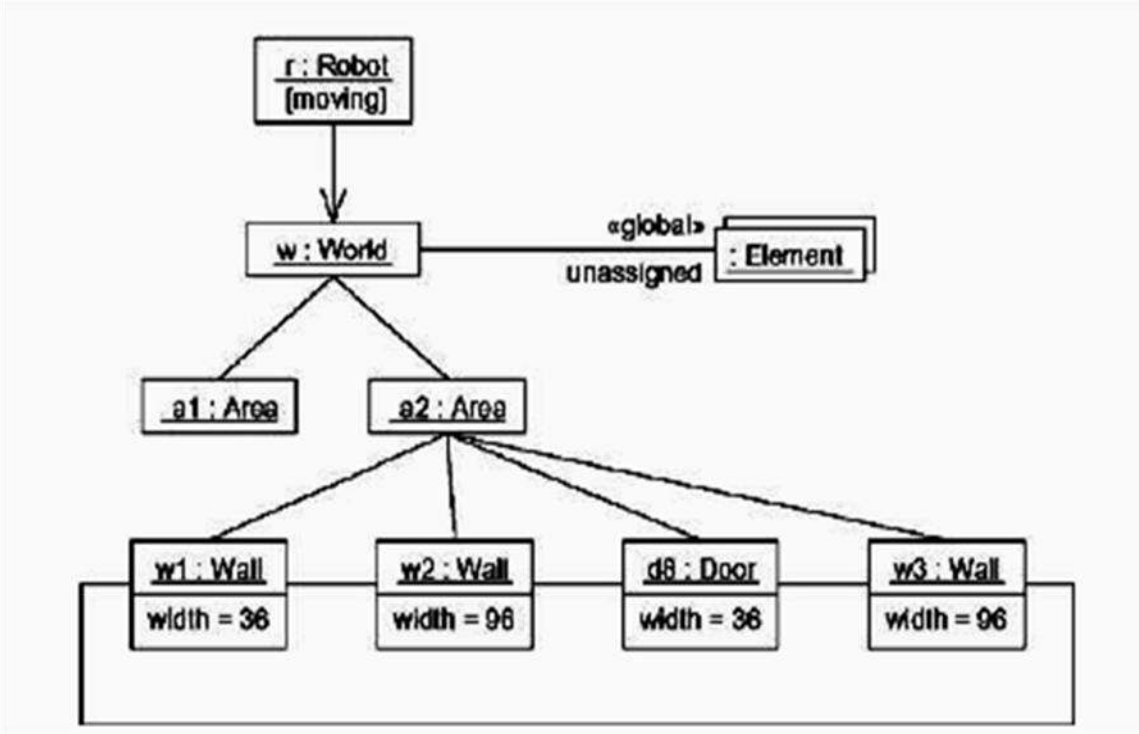
As Figure shows, these things can be modeled in the UML as types or enumerations, which are rendered just like classes but are explicitly marked via stereotypes. Things like integers (represented by the class Int) are modeled as types, and we can explicitly indicate the range of values these things can take on by using a constraint. Similarly, enumeration types, such as Boolean and Status, can be modeled as enumerations, with their individual values provided as attributes.

Common Modeling Techniques of object diagrams:

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.
- 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.



For example, Figure shows a set of objects drawn from the implementation of an autonomous robot. This figure focuses on some of the objects involved in the mechanism used by the robot to calculate a model of the world in which it moves. There are many more objects involved in a running system, but this diagram focuses on only those abstractions that are directly involved in creating this world view.

As this figure indicates, one object represents the robot itself (r, an instance of Robot), and r is currently in the state marked moving. This object has a link to w, an instance of World, which represents an abstraction of the robot's world model. This object has a link to a multi object that consists of instances of Element, which represent entities that the robot has identified but not yet assigned in its world view. These elements are marked as part of the robot's global state.

At this moment in time, w is linked to two instances of Area. One of them (a2) is shown with its own links to three Wall and one Door object. Each of these walls is marked with its current width, and each is shown linked to its neighboring walls. As this object diagram suggests, the robot has recognized this enclosed area, which has walls on three sides and a door on the fourth.

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 can't exactly instantiate these objects from the outside.

Reverse engineering (the creation of a model from code) an object diagram is a very different thing. 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.

If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

- **Object Diagram**