ICS 3105
OBJECT ORIENTED SOFTWARE ENGINEERING

Chapter 5.3

Object Diagrams

Learning Outcomes

- By the end of this chapter, the learner should be able to:
  - Identify classes and objects.
  - Draw object diagrams.
  - Use aggregations and compositions in object diagrams.

Introduction

- Class diagrams tell software engineers what classes will exist in a given system, but they are quite abstract.
- Sometimes it can be hard to visualize the relationships among the objects that will exist at run-time.

Object diagrams

- An object diagram shows an example configuration of objects and links that may exist at a particular point during execution of a program.
- Objects are shown as rectangles, just like classes; the difference is that the name of the class is underlined and preceded by a colon(:).

Object diagrams

- For example :Employee.
- Software engineers can also give a name to each instance before the colon, as in Pat:Employee, or even omit the class name entirely if it is clear from the context, such as Pat:.

Object diagrams

- A link between two objects is shown as a simple line.
- Each of the two objects contains a pointer to the other object joined by the link.

Relationship between Object and Class diagrams

- It is important to understand the relationship between a class diagram and an object diagram.
  - A class is an abstract representation of all the instances of that class that can ever exist.

Relationship between Object and Class diagrams

- Similarly, an association represents all the links between two classes that can ever exist.
- It should be clear from this that while we put multiplicity symbols on associations, we never put them on links.

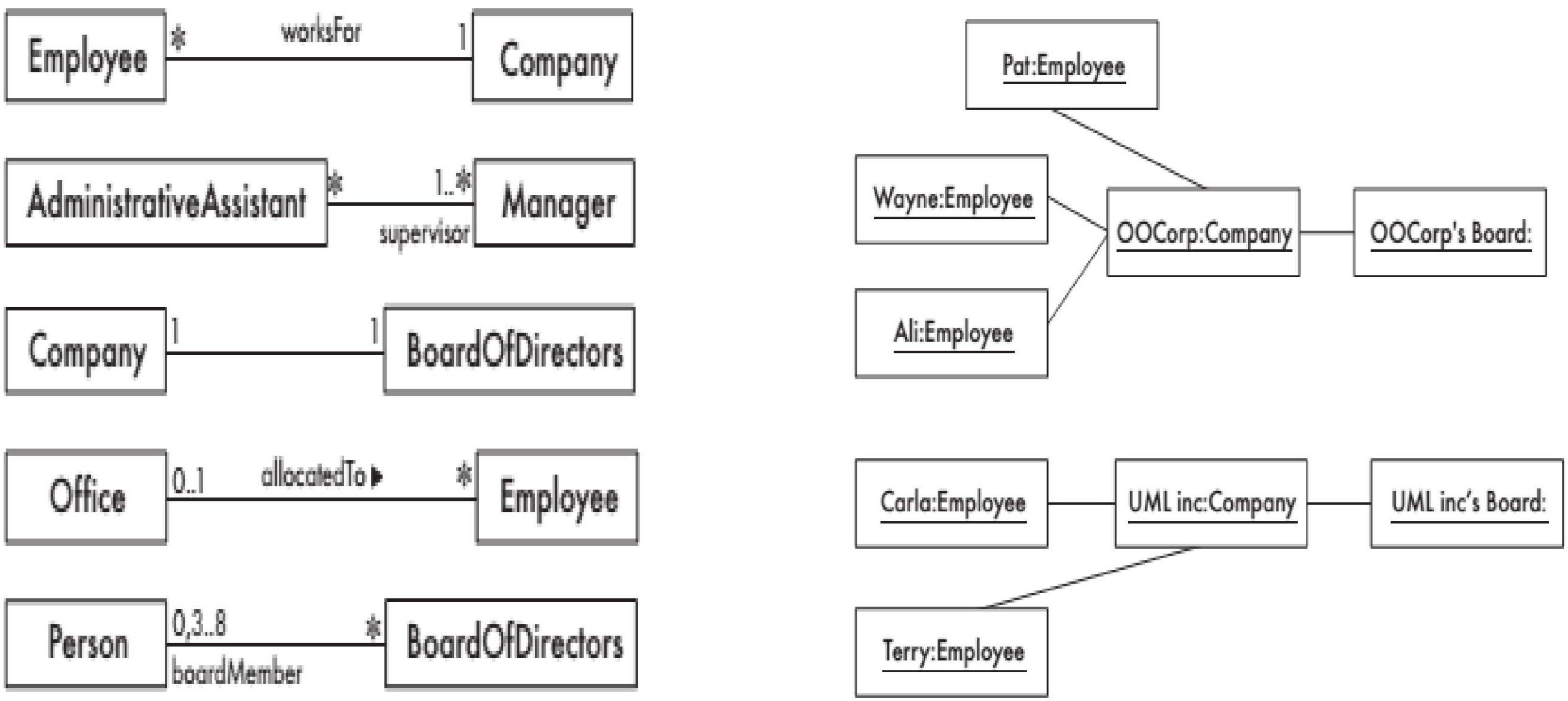
Relationship between Object and Class diagrams

- A given object diagram is generated by a class diagram i.e. it contains instances and links of the classes and associations present in the class diagram.

Relationship between Object and Class diagrams

- It also means that the numbers of links among instances are consistent with the multiplicity of that class diagram.
- A class diagram can generate an infinite number of object diagrams.

Class and Object diagrams



Associations versus generalizations in the context of object diagrams

- It is a common mistake for beginners to think of generalizations as special associations.
- This misconception arises because both generalizations and associations connect classes together in a class diagram.

Associations versus generalizations in the context of object diagrams

- However, the differences between the two concepts are profound.
  - An association describes a relationship that will exist between instances at runtime.
  - A generalization describes a relationship between classes in a class diagram.

Associations versus generalizations in the context of object diagrams

- An object diagram can never contain a generalization, and can only contain links generated by associations, not the associations themselves.
- When you show an object diagram generated by an association, you show instances of both classes joined by that association.

Associations versus generalizations in the context of object diagrams

- On the other hand, when you show an object diagram generated by an inheritance hierarchy, you show a single instance of one of its concrete classes.
- That single instance will contain values of the attributes defined in its class, as well as those attributes inherited from super classes.

Associations versus generalizations in the context of object diagrams

- In other words, an instance of any class should also be considered to be an instance of each of that class's super classes.

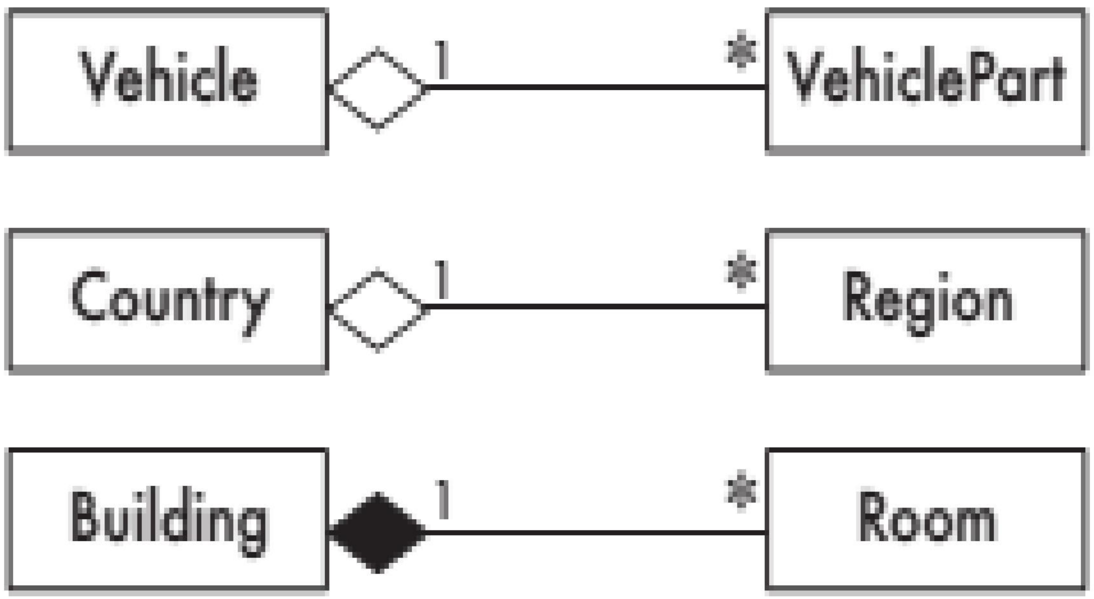
More advanced features of class diagrams

- Additional features can be used to add more specific information to the diagrams.
- It is important to be able to understand the meaning of these features in class diagrams, but most modeling, especially at the analysis level, can be done without them.

Aggregations

- Aggregations are special associations that represent 'part–whole' relationships.
- The 'whole' side of the relationship is often called the assembly or the aggregate.
- Aggregations are specified using a diamond symbol, which is placed next to the aggregate.

Aggregation



Aggregation

- This symbol is a shorthand notation that saves you from having to write an association name such as *isPartOf* or its inverse *hasParts*.
- Many aggregations are one-to-many, but this is not a requirement.

Aggregation

- When to use an aggregation instead of an ordinary association has always been a source of confusion.

Aggregation

- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that the parts 'are part of' the aggregate, or the aggregate 'is composed of' the parts.
  - When something owns or controls the aggregate, then they also own or control the parts.

Composition

- A composition is a strong kind of aggregation in which if the aggregate is destroyed, then the parts are destroyed as well.
- A composition is shown using a solid (filled-in) diamond, as opposed to an open one.

Composition

- The parts of a composition can never have a life of their own; they exist only to serve the aggregate.
- For example, the rooms of a building cannot exist without the building.

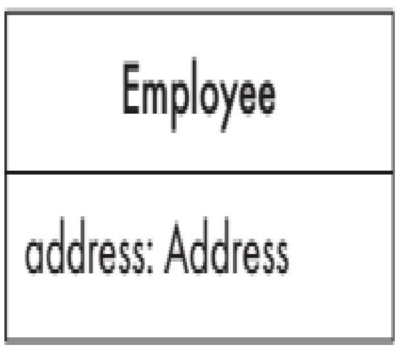
Composition

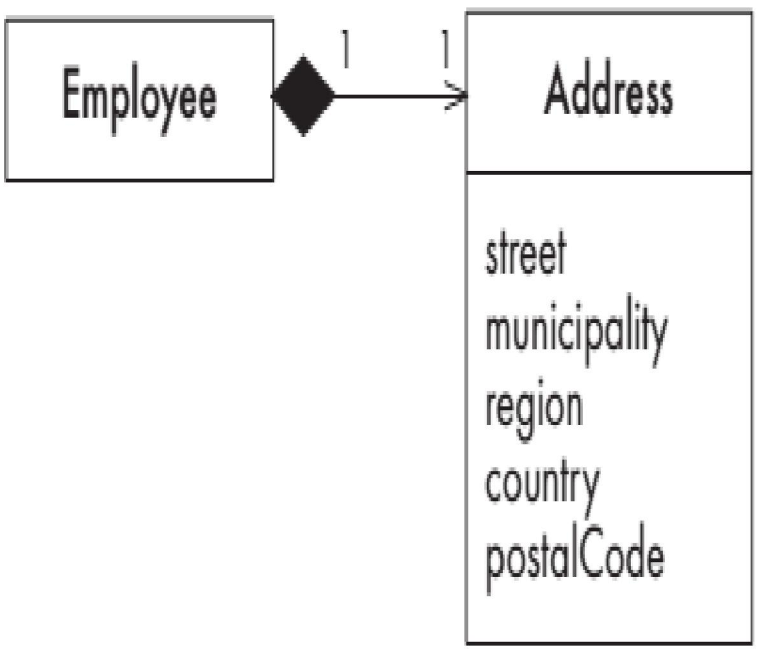
- In ordinary aggregations, on the other hand, the parts can exist on their own.
- For example, the engine can be taken out of one vehicle and placed in another, or a region can secede from one country and become independent.

Composition

- A one-to-one composition often corresponds to a complex attribute.
- You can therefore show it as an attribute or, if you want to emphasize the details of the composed class, you can show it as a composition.

Composition

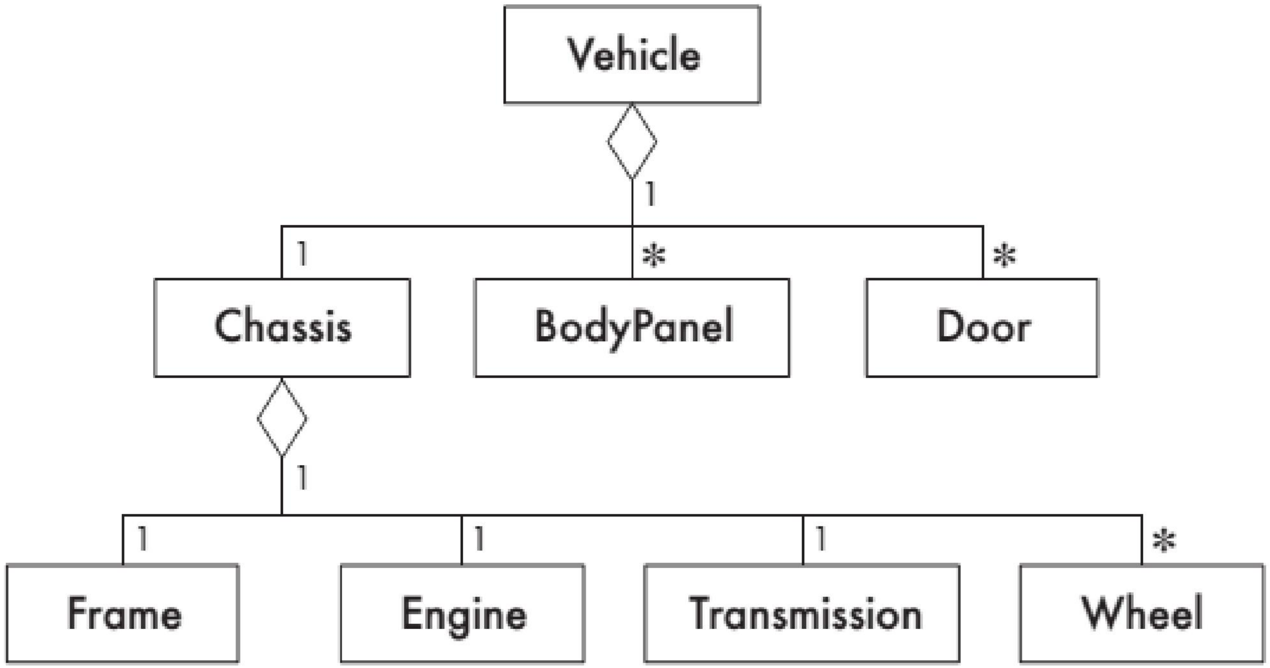




Composition

- The address of an employee represented as an attribute or as a composition.
- Unlike other associations, UML allows aggregations to be drawn as a hierarchy.
- However, the use of such hierarchies in valid models is quite rare.

An aggregation hierarchy



Advantage of explicit aggregation

- An advantage of explicitly identifying aggregation is that it provides useful information to the designer.
- In particular, the designer can improve the encapsulation of the system by arranging for the part objects to be hidden inside the aggregate object.

Advantage of explicit aggregation

- Methods in the system would be able to perform most operations on the aggregate, without needing to know about the existence of the parts.

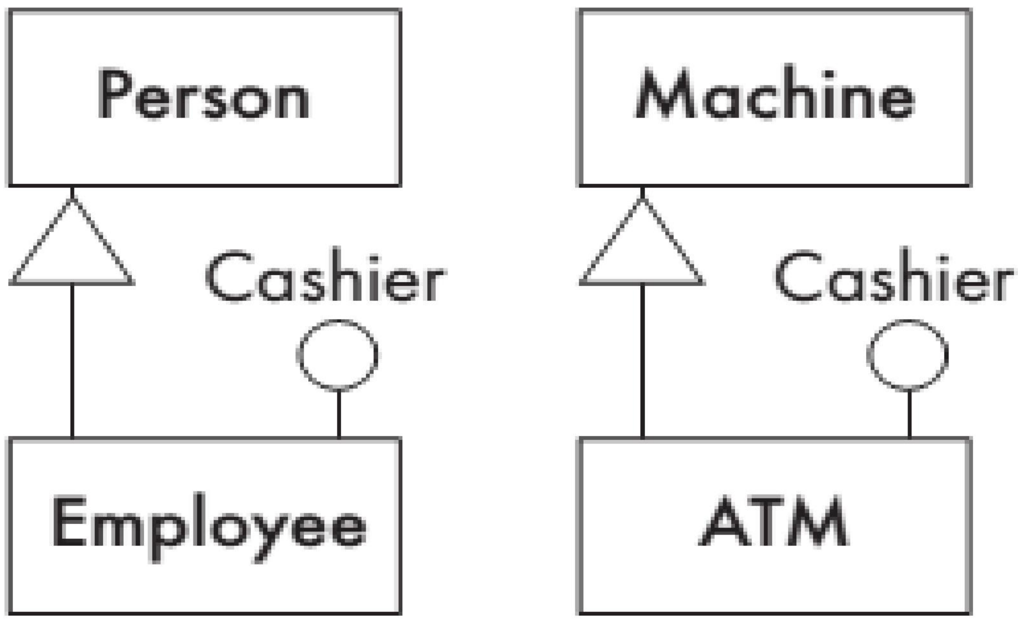
Interfaces

- An interface is similar to a class, except it lacks instance variables and implemented methods.
- It normally contains only abstract methods although it may also contain class variables.
- An interface describes a portion of the visible behavior of a set of objects.

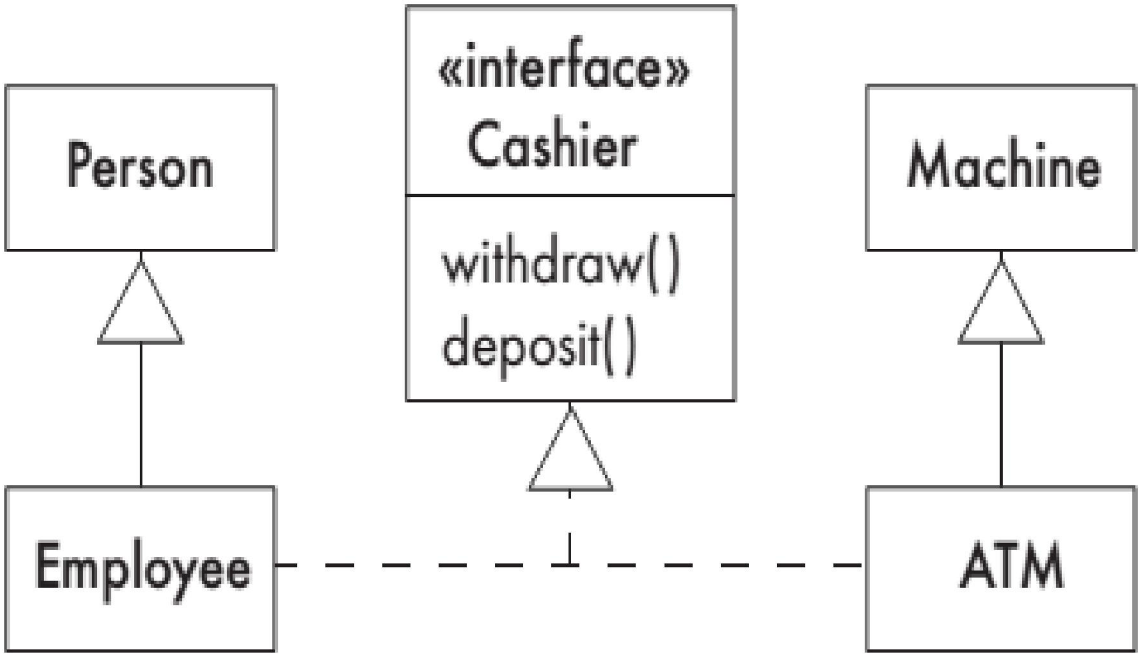
Specifying interfaces

- Two ways:
  - As a small circle (like a lollipop), labeled with the name of the interface.
  - As a class rectangle, with the expression «interface» at the top, and (optionally) a list of supported operations.

Interfaces



Interfaces



«interface»
Cashier
withdraw()
deposit()
Person
Machine
Employee
ATM

Interfaces

- The «interface» notation is an example of a stereotype in UML.
- A stereotype is a way to use some of the standard UML notation (here a class box) to represent something special (here an interface).

Interfaces

- Note that the « and »symbols are called guillemets; they should preferably be written using the special characters available in most fonts, not using pairs of less-than or greater-than signs.

Interfaces

- In some programming languages, interfaces are simply created using super classes containing only abstract methods.
- But interfaces should not be confused with generalizations since the basic relation is not the same.

Generalizations and Interfaces

- Generalization is characterized by an *isa* relationship between a subclass and a super class.
- In the case of interfaces, the relationship between the implementing class and the interface can be described as '*can-be seen-as*'.

Generalizations and Interfaces

- Classes representing bank employees and automatic teller machines; both can be seen as a sort of cashier.
- That is, it is possible to interact with one or the other in order to deposit or withdraw money.

Generalizations and Interfaces

- However, although Employee and ATM share common operations, they have different super classes.
- This means that they cannot be put in the same inheritance hierarchy; therefore an interface called Cashier is used.

Advantage of using interfaces

- A key advantage of using interfaces is that they reduce what is called the coupling between classes.

Constraints, notes and descriptive text

- Very often, in a class diagram, you want to say more than the graphical UML notation readily allows.

Constraints, notes and descriptive text

- There are three ways in which you can add additional information to a UML diagram:
  - Constraints.
  - Notes.
  - Descriptive text.

Descriptive text and other diagrams

- It is highly recommended to embed your diagrams in a larger document that describes the system more fully.
- Such text can explain aspects of the system using any notation you like.

Descriptive text and other diagrams

- It is best not to repeat what is shown in the UML diagrams, but you can highlight and expand on important features, and give rationale for why certain decisions were made.

Notes

- In contrast to the descriptive text, a note is a small block of text embedded in a UML diagram.
- The box has a ‘bent corner’.
- The note can explain a detail, and acts like a comment in a programming language.

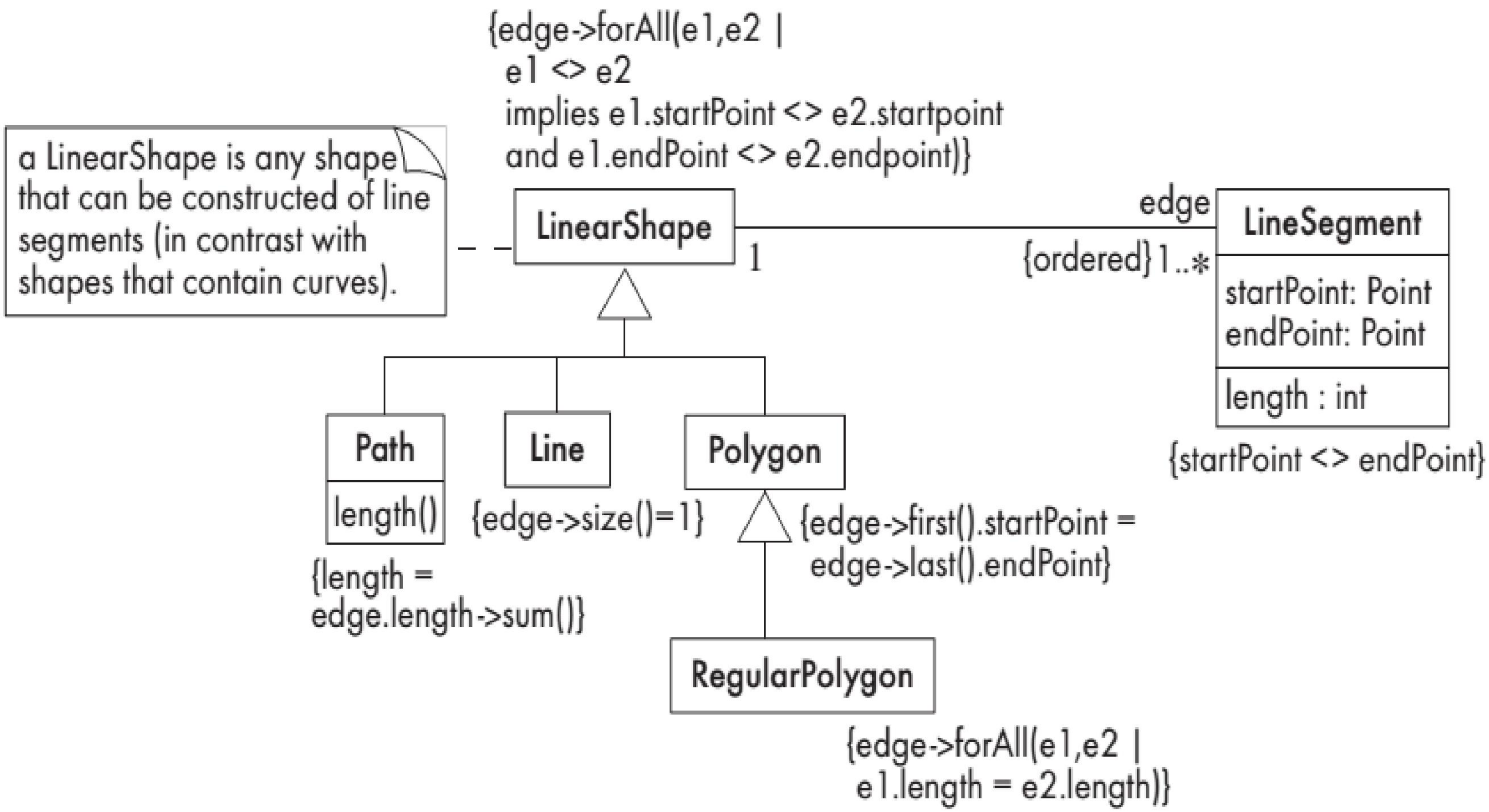
Constraints

- A constraint is like a note, except that it is written in a formal language that can be interpreted by a computer.
- In a UML diagram, a constraint is shown in curly brackets (also called ‘braces’).

Constraints

- A constraint expresses a logical statement that should evaluate to true.
- UML allows constraints to be written in any language supported by a given tool; however, the recommended language is Object Constraint Language (OCL).

Notes and constraints



Steps for drawing class diagrams

- Identify a first set of candidate classes.
- Starting with the most important classes, add any associations and attributes that clearly will be needed.
- Work out the clearest generalizations.

Steps for drawing class diagrams

- List the main responsibilities of each class.
- Based on responsibilities, decide on specific operations that are needed.
- Iterate over the entire process, examining the model to see if you need to add or delete classes, associations, attributes, generalizations, responsibilities or operations.

Steps for drawing class diagrams

- Repeat the previous step as needed until the model is satisfactory

Identifying classes

- A simple technique for discovering the initial set of domain classes for a system is to look at source material such as a description of requirements.
- From this, you extract the nouns and noun phrases.
- A noun phrase is simply a string of nouns, or a noun modified by one or more adjectives.

Identifying associations and attributes

- Once you feel you have a good initial list of classes, it is time to turn your attention to identifying associations and attributes.
- The best way to do this is to start with the class or classes that you think are most central and important to the system.

Identifying associations and attributes

- For each of these, decide on the clear and obvious data it must contain and its relationships to other classes.
- Then work outwards towards the classes that are less important.

Tips about identifying and specifying valid associations

- To find out whether an association should exist, ask yourself if one class possesses, controls, is connected to, is related to, is a part of, has as parts, is a member of, or has as members some other class in your model.

Tips about identifying and specifying valid associations

- You will often find statements of these types by scanning the document from which you extracted the original list of classes.
- As you add each association, remember to specify the multiplicity at both ends and label it clearly.

Tips about identifying and specifying valid attributes

- Attributes can be identified by looking at the description of the system and searching for information that must be maintained about instances of each class.
- Several of the nouns you may have originally identified, but rejected as classes, may now become attributes.

Tips about identifying and specifying valid attributes

- Remember that an attribute should generally be a simple variable – typically an integer or string.
- An attribute should not normally represent a variable number of things (i.e. it should not have a plural name).

Identifying generalizations and interfaces

- There are two ways to identify generalizations:
  - bottom-up and
  - top-down.

Identifying generalizations and interfaces

- The bottom-up approach groups together similar classes, creating a new super class, whereas the top-down approach divides up a complex class, creating new subclasses.
- To use the bottom-up approach, you look for classes that have features in common.

Identifying generalizations and interfaces

- In general, if you find two or more classes that have similar attributes, associations or operations then you should consider creating a common super class.

Allocating responsibilities to classes

- A responsibility is something that the system is required to do.
- The prime responsibility of performing each functional requirement must be attributed to one of the classes, although other classes will likely collaborate with it to help perform the task.

Allocating responsibilities to classes

- In general, it is important to distribute the responsibilities among the classes so that no one class has an unfair share, and hence becomes unduly complex.
- If a class has too many responsibilities then you should examine it to see whether it can be split into several distinct classes.

Allocating responsibilities to classes

- Also, all the responsibilities of a given class should be clearly related to each other and to the attributes and associations of the class.
- If a class has no responsibilities attached to it, then it is probably useless.

Allocating responsibilities to classes

- On the other hand, when a responsibility cannot be attributed to any of the existing classes, then a new class should be created.
- A good way to determine responsibilities is to perform use case analysis.

Allocating responsibilities to classes

- Another good source of information about responsibilities is to look for verbs and nouns describing actions in the system description.

Categories of responsibilities

- Setting and getting the values of attributes.
- Creating and initializing new instances.
- Loading to and saving from persistent storage.
- Destroying instances.
- Adding and deleting links of associations

Categories of responsibilities

- Copying, converting, transforming, transmitting or outputting.
- Computing numerical results, such as the fine on an overdue library book.
- Navigating and searching.
- Specialized work needed by the particular application that does not fit in any of the above categories.

Setting and getting the values of attributes

- It is good practice to make attributes themselves private and to create public methods where necessary to allow access to them.
- This allows the class to have more control over its attributes – it can ensure that they are given only valid values.

Setting and getting the values of attributes

- It also allows you to change the internal design of the class without affecting how users of the class interact with it.
- For some classes, all the responsibilities fall into this category.

Setting and getting the values of attributes

- A Date class, for example, might have no other responsibility than holding the day, month and year of a date, and allowing access to the values of these attributes.

Creating and initializing new instances

- Often, the primary responsibility for creating an instance of a class is given to some other class. (That other class has to call the constructor of the class being instantiated.)

Creating and initializing new instances

- However, Sometimes responsibility is placed directly in the class whose instance is being created (implemented as a static operation).
- There is often a need for several classes to collaborate in this type of responsibility.

Destroying instances

- Like the process of creating instances, this also often requires collaboration with other classes.

Adding and deleting links of associations

- Adding and deleting links of associations, such as recording that a particular professor will teach a certain course.
- Responsibilities of this kind are similar to manipulating attributes.

Adding and deleting links of associations

- However, they are more complex since there is the need to collaborate with other classes to ensure that the bi-directional nature of most associations is maintained properly.

Copying, converting, transforming, transmitting or outputting

- Many applications have responsibilities of these types, which require changing the information to some other form.
- A common example is the toString method in Java, which creates a String representation of an object.

Navigating and searching

- For example, there might be a need for capabilities to look up a particular customer by name, or to find all the customers that match a certain criteria.

Specialized work needed

- Specialized work needed by the particular application that does not fit in any of the above categories.

Object Constraint Language (OCL)

- OCL is a formal language designed to enhance the modeling capabilities of UML.
- It was originally designed exclusively to specify constraints in UML models.
- However, OCL can also be used to specify such things as navigation paths that allow you to formulate queries for information in UML models.

Object Constraint Language (OCL)

- OCL is a specification language, not a full programming language.
- The OCL statements simply specify logical facts (constraints) about the system that must remain true.

Object Constraint Language (OCL)

- OCL statements need not themselves be compiled and executed.
- however, designers must ensure that the code they write always respects the constraints imposed by each OCL statement.

Object Constraint Language (OCL)

- Automatic code generators must also ensure that code adheres to what the OCL statements say.
- A constraint cannot have any side effects; it can only compute a Boolean result and cannot modify any data.

Object Constraint Language (OCL)

- OCL statements in class diagrams can specify what the values of attributes and associations must be.
- They can also state the preconditions and postconditions of operations, although we will not discuss that usage here.

OCL statements

- The simplest OCL statements can be built out of the following elements:
  - References to role names, association names, attributes and the results of operations
  - The logical values true and false
  - Logical operators such as and, or, =, >, <or <> (not equal)

OCL statements

- String values such as: 'a string'
- Integers and real numbers (the latter having a decimal point)
- Arithmetic operations *, /, +, -

Examples of OCL statements

- {startPoint <> endPoint} constrains the two ends of a LineSegment to be different.
- {edge->size() = 1} constrains the number of edges in a line always to equal one.
- {length = edge.length->sum()}constrains the length of a Path to be equal to the sum of all the separate values of edge.length.

End of chapter 5.3