UML for Domain Analysis – Static Modelling

Ananda M. S. Amatya

Department of Computer Science
University of Warwick
Coventry, CV4 7AL, U.K.
Email: Ananda.Amatya@dcs.warwick.ac.uk

Abstract. Domain Analysis involves investigating those aspects of the problem domain that are relevant for the Requirement Specification of the software being developed. UML static models are used to determine the structure of the system during the domain analysis. UML dynamic models are used to determine the interactions that describe the run-time behaviour of the system during the domain analysis. UML models for domain analysis will use UML Class, Object, Sequence, Collaboration, State and Activity diagrams. UML Static models use mainly Class diagrams showing Associations, Aggregations, and Inheritance relationships between domain Classes. Object diagrams give particular run-time instances of corresponding Class diagrams.
Key-words: Aggregation, Analysis, Association, Class, Domain, Dynamic, Inheritance, Object.

1 Introduction

This article describes the UML features that support the *Analysis* of the software being developed. This is commonly referred to as *Domain Analysis* as it involves investigating those aspects of the problem domain that are relevant for the *Requirement Specification*. Use of UML for *domain analysis* results in two types of models, (a) *static model* that deals with the structure of the software, and (b) *dynamic model* that deals with the behaviour of the software. *Static models* capture the classes and their relationship in the software and are given as class diagrams. *Dynamic models* capture the run time behaviour of the software and are given as object interaction (sequence and collaboration) diagrams and state and activity diagrams. Here we concentrate on static modelling and class diagrams. Dynamic modelling will be discussed in another article.

2 Terminology of Use Case Driven Static Modelling

Use Case Scenarios are analysed to obtain domain **Classes** and **Relationships** between them. **Aggregation** and **association** are **class relationships** with **multiplicities** at both class ends. This involves investigation into **event** flows (**scenarios**) and **activity diagrams** that specify **use cases**. **Objects** are (**class instances**) that are arrived at from particular **scenarios**. **Object**

diagrams are **instances** of **class diagrams** with **objects** and **links**, but no **multiplicities**. **Class diagrams** give the **Static Models** of the problem. **Inheritance** is a way of abstracting **generalisation** (Super Class), **specialisation** (Sub Class) relationships.

3 Classes

Class structure is static as it is internal to the **Class**. UML uses a rectangular box with horizontal compartments for (a) **Name** (identifier), (b) **Attributes** (variables), and (c) **Operations** (functions). Its documentation specifies **Class Responsibilities** which record information normally held in **CRC cards** (**Class Responsibility Collaboration**). **Class Stereotypes** are shown to imply characteristics specific to particular application areas. **Objects** are items of a running system that get manipulated during the run. **Real objects** are represented by **system objects**. The **class** of an **object** behaves as the **type** of its **objects**, and determines the characteristic (signature) of an **object**. **Instantiation** creates an **object** of a **class**. While **type operations** are primitives requiring no **implementation**, **class methods** need **implementations** by the developer. **Inheritance** simplifies **class implementations** in system development. **Problem domain objects** are **realized** by **classes** in the system. System software **classes** are files, windows, icons, etc. Information system **classes** are customer, invoice, asset, quote, etc. Technical system **classes** are engine, sensor, display, buttons, etc.

3.1 Finding Classes

Domain experts help to determine **classes** in an application. **Class** name should reflect **domain object** name. Information needed to be stored, transformed or analysed is represented by **persistent objects** (saved in a database). **Temporary classes** are created as the program is started and then lost when the program is terminated. **External systems** with which the system needs to interact will be **classes** in the system. **Libraries, components and patterns** contain **classes** which will be used in a system. **Devices** (e.g., Printer) that a system needs to handle are **classes** in the system. **Organisational parts** in a system are **classes** in the system, e.g., Finance system. **Roles** played by actors give **classes** in the system, e.g., user, operator, customer, etc. **Requirement specification** document gives **domain specific classes** (**noun words**).

3.2 Class Name

The **top compartment** of the rectangle representing a **class** uses a unique identifier for the **class name**. It should be in *bold, starting with a capital letter* and *singular*, a *single* word (no suffix or prefix), unambiguous, meaningful and reflect the domain **object**. **Domain classes** are **stereotyped** (optionally): e.g., `<<entity>>`. An **abstract class** is shown *italicised*.

3.3 Class Attributes

Class attribute types) must be specified whether **primitive** or **class**. **Visibility** may be given (+: `public`, -: `private`, #: `protected`). **Class** scope **attributes** are shown **underlined**. **Enumeration type** is shown by a **property-string**: `{..,..}`. Syntax for **attribute** is given as:

```
visibility name type expression = initial-value {property-string}
```

Property-string used to show **persistence** of **attributes**, is determined by system **requirements**. **Visibility** when not given is assumed **public**. **Attribute** values are assigned at **object** creation. Every **class scope attribute** (cf. Java static variable) has the same value for all the **objects** of the **class**. **Attributes** are **elided** when not needed for the particular context.

3.4 Class Operations

Class operations manipulate **class attributes** and may perform other **operations** on its **objects**. **Operations** (sometimes, referred to as functions) can be applied onto its *own* **objects** only. **Return types** give **types** of returned **values**. **Operation names** when **overloaded** are differentiated by **parameters** passed. **Operation parameters** have **types**, and may have default values. **Operation** syntax should be as follows:

```
visibility name (parameter-list): return type expression
{property-string} name: type expression=default-value
```

Visibility and scope of **operations** may be restricted (as for **attributes**). **Static operations** are referred to as **class scope operations**, as they apply to all the **instances** of the **class**. They may be called directly using **class name** and without creating any **instances**. **Class scope operations** can access only **class scope attributes** of the **class**. **Class scope operations** carry out **generic operations** of the **class** e.g., creating **objects**, finding **objects**, etc. **Operations** are applied to an **object** of a **class** by calling it from that **object**. **Operations** hold the **services** offered by the **class**; they form the **interface** of the **class**. Default values of parameters are as supplied in **class** definition. **Operations are implemented by methods**; **methods are NOT operations**. **Operations** are specified (in the documentation file) using **preconditions**, **postconditions**, **algorithms**, and **effects on objects**. **Preconditions** must be true **before** the **operation** can be applied. **Postconditions** must be true **after** the **operation** has been completed. Operation properties are **documented** in a file, and may be accessed by tools. **Persistent classes** will have their **objects** stored in files so that they may be accessed even after the program has finished. **Class scope operations** of a persistent **class** perform the **storing & retrieval** of **objects**; some such **operations** are: `store(), load(), create()`. Persistent **class operations** are shown by **{ppersistent} tagged** values.

3.5 Attribute & Operation Types

Type expression of **attributes**, **operations** and **parameters** may be primitive or **class name**. **Primitive type operations** are already implemented and these **implementations** cannot be changed. **Class type operations** must be implemented by **methods** before use. Primitive **type attributes** hold **values** while **class type attributes** hold **references**.

3.6 Class Structure

Class attributes are obtained from domain **object** properties during Static Modelling. **Class operations** are obtained from **interaction diagrams**, and also **independently** (to help another **operation**), mainly during Dynamic Modelling. **Class structure** gives properties and behaviour of **objects** belonging to it. A **class** has **name**, **attributes**, **operations** and **responsibilities**.

3.7 Class Details (Eliding)

Classes are shown with only those details that are relevant. In (**Figure 1**) different details are shown: (a) Car **Class**, (b) Car **Class** with **attributes**, (c) Car **Class** with **attributes** and their **types**, (d) Car **Class** with **attributes**, their **types** and their visibility (+: public, -: private, #: protected) (if not shown visibility is still defined), and (e) Car **Class** with **attributes** (**types, visibilities**) and **operations** (parameters, return **types, visibilities**). **Class scopes** are shown **underlined**; all **object** instances will have the same value for the static **attribute**. **Enumeration type** may be used for **attribute type**, e.g., `Status = {unpaid, paid}`.

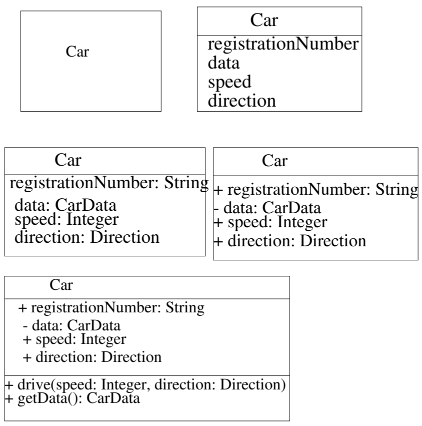

Fig. 1. Car Class

3.8 Example: Invoice Class

Invoice class (**Figure 2**) has **attributes**: visibility, name, type, and default value, class scope (static) **attribute**: number of invoices (all **object** instances have the same value), and **Enumeration types** used for **attribute types** (e.g., `Status = {unpaid, paid}` in the **attribute** status).

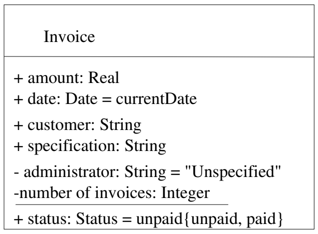

Fig. 2. Invoice Car

3.9 Stereotype Classes

Object patterns determine **classes**: (a) **entity classes** that are domain specific (noun phrases to accomplish responsibilities), (b) **boundary classes** that handle communications between **actors** and **use cases** interfaces, and (c) **control classes** that handle sequencing in **use cases**. See **Figure 3**.

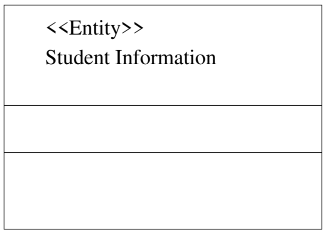

Fig. 3. Stereotyped Class Name

3.10 Documenting Classes (Data Dictionary)

Documentation of **Name**, **Stereotype**, **Attributes** (**Type** and **export** control and **Operations** (**Responsibility**) specify the class.

4 Class Relationships

There are various ways **classes** can be related in UML. (a) **Association**, (b) **Dependency**, (c) **Inheritance: Generalisation/Specialisation**. **Association** is a general connection between **classes**. A **link** is an **instance** of an association. **Objects** are connected by **links**.

4.1 Association

Association is normally a **bidirectional** semantic connection between two classes. Both objects in an instance being aware of each other. **Navigable association** is **unidirectional**, an arrowhead showing the direction of navigation. One class can use another class only if the **association** between them is navigable in that direction. **Aggregation** is a specialised **association** in which one class is a **part** of another.

4.2 Naming Relationships

A relationship is **named** to convey its meaning. **Bidirectional association** (default) has names in both directions (shown by an arrow next to each name). **Unidirectional association** has a name only in the association direction. These are referred to as **association** names. These names reflect the relationships in the problem domain (e.g., **verb words** in the problem spec).

4.3 Multiplicity Indicators

Multiplicity (**Figure 4**) is specified (a) **exactly** (e.g., 1:1), (b) **zero or more** (e.g., 0..*), (c) **one or more** (e.g., 1..*), (d) **zero or one** (e.g., 0..1), (e) **specific range** (e.g., (5,6,7,8), 5..8), and (f) **combination** (e.g., (4,5,6,7,9), 4..7,9). **Default multiplicity is 1**.

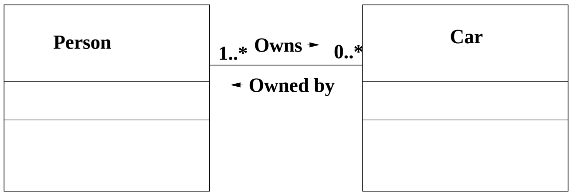

Fig. 4. Association Names & Multiplicities

4.4 Role Names

Role names identify the roles at the ends of an association. One or both classes in the **association** are given **role** names. For example, (**Figure 5**) **association** between Car and Person with **association names** "drives" and "driven by", and **role names** "company Car" and "driver".

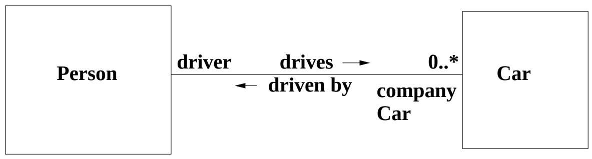

Fig. 5. Association Names, Role Names & Multiplicities

4.5 Navigable Association Example

A person may have 0 or more cars, but there is no information about how many persons can own a car (**Figure 6**). Used in specification and implementation constraints.

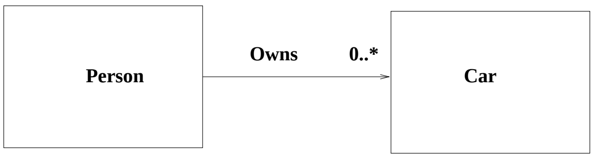

Fig. 6. Navigation

4.6 Aggregation

Aggregation is a special (**stronger**) form of **association**. It signifies a **whole-part** relationship, e.g., "consists of", "is part of", "has a", "contains". Sometimes it is referred to as **normal aggregation**. Parts may be **shared** by more than one **aggregate** relationship, such that parts may remain even after an **aggregate** whole has ceased to exist. An aggregate has **name, direction, multiplicities, roles, and navigation** as in **association**.

Shared Aggregation Components may belong to more than one **aggregate** by being shared. For example, a person may belong to more than one team in an **aggregation** relationship between Team and Person (**Figure 7**).

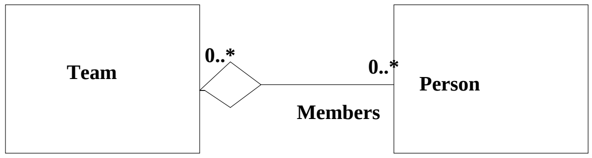

Fig. 7. Shared Aggregation

Composition Aggregation **Composition aggregation** is a **stronger** form of **aggregation** in which **a part object** may belong to only one **whole object**. Parts **live and die with the whole** object. Any **deletion of whole cascades down to parts** The **multiplicity on the whole side must be zero or one (0..1)**, but the multiplicity on the part side may be any (0..*).

A **composition aggregation** forms a **tree of parts** and may be shown as one solid diamond for each part (**Figure 8**).

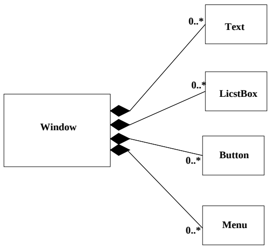

Fig. 8. Composition Aggregation: 1 Diamond per Part

Alternatively one solid diamond is used as the root of the tree of parts (**Figure 9**).

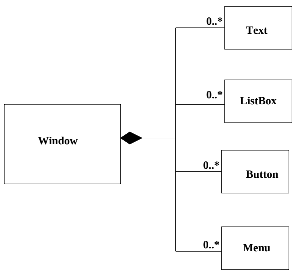

Fig. 9. Composition Aggregation: Tree

A third (intuitively better) representation is to show all parts contained inside the whole (**Figure 10**).

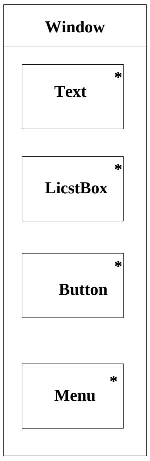

Fig. 10. Composition Aggregation: Parts inside Whole

Role names appear only at the parts ends (**Figure 11**).

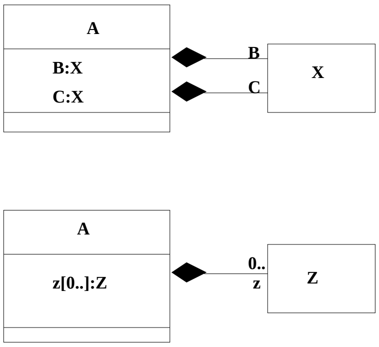

Fig. 11. Composition Aggregation

4.7 Qualified Association

Qualifier used to distinguish among the set of objects at the **many end** of a **many to 1** or **many to many association**. Converts a single **many to 1**

relationship into **many 1 to 1** relationships, by distinguishing each **association** by a qualifier id. Used for uniquely identifying a class by giving the **qualifier** id of the **association**. Can subset the objects at **many end** into groups, each group being associated to a **qualifying group** id at the **1** end. This converts a **1 to many** relationships to **many 1 to many** relationships.

In the figure example (**Figure 12**), figure id will identify a unique figure or a group of figures depending on whether the qualifier is for each object or each group of objects.

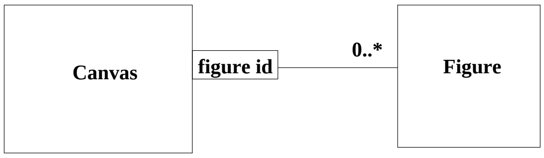

Fig. 12. Qualified Association

4.8 Or-Association

Mutually exclusive association, in which **association** is with one or the other, but not both. **Dashed line labelled "or"** between the alternatives. **Or-association** (**Figure 13**) is used to group classes into choice classes (**exclusive or**).

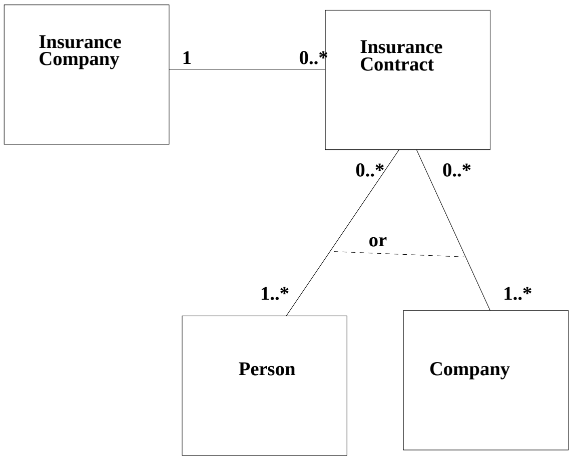

Fig. 13. Or-Association

4.9 Ordered Association

When multiplicity greater than 1, **ordered** constraint is used if ordering of the **association** is desired (list, instead of the default of unordered set). Labelled `{ordered}` at the **many end** of the **association** (**Figure 14**). Order needs to be preserved when new objects are added. Ordering criterion (and whether increasing or descending) needs to be specified. **Default is NO order**, i.e., new object added anywhere, usually at the end of a list.

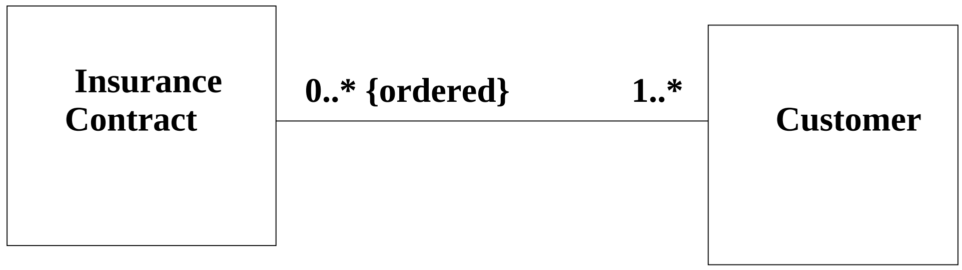

Fig. 14. Ordered Association

4.10 Reflexive Relationships: Recursive Association

Multiple objects **belonging to the same class** communicate with each other (* means 0 or more) (**Figure 15**).

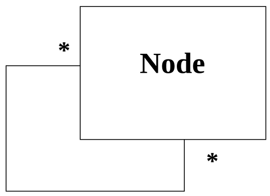

Fig. 15. Reflexive Relationship: Recursive Association

Reflexive relationship makes semantic connection between **many objects of the same class**. **Objects** of the class are linked to each other by **links**. There may be **one or more links** to the same object (**Figure 16**).

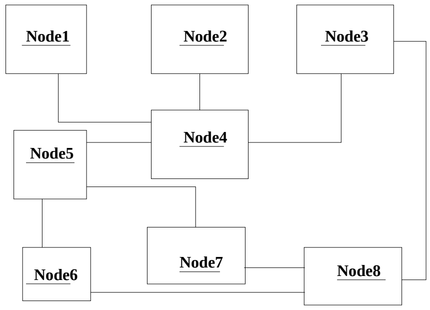

Fig. 16. Recursive Association

4.11 Association Class

Association relationship with **attributes** and **operations** is regarded as a **class**. Dashed line to **association** line. For example, (**Figure 17**) Employment **association** between Person and Company with **attributes**: date, salary and **operations**: changeContract(), increaseSalary(). Another example would be Loan as a **association class** with **attributes**: book name, borrower name, return date, and **operations**: extendLoan(), cancelLoan(), etc.

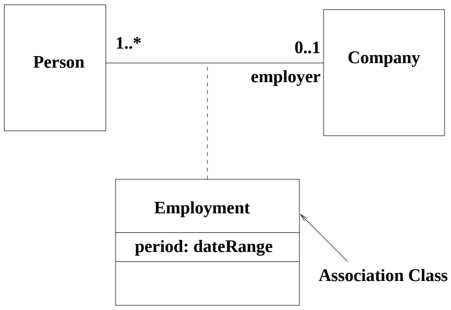

Fig. 17. Association Class

4.12 Ternary Association

Association between **three** classes shown by connecting them to a **Diamond**. May have **association class** connected at one corner of the diamond (**Figure 18**). Examples: **Association** between Insurance Contract, Person, and Insurance Policy; Doctor, Patient, and Drug. No qualifier allowed in **ternary association**; Can have an **association** class, e.g., Treatment (attributes: patient, doctor, drug, date; operation: startTreatment(), postponeTreatment()).

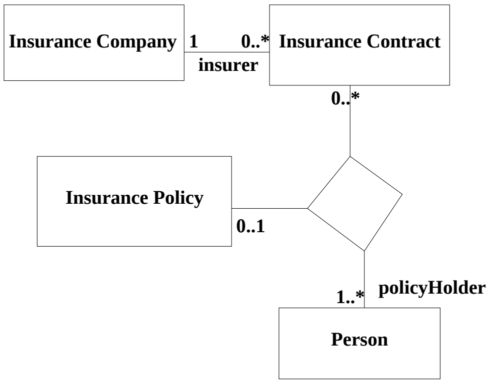

Fig. 18. Ternary Association

5 Class Diagram

Gives the structure of classes in a system or subsystem and **relationships** between them. It gives a **static** view of the system.

5.1 Insurance Business

For example, an insurance business class diagram (**Figure 19**) showing: (a) one insurance company having many insurance contracts (0..*), (b) one insurance customer having many insurance contracts (0..*), (c) an insurance contract being related to one insurance company (1), and (d) an insurance contract being related to one or more insurance customers (1..*), (e) an insurance customer has zero or more insurance policies (0..1).

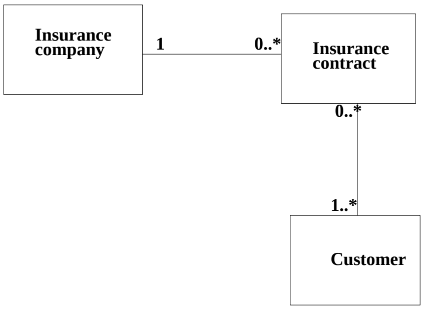

Fig. 19. Insurance Business Model

5.2 Order Processing

An object **collaboration diagram** for Order Processing shows the order being entered and processed (**each order line**) by **delivering the item** if in stock and **reordering if not in stock** or if **order level reached reorder limit** (**Figure 20**). **Order** contains a pointer to customer, and order line a pointer to product. This interaction diagram leads to the Class Diagram (Model).

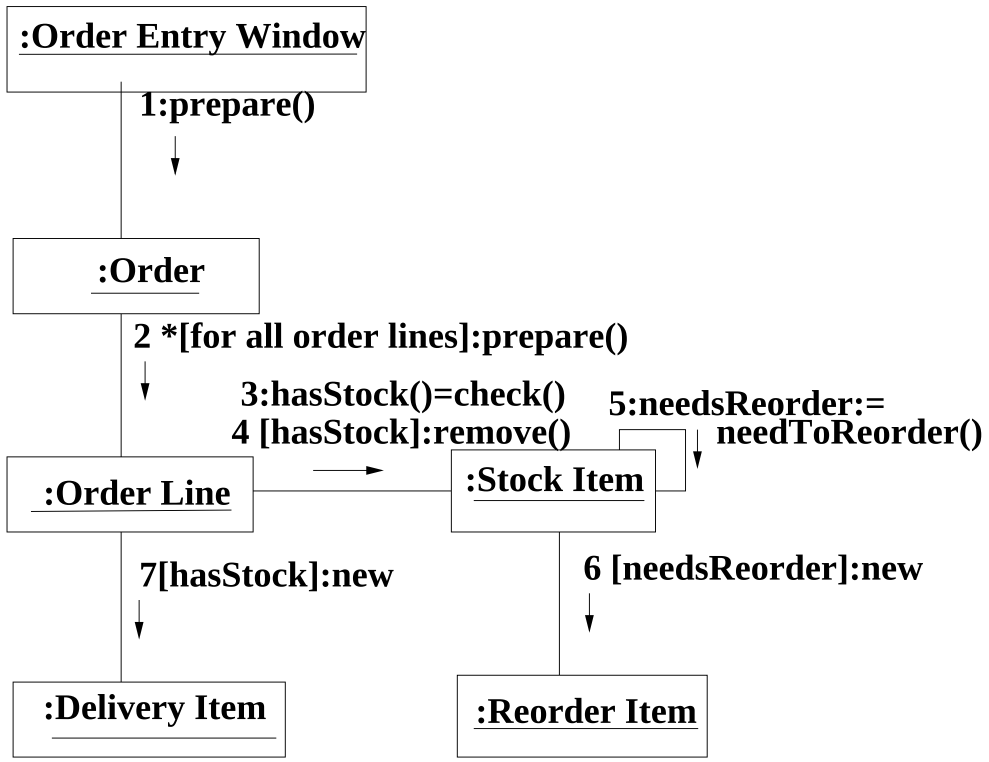

Fig. 20. Order Processing Collaboration diagram

5.3 Navigability

Navigable from an **order to the customer credit rating**. Customer credit rating is checked only for a given order. **Cannot find order from a customer credit rating** (**Figure 21**).

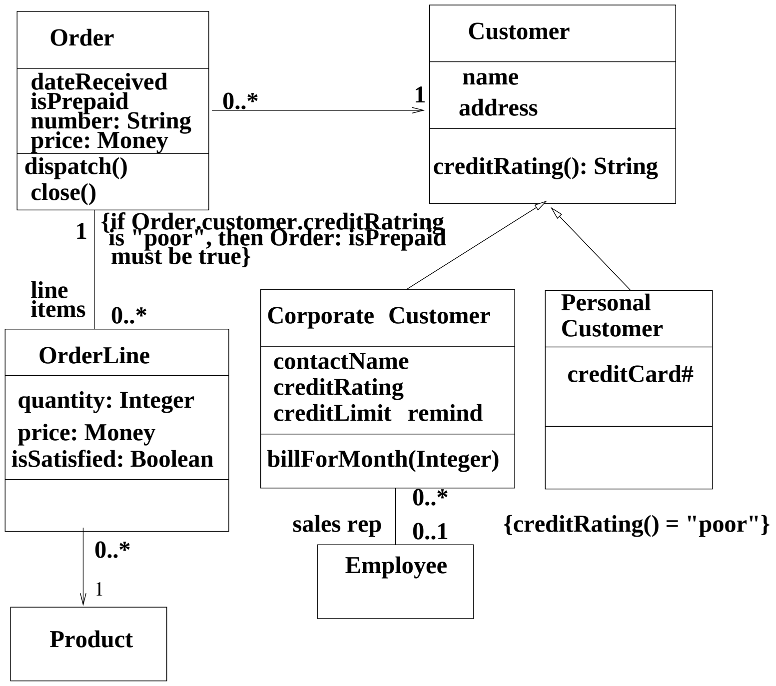

Fig. 21. Order Processing

6 Object Diagram

Object instances giving **snapshot** of class diagram with labelled rectangles containing object name **underlined** (**Figure 22**).

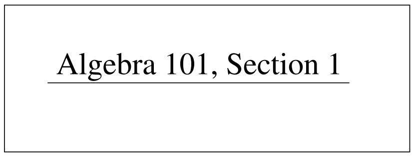

Fig. 22. Object Diagram

7 Object Diagram vs Class Diagram

Object instances are connected by **links**. Object diagram is an instance of the corresponding class diagram. **No multiplicities** are shown as each link is between two objects. **Underlined** class name is used for a corresponding object name; class name may be **elided**; **anonymous** object name is used when the object name is not relevant; A separator ":" is used the object name and its class name. (**Figure 23**)

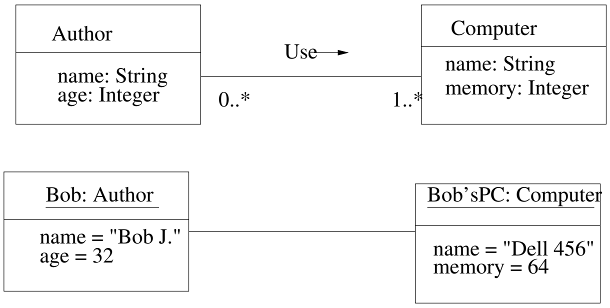

Fig. 23. Object Diagram Vs Class Diagram

7.1 Object Diagram with Roles

Roles are parts of **collaborations**. They are represented like classes with **role name** followed by **colon** followed by **type**. **Not underlined**. **Instances** are **underlined** and shown with optional **name**, followed by optional **slash**, followed by **list of roles**, **colon** and **list of types**. See **Figure 24**.

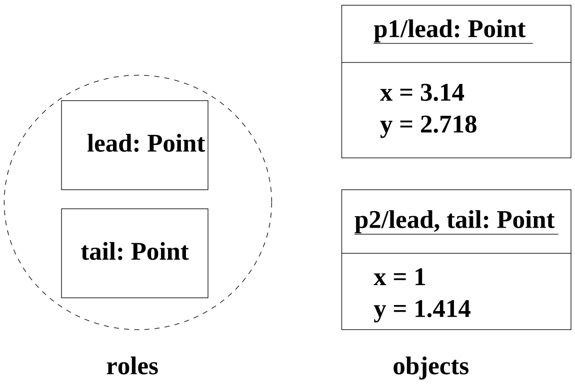

Fig. 24. Object Diagram with Roles

8 Generalisation (Inheritance) Relationships

A **subclass inherits attributes**, **operations** and **associations** from its **superclass**. Usually further information is **added** to the subclass. An **instance** of a **subclass** can be used instead of an **instance** of its **superclass**. **Inheritance** is a **class (type)** relationship not a **object** relationship. A **subclass object inherits** a **superclass object** characteristics through the **inheritance** relationship between the **subclass** and the **superclass**. **Public superclass operations** and **attributes inherited** remain **public** in the **subclass**. **Private superclass operations** and **attributes inherited** remain **private** in the **subclass** and so inaccessible to the **subclass**. **Protected superclass operations** and **attributes** remain **protected** and so accessible to the **subclass**. UML uses **-** for **private**, + for **public** and # for **protected**. A **class** in a hierarchy may be a **superclass** as well as a **subclass**. See **Figure 25**

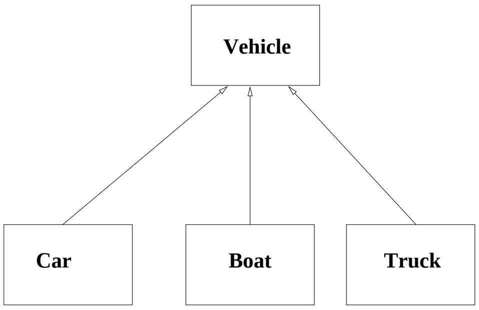

Fig. 25. Vehicle Inheritance

Another representation is to use the tree structure (**Figure 26**)

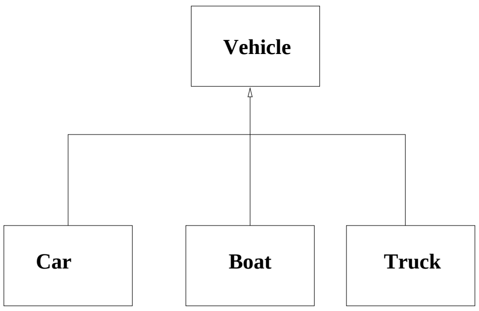

Fig. 26. Inheritance Tree

Figure 27 shows an inheritance hierarchy.

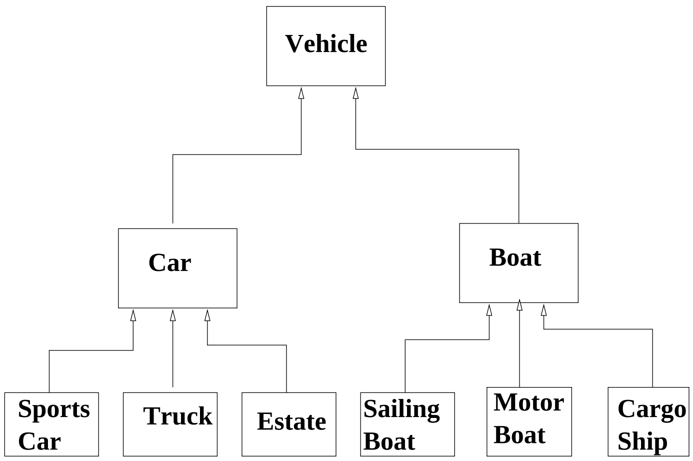

Fig. 27. Inheritance Hierarchy

9 Abstract Class

An **abstract class** cannot have any **objects** (or **instances**), but it can have **subclasses** which may be **abstract** or **concrete**. UML uses **tag** `{abstract}` below the name in the name compartment of the **class**. Also the name is given in *italic*. An **abstract class** has normally **abstract operations**. An **abstract operation** has only signature but no implementing method in the class where the operation is specified. In UML an **abstract operations** name is given in *italic* and its signature is followed by the tag `{abstract}`. A class with at least one **abstract operation** must be an **abstract class**. The **abstract operations** are implemented in **subclasses**. Thus `drive()` **abstract operation** of vehicle is implemented in its **subclasses** car, boat, truck. **Concrete classes** can be instantiated into **objects**. All their **operations** have implementations in the **class**. However these **operations** may be redefined in its **subclasses**, though their signatures must remain the same. All instances of a **subclass** will use the **operations** (with their new definitions if changed). **Subclasses** allow new **attributes**, **operations** and **associations** to be added to those inherited from the **superclass**. This possibility of using newly defined **methods** in **subclasses** for the **operation inherited** from the **superclass** is called **polymorphism**. **Discriminators** specify the basis for **inheritance** , e.g., the characteristics **inherited**. For example **vehicles** may be **subclassed** on the basis of **methods** used for propulsion or on the basis of number of wheels.

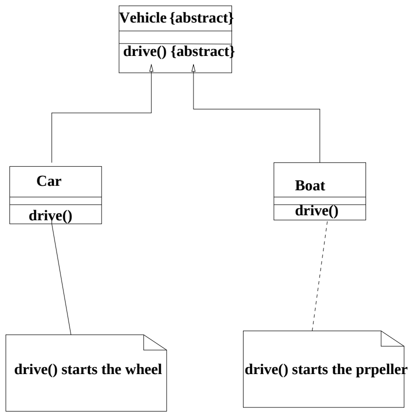

Fig. 28. Inheritance Relationship: Abstract Operations

Figure 28 shows Abstract Operations in inherited class.
Figure 29 shows Association to inherited class.

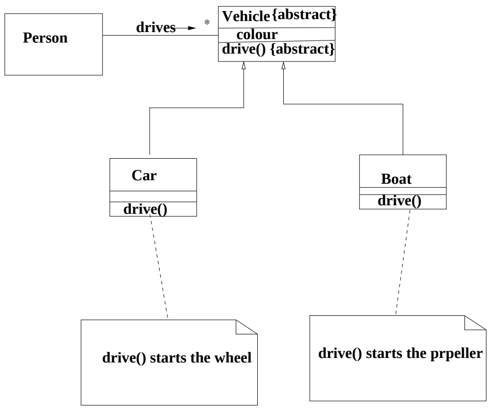

Fig. 29. Inheritance Relationship: Association

Figures 30 & 31 show Inherited attributes.

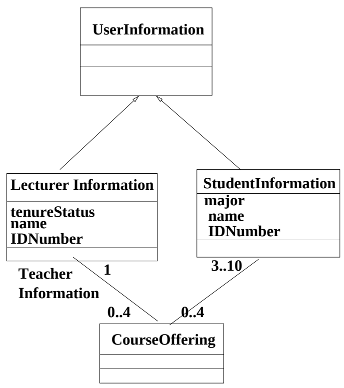

Fig. 30. Inheritance Relationship: Attributes

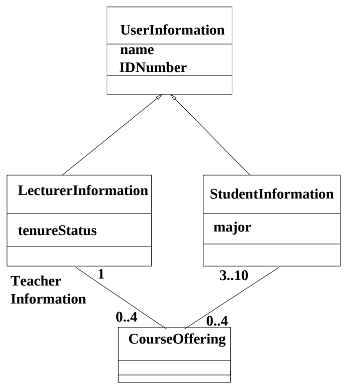

Fig. 31. Inheritance Relationship: Attributes

10 Inheritance vs Aggregations

Inheritance separates commonality from specifics. **Aggregation** shows composite & containment relationships. See **Figure 32**.

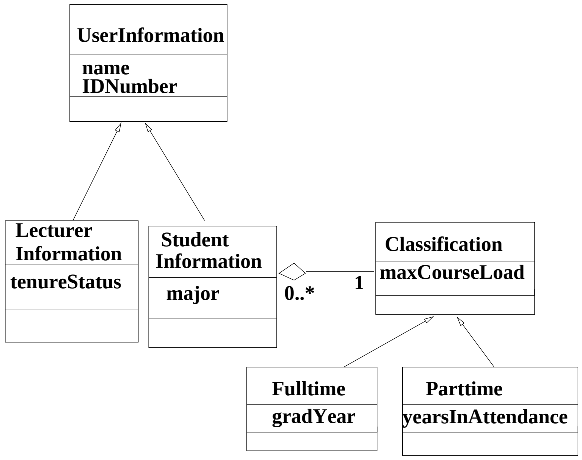

Fig. 32. Inheritance & Aggregates

10.1 Generalisations: Inheritance Constraints

4 types of constraints on generalisations: `{complete}`, `{disjoint}`, `{incomplete}`, `{overlapping}`. These are constraints on the discriminator for the inheritance relationship. **Figure 33** shows Inheritance Constraints.

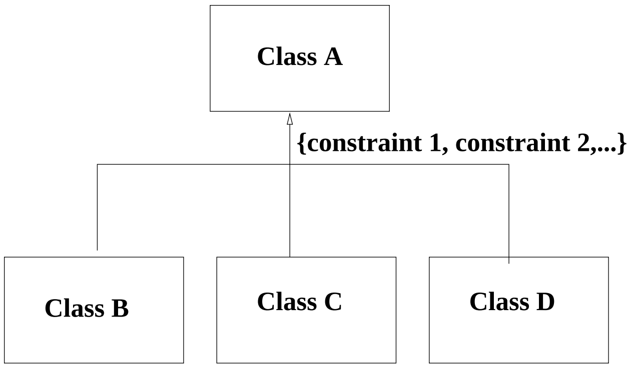

Fig. 33. Generalisations: Inheritance Constraints

Figure 34 shows alternative way of representing Inheritance Constraints.

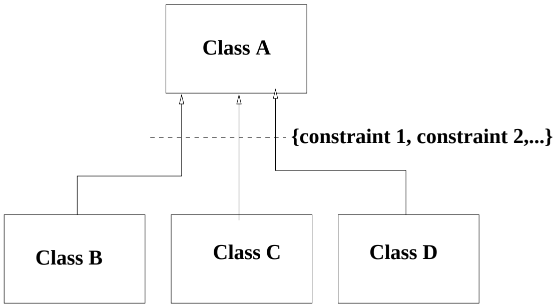

Fig. 34. Generalisations: Inheritance Constraints

10.2 Multiple Inheritance: Overlapping Classes

Can have **multiple inheritance** with a common **superclass**. Default: not overlapping. **Figure 35** shows Overlapping Inheritance Constraint.

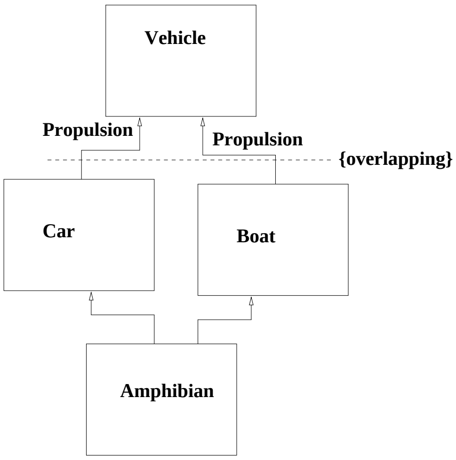

Fig. 35. Class Diagrams: Overlapping

10.3 Disjoint Generalisation

Can NOT have **multiple inheritance** with a common **superclass**. This is the default. **Figure 36** shows Disjoint Inheritance Constraint.

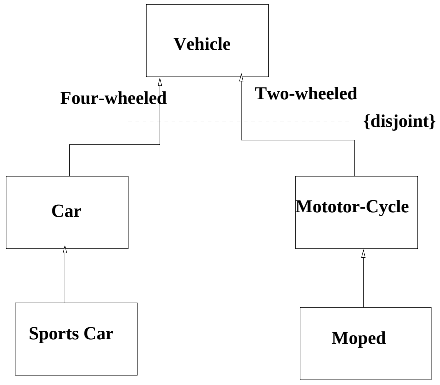

Fig. 36. Disjoint Generalisation

10.4 Complete Generalisation: No other Sub-class

Can NOT have any further subclasses at that level. The default is `{incomplete}`, i.e., more subclasses are present. **Figure 37** shows Complete Inheritance Constraint.

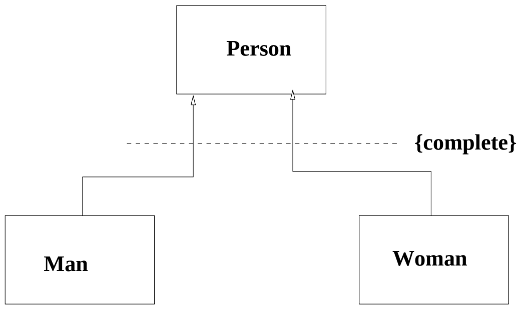

Fig. 37. Complete Generalisation: No other Sub-class

11 Derived Association

Derived attribute is denoted by `/derived {attribute} name`, e.g., `/age`, `/VIP Customer`. **Figure 38** shows Derives Association.

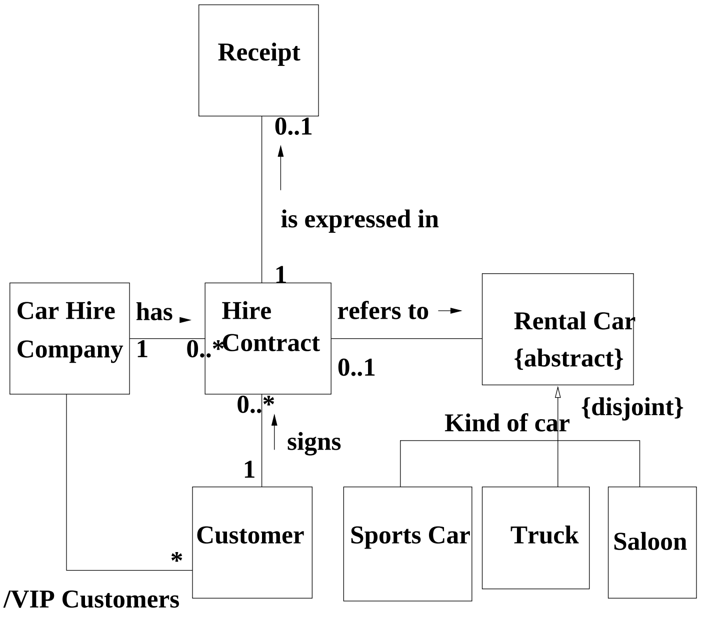

Fig. 38. Derived Association

12 Constrained Association

Or-association, `{ordered}` association are forms of constrained **associations**. A constrained **association** may be shown as a `{subset}`, e.g., **Association** `Party leader of` is a `{subset}` of the **association** `Member of`. **Figure 39** shows Constrained Association.

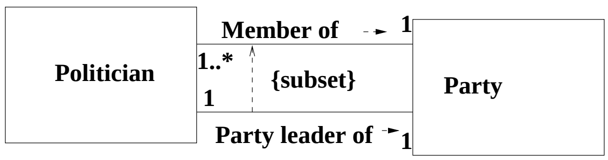

Fig. 39. Constraint Association

12.1 Attributes Constraint

Static **attribute** `number of invoices` is updated automatically as each invoice **object** is created. Enumerated type value range is given as a `property-list`, **Figure 40** shows Constraint in Attributes.

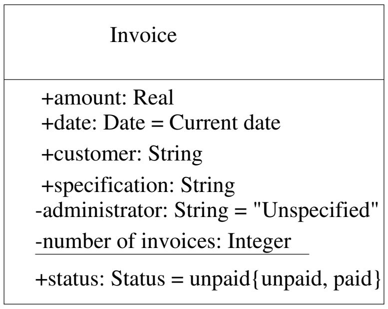

Fig. 40. Attribute Constraint

13 Derived Attribute

OCL is used to express UML constraints syntax and semantics. `/profit` is the **derived attribute**. **Derivation formula** is shown at the bottom of the **class diagram**. Derived **attributes** are always calculated and never read from previously stored values. **Figure 41** shows Derived Attributes.

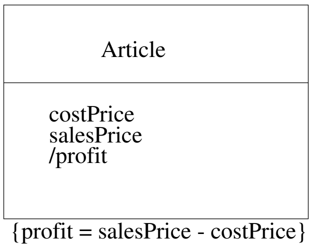

Fig. 41. Derived Attribute

14 Bibliography

1. Booch, Grady and Rumbaugh, James and Jacobson, Ivar, *The Unified Modelling Language User Guide*, Addition-Wesley, 1998.
2. D'Souza, Desmond F. and Wills, Alan Cameron, *Objects, Components and Frameworks with UML The Catalysis Approach*, Addison-Wesley, 1998.
3. Eriksson, H. and Penker, M., *UML Toolkit*, John Wiley, 1998.
4. Fowler, Martin and Scott, Kendall, *UML Distilled: Applying the Standard Object Modelling Language*, Addison-Wesley, 1998.
5. Rumbaugh, James, Jacobson, Ivar and Booch, Grady, *The Unified Modelling Language Reference Manual*, Addison-Wesley, 1998.