Chapter 17

Domain Analysis and Modelling with UML

Recommended Reading

Timothy C. Lethbridge and Robert Laganière. *Object-Oriented Software Engineering*. McGraw Hill, Shoppenhangers Road, Maidenhead, Berkshire, SL62QL, second edition, 2005. Practical Software Development using UML and Java.

Chapter 5,8.1

Ian Sommerville. *Software Engineering*. International computer science. Addison-Wesley, ninth edition, 2010.

Section 5.3, 7

Simon Bennett, Steve McRobb, and Ray Farmer. *Object Oriented Systems Analysis and Design Using UML*. McGraw Hill, Shoppenhangers Road, Maidenhead, Berkshire, SL6 2QL, third edition, 2006.

Chapter 7 and Appendix 3

Chapter 3 introduced the elaboration of *use cases* and *domain models* as key methods for documenting the requirements for a system. User stories (described in Chapters 13 and 15) are used to specify the functionality of a system, i.e. what the system will do. This chapter explores techniques for capturing and documenting the things in the problem domain that need to be represented in a software system. The overall term for this activity is called *domain analysis*.

17.1 Systems, Models and Modelling Notations

Before we go any further, we should get our terminology straight. A *system* is a mental construct, denoting some aspect of the world with state and/or behaviour. A system is distinguished from its environment by a system boundary. We may be also be able to discern the purpose of systems we identify, for example, the purpose of a car engine is to provide kinetic electric power to the car's wheels.

A *model* is an abstraction of some aspect or perspective of a system. Models can serve two purposes in software engineering:

Reasons for modelling (441)

- analytical models are used to describe and better understand the properties of existing systems; and
- constructive models are used to describe proposed systems, in order to predict their future properties. In some cases it may also be possible to derive an implementation directly from a sufficiently detailed constructive model.

In addition to reasons for modelling, we can also distinguish between a modelling method and a notation:

Modelling notations and methods (442)

- modelling *methods* are descriptions of systematic procedures that can be followed to construct a particular type of model of a system; and
- modelling *notations* are standardised syntax used to represent the results of applying modelling methods.

Note that modelling methods are typically for construction of a particular kind of model (analytical or constructive), but that notations may be used for one or either.

Categorising modelling notations and methods (443)

Modelling notations can be categorised in a variety of ways:

- by system aspect
  - structural
  - behavioural

  or both; and
- by format
  - informal
  - graphical
  - algebraic.

In summary, *analysis* is concerned with documenting the features of entities in the problem domain and the relationships between them. *Design* is concerned with translating the domain representation into a system architecture suitable to be implemented in an object oriented programming language. So, domain models capture the essential elements and features of the problem domain and design models document the software components to be implemented. This chapter is concerned with early stage *object oriented analysis*, but we will return to object oriented design in Part III.

17.2 The Unified Modelling Language

Graphical notations are particularly common in software engineering, since they are useful in managing the complexity of software problems. The *Unified Modelling Language* (UML) is a graphical modelling notation derived from three object oriented modelling methods:

A little history (444)

- object oriented design [Booch, 1982];
- the object modelling technique, including state and class diagrams [Rumbaugh et al., 1991]; and
- the ObjectOry method including component and use case diagrams, as well what became the phases in RUP [Kruchten, 1997].

UML is said to be a *semi-formal* notation, since it has a well defined syntax and semantics, with some informal extensions. Some of the features of UML are:

Graphical modelling with UML (445)

- a notation used in many modelling methods;
- well defined semantics, implemented in different syntactic flavours;
- extension mechanisms via *stereotypes*; and
- an algebraic formal constraint language (OCL) supporting design by contract.

Figure 17.1 illustrates some of the different types of diagram provided in the UML. UML diagram types are broadly categorised as either structural or behavioural, with behavioural diagrams having a further sub-category of interaction diagrams.

UML diagram types (446)

However, the figure illustrates that the different diagram types exist on a spectrum from purely structural to purely behavioural, with those types in the middle exhibiting elements of both. We will principally focus on the use of class diagrams in this chapter, for the purpose of developing object oriented models of the problem domain for a software application. Other diagram types may be introduced throughout these notes, as UML can be used for describing many different software engineering concepts. For example, Chapter 25 shows how to use activity diagrams to represent the flow of events in a system to develop test

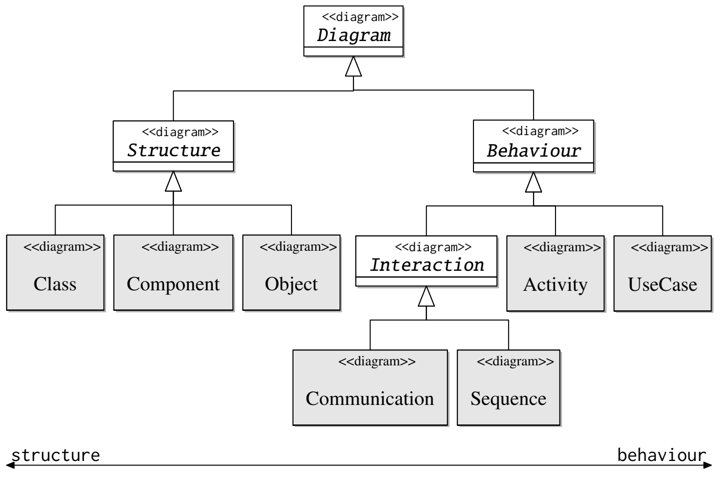


Figure 17.1: UML diagram type hierarchy

cases and the use of component diagrams to represent a system connected to a test harness.

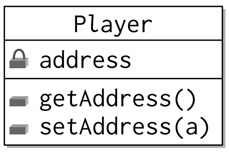
One benefit of an object oriented approach to modelling is that the same notations can be adopted at several stages of software development. However, the application of object oriented techniques are slightly different at each stage in the process, as illustrated in Figure 17.2.

Object oriented analysis and design (447)

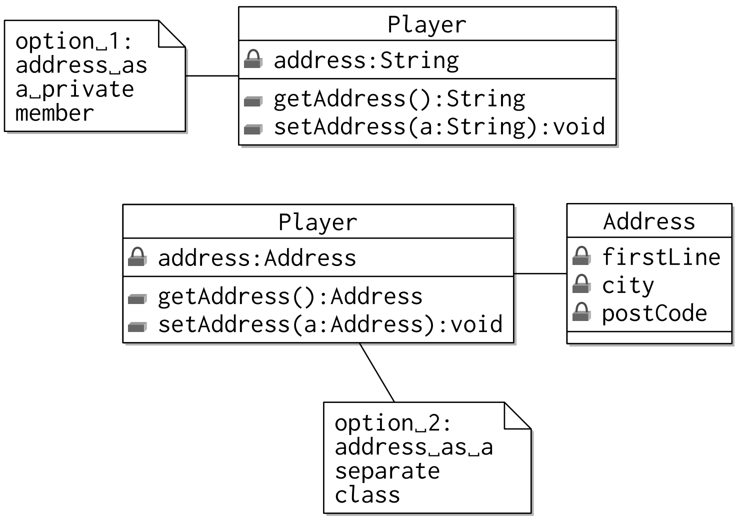
during analysis an object oriented approach is used to identify the entities in the domain, as referred to in the original problem description and user stories. These are documented as a high level design which identifies each domain entity type, their attributes and the relationships between them. Figure 17.2(a) documents the features of the `Player` class, identified for the Sportster system so far. At this stage, it is identified that the player has an address, however, further details are omitted.

during design the class models are refined and extra detail is added. Design alternatives are also identified and resolved at this stage. Further classes may be added and the architecture of the system gradually becomes established. Further refinements may be made to the design later on to accommodate the constraints of external libraries or platforms, or to leverage *design patterns*. Figure 17.2(b) shows two different design alternatives for the `Player` class. In option 1, the author is represented as a class attribute. In option 2, a player's address is represented as a separate class, with its own attributes. Option 2 is selected, to allow for the possibility of a many to many relationship between players and addresses.

during implementation the chosen design is translated into an executable object oriented system, using a language such as C++, Java, Python or C#.



(a) domain analysis



(b) design choices

```
public class Player
 implements Rateable{
  private Address address;

  public Address getAddress(){
   return address;
  }

  public void setAddress(Address address){
   this.address = address;
  }
```

(c) implementation

Figure 17.2: object oriented analysis, design and implementation

Figure 17.2(c) shows the implementation of design option 2 in Java. CASE tools such as Umbrello or StarUML can be used to translate class diagrams into object oriented program *stubs*, containing the attribute and method signatures of the class.

17.3 Developing Object Oriented Domain Models

A key challenge in software engineering is to understand and accurate represent the problem domain and then translate the resulting requirements into a system design.

Stages of modelling (448)

During problem *domain analysis*, activities are undertaken to:

- identify key domain artifacts, their attributes and relationships that must be represented in object oriented system;
- allocate behavioural responsibilities to domain entities; and
- provide initial inputs to software design processes.

During design, the domain model is translated into a software design model which:

- documents the architectural decisions made by the software engineer; and
- identifies the sub-systems and components that constitute the overall system.

Domain models document the aspects of the problem domain that need to be represented in the software system. Design models explain the structure of an object oriented system. They may contain many elements that are identifiable in the domain model. However, these will be augmented with additional features to support functionality in the software. In addition, design model elements may be reorganised to improve software design quality, as we will see later on.

Domain modelling process (449)

The domain modelling process is approximately:

1. Analyse the available problem domain documentation
2. Develop an initial collection of 'high priority' classes with attributes
3. Identify relationships between classes
4. allocate types to attributes and extend the set of classes with new types as necessary
5. Simplify model by applying appropriate generalisations and abstractions
6. Identify and allocate further responsibilities and behaviour where appropriate using behavioural diagrams

artifact	**look for**
classes	the nouns (things) in the environment
attributes	simple characteristics of nouns that can be represented as a primitive or 'utility' type
relationships	ownership statements (has, part of, contains) between subject and object nouns, complex characteristics of alredy identified nouns
operations	descriptions of behaviour, things that can be done *to* class instances, descriptions of responsibilities

Table 17.1: identifying problem domain artefacts

7. Review domain model with stakeholders and identify missing or incorrect aspects
8. repeat!

In this chapter, we will examine methods for constructing domain models in the UML *class diagram* notation. We will be using the sports league management system problem description, introduced in Figure 2.1 in Chapter 2 as a motivating example.

17.4 Class Diagrams

In UML, a class is denoted as a rectangle, which can be divided into compartments by horizontal separators, as shown in Figure 17.3. Every class must be given a type label, shown in the top rectangle of each stack. In addition, classes may have attributes and/or operations listed in the lower two compartments. If both attributes and operations are present then the attributes are given in the middle compartment.

The first step in producing a domain model using the UML class diagram notation is to elicit and categorise the key artifacts from the problem description. Table 17.1 summarises the key artifact types to be identified, and what an analyst should look for in the problem description in each case.

Problem domain analysis (450)

A first pass collection of classes for the sportster application is shown in Figure 17.3. Notice that the attributes and operations of the sportster classes have not been assigned types in Figure 17.3. In addition, many of the classes have not been assigned any behaviour at this stage. The priority during domain analysis is to identify the classes, attributes and relationships. Type information and more operations will be added as the class diagram is refined, during later stages of the design process.

Denoting classes in UML (451)

Figure 17.4 illustrates how classes are annotated with type information for their attributes and operations. Types are placed at the end of the member identifier, after a : colon. Operation parameters can also be added to a class inside the operation signature's parentheses, and typed in the same way as attributes. It is common to omit attributes and operation types during early domain modelling, while the overall class structure is established. However, we will need to use this feature later on.

Denoting classes in UML (452)

Attributes and operations also need to be annotated with their *visibility* as the class diagram is gradually refined. Visibility is particularly important during the later design stages of an object oriented development, when considerations for encapsulation, information hiding and abstraction become important. Table 17.2 summarises the different visibility modifiers for class members.

Member visibility modifiers (453)

17.4.1 Associations and Constraints

Associations are used to denote relationships between classes. Associations are solid lines drawn between classes. An undecorated association can be read as a 'has a' relationship. Figure 17.5 illustrates some example associations between sportster classes.

Denoting association constraints between classes (454)

Constraints on associations are placed at the ends of a relationship denote how many *instances* of the classes may be a member of the relation at run time. Figure 17.5 illustrates the one-to-one, one-to-many and many-many constraint combinations on relationships. Constraints are read at the far end of the relationship from the class of interest. For example, "a sports division has many seasons".

Constraints can be used to indicate ranges of legal instance numbers in a relation. By default, an undecorated relation means one-to-one. Placing a * on a relation means "one to many" and an explicit range of values can be denoted by x..y, which means any number between x and y, inclusive.

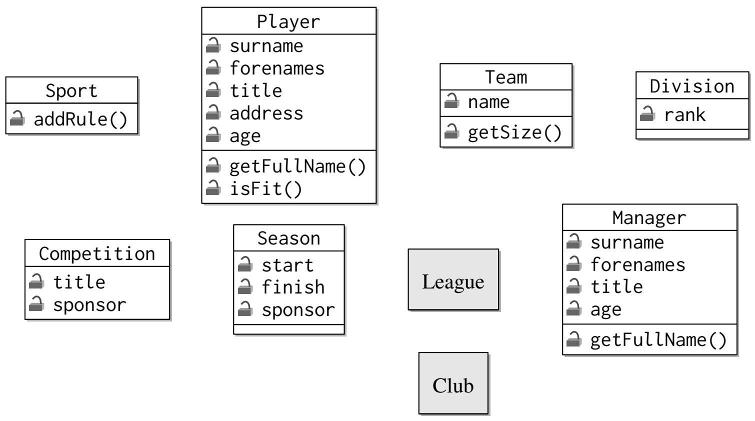


Figure 17.3: UML classes with un-typed attributes

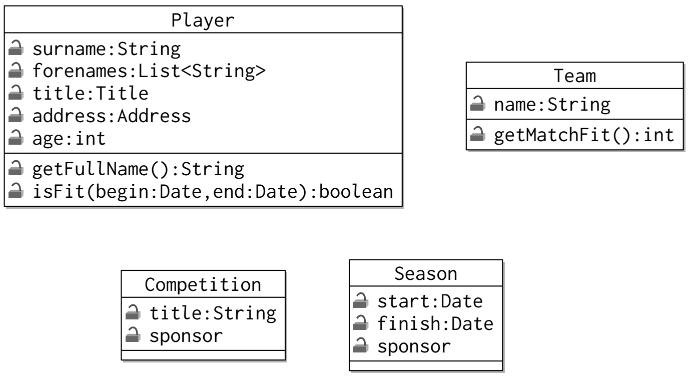
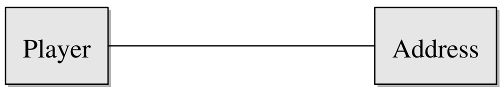
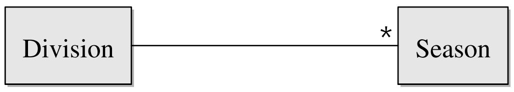


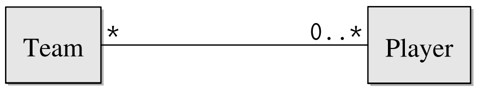
Figure 17.4: extending classes with types



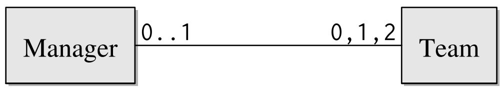
a player has an address, each address is associated with one player (one-to-one)



a division has 1 or more seasons and a season is for one league (one-to-many)



a team consists of zero or more players and a player can play for many different teams (many-to-many)



a team has at most one manager and a manager can manage no, one or two teams

Figure 17.5: associations and constraints in UML

Associations can be labelled with identifiers. These can then be used to provide specific interpretation of the association, instead of a default 'has a' or 'has some'. Figure 17.6 illustrates different uses of labels for the sportster class.

Labelling associations (455)

Associations are implemented as attributes in an object oriented programming language.

17.4.2 Validating Class Associations with Instance Diagrams

Take care when you are constructing class diagrams to read the associations denoted between classes carefully to make sure they make sense for the problem domain.

UML *instance* or *object* diagrams are useful for denoting the state of object oriented systems during execution (remember that an object oriented program's state is the objects, their attribute values and the associations between them). Instance diagrams are used for similar purposes as component diagrams, except they tend to be used to denote smaller scales of detail. Figure 17.7 illustrates the instance diagram notation.

Instance diagrams (456)

An instance is denoted as rectangle and can be labelled with a name, a type, both or neither. In the diagram there is one instance of type `CricketPlayer`, one instance of type `CricketTeam` labelled `england` and another instance labelled `australia` without a type (although we can infer it is probably a `CricketTeam`). The state of instance attributes can also be denoted in slots, for example, the surname of the `CricketPlayer` instance is `"Collingwood"`.

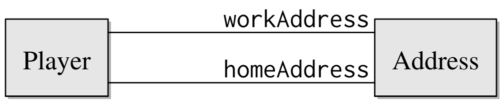
Associations between instances can also be shown on an instance diagram. For example, Collingwood's membership of the England cricket team is shown as an association in Figure 17.7. If necessary, instance associations can be labelled with the class associations they instantiate. Notice that unlike class diagrams, instance diagram associations are one-to-one between concrete instances.

Instance diagrams can be useful in determining the multiplicity of relationships between classes on a class diagram. Figure 17.8 illustrates a sketched instance diagram for the state of the sportster framework for managing cricket teams. From the diagram we can infer that:

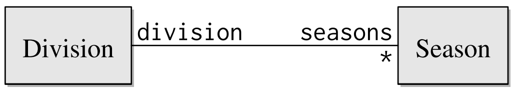
- there is a many-many relationship between cricket players and teams (a team may play for more than one team and a team consists of many players); and
- there is a one-many relationship between teams and sides (each side is drawn from a particular team).

▬, +	**public**	globally visible to all other classes
🔓, #	**protected**	visible to child classes
🔒, -	**private**	only visible internally in the owning class

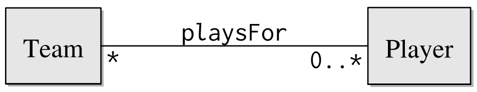
Table 17.2: member visibility modifiers



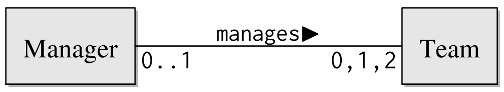
a player has a home address and a work address



a division has 1 or more seasons and a season is for one division (one-to-many)



a team consists of zero or more players and a player can play for many different teams (many-to-many)



a team has at most one manager and a manager can manage no, one or two teams

Figure 17.6: associations and constraints in UML

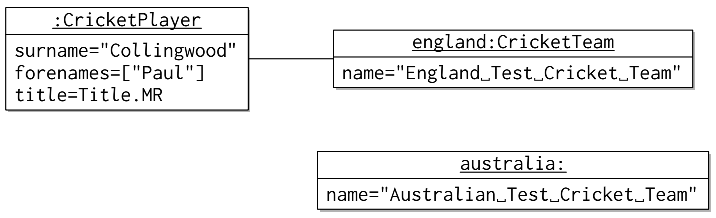


Figure 17.7: denoting class instances in UML

Using instance diagrams to understand class relationships (457)

Be wary of taking too many design decisions before the full class structure has been identified, as this can mean lots of re-working as the architecture of the system to be built is better understood. In particular, allocating types to attributes too early may prevent necessary further classes being identified in the domain model. In general, attributes are simple data items such as boolean values, numbers or strings. More complex data types should be represented as classes in their own right. Determining the difference between a primitive and complex attribute can be difficult early in the modelling process.

For example, we may choose during design to represent a player's address as a separate class or as a `String`, depending on the needs of the application and the complexity of the data to be stored. Figure 17.9 illustrates these options.

Deciding between relations and attributes (458)

In addition, 'utility' classes such as dates, colours, coordinates or locales are usually best represented as attributes because their internal design is not of interest for the design of the system under consideration.

17.4.3 Decorating Associations

There are several decorations that can be added to associations which have formal meaning in UML.

Sometimes, it is convenient to place a class between a many-to-many relationship between two other classes. This is of particular use when class diagrams are used to model relational databases, which don't support many-to-many relationships directly, but instead use *link tables*. Figure 17.10 illustrates how a `Match` association class can be inserted into the many-to-many relationship between `Season` and `Team`.

Association classes (459)

The association class can either be portrayed as having many-one relations to the two main classes in the relationship, or alternatively by drawing a dashed link from the association class to the association. The two portrayals of the relationship between `Season`, `Team` and `Match` are equivalent.

By default, associations are bi-directional, meaning that in an implementation of the model both classes involved in the association relationship will have an attribute of the other type. Sometimes it is unnecessary for one class to have access to the other class, and indeed, it is desirable to remove unneeded attributes in order to reduce coupling. *Directed* associations indicate that the relationship can only be navigated one way, i.e. only class 'knows' of the existence of the other.

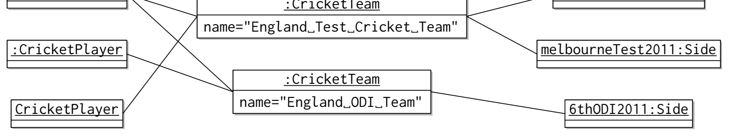


Figure 17.8: using UML instance diagrams to understand class relationships

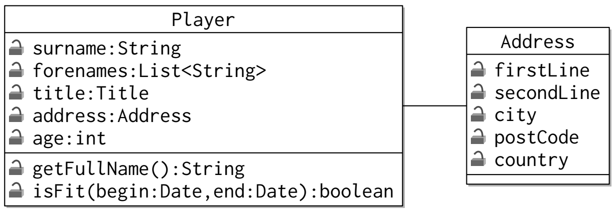


Figure 17.9: refining attribute types during software design

Figure 17.11 illustrates how to denote directed associations on UML class diagrams. The `fieldedBy` association can be navigated from the `Side` class to the `Team` class, but not vice-versa. This means it is possible to discover which team a side was drawn from, but that a team is not used to track the sides that it fielded.

Directed associations (460)

Sometimes, classes maintain associations to themselves, which means that objects of the same class will have references to each other. This can be useful when recursive algorithms must be defined on a class, for example. Figure 17.12 illustrates how to denote a reflexive relationship on a class diagram.

Reflexive associations (461)

The `Season` class has a reflexive relation showing that each instance of the class is preceded by `previous` season and succeeded by a `next` one. This is a similar data structure to a doubly linked list.

Similarly, a `Player` class can be denoted with a reflexive relationship indicating that a player may have a number of `proteges` who will learn from their experience, but that each player can have only one mentor. This illustrates that multiplicities can be applied to reflexive associations in the normal way.

Some classes serve as groupings of instances of other classes. Each of the constituent classes can be said to be a 'part of' the grouping class. This role can be made explicit on a class diagram by using an aggregation association. Figure 17.13 illustrates the use of the aggregation decoration.

Aggregations (462)

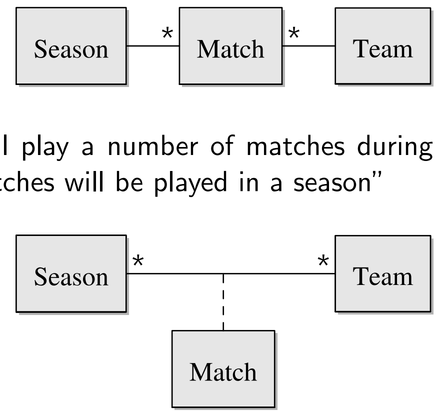

Season
*
Match
*
Team
"each team will play a number of matches during a season and a number of matches will be played in a season"
Season
*
*
Team
Match

Figure 17.10: using association classes for many-many associations

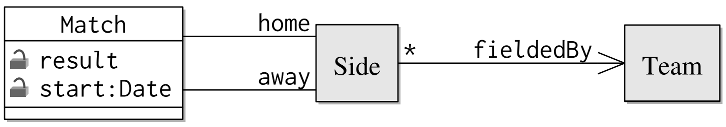


Figure 17.11: directed associations in UML

The class which acts as a grouping of the instances of the constituent class is referred to as the 'aggregate' or 'assembly'. The 'is part of' relationship is indicated by decorating the association with a white diamond head at the aggregate end. In the figure, a Match is denoted as being 'part of' a Season. Note that multiplicities can be added to aggregate associations as normal.

Sometimes, aggregation relationships can be denoted as being stronger than just a 'is part of' relationship. When one class is said to have *life-cycle* control over the instances of another class, this can be denoted on a class diagram as a *composition*. Life cycle control means that should the instance of the composite class be destroyed, all the instances of the constituent class in the composition are destroyed as well. Figure 17.14 illustrates the use of the composition relationship.

Compositions (463)

In the figure, a Club is denoted as being composed of a number of Teams. Should a Club instance be destroyed (perhaps because the club closes) then all the teams drawn from the club will be destroyed as well. This contrasts with the relationship between Team and Player. Although a Player is denoted as being part of a Team, should the Team be destroyed, the Player instances won't be, because the Players could find other Clubs to join and Teams to play for.

Similarly, a Sport has a RuleBook that will be destroyed if the Sport instance is destroyed. However, the Rules that constitute the RuleBook won't be destroyed, as they may be used in more than one RuleBook. Note that an instance can only be grouped into one composite at a time - if the instance is a member of two composites, then either of the composites might try to destroy it when they were disposed with.

The overall functionality of aggregates (and composites) can often be re-

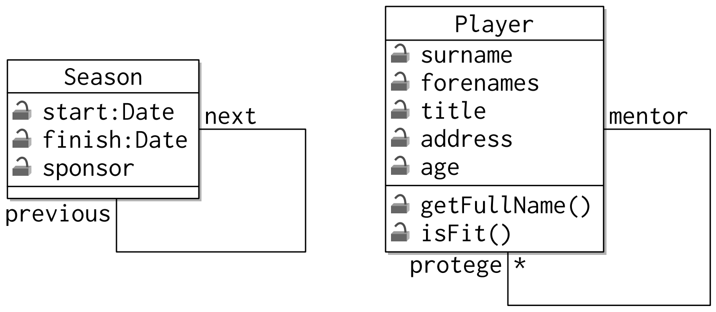


Figure 17.12: reflexive associations in UML

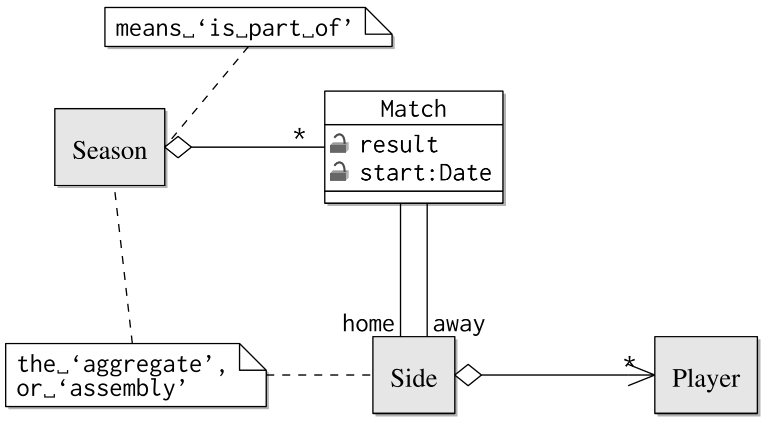


Figure 17.13: aggregation decorations of associations in UML

alised by distributing the computation amongst the member elements. This can be particularly useful when the member elements of the aggregate have different concrete implementations of the task to be computed. This arrangement is called *propagation*, because the computational task is propagated to each of the member elements of the aggregate, without the composite knowing of the precise implementation details of the member elements.

Figure 17.15 illustrates the propagation of functionality for calculating the total runs scored by a cricket team during a single innings. A `SideInnings` instance is composed of a number of `PlayerInnings`. The `SideInnings` class propagates responsibility for storing the total runs scored be a player to the `PlayerInnings` class. Then, the `SideInnings` only has to iterate over the number of individual `PlayerInnings` classes to produce a total number of runs for the side. In addition, the second method `getRunsByBattingOrder()` can also be largely delegated to each of the individual `PlayerInnings` instances.

Propagating Functionality (464)

`SideInnings` (465)

The code below shows how propagation of work in the `getTotalRunsScored()` method works from the `SideInnings` to `PlayerInnings` classes.

```
Map<CricketPlayer,PlayerInnings> innings;
```

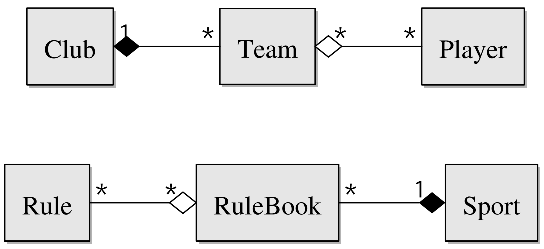


Figure 17.14: composition decoration of associations in UML

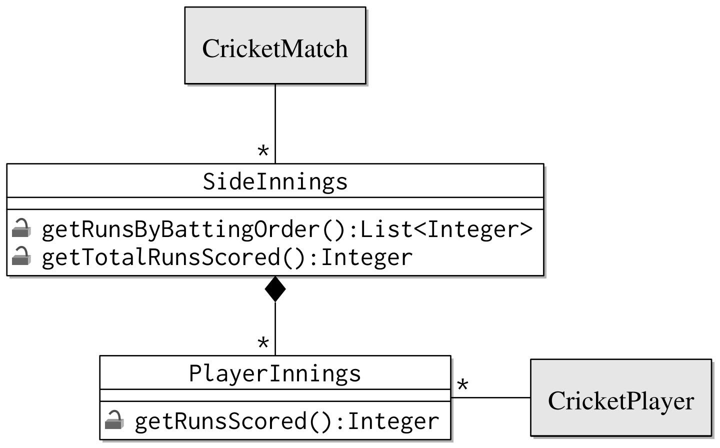


Figure 17.15: using compositions for functional propagation

```java
public Integer getTotalRunsScored(){
 Integer result = 0;
 for (PlayerInnings players: innings.values())
  result += players.getRunsScored();

 return result;
}
```

SideInnings (466)

```java
List<CricketPlayer> battingOrder;

public List<Integer> getRunsByBattingOrder(){
 List<Integer> result = new ArrayList<Integer>();

 for (CricketPlayer player: battingOrder){
  PlayerInnings pInnings = innings.get(player);
  if (pInnings != null)
   result.add(pInnings.getRunsScored());

 }
 return result;
}
```

Delegation is a similar technique to propagation, and is also associated with composites. Sometimes the desired functionality for a class can already be found to have been substantially implemented in another. Recall the `Map` interface (and implementing classes) from the collections framework covered in Chapter 2. The `java.util` package also contains a `Properties` class used to manage configuration options for applications and for the Java Virtual Machine. A single property is a key/value combination that can be stored in a `Properties` instance, or serialised out to an XML or text file stream. Rather than re-implementing the functionality necessary to store key/value combinations, the

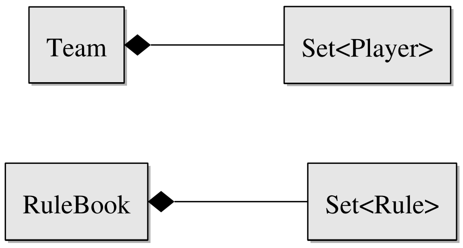


Figure 17.16: using compositions for functional delegation

Properties class delegates responsibility for this to an implementation of the Map interface. The Properties class provides the extra functionality for serializing properties to the various available formats.

Figure 17.16 illustrates another example of delegation. The Team class is responsible for storing the Player instances currently associated with the team. Rather than re-implement (and hard code) the functionality for storing the Player instance, the model is arranged so that responsibility for storage is delegated to the Set class. The Set class already enforces the rule that a player can only be in the team once, and provides methods for accessing and iterating over the Player members. Every Team instance then has a players:Set<Player> attribute that is used to store players. This form of delegation happens so often, that usually it is not made explicit.

Delegating Functionality (467)

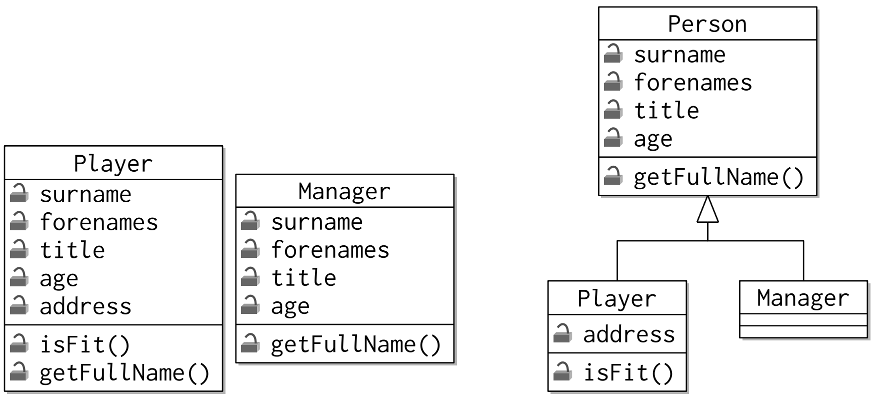
17.4.4 Polymorphism during analysis

The initial stages of the object oriented modelling process are concerned with the identification of concrete types from the problem domain and their features and behaviours (represented as attributes and operations). As the model is refined towards an object oriented software design, we need to begin applying good design principles to the model. In particular, we need to avoid replicating implementation details shared between different classes in the problem domain. We have already introduced the notion of polymorphism in Section 2.3, including concepts such as generalisation/specialisation and inheritance.

There are several guidelines for appropriate use of generalisation relationships when refining an object oriented model:

Guidelines for using polymorphism during analysis (468)

- generalise by grouping common operations and attributes in the super-class;
- check the *is a* rule for all specialisations;
- specialise the behaviour of sub-classes by over-riding explicit abstract operations in the super class;
- don't over-specialise, particularly for situations where only attributes are different; and



(a) initial modelling (b) refinement

Figure 17.17: grouping common operations and attributes in abstract classes

- avoid multiple inheritance for one class where possible.

Lets consider these guidelines in more detail.

The first step during object oriented model refinement is to identify areas of duplication in the domain model. Duplication may occur because the same features need to be represented in different concrete classes. When attributes which store the same data, or operations that exhibit the same behaviour are identified in two different classes, it may be appropriate to group the common features together in a more abstract general class.

Figure 17.17 illustrates the collection of common attributes and operations from the `Player` and `Manager` class into a more abstract `Person` class. The `Player` class retains the player specific features, the `address` attribute and the`isFit()` operation.

Generalise by grouping common operations (469)

The `Manager` class is retained as a separate entity for now, since further operations or behaviours may be added in the future. This illustrates a further issue during modelling. Sometimes, a class may appear temporarily redundant and it may be tempting to remove it from the model to reduce complexity. Consideration must, however, be given to the possibility that the features of the model are in-complete. In general, it is better to consider removing classes from the model later in the development cycle, when the scope of the model is better understood.

Notice that in the example above, the members that are grouped into the common `Person` class have the same *semantic* meaning in the `Player` and `Manager` classes. A manager's surname, for example has the same meaning as a player's surname.

It would even be appropriate (with a little refactoring) to group together attributes or operations with the same semantic meaning, even if they had slightly different labels or signatures. For example, if the `Manager` and `Player`

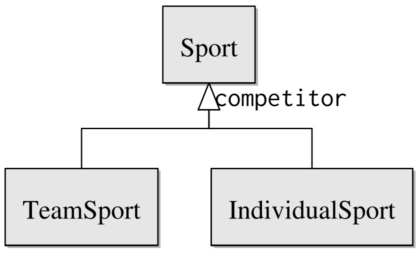


Figure 17.18: inheritance discriminators

class were identified by two different developers during problem domain analysis, it is possible that one developer would define an attribute `Player.forenames` and the other defined an attribute `Manager.givennames`. In this case, the attributes could still be merged into the more general `Person` class, using one of the two labels.

However, it would not be appropriate to merge two classes based on members with the same label, if the two attributes have different semantic meanings. A refinement of this kind would violate the is-a rule, because the member in the super class would be used for different purposes in the sub-classes of the generalisation relationship.

Complementary to this is the need to identify a valid *discriminator* for the specialised sub-classes. A discriminator is a label for a generalisation relationship that can be used to justify the difference between each of the sub-classes of a super-class. Figure 17.18 illustrates a discriminator for the `Sport` class sub-classes.

Inheritance discriminator (470)

In the figure, the `competitor` discriminator justifies the existence of two sub-classes, `TeamSport` for sports played by teams (football, rugby, cricket and so on) and `IndividualSport` for sports played by individuals (marathon running, javelin etc.).

A related problem in refinement is when instances within a single class share attribute *values*. When the value of an attribute changes in one instance, the value must change in all the other instances. This is effectively a one-many relationship between attributes within a single class.

In this situation, it can be tempting to generalise the commonly shared attribute into a super class to avoid duplication. However, this is inappropriate, because:

- the generalisation will probably violate the *is-a* rule; and
- the value of the attribute will still be shared between all the concrete instances, even though it is defined in a super class.

Figure 17.19(a) illustrates an example of this situation. The developer has identified that sub-groups of instances of the `Side` class share the value of the

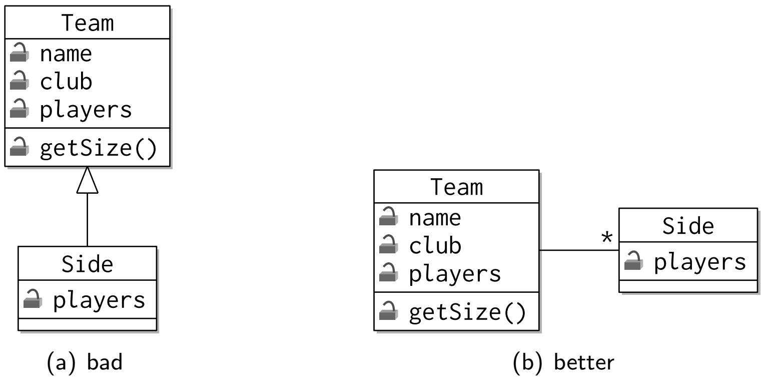


Figure 17.19: a bad generalization violating the *is a* rule

name class. For example, if the name of a football team should change from ‘Caledonian Thistle’ to ‘Inverness Caledonian Thistle’ then the change will need to be reflected in all instances of the Side class drawn from that team.

Abstraction-occurrence design pattern (471)

The developers have attempted to remedy the problem by generalising the name attribute into the Team class. However, this is inappropriate for the reasons given above.

Figure 17.19(b) illustrates the use of the abstraction-occurrence pattern (see Section 20.2.1) as a better way of solving the refinement problem. The one-many relationship between name and the other attributes of Side is represented by a one-many relationship between Team and Side. Reading the relationship to validate it says, “A team fields many sides and a number of sides are drawn from the same team”. The Team is called the *abstraction* because it stores the values that remain unchanged. The Side is called the *occurrence* because it stores the things that will change each time a team fields a side to play a match.

Sometimes it is useful to specify a general abstract class before any of the concrete classes have been identified. This is more likely to occur during the later stages of refinement, as considerations regarding software re-use and re-usability begin to emerge. Sometimes, abstract classes are used to provide the general functionality for realising an interface, with the final implementation details left to concrete classes. The AbstractCollection class, for example, provides almost all the functionality for accessing the elements of a Collection, leaving only the very low level methods regarding adding, removing and iterating over the collection to be provided in the concrete class.

When specifying abstract classes for later implementation, it is good practice to specify the signatures of abstract methods that any sub-classes must provide. This:

- proscribes the functionality of the sub-class;
- allows instances of the sub-classes to be fully treated as instances of the super-class.

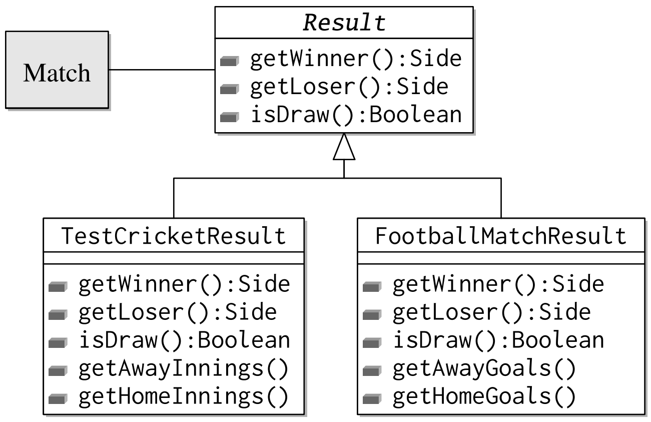


Figure 17.20: making specializations explicit with abstract operations

Figure 17.20 illustrates this practice. The Match class is associated with the abstract Result class. Notice that results is denoted as abstract by using an italic font for the class name. The Result class is sub-classed by two concrete class's TestCricketResult and FootballMatchResult which provide the implementation details for deciding the outcome of a test cricket and football match, respectively. The implementation of these two classes is enforced by the specification of abstract method signatures for deciding whether the match was a draw and for identifying a winner and loser in the match.

Specialise by explicit over–riding (472)

By arranging Result as an abstract class, whose concrete sub-classes provide sport specific implementations, the Match class can utilise the behaviour of the general class, without knowing precisely how each of the abstract methods is implemented.

When no methods are over-ridden in sub-classes, it is worth considering whether the generalisation is needed at all. Similarly, avoid over generalisation for attributes only, particularly when the generalisation is used to manage the partition of attribute values, rather than the identification of extra attributes in sub-classes. Figure 17.21(a) illustrates an example of excessive specialisation. The Player class is sub-classed into ProfessionalPlayer and AmateurPlayer. This specialisation appears to have no role, except to distinguish the career status of a sports player. Consequently it is probably better to remove the specialised classes, and introduce, an attribute to record whether a Player instance represents a professional or not, as shown in Figure 17.21(b).

Don't specialise by only attributes (473)

Specialised classes that only have different attribute values from other sub-classes may be better represented as instances. In addition, over-specialisation may mean that an object's concrete class may need to change over its life-time. This can be error prone and should be avoided.

The final guideline concerns the use of multiple inheritance, a feature that is not supported by all object oriented languages. Multiple inheritance is available in UML, and, as illustrated in Figure 17.22(a) the usual rules for validating the generalisation rules should apply. The `PlayerManager` represents a person who is both a `Player` in, and a `Manager` of a team.

Avoid multiple inheritance (474)

Despite its availability, multiple inheritance should be avoided if at all possible, because:

- of the additional complexity that is introduced into an object oriented model. In particular, a software designer must manage how identical members are inherited by a sub-class from two or more super-classes; and

- multiple inheritance often represents situations where an object is obliged to change its type over time. For example, a `Person` may begin their sporting career as a `Player`, progress to being a `PlayerManager` and finally end their career as a manager of a team only. This process of copying attributes from an instance of an old type to an object of the new type must usually be done manually and can be error-prone.

The *player-role* design pattern (see Section 20.2.2) can often be employed to solve the problem of multiple inheritance. Figure 17.22(b) illustrates the use of the pattern. The `Person` class acts as a *player*, which has a one-many relationship with a number of *roles* which can be played in the object oriented system. The roles assigned to a player may change over time without changing the type of the player object. In the example, a `Person` instance can be assigned a number of `SportRole` roles (which is an interface), such as as a `Player`, `Manager`.

The player-role pattern example above introduces the notation for *interfaces* in UML class diagram. Recall that interfaces are used to specify the operations that a concrete class must implement as methods.

Figure 17.23 illustrates two variants of the notation. In both cases interfaces are denoted in a similar way to classes, except that an interface does not by default have a compartment for attributes. The interface on the right is classified explicitly using a *class stereotype*. The interface on the left is denoted using a *visual stereotype* - the italic font for the interface class name.

Denoting interfaces (475)

Note that interfaces cannot have instance attributes, since interfaces do not provide implementation for instances). Consequently, it is not appropriate to draw an association from an interface to another entity (although the reverse is

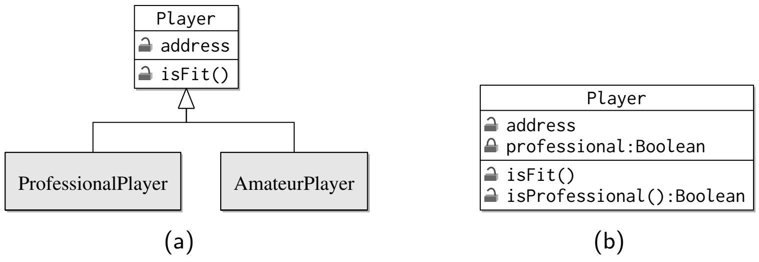


Figure 17.21: merging excessive specializations

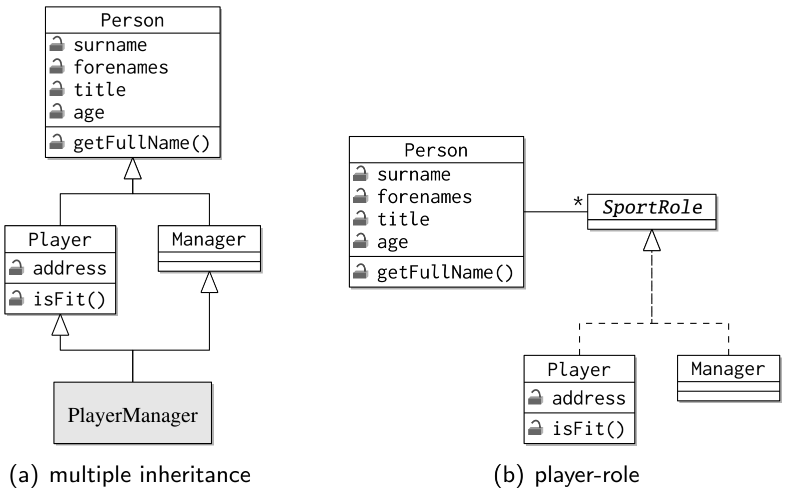


Figure 17.22: using the player-role design pattern to avoid multiple inheritance

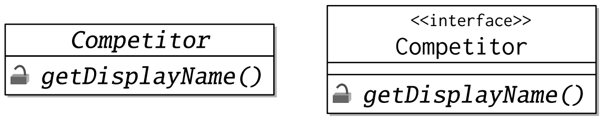


Figure 17.23: denoting intefaces in UML

fine). However, it is possible to denote an interface as having a class attribute. For example, it may be necessary to specify the flags that an interface operation can accept.

Figure 17.24 illustrates the two ways that the realisation of an interface by a class can be denoted on a class diagram. Figure 17.24(a) shows the use of a realisation relationship between a class and an interface. Realisation relationships are similar to generalisations, except that the line is dashed.

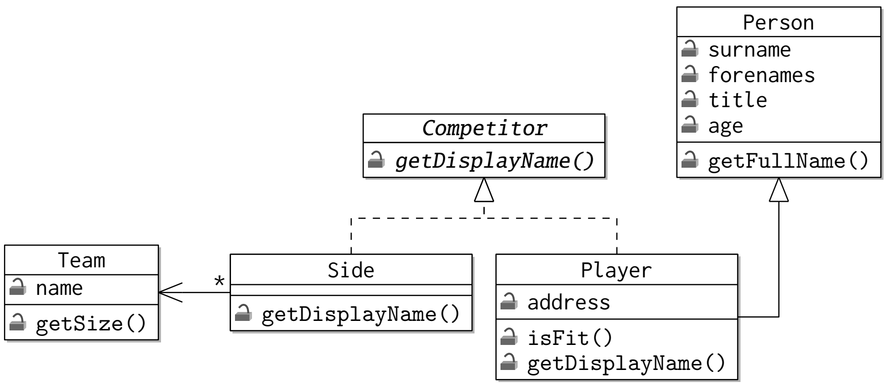
Realising Interfaces (476)

Figure 17.24(b) illustrates the use of an exported interface from a class. We have already seen exported interfaces on component diagrams - a 'lolly pop' attached to the realising entity and labelled with the name of the realised class. This form of interface notation is generally less common, although it can be useful for denoting the realisation of 'common' interfaces that occur throughout an object oriented program.

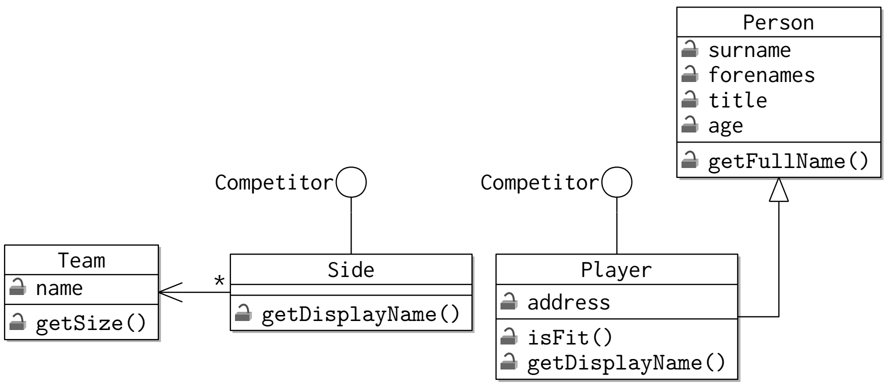
17.4.5 Extending Class Diagrams

The descriptive capabilities of UML class diagrams can be extended with:

- informal documentation to aid understanding, in the form of notes; and
- formal extensions using the *stereotype* mechanism.

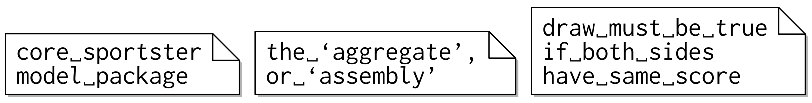


(a) realization relationship



(b) providing an interface

Figure 17.24: realizing intefaces in UML



(a) labelling diagrams (b) adding additional information (c) describing more complex constraints

Figure 17.25: uses of notes in UML

Notes are a visual representation of informal documentation on all types of UML diagrams. Notes can be used for: Using notes (477)

- labelling diagrams to provide a reference identifier. This can be particularly useful for linking between diagrams;
- adding additional explanation to explain a particular design choice or link with a particular requirement that a design feature satisfies; and
- informally describing more complex constraints that cannot be expressed using multiplicities.

Figure 17.25 gives some examples of the various uses of notes on a class diagram. In fact, we have already seen how to associate notes with particular elements of a class diagram in Figure 17.13, by drawing a dashed link between the note and the element of interest.

Notes can also be used to annotate diagrams with formal constraints in the Object Constraint Language (OCL) Constraints. Chapter 30 describes the use of formal methods, including OCL, in the specification and design of software systems.

In some cases, the standard relationships for class diagrams are not appropriate to denote the particular kind of relationship needed. In this case, a general *dependency* relationship can be used, indicating that one class is dependent in some way on the other.

Dependencies can be annotated with a *stereotype*, a formal meaning, which extends of the core UML standard. The stereotype label is depicted between a pair of *guillemots*.

Figure 17.26 illustrates the use of a stereotyped dependency. The figure shows that the `Main` class is responsible for creating instances of the `League` class. The `Main` class is probably an entry point into an application, responsible for parsing any command line options. The `League` class is probably a *singleton* instance representing the front end of the functionality of the sportster framework.

In fact, stereotypes can be used to annotate general dependency links and other UML diagram elements such as classes. Some standard dependency stereotypes for classes are: Dependencies and stereotypes (478)

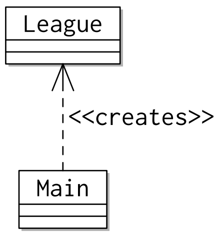


Figure 17.26: using dependencies and stereotypes in UML

- `<<boundary>>`, which denotes that the class is an intermediary between the object oriented system and its environment. Examples of boundary classes include user interfaces and classes that interact with external system resources;
- `<<control>>`, which denotes that the class manages the flow of control between other classes;
- `<<thread>>`, which denotes that the class defines an independent *thread* of execution, including its behaviour and state;
- `<<table>>`, which denotes that the class represents a table in a relational database. Class diagrams are suitable for describing entity relationship models; and
- `<<enumeration>>` which denotes that the class has a set number of predefined and labelled instances.

Some standard stereotypes for dependencies are:

- `<<creates>>`, which denotes that instances of one class are created by another; and
- <<**import**>>, which denotes that one *package* is dependent on the classes in another. We will look at package diagrams in Chapters 20 and 21.

In fact, much of the UML standard is defined using the stereotype mechanism. A stereotype may have an equivalent visual representation.

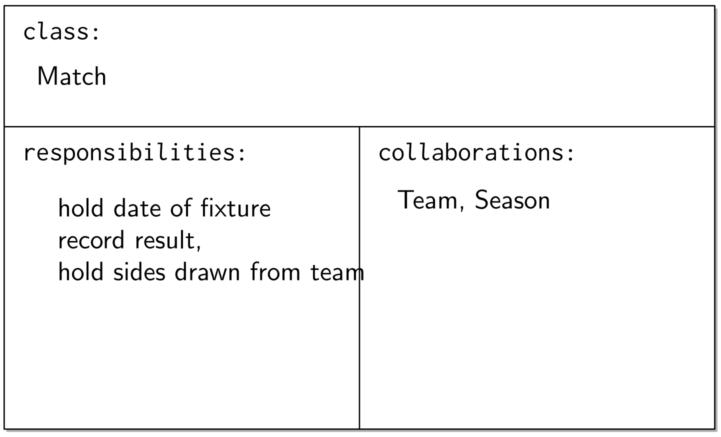
Two or more dependencies with the same stereotype should be interpreted in the same way. Stereotypes can be used throughout the UML. Many common stereotypes are given a graphical depiction - many of the relationships we have covered in this chapter are actually stereotyped dependencies.

17.5 Class Responsibility Collaboration

Problem descriptions and use cases are the primary source of information for domain analysis, since they implicitly describe the entities in the problem environment that must be represented in the system. As with other aspects of

A company sells a system for managing sports leagues and competitions. Each league has a number of teams (e.g. football), or individual players (e.g. singles tennis). Teams consist of a number of players. Each team plays other teams in matches during a season. The purpose of the system is to support all aspects of managing a sport, including scheduling matches during a league season, one-off competitions, and recording statistics about matches played, such as player performance.

Figure 17.27: identifying entities in a problem description



class:

Match

responsibilities:

hold date of fixture
record result,
hold sides drawn from team

collaborations:

Team, Season

Figure 17.28: match class CRC card

requirements elaboration, the domain model must be developed iteratively as the set of use cases becomes more refined.

Eliciting domain entities (479)

As we have already seen, a straight-forward method for identifying domain entities is to extract *nouns* in problem descriptions. Consider the problem description in Figure 17.27. Each of the nouns has been emphasised. Of immediate interest are the nouns *Season*, *Match* and *Team*.

We now need to translate from the domain entities we have identified to a UML class diagram. Class-responsibility-collaboration (CRC) is a method for eliciting and documenting domain classes during analysis Beck and Cunningham [1989]. Candidate classes are recorded on index cards that can be inspected and re-organised to elicit the structure of domain relationships.

Figure 17.28 illustrates the format of a CRC card, using the `Match` class as an example. The name of the class is placed at the top of the card. The rest of the card is divided into sections used to describe the *responsibilities* of the class and *collaborations* with other classes.

CRC card schema (480)

Responsibilities can be categorised as:

doing: such as coordinating activity, invoking operations or providing services; or

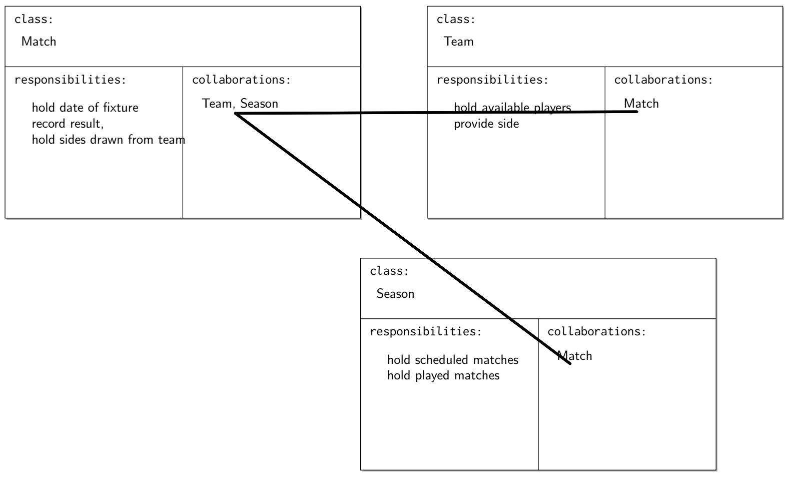


Figure 17.29: class collaborations

knowing: the private data and associations held by instances of the class or how to calculate to a value.

The class descriptions are documented by hand on CRC cards should be informal and used for a basis for discussion. As well as recording them on the card, collaborations can also be visualised by laying out the cards and drawing lines between them, e.g. on a white board in the team's area. Figure 17.29 illustrates the collaborations between three of the different classes identified so far from the problem description.

Finding collaborations (481)

The CRC cards are grouped together to investigate how they collaborate with one another. Lines are drawn between collaborating classes indicating the basic structure of the interaction.

17.6 CRC Cards to UML Class Diagrams

We can now use the CRC cards to begin developing a class diagram. We need to define a class with a name, operations and attributes for each of the CRC cards. The name of the class comes directly from the name on the CRC card. The attributes and operations can be derived from the responsibilities identified for the class. Broadly, knowing responsibilities will be represented as attributes of the class and doing responsibilities will be represented as operations. Figure 17.30 illustrates an initial class diagram containing classes for each of the CRC cards developed from Figure 17.29.

Notice that much of the functionality is concerned with manipulating the information we wish to represent about the domain. An initial attempt at establishing the likely attributes and relationships has been made, however, further functionality will be added during later design stages as the architecture be-

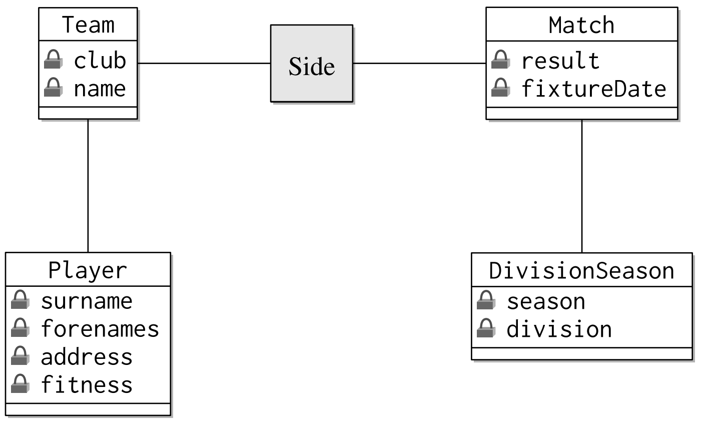


Figure 17.30: UML class diagram of the `Branch`, `Request`,`User` and `Book` classes

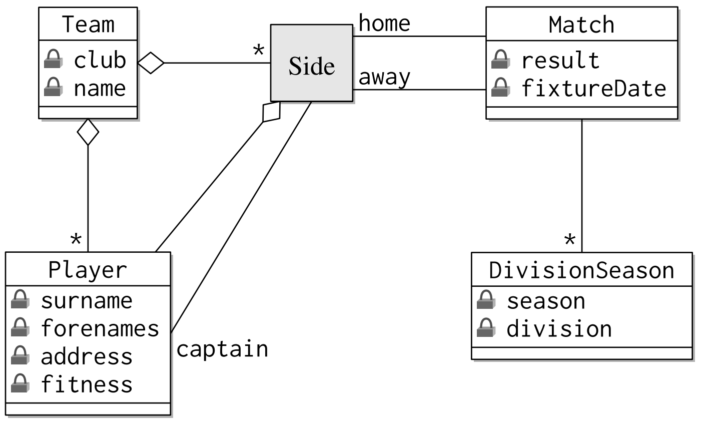


Figure 17.31: revised UML class diagram of the `Branch`, `Request`, `User` and `Book` classes

comes more established. The domain model will also be extended further as each use case is examined for domain entities.

CRC cards to UML class diagrams (482)

Attributes can be derived directly from use cases and the problem description; or from the identification of 'knowing' responsibilities and equivalent operations. Alternatively, the absence of details of the attributes for a class may indicate that further requirements elicitation activities are required. The initial analysis does not need to establish the precise representation of an association as this will be done at the design and implementation stages.

Returning to the class diagram introduced in Figure 17.30, we can now refine the attributes and relationships with further detail. Figure 17.30 illustrates the same class diagram with further detail. The association decorations from Figure 17.4.3 have been added, as well as the multiplicity relationships between the classes.

Revised class diagram (483)

The diagram does raise the question as to whether the attributes identified for the `DivisionSeason` class is correct. We have assumed that a match is played in a division in a particular season, but we have not yet decided how to represent this information in the system. We have also also assumed that all matches will be played on their scheduled date (i.e. that they don't ever need to be postponed). However, we can't be sure from the current user stories whether this is correct. We may need to investigate these questions by developing a prototype and evaluating it with the customer.

Summary

This chapter has looked at the use of UML class diagrams for documenting the key elements of a problem domain. Class diagrams can be used to denote the key elements in the domain as classes, and the relationships between the elements as associations.

We also looked at how class diagrams can be refined and enriched as a software design is developed. This process exploits the polymorphic features of objects. When refining class diagrams it is vital to validate the models constructed by checking the meaning of the different relationship types used. The two high level types of relationship on a class diagram are associations, which should be read as *has a* and generalisations, which should be read as *is a*.

Finally, we looked at using interaction diagrams to translate from descriptions of the interactive features of a system into structural representations, using sequence and communication diagrams.

17.7 Exercises

1. Consider the problem description given in Figure 17.32.

   A forensic software company is developing an application for acquiring forensic evidence from mobile devices. The software must work with a unknown number of different types of mobile device, including different brands and manufacturers of mobile phones, media players and smartphones. Mobile devices have a model and manufacturer. A model has a memory capacity and a form factor (candy bar, touch screen etc.).

   A typical *use case* for the application is for the mobile device to be connected via a USB cable to a personal computer. The software application will then utilise a device driver (a software library for interacting with hardware) to interact with the mobile device via the USB cable. Some manufacturers provide device drivers for their mobile devices, while others must be custom written by the company's development team. Many generic device drivers are suitable for use with a number of different models.

   It is assumed that each device stores its data on one or more NAND chips. The entire image of the device must be transferred to the desktop application for analysis via the device driver.

   The memory image of any phone is organised into a particular file system arrangement (such as TFAT32, iOS or YAFFS) Once the image is transferred, the file system must be decoded and the contents presented to the user. The file system may be flat or organised into a hierarchy of directories. There may be many different types of file that must be presented to the user, including images, word processor documents, voice recordings, sound tracks, call records and SMS message databases. Each file type must also be decoded so that the contents can be presented to the user for further analysis.

   Figure 17.32: mobile forensics application problem description

   (a) Identify the key concrete classes in the problem description
   (b) Identify the attributes of the classes.
   (c) draw a class diagram showing the initial associations between classes that you can identify.
   (d) List the responsibilities in the description.
   (e) Draw a sequence diagram for the acquisition of memory image from a mobile device. If your diagram identifies new classes of instance add them to the class diagram.
   (f) Refine the class diagram by identifying potential areas where generalisation relationships could be applied.
   (g) Allocate the responsibilities to classes as *collaborations* and decide what class operations are needed to fulfil them.

(h) Refine the diagram further by adding types and arguments to operations, and identify new classes.

2. Consider the problem description given in Figure 17.33.

   The Sock and Piddle micro-brewery needs a new system for tracking inventories of beer in a warehouse. Beer is sold to local public houses (pubs) and shops. There are a number of different brands sold by the brewery, each with a name (e.g. 'Piddle's Best'), hop (brown beer, golden or stout only), an alcoholic strength (expressed as a percentage of alcohol in the beer by volume), and a price per litre sold.

   Beer is stored in bottles and barrels. Both bottles and barrels of beer have a sell by date. Bottles of beer have a fixed volume of one litre. The cost of a bottle of beer is the cost of one litre of the beer and the cost of the bottle.

   The brewery sells several different barrel sizes by volume. Barrels are returned to the brewery when empty so the price of a barrel of beer is calculated by multiplying the price per litre of the beer by the volume of the beer (i.e. the barrel itself is not charged for).

   Pubs only buy barrels of beer and shops only buy bottles. All sales (to pubs and shops) are recorded on an invoice, which is used to prepare the delivery and adjust stock levels in the warehouse. The invoice includes the date of the sale, the name and address of the purchaser and the barrels or bottles to be sold.

   Figure 17.33: the Sock and Piddle micro-brewery domain description

   Model the problem domain with a UML class diagram. You should include the key classes, their attributes and any operations that can be identified from the description. In addition you should show the relationships (including any multiplicities) between the classes in the diagram.

3. Consider the class diagram shown in Figure 17.34 of part of a music player.

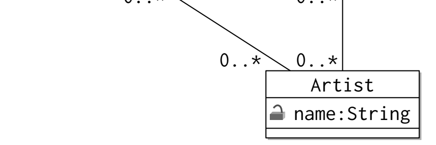


Figure 17.34: class diagram of a music player

Draw an instance diagram that represents the state of the music player when a single music track called "Drinking Piddle's Best" from the album "Sock and Piddle Nights" is stored in the application. The track is a collaboration between two artists called "Barman" and "Landlord".

4. Consider the instance diagram shown in Figure 17.35.

   The figure shows a collection of objects representing newspapers, magazines, cities and countries

   (a) draw a class diagram to represents the structure of an object oriented system that could contain the objects and relationships shown in the diagram. Include the necessary associations and multiplicities.
   (b) Review the class diagram you have drawn; could it be refined by generalising some of the classes?
   (c) A decision is made to record every edition of each newspaper and magazine.
      i. What *design pattern* would you apply to achieve this?
      ii. Add the changes you propose to your class diagram.

5. Take another look at the problem description shown in Figure 17.32

   (a) Draw a sequence diagram that represents the series of interactions described for recovering a memory image from a mobile device.
   (b) Translate the sequence diagram into an equivalent communication diagram.
   (c) Draw a class diagram of the types shown on your communication diagram.
   (d) How does the class diagram compare to the structure you developed for the exercise above?

6. Recall the problem description given in Figure 15.7

   (a) create a set of CRC cards for the key candidate classes needed to realise the use cases (you will be provided with some index cards to do this);
   (b) produce a class diagram showing the class names, attributes, operations and associations. Try creating two versions of the class diagram. The first version should just show the class names and associations between the classes. The second version should add attributes and operations to the key domain classes;
   (c) identify the three most important analysis questions you would want to answer to improve or validate your domain model.

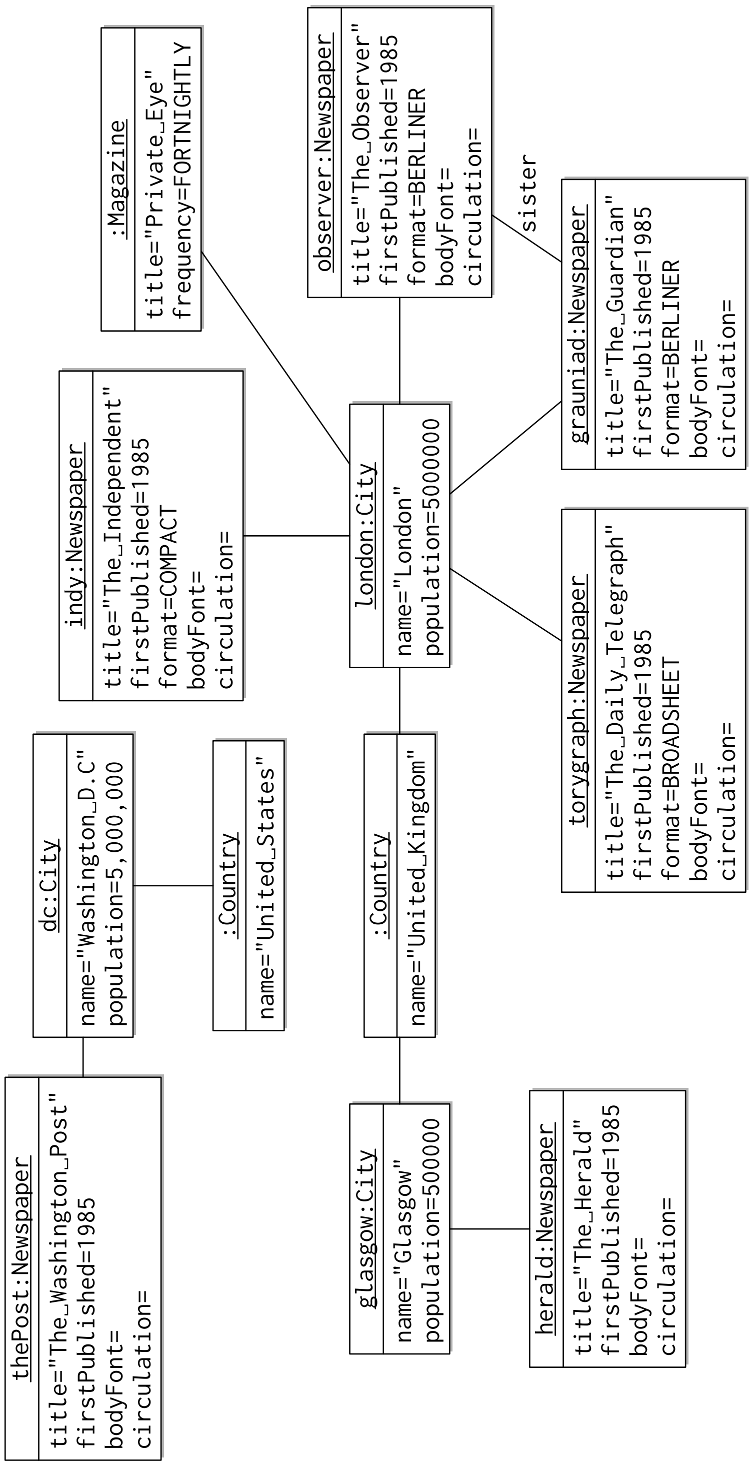


Figure 17.35: an instance diagram showing, newspapers, magazines, cities and countries