2.3.5 Subtypes and Multiple Inheritance

A subtype inherits all attributes, operations and exception definitions from its supertype.

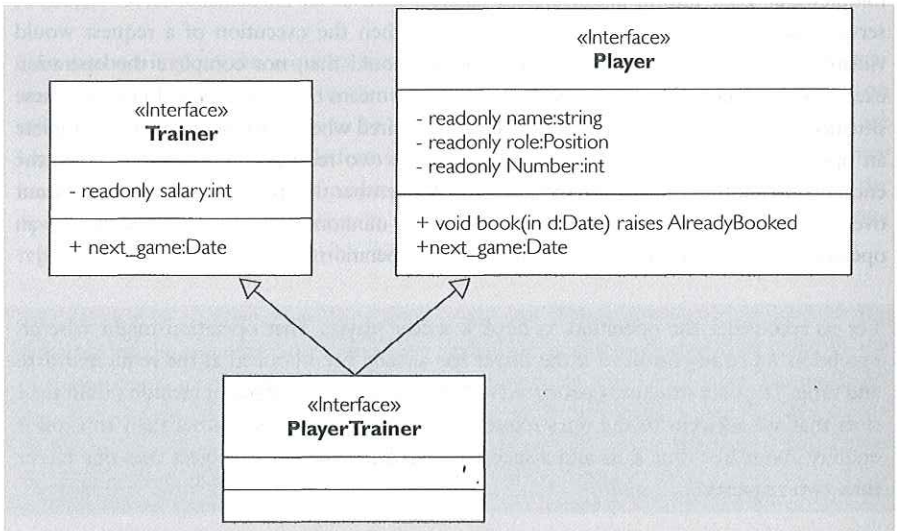
Exceptions, attributes and operations are defined for object types. Object types may have exceptions, attributes and operations in common with other types. A subtype has all the attributes, exceptions and operations declared for its supertype; the subtype *inherits* these attributes from the supertype. The subtype may add attributes, exceptions and operations and become a specialization of its supertype.

A subtype can inherit from more than one supertype.

An object type may inherit from more than one supertype. This concept is referred to as *multiple inheritance*. If the component model supports multiple inheritance, the inheritance relationship between object types is a graph, rather than a tree. As an example, consider a soccer player who is also a trainer (as occurs in many amateur teams). A type `PlayerTrainer` would, therefore, be a subtype of both `Player` and `Trainer`.

Although very elegant at first glance, multiple inheritance can cause problems (see Example 2.12). Multiple inheritance can lead to name clashes if a subtype inherits two properties from two different supertypes or if an operation inherited from a common supertype is re-defined on one branch and not re-defined on another branch.

Example 2.12
Name Clash with Multiple Inheritance



Both types `Player` and `Trainer` define an operation `next_game` that returns details about the next game in which the player or the trainer have to engage. This leads to a name clash in type `PlayerTrainer`. If a client of a `PlayerTrainer` object requests execution of `next_game` it is unclear whether the operation should be taken from `Player` or from `Trainer`.

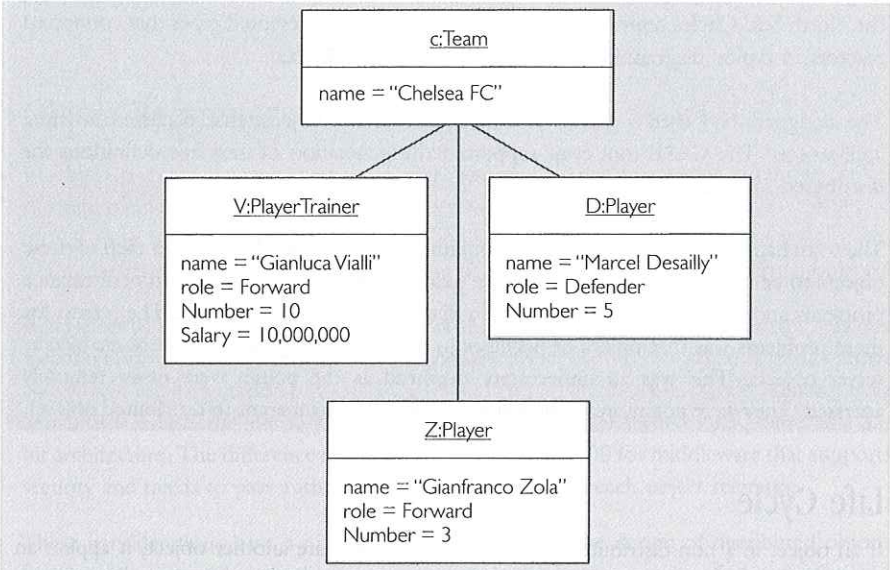
The concept of subtypes is one of the strongest advantages of object-oriented meta-models, for several reasons. First, the design and successive implementation of object types can be reused in different settings where types can be extended in new subtypes. This increases development productivity considerably. Secondly, changes to common attributes, exceptions and operations are easier to manage, because they only affect one object type. Thirdly,

subtyping supports different levels of abstractions in a design of distributed objects and they become easier to understand. Finally, whenever an instance of a particular type is expected, for example as a parameter or in an attribute, instances of all subtypes can be used. Hence, subtyping enables polymorphism.

2.3.6 Polymorphism

Object types restrict the domain of attributes and parameters. Object models that support *polymorphism* not only allow instances of the attribute or parameter type to be assigned to attributes and parameters, but also instances of all the type's subtypes.

Polymorphism means that an attribute or parameter can refer to instances of different types.



Example 2.13
Polymorphism

This `Team` object uses polymorphism: it refers to both `Player` and `PlayerTrainer` objects.

Subtypes may adjust inherited operations to fit the subtype's needs. This is achieved by redefining operations. *Redefinition* is achieved by declaring the operation with the same name and the same signature in the subtype and implementing it in the subtype. The correct operation is chosen by means of late binding.

Late binding means that it is decided at run-time, rather than at compile-time, which operation executes in response to a request. The decision is based on the dynamic type of an object. The search for an operation to implement a request starts at the type from which the object was instantiated. If that type does not implement the operation, the next supertype is searched until a supertype is found that does implement the operation.

Late binding means that it is decided at run-time which operation executes a request.

Late binding is often used together with polymorphism. Clients use operations that are defined for the static type of an attribute or parameter. At run-time, instances of subtypes may be assigned to the attribute or parameter. If the attribute or parameter is used in a request, late binding is used to determine which operation actually performs the request.

2.4 Local versus Distributed Objects

The design of local objects that reside in centralized applications is reasonably well understood. The software industry has agreed on UML as a notation that is highly suitable for such designs. CASE tools are available that support editing the different UML diagram types and provide code generation facilities that map to C++, Java or Smalltalk. In this section, we explain the differences between designing local and distributed objects to avoid pitfalls.

Example 2.14
Differences between Local and Distributed Objects

Some time ago, we were called in to advise on some problems with a distributed design for an engineering data warehouse. The purpose of the warehouse was to manage engineering diagrams for drilling-rigs, pipelines and pumping-stations for a new oil-field exploration in the North Sea. Circles represented reactors and polylines represented pipes that connected reactors. A typical diagram included some 50,000 such objects.

The designers had used a CASE tool to model all the objects that occurred in these applications. The CASE tool even supported the generation of interface definitions for distributed objects.

The team happily used that functionality and implemented types that enabled each of these objects to be requested remotely. We were called in when the team detected performance problems and were surprised that loading a diagram took several minutes. The reason for these problems was that points of polylines in the engineering diagrams were enabled as server objects. This was an unnecessary overhead as the points were never remotely accessed. They were not aware of the request latency that is inherent to distributed objects.

2.4.1 Life Cycle


Creation, migration and deletion of distributed objects is different from local objects.

If an object in a non-distributed application wants to create another object, it applies an object creation operator to a class or an object type. The operator invokes a constructor, which then creates the object. The new object will reside on the same host and in the same process space as the object that requested its creation. The object also will not leave that virtual machine until a point in time when it is deleted. Therefore, object creation is considered an implementation problem.

Now consider a client object in a distributed application that wishes to create objects on some other host. Object creation operators and constructors are not capable of achieving this. We need to use them as primitives for designing object types that are capable of creating objects elsewhere. Hence, distributed object creation is a design problem.

The principle of location transparency in this context also means that where to create objects should be determined in such a way that neither client nor server objects have to be changed when a different host is designated to serve new objects.

If a host becomes overloaded or needs to be taken out of service, objects hosted by that machine need to be moved to a new host. This is referred to as *migration*. Migration is a problem that does not exist with local objects. Migration has to address machine heterogeneity in hardware, operating systems and programming languages.

Objects in a non-distributed application may be deleted implicitly by garbage collection techniques, which are available in Java, Eiffel or Smalltalk. As we will see in Chapter 9, distribution complicates the application of garbage collection techniques, since it would require that objects know how many other distributed objects have references to them. It is rather expensive to achieve referential integrity in a distributed setting and therefore most distributed object systems do not fully guarantee referential integrity. This has implications on the design of client objects, as they have to be able to cope with the situation that their server objects are not available any more.

In summary, the distributed object life cycle has to consider object creation, migration and deletion. We will discuss techniques, interfaces and protocols for the distributed object life cycle in Chapter 9.

2.4.2 Object References

Object references are larger for distributed objects than for local objects.

In object-oriented programming, references are handles to objects that are implemented through memory addresses. The fact that pointers are memory addresses may be visible (as in C++) or hidden (as in Smalltalk, Eiffel or Java). In any case, pointers are rather lightweight structures and there is usually no big penalty for having many pointers to objects.

References to distributed objects are more substantial data structures. They need to encode location information, security information and data about the type of objects. Orbix, a rather lightweight implementation of CORBA, uses 40 bytes for an object reference. Hence, Orbix demands ten times the memory for an object reference than is needed for a pointer in a 32-bit architecture. The difference may increase to a factor of 100 for middleware that supports security and needs to pass authentication information with each object reference.

These considerations have a number of implications for the design of distributed object-based applications. First, applications cannot maintain large numbers of object references since they would demand too much virtual memory on the client side. Secondly, the distribution middleware has to know all object references and must map them to the respective server objects. Given the size requirements of object references it seems unfeasible to have a large number of server objects and a large number of references to them. When designing distributed object-based applications we have to bear this in mind and minimize the number of objects. We should choose the granularity of objects such that both clients and middleware can cope with the space implications of object references.

2.4.3 Request Latency

A distributed object request is orders of magnitude slower than a local method invocation.

An object request in an object-oriented programming language is implemented through a member function call. Performing a local method call on a modern workstation requires less than 250 nanoseconds. In addition, programming language compilers are built in such a way as to optimize method invocations by including the code in the calling code (inlining) when appropriate, thus eliminating the need for a call. Hence, the overhead of calling a method is negligible for the design of an application.