MR. ARCHITECTURE

by Kade Hansson

Building a Database Application

What is Mr Architecture?

Mr Architecture is a component architecture based on **Enterprise Java Beans** (**EJB**.) Enterprise Java Beans simplifies the development of medium to large scale applications by reducing the amount of low-level services the application developer needs to design and implement. It is a middleware tier at a higher level of abstraction than databases and transaction handling which leverages object-oriented design to support rapid application development.

Mr Architecture takes a part of EJB- arguably the most promising part in developing efficient applications based on the technology- and emphasizes still further the possibility of automating low-level considerations through the inference of bean relationships and the automatic generation of bean components.

The hallmark properties of Mr Architecture are:

- Simplicity
    - Bean provider supplies three components per business object type, with one being completely generated automatically
      (under EJB, the bean provider must also supply a deployment descriptor)
    - First component is partially defined through inheritance, partially defined through graphical modelling (e.g. *Rational Rose*) and partially defined by automatic code generation scripts in the modelling tool
    - Second component is partially defined through graphical modelling (e.g. *Rational Rose*) and partially defined by an automatic code generation script
    - Third component is generated entirely by a script in a graphical modelling tool (e.g. *Rational Rose*)
    - Deployment descriptor (describing bean interrelationships) is inferred given naming conventions
    - One-to-one mapping between business object types and database tables, which are synchronized by tools (e.g. *Rational Rose*)
- Excellent scalability
    - Java makes it easy to migrate to more powerful hardware to accomodate new demands in both the server and client
    - No limits on number of business objects in the architecture

  - No complex configuration needed to include new business objects...
    1. generate the required components direct from an object model using scripts within the modelling tool
    2. use the modelling tool to generate the database tables
    3. add two lines to the application's *MrContainerFactory*
    4. generate the Mr Architecture implementation classes using *DeploymentTool*
- High portability
  - Java offers write once, run anywhere capability
  - Server deployment can occur in any servlet enabled web server (e.g. *Apache Tomcat* or Sun Microsystems *iPlanet*)
  - Client deployment can occur in any Java-enabled web browser or computer with standalone Java runtime environment
- High speed
  - Unlike EJB, object interations occur at object granularity instead of method granularity (sacrificing concurrency and load balancing capacity for a performance trade off)
  - Java's *HotSpot* performance engine delivers object code which runs as fast as C
  - Transport layer uses Java object serialization for a compact and robust data representation
  - Transport layer capable of communicating a virtually unlimited number of related business objects in one or two handshakes
  - Server is multi-threaded for low latency
  - Server caches object representations of frequently accessed database records and record sets
  - Database connections are pooled so that during periods of high demand, many transactions from multiple clients can be processed simultaneously
- Modest resource demands
  - Mr Architecture demands no more resources than any other enterprise grade application. It requires:
    - space for recording metadata about business objects
    - space for recording transaction states
    - some space for caching (but flexible)
    - time as required for multiple concurrent transactions from application clients to progess

    - enough space for all the data in any concurrent transactions application clients might make
    - a reasonably efficient network transport
- Based on accepted standards
    - Java 2 Enterprise Edition (J2EE)
    - Java Database Connectivity (JDBC)
    - Enterprise Java Beans (EJB)
    - Java Servlet API
    - Java Transaction API

What are beans?

The core component type of a Mr Architecture application is a **bean**. A Mr Architecture bean (or **Mr Bean** for short) is a variation of an **Enterprise Java Bean** (**EJB**.) Specifically it is a slightly non-standard form of a **local entity bean** with **container-managed persistence** (**CMP**.) The full meaning of this will be explained in the next section.

A Mr Bean is a component which encapsulates the business logic of a business entity. It can be a client-side or a server-side component. The business logic is the code that allows the business entity to perform the tasks which an application needs to fulfill the needs of an enterprise. For example, **business methods** defined in an *Order* bean, such as `relateOrderItem` and `cancelOrder`, may implement part of the business logic of an ordering application for an online store.

Mr Beans hae the following characteristics:

- Mr Beans are **persistent-** they are able to exist beyond the lifetime of the application in which they run.
- They can participate in **relationships** with other Mr Beans.
- They have **primary keys** so they can be uniquely identified.
- They can be accessed by **multiple clients** simultaneously.

Persistence

A Mr Bean is an object with a persistent representation in a database. A class of Mr Beans of a particular business object type (say an *Order*) is represented in a relational database table of the same name. Each record in the table corresponds to a single object instance.

Relationships

Each Mr Bean may have relationships to other Mr Beans. For example, an *Order* bean may be related to other persistent business objects (such as *OrderItems*), and these relationships are mapped onto database relations in a way which should be largely transparent to the application developer and of only minor concern to the bean developer.

Primary key

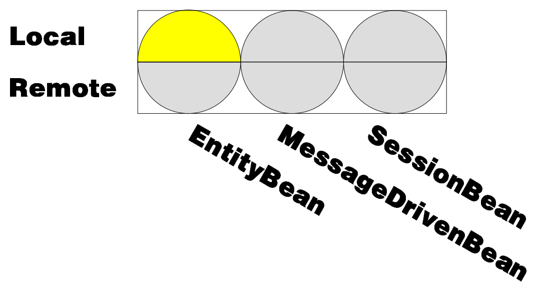
Just as a relational database table has a primary key field, so does a class of Mr Beans of a particular type. The primary key value assigned to a Mr Bean instance is a unique identifier among Mr Beans of the same type (e.g. you cannot have two *Order*s numbered 1,) just as database row's primary key value is unique in its table.

Multiple clients

Beans may be accessed simultaneously by multiple clients. Because one or more of these clients may wish the change a group of beans at the same time but in mutually incompatible ways, they may protect a sequence of changes by wrapping them in a transaction. That is, either all the changes in a single transaction become persistent, or none of them do.

How are Mr Beans different from EJBs?

An EJB is a server-side component encapsulating the business object of an application, but not necessarily a business entity. An EJB may be an **entity bean** (like a Mr Bean), a **session bean** or a **message-driven bean.** Additionally an entity bean may be **remote** or **local**. EJBs are supported by an implementation of the EJB specification, which must exist in each server supporting EJB applications.



🟨 Supported by Mr Architecture

⬜ In EJB, but not Mr Architecture

The Mr Architecture **container** is a slightly non-standard EJB implementation which does not support beans that are anything other than local entity beans with container-managed persistence. Put simply, it only supports Mr Beans, or at least beans which are capable of acting like Mr Beans.

- A Mr Bean is *local* because it is always accessed within the scope of a single Java virtual machine.
- It's an *entity bean* because it maps onto a business entity.
- It's *container-managed* because database accesses occur indirectly through the action of the architecture, and do not need to be manually coded. Mr Architecture cannot support beans with **bean-managed persistence** (**BMP**,) which implement their own database access methods.
- It's non-standard in two main areas:

1 Life-cycle diverges from the EJB 2.0 specification, and therefore transaction management is also divergent.

- The semantics of **create** are different. A Mr Bean does not exist in the database until it is explicitly stored subsequent to the create. A create occurs immediately irrespective of the current transaction state.
- The life-cycle events **load** and **store** are under client control. Mr Beans are therefore a form of entity bean with **client-managed persistence**[1]- a concept which EJB 2.0 does not define. A load occurs immediately irrespective of the current transaction state, while a store only occurs immediately outside of a transaction.
- Because transactions are managed only for bean life-cycle events, encapsulating business methods in transactions has no effect in an EJB application deployed under Mr Architecture. All business methods execute immediately, and only their life-cycle calls will be transaction managed (in accordance with the foregoing.)

2 A bean is referred to through the Java class names of a bean's components, and references are obained through the Mr Architecture container, not the Java Directory and Naming Interface (JDNI.)

It is not currently possible (under Mr Architecture 2.0) to write a generally useful Mr Bean which complies fully with the EJB 2.0 specification. Provision for this is likely to be made in future incarnations of Mr Architecture, along with facilities for converting old beans to use facilities compatible with current and future EJB specifications.

The Mr Architecture container, unlike other EJB containers, is not entirely transparent. However, like other EJB containers, it does provide services such as transaction management. Also, because Mr Architecture beans are always local, Mr Architecture containers exist in both server and client, wheras in a typical EJB application containing remote beans, the client communicates with beans in a server container.

A consequence of Mr Beans not being restricted to a server context is that multiple copies of a bean may exist in multiple clients (as well as in the server) to service simulataneous access. In such situations, the first copy successfully committed to persistent storage as part of a transaction will stand, and successive transactions involving copies of the same or older generations will not succeed.

[1] Client-managed persistence is a form of container-managed persistence where the container is involved in implementing life-cycle event handling, but not in determining when those life-cycle events occur. i.e. The client determines at what point a Mr Architecture bean is loaded or stored, but the Mr Architecture container does the work.

This behaviour is available only for beans with a `lastUpdateDateTime` field of type *java.util.Date* (backed in the database by an SQL timestamp data type.) Otherwise, the last bean committed wins persistence (so that it is possible for data to regress.)

It is the local nature of Mr Architecture beans which gives applications built for this architecture a greater efficiency over most other EJB applications, which rely heavily on **remote method invocation** (**RMI**.) This efficiency comes about because communication in Mr Architecture occurs in brief bursts, as beans are transmitted to and from clients. In EJB applications using remote beans, there will be a continuous stream of handshakes as the client interacts with a bean on a remote server. Mr Architecture therefore takes advantage of a larger granularity of network communication.

The overhead of using remote beans is widely acknowledged in EJB literature, and this overhead was indeed the motivation for the inclusion of local beans in EJB 2.0. In particular, many book and tutorial authors encourage the creation of business methods in remote beans which do larger volumes of work than they otherwise might to mitigate the overhead of remote interactions. Still other authors suggest using technologies other than EJB to write database applications, or advocate the transmission of local beans directly to clients. This latter approach is the one taken by Mr Architecture.

To not use EJB is to sacrifice the powerful object-oriented approach to data which it provides, and to workaround system overheads by increasing method granularity is counter to good object-oriented design pricip,les. Mr Architecture is therefore probably in line with the future evolution of EJB to support a wider variety of database applications more efficiently.

Bean development and deployment

Like EJB, Mr Architecture divides the work of constructing an application into two roles.

- The **bean developer** or **bean provider** is responsible for modelling and creating the bean components which may make up applications. The bean developer should aim for platform-independence and the capacity for reuse in the widest variety of applications possible.
- The **bean deployer** or **application developer** is responsible for assembing the beans into a complete working application. The bean deployer should be able to insert ready-made beans into their application and proceed to code business workflows and user interfaces based around them.

For small and medium sized projects it may be appropriate for team members to play both roles, perhaps dividing the work by functional area rather than along strict bean developer and deployer lines. However, even where this is the case, each team member should realise the requirements and aims of the role they are playing in each software development task.

Responsibilities of the bean developer

For the bean developer, a Mr Bean is made up of three components, two of which are different views of the same thing. The central component is the **entity bean class**, which consists of **life-cycle methods**, **persistent fields**, **relationship fields**, **business methods** and **internal methods and fields**. Related to this component is the **local component interface**, which exposes the persistent fields, relationship fields and business methods to clients, effectively hiding lifecycle and internal methods. The final component is the **home interface**, which provides for the creation of new beans and the finding of existing beans. In addition to these three mandatory components, support classes may also be provided.

In order to deploy an EJB, at least one additional component is required- an XML document called the **deployment descriptor** describing the bean's relationships. By contrast, Mr Beans do not require a deployment descriptor because Mr Architecture uses naming conventions in the bean classes to assist it in inferring bean interrelationships. For example, a one-to-many relationship from an *Order* bean to *OrderItem* beans can be expressed by including `getOrderItems` and `setOrderItems` methods in the *Order* bean.

A **remote interface** may also be supplied as part of an EJB. Even when present, a remote interface will not be implemented by Mr Architecture.

Mr Beans rarely exist in isolation, and so it is expected that many entities will be developed at once to support a single functional aspect of an enterprise. The approach this tutorial will take is to encourage the modelling of such entity groupings and their interrelationships using a UML class diagram in a graphical modelling tool. The classes on this diagram will first evolve into entity bean classes, which will then immediately give rise to corresponding local interfaces. At the same time, skeleton definitions of the home interfaces will be created, and final tweaking of these will give rise to complete and deployable Mr Beans.

Responsibilities of the bean deployer

It is also expected that the bean deployer will use a modelling tool to assist in deploying bean groups in the final application. In this case, the modelling tool needs to be sufficiently powerful to support the creation of a relational schema fragment corresponding to the class diagram. Tweaking this schema fragment to include field widths should lead to the creation of two further components per bean type which are needed before an application can be constructed from the beans- tables supporting each bean and its related beans, and metadata used to give presentation clues.

The final step is to have Mr Architecture create implementations of the modelled components which interact with their respective tables and the metadata in the complete application. This step is achieved entirely through the use of the Mr Architecture *DeploymentTool*. The final deployed bean consists of:

- three components from the bean developer- the entity bean class, the local interface and the home interface
- two components from the bean deployer- a database table and a metadata file
- three further components generated by the *DeploymentTool*- an entity bean implementation, an entity bean wrapper implementing client-managed persistence and a home interface implementation

Modelling entities and relationships

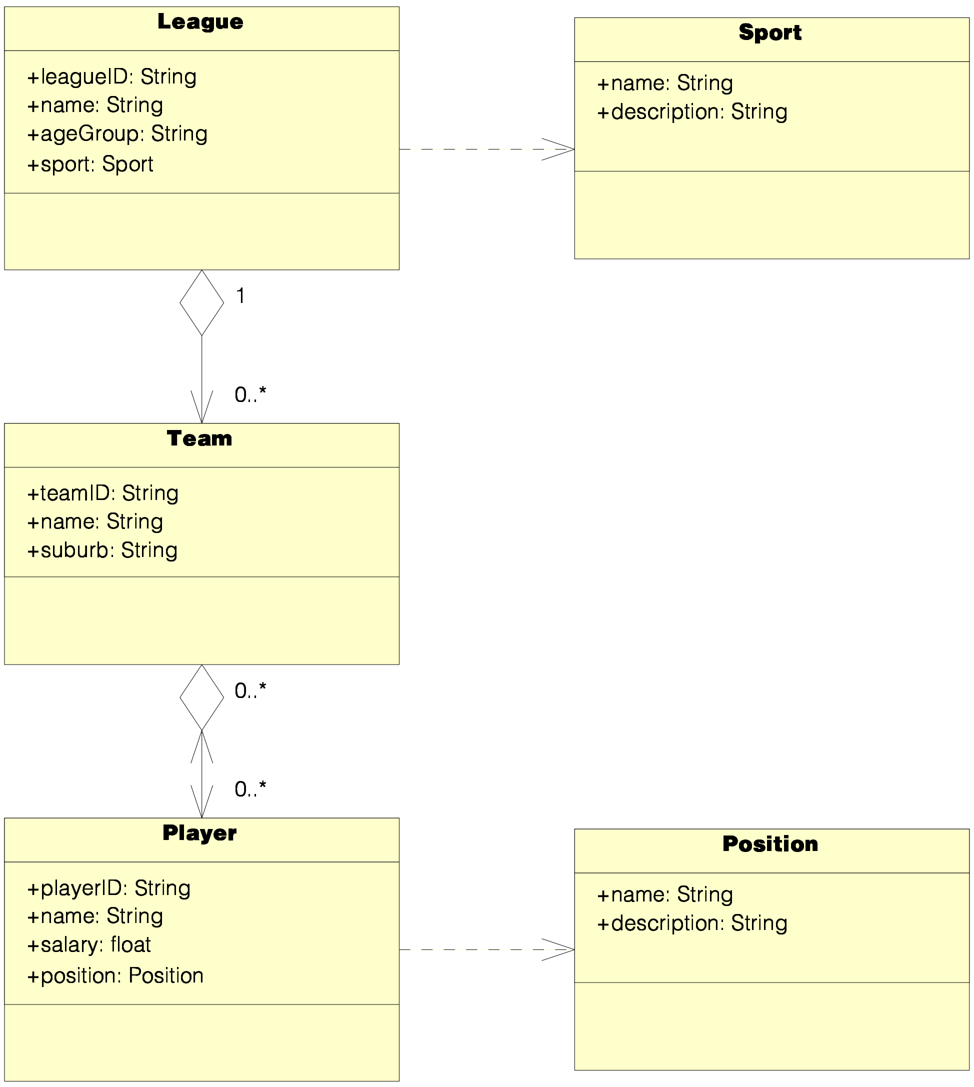
The first step the bean developer should take in modelling the functional area of an enterprise is to model candidate entities and relationships on a UML class diagram. There are many good reasons for doing this, not the least of which is the possibility of generating code directly from the modelling tool you choose. This tutorial will use Rational Rose, and some scripts for this tool are provided in the accompanying materials. You may use a different tool or modelling scheme if you prefer, but you will need to either translate the diagram into components manually or write suitable scripts for that tool to perform the transformations discussed.

The example we have used so far is that of an ordering application. This is used in the *Enterprise Beans* chapter of the *J2EE Tutorial*. However, because Mr Beans are specifically entity beans with container-managed persistence, this tutorial will follow the example application suggested in the chapter *Container-Managed Persistence Examples*, namely *RosterApp*.

RosterApp automates the enterprise of managing teams of players in several sporting leagues, such as a school board might need to arrange extra-cirricular sporting events between schools in a particular region. The main entities we will model will be *Player*, *Team* and *League*, but we may discover other minor entities as we construct the model. We will not model an entity like *School* simply because that would limit the applicability of our beans unnecessarily[2]. For example, it is possible that an association of amateur sporting clubs not affiliated with schools may want to use our beans in their application.

Let us make a first attempt at drawing the entities and their relationships on a UML class diagram. In a UML class diagram, we model entities using **classes** (shown as boxes) and relationships using **associations** (shown as solid lines with only arrowhead and diamond adornments.) Associations can be **unidirectional** or **bidirectional**, and directionality is shown by arrowheads**.** A unidirectional association implies that class at the end without the arrowhead has visibility of the class at the end with the arrow. A diamond indicates an **aggregation.** Multiplicities, bounding the number of instances of each class which can participate in a relationship, are shown using figures and ranges.

[2] It is likely we could find a way to include *School* (or perhaps the more general *Club*) as an optional bean which can cooperate with the rest of the beans in our model as and when required. This is left as an exercise for the reader.



The diagram above also shows the persistent fields of each of the entities we are modelling as UML **attributes**. You will notice we have also modelled classes for *Sport* and *Position*. The dashed arrows indicate that we only see that these entities are needed to make the details of *League* and *Player* complete, and we have not yet decided whether these pairs are related, only that they depend on each other. In particular, the choice to make *Sport* and *Position* entity beans is left open by this approach. We may, for example, choose to collapse the fields `name` and `description` in *Sport* into *League*, giving them names like `sportName` and `sportDescription`.

Relationship fields are not normally shown on a UML diagram- they are implied by the associations shown between classes. In fact, it is common for association ends to be labelled with **role names** as well as multiplicity. We have not used this feature of UML at this stage because Mr Architecture requires that we use the class name (possibly in plural form) as the role name, so it is possible to generate such annotations automatically using a script (as we shall shortly see.)

Modelling entities as Mr Beans

The first step in turning a vanilla UML class diagram with no stereotypes and limited adornments into a diagram containing entity bean classes and their relationships is to decide which classes are business entities. Mr Architecture limits us in some ways but provides more options as well:

- The decision to model a class as an entity bean does not need to pay a great deal of attention to inefficiencies of such an approach. Mr Beans carry very little extra overhead beyond a pure object. In other words, you can get many of the advantages of EJBs at very little cost.

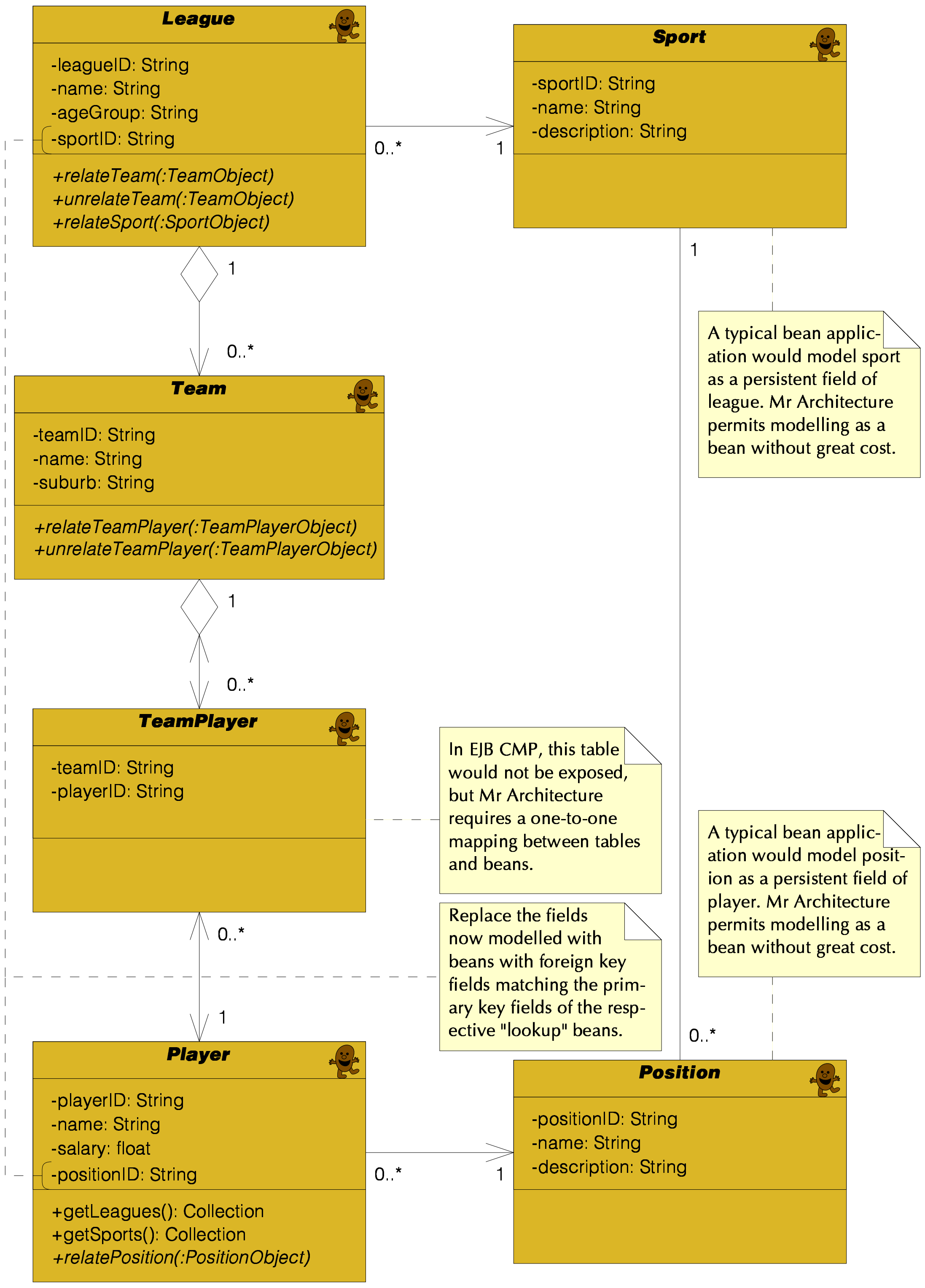
  →For this reason, we will choose to model *Position* as *Sport* as business entities with mappings to lookup tables in the relational schema. This descision might be different if we were using pure EJBs. Before we can transform *Position* and *Sport* into entities, we must add primary keys: we choose `positionID` and `sportID`, with `String` values, in line with our other keys. It would be equally valid to use numbers (**`int`** or **`long`** types), particularly if we wanted to take advantage of database-generated sequences.

- Mr Architecture forces us to use a one-to-one mapping between entities and relational database tables. This makes it exceedingly easy to see the mapping between the class diagram and a relational schema, but also encourages the modelling of some additional objects which would probably be better off hidden.

  →For this reason, we will be required to add an entity, *TeamPlayer*, to model the many to many relationship between *Team* and *Player*. This would only have been necessary in pure EJB if the association between team and player had an associated entity containing, say, a `playerNumber` field.

The next step is to add business methods.

- Relator methods are special abstract methods whose implementations are supplied by the Mr Architecture *DeploymentTool*. For example, the League bean has three relator methods:
  - `relateTeam` adds a new *Team* object to the one-to-many association between *League* and *Team*
  - `unrelateTeam` removes a *Team* object from the same association
  - `relateSport` changes the *Sport* participating in the one-to-one association between *Team* and *Sport*
- The `getLeagues` and `getSports` methods will allow a client application to find out which *Leagues* and *Sports* a *Player* participates in. These methods are not abstract, with implementations provided by the bean provider.


League
-leagueID: String
-name: String
-ageGroup: String
-sportID: String
+relateTeam(:TeamObject)
+unrelateTeam(:TeamObject)
+relateSport(:SportObject)
Sport
-sportID: String
-name: String
-description: String
0..*
1
1
A typical bean application would model sport as a persistent field of league. Mr Architecture permits modelling as a bean without great cost.
1
0..*
Team
-teamID: String
-name: String
-suburb: String
+relateTeamPlayer(:TeamPlayerObject)
+unrelateTeamPlayer(:TeamPlayerObject)
1
0..*
TeamPlayer
-teamID: String
-playerID: String
In EJB CMP, this table would not be exposed, but Mr Architecture requires a one-to-one mapping between tables and beans.
A typical bean application would model position as a persistent field of player. Mr Architecture permits modelling as a bean without great cost.
Replace the fields now modelled with beans with foreign key fields matching the primary key fields of the respective "lookup" beans.
0..*
1
0..*
Player
-playerID: String
-name: String
-salary: float
-positionID: String
+getLeagues(): Collection
+getSports(): Collection
+relatePosition(:PositionObject)
Position
-positionID: String
-name: String
-description: String
0..*
1

The entities in this diagram have had their fields assigned private visibility, have been made abstract, and have been assigned **entity bean stereotype**. You do not need to do this manually- the script introduced in the next section will do these tasks and more.

Modelling entities as *EntityBean* classes

The tasks we next need to perform to convert our modelled classes to entity beans are rather automatic.

- Add getter and setter methods for each persistent field.
- Add getter (or retriever) and setter methods for each relationship field, following the Mr Architecture naming conventions for such methods.
- Make fields have private visibility.
- Add EJB life-cycle methods, particularly `ejbCreate` and `ejbPostCreate`.
- Optionally, add a `validate` method.
- Make the entities abstract.

A script which does this and more for selected entities in a Rational Rose class diagram is provided in the course materials. It is called *CreateGetSet*. The result of running this script is shown on the next page.

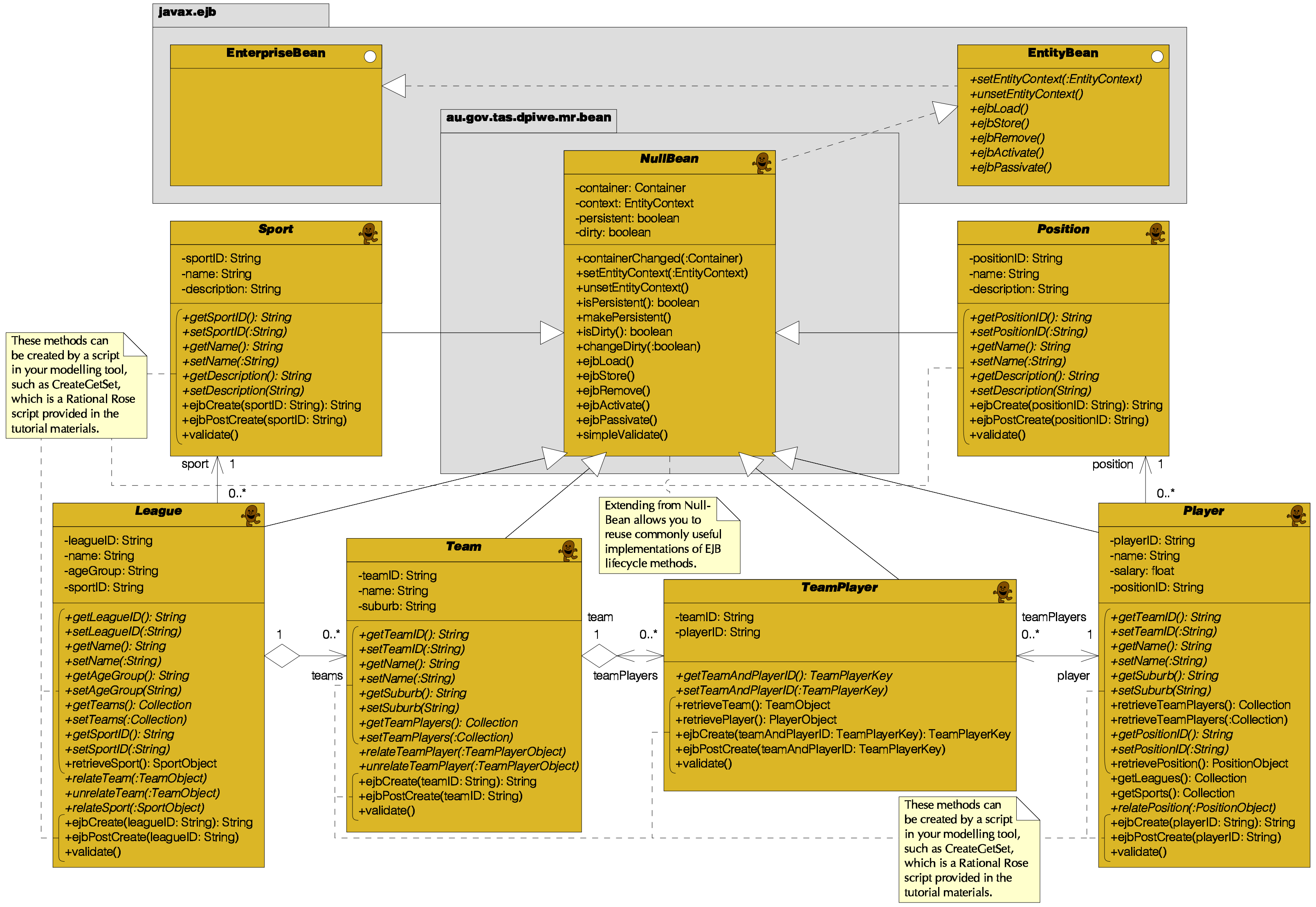
The main change you will notice is that the beans have all been connected to *NullBean* by a generalization relationship. The *NullBean* class is provided by Mr Architecture and gives empty implementations of most EJB life-cycle methods, except:

- `setEntityContext`, which has an implementation that sets the protected field `context` to reference the *EntityContext* object passed in as a parameter.
- `unsetEntityContext`, which has an implementation which sets `context` to null.
- The methods `ejbCreate` and `ejbPostCreate` cannot be specified in NullBean because their signatures depend on the primary key of the bean in which they are defined (and *NullBean* has no primary key.)

It also provides additional functionality which works only under Mr Architecture:

- `simpleValidate`, a method which does a metadata based validation of the persistent fields in the bean.
- `containerChanged`, a method which is called by the Mr Architecture container to inform a bean of its assignment to a *Container* instance. The method records a reference to the container in the protected field `container`.

The other changes involve adding getter, setter, retriever, `ejbCreate`, `ejbPostCreate`, and `validate` methods to each entity bean, and assigning role names to the visible ends of assocaitions. The `ejbCreate` and `ejbPostCreate` methods produced by the script have been fixed up for this diagram- the script uses *Object* as a type placeholder because it cannot know the primary key type of the beans. Additionally, the `ejbCreate`, `ejbPostCreate` and `validate` methods need appropriate implementations- these are to be supplied by the bean provider.


javax.ejb
EnterpriseBean
EntityBean
+setEntityContext(:EntityContext)
+unsetEntityContext()
+ejbLoad()
+ejbStore()
+ejbRemove()
+ejbActivate()
+ejbPassivate()
au.gov.tas.dpiwe.mr.bean
NullBean
-container: Container
-context: EntityContext
-persistent: boolean
-dirty: boolean
+containerChanged(:Container)
+setEntityContext(:EntityContext)
+unsetEntityContext()
+isPersistent(): boolean
+makePersistent()
+isDirty(): boolean
+changeDirty(:boolean)
+ejbLoad()
+ejbStore()
+ejbRemove()
+ejbActivate()
+ejbPassivate()
+simpleValidate()
Sport
-sportID: String
-name: String
-description: String
+getSportID(): String
+setSportID(:String)
+getName(): String
+setName(:String)
+getDescription(): String
+setDescription(String)
+ejbCreate(sportID: String): String
+ejbPostCreate(sportID: String)
+validate()
Position
-positionID: String
-name: String
-description: String
+getPositionID(): String
+setPositionID(:String)
+getName(): String
+setName(:String)
+getDescription(): String
+setDescription(String)
+ejbCreate(positionID: String): String
+ejbPostCreate(positionID: String)
+validate()
These methods can be created by a script in your modelling tool, such as CreateGetSet, which is a Rational Rose script provided in the tutorial materials.
sport
1
0..*
position
1
0..*
League
-leagueID: String
-name: String
-ageGroup: String
-sportID: String
+getLeagueID(): String
+setLeagueID(:String)
+getName(): String
+setName(:String)
+getAgeGroup(): String
+setAgeGroup(String)
+getTeams(): Collection
+setTeams(:Collection)
+getSportID(): String
+setSportID(:String)
+retrieveSport(): SportObject
+relateTeam(:TeamObject)
+unrelateTeam(:TeamObject)
+relateSport(:SportObject)
+ejbCreate(leagueID: String): String
+ejbPostCreate(leagueID: String)
+validate()
Team
-teamID: String
-name: String
-suburb: String
+getTeamID(): String
+setTeamID(:String)
+getName(): String
+setName(:String)
+getSuburb(): String
+setSuburb(String)
+getTeamPlayers(): Collection
+setTeamPlayers(:Collection)
+relateTeamPlayer(:TeamPlayerObject)
+unrelateTeamPlayer(:TeamPlayerObject)
+ejbCreate(teamID: String): String
+ejbPostCreate(teamID: String)
+validate()
Extending from NullBean allows you to reuse commonly useful implementations of EJB lifecycle methods.
TeamPlayer
-teamID: String
-playerID: String
+getTeamAndPlayerID(): TeamPlayerKey
+setTeamAndPlayerID(:TeamPlayerKey)
+retrieveTeam(): TeamObject
+retrievePlayer(): PlayerObject
+ejbCreate(teamAndPlayerID: TeamPlayerKey): TeamPlayerKey
+ejbPostCreate(teamAndPlayerID: TeamPlayerKey)
+validate()
Player
-playerID: String
-name: String
-salary: float
-positionID: String
+getTeamID(): String
+setTeamID(:String)
+getName(): String
+setName(:String)
+getSuburb(): String
+setSuburb(String)
+retrieveTeamPlayers(): Collection
+retrieveTeamPlayers(:Collection)
+getPositionID(): String
+setPositionID(:String)
+retrievePosition(): PositionObject
+getLeagues(): Collection
+getSports(): Collection
+relatePosition(:PositionObject)
+ejbCreate(playerID: String): String
+ejbPostCreate(playerID: String)
+validate()
1
0..*
teams
team
1
0..*
teamPlayers
teamPlayers
0..*
1
player
These methods can be created by a script in your modelling tool, such as CreateGetSet, which is a Rational Rose script provided in the tutorial materials.

Mr Bean field accessor method summary

A field is either a persistent field, listed in the second section of a UML class box, or a relationship field, one corresponding to each role name on associations giving visibility to other classes.

- A persistent field is named for the database column it is mapped to in the relational schema.
- A relationship field is named for the entity bean at the target end of the relationship, with "s" added in the case of one-to-many relationships to form the plural.[3]

It is conventional to name all fields with a lower-case letter for the first word or sequence of initials, and upper-case letters for the first letter in each remaining words or subsequent initials, with lower-case letters for the remainder of each word. When the field is used as part of a method name, in an SQL query, or in a Mr Architecture constrained field list, it is conventional to name the field with an upper-case letter for the first letter in each word and for each initial, and lower-case letters for the remainder of each word. Words like "ID" and "OK" are usually treated as sequences of initials, so that names like `sportID` and `idNumber` follow this convention.

Under EJB 2.0 CMP, all relationship getters and setters follow the same naming convention:

- get<*role-name*> to obtain the related bean at the other end of a many-to-one or one-to-one relationship (the return type is the local component interface for the foreign bean), or to obtain a *Collection*[4] of related beans at the other end of a one-to-many relationship (the *Collection* is composed of instances of the local component interface for the foreign bean).
- set<*role-name*> to change the related bean at the other end of a many-to-one or one-to-one relationship (the parameter type is the local component interface for the foreign bean), or to submit a new *Collection*[4] of related beans to populate the other end of a one-to-many relationship (the *Collection* is composed of instances of the local component interface for the foreign bean).

Because Mr Architecture does not use deployment descriptors, does not support the modification of relationship collections, and also because Mr Architecture allows all relationships defined by getters and setters to be automatically populated as if they were aggregations, it introduces some additional method types: retrievers and relators.

[3] Mr Architecture does not (as of version 2.0) make exception for any words, but it may in future make exception for words ending in "s". e.g. a many-to-many relationship with visibility from *Employee* to *Boss* could be represented by a field named bosses.

[4] Mr Architecture 2.0 also allows *List*, *Set* and *Map* to be used instead of *Collection*. Where a *Map* is used, the mapping is from primary keys to instances of the local component interfaces of related beans. EJB 2.0 only allows *Collection* and *List*.

- `retrieve`<*role-name*> is used where the relationship is not an aggregation, to prevent Mr Architecture from collating too much data as part of a `findAll`... method, and also where the relationship is not based on the primary key of the local bean.
- `relate`<*role-name-singular*> and `unrelate`<*role-name-singular*> modify a relationship by including or excluding beans. How and when these changes will be made persistent is described later in this tutorial.

The following table summarizes the method names used to represent associations with various adornments:

..	1 or *[5] (on primary key *A*)	1 or *[5] (on foreign key *A*)
◇ 1 (on foreign key *B* named for primary key *A*)	*get*<*foreign-bean-name*> *set*<*foreign-bean-name*> [*relate*<*foreign-bean-name*>][6]	
1 (on foreign key *B* named for primary key *A*)	`retrieve`<*foreign-bean-name*> [*relate*<*foreign-bean-name*>][6]	
1 (to same bean on primary key *B*)		*getParent*<*bean-name*> [`setParent`<*bean-name*>] (foreign key fields in *A* prefixed with word "parent")
1 (on primary key *B*)		`retrieve`<*foreign-bean-name*> [*relate*<*foreign-bean-name*>][6]
◇ * (on foreign key *B* named for primary key *A*)	*get*<*foreign-bean-name*>`s` *set*<*foreign-bean-name*>`s` [*relate*<*foreign-bean-name*>] [*unrelate*<*foreign-bean-name*>]	
◇ * (to same bean on foreign key fields *B*)	*getChild*<*bean-name*>*s* *setChild*<*bean-name*>*s* [*relateChild*<*bean-name*>] [*unrelateChild*<*bean-name*>] (foreign key fields in *B* prefixed with word "parent")	
* (on foreign key *B* named for primary key *A*)	`retrieve`<*foreign-bean-name*>`s` [*relate*<*foreign-bean-name*>] [*unrelate*<*foreign-bean-name*>]	
* (on foreign key *B*)		`retrieve`<*foreign-bean-name*>s

[5] Mr Architecture does not allow many-to-many relationships to be modelled directly. Model these as two one-to-many relationships as we have done for the many-to-many *Team* to *Player* relationship in the tutorial example.

[6] While there is no explicit unrelator method for a one-to-one or many-to-one relationship, one can remove a bean from a such a relationship using `relate`<*foreign-bean-name*>(**`null`**)

Retriever methods are written by the bean provider, while relators are generated by the *DeploymentTool*. Relators are not included automatically by the *DeploymentTool* to provide for relationships which can only be modified in certain ways (e.g. UML addOnly) or not at all.

Creating home and local component interfaces

Each Mr Bean, modelled so far as an entity bean class alone, actually has three components which need to be provided by the bean provider. The two components we are missing are the **local component interface** and the **home interface.**

- By in large, the **local component interface** arises pretty much directly from the entity bean class. It simply lists all the methods in the entity bean class except those dealing with entity bean life-cycle and implementation- collectively these methods are referred to as the business methods. (Business methods include getters and setters irrespective of the source of their implementation, be it the bean developer or the *DeploymentTool*.)

  →The purpose of the local component interface is to provide a means through which clients can access the business methods of an entity bean which coexists in the same virtual machine (VM).

- The **home interface** typically does not differ greatly from entity bean to entity bean. It lists methods which allow new entity beans to be created (**creator methods**), methods which allow existing beans to be looked-up based on a primary key value, and methods which allow more general search functions (**finder methods**). A home interface may also define some business methods of its own- for each method named <*home-method-name*> (not a creator or finder) in the home interface, a corresponding method called `ejbHome`<*home-method-name*> is expected in the entity bean class. A call to a business method in the home is mapped to a call to the corresponding home method in a **pooled entity bean**[7].

  →The purpose of the home interface is to provide a home in which all entity beans belonging to a particular class are housed. The home interface is therefore implemented as a singleton with respect to each Mr Architecture container. It is the point-of-call for external clients wishing to obtain references to business objects.

The local component interface can be generated entirely automatically, and a good starting point for a home interface can be constructed in a similar way. A script for Rational Rose which does the job given a selection of entity bean classes is called *GenerateEJB*. This can be found in the accompanying materials.

After the home and local component interfaces have been constructed:

[7] A **pooled entity bean** is an entity bean instance which has no assigned identity. i.e. it has no primary key value.

- The create method should take a primary key value as its only parameter. A creator method with no parameters can be used where the primary key value is generated by a database sequence.
- Ensure that the findByPrimaryKey method (and findAllByPrimaryKey method, if required) also takes a single parameter of the primary key type. Here, as for create, the script assumes *Object* type.

The script also generates findWhereFieldsEqual and findAllWhereFieldsEqual methods. These are additional finders for which *DeploymentTool* provides implementations. The result of a findWhereFieldsEqual is a *Collection*[4] of beans of the type to which the home corresponds. These beans are obtained by querying relational database storage with the constraint that only records where the non-null parameters (named for fields in the bean) are equal (or LIKE[8], in the case of *String* parameters) to the corresponding fields in the bean. To search for NULL values in the database, you will need to create a **nullable object**[9], and change the parameter type to accomodate such nullable objects.

The difference between find... and findAll... methods are the composition of the beans returned and the overhead of the call. A find... method does not populate bean relationships, so that the server (in the client container) or database (in the server container) will need to be called upon as each of these relationships is traversed. This is a lazy loading strategy, and is appropriate for most purposes. However, if you know that you are going to be traversing most if not all of the relationships in the returned beans, you can call upon a findAll... method, whose implementation utilizes an eager loading strategy which will populate bean relationships down to the level of retriever methods (which are used as a short-circuit to prevent the findAll... returning half of the database).

One restriction Mr Architecture places on these methods is that the findWhereFieldsEqual method with the longest signature must list all the foreign key fields that this bean is likely to be looked-up on. Additionally, a findAllWhereFieldsEqual method with the same signature is required if this bean is one which might populate a relationship in a bean from a findAll... method in another home.

You can affect the order of the beans returned by the findWhereFieldsEqual and findAllWhereFieldsEqual methods by including a static final variable called defaultOrderFields in the home interface. This field should be an array of *ConstrainedFields* of type Constrained_Sort, named for fields in beans, where the first *ConstrainedField*s in the array are most important to the sort order.

[8] LIKE is an SQL operator which does a Knuth Soundex comparison of textual fields with constrained text. Additionally, the characters underscore ("_") and percent ("%") are single and multiple character wildcards.

[9]

Java type	Nullable type	Examples
int, *Integer*	*NullableInteger*	**new** NullableInteger(**null**), **new** NullableInteger(1)
long, *Long*	*NullableLong*	**new** NullableLong(**null**), **new** NullableLong(1<<32)
String	*NullableString*	**new** NullableString(**null**), **new** NullableString("foobar")
Date	*NullableDate*	**new** NullableDate(**null**), **new** NullableDate(**new** Date())

Alternatively, the *DeploymentTool* will provide implementations for methods called `find[All]In`<*field-name*>`Order` which return beans ordered on a particular field, <*field-name*>, which takes precedence over the `defaultOrderFields`.

Mr Bean component summary

For each bean in our model we now have three components.

Bean name	Components (bean class, local component interface, home interface)
MrLeague	*League, LeagueObject, LeagueHome*
MrTeam	*Team, TeamObject, TeamHome*
MrTeamPlayer	*TeamPlayer, TeamPlayerObject, TeamPlayerHome*
MrPlayer	*Player, PlayerObject, PlayerHome*
MrSport	*Sport, SportObject, SportHome*
MrPosition	*Position, PositionObject, PositionHome*

The naming convention used by Mr Architecture for the various components is not the same as that recommended for EJB.

Component	Mr Architecture name syntax	EJB name syntax
Enterprise bean	`Mr`<*name*>	<*name*>`EJB`
Bean class subcomponent	<*name*>	<*name*>`Bean`
Local interface subcomponent	<*name*>`Object`	`Local`<*name*>
Local home interface subcomponent	<*name*>`Home`	`Local`<*name*>`Home`

Generating skeletal components

As we have now modelled all the components which the bean developer must supply, it is time to produce some code. Rational Rose, with the assitance of the scripts we have already seen, is capable of taking our model and generating skeleton code which requires very little modification to make it work.

The tasks that Rose automates for us at this step are:

- Each abstract class (with an italicized name) is converted to a Java source file which contains a single abstract class declaration.
    - Each abstract method (with an italicized name) is converted to a Java abstract method with no body.
    - Each concrete method is converted to a Java method with an empty body which we need to provide.
- Each interface is converted to a Java source file which contains a single interface declaration. Methods are transferred into the interface.

Some mundane tasks which may need to be done manually:

- If the model was not in an appropriately named subpackage of the model, a **package** directive may need to be added.
- **import** directives may need to be added if classes from the *javax.ejb*, *javax.transaction* or *au.gov.tas.dpiwe.mr.bean* were not present in the model from which the code was generated.
- Persistent fields should be removed from the entity bean class. These will be supplied by *DeploymentTool*. While we could have removed these from the model, this makes the model less clear.

  →Beware of reverse engineering entity bean classes which have had their persistent fields removed, as this will remove the fields from the model. If you merely comment out the persistent fields, you will able to put them back in order to reverse engineer the class correctly.

- **abstract** modifiers may need to be added, and bodies may need to replaced with semi-colons, for abstract methods.[10]
- **throws** clauses may need to be added to methods.[10]

Implementing business methods

Once all the components for a bean have been generated, the final step for the bean developer is the implementation of business methods. The only business methods we have defined in this example that aren't implemented by the *DeploymentTool* are retriever methods and non-abstract getter methods.

A typical implementation of a retriever method involves the use of a relationship field defined in the abstract bean class, which is set to null initially:

- If the relationship is not defined, perhaps due to the absence of a foreign key, return a null value (in the case of a one-to-one or many-to-one relationship) or the empty collection (in the case of a one-to-many relationship).
  e.g. `if (sportID==null) return null;`
- If the relationship field has already been populated, return the content.
  e.g. `if (sport!=null) return sport;`
- Obtain a reference to the foreign bean home via the Mr Architecture container. (The next section describes home interfaces in more detail.)
  e.g.
  ```
  SportHome home=container.getHomeForEntityBean(
        new ThinSport()
  );
  ```

[10]This is not necessary if you use the scripts provided in the course materials.

- Lookup the foreign bean(s) and store them in the relationship field. Do not use a `findAll`... method, as this retriver method (particularly in the case of an association which is not an aggregation) may be the only thing preventing a `findAll`... method from returning half of the database.
  e.g. lookup the *Sport* with a primary key matching the foreign key `sportID`.
  i.e. `sport=home.findByPrimaryKey(sportID);`[11]
- Return the value of the relationship field.
  e.g. **`return`** `sport;`

Instead of retriever methods, a bean provider may define a non-abstract getter method. However, the advantage of retriever methods is that they are called automatically by the *ClientContainer* upon receiving one or more beans as the result of a `findAll`... method. These calls happen as part of a batch, so that many relationships can be populated during a single network transaction. If this is not the behaviour you want, you must use a non-abstract getter method.

The `getLeagues` and `getSports` business methods defined in *Player* are examples of such non-abstract getter methods. The implementation of these methods will take a similar form to retriever methods except that they will not record the result of the finder method call in a relationship field.

Deploying beans for use in a client application

For the purpose of this section, we assume the client application is using an implementation of *MrContainerFactory* to construct the Mr Architecture containers. This approach allows us to use the *DeploymentTool* and not code our own tool which makes calls to *DeploymentContainer* in order to deploy beans.

Each Mr Bean may be packaged in an Mr Jar file, just as an EJB is packaged in an EJB jar file. However, in Mr Architecture it is more common to provide the source files to the bean deployer so they can compile them along with the rest of their project.

The differences between a Mr Jar file and an EJB Jar file are:

- A Mr Bean is specific to the Mr Architecture containers.
- A Mr Jar file contains no deployment descriptor.
- A Mr Jar file may contain a prototype metadata CSV file

If no prototype metadata is provided, then the bean deployer will create one from scratch if they are planning to use the *MetaData* interface provided by Mr Architecture for things like validation. The metadata CSV file is stored in the same directory as the bean's compiled classes and contains lines specifying the following information:

[11] Exception handling is not shown. In particular, *FinderException* would need to be caught, and handled by returning the null value. In the case of a one-to-many relationship retriever impementation, it would also be appropriate to rethrow an exception from a finder method by wrapping it in an *EJBException*.

<*field-name*>,<*database-type-name*>,<*width*>[,[<*precision*>][,
[<*human-readable-name*>][,[<*upper-case-flag*>][,[<*table-heading*>]]]]]

If the bean deployer has access to the Rose model of her schema, she can produce the first three or four fields automatically using the *MetaGen* script. If she does not have the Rose model of her schema, but has the Rose object model, as we do from our earlier modelling, she can use this to generate a schema which she can then mark-up with database types and widths.

This brings us to the next task a bean developer must carry out before a Mr Bean can be deployed in a client application- the creation of a schema fragment. As we have suggested, Rose allows us to create the schema representation direct from the object model. The only manual labour required is marking database types and widths, and primary and foreign key constraints. Rose can use the modelled schema to produce a DDL script, or it can connect directly to a database and execute the necessary statements.

Once the schema and metadata exist, a bean can be deployed. This is achieved by adding a line to the `populateDeploymentContainer` method of the application's *MrContainerFactory* implementation, recompiling this class and then running *DeploymentTool*.

The line for our *League* bean might look like:

```
container.addType(
        League.class,
        League.class.getMethod(
                "getLeagueID",
                new Object[] { String.class }
        ),
        League.class.getMethod(
                "setLeagueID",
                new Object[] { String.class }
        ),
        LeagueObject.class,
        LeagueHome.class
);
```

The deployed bean consists of three further components, which together constituate a bean type which can be added to Mr Architecture client and server containers alike. For this we add another line to the applications *MrContainer-Factory*, this time the `populateContainer` method, and recompile this along with the source files generated by *DeploymentTool*:

```
container.addType(
        ThinLeague.class,
        ThinLeague.class.getMethod(
                "getLeagueID",
                new Object[] { String.class }
        ),
        ThinLeague.class.getMethod(
                "setLeagueID",
                new Object[] { String.class }
        ),
        MrLeague.class,
        MrLeagueHome.class
);
```

Writing a client application

As we indicated in the last section, the best approach for constructing a Mr Architecture application is to base it around a *MrContainerFactory* implementation.

One of the first things your client application will do is instantiate an appropriate Mr Architecture *Container* instance to house the beans which will be manipulated during its execution. As we have seen, instantiating a *Container* not only involves constructing it, but populating it with the bean types it requires, which is why Mr Architecture encourages the use the factory method design pattern.

For our example, we will create a client which uses the *ServerContainer* directly. Normally a Mr Architecture client, as its name might suggest, uses a *ClientContainer* to indirectly connect to a remote *ServerContainer* via a *CommandServlet* instance. The way in which you use each type of container is identical once it is constructed.

-
  ```
  Container container=createServerContainer(
          new MrDatabase("jdbc:cloudscape:localhost","","")
  );
  ```

-
  ```
  Container container=createClientContainer(
          new URL("http://localhost:8080/mrroster/command/")
  );
  ```

One issue which constructing an application in this fashion avoids is the need for user authentication, because it is the *CommandServlet*s which provide this service. If we were a *ClientContainer* connecting via a *CommandServlet*, we would need to commit a *LoginEntityBean* instance and *SessionEntityBean* instance to persistent storage as part of our first transaction before we could do any operations. The creation and storage of a *SessionEntityBean* establishes a session. The creation and storage of a *LoginEntityBean* containing a valid username and password then authenticates the user for this session.

During the execution of our application we will perform transactions involving beans. Here are some examples of some transactions we may perform:

- e.g. Allow the user to start a new league.

    - Obtain the *MrLeagueHome*.

      i.e.
      ```
      LeagueHome leagueHome=
      container.getHomeForEntityBean(new ThinLeague());
      ```

    - Once the user has entered a `leagueID`, create a new *MrLeague* for this ID. Display a dialogue box and clear the field if the ID clashes with one that already exists.

      i.e.
      ```
      try {
              leagueHome.create(leagueID);
      } catch (DuplicateKeyException failure) {
              // Display a dialogue box
              . . .
      }
      ```

    - Allow the user to type the `name` and `ageGroup`, and store it in the bean.

      i.e.
      ```
      if (event.getSource()==nameBox) {
              league.setName(nameBox.getText());
      } else if (event.getSource()==ageGroupBox) {
              league.setAgeGroup(ageGroupBox.getText());
      } else . . .
      ```

    - Allow the user to enter a sport using a combo box populated with all *MrSport* beans.

        - Obtain the *MrSportHome*.

          i.e.
          ```
          SportHome sportHome=
          container.getHomeForEntityBean(
                  new ThinSport()
          );
          ```

        - Call `findWhereFieldsEqual` to obtain all *MrSports*.

          i.e.
          ```
          sports=sportHome.findWhereFieldsEqual();
          ```

- If the user clicks **OK**, call store; **Cancel**, call remove.

  i.e.
  ```
  if (event.getSource()==okButton) {
        league.store();
        . . .
  } else if (event.getSource()==cancelButton) {
        league.remove();
        . . .
  }
  ```

- e.g. Allow the user to remove a team.

  i.e.
  ```
  TeamHome teamHome=container.getHomeForEntityBean(
        new ThinTeam()
  );
  TeamObject team=teamHome.findAllByPrimaryKey(teamID);
  Iterator teamPlayers=player.getTeamPlayers().iterator();
  container.begin();
  while (teamPlayers.hasNext()) {
        TeamPlayerObject teamPlayer=
        (TeamPlayerObject)teamPlayers.next();
        Collection relatedTeamPlayers=
        teamPlayer.retrievePlayer().retrieveTeamPlayers();
        if (relatedTeamPlayers.size()==1) {
              relatedTeamPlayers.iterator().next().remove();
        }
        team.unrelateTeamPlayer(teamPlayer);
  }
  team.removeUnrelated();
  team.remove();
  container.commit();
  ```

- e.g. Allow the user to remove a player from all teams.

  i.e.
  ```
  PlayerHome playerHome=container.getHomeForEntityBean(
        new ThinPlayer()
  );
  PlayerObject player=playerHome.findAllByPrimaryKey(playerID);
  Iterator teamPlayers=player.retrieveTeamPlayers().iterator();
  container.begin();
  while (teamPlayers.hasNext()) {
        ((TeamPlayerObject)teamPlayers.next()).remove();
  }
  player.remove();
  container.commit();
  ```

- e.g. Display a table linking players and their sports.

```
i.e. class PlayerSportTableModel
extends javax.swing.table.AbstractTableModel
{

    List players=new Vector(); // The players (may be dups.)
    List sports=new Vector(); // The corresponding sports

    PlayerSportTableModel(Container container) {
        Iterator playersAndSports=
        container.selectSiameseBeanObjects(
            "select Player.*, Sport.*
            from Player, TeamPlayer, Team,
            League, Sport where
            TeamPlayer.PlayerID=Player.PlayerID and
            Team.TeamID=TeamPlayer.TeamID and
            League.LeagueID=Team.LeagueID and
            Sport.SportID=League.SportID"
        ).iterator();
        while (playersAndSports.hasNext()) {
            Iterator playerAndSport= (
                (SiameseBeanObject)
                playersAndSports.next()
            ).getBeanObjects().iterator();
            players.add(
                (PlayerObject)playerAndSport.next()
            );
            sports.add(
                (SportObject)playerAndSport.next()
            );
        }
    }

    . . .

}
```