THEORY CENTER

A guide to using UML and Theory Center's eBusiness Smart™ Components to design and implement Enterprise Java Beans.

Modeling with eBSCs

A GUIDE TO USING UML AND THEORY CENTER'S EBUSINESS SMART COMPONENTS
TO DESIGN AND IMPLEMENT SERVER SIDE COMPONENTS

Modeling with eBSCs

Chapter

1

Introduction

Using UML to model Enterprise Java Beans.

Theory Center has developed an approach to creating Enterprise Java Beans(EJB) components by modeling them using the Unified Modeling Language(UML) and then generating Java source code. This technique utilizes a UML drawing tool, in this case Rational Rose™ , and creates an intermediate file that describes that model. That file is transformed into the Java classes that make up one or more EJBs.

Code generation from UML has long been recognized as a promising technology. This technique is powerful because it allows the designer to model the components in a natural way without being concerned with implementation-specific details.

Despite its promise, this technique has never been adopted widely for a number of reasons: The generated code is often thought to be inferior, and there is no easy way to generate the implementation of the business logic; Lack of round trip engineering support, which means that most tools can only be used to generate the first attempt at the classes; and an associated problem, often times the model and the code become unsynchronized and much of the model's value is lost.

The Theory Center's utilities solve these problems by going a step further. The utilities do not assume a direct mapping from the model to the underlying language constructs. The user models the business objects and the SmartGenerator™ creates a set of classes that implements these objects with reference to the Enterprise JavaBeans specification. Many of the laborious tasks of creating access methods and handling containment of references is automatically handled. The SmartGenerator also uses intelligent algorithms to generate sensible naming for collections and methods. In addition, it generates documentation for these classes using the same intelligent naming scheme. Because the SmartGenerator embeds code markers, it is possible for developers to add the business logic and then resynchronize those changes with the model. Figure 1 describes the steps in the process so that you will have a frame of reference for the discussions that follow.

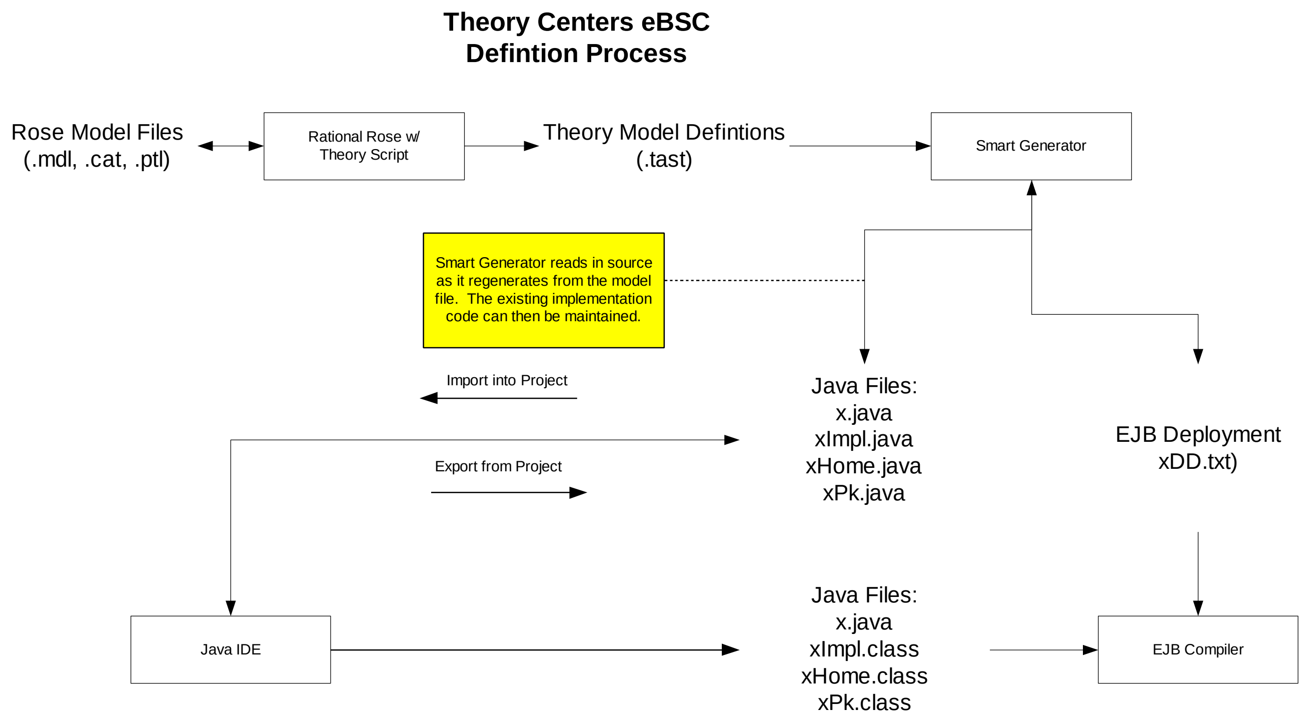


Figure 1

This document takes a step by step approach to explain our technique to creating EJBs from UML. It does not assume a familiarity with either of these topics and provides very introductory explanations of the key elements of each. While knowledge of these specific technologies is not assumed, familiarity with the underlying concepts of object-oriented design, distributed objects, and transaction services is.

There are a number of references in this document to various "Design Patterns" and "Analysis Patterns". There is a welcome trend towards documenting these axiomatic solutions to common computer science problems. Two books, that are titled as such, have taken steps to formally document these patterns and allow us to simply refer to the patterns without going into detailed explanation. These and other resources related to EJB and UML are listed in a bibliography at the end of this guide.

Conventions

```
Code samples are displayed in a fixed pitch font and are labeled as such.  Where code samples of
generated code are included, most of the generated comments have been removed for simplicity.
```

Document Organization

Chapter 2, The Foundation, provides a brief overview of the classes and interfaces that will form the basis for all of the EJBs that you will be creating.

Chapter 3 , Modeling Concepts, is a brief review of the UML notations that are interpreted by Theory Center's Smart Generator™ .

Chapter 4, Mapping UML to EJB, provides a detailed description of how Theory Center maps UML to EJB.

Chapter 5, Design Decisions, reviews some of the important design decisions and discusses their trade offs.

Appendix A, Using Rational Rose™ , talks specifically about how to install and use Theory Center's add-in for Rose™ .

Appendix B, Foundation Examples, contains a simple model which illustrates the use of the foundation package of Theory Center's eBusiness Smart ™ components.

Chapter

2

The Foundation

Using the Foundation Package with Stereotypes.

The foundation package is a set of classes from which Theory Center's components are built. These classes provide the building blocks for the value added features of our components. Most of the classes that are generated from the model are derived from classes in the Foundation package. To simplify the complexity of the UML diagrams, the Foundation package relationships are described through class stereotypes rather than inheritance. Each of these stereotypes is used to model certain behaviors and implies the presence of additional methods. This chapter discusses the Foundation package from a conceptual point of view so that users who are unfamiliar with Foundation topics can grasp them before using them in modeling.

Belongings

A Belonging, the simplest form of eBusiness Smart Component, is a lightweight, local object that can be serialized. A Belonging gets its name because it must "belong" to, or be acquired from, another object, typically a Session or Entity. It must be serializable so that it can be persisted with the class to which it belongs and passed remotely as a parameter.

One of the key characteristics of a Belonging is that it must be implemented using the Abstract Factory pattern. This means that for each belonging there is a home class, an interface, and at least one implementation of that interface. Because access to the object is through an interface, there is a guaranteed level of abstraction. This provides a great deal of flexibility because it means that you can substitute implementations. You could, for instance, make the object remote without changing the code that uses it. Alternatively, you might substitute different business logic at runtime by changing the implementation returned by the home class.

Implementing all these classes by hand is lot of work. The Theory Center has simplified the process by generating all of the necessary classes automatically. You can fully concentrate on modeling the attributes and methods so that they fit the needs of your business.

Sessions

Session components (implented as Session EJBs) are used to model service-oriented objects. The key concept is that a Session is an object that provides access to a service implemented in itself or somewhere else on the network. Attributes of a session are used only to configure it for use during the lifecycle of that session. It is important to note that the attributes of a Session are not persistent. The business methods are the most important part of a Session.

Sessions provide a way of remotely implementing business logic, thus extending the reach of your client application. For instance, when you need to perform an extended set of operations on a collection of remote objects it often makes sense to create a "Manager". The Manager object can be co-located with the objects it will be operating on. This will reduce the network overhead and latency.

Sessions are also commonly used to provide an interface to a legacy system or to a service that is pinned to a specific piece of hardware. The remote interface allows the client software to access the remote device as if it were local.

Finally, by wrapping a subsystem and factoring out the functions common to similar systems it is possible to provide a level of redundancy. An example of this would be the case where there are multiple providers of credit card validation services. These systems would likely have similar function but different implementations. By creating a common interface to use the different implementations, it is possible to load balance between them or substitute one for the other.

Entity

An Entity (implemented as an Entity EJB) is an object with staying power. Persistence is the key aspect of an Entity object. In its simplest form, an instance of an entity could be the equivalent of a single row in a relational database. This is an over-simplification because each Entity may include collections of attributes and implement business methods.

Entities are representative of the attributes of which they are composed. This is what distinguishes them from Sessions, which represent a collection of services. As a general rule Entities do not implement sophisticated business logic, instead, they are the components that are acted upon.

Configurable Entity

In addition to the standard qualities associated with an EJB Entity, Theory Center provides dynamic configuration. Dynamic configuration is the ability to add properties and methods at runtime and is provided by the Configurable Entity. The Configurable interface allows the programmer to associate a named value with the Entity. These values are persisted separately so that they are permanently associated with the object without affecting the underlying schema.

When the value stored in a Configurable Entity is a method, the result is the ability to exchange the implementation of a method dynamically or a "Pluggable Method" which is the implementation of the "Strategy" pattern.

Business Policy

Configurable Entities can be arranged in a hierarchy of successors. When such hierarchy is in place, a request to retrieve a value from a Configurable Entity triggers an upward search through the hierarchy of successors until a matching value is found or the top of the hierarchy is reached. This is the implementation of the "Chain of Responsibility" design pattern.

The combination of "Pluggable Methods" and the hierarchy of succession is what Theory Center calls Business Policy.

Workflow

For many business applications a simple mechanism to maintain internal state is all that is required to achieve a basic level of workflow. The Theory Center provides such a capability for defining and verifying the states and events that describe a business process. What this means to the developer is that they can represent this process as a state diagram and then verify the legitimacy of business method invocations with a single method call to ask for a transition. Adding a step is as simple as adding a new state. The engine will then enforce the rule that this step must be taken without changes to existing code.

Smart Features

Theory Center has designed and integrated advanced features into their components. These Smart Features considerably improve the ease of use and efficiency of the final system.

SmartKey

The EJB specification requires that for each Entity there is a class that represents the attributes of the primary key of that class. This Primary Key class is used to find and test the equality of instances of Entity objects. To accomplish these simple goals the EJB specification only requires that the Primary Key class must be serializable.

The SmartKey interface extends this functionality and requires the implementation of the Comparable interface from the java collection API. This is so that SmartKeys can be easily compared and stored in ordered lists. The result is that it is easy to model relationships that require the ordering of Entities.

The `toString` method of a SmartKey simplifies the implementation of profiling and debugging code.

SmartHandle

The EJB specification provides for the passing of lightweight references to Enterprise Java Beans through the use of Handles. A handle in EJB is an opaque type that can be converted to and from an EJB Object. A handle is required to implement a test for equality such that given two handles it is possible to determine if they refer to the same Session or Entity object.

For a Theory Center Entity component it is possible to create a SmartHandle that includes the object's associated SmartKey. Because the SmartKey implements the Comparable interface it is possible to order a list of smart handles without accessing the remote objects that they refer to. This simple mechanism greatly improves performance.

SmartValue

Each Entity is composed of the attributes that describe it. In order to encapsulate the remote objects all attributes must be read and written through accessor methods, typically named get<Attr> and set<Attr>. This has the negative consequence that retrieving the attributes of an entity may result in many remote method invocations. To alleviate this problem the Theory Center provides a convenience class, derived from SmartValue, that contains a copy of all the top-level attributes.

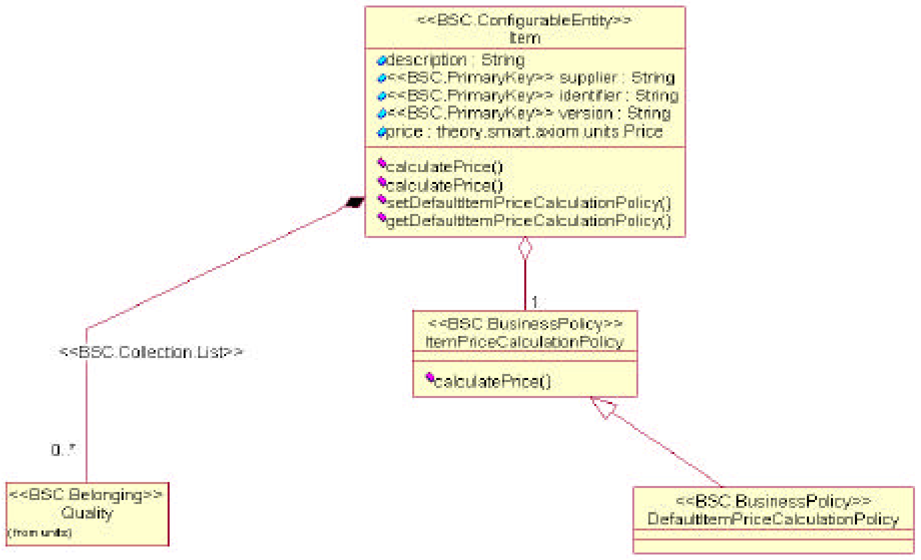
Chapter

3

Modeling Concepts

A review of some key UML notations.

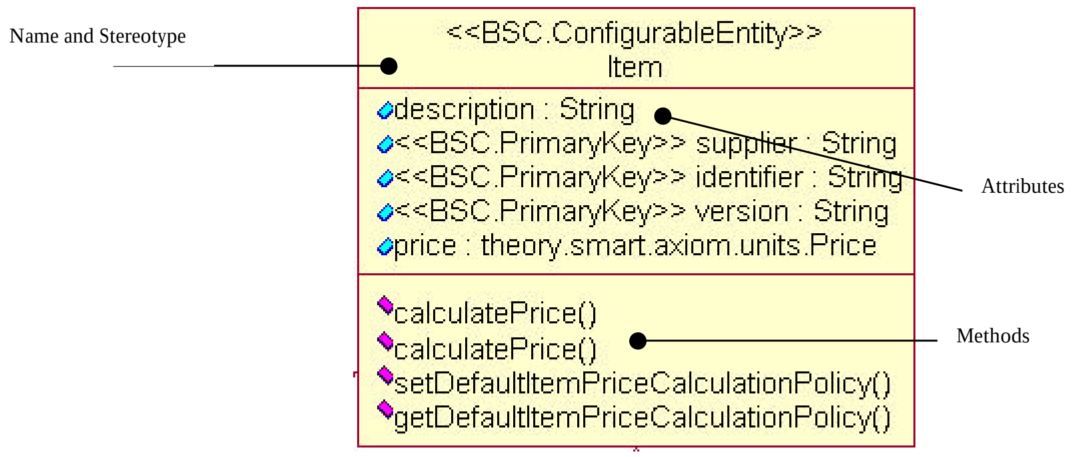
The Unified Modeling Language describes objects and their relationships graphically. Theory Center has adopted this industry standard as a mechanism for simplifying the design and implementation of Enterprise Java Beans. Before we get into the details we will review some of the UML notation from a higher level perspective. In this section we focus on the aspects of the notation that are of particular interest to Theory Center's SmartGenerator™ Figure 2: Sample UML, will serve as an example .



<<BSC.ConfigurableEntity>>
Item
description : String
<<BSC.PrimaryKey>> supplier : String
<<BSC.PrimaryKey>> identifier : String
<<BSC.PrimaryKey>> version : String
price : theory.smart.axiom.units.Price
calculatePrice()
calculatePrice()
setDefaultItemPriceCalculationPolicy()
getDefaultItemPriceCalculationPolicy()
1
<<BSC.BusinessPolicy>>
ItemPriceCalculationPolicy
calculatePrice()
<<BSC.Collection.List>>
0..*
<<BSC.Belonging>>
Quality
(from units)
<<BSC.BusinessPolicy>>
DefaultItemPriceCalculationPolicy

Figure 2: Sample UML

Classes and Stereotypes


Name and Stereotype
<<BSC.ConfigurableEntity>>
Item
description : String
<<BSC.PrimaryKey>> supplier : String
<<BSC.PrimaryKey>> identifier : String
<<BSC.PrimaryKey>> version : String
price : theory.smart.axiom.units.Price
Attributes
calculatePrice()
calculatePrice()
setDefaultItemPriceCalculationPolicy()
getDefaultItemPriceCalculationPolicy()
Methods

Each of the rectangles in a diagram is a representation of a class in UML. There are generally three compartments in each class box. A compartment may be left out if it is empty or if the details of the contents are not pertinent to a particular diagram. The latter is often the case when an object from another package is being reffered to.

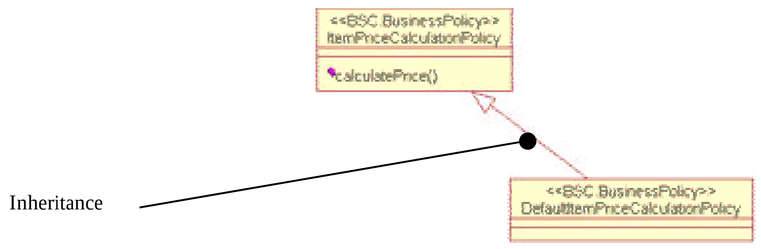
The upper most box holds the class name and its stereotype. A stereotype is a "sub-classification" of an element in the model. It is represented as the name of the stereotype enclosed in guillemets, as in <<stereotype>>. In the UML pretty much anything can be tagged with a stereotype. In this case the Item class is stereotyped as a Configurable Entity. This means that it would have the qualities of one as described in the section Entity.

Attributes are listed in the second compartment. In UML the name of the attribute is specified first followed by its type. The name and the type are separated by a colon. It is notable because it is different from the Java language. It works well for object oriented modeling which is generally an iterative process. Often times a designer will list the attributes of class with out specifying types the first time through. The same techinique holds true when specifying the arguments to a method. Note that as already mentioned, attributes can be decorated with a stereotype. The stereotype precedes the attribute and is embedded in guillemets as before.

The third and final compartment lists the methods. The only tricky thing here is that the return type is listed after the closing parentheses and is separated from the class definition with a colon. Often times the display of the parameters and the return value are supressed on the diagram because they consume a great deal of space.

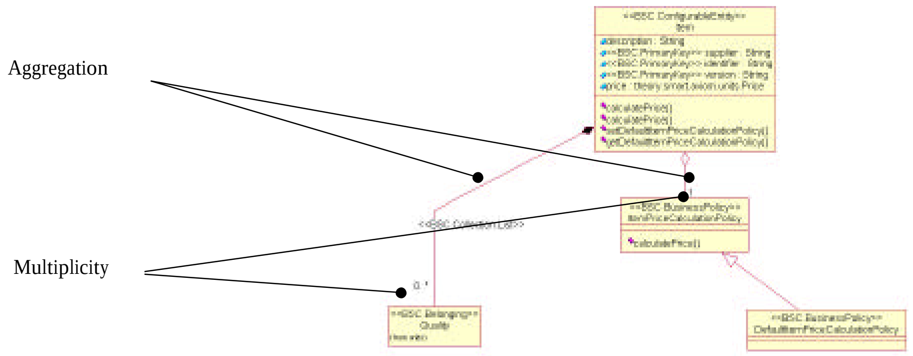
When specifying attributes and methods it is possible in the UML to connote whether or not they are private, protected, or public. The "tilted brick" icon to the left will have slight variations depending on this.

Inheritance



Inheritance is depicted on a UML diagram as an unfilled arrow that points from the subclass towards its parent. In this case the ItemPriceCalculationPolicy will have a calculatePrice method through inheritance. The subclass will share all of the properties and attributes of its parent.

Aggregation



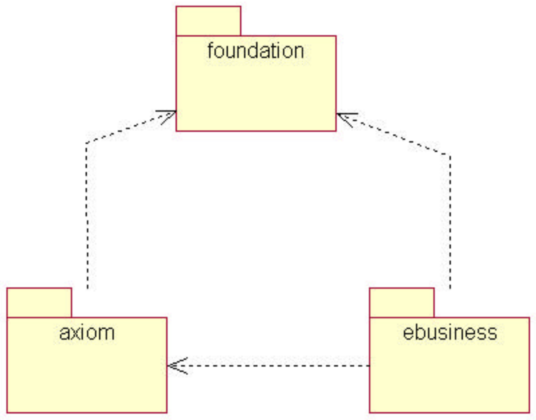
Aggregation
<<BSC ConfigurableEntity>>
Item
description : String
<<BSC PrimaryKey>> supplier : String
<<BSC PrimaryKey>> identifier : String
<<BSC PrimaryKey>> version : String
calculatePrice()
calculatePrice()
setDefaultItemPriceCalculationPolicy()
getDefaultItemPriceCalculationPolicy()
<<BSC CollectionList>>
Multiplicity
0..*
1
<<BSC BusinessPolicy>>
ItemPriceCalculationPolicy
calculatePrice()
<<BSC Belonging>>
Quality
<<BSC BusinessPolicy>>
DefaultItemPriceCalculationPolicy

Aggregation is used to describe a containment relationship between classes. This is an alternative to simply defining an attribute with the type of the class. In UML this means that the contained object shares a life cycle with the containing object. That is to say that the containing object holds the only reference to it and is responsible for removing the object upon when it, itself, is removed.

Aggregation is depicted in UML with a line that extends from the containing to the contained item. The line begins with an oblong diamond that specifies a category of containment. A hollow diamond is used to show that the object is being contained by reference. A solid diamond specifies that the object is contained by value.

It is also possible to specify a multiplicity for the object being contained. Options are 1 (one to one), 0..1 (optionally null for references), or 0..* (one to many). As with all other elements of the UML it is possible to stereo type the relationship. It is also possible to name an aggregation, although there is no example of this in the above diagram.

Packages



Packages are used to group classes and other packages in to a hierarchy. Each package will contains classes and/or other packages. When the classes of one package use the classes of another this is depicted as a dotted line with an arrow in the appropriate directions. This same "uses" notation can be applied to classes as well.

Chapter

4

Mapping UML to EJB

A detailed description of how Theory Center maps UML to EJB

Now that we have reviewed both UML and EJB it is time to marry the two and desribe how a UML diagram is transformed into eBusiness Smart Components. This chapter will describe in detail the Java code that will be generated as the result of making specific notations in a UML diagram.

Classes

Only classes in the model that are stereotyped as eBusiness Smart Component (eBSC) will result in the generation of java classes. There is not a one to one mapping between each class in the UML model and Java. In particular, all eBSCs are implemented using the Abstract Factory pattern. This means that there will be at least one interface and two Java classes generated for each eBSC that is modeled in UML. In addition, each Entity eBSC will have an associated Primary Key and Value class that is generated as well. The table below describes the mapping of classes based on the class stereotype.

Stereotype	Class Only	Interface	Home	Impl	PK	Value
BSC.Belonging		✓	✓	✓		
BSC.Session		✓	✓	✓		
BSC.Entity		✓	✓	✓	✓	✓
BSC.Workflow	✓					
BSC.BusinessPolicy	✓					

The naming convention for the generated classes is a follows:

- Class Only - The class will implement the respective interface and will be given the same name as the class in the model.

- Interface - The interface will be given the same name as the class in the model.

- Home - The Home interface/class will be the class name with the word Home appended (e.g. ItemHome). For the Session and Entity objects this will be an interface that is used by the EJB Compiler to generate the home implementation.

- Implementation - The Implementation class will be the class name with the letters "Impl" appended (e.g. ItemImpl).

- Primary Key - The Primary Key class will be the class name with the letters "Pk" appended (e.g. ItemPk).

- Value - The Value class will be the class name with the letters "Value" appended (e.g. ItemValue).

Primary Key and Value

For Entity Components there are two special classes that are generated. The Primary Key class is a Java class with public members for each of the attributes that are stereotyped as <<BSC.PrimaryKey>>. The primary key class is used by the create and findByPrimaryKey methods of the generated home class. Code Sample 1: Use of the PrimaryKey Class demonstrates usage of the primary key class.

```
public class OrderPk extends SmartKey implements java.io.Serializable
{
 public String key;

 public OrderPk()
  {
        super();
  }
        … more stuff here
}

public interface OrderHome extends SmartEJBHome
{
        public Order create(theory.smart.ebusiness.order.OrderPk orderPk )
               throws CreateException, RemoteException;
        Order findByPrimaryKey(theory.smart.ebusiness.order.OrderPk orderPk)
               throws RemoteException, FinderException;
}
```

Code Sample 1: Use of the PrimaryKey Class

The Value class is a Java class with public members for each of the attributes of the associated Entity. This includes attributes that are specified through aggregation. This class is used by the generated Value accessor methods. The purpose of these method is to simplify the retrieval of multiple attributes and reduce the overhead associated with remote method invocation. Code Sample 2 describes the usage of Value objects in Theory Center's generated code. Caution must be used when using the setByValue as there is no built in Entity locking. When using the setByValue on an Entity object it is important to realize that the attributes which are members of the primary key cannot and will not be updated. This is because as part of the identity of the Entity they are immutable.

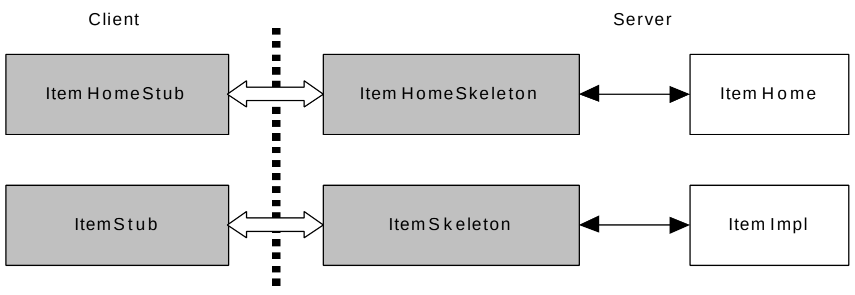
```
public class ItemValue extends SmartValue
{
  public String version;
  public String identifier;
  public String supplier;
  public String description;
  public theory.smart.axiom.units.Price price;
  public LinkedList qualities;

 protected ItemValue()
 {
   super();
  }
}

public interface Item extends ConfigurableEntity
{
  public ItemValue getItemByValue() throws RemoteException;
  public void setItemByValue(ItemValue value) throws RemoteException;

//…
}
```

Code Sample 2:Use of Value Objects

Interfaces, Homes, and Implementations

The Abstract Factory pattern requires that objects be accessed only through their interfaces and that the classes that implement those interfaces be acquired only through a factory class. The factory class in the case of EJB is referred to as a Home. This has slightly different implications for EJB components and Belongings.

When dealing with Session and Entity objects a separate tool --provided by the Application server vendor-- called the EJB compiler will be run and will create the appropriate proxies stubs and skeletons. At deployment



time the application server will be responsible for registering the home interface with the Java Naming and Directory Interface(JNDI) so that users of the EJBs will be able to create and find them.

For Belongings, the home, interface, and implementation will reside wherever they are instantiated. Belongings are always passed by value. When a belonging is used as the parameter to a method of a Session or Entity it will be serialized and then reinstantiated on the server. To make this happen the Java class associated with the belonging must be available in the class path on the server. The deployment implication is that the release of these classes must be coordinated between the client and the server.

The home interface is where finder methods reside. A finder method is one that locates one or more preexisting entities. Theory Center's SmartGenerator will automatically generate a finder method based on the primaryKey as in Code Sample 1: Use of the PrimaryKey Class. There are often times when it is necessary to create finders that search for entities based on the values of some other attributes. Adding an operation to the main class and stereotyping it as <<BSC.Home.Operation>> will accomplish this. The resulting method will be generated into the associated home class.

Attributes

For each attribute that is specified in the model a pair of accessor methods are generated. The get<AttributeName> method will retrieve the value of the attribute from the remote object and return it to the client. The set<AttributeName> method will pass the attribute to the remote object where it will be updated. In the case of an Entity the entire object will be marked as dirty such that the application server will know that

```
public interface Item extends ConfigurableEntity
{
  public String getSupplier() throws RemoteException;
  public String getIdentifier() throws RemoteException;
  public String getVersion() throws RemoteException;

  public String getDescription() throws RemoteException;
  public void   setDescription(String description) throws RemoteException;
  public theory.smart.axiom.units.Price getPrice() throws RemoteException;
  public void   setPrice(theory.smart.axiom.units.Price price) throws RemoteException;
}

public class ItemImpl extends ConfigurableEntityImpl
{
  public String version;
  public String identifier;
  public String supplier;

  public String description;
  public theory.smart.axiom.units.Price price;


  public String getDescription()
  {
    return (String) description;
  }
  public void setDescription(String description)
  {
    isDirty = true;
    this.description = (String) description;
  }
  public String getSupplier()
  {
    return supplier;
  }
  public String getIdentifier()
  {
    return identifier;
  }
  public String getVersion()
  {
   return version;
  }
  public theory.smart.axiom.units.Price getPrice()
  {
    return (theory.smart.axiom.units.Price) price.value();
  }
  public void setPrice(theory.smart.axiom.units.Price price)
  {
    isDirty = true;
    this.price = (theory.smart.axiom.units.Price) price.value();
  }
}
```

Code Sample 3: Generated Accessors

the changed values need to be persisted in the database[1]. This is true of Sessions to a lesser degree in that many application servers perform a serialization of Session beans for the purpose of optimizing the caching of Sessions.

One obvious omission in this example is that there are no methods for *setting* attributes that are stereotyped as part of the PrimaryKey for an entity. This is because those attributes are part of the identity of the object and as such they are immutable, cannot be changed.

Accessors are generated for belongings as well. The call to an accessor of a belonging is a direct call to the implementation object.

All of the attributes must be serializable. This also ensures that they can be persisted.

Aggregation

Aggregation allows for the defintion of an attribute of a class by drawing a line between it and another class which will be a included as a member. The following rules describe the allowable notations:

- A Belonging may only be contained by value (solid diamond).
- An Entity may only be contained by reference (hollow diamond). In such cases the attribute is stored as a SmartHandle.
- A Workflow is similar to a belonging and is always contained by value. A workflow is persisted using a WorkflowContext.
- A BusinessPolicy is similar to a belonging and is always contained by value. At present the accessors for the BusinessPolicy must be explicitly specified as business methods.
- If an aggregation is named, that name will be used when generating the accessors for that attribute. This is necessary so that multiple relationships to the same class can be modeled.
- If an aggregation is not named the accessors will be generated based on the name of the class that is being contained.
- Multiplicity may be defined as described in the section on Collections and Iterators.

Collections and Iterators

One of the most challenging issues when designing distributed object systems is implementing one to many relationships between objects. When modeling eBSC in UML such relationships are described by stereotyping

[1] While the "isDirty" attribute is specific to BEA WebLogic™ , it is expected that most application servers will support similar functionality.

either an attribute or an aggregation with a multiplicity of zero or more. Theory Center has based the collection mechanisms on the Java 1.2 collection API. When an aggregation relationship is stereotyped as a particular collection type, the internal attribute reflects that choice and the appropriate accessors are generated. The table below describes the options, a brief description of their usage, and the JDK 1.2 class upon which their implementation is based. The JDK documentation will provide significantly more detail as to the features of each collection type.

Table 1: Collection Stereotype Mappings

BSC.Collection.Set	A collection that contains no duplicates in which there is no implied ordering.	java.util.Collection.TreeSet
BSC.Collection.Array	An ordered collection that is stored as contiguous elements. This allows for optimal random access so that operations like re-sorting can be executed quickly.	java.util.Collection.ArrayList
BSC.Collection.List	An ordered list that optimizes insertions at the ends.	java.util.Collection.LinkedList
BSC.Collection.Map	A collection that is indexed by string and optimized for quick lookup. Iteration will be in ascending order according to the natural sort method.	java.util.Collection.TreeMap

The accessors for collections are generated for each stereotype as described in Table 2: Generated Accessors by Stereotype. This table uses a short hand syntax to convey which accessors are generated when a given stereotype is chosen. The token <Attribute> is replaced by the name of the attribute or aggregation as specified in the model. In the case of methods that accept or return a collection, the type is stereotype specific as defined in Table 1: Collection Stereotype Mappings. The details of the parameters and return values are implied so that the table itself can be concise. While there is no true inheritance relationship, it should be considered that Set serves as a basis for Array, which is a basis for List. Map is different in that it supports lookup by key.

All collections support the use of RemoteIterators. A RemoteIterator is stored on the server and is used to selectively retrieve the members of a collection. Their use requires overhead and they should be used only when appropriate. One such case arises when the contents of the collection are large and the client wishes to retrieve only a subset of the collection at a time.

In the case where an aggregation to an entity is specified by value, an additional group of methods is generated. These methods simplify the maintenance of the ownership relationship by ensuring that the underlying Entity is removed from it's home in conjunction with the removal of its reference from the list. The converse, add by value, is not supported because it would require that the containing entity be aware of the home of the entity to be added.

The Set provides methods for adding and removing attributes from a collection, it provides a "bag" type collection mechanism. The Array provides random access methods and is optimized for random access by integral position, for this reason it is especially useful when multiple sort orders are required. The List provides random access but is optimized for adding at the ends; this makes it good candidate for use when stacks or queues are called for.

The Map makes it possible to index a collection by a String.

Table 2: Generated Accessors by Stereotype

	Accessors	Iterator Methods	Entity by Value

Set	`add<Attribute>`
	`add<Attributes>(<CollectionType>)`
	`contains<Attribute>`
	`is<Attributes>Empty`
	`removeAll<Attributes>`
	`get<Attributes>():<CollectionType>`
		`create<Attribute>Iterator`
		`hasNext<Attribute>`
		`getNext<Attribute>`
		`remove<Attribute>At`
			`remove<Attribute>ByValue`
			`remove<Attributes>ByValueAt`
			`removeAll<Attribute>ByValue`

Array	**`<All of  Set > +`**
	`add<Attribute>( int position,…)`
	`set<Attribute>( int position, …)`
	`get<Attribute>( int position)`
	`get<Attributes>( int from, int to)`
	`remove<Attribute>( int position)`
	`indexOf<Attribute>`
	`lastIndexOf<Attribute>`
		`<All of Set> +`
		`add<Attribute>At`
		`set<Attribute>At`
		`getNext<Attribute>`
		`getPrevious<Attribute>`
		`getNext<Attribute>Index`
		`getPrevious<Attribute>Index`
			`<All of Set> +`
			`remove<Attribute>ByValue(int)`

List	**`<All of Array> +`**
	`addFirst<Attribute>`
	`addLast<Attribute>`
	`getFirst<Attribute>`
	`getLast<Attribute>`
	`removeFirst<Attribute>`
	`removeLast<Attribute>`
		`<All of Array>`	**`<All of Array> +`**
			`removeFirst<Attribute>ByValue`
			`removeLast<Attribute>ByValue`

Map	`put<Attribute>(String key)`
	`put<Attributes>(TreeMap)`
	`get<Attribute>ByKey`
	`get<Attributes>(String)`
	`contains<Attribute>Key`
	`contains<Attribute>Value`
	`remove<Attribute>ByKey`
	`removeAll<Attributes>`
		`create<Attribute>Iterator`
		`hasNext<Attribute>`
		`getNext<Attribute>`
		`remove<Attribute>At`
			`<Accessors are defined using WithKey instead of ByKey>`
			
			`put<Attribute>ByValue`
			`remove<Attribute>ByValueWithKey`
			`removeAll<Attributes>ByValue`

Chapter

5

Design Decisions

A review of some important design decisions and their associated trade offs

Now that we have a solid understanding of the basics it time to discuss some of the choices that you will need to make during the design process. While it would be nice to allow the modeler to design without consideration for implementation details, the reality is that truly good designs take into account deployment time issues.

Entities vs. Sessions

One of the most common issues when modeling EJB is related to legacy systems. These systems very typically provide an API or message-based protocol to allow external systems to access their functionality. The tendency in such cases is to simply model access to such systems as a Session component where each function in the API is a method of the Session bean. In the case of legacy systems that store complex business data and relationships, this is a mistake. In such cases it is best to model the internal objects as Entities where appropriate. This will provide for a more understandable system definition that takes advantage of the important caching and transaction services features of the EJB specification.

Implementing Business Logic in an Entity

In general, Session beans provide a sensible mechanism for implementing "workflow" related business logic. Workflow in this case is logic that coordinates the usage of any number of Entity beans. This has the performance-improving effect of reducing the network overhead associated with executing extended operations remotely. In the case where an Entity bean needs to perform complex business logic on classes that it references, it is best to implement that logic as a method of the Entity bean. This places the business logic where it belongs, with the data that it is manipulating.

Modeling from a Message Specification

There is a strong trend in the industry to translate message specifications, particularly XML DTDs, directly into business objects. While this may be convenient, it may not result in a clean description of the business objects.

This is similar to attempt to model a system based solely on the API. A better approach is to consider a message specification as providing insight into a single users perspective of the system. One approach is to consider the messages as method invocations to one or more underlying business objects. The contents of the message can then be modeled as attributes of various underlying objects.

Changing Method Signatures

If you change the signature of a method it will not be properly managed by the SmartGenerator. This is because the round trip engineering feature works by matching the exact signature of the method and the parameters. If a generated method is no longer present in the model it will simply be deleted, along with the associated implementation. To avoid this you must make matching changes in the model and the source code. As a consequence it is extremely important to consider the parameters to methods up front so as to avoid this problem.

Appendix

A

Rational Rose™ for Modeling eBSC

Installing and Using Theory Center's Rational Rose Add In.

Theory Center has developed an Add In for Rational Rose™ that will allow you to use it as platform for Modeling eBusiness Smart Components. This appendix describes the process by which Rose™ is used to execute the concepts described throughout this document.

Installation

If Rational Rose is already installed in the target Windows platform, the Rose eSC Link is installed as part of JumpStart. If Rational Rose is not already installed at the time of the JumpStart installation, the Link will not be installed. The process to install the link in the latter case is to reinstall JumpStart after installing Rational Rose.

The "Add-Ins/Add-In Manager" menu option will contain a checkable option for "Theory Center". When this option is checked the Theory Center Tools menu and stereotypes will be available for any model. eBSC stereotypes are associated with the appropriate object types and will be available where appropriate.

Creating a New Project

When you start Rational Rose it will present a list of new types of models you can create. To model using eBusiness Smart Components select "Theory Center eBSC". This will create an eBSC template model you can use.

Appendix

B

Foundation Examples

Data Modeling with the Foundation Package of Theory Centers' eBusiness Smart ™ Components

The Foundation Package provides the building blocks for the value-added features of eBusiness Smart ™ Components. This appendix contains a simple model which illustrates the use of the foundation package.

Prerequisites

To fully understand this appendix, you should be familiar with the JumpStart design, implementation, and deployment cycle as described in the *JumpStart Technical Tour*. You should also be familiar with the modeling concepts discussed in previous chapters of this document.

Scenario

Imagine a magical land called EJB where mystical creatures called HumanBeans live. The land of EJB has a strong economy because HumanBeans like to collect and count BeanieToys. HumanBeans socialize and collect their BeanieToys on ShoppingTrips.

On a typical ShoppingTrip, three HumanBeans named JoeBean, MaryBean, and EthylBean buy some BeanieToys. JoeBean buys three BeanBags, two BeanieBabies, and a JellyBean. MaryBean bean buys one GreenBean and five CoffeeBeans. EthylBean buys two BeanBags, and one BeanieHat. The next day, the three friends go on another ShoppingTrip and add more BeanieToys to their collection.

How do we use eBusiness Smart ™ Components to model the magical land of EJB?

UML Model

Lets examine the happenings in the land of EJB in a little more detail:

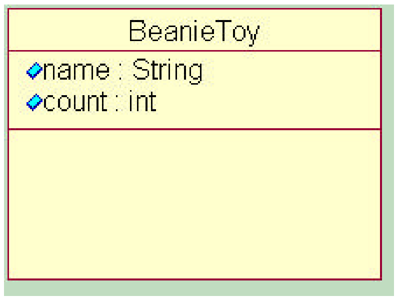
- **BeanieToys** are collected and counted by HumanBeans. HumanBeans only care about the name of the BeanieToy (JellyBean, BeanBag, CoffeeBean, etc.), and how many are owned.

- **HumanBeans** are identified by their name (JoeBean, MaryBean, EthylBean). HumanBeans remember their collection of BeanieToys from day to day and add to it on ShoppingTrips.

- **ShoppingTrips** are where HumanBeans gather to buy BeanieToys. The only way a HumanBean can buy a BeanieToy is through a ShoppingTrip. Although ShoppingTrips may be repeated from day to day, nothing about a prior ShoppingTrip influences the next ShoppingTrip.

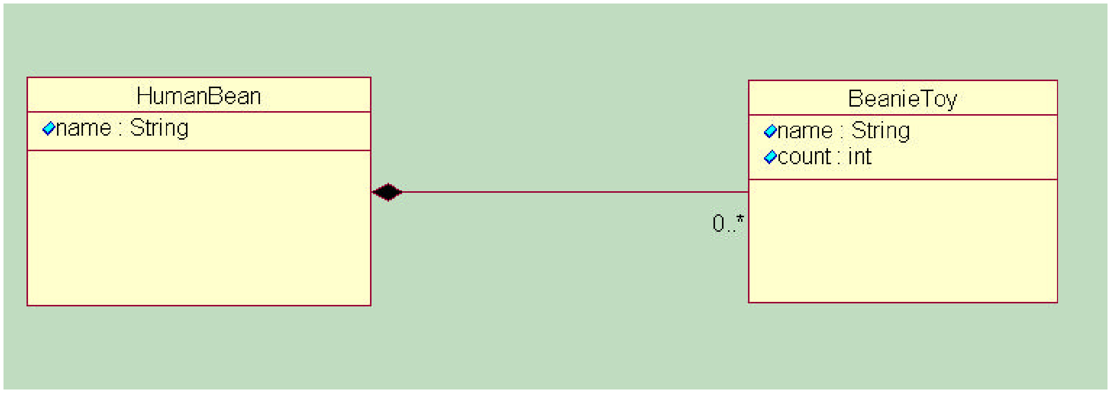
BeanieToys, HumanBeans, and **ShoppingTrips** each have distinct characteristics and activities. We can model each of them as a ***class*** in our UML Model of the magical land of EJB.

In UML, the class description of a BeanieToy looks like:



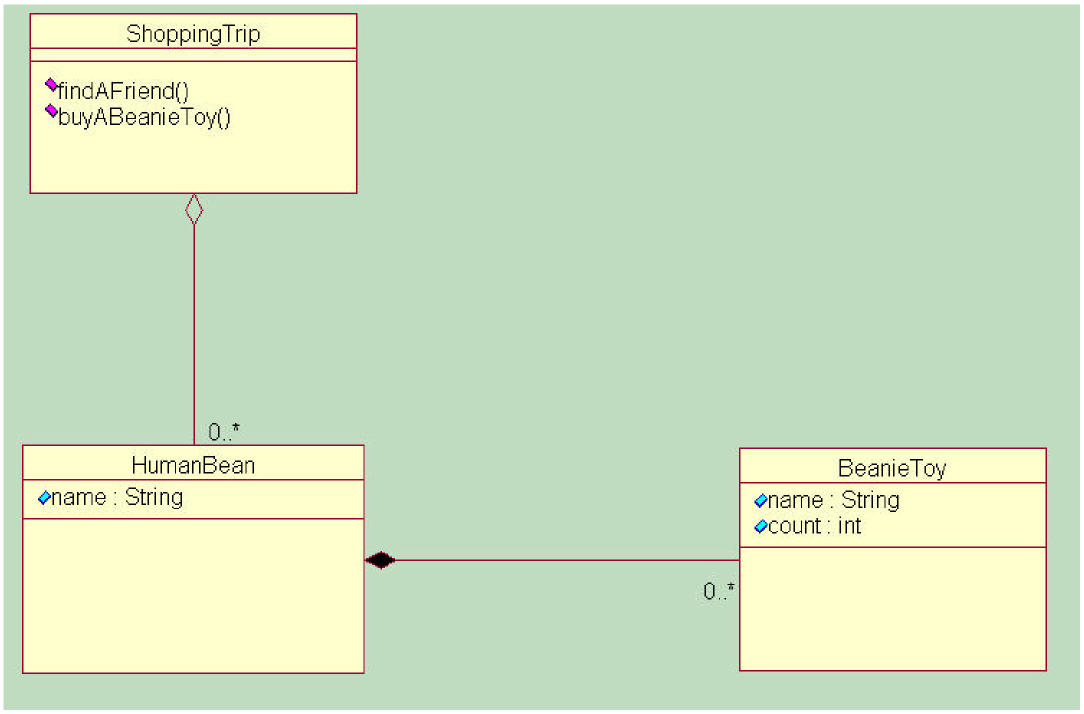
This class description says that the BeanieToy has two attributes, a name and a count.. BeanieToys perform no activities so the class has no methods.

In UML, the class description of a HumanBean looks like:



This class description says that a HumanBean has two attributes, its name and an aggregation of many BeanieToys. Note the "by value" relationship indicated by the solid diamond on the aggregation line.

In UML, the class description of the ShoppingTrip looks like:



This class description says that a ShoppingTrip has one attribute, an aggregation of many HumanBeans. It also has two methods, `void findAFriend(String humanBeanName)` and `void buyABeanieToy(String humanBeanName, String toyName)`. These methods are used to collect HumanBeans on a ShoppingTrip, and allow them to buy BeanieToys. Note the "by reference" relationship indicated by the hollow diamond. This means that HumanBeans exist outside the context of the shopping trip.

UML and eBSC

How do we include eBusiness Smart ™ Components in our model?

Lets look a little further at our software requirements:

- **BeanieToys** belong to HumanBeans. They must persist as long as the HumanBean who owns them persists.
- **HumanBeans** persist from day to day. They are distinguished by their name. They possess no sophisticated logic and must be on a ShoppingTrip to fulfill their mission, to buy BeanieToys.
- **ShoppingTrips** have no identifying attributes. They possess services to aggregate HumanBeans and allow them to buy BeanieToys.

Chapter 2 of this document introduced several classes from the foundation package of Theory Center's eBusiness Smart ™ Components. Recall three of these classes:

- **Belonging:** is a lightweight, local variable that can be serialized. It "belongs to" another object such as an Entity or Session, and may be persisted with the class to which it belongs.

- **Entity:** the key aspect of this class is *persistence*. They do not implement complex business logic, but rather represent a unique instance of attributes and other data.

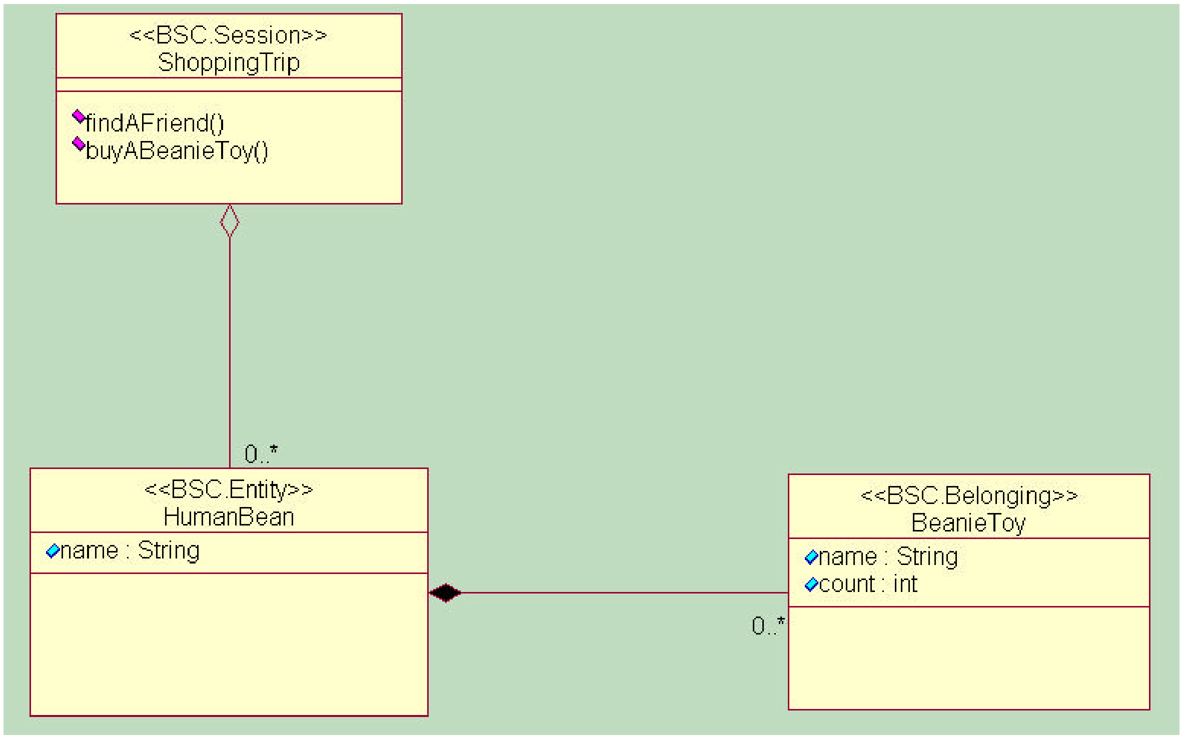
- **Session:** a session component models a service object. The business methods are the most important part of the Session, because they allow access to the services provided by the session.

Based on this review, we can see a **BeanieToy** has the characteristics of a **Belonging**. The BeanieToy should persist with the HumanBean that contains it. The BeanieToy has no activity outside the context of its HumanBean.

The **HumanBean** has the characteristics of an **Entity**. HumanBeans have no complex logic, but do have attributes and must persist to shop another day.

The **ShoppingTrip** has the characteristics of the **Session**. The emphasis of the ShoppingTrip is the services it provides, to collect HumanBeans and enable them to buy BeanieToys.

In UML, we apply the eBSC stereotypes to our classes. The magical world of EJB now looks like:

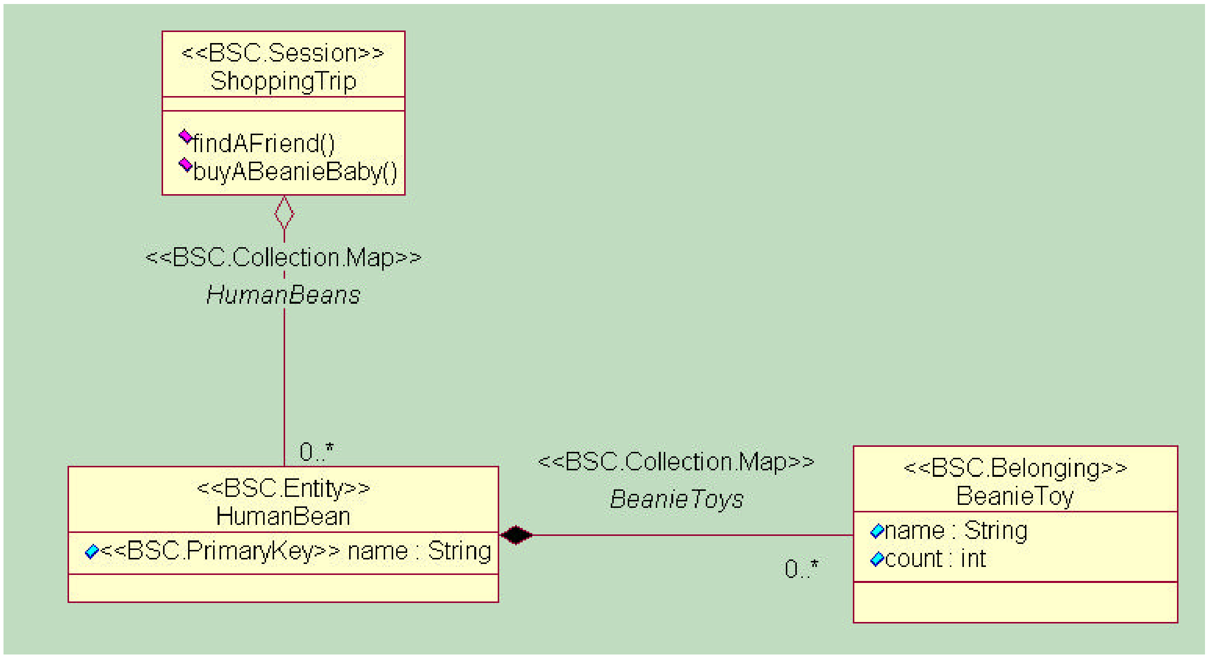


Theory Center's eBusiness Smart ™ Components help with two additional requirements of our software design:

- **Entity Primary Key:** the Enterprise JavaBeans specification requires that for each Entity there is a class that represents the attributes of the primary key of that class. Since the name of each HumanBean is its identifier, we apply the BSC.PrimaryKey stereotype on the name attribute of HumanBean.

- **Characterizing Aggregation:** eBusiness Smart ™ Components allow several ways to characterize containment: List, Array, Map, and Set. The BSC.Map stereotype defines Map containment: the contained object is accessed via a key. We characterize both aggregations in our model using the BSC.Map stereotype. We also provide a name for each aggregation.

The final UML for the land of EJB looks like:



Under The Hood with eBSC

Generate your own Smart ™ Components

Use the *JumpStart TechnicalTour* as a guide through this process.

1. Create a new package called `examples.umlebsc` in the Theory Center's EBusiness Smart Rational Rose model, *smart.mdl*.

2. Add the ShoppingTrip, BeanieToy, and HumanBean classes to the `examples.umlebsc` package. When you have completed this step, your Rational Rose model should look like the final UML Diagram in the previous section.

3. Export the model from Rational Rose using the Theory Center's plugin. Use the SmartGenerator to generate source code for the three new classes.

The SmartGenerator creates the following files:

Class	File	Use in Sun's EJB Specification	Description
BeanieToy	BeanieToy.java	n/a	Local, serializable interface that specifies the BeanieToy class.
	BeanieToyHome.java	n/a	Factory for instantiating BeanieToys
	BeanieToyImpl.java	n/a	Implementation of the BeanieToy interface. This implementation is used by the BeanieToyHome factory to create BeanieToys.
HumanBean	HumanBean.java	Remote Interface	Specification of HumanBean class.
	HumanBeanHome.java	Home Interface	Factory for instantiating and managing the life-cycle of HumanBeans.
	HumanBeanImpl.java	Bean Class	Server-side implementation of a HumanBean.
	HumanBeanPk.java	Primary Key	Unique key for locating an instance of a HumanBean.
	HumanBeanValue.java	n/a	Convenience class with top-level attributes of a HumanBean
ShoppingTrip	ShoppingTrip.java	Remote Interface	Specification of ShoppingTripClass
	ShoppingTripHome.java	Home Interface	Factory for instantiating and managing the life-cycle of HumanBeans.
	ShoppingTripImpl.java	Bean Class	Server-side implementation of a ShoppingTrip.

Under the hood of the BeanieToy

Examine the file BeanieToy.java:

```
public interface BeanieToy extends Belonging
{
    public String getName();
    public void setName(String name);
    public int getCount();
    public void setCount(int count);
}
```

Notice the following:

1. The set<attribute> and a get<attribute> method for each attribute from our model.

2. The Belonging interface requires that all subclasses implement the java.util.Comparable interface and implement java.io.Serializable.

Examine the file BeanieToyImpl.java:

```
public class BeanieToyImpl extends BelongingImpl implements BeanieToy
{
    public String name;
    public int count;

    protected BeanieToyImpl()
    {
        super();
        name = "";
        count = 0;
    }
    public String getName()
    {
        return (String) name;
    }
    public void setName(String name)
    {
       isDirty = true;
       this.name = (String) name;
    }
    public int getCount()
    {
        return (int) count;
    }
    public void setCount(int count)
    {
        isDirty = true;
        this.count = (int) count;
    }

    public Belonging value()
    {
        examples.umlebsc.BeanieToy aCopy = examples.umlebsc.BeanieToyHome.create();

        aCopy.setName(name);
        aCopy.setCount(count);
        return aCopy;
    }

    public int compareTo(Object o)
    {
        int result = 0;
        if (o instanceof BeanieToy == false)
        {
            throw new ClassCastException();
        }
        BeanieToy target = (BeanieToy) o;
        result = this.name.compareTo(target.getName());
        if (result == 0)
        {
            result = (this.count == target.getCount())
                     ? 0 : (this.count < target.getCount()) ? -1 : 1;
        }
        return result;
    }
}
```

Notice the following:

1. The BelongingImpl superclass provides a protected attribute isDirty, which is set to true whenever an attribute is set.
2. The implementation has an attribute variable for each attribute in the class.
3. The get<attribute> and set<attribute> methods.
4. The default constructor is protected. This visibility ensures only the BeanieToyHome class calls the constructor.
5. The implementation of the java.util.Comparable interface.
6. That the method value() returns a copy of this BeanieToy.

Examine the file BeanieToyHome.java:

```
public class BeanieToyHome implements SmartHome, java.io.Serializable
{
    public static BeanieToy create()
    {
        return new BeanieToyImpl();
    }
}
```

Notice the single create() method instantiates a new BeanieToyImpl() and returns it as the BeanieToy interface.

Under the hood of the HumanBean

Examine the file HumanBean.java:

```
 public interface HumanBean extends Entity
 {
     public HumanBeanValue getHumanBeanByValue() throws RemoteException;
     public void setHumanBeanByValue(HumanBeanValue value) throws RemoteException;

     public String getName() throws RemoteException;
```

```
    public RemoteIterator createBeanieToysIterator() throws RemoteException;
    public void removeBeanieToysAt(RemoteIterator rit) throws RemoteException;

    public boolean containsBeanieToysKey(String key) throws RemoteException;
    public boolean containsBeanieToysValue(examples.umlebsc.BeanieToy BeanieToys) throws
RemoteException;

    public examples.umlebsc.BeanieToy getBeanieToysByKey(String key) throws RemoteExcepti
    public TreeMap getBeanieToyses() throws RemoteException;
    public examples.umlebsc.BeanieToy getNextBeanieToys(RemoteIterator rit) throws
RemoteException;
    public int getNumberOfBeanieToyses() throws RemoteException;
    public boolean hasNextBeanieToys(RemoteIterator rit) throws RemoteException;
    public boolean isBeanieToysesEmpty() throws RemoteException;
    public void putBeanieToys(String key, examples.umlebsc.BeanieToy BeanieToys) throws
RemoteException;
    public void putBeanieToyses(TreeMap BeanieToyses) throws RemoteException;
    public void removeAllBeanieToyses() throws RemoteException;
    public examples.umlebsc.BeanieToy removeBeanieToysByKey(String key) throws RemoteExcep

}
```

Notice the following:

1. By inheriting from theory.smart.foundation.Entity, the HumanBean implements `java.io.Serializable` and `javax.ejb.EJBObject`.

2. The convenience methods `getHumanBeingByValue()` and `setHumanBeanByValue()` allow access to all the attributes of a HumanBean through one remote function call.

3. Read-only access to the name attribute via `getName()`. Recall the `<<BSC.PrimaryKey>>` stereotype we applied to HumanBean.name in the UML diagram. No corresponding `setName()` method was generated for this attribute because the primary key of an entity cannot be changed.

4. Implementation of the BeanieToys aggregation includes a `RemoteIterator` and methods that correspond to many methods in `java.util.Map` interface.

Examine the file HumanBeanImpl.java:

```
public class HumanBeanImpl extends EntityImpl
{
    public String name;

    public TreeMap BeanieToyses;
    private TreeMap BeanieToysIterators = new TreeMap(SmartComparator.getInstance());


    public HumanBeanImpl() throws CreateException
    {
        super();
    }
    public HumanBeanValue getHumanBeanByValue() throws RemoteException
    {
        HumanBeanValue value = new HumanBeanValue();
        value.name = name;
        value.BeanieToyses = getBeanieToyses();
        return value;
    }

    public void setHumanBeanByValue(HumanBeanValue value) throws RemoteException
    {
        removeAllBeanieToyses();
        putBeanieToyses(value.BeanieToyses);
    }

    public void ejbCreate(examples.umlebsc.HumanBeanPk humanBeanPk) throws CreateException
    {
        super.ejbCreate((SmartKey) humanBeanPk);


        name = humanBeanPk.name;

        BeanieToyses = new TreeMap(SmartComparator.getInstance());
    }

    public void ejbPostCreate(examples.umlebsc.HumanBeanPk humanBeanPk) throws CreateExcept
    {
        super.ejbPostCreate((SmartKey) humanBeanPk);
    }

    public void ejbLoad() throws java.rmi.RemoteException
    {
        super.ejbLoad();
    }

    public void ejbStore() throws java.rmi.RemoteException
    {
        super.ejbStore();
    }

    public void ejbRemove() throws java.rmi.RemoteException, javax.ejb.RemoveException
    {
        super.ejbRemove();
    }

    public void ejbActivate() throws java.rmi.RemoteException
    {
        super.ejbActivate();
    }

    public void ejbPassivate() throws java.rmi.RemoteException
    {
        super.ejbPassivate();
    }

    public void setEntityContext(EntityContext ctx) throws java.rmi.RemoteException
    {
        super.setEntityContext(ctx);
    }

    public void unsetEntityContext() throws java.rmi.RemoteException
    {
        super.unsetEntityContext();
    }

    public String getName()
    {
        return name;
    }

    public boolean containsBeanieToysKey(String key)
    {
        return BeanieToyses.containsKey(key);
    }
    public boolean containsBeanieToysValue(examples.umlebsc.BeanieToy BeanieToys)
    {
        return BeanieToyses.containsValue(BeanieToys);
    }
    public RemoteIterator createBeanieToysIterator()
    {
        Iterator it = BeanieToyses.values().iterator();
        RemoteIterator rit = new RemoteIterator(it);
        BeanieToysIterators.put(rit, it);
        return rit;
    }
    public examples.umlebsc.BeanieToy getBeanieToysByKey(String key)
    {
        return (examples.umlebsc.BeanieToy) BeanieToyses.get(key);
    }
    public TreeMap getBeanieToyses()
    {
        TreeMap map = new TreeMap(SmartComparator.getInstance());
        map.putAll(BeanieToyses);
        return map;
    }
    public examples.umlebsc.BeanieToy getNextBeanieToys(RemoteIterator rit)
    {
        Iterator it = (Iterator) BeanieToysIterators.get(rit);
        return (examples.umlebsc.BeanieToy) it.next();
    }
    public int getNumberOfBeanieToyses()
    {
        return BeanieToyses.size();
    }
    public boolean hasNextBeanieToys(RemoteIterator rit)
    {
        Iterator it = (Iterator) BeanieToysIterators.get(rit);
        return it.hasNext();
    }
    public boolean isBeanieToysesEmpty()
    {
        return BeanieToyses.isEmpty();
    }
    public void putBeanieToys(String key, examples.umlebsc.BeanieToy BeanieToys)
    {
        isDirty = true;
        BeanieToyses.put(key, BeanieToys);
    }
    public void putBeanieToyses(TreeMap BeanieToyses)
    {
        this.BeanieToyses.putAll(BeanieToyses);
        isDirty = true;
    }
    public void removeAllBeanieToyses()
    {
        isDirty = true;
        BeanieToyses.clear();
    }
    public void removeBeanieToysAt(RemoteIterator rit)
    {
        isDirty = true;
        Iterator it = (Iterator) BeanieToysIterators.get(rit);
        it.remove();
    }
    public examples.umlebsc.BeanieToy removeBeanieToysByKey(String key)
    {
        isDirty = true;
        return (examples.umlebsc.BeanieToy) BeanieToyses.remove(key);
    }
```

Notice the following:

1. Through the EntityImpl abstract class, HumanBeanImpl inherits two fields, `boolean isDirty` and `EntityContext ctx`. `isDirty` is set whenever an attribute changes. (There are no changeable attributes in the HumanBean.)

2. The implementation has an attribute variable for each attribute in the class.

3. Implementations required by Sun's EJB Specification:

   - `setEntityContext()` and `unsetEntityContext()` methods.
   - The methods `ejbCreate(), ejbPostCreate(), ejbLoad(), ejbStore(), ejbRemove(), ejbActivate(), ejbPassivate().`

Examine the file HumanBeanHome.java:

```
public interface HumanBeanHome extends SmartEJBHome
{
    public HumanBean create(examples.umlebsc.HumanBeanPk humanBeanPk) throws CreateExcep
RemoteException;
    HumanBean findByPrimaryKey(examples.umlebsc.HumanBeanPk humanBeanPk) throws
RemoteException, FinderException;
}
```

Notice the following:

1. HumanBeanHome is an interface as required by Sun's EJB Specification.

2. The methods `create(examples.umlebsc.HumanBeanPk humanBeanPk)` and `findByPrimaryKey(examples.umlebsc.HumanBeanPk humanBeanPk)` as required by Sun's EJB Specification. Note these methods use the generated `HumanBeanPk` class as the primary key.

Examine the file HumanBeanPk.java:

```
public class HumanBeanPk extends theory.smart.foundation.SmartKey implements
java.io.Serializable
{
    public String name;
    public HumanBeanPk()
    {
        super();
    }
    public HumanBeanPk(String name)
    {
        super();

        this.name = name;
    }
    public int compareTo(Object o)
    {
        if (o instanceof HumanBeanPk == false)
        {
            throw new ClassCastException();
        }
        int result = 0;
        HumanBeanPk target = (HumanBeanPk) o;
        result = this.name.compareTo(target.name);
        return result;
    }
    public String toString()
    {
        return "" + name;
    }
```

Notice the following:

1. By inheriting from `theory.smart.foundation.SmartKey`, HumanBeanPk implements the `java.io.Serializable` interface.
2. HumanBeanPk has a field for each attribute that comprises the primary key of HumanBean.
3. The implementation of the `java.util.Comparable` interface.
4. The `toString()` method returns the primary key.

Examine the file HumanBeanValue.java.

```
public class HumanBeanValue extends SmartValue
{
    public String name;
    public TreeMap BeanieToyses;

    protected HumanBeanValue()
    {
    }
}
```

Notice the following:

1. By inheriting from `theory.smart.foundation.SmartValue`, implements the `java.io.Serializable` interface.
2. Public fields for each attribute in HumanBean, including the BeanieToys aggregation.
3. The constructor is protected, helping to ensure the HumanBeanValue is only instantiated by a HumanBeanImpl.

Under the hood of the ShoppingTrip:

Examine the file ShoppingTrip.java:

```
public interface ShoppingTrip extends Session
{
  public boolean containsHumanBeansKey(String key) throws RemoteException;
  public boolean containsHumanBeansValue(examples.umlebsc.HumanBean HumanBeans) throws
RemoteException;
  public RemoteIterator createHumanBeansIterator() throws RemoteException;
  public examples.umlebsc.HumanBean getHumanBeansByKey(String key) throws RemoteExceptio
  public TreeMap getHumanBeanses() throws RemoteException;
  public examples.umlebsc.HumanBean getNextHumanBeans(RemoteIterator rit) throws
RemoteException;
  public int getNumberOfHumanBeanses() throws RemoteException;
  public boolean hasNextHumanBeans(RemoteIterator rit) throws RemoteException;
  public boolean isHumanBeansesEmpty() throws RemoteException;
  public void putHumanBeans(String key, examples.umlebsc.HumanBean HumanBeans) throws
RemoteException;
```

```
   public void putHumanBeanses(TreeMap HumanBeanses) throws RemoteException;
   public void removeAllHumanBeanses() throws RemoteException;
   public void removeHumanBeansAt(RemoteIterator rit) throws RemoteException;
   public examples.umlebsc.HumanBean removeHumanBeansByKey(String key) throws RemoteExcep

   public void findAFriend(String humanBeanName,  argname) throws RemoteException
   public void buyABeanieToy (String humanBeanName, String toyName) throws RemoteException
 }
```

Notice the following:

1. By inheriting from theory.smart.foundation.Session, the ShoppingTrip is serializable and implements `java.ejb.EJBObject`.

2. Since ShoppingTrip has no attributes except the HumanBeans aggregation, there are no get<attribute> and set<attribute> methods.

3. The RemoteIterator and additional methods for accessing the HumanBeans aggregation.

4. The service methods `findAFriend()` and `buyABeanieToy().`

5. Since ShoppingTrip is stereotyped as a `<<BSC.Session>>`, there is no support for a primary key.

Examine the file ShoppingTripImpl.java:

```
 public class ShoppingTripImpl extends SessionImpl
 {
     public TreeMap HumanBeanses;
     private TreeMap HumanBeansIterators = new TreeMap(SmartComparator.getInstance());

     public ShoppingTripImpl() throws CreateException
     {
         super();
     }
     public void ejbCreate() throws CreateException
     {
         super.ejbCreate();
         HumanBeanses = new TreeMap(SmartComparator.getInstance());
     }
     public void ejbPostCreate() throws CreateException
     {
         super.ejbPostCreate();
     }
     public void ejbActivate() throws java.rmi.RemoteException
     {
         super.ejbActivate();
     }

     public void ejbPassivate() throws java.rmi.RemoteException
     {
         super.ejbPassivate();
     }

     public void ejbRemove() throws java.rmi.RemoteException
     {
         super.ejbRemove();
     }

     public void setSessionContext(SessionContext ctx) throws java.rmi.RemoteException
     {
         super.setSessionContext(ctx);
     }
     public boolean containsHumanBeansKey(String key)
```

```
    {
        return HumanBeanses.containsKey(key);
    }
    public boolean containsHumanBeansValue(examples.umlebsc.HumanBean HumanBeans)
    {
        try
        {
            return HumanBeanses.containsValue(new SmartHandle(HumanBeans));
        }
        catch (Exception e)
        {
            return false;
        }
    }
    public RemoteIterator createHumanBeansIterator()
    {
        Iterator it = HumanBeanses.values().iterator();
        RemoteIterator rit = new RemoteIterator(it);
        HumanBeansIterators.put(rit, it);
        return rit;
    }
    public examples.umlebsc.HumanBean getHumanBeansByKey(String key) throws
java.rmi.RemoteException
    {
        SmartHandle sh = (SmartHandle) HumanBeanses.get(key);
        if (sh == null)
        {
            return null;
        }
        else
        {
            return (examples.umlebsc.HumanBean) sh.getHandle().getEJBObject();
        }
    }
    public TreeMap getHumanBeanses()
    {
        TreeMap map = new TreeMap(SmartComparator.getInstance());
        map.putAll(HumanBeanses);
        return map;
    }
    public examples.umlebsc.HumanBean getNextHumanBeans(RemoteIterator rit) throws
java.rmi.RemoteException
    {
        ListIterator it = (ListIterator) HumanBeansIterators.get(rit);
        SmartHandle sh = (SmartHandle) it.next();
        return (examples.umlebsc.HumanBean) sh.getHandle().getEJBObject();
    }
    public int getNumberOfHumanBeanses()
    {
        return HumanBeanses.size();
    }
    public boolean hasNextHumanBeans(RemoteIterator rit)
    {
        Iterator it = (Iterator) HumanBeansIterators.get(rit);
        return it.hasNext();
    }
    public boolean isHumanBeansesEmpty()
    {
        return HumanBeanses.isEmpty();
    }
    public void putHumanBeans(String key, examples.umlebsc.HumanBean HumanBeans) throws
java.rmi.RemoteException
    {
        isDirty = true;
        HumanBeanses.put(key, new SmartHandle(HumanBeans));
    }
    public void putHumanBeanses(TreeMap HumanBeanses)
    {
        this.HumanBeanses.putAll(HumanBeanses);
        isDirty = true;
    }
```

```
     public void removeAllHumanBeanses()
     {
         if (HumanBeanses.size() > 0)
         {
             isDirty = true;
             HumanBeanses.clear();
         }
     }
     public void removeHumanBeansAt(RemoteIterator rit)
     {
         isDirty = true;
         Iterator it = (Iterator) HumanBeansIterators.get(rit);
         it.remove();
     }
     public examples.umlebsc.HumanBean removeHumanBeansByKey(String key) throws
 java.rmi.RemoteException
     {
         isDirty = true;
         SmartHandle sh = (SmartHandle) HumanBeanses.remove(key);
         if (sh == null)
         {
             return null;
         }
         return (examples.umlebsc.HumanBean) sh.getHandle().getEJBObject();
     }
     public void findAFriend(String humanBeanName,  argname) throws java.rmi.RemoteExcept
     {
         return ;
     }
     public void buyABeanieToy(String humanBeanName, String toyName) throws
 java.rmi.RemoteException
     {
         return ;
     }
```

Notice the following:

1. Through the EntityImpl abstract class, ShoppingTripImpl inherits two fields, `boolean isDirty` and `SessionCtx ctx`. `isDirty` is set whenever an attribute changes. (There are no changeable attributes in the HumanBean.)

2. Implementations required by Sun's EJB Specification:

   - The methods ejbCreate(), ejbPostCreate(), ejbLoad(), ejbStore(), ejbRemove(), ejbActivate(), ejbPassivate().

3. The implementation of findAFriend() and buyABeanieToy() are the only user-required coding.

Examine the file ShoppingTripHome.java:

```
public interface ShoppingTripHome extends SmartEJBHome
{
    public ShoppingTrip create() throws CreateException, RemoteException;
}
```

Notice the following:

1. ShoppingTripHome is an interface as required by Sun's EJB Specification.

2. The method `create()` and `findByPrimaryKey(examples.umlebsc.HumanBeanPk humanBeanPk)` as required by Sun's EJB Specification. Note there is no support for primary keys.