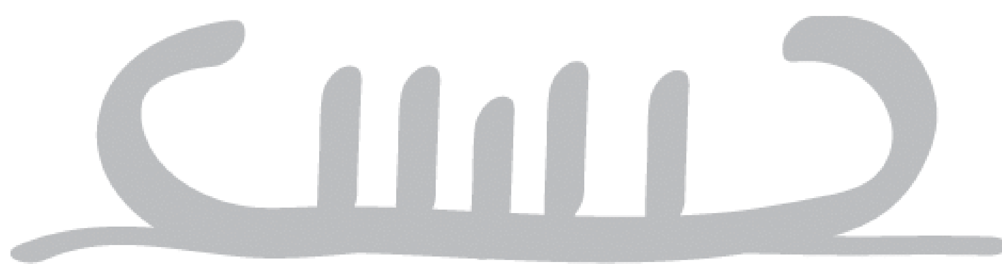


AGETOR®

Application Developer’s Guide

Contents

Preface 3
The entity component model 4
    Requirements 4
Overview of the entity component model 5
    The entity and its business interface 5
    Entity home 6
    Proxy 7
    Container 7
    Component system interfaces 7
    Entity 8
    Context 8
    EntityHome 8
Building an application 10
    Identifying datacomponents and entity components 10
  Datacomponents 11
    IDL-generated datacomponent 11
    Service wrapper layer 11
    Business logic layer 11
  Entities 12
    Defining the business interfaces 12
    Implementing the business interfaces 12
    Creating the home classes 13
    Guidelines 14

Preface

This guide tells you how to build business applications for the AGETOR Application Server. It assumes that you are familiar with Java and general object-oriented design principles.

The entity component model

The AGETOR Application Server has a component architecture that aims at creating a suitable environment for entity objects. An entity is an object that models persistent data in the domain model, whereas datacomponents are used for controlling sessions and for stateless controller functions. The entity component corresponds to the EJB (Enterprise JavaBeans) model's EntityBean.

The main features of the entity component model are:

- A caching mechanism that is fully transparent for clients
- Separation of the business logic from technical details like the actual storage mechanism
- Possibility of optimizing retrieval of data from backend systems
- Life cycle management of objects that isn't intertwined with the business logic

Requirements

- ADK 2.0.0

Overview of the entity component model

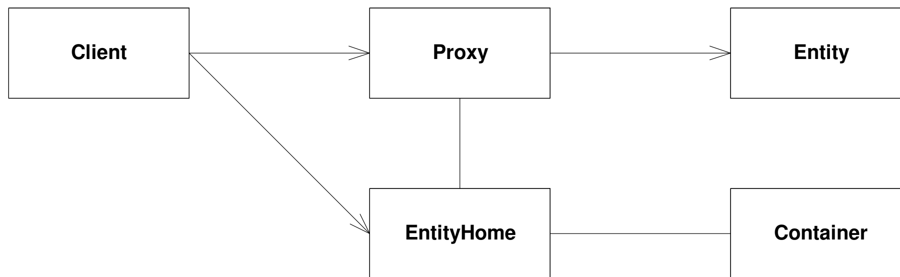
The entity component model has four main elements:

- Entity
- Entity home
- Proxy
- Container

Entities are the primary objects of the system. An entity consists of a business interface and an entity class. The entity class implements the business logic defined in the business interface but it contains no infrastructural code related to storage etc.

An entity is obtained through its home class which functions as a factory for entities. Each home class is, in turn, related to a container that assists the home class in managing the life cycle of the entities.

To make the life cycle handling of an entity transparent to the client, the client does not obtain a reference to the entity itself, instead it "talks" with the entity through a proxy object. To the client it will seem as if the entity is always present, but in fact the entity may not have been loaded into memory before the client called one of the business methods of the entity.



The client only interacts with the EntityHome and the proxy (although the client perceives the proxy as being the true entity).

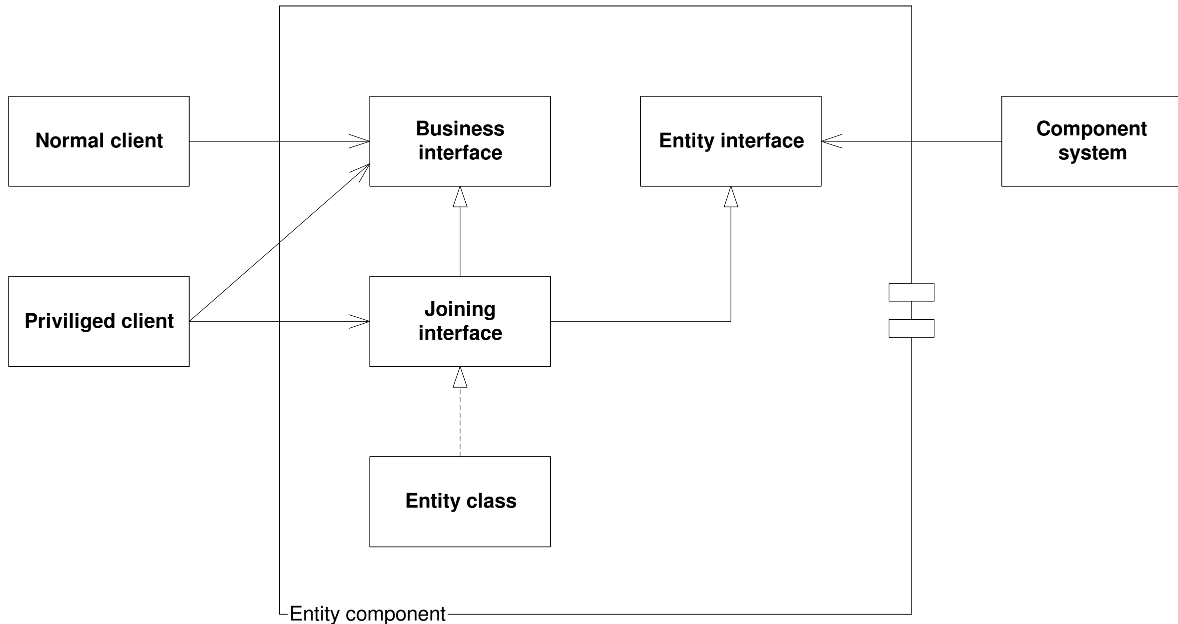
The entity and its business interface

An entity represents some object in the domain model. Each entity implements one (or more) business interface and the entity is accessed through this interface, both by other entities and by clients such as datacomponents and IML/JSP documents.

The interfaces and implementation classes may belong to different packages in order to accentuate the division between interface and implementation.

To support the component system, all entities must implement the *Entity* interface. At the same time, the proxy can only support a single interface, so the solution is to create a joining interface, that extends the business interface(s) and the *Entity* interface. Another solution is to have the business interface extend the *Entity* interface.

Should the implementation of an entity require some additional methods that need not (or should not) be present in the business interface (such as methods needed by the entity's home class for storage-related reasons), they can be defined in the joining interface, which is placed in the same package as the implementing class.



The normal clients (such as classes implementing business interfaces) only use the business interface, whereas the privileged client (e.g., a home class or a datacomponent) accesses the joining interface.

Entities can be shared by multiple users and should not contain user-specific state (session handling is the job of datacomponents). Of course, a customer entity or a persistent cart should not be shared, but as they survive sessions and are persistent by nature they are modelled as entities.

Entity home

The home classes have several important roles:

- Factories for entities
- Control the life cycle of the entities
- Adapters to the actual back-end system
- Facilitate performance optimizations

An entity home is a factory for accessing and creating entities and each entity type has an entity home. A client never instantiates an entity directly using the `new` operator, but gets a reference to an entity by calling the `getEntity()` method on the appropriate home class.

The entity homes are responsible for loading and storing the entity. The entity home encapsulates the details of getting the information for loading the entity. This could be data from a database or objects retrieved by making an IDL-method call.

Separating the business logic implemented by the entity classes from the storage mechanism makes it possible to adapt an application for different storage and communication models by modifying the home classes only. The SpeedShop entities, for instance, are loaded by making IDL calls to the SpeedShop server but the entity homes could get the information needed for loading the entities from some other source without affecting the implementation of the business logic.

The details of the life cycle management are transparent for the entity itself, as they are handled by the home classes and the container.

Proxy

The entity component model uses a proxy model to achieve a transparent caching mechanism. When a client requests an entity from its home class (see below), it doesn't get a reference to the entity itself. Instead, it gets a reference to a proxy object that implements the business interface of the entity.

The proxy acts a thin shell that makes it transparent to the client whether the entity actually has been loaded into memory or not. When a method requiring the entity to be present is invoked on the proxy, the entity is automatically loaded and the method is executed.

Container

By inserting a proxy between the client and the entity and thus only requiring the presence of the entity when it is needed, it is possible to minimize the memory requirements of the application server as the entities can be removed when they are not needed. At the same time, the container holds on to the loaded objects so that they are cached and ready for use if a client requests them again.

The entities all belong to a container – one for each entity type – that keeps track of the usage of the entities and removes old and unused entities according to certain settings.

One of the features of this caching mechanism is that it is partly memory-sensitive. When an entity is no longer entitled to be in its loaded state, it is not removed immediately but passivated, and it is up to the virtual machine to decide when it should be removed. How this decision is made depends upon the concrete virtual machine implementation, but the memory consumption and available memory are likely to be some of the deciding factors.

The cache behaviour of the container is controlled by four parameters:

- maxSize: the maximum number of entities that are guaranteed to be loaded. A maxSize of 0 means that there is *no* limit on the number of loaded entities.
- minIdleTime: the minimum period of time an entity must have been idle before it can be passivated. This ensures that entities are not passivated shortly after their insertion if the maxSize has been set too low, i.e., minIdleTime will overrule maxSize. This feature can disabled by setting minIdleTime to 0.
- maxIdletime: the maximum period of time that an entity can be idle before it is passivated. A value of 0 indicates that there is *no* limit to an entity's idle time.
- maxAge: the maximum period of time an entity can be used before being reloaded. This guarantees that no entity in use is older than maxAge. If maxAge is set to 0 the entity is not reloaded unless it is removed first. The maxAge is useful in cases where the entities are modified in the backend system that stores it. Updated versions can then be fetched automatically at a regular interval.

Component system interfaces

In the following, the most important interfaces of the component system are described. These interfaces are part of the infrastructure that maintains the run-time environment of the entities.

Entity

This interface must be implemented by all entities in the system. An entity is an independent object whose life cycle is not determined by other objects, e.g., a shopping cart is an entity whereas the cart lines (items) are not as they only make sense in the context of a cart.

The entity interface has two methods:

- getIdentity
  The identity must be an object that is unique within the entity type, e.g., all items have different identities.

Context

The context is a marker interface that works as an invocation context of the methods defined in the business interfaces. To ensure that the methods are invoked in the right context (e.g., regarding user identity and security information), each method must have a Context as the first parameter.

This interface should be implemented by a class, which contains, or has access to, properties such as user and security information. Within the AGETOR framework this could be a locally scoped datacomponent.

A subinterface called *AnonymousContext* is used to indicate that a method is invoked within an unspecified context.

EntityHome

The home class of an entity manages its life cycle together with its container. It has the following methods:

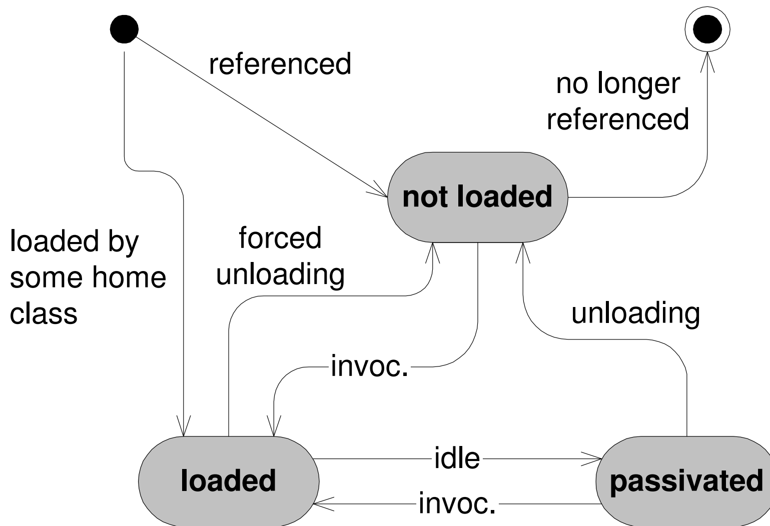
- `getEntity(Object identity)`
  Gets a proxy for the entity with the given identity
- `setEntity(Entity entity)`
  Uses the given entity to create its corresponding proxy, and returns the proxy.
- loadEntity (Context ctx, Object identity)
  This method is called when the entity must be present to handle method invocations.
- storeEntity (Context ctx, Entity entity)
  Stores the entity. This method can also be called explicitly, but it is also called immediately before an entity is passivated. In the latter case the context will be an instance of AnonymousContext, and it is up to the specific implementation to handle this.
- `afterInvocation(Context ctx, Entity entity, Method m, Object[] args, Object result)`
  Enables the home class to intercept method calls before they reach the entity.
- `beforeInvocation(Context ctx, Entity entity, Method m, Object[] args, Object result)`
  Enables the home class to do some processing after an entity method has been invoked.
- `getContainer()`
  Gets the container holding the entities.

Life cycle

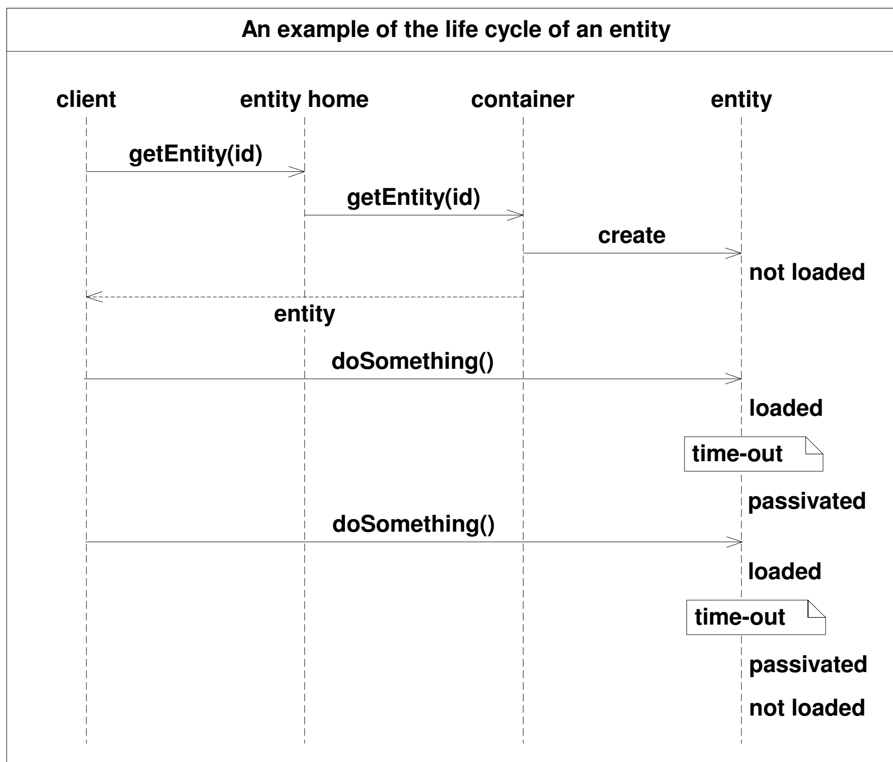
The states:

- Loaded: Business methods can be invoked on the entity and the entity can only be unloaded forcibly (e.g., by clearing the container)

- Not loaded: The entity can be referenced but invoking a method will force it to be loaded before execution of the method
- Passivated: The entity is eligible to be unloaded. Without method invocation activity, it will be unloaded within an undefined period of time.
  The entity will be stored automatically before it is passivated
- This life cycle is controlled by the component system



Example:



Building an application

The process of building an application involves defining a business model and deciding what type of components to use for implementing the objects of the model.

This section will show you how to build a new application, focusing on entity components and datacomponents. Creating the user interface and any servers that may be needed, is not covered here. The steps involved in this process include the following:

- Identifying datacomponents, entity components and ordinary objects
- Implementing the datacomponents
- Defining the business interfaces of the entities
- Implementing the business interfaces
- Implementing the home classes

It is assumed that you have already established the elements of the domain model, and that you have defined their internal relationships. Moreover, building the application should be considered an iterative process, and the steps may have to be carried through in another order in some cases.

Identifying datacomponents and entity components

The AGETOR Application Server offers two component types: datacomponents and entity components. The entity components are suitable for representing persistent data, while datacomponents can be used for controlling user sessions and performing session-independent tasks such as generating data for use in global views (menus, news lists etc.). This corresponds to the object types found in most object-oriented methodologies (often called controller objects and entity objects), so it should be possible to identify datacomponents and entities in a domain model.

In general, objects whose life time span is a session should be implemented using a datacomponent. Likewise, control objects are often user-specific in some way and this also points at using datacomponents. Objects that are persistent are normally best implemented as entities.

In some cases it may be difficult to discern between the two types. For instance, a shopping cart may be implemented as a datacomponent or an entity. If it is not stored persistently, it has a controller-like function (keeping track of the items) and should be implemented as a datacomponent. On the other hand, a persistent cart should be implemented as an entity, although it is still user-specific. The point lies in distinguishing between “user-specific” and “session-specific”. A non-persisted cart is user-specific *and* session-specific, while a persistent cart is user-specific but also exists across sessions, so the deciding factor is whether an object survives a session, or not.

Besides that, there will be some additional objects that do not belong to any of the other categories. Typical examples are objects whose life cycle is fully dependent on other objects such as the lines of an order object, or the address of an employee. If objects of this kind belong to a persistent entity, the information for creating them should be loaded by this entity’s home class.

Datacomponents

There are several strategies for implementing the functionality of datacomponents. The strategy below focuses on separating the different responsibilities of a datacomponent and placing them in different classes in an inheritance hierarchy.

IDL-generated datacomponent

The idl2java tool creates a default datacomponent, which has three characteristics:

- Easy access to IDL methods
- A set of public data members corresponding to the method parameters of the methods defined in the IDL
- Automatic extraction of data from servlet parameters to the public data members

By providing this functionality, some of the trivial work involved in building a web application is alleviated.

Service wrapper layer

In most applications it will be necessary to wrap the basic IDL calls in order to control the state of the datacomponent and introduce aspects such as logging. By encapsulating service and basic state control in a separate class, the business logic and lower level aspects of a datacomponent are kept apart.

Examples of the functionality of the service wrapper layer:

- Logging of service method calls
- Method for getting and setting objects that are maintained using the datacomponent

Business logic layer

On top of the basic functionality, the business logic must be added:

- Session handling (login etc.)
- Event handling

The event handling will most often be concerned with maintaining the objects that the datacomponent controls access to, i.e., creating, updating, and deleting these objects.

Entities

There are three main steps involved in building the entity components. The first step is to create the logical model by defining the business interfaces in a separate package. In the second step, the entities are implemented and additional methods are added to the interface that joins the business interface and the Entity interface. The last step is to create the home classes of the entities.

Defining the business interfaces

The business interfaces should be defined in a separate package and none of the interfaces should refer to the implementations of the other interfaces. Likewise, the interfaces should not refer to any other implementation-specific classes or interfaces such as those representing a particular back-end system or storage mechanism. This will result in a purely logical object model of the business domain.

Example:

```
public interface ItemList {
   …
}

public interface Item {
   ItemList getPrimaryItemList(Context ctx);
   …
}
```

Implementing the business interfaces

We recommend implementing the business interfaces in a separate package to maintain a clear separation of interfaces and implementing classes.

To tie the business model to the entity model a helper interface is created. This helper interface joins the logical model to the entity component model by extending the business interface(s) and the Entity interface and, furthermore, it leaves room for implementation-specific methods the are unwanted in the business interface. An example from the SpeedShop application:

Business interface

```
public interface Cart {
   // Business methods
   …
}
```

SpeedShop-specific interface

```
public interface SpeedCart extends Cart, Entity {
   // Methods related to the SpeedShop-specific user interface
   CartView getSmallCartView(Context ctx);
   …
   // Life cycle related methods
   boolean isDirty(Context ctx);
   …
}
```

Implementing class

```
class SpeedCartImpl implements SpeedCart {
   // Implementation of Cart interface (general business methods)
   …
   // Implementation of methods present only in SpeedCart interface
   …
   // Implementation of Entity interface
   …
}
```

The entity classes must

- implement a joining subinterface of the business interface, that also extends the Entity interface
- always get or create another entity using that entity's home class
- never refer to the other classes in the package, instead, they should refer to the interfaces that these classes implement

Creating the home classes

Each entity must have a home class that is capable of loading and storing the entity. In most cases, you will probably be able to identify some common functionality that can be used in all your home classes. Typically, this will include:

- Methods for getting and setting the entities
- Resolution of a service based on the invocation context
- Empty implementations of some of the methods in the EntityHome interface (e.g., beforeInvocation and afterInvocation) for convenience
- Logging

When this has been implemented in a common base class, the concrete home classes are ready to be implemented. The `loadEntity()` method (and other methods that access the back-end services) requires some way of transforming the information from the back-end service to the actual entity implementation class.

It is recommended that this transformation is placed in a separate method, as this will isolate changes in the back-end service as much as possible, and make it feasible to adapt the home class to another back-end service. The SpeedShop application uses IDL method invocation to obtain data for creating the entities, and the transformations are placed in methods called `fromIDL()` and `toIDL()`.

The home class can be used to optimize the communication with the back-end services by fetching more data than it needs for creating the single entity that it was asked to load. By getting data for related entities in the same call, the home class can reduce the number of back-end calls and, furthermore, load entities that are expected to be used in context with the entity that was requested originally.

For instance, when the SpeedShop application fetches an ItemList all the Items in the list are fetched at the same time and handed over to their home class using the `putEntity()` method (which calls the `setEntity()` method). When an ItemList has been requested, it is very likely that the Items of the list will be requested soon after and, therefore, fetching these Items at the same time can constitute a significant performance improvement.

Static and dynamic proxies

The application server can use two different types of proxies: dynamic or static. The dynamic proxies are based on the built-in proxy model present in Java since JDK 1.3. These proxies are dynamic as they can implement an interface that is passed to them at run-time. This is only possible because the proxies are treated by the JVM in a special way. The dynamic proxies use reflection to invoke methods, which means that the performance somewhat lower than that of ordinary objects.

It is also possible to use static proxies instead of Java's reflection-based proxies (since ADK 1.3.9). The static proxies are generated specifically for each entity interface so there is no need for reflection, which improves performance significantly. The static proxies can be generated at run-time, which is convenient during development, or they can be pre-compiled using the *proxygen* tool.

Exception handling

The run-time behaviour of the two types of proxies is almost identical. Due to the nature of the dynamic proxies there is, however, one small difference.

For every method, the dynamic proxy is able to throw a *checked* exception that has *not* been declared by the method. This rather surprising behaviour cannot be imitated by a static proxy in any way, so if an undeclared checked exception occurs during invocation of a method (e.g., when beforeInvocation() is invoked on the entity home), the static proxy will wrap the checked exception in a RuntimeEntityException and throw this one instead.

Performance

Using reflection for invoking methods is more expensive than normal method invocation in terms of performance. Because a static proxy is generated to match a specific entity interface, it does not need to use dynamic invocation. There is a rather big performance penalty for dynamic invocation due to wrapping and unwrapping of parameters and return values, checking access rights, and checking that the parameters have the correct type.

Dynamic invocation performance is typically about 20-100 slower than normal invocation. However, the total performance of an application is affected by many other factors such as object creation, IO operations, and garbage collection so the overall performance of a web application that uses dynamic invocation extensively is more likely about 2-3 times lower than a corresponding reflection-free application. The responsiveness of the application is also dependent on network latency, and back-end systems but, other things being equal, the reflection-free application will require less CPU cycles and give faster responses.

Configuration

To use static proxies, you must set this property to true:

```
agetor.entity.proxy.static
```

Guidelines

To ensure that the container works properly and efficiently, the following guidelines should be observed:

- *Always* use the `getEntity()` to obtain an entity.
  If entities are accessed in any other way, such as instantiating an entity implementation directly or invoking the `loadEntity()` method, the entity container will not work properly
- *Never* invoke the `loadEntity()` method
  This method is called automatically by the container and no other object should invoke it.
- Keep in mind that the purpose of the `loadEntity()` method is to *construct an instance of the entity implementation*
  The `loadEntity()` method must obtain the necessary information, construct an instance, and return this instance. The information can come from various sources such as IDL method calls, JBDC, and other home classes.
- Use the `setEntity()` method to implement *performance optimizations*
  When loading a requested entity, you should consider if there are some related entities that are likely to be requested afterwards. If this is so, load the information needed for these entities too, construct the entities, and hand them over to their home class(es) using the `setEntity()` method.
- Use static proxies to increase performance.