Design choices in the business tier

To develop an enterprise system using an MDA approach to model-driven development, following steps can be carried out:

- Define PIM data model and use cases.
- Derive PSM data model (eg, for relational database implementation) by applying model transformations to PIM.
- Derive architecture and implementation of system, using PIM constraints to derive operation and transaction code.

Together, these steps ensure that system satisfies its specification and is correct by construction: specification properties remain true (possibly in a rewritten form) after model transformations, and code generation step produces code designed to maintain these properties at all observable time points in execution of system.

Constraints which are class invariants define data validity checks, carried out by entity bean derived from the class: checks determine if invariants hold for particular data (eg, data received from an HTML form used to create a new instance of class). Can also be used to define transaction which modifies dependent attributes when an attribute changes value.

Constraints linking data of two different classes can be used to define transactions involving operations of entity beans of both classes. Any change to data of one entity may require change to data of other, in order to maintain constraint. Updates should take place within uninterruptible transaction, so constraint is true at all observable time points of system.

Constraints also influence architectural decisions: if a constraint links two classes, would normally implement use cases for both classes in same session bean.

Examples of EIS design

Four example applications:

- Example of a stateless session bean to calculate the maximum mortgage loan for someone, based on their monthly income and the term (duration) of the loan.
- Pet insurance system.
- BMP entity bean example – an estate agent system.
- CMP entity bean and statefull session bean example of a bank account.

Mortgage Calculator

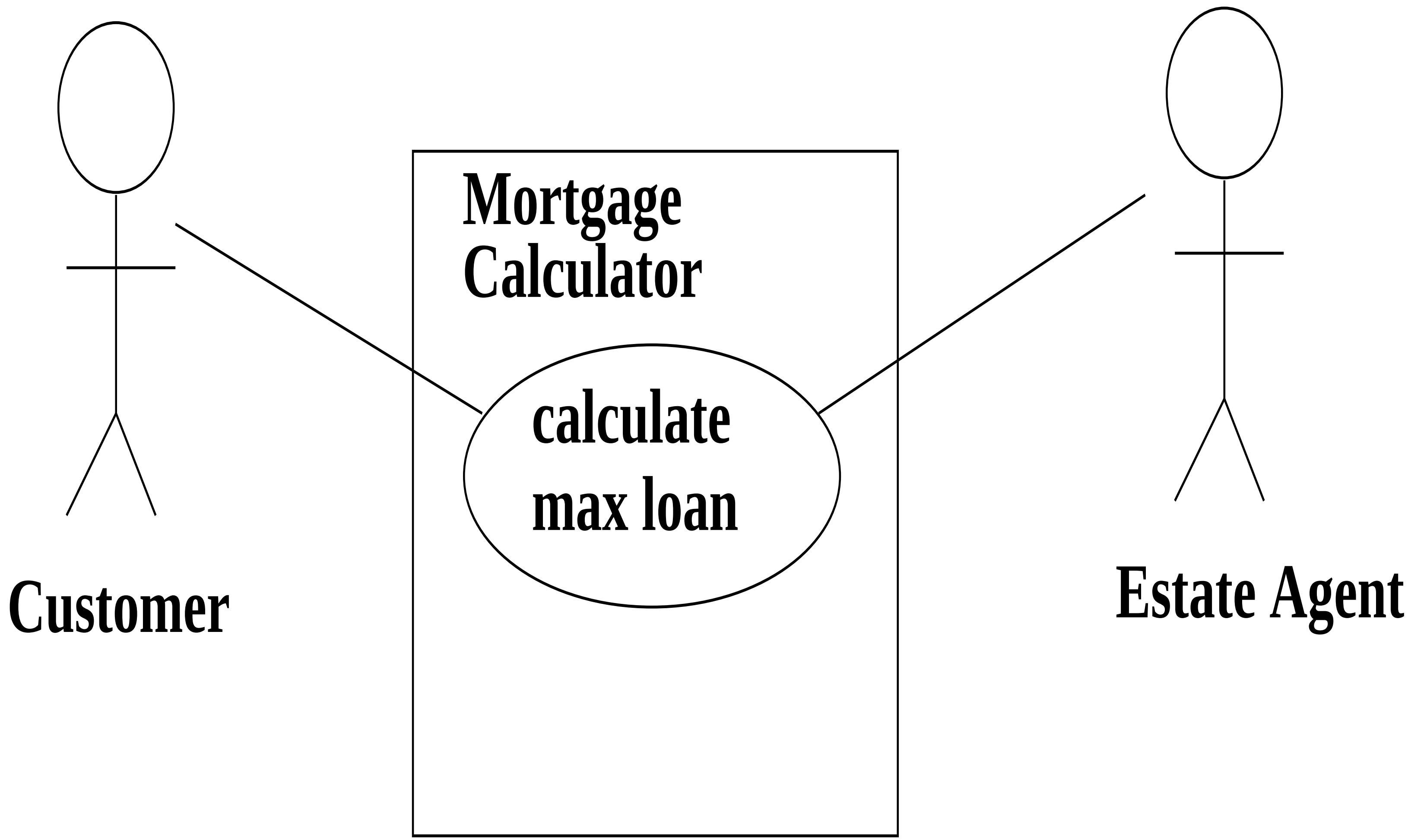
The aim of this system is to provide guidance to someone on what loan they could obtain, based on current rate of interest, length of loan, and monthly income (after tax). Could be used as part of a property search system, and possibly internally by an estate agent.

We have stereotyped dependencies from presentation tier to business tier as *remote*, and loan calc component as a stateless session bean. Web interface component will be a *view*, such as a PHP, JSP, ASP or other component designed to generate web pages.

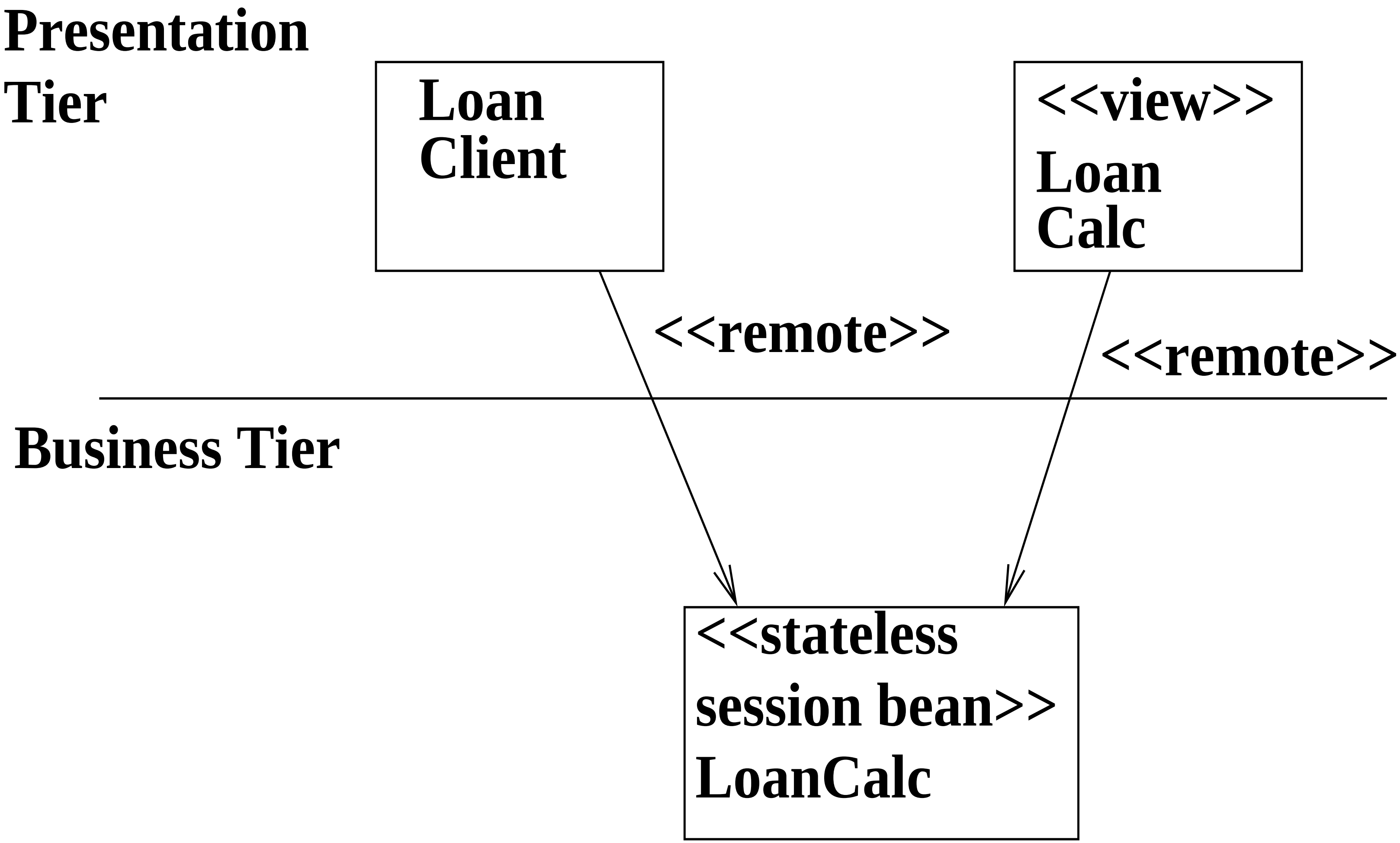
LoanCalc

```
maxLoan(rate: Integer,
        years: Integer,
        sal: Integer): Integer
pre: rate >= 0 &
    years >= 0 & sal >= 0
post:
  result = (400*years*sal)/
           (rate*years + 100)
```

PIM class diagram of loan calculator



Use case diagram of loan calculator



PIM design of loan calculator

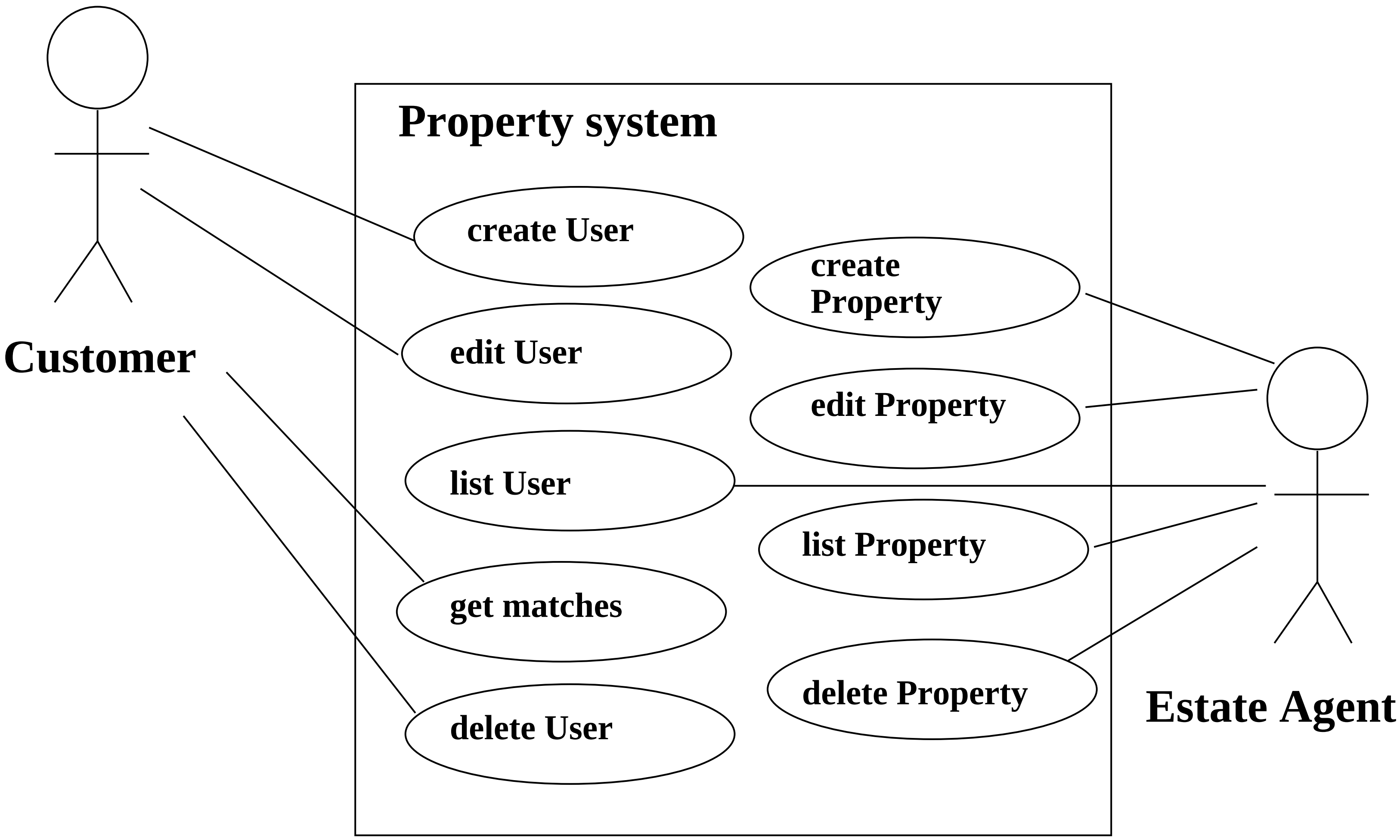
Property System

This is a simple online property search system for an estate agent: users may register their requirements for a property, and then carry out searches for properties that match these requirements.

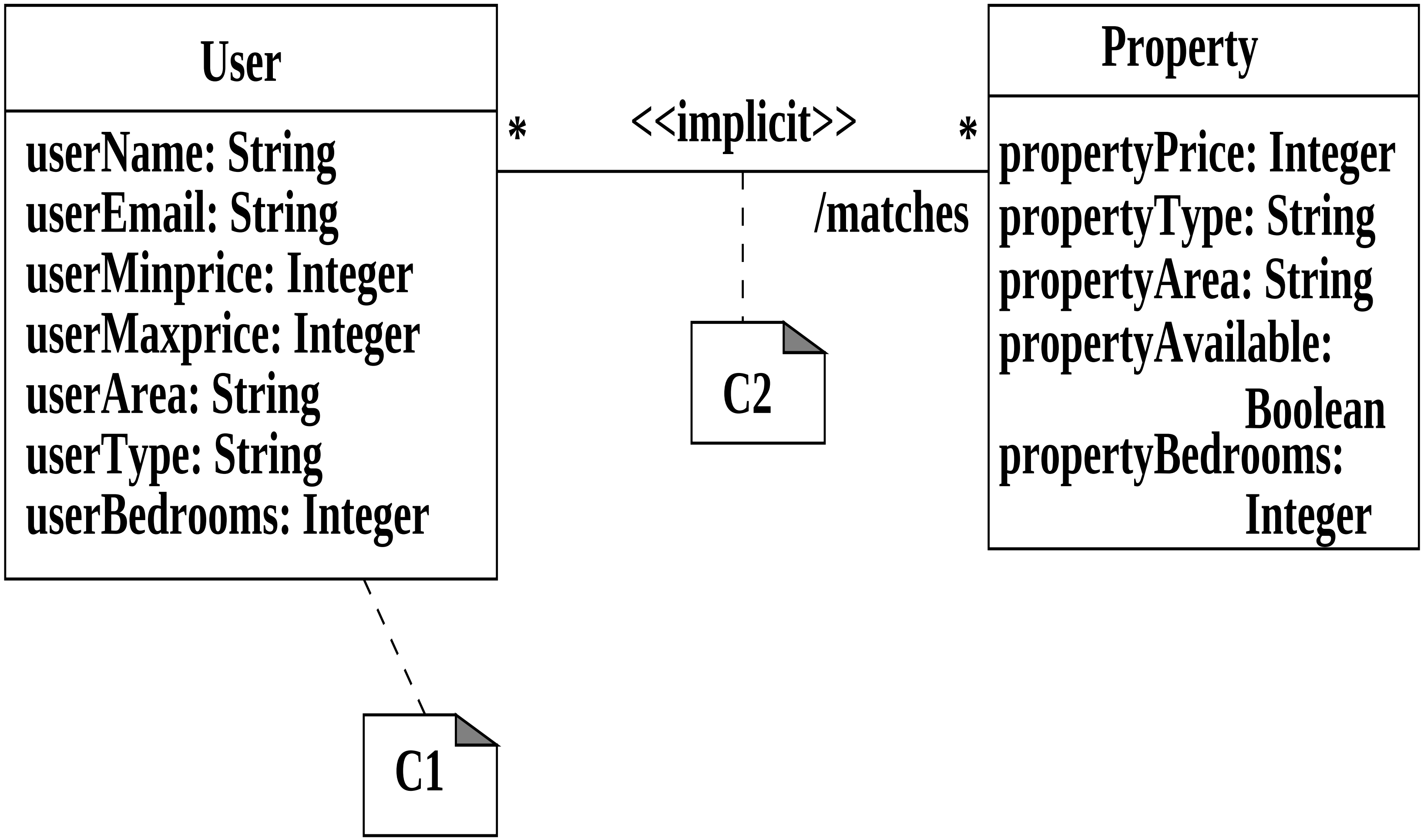
In property PIM $C1$ is an example of a class invariant constraint, of $User$:

$$userName.size > 0 \;\&$$
$$userMinprice \leq userMaxprice$$

This can be used to define data validation checks in the entity bean of $User$: for checking data which is input to system for creation of new instances of the class, or for modification of instances of the class.



Use cases of property system

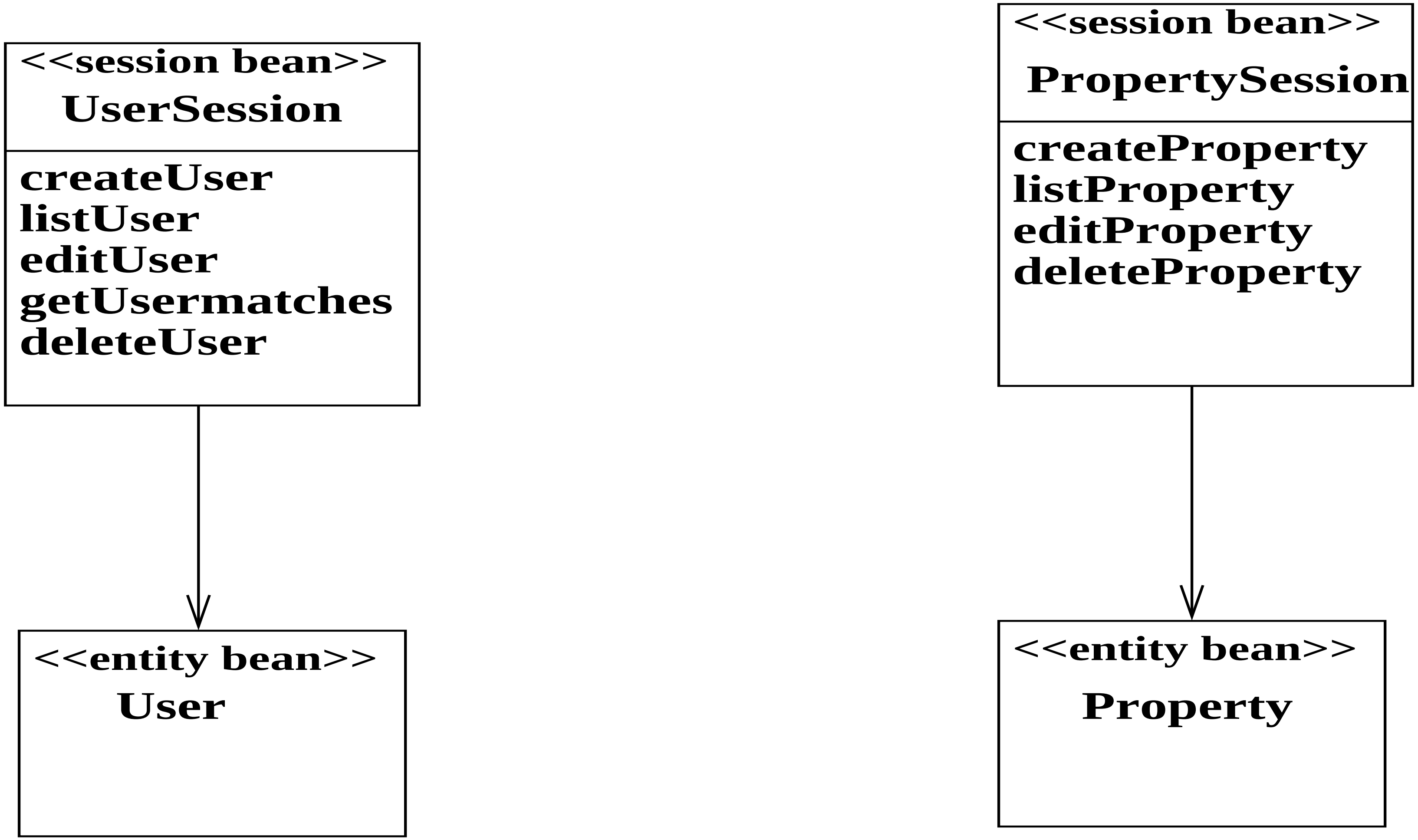


PIM of property system

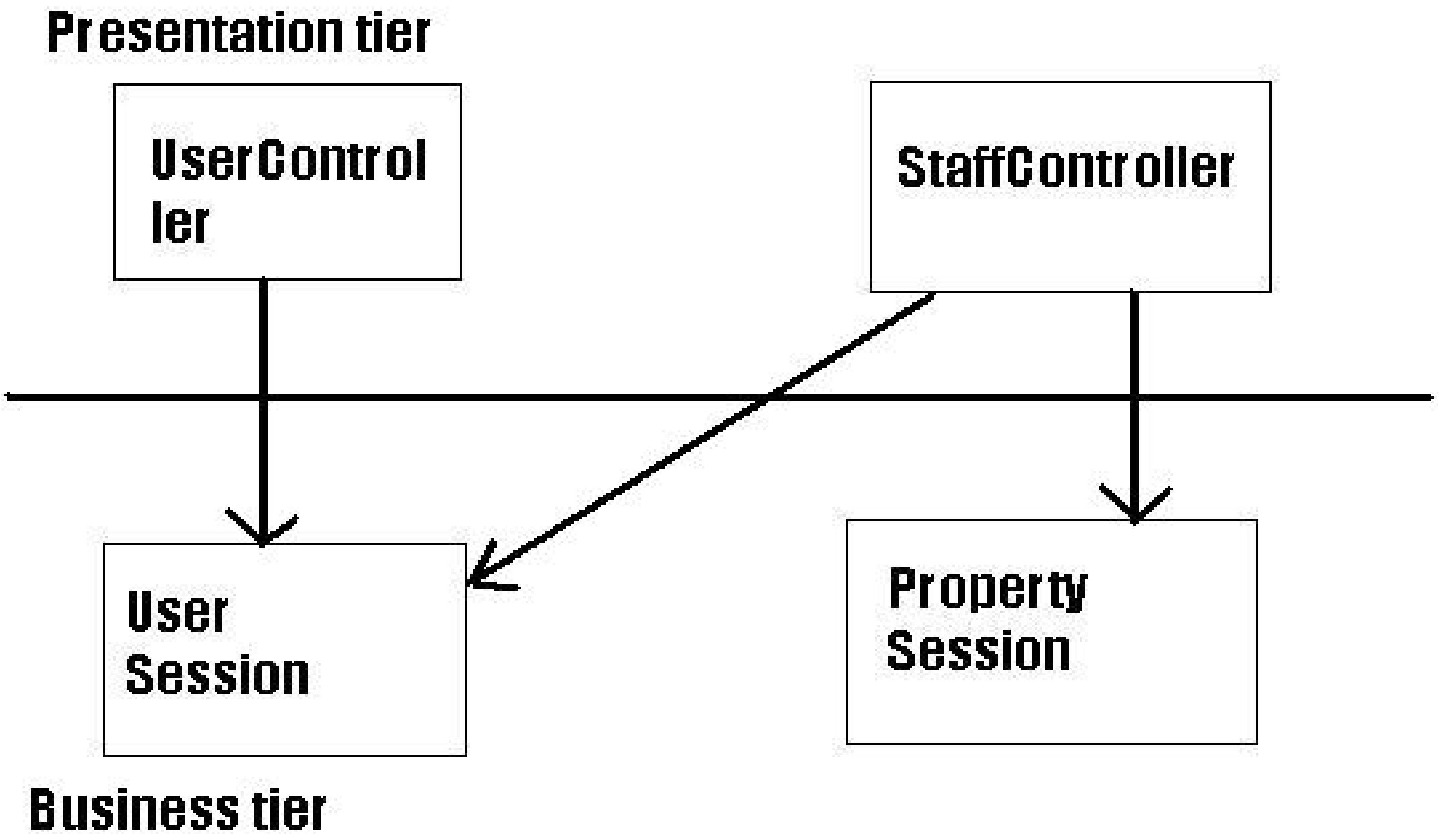
In contrast, constraint $C2$ attached to association defines set of elements that are linked by the association:

$$\begin{array}{l} userArea = propertyArea \ \& \\ propertyPrice \leq userMaxprice \ \& \\ userMinprice \leq propertyPrice \ \& \\ userBedrooms \leq propertyBedrooms \ \& \\ userType = propertyType \ \& \\ propertyAvailable = true \end{array}$$

This is used to derive the code of *getUsermatches* operation to find all properties which meet a particular user's requirements.



Business tier architecture of property system (1st version)

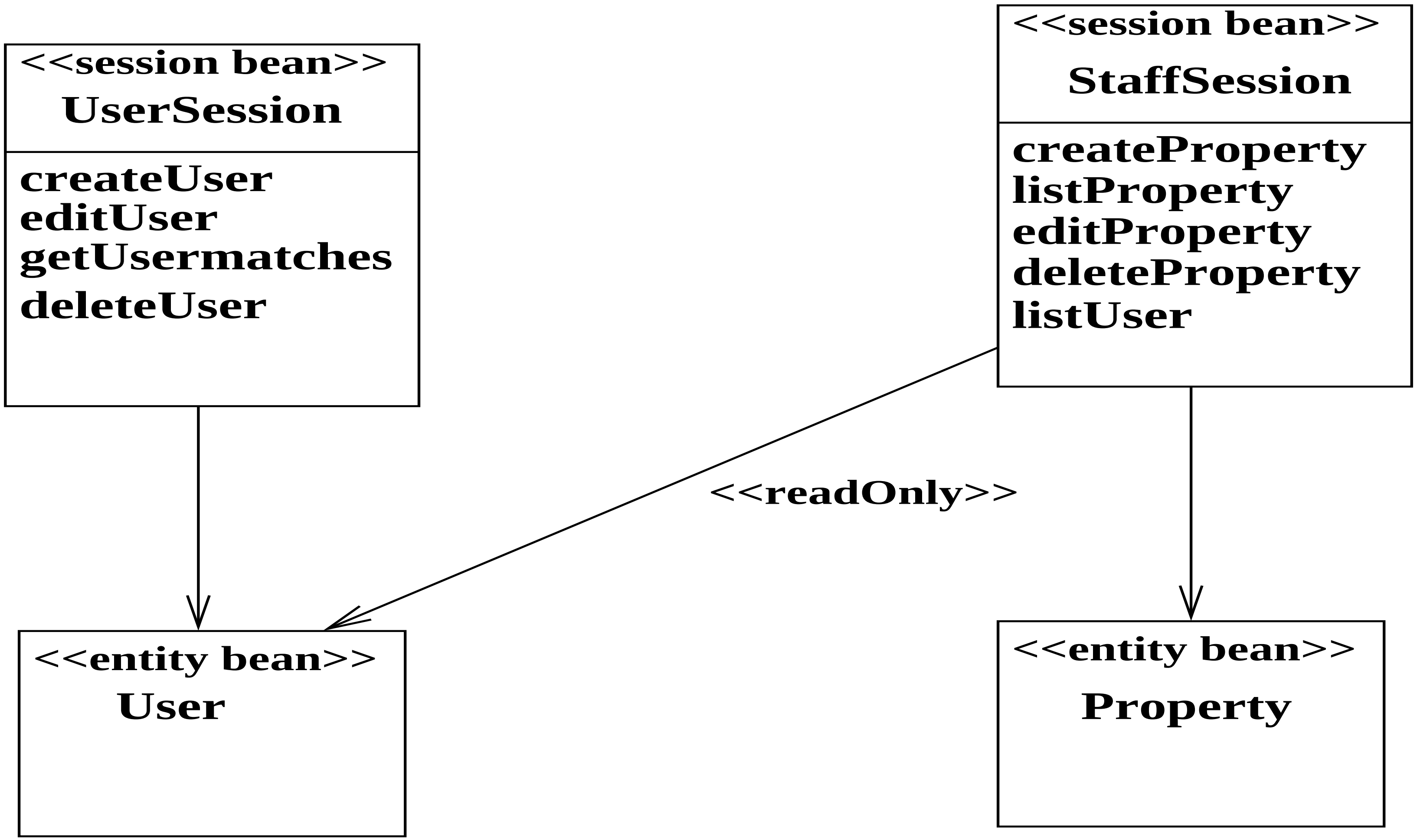


Presentation tier/Business tier architecture (1st version)

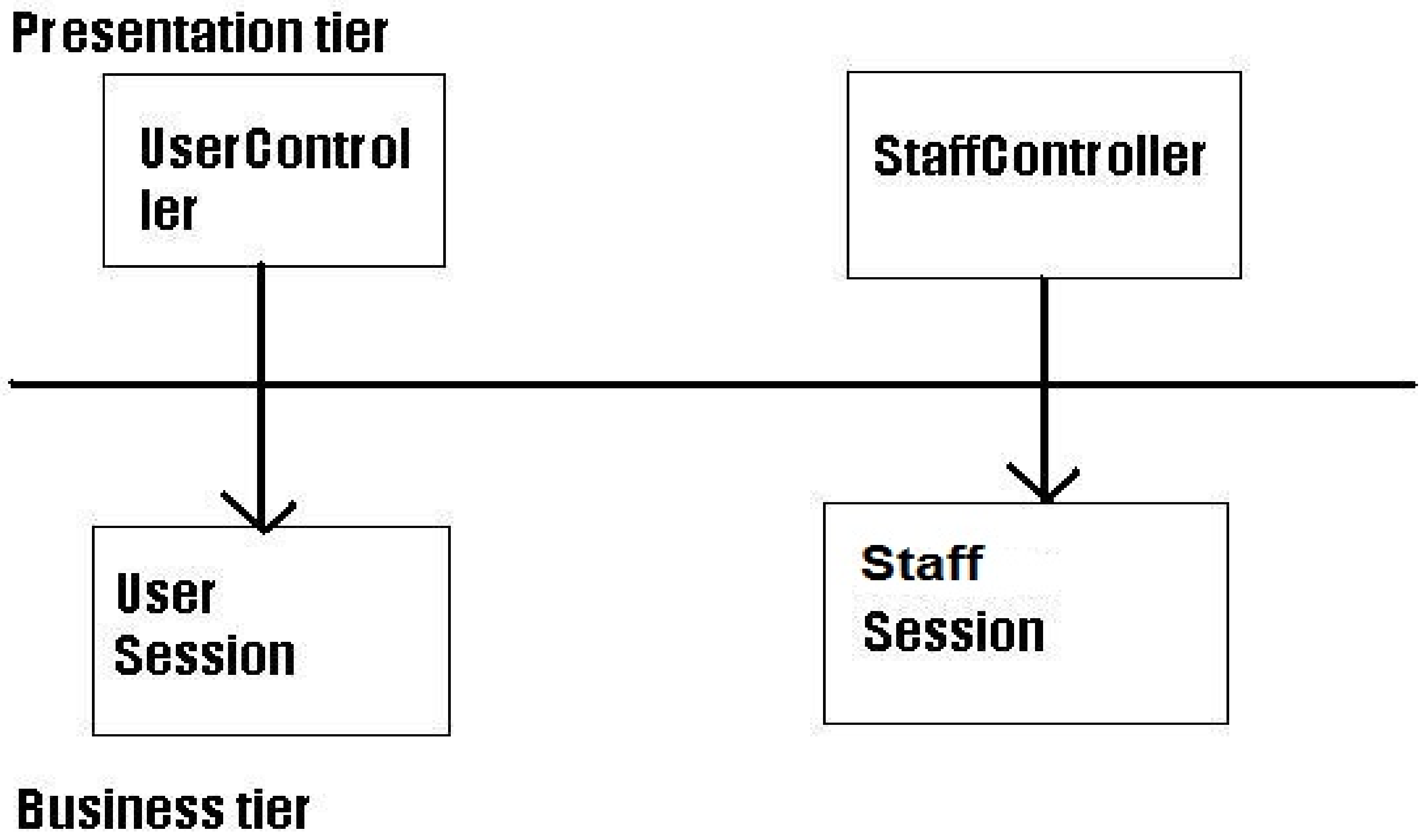
Separate session beans can be used, since none of use cases operating on *User* involve updating *Property*, and use cases on *Property* do not involve *User*. There are no constraints connecting the two classes.

In this case we have grouped use cases into beans on basis of what entity they operate on. An alternative would be to group them according to the actor of the use case: this would place *listUser* in the *StaffSession* session bean, and require read-only access from this to the *User* entity bean.

This has benefit that a single session bean can be used by each interface (actor) of the system, although it slightly increases number of dependencies in business tier. Also, users should not have access to *listUser*.



Business tier architecture 2 of property system



Presentation tier/Business tier architecture (2nd version)

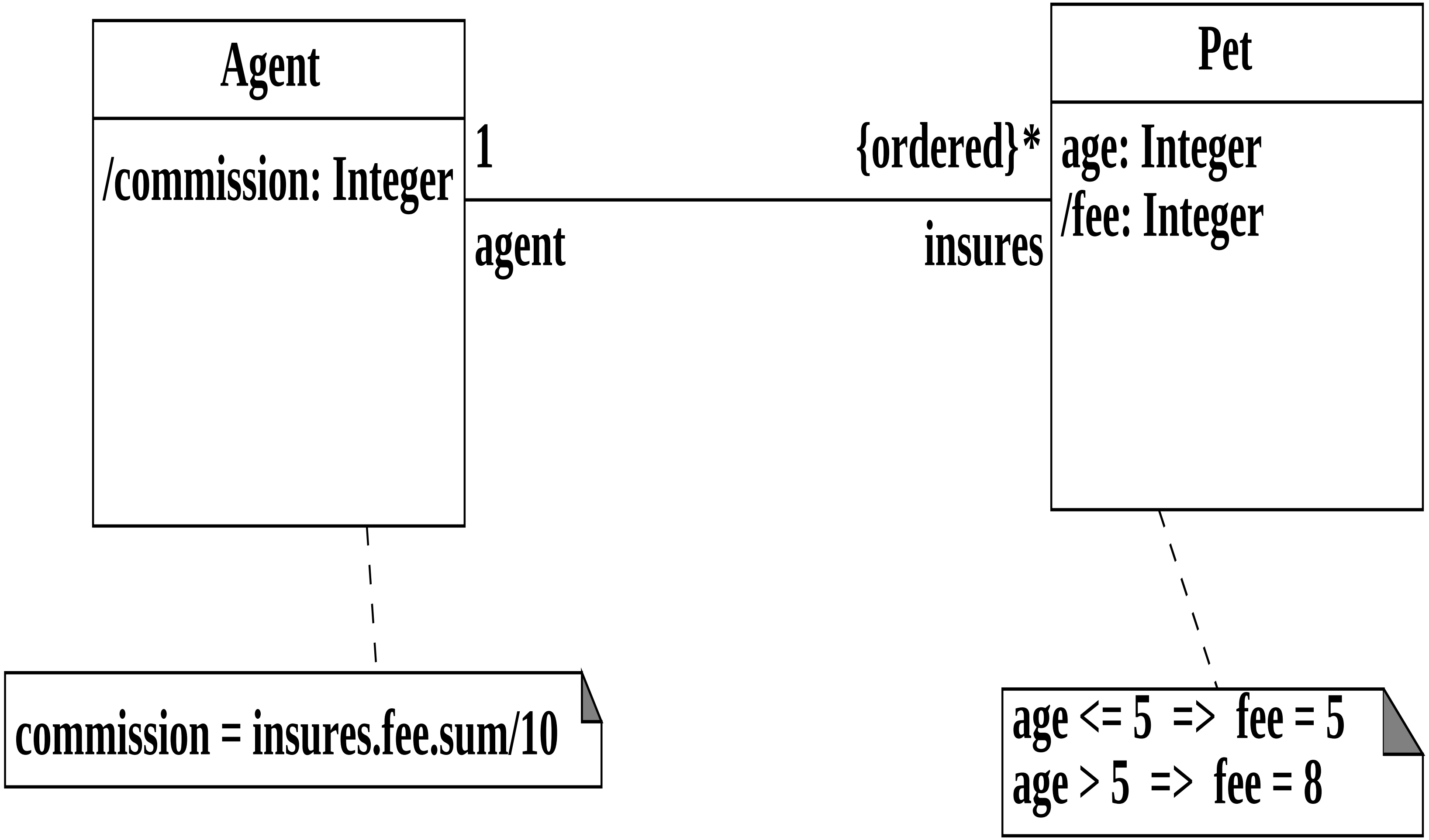
Pet Insurance System

This example illustrates how constraints linking classes are treated. The constraint

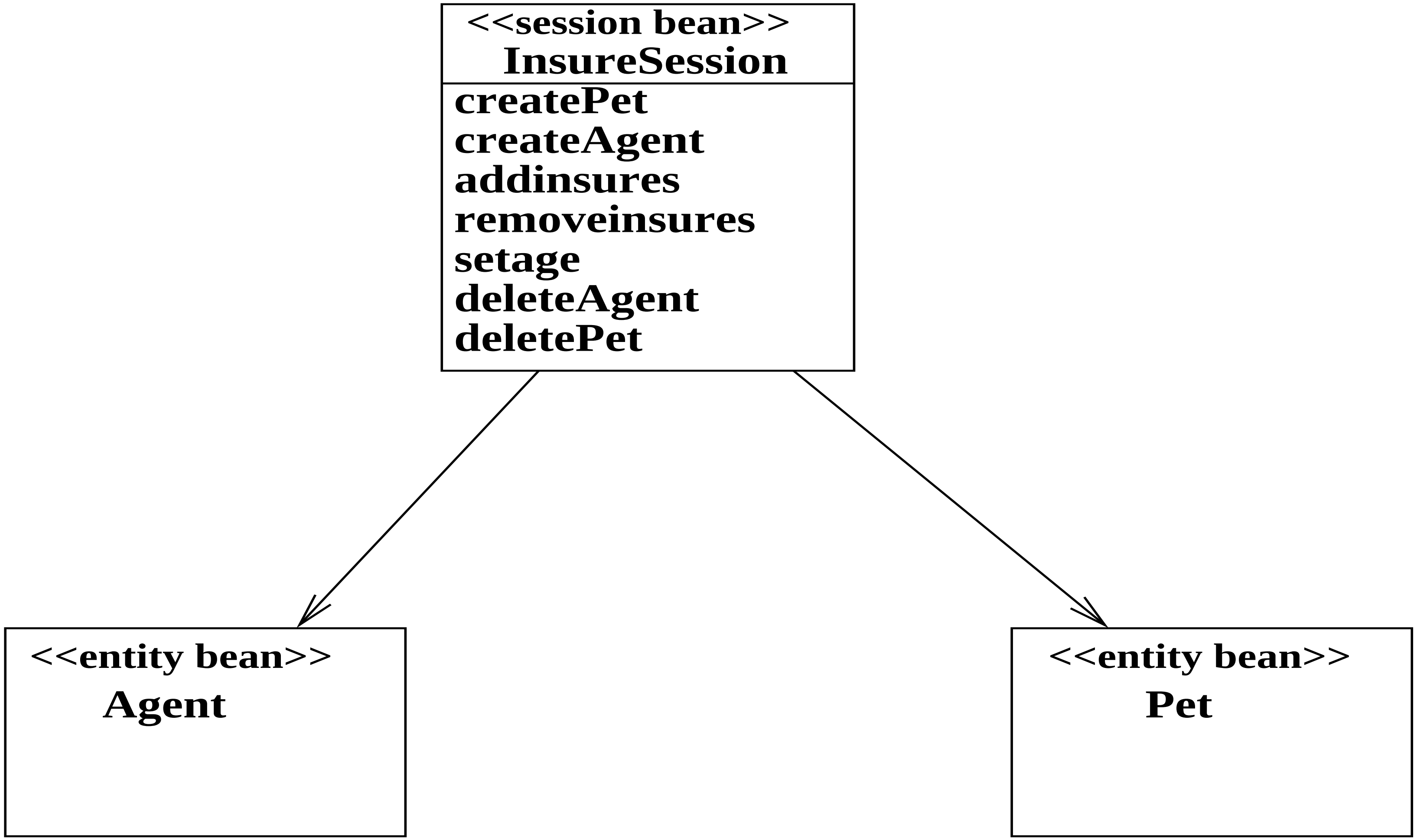
$$commission = insures.fee.sum/10$$

links state of *Agent* and *Pet*, so that an operation *setage* which affects state of *Pet* may also require changes to data of connected *Agent* objects.

A session bean component is therefore required, which ensures the inter-class constraint by carrying out updates to *Pet* and *Agent* objects within single transactions.



PIM of pet insurance system



Architecture of insurance system

Transaction example

The use cases *addinsures*, *removeinsures*, *setage* and *deletePet* all involve both entity beans.

For example, *addinsures* could have outline code:

```
public void addinsures(String agentId, String petId)
{ Agent agentx = agenthome.findByPrimaryKey(agentId);
  Pet petx = pethome.findByPrimaryKey(petId);
  List insuresx = agentx.getinsures();
  insuresx.add(petx);
  agentx.setcommission(insuresx.getfee().sum()/10);
}
```

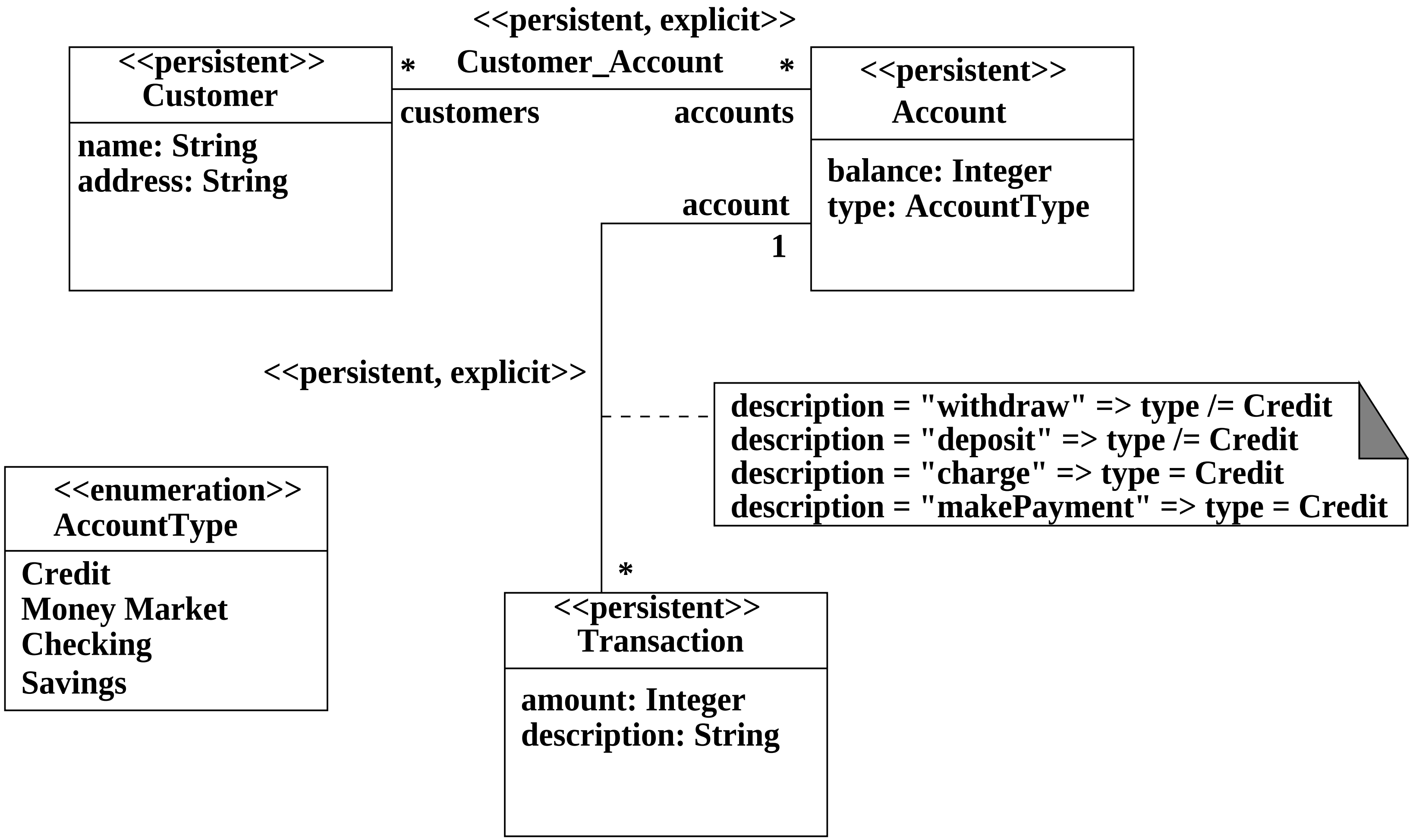
This is a complete transaction: updates to the agent commission and set of insured pets should either both succeed or both fail (be undone).

Online Bank System

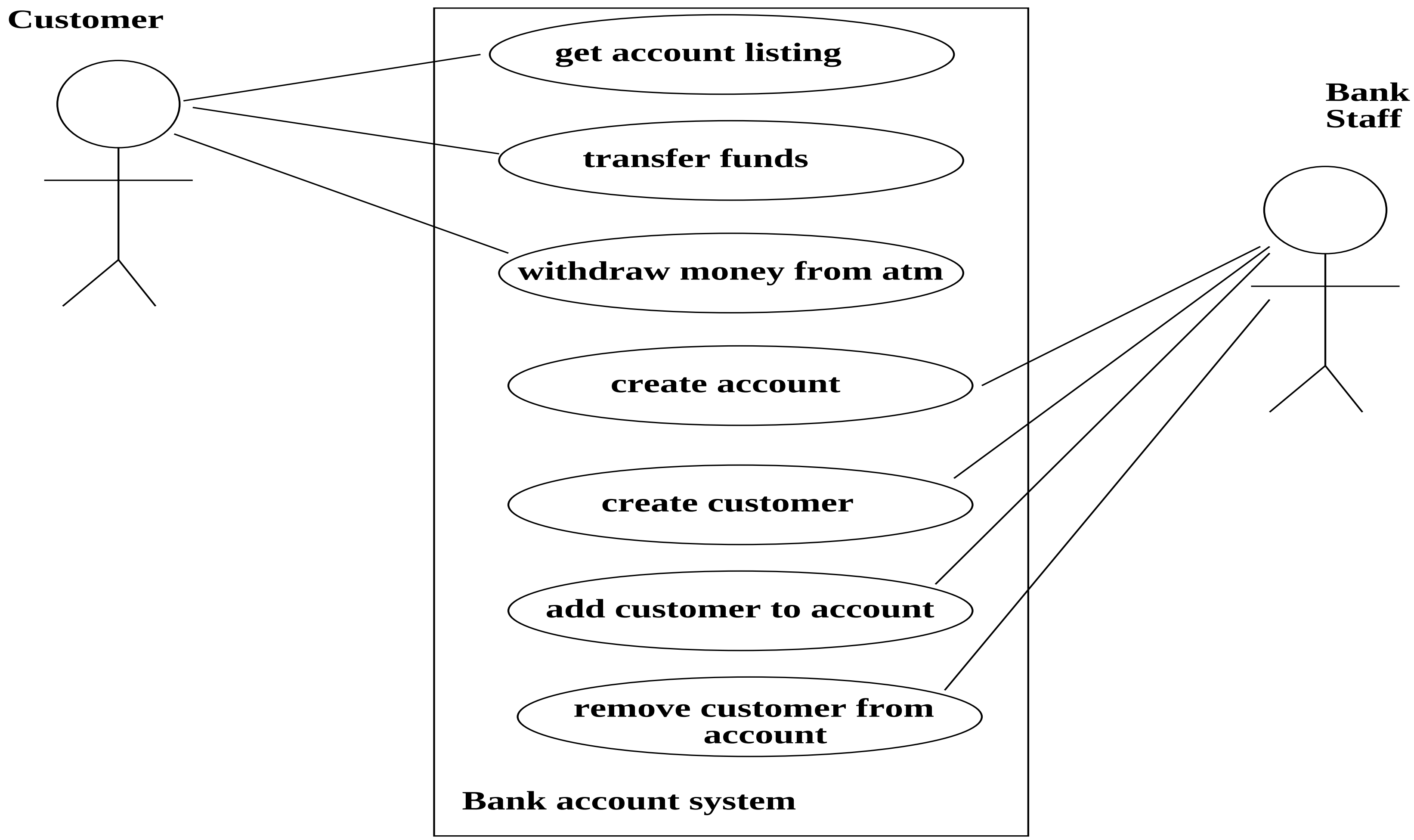
This is basic online banking system, with entities *Account*, *Customer* and *Tx* (transactions). Constraints of system are placed on *Account*-*Tx* association. Will be enforced by *TxControllerBean* session bean.

System has web interface for customers to view their accounts, and do transfers, and non-web interface for bank staff to create accounts and customers and to add or remove customers from accounts.

Constraint between *Tx* and *Account* is managed by *TxControllerBean* session bean, which only permits transactions to be processed if they satisfy the constraint.



PIM class diagram of account system



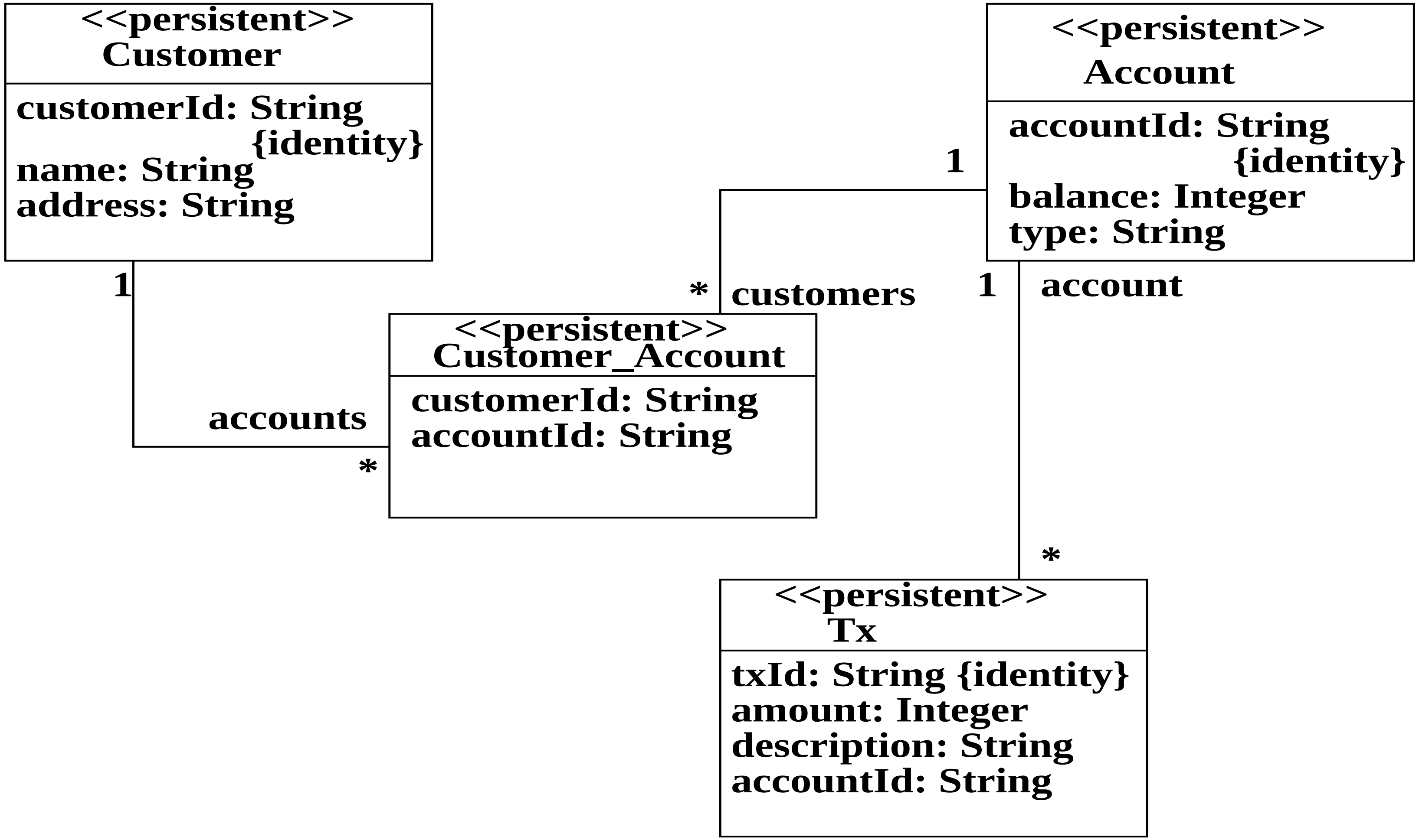
Use cases of account system

Design of account system

We need to take the following steps:

- Transform the PIM class diagram to a class diagram for a relational data model implementation.
- Identify components and architecture of the system.

The basic idea of the architecture is to use session beans to implement the use cases (cf. the Session Facade pattern), operating on entity beans for each entity.



PSM class diagram of account system

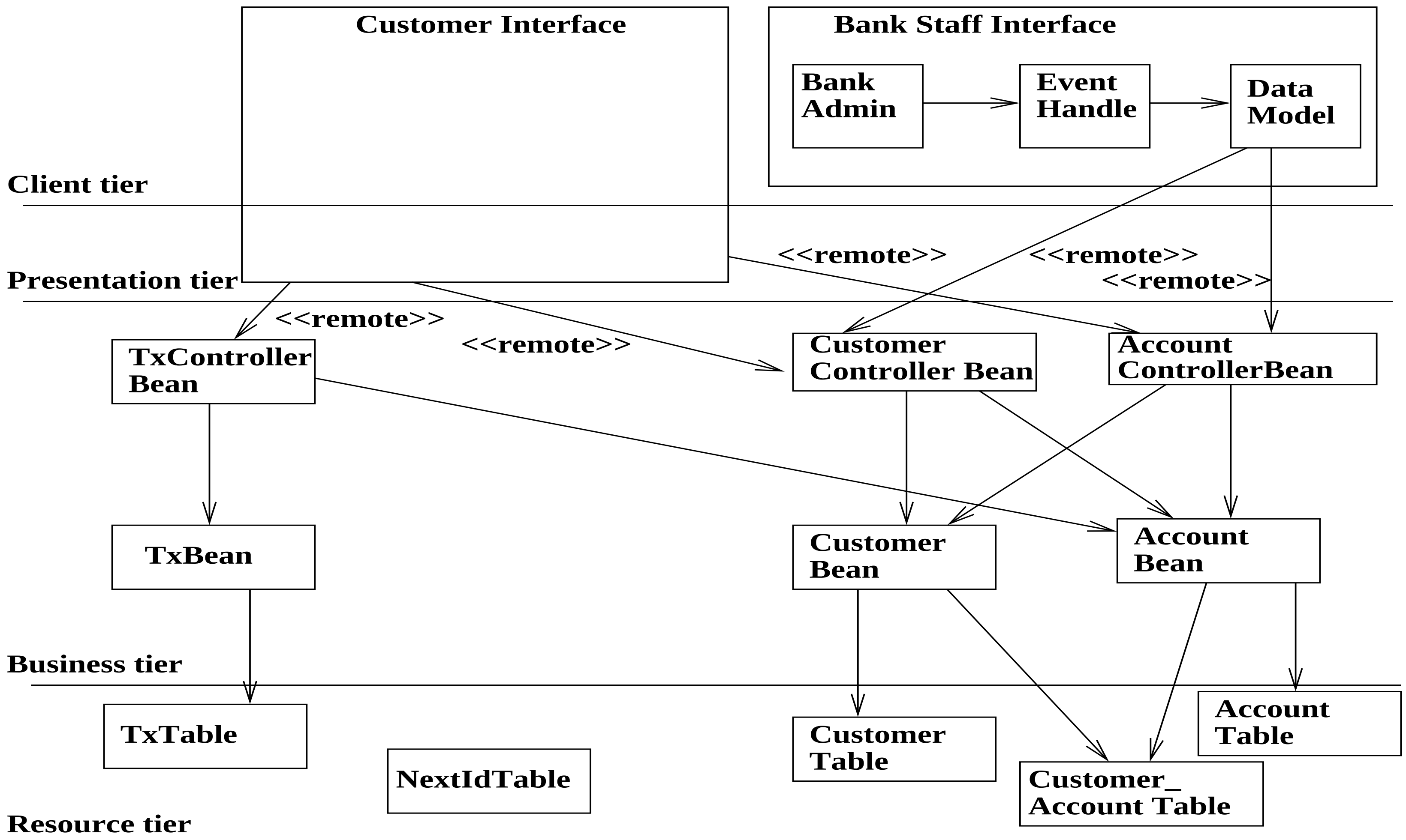
Components of account system

Introduce *AccountDetails*, *CustomerDetails* and *TxDetails* value object classes to transfer entity data.

Interface for bank staff will be a Swing GUI, sending commands to the session beans *AccountController*, etc.

Because *Customer*, *Account* are targets of associations (on the 'one' side of an association in design data model), their entity beans must be accessed locally, not remotely.

Current maximum id used in each entity table is stored in a *NextId* table. This is used to assign new (unused) ids for new instances of *Customer*, *Account* and *Transaction*.



Architecture of account system

Architecture validity

The session beans are *TxControllerBean*, *CustomerControllerBean*, *AccountControllerBean*.

Although there is shared write access in this example, by the session beans on *AccountBean*, the updates by *AccountControllerBean* and *CustomerControllerBean* on *AccountBean* cannot affect the constraints linking *Tx* and *Account*, so this architecture is valid.

Business tier components of account system

The session beans are:

- *AccountControllerBean*, implementing the *createAccount*, *addCustomerToAccount* and *removeCustomerFromAccount* use cases.
- *TxControllerBean*, implementing the *getAccountListing*, *transferFunds* and *withdrawMoney* use cases.
- *CustomerControllerBean*, implementing the *createCustomer* use case.

These components encapsulate the use cases of the system, as operations which make use of the entity beans. In particular all use cases for the Customer actor are implemented by *TxControllerBean*, and those for the Bank Staff are implemented by the other session beans.

Remote access is used for these beans because they may be used by presentation tier components on remote computers (eg, the web interface elements, which may reside on dedicated computers, separate to computers running business tier).

The entity beans are:

- *AccountBean*.
- *TxBean*.
- *CustomerBean*.
- *NextIdBean*.

Local access is used for these components, because they represent entities within the same database, connected by reference relationships. Such navigation between data would be very inefficient if carried out by remote method calls.

In addition there are auxiliary helper classes:

- *AccountDetails*, *CustomerDetails*, *TxDetails* value objects for the entities.
- *DBHelper* – used to generate next primary key values.
- *DomainUtil* – holds information about allowed types of account.
- *EJBGetter* – encapsulates bean lookup methods (cf, Service Locator pattern).

Data is passed between presentation and business tiers as $*$*Details* objects, which hide details of the entity beans from higher tiers. They are examples of *value objects*.

An alternative architecture would combine the account and customer session beans into a single *StaffOperations* session bean. This would reduce the number of dependencies in the system, but decrease modularity.

EIS Design Issues

Design issues for EIS cover all tiers of an EIS application, from security protection to database interaction approaches.

Examples include:

- Data security
- Separating presentation, business logic and data processing code
- Pooling database connections

Data security

- Passwords should only be stored in encrypted format: so they can be compared with (encrypted) user input but never exposed
- Organisations which send your password to you on request must be storing the data in unencrypted form! Instead, give option to user to reset their password.
- Https should be used systematically in all security-critical parts of a website.

Remove web-specific coding from business tier

Business tier code should not refer to HTTP request structures:

```
public class House
{ String address;
  String style;

  public House(HttpServletRequest req)
  { address = req.getParameter("address");
    style = req.getParameter("style");
  }
}
```

Using *HttpServletRequest* as input parameter type prevents non-web clients from using this business object. Instead, use data based on PIM or PSM class diagram of the system:

```
public class House
{ String address;
  String style;

  public House(String addr, String stl)
  { address = addr;
    style = stl;
  }
}
```

Separation of code

Another important principle is to separate presentation, business logic and data processing. Code concerned with database interaction should be separated from presentation (UI) code and from business logic, to improve flexibility.

EIS components are designed for specific tasks and give basis of this separation:

- Controller and view components for presentation processing
- Session beans for business processing
- Entity beans for complex business data

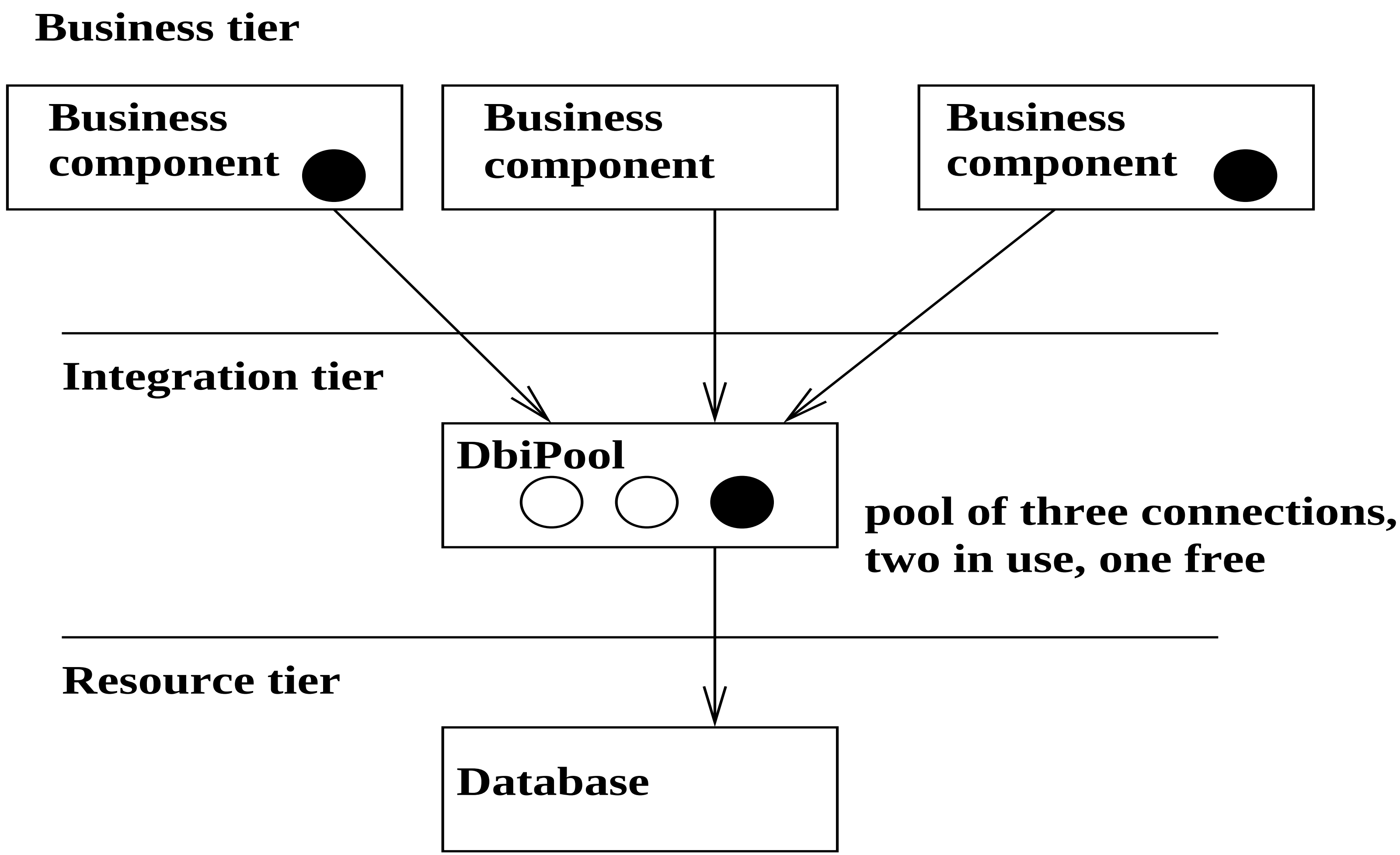
Database connection pooling

A particular technique useful for increasing the efficiency of a database interface is introduction of a connection pool: creating a connection to a database is expensive, and should be minimised.

This can be achieved by creating a connection management component, *ConnectionPool*, which holds a set of pre-initialised connections.

Components which require a connection must ask the pool for a free connection: $con = pool.getConnection()$. When they have finished using it they must return it to free set: $pool.returnConnection(con)$.

JDBC provides pooling in $javax.sql.DataSource$.



Introducing a connection pool