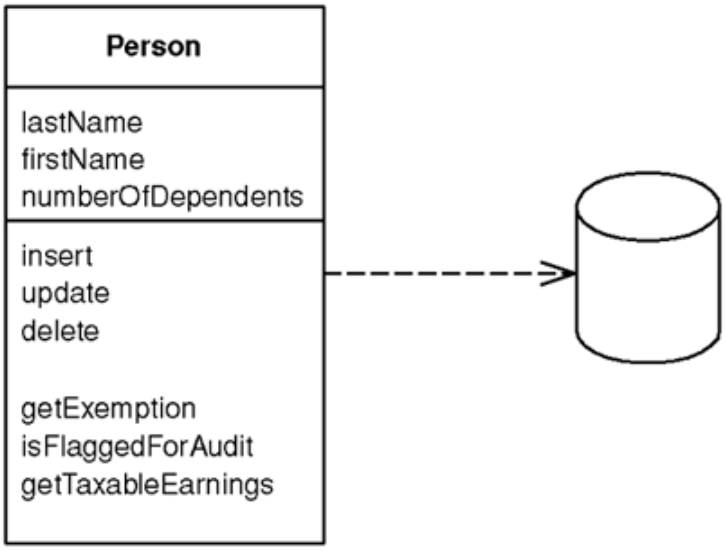
## Active Record

*An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.*




*An object carries both data and behavior. Much of this data is persistent and needs to be stored in a database. Active Record uses the most obvious approach, putting data access logic in the domain object. This way all people know how to read and write their data to and from the database.*

### How It Works

The essence of an Active Record is a Domain Model (116) in which the classes match very closely the record structure of an underlying database. Each Active Record is responsible for saving and loading to the database and also for any domain logic that acts on the data. This may be all the domain logic in the application, or you may find that some domain logic is held in Transaction Scripts (110) with common and data-oriented code in the Active Record.

The data structure of the Active Record should exactly match that of the database: one field in the class for each column in the table. Type the fields the way the SQL interface gives you the data—don't do any conversion at this stage. You may consider Foreign Key Mapping (236), but you may also leave the foreign keys as they are. You can use views or tables with Active Record, although updates through views are obviously harder. Views are particularly useful for reporting purposes.

The Active Record class typically has methods that do the following:

- Construct an instance of the Active Record from a SQL result set row
- Construct a new instance for later insertion into the table
- Static finder methods to wrap commonly used SQL queries and return Active Record objects
- Update the database and insert into it the data in the Active Record
- Get and set the fields

### Implement some pieces of business logic

The getting and setting methods can do some other intelligent things, such as convert from SQL-oriented types to better in-memory types. Also, if you ask for a related table, the getting method can return the appropriate Active Record, even if you aren't using Identity Field (216) on the data structure (by doing a lookup).

In this pattern the classes are convenient, but they don't hide the fact that a relational database is present. As a result you usually see fewer of the other object-relational mapping patterns present when you're using Active Record.

Active Record is very similar to Row Data Gateway (152). The principal difference is that a Row Data Gateway (152) contains only database access while an Active Record contains both data source and domain logic. Like most boundaries in software, the line between the two isn't terribly sharp, but it's useful.

Because of the close coupling between the Active Record and the database, I more often see static find methods in this pattern. However, there's no reason that you can't separate out the find methods into a separate class, as I discussed with Row Data Gateway (152), and that is better for testing.

As with the other tabular patterns, you can use Active Record with a view or query as well as a table.

### When to Use It

Active Record is a good choice for domain logic that isn't too complex, such as creates, reads, updates, and deletes. Derivations and validations based on a single record work well in this structure.

In an initial design for a Domain Model (116) the main choice is between Active Record and Data Mapper (165). Active Record has the primary advantage of simplicity. It's easy to build Active Records, and they are easy to understand. Their primary problem is that they work well only if the Active Record objects correspond directly to the database tables: an isomorphic schema. If your business logic is complex, you'll soon want to use your object's direct relationships, collections, inheritance, and so forth. These don't map easily onto Active Record, and adding them piecemeal gets very messy. That's what will lead you to use Data Mapper (165) instead.

Another argument against Active Record is the fact that it couples the object design to the database design. This makes it more difficult to refactor either design as a project goes forward.

Active Record is a good pattern to consider if you're using Transaction Script (110) and are beginning to feel the pain of code duplication and the difficulty in updating scripts and tables that Transaction Script (110) often brings. In this case you can gradually start creating Active Records and then slowly refactor behavior into them. It often helps to wrap the tables as a Gateway (466) first, and then start moving behavior so that the tables evolve to a Active Record.

Example: A Simple Person (Java)

This is a simple, even simplistic, example to show how the bones of Active Record work. We begin with a basic Person class.

class Person...

```
  private String lastName;
  private String firstName;
  private int numberOfDependents;
```

There's also an ID field in the superclass.

The database is set up with the same structure.

```
create table people (ID int primary key, lastname varchar,
                     firstname varchar, number_of_dependents int)
```

To load an object, the person class acts as the finder and also performs the load. It uses static methods on the person class.

class Person...

```
  private final static String findStatementString =
      "SELECT id, lastname, firstname, number_of_dependents" +
      "  FROM people" +
      "  WHERE id = ?";
  public static Person find(Long id) {
    Person result = (Person) Registry.getPerson(id);
    if (result != null) return result;
    PreparedStatement findStatement = null;
    ResultSet rs = null;
    try {
      findStatement = DB.prepare(findStatementString);
      findStatement.setLong(1, id.longValue());
      rs = findStatement.executeQuery();
      rs.next();
```

```
      result = load(rs);
      return result;
    } catch (SQLException e) {
      throw new ApplicationException(e);
    } finally {
      DB.cleanUp(findStatement, rs);
    }
  }
  public static Person find(long id) {
    return find(new Long(id));
  }
  public static Person load(ResultSet rs) throws SQLException {
    Long id = new Long(rs.getLong(1));
    Person result = (Person) Registry.getPerson(id);
    if (result != null) return result;
    String lastNameArg = rs.getString(2);
    String firstNameArg = rs.getString(3);
    int numDependentsArg = rs.getInt(4);
    result = new Person(id, lastNameArg, firstNameArg, numDependentsArg);
    Registry.addPerson(result);
    return result;
  }
```

Updating an object takes a simple instance method.

class Person...

```
  private final static String updateStatementString =
      "UPDATE people" +
      "  set lastname = ?, firstname = ?, number_of_dependents = ?" +
      "  where id = ?";
```

```
  public void update() {
    PreparedStatement updateStatement = null;
    try {
      updateStatement = DB.prepare(updateStatementString);
      updateStatement.setString(1, lastName);
      updateStatement.setString(2, firstName);
      updateStatement.setInt(3, numberOfDependents);
      updateStatement.setInt(4, getID().intValue());
      updateStatement.execute();
    } catch (Exception e) {
      throw new ApplicationException(e);
    } finally {
      DB.cleanUp(updateStatement);
    }
  }
```

Insertions are also mostly pretty simple.

class Person...

```
  private final static String insertStatementString =
      "INSERT INTO people VALUES (?, ?, ?, ?)";
  public Long insert() {
    PreparedStatement insertStatement = null;
    try {
      insertStatement = DB.prepare(insertStatementString);
      setID(findNextDatabaseId());
      insertStatement.setInt(1, getID().intValue());
      insertStatement.setString(2, lastName);
      insertStatement.setString(3, firstName);
      insertStatement.setInt(4, numberOfDependents);
```

```
      insertStatement.execute();
      Registry.addPerson(this);
      return getID();
    } catch (Exception e) {
      throw new ApplicationException(e);
    } finally {
      DB.cleanUp(insertStatement);
    }
  }
```

Any business logic, such as calculating the exemption, sits directly in the Person class.

class Person...

```
  public Money getExemption() {
    Money baseExemption = Money.dollars(1500);
    Money dependentExemption = Money.dollars(750);
    return
baseExemption.add(dependentExemption.multiply(this.getNumberOfDependents(
)));
  }
```