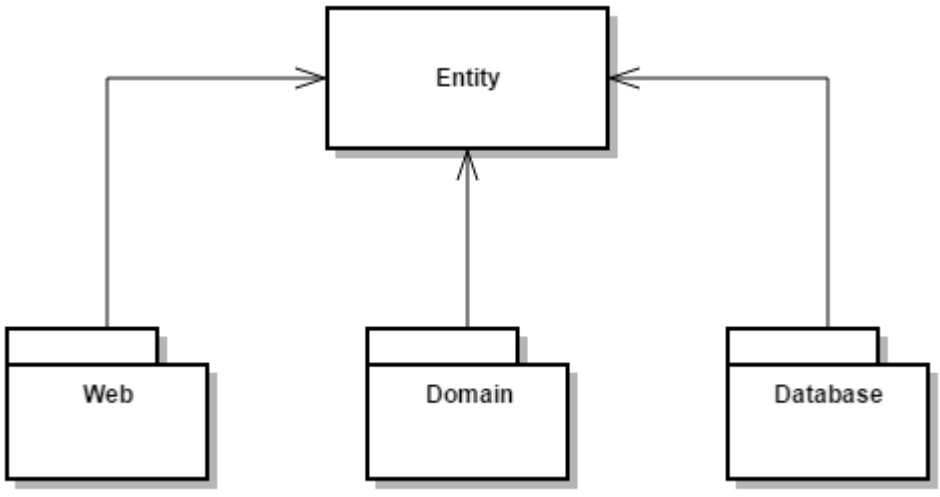
# Domain Object Persistence Approaches

Перевод статьи: Domain Object Persistence (Grzegorz Ziemoński)

Persistence (или отображение/сохранение объектов в БД) является частью почти каждого приложения. К сожалению, сохранение объектов предметной области (уровня домена) является непростой задачей, которая создает много проблем, особенно когда мы используем реляционную базу данных. Давайте рассмотрим различные подходы к решению данной проблемы.

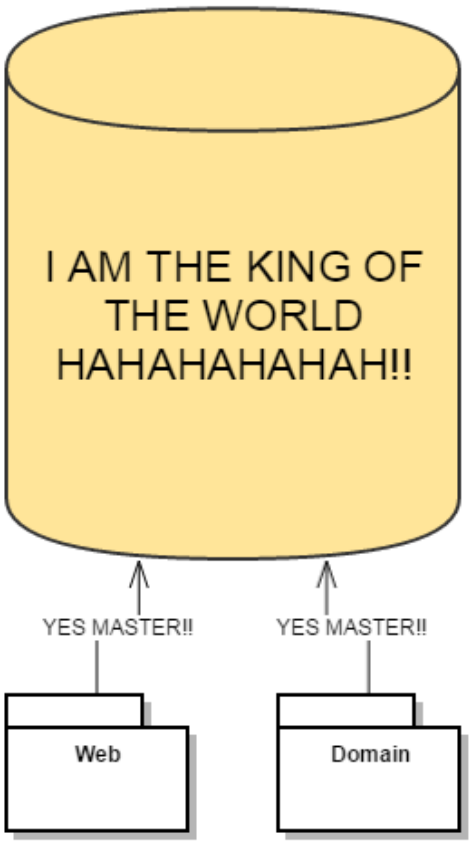
## All-in-one Model

Самый простой подход, присутствующий в большинстве приложений, заключается в использовании одной и той же модели (класса) на инфраструктурном утовне (т.е. для сохранения в БД), на уровне домена, а так же, как часто бывает, и на уровнях приложения и представления. Упрощенно мы получаем что-то вроде этого:



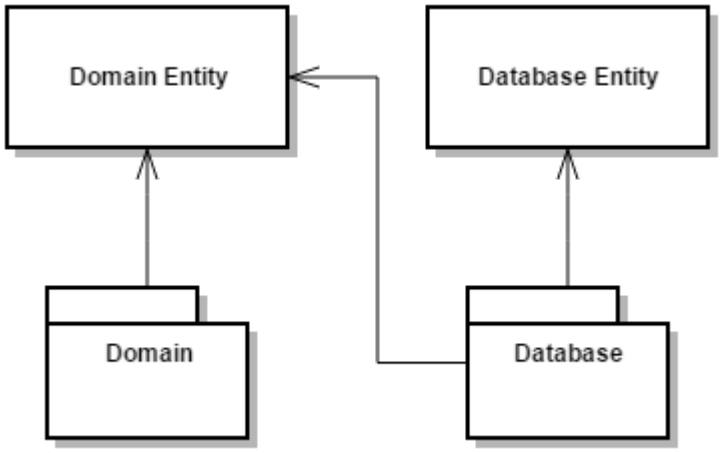
При данном подходе класс(ы) предметной области скорее всего содержат некоторые аннотации и/или дополнительные поля, которые необходимы для отображения объектов данного типа(-ов) в базе данных (например, аннотации JPA). Т.е. мы делаем все наши классы предметной области зависимым от деталей базы данных. И как только мы обновляем что-либо в схеме базы данных, мы должны обновить классы предметной области. Проблемы, связанные с механизмами ORM распространяются по всему приложению, например, мы могли бы получить lazy-init исключение в нашем веб-контроллере. Что еще хуже (с точки зрения домена), разработчики часто добавляют проверочные аннотации к своим сущностям, чтобы гарантировать, что данные, записанные в базу данных, верны. На данный момент люди крайне неохотно добавляют логику (методы/поведение) предметной области в такие классы, потому что они уже и так содержат много кода, связанного с базой данных. Давайте добавим кучу геттеров и сеттеров (некоторые инструменты отображения требуют этого), и мы получим рецепт для архитектуры, ориентированной на базу данных. И в этом нет ничего хорошего.

```java
@Entity
@Table(name = "TODO_LISTS")
public class TodoList {

    @Id
    @GeneratedValue
    @Column(name = "ID")
    private Long id;

    @Column(name = "NAME")
    @NotEmpty
    private String name;

    @OneToMany(mappedBy = "list")
    @NonNull
    private List<Todo> todos;

    // getters and setters
}
```



## Separate Models (Mapping Approach)

Принимая во внимание все проблемы предыдущего подхода, следующим простейшим решением может быть создание отдельных классов (моделей) для уровня домена (классы сущностей предметной области) и для инфраструктурного уровня (для сохранения в БД).




Теперь уровень инфраструктуры (базы данных) отвечает за mapping (сопоставление) сущностей базы данных с объектами домена и наоборот. Классы предметной области ничего не знают об классах (объектах) инфраструктурного уровня (о базе данных), что, безусловно, хорошо. Мы также свободны от проблем, связанных с JPA, таких как lazy-init исключения и т.п.

В коде наши модели могут выглядеть следующим образом. Класс предметной области:

```
1 public class TodoList {
2     private Long id;
3     private String name;
4     private List<Todo> todos;
5 
6     // business methods
7 
8     // builder or setters
9 }
```

Класс для отображения в БД:

```java
@Entity
@Table(name = "TODO_LISTS")
public class DbTodoList {

    @Id
    @GeneratedValue
    @Column(name = "ID")
    private Long id;

    @Column(name = "NAME")
    @NotEmpty
    private String name;

    @OneToMany(mappedBy = "list")
    @NonNull
    private List<DbTodo> todos;

    // getters and setters
}
```

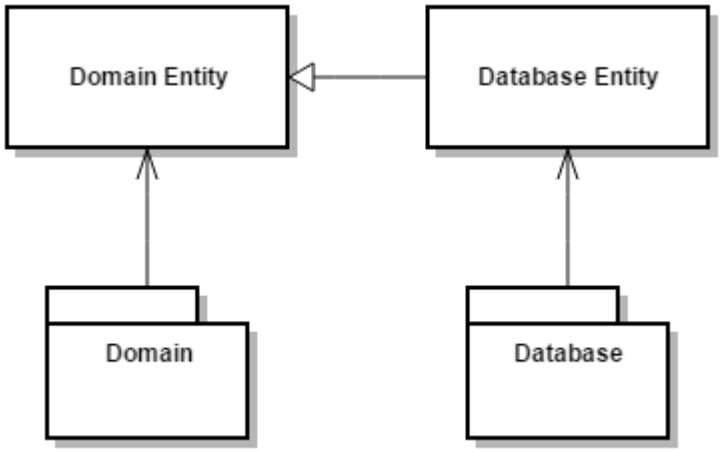
Очевидным недостатком этого подхода является то, что мы должны добавить много шаблонного mapping-кода, такого как этот:

```java
public class Db2DomainTodoListMapper {

    public static TodoList toDomainModel(DbTodoList dbTodoList) {
        TodoList todoList = new TodoList();
        todoList.setId(dbTodoList.getId());
        todoList.setName(dbTodoList.getName());
        todoList.setTodos(toDomainModel(dbTodoList.getTodos()));
        return todoList;
    }

    // toDomainModel(List<DbTodo> todos) etc.
}
```

Если у нас много классов с большим количеством полей, это может быть неприятно (или, скорее, скучно). С другой стороны, сложные отображения, такие как enum-to-strategy или коллекции, довольно просты в этом решении. Кроме того, если мы используем builder'ы вместо setter'ов, то наша инкапсуляция значительно улучшается. Никто не может бесконтрольно изменить состояние нашего объекта.

## Inheritance Approach

Мы можем решить, что эти отдельные классы для уровня домена и уровня инфраструктуры, и дополнительные классы mapping'а делают наше приложение слишком сложным. В таком случае мы можем использовать наследование для достижения правильного отделения классов домена от деталей их сохранения в БД. Посмотрите на диаграмму:




Теперь мы используем уровень базы данных, чтобы обеспечить классы предметной области необходимым состоянием.

Посмотрите на код и насладитесь красотой чистого, абстрактного класса предметной области и его бедного слуги (объекта для сохранения в БД):

```
1  public abstract class TodoList {
2      protected abstract Long getId();
3      protected abstract void setId(Long id);
4      protected abstract String getName();
5      protected abstract void setName(String name);
6      protected abstract List<Todo> getTodos();
7      protected abstract void setTodos(List<Todo> todos);
8
9      // business methods
10 }
```

```java
@Entity
@Table(name = "TODO_LISTS")
public class DbTodoList extends TodoList {

    @Id
    @GeneratedValue
    @Column(name = "ID")
    private Long id;

    @Column(name = "NAME")
    @NotEmpty
    private String name;

    @OneToMany(mappedBy = "list")
    @NonNull
    private List<Todo> todos;

    // getters and setters overriding the abstract methods
}
```

Этот подход показался мне очень странным, когда я впервые его увидел. Но он работает и работает очень хорошо. У нас есть необходимое разделение, и у нас нет никакого шаблонного кода. С точки зрения инкапсуляции, мы не должны выставлять состояние объекта предметной области напоказ, пока нам это действительно не понадобится. Идеальное решение?

Не совсем. Теперь мы снова сталкиваемся с проблемами, связанными с ORM, потому что “под капотом” наши доменные объекты – это объекты JPA. Кроме того, в этой настройке может быть довольно сложно эффективно использовать сопоставления (mapping’и), такие как коллекции или enum-to-strategy. Это не невозможно, но определенно сложнее, что может заставить коллег-разработчиков (или нас самих) неохотно решать эту проблему.