# Design Patterns

## MSc Computer Science

---

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

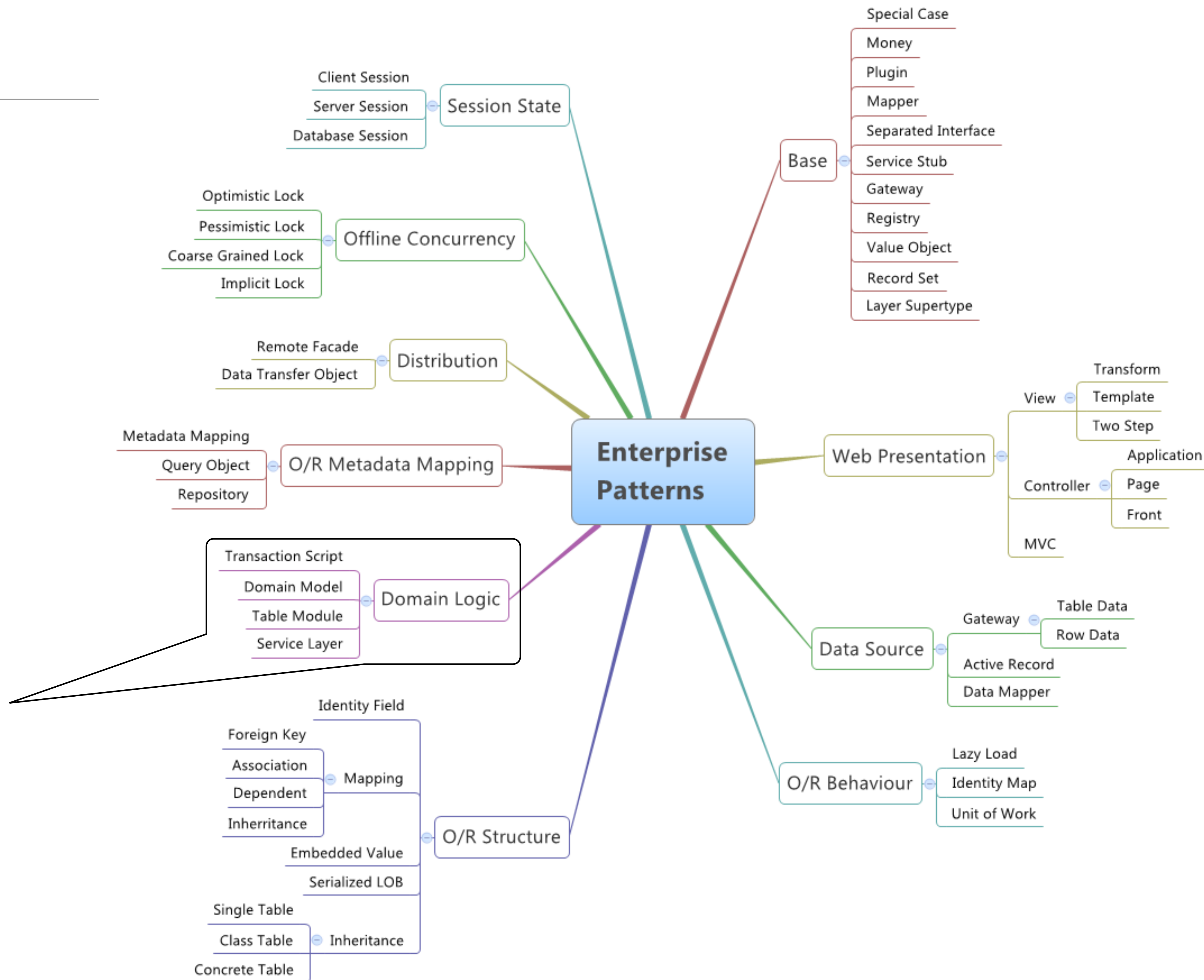
# Domain Logic

# Domain Logic

- Also also referred to as business logic.
- The work that this application needs to do for the domain you're working with.
- It involves calculations based on inputs and stored data, validation of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch, depending on commands received from the presentation.

# Domain Logic Patterns

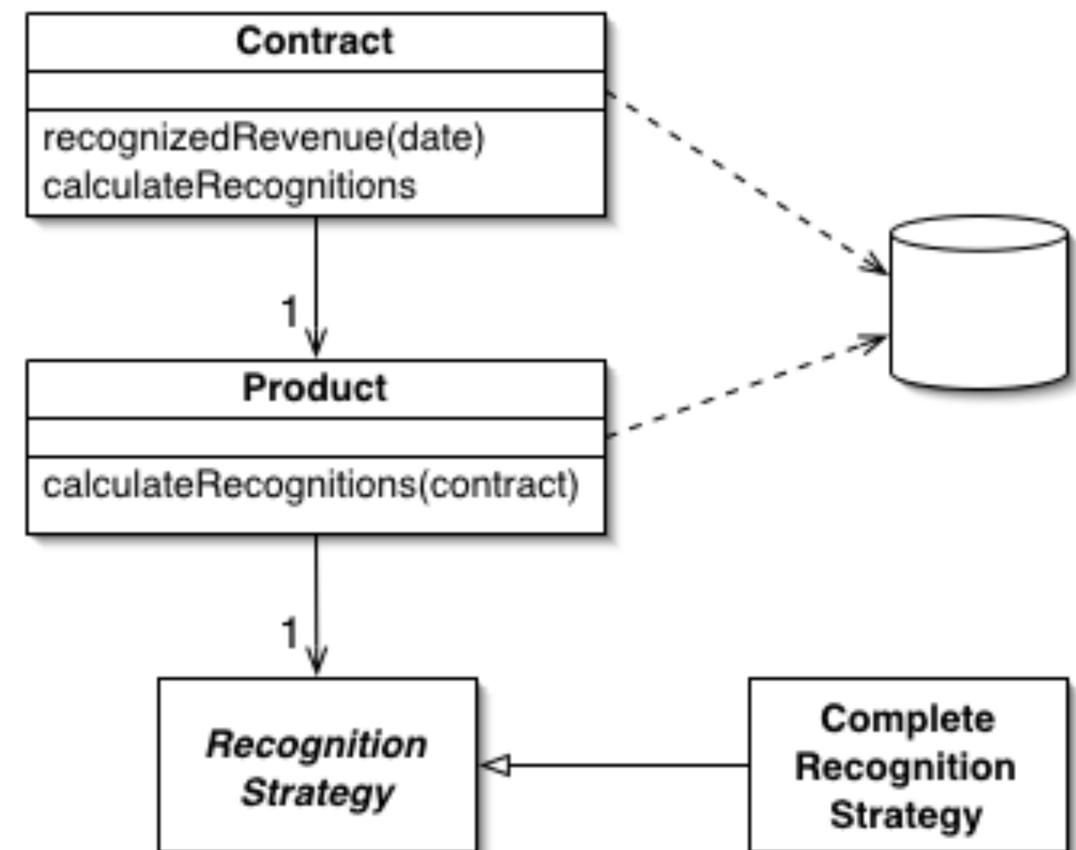
- Domain Model
- Table Module
- Service Layer

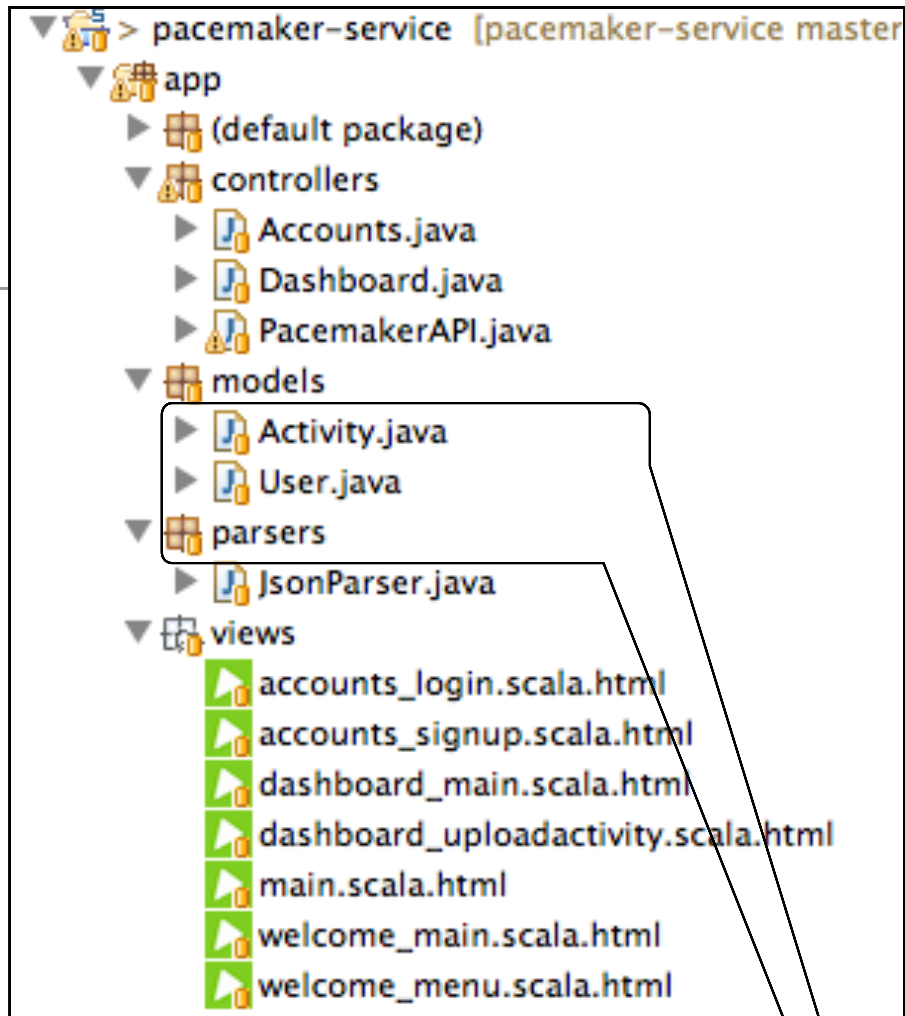


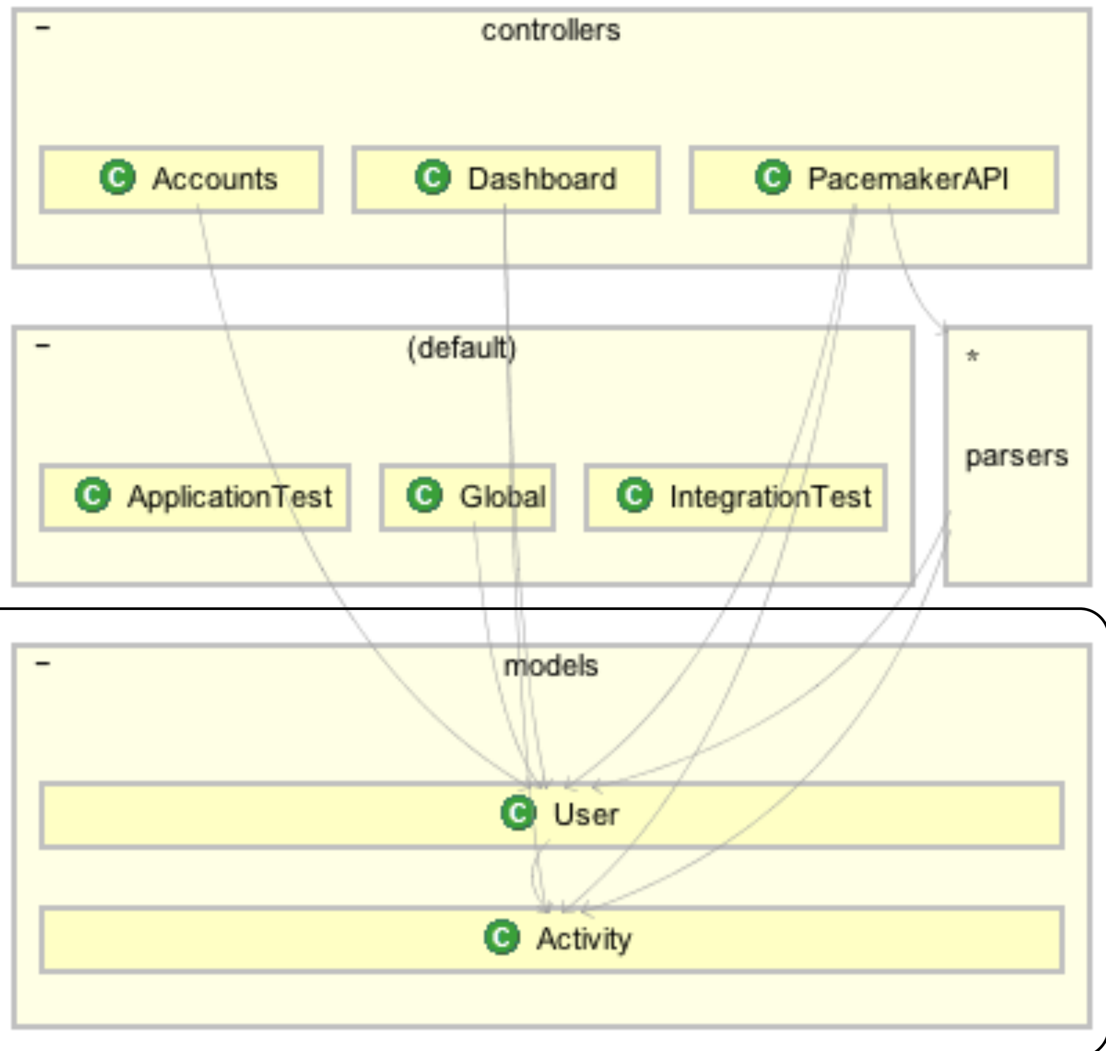
# Domain Model

An object model of the domain that incorporates both behaviour and data

- At its worst business logic can be very complex.
- Rules and logic describe many different cases and slants of behaviour, and it's this complexity that objects were designed to work with.
- A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form





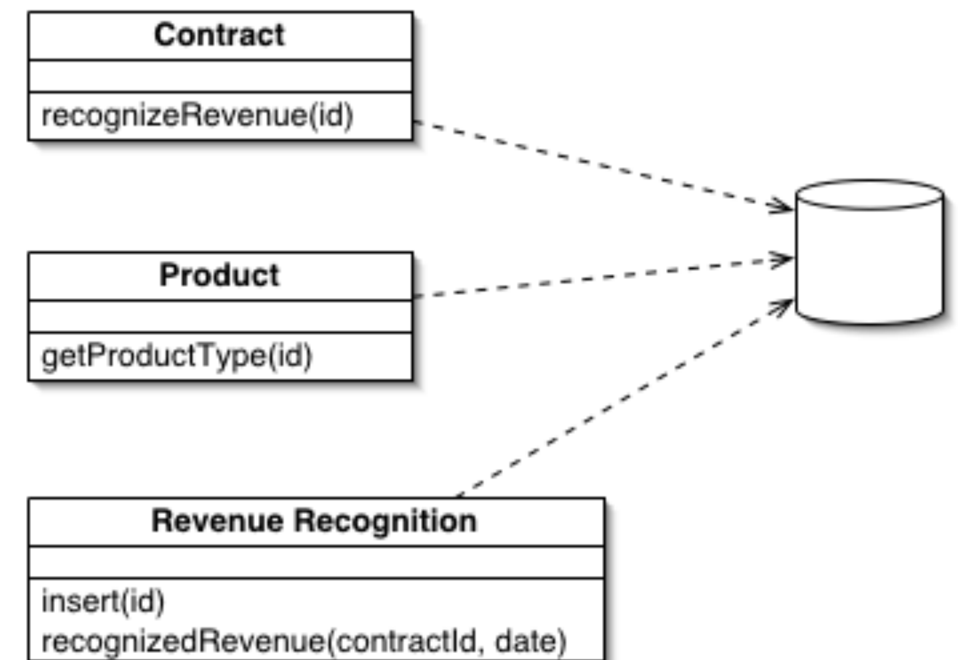




Domain Model

# Table Module (1)

A single instance that handles the business logic for all rows in a database table or view.

- One of the key messages of object orientation is bundling the data with the behaviour that uses it.
- The traditional object-oriented approach is based on objects with identity, along the lines of Domain Model. Thus, if we have an Employee class, any instance of it corresponds to a particular employee.
- This scheme works well because once we have a reference to an employee, we can execute operations, follow relationships, and gather data on him.



# Table Module (2)

- One of the problems with Domain Model is the interface with relational databases.
  - Domain Model keeps the database a arms length - usually requiring an Object Relation Mapping layer to persist the model to the database.
- A Table Module eschews ORM, and organizes domain logic with one class per table in the data-base, and a single instance of a class contains the various procedures that will act on the data.
- The primary distinction with Domain Model is that, if you have many orders, a Domain Model will have one order object per order while a Table Module will have one object to handle all orders.

# Service Layer

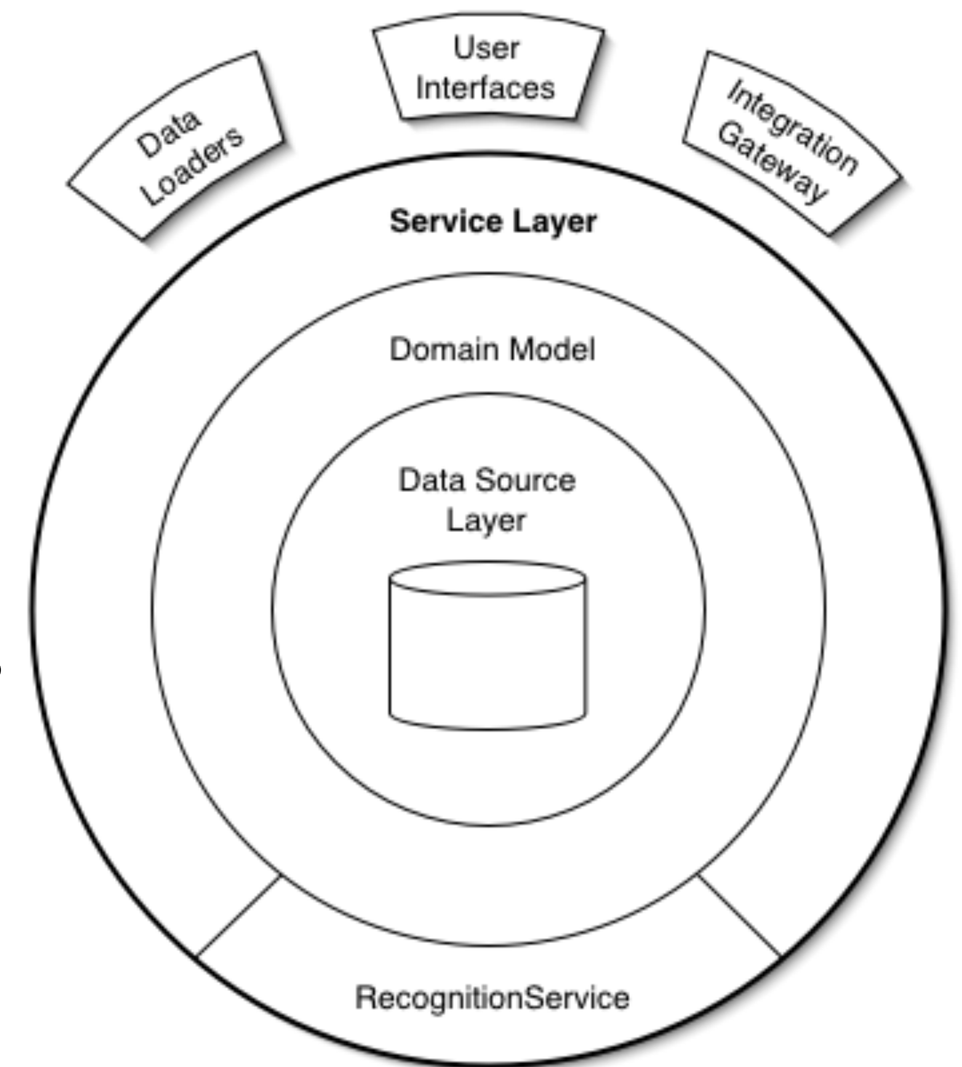
Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

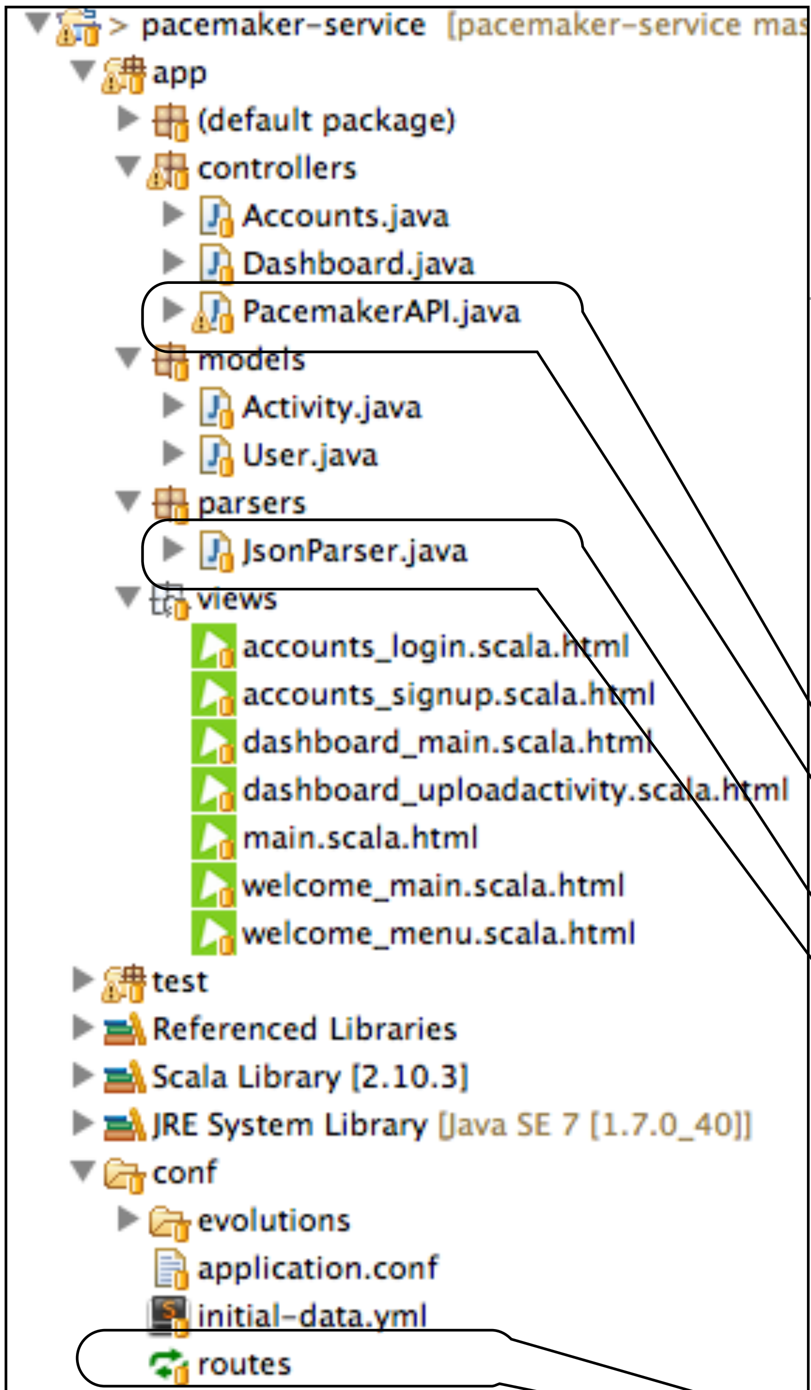
- Enterprise applications often require different kinds of interfaces to the data they store and the logic they implement

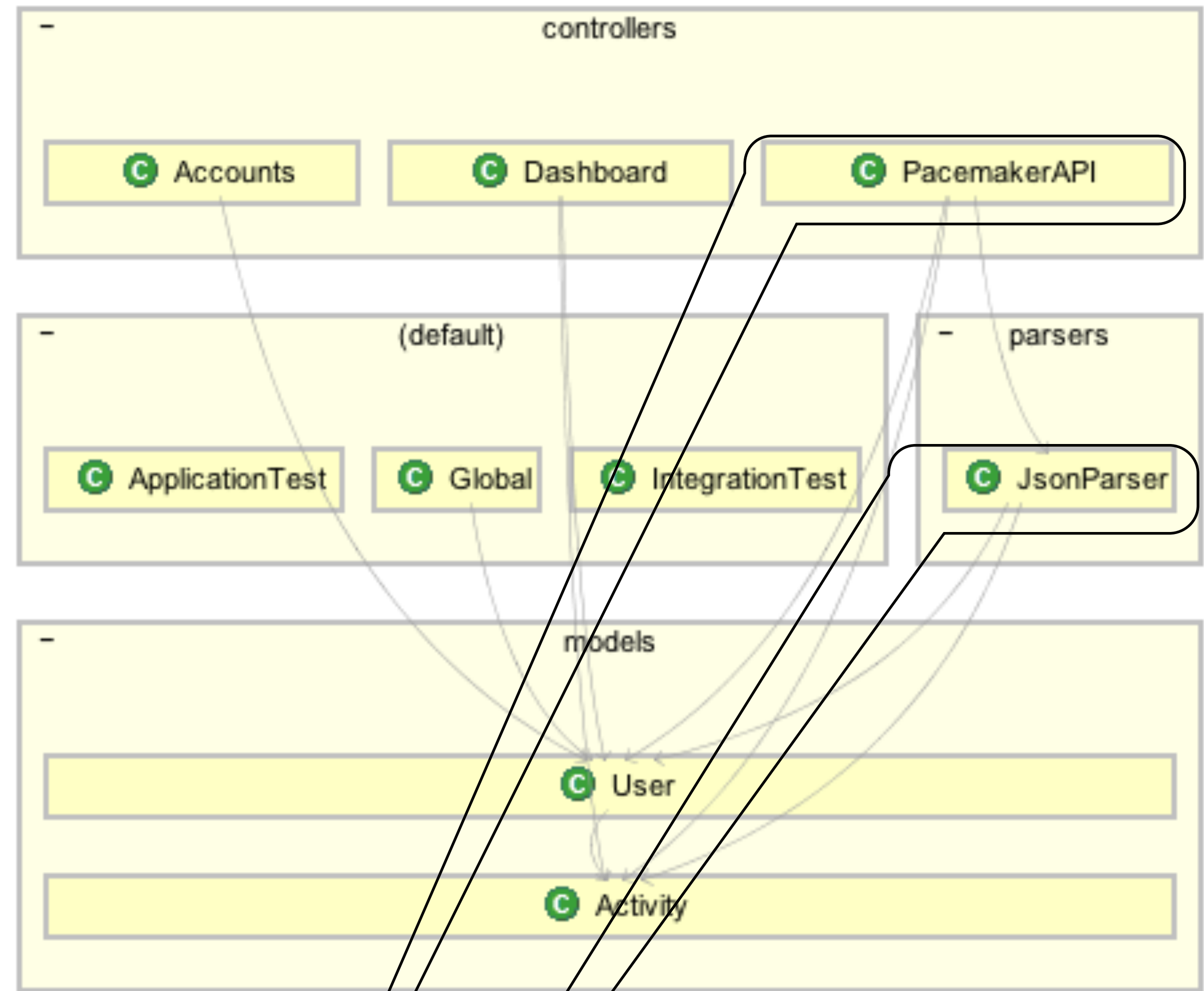
- These interfaces need common interactions with the application to access and manipulate its data and invoke its business logic.

- May be complex, involving transactions across multiple resources and the coordination of several responses to an action.

- A Service Layer defines an application's boundary and its set of available operations from the perspective of interfacing client layers.

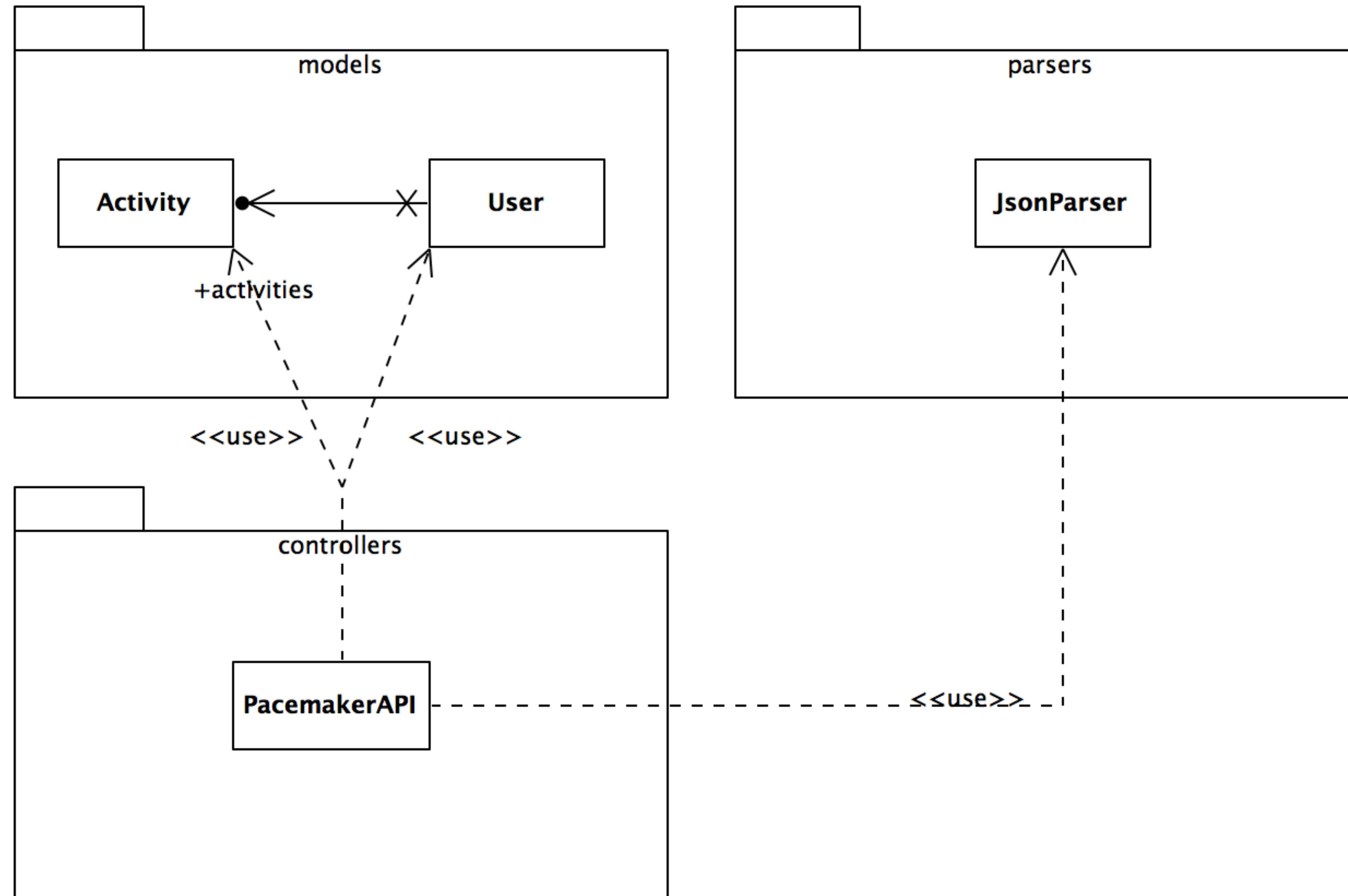


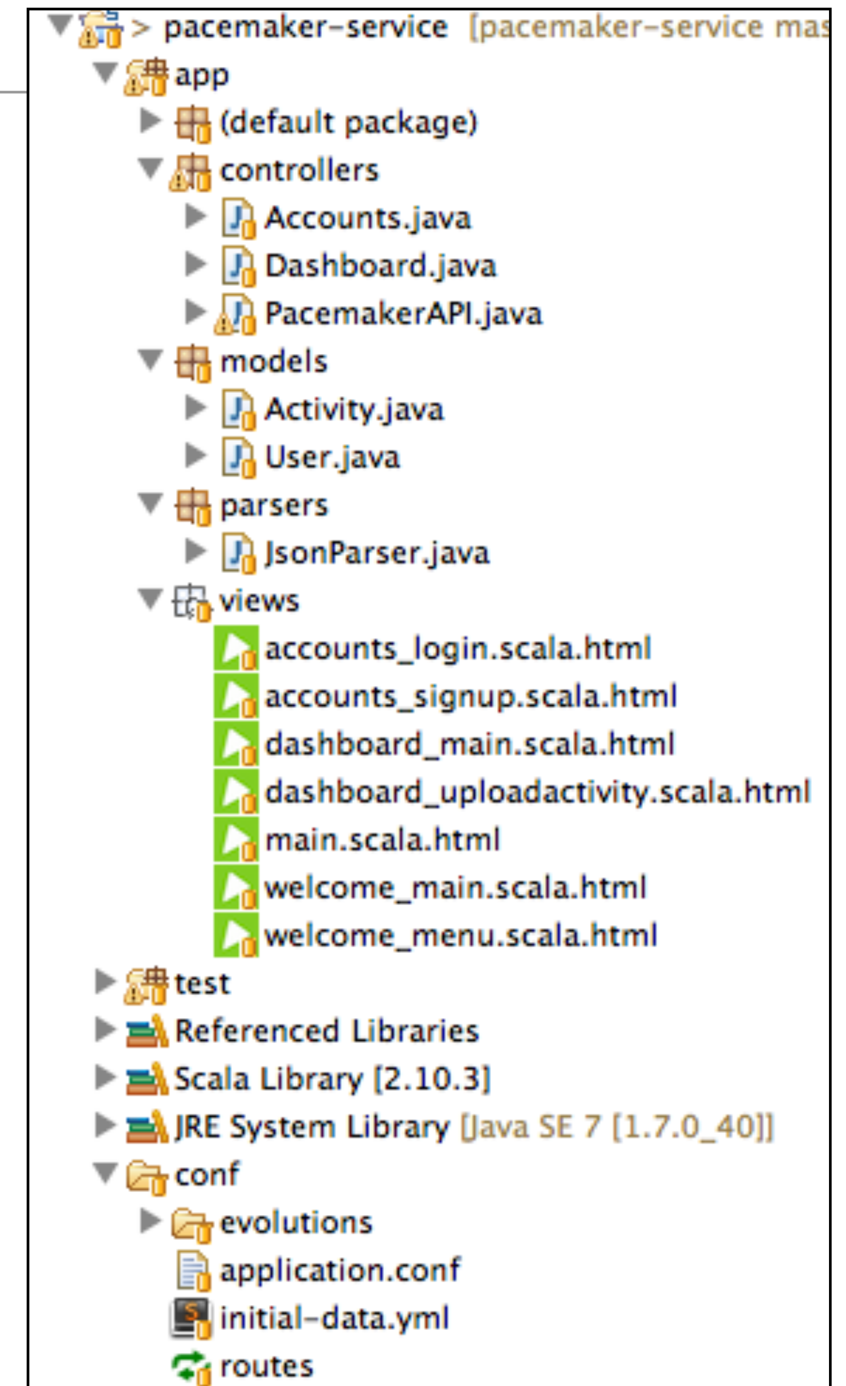


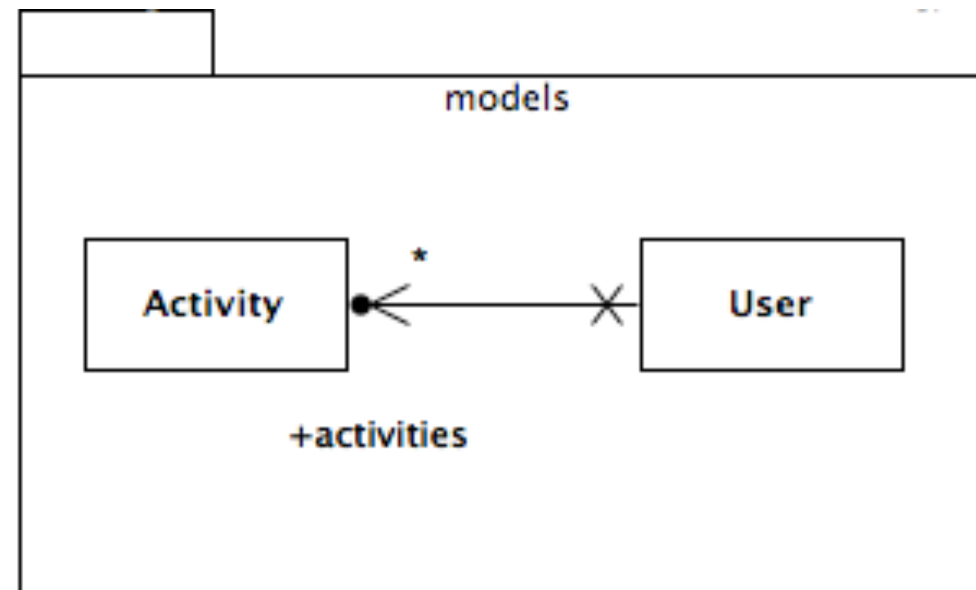

Service Layer

# Pacemaker-Service



# Domain Model




```
@Entity
public class User extends Model
{
  @Id
  @GeneratedValue
  public Long   id;
  public String firstname;
  public String lastname;
  public String email;
  public String password;

  @OneToMany(cascade=CascadeType.ALL)
  public List<Activity> activities = new ArrayList<Activity>();

  //…
}
```

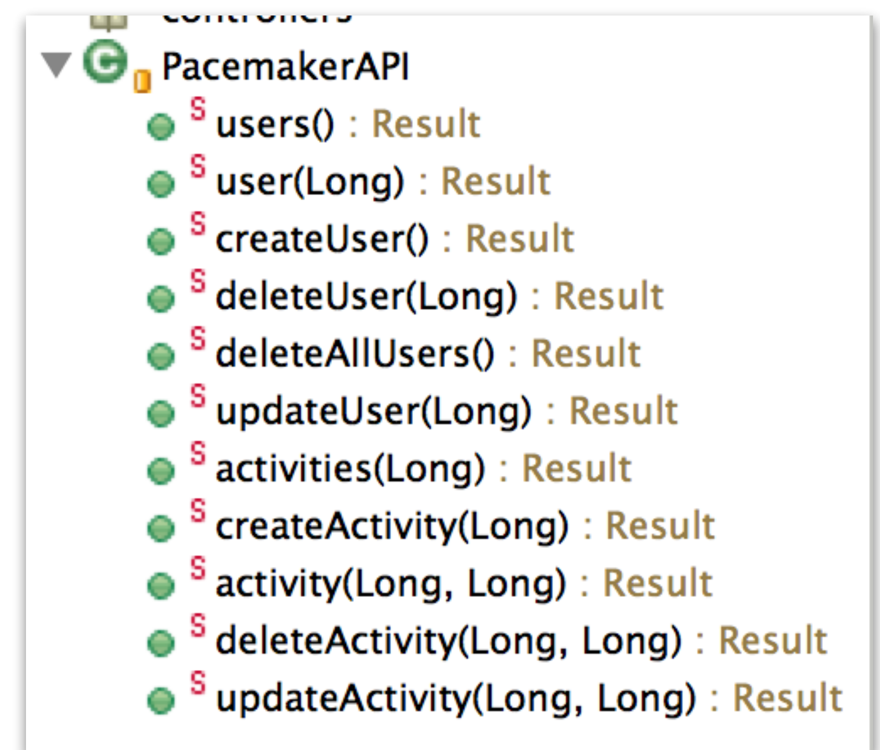
```
@Entity
public class Activity extends Model
{
  @Id
  @GeneratedValue
  public Long   id;
  public String type;
  public String location;
  public double distance;

  //…
}
```

# Service Layer: Routes + PacemakerAPI

```
GET     /api/users                                  controllers.PacemakerAPI.users()
DELETE  /api/users                                  controllers.PacemakerAPI.deleteAllUsers()
POST    /api/users                                  controllers.PacemakerAPI.createUser()

GET    /api/users/:id                               controllers.PacemakerAPI.user(id: Long)
DELETE /api/users/:id                               controllers.PacemakerAPI.deleteUser(id: Long)
PUT    /api/users/:id                               controllers.PacemakerAPI.updateUser(id: Long)

GET     /api/users/:userId/activities               controllers.PacemakerAPI.activities(userId: Long)
POST    /api/users/:userId/activities               controllers.PacemakerAPI.createActivity(userId: Long)

GET     /api/users/:userId/activities/:activityId  controllers.PacemakerAPI.activity(userId: Long, activityId:Long)
DELETE  /api/users/:userId/activities/:activityId  controllers.PacemakerAPI.deleteActivity(userId: Long, activityId:Long)
PUT     /api/users/:userId/activities/:activityId  controllers.PacemakerAPI.updateActivity(userId: Long, activityId:Long)
```

- Each route maps to a PacemakerAPI method

- Support standard Create/Read/Update/Delete operations

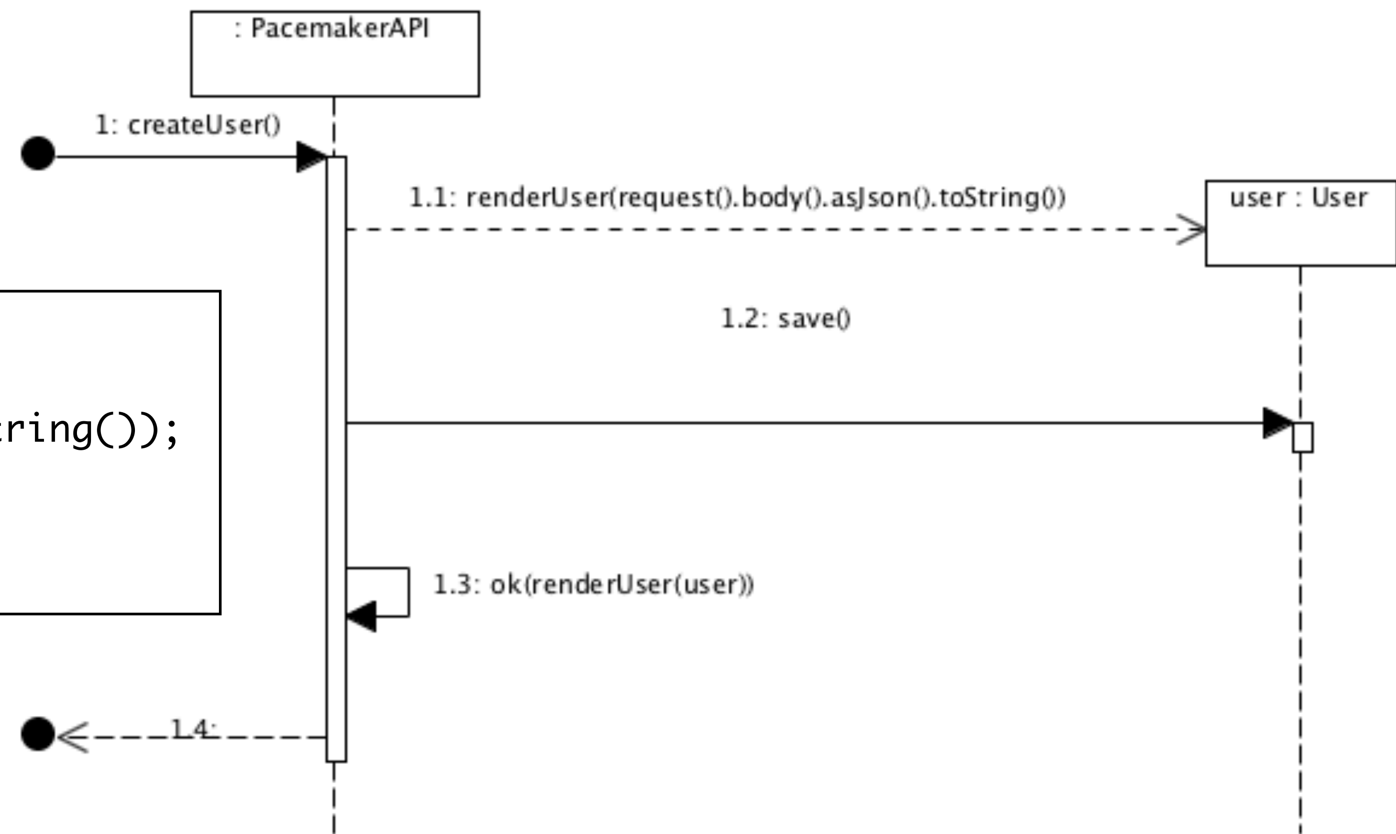


# API Namespace

| | |
|---|---|
| Retrieve / Delete all users | `GET /api/users`<br>`DELETE /api/users` |
| Create a User | `POST /api/users` |
| Retrieve / Delete / Update specific user | `GET /api/users/:id`<br>`DELETE /api/users/:id`<br>`PUT /api/users/:id` |

| | |
|---|---|
| Retrieve all activities for a user | `GET /api/users/:userId/activities` |
| Create an activity for a user | `POST /api/users/:userId/activities` |
| Retrieve / Delete / Update specific activity | `GET /api/users/:userId/activities/:activityId`<br>`DELETE /api/users/:userId/activities/:activityId`<br>`PUT /api/users/:userId/activities/:activityId` |

# createUser - Sequence Diagram



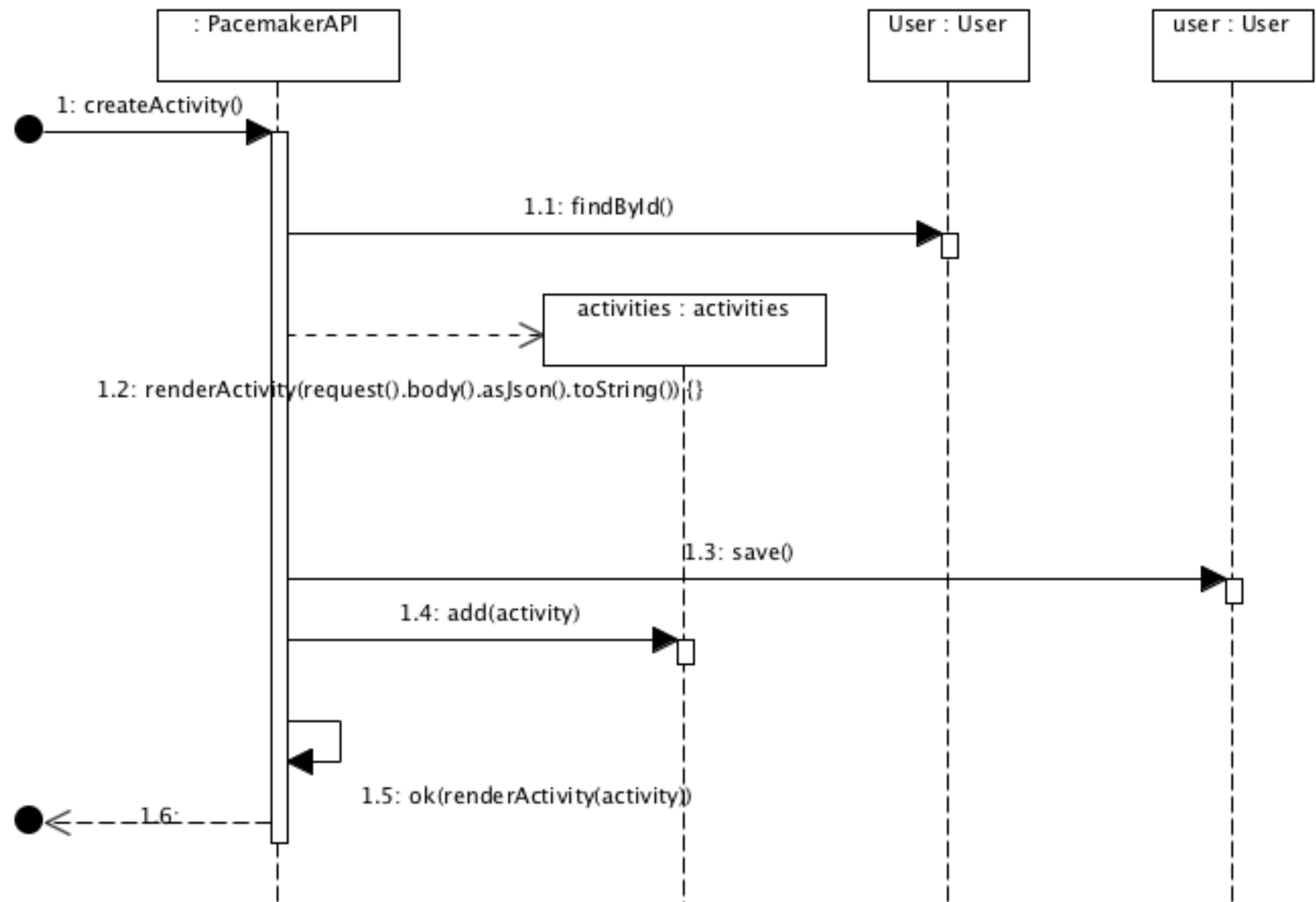

```
public static Result createUser()
{
  User user = renderUser(request().body().asJson().toString());
  user.save();
  return ok(renderUser(user));
}
```

# JSON

- All responses delivered as JSON objects via this parser

```
public class JsonParser
{
  private static JSONSerializer  usersSerializer     = new JSONSerializer().exclude("class");
  private static JSONSerializer  userSerializer      = new JSONSerializer().exclude("class").include("activities");
  private static JSONSerializer  activitySerializer = new JSONSerializer().exclude("class");

  public static User renderUser(String json)
  {
    return new JSONDeserializer<User>().deserialize(json, User.class);
  }

  public static String renderUser(Object obj)
  {
    return userSerializer.serialize(obj);
  }

  public static String renderUsers(Object obj)
  {
   return  usersSerializer.serialize(obj);
  }

  public static List<User> renderUsers(String json)
  {
    return new JSONDeserializer<ArrayList<User>>().use("values", User.class).deserialize(json);
  }

  public static Activity renderActivity(String json)
  {
    Activity activity = new JSONDeserializer<Activity>().deserialize(json,   Activity.class);
    return activity;
  }

  public static String renderActivity(Object obj)
  {
    return activitySerializer.serialize(obj);
  }

  public static  List<Activity> renderActivities (String json)
  {
    return new JSONDeserializer<ArrayList<Activity>>().use("values", Activity.class).deserialize(json);
  }
}
```

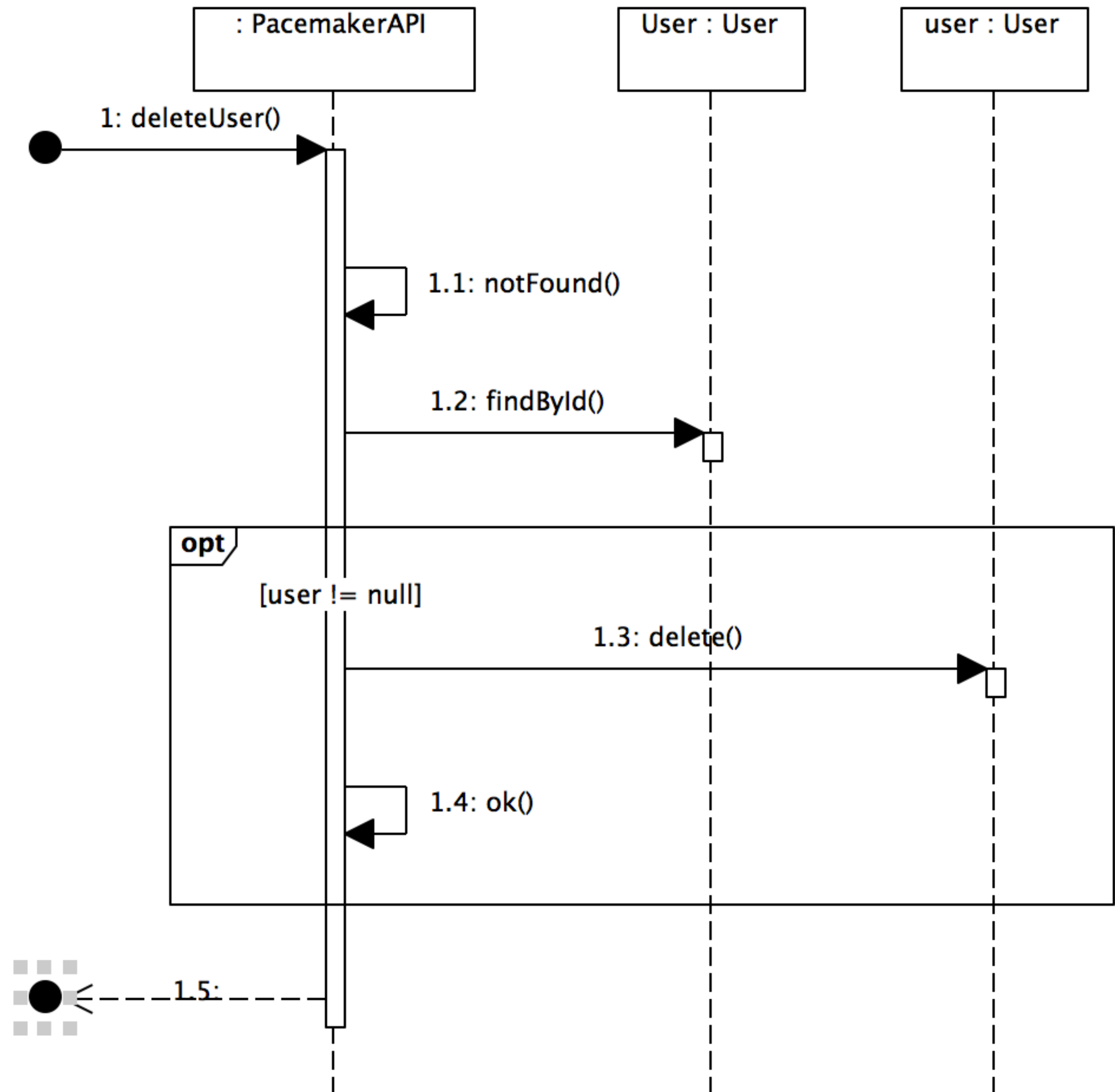
# createActivity Sequence Diagram

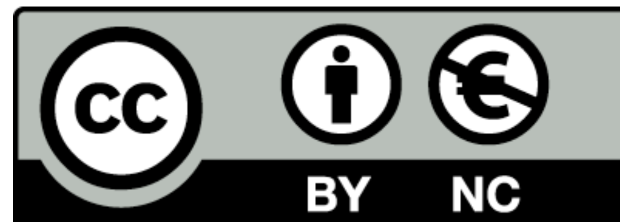



```java
public static Result createActivity (Long userId)
{
  User     user     = User.findById(userId);
  Activity activity = renderActivity(request().body().
                               asJson().toString());

  user.activities.add(activity);
  user.save();

  return ok(renderActivity(activity));
}
```

# deleteUser Sequence Diagram

```
public static Result deleteUser(Long id)
{
  Result result = notFound();
  User user = User.findById(id);
  if (user != null)
  {
    user.delete();
    result = ok();
  }
  return result;
}
```






Except where otherwise noted, this content is licensed under a Creative Commons Attribution-NonCommercial 3.0 License.

For more information, please see http://creativecommons.org/licenses/by-nc/3.0/

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE