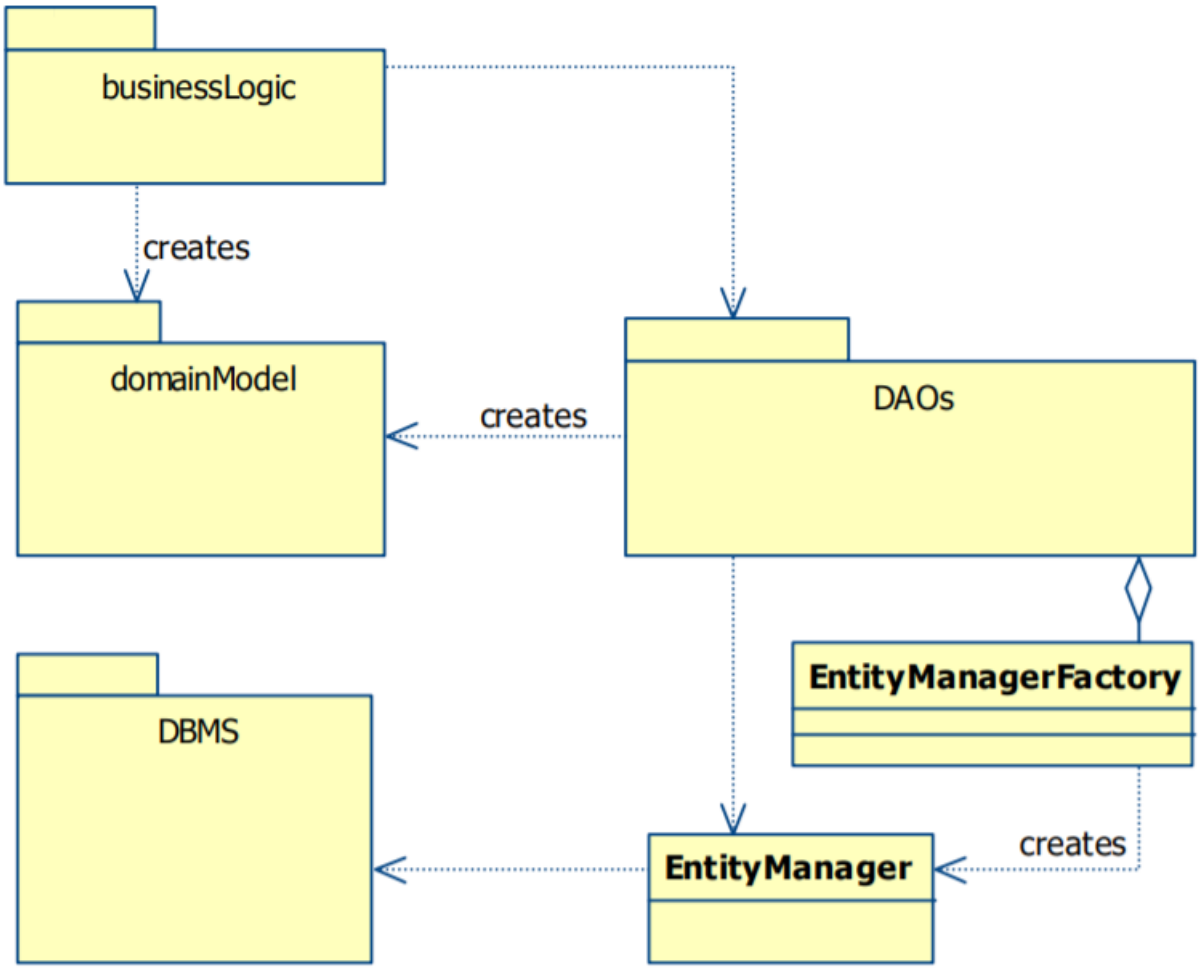
Progetto JPA (Hibernate) con database relazioni (MySql e PostgreSQL)

JPA, acronimo di Java Persistence API è una specifica Java per la gestione dei dati tra oggetti Java e database relazionali.

Hibernate è un framework che implementa questa specifica e offre delle API che permettono di accedere, persistere e gestire i dati in un database relazionale. In questo progetto, verrà usato il database MySql come database.

L'architettura utilizzata sarà la seguente:



Nel domainModel ci saranno le entità (Entity) che rispecchieranno gli oggetti che verranno persististi nel Database.

I DAOs (Data Access Object) sono classi Java che rappresentano un'interfaccia per la gestione degli oggetti persistiti nel database e permettono di isolare l'accesso ad una tabella tramite query.

La Business logic conterrà la logica dell'applicazione, ovvero tutte quelle operazioni che l'utente intende compiere.

L’EntityManager è l’oggetto Java che viene messo a disposizione dalle API JPA per comunicare a basso libello con il DBMS, ovvero il database.

L’EntityManagerFactory è l’oggetto Java che mette in vita l’ EntityManager.

Database Relazionale (MySql)

- Configurazione Maven

Innanzitutto è necessario creare un progetto Maven ed inserire tra le dipendenze `hibernate-core` e `mysql-connector-java` e `hibernate-jpa-2.1-api`

```
            <!-- Definition of JPA APIs intended for use in developing
Hibernate JPA
                  implementation -->
            <dependency>
                  <groupId>org.hibernate.javax.persistence</groupId>
                  <artifactId>hibernate-jpa-2.1-api</artifactId>
                  <version>1.0.0.Final</version>
            </dependency>

            <!-- Hibernate's core ORM functionality -->
            <dependency>
                  <groupId>org.hibernate</groupId>
                  <artifactId>hibernate-core</artifactId>
                  <version>5.3.1.Final</version>
            </dependency>

            <!-- JDBC (Java DataBase Connectivity) for mySql -->
            <dependency>
                  <groupId>mysql</groupId>
                  <artifactId>mysql-connector-java</artifactId>
                  <version>5.1.46</version>
            </dependency>
```

- Container Docker

Per far funzionare tutto è necessario utilizzare un container Docker per poter creare il database Mysql e poter effettuare tutti i test necessari.
Creiamo quindi un file docker-compose.yml con il seguente contenuto:

```
version: "2"

services:
  # Use root as user and a blank password
  mysql:
    image: mysql:5.7
    networks:
      - mysqlnet
    restart: always
    ports:
      - 3306:3306
    environment:
      - MYSQL_DATABASE=timekeep
      - MYSQL_ALLOW_EMPTY_PASSWORD=yes
```

```
networks:
  mysqlnet:
    driver: bridge
```

- Prima Entity

Creiamo nel package la nostra prima Entity, ovvero una classe Java che rappresenterà i dati che saranno permanentemente memorizzati in un database relazionale.

Saranno presenti i campi Id che rappresenterà un identificativo unico per l'oggetto, un titolo e una descrizione.

```
public class Project {

	private Long id;
	private String title;
	private String description;

	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getTitle() {
		return description;
	}
	public void setTitle(String title) {
		this.title = title;
	}
	public String getDescription() {
		return description;
	}
	public void setDescription(String description) {
		this.description = description;
	}
}
```

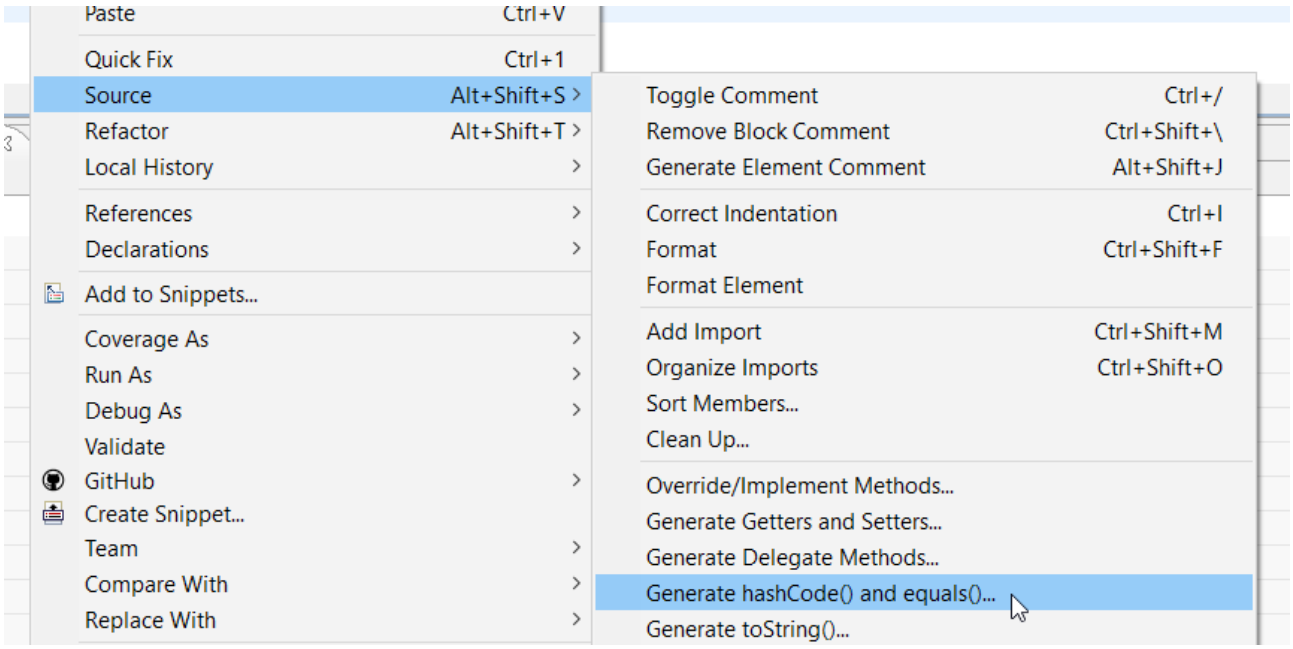
Aggiungiamo adesso le annotazioni `@Entity` e `@Id` e `@GeneratedValue` per farla diventare un'entità e fare in modo che venga generata una chiave primaria, Id in questo caso, che possa identificare ogni singolo Progetto.

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Project {

	private Long id;
	private String title;
	private String description;

	@Id
	@GeneratedValue
	public Long getId() {
```

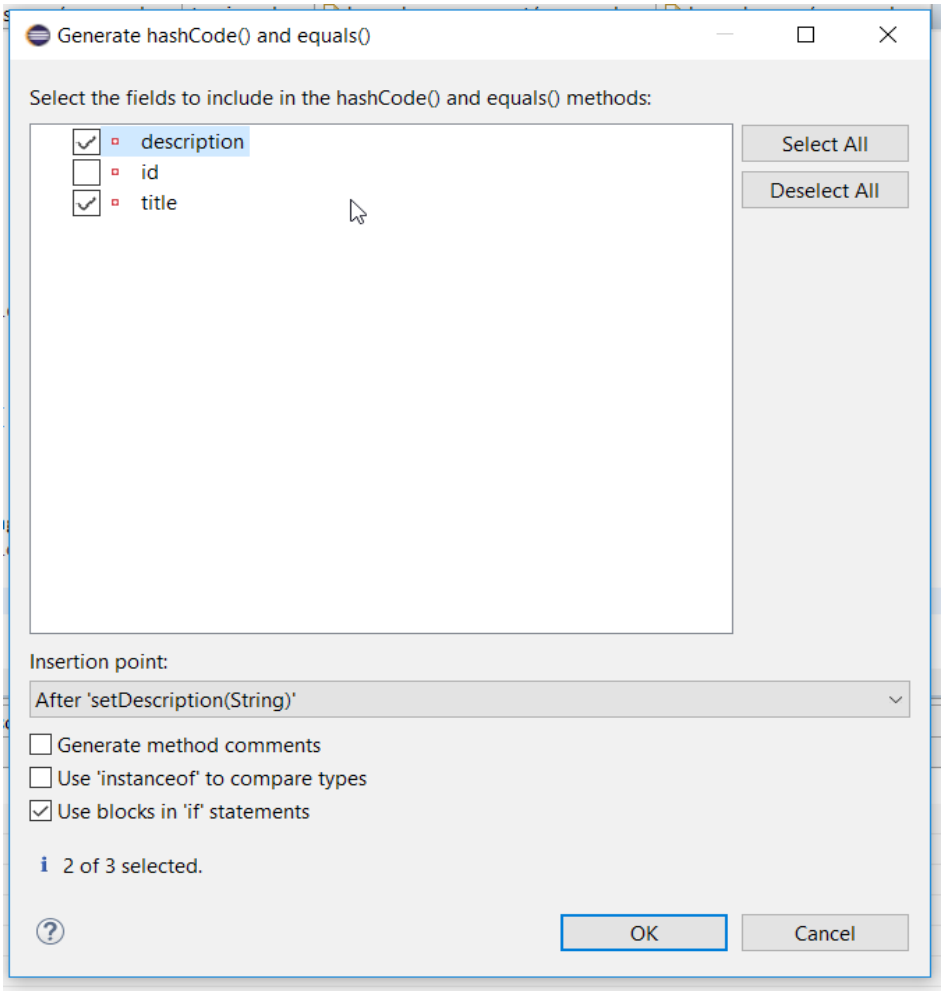
```
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getTitle() {
		return title;
	}
	public void setTitle(String title) {
		this.title = title;
	}
	public String getDescription() {
		return description;
	}
	public void setDescription(String description) {
		this.description = description;
	}
}
```

Facciamo anche l'override del metodo equals e hashcode che ci permetterà di controllare se due Progetti sono uguali.

Andiamo quindi a cliccare tasto destro sul codice, Source e successivamente Generate hashCode() and equals()…



e successivamente andiamo a selezionare tutto tranne il campo Id.
Utilizziamo quella che viene chiamata Business key equality, ovvero controlliamo ogni singolo campo che ha una semantica nel modello, quindi non la chiave primaria Id.



Questo sarà il codice che verrà generato:

```
@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((title == null) ? 0 : title.hashCode());
		return result;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Project other = (Project) obj;
		if (title == null) {
			if (other.title != null)
				return false;
		} else if (!title.equals(other.title)) {
			return false;
		}
		return true;
	}
```

- Configurazione Persistence.xml

Adesso è necessario creare una cartella chiamata META-INF in src/main/resources/ e successivamente creare il file persistence.xml nella cartella META-INF.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
      xmlns="http://xmlns.jcp.org/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

<persistence-unit name="mysql-pu"
      transaction-type="RESOURCE_LOCAL">
      <!-- The persistence-unit name must to be UNIQUE -->
      <!-- The value RESOURCE_LOCAL to replace the default value (JTA) allows
            you to manage connection with database manually -->

      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <!-- The provider specifies the JPA implementation that you want to use
            (Hibernate) -->

      <class>com.garritano.keepchronos.model.Project</class>
      <!-- Specify names of managed persistable classes -->

      <properties>
            <property name="hibernate.dialect"
                  value="org.hibernate.dialect.MySQL5InnoDBDialect" />
      <!-- The dialect specifies the language of your DB (MySQL 5 InnoDB) -->

            <property name="hibernate.hbm2ddl.auto" value="create" />
      <!-- Possible values for hbm2ddl:
validate: validate the schema, makes no changes to the database.
update: update the schema.
create: creates the schema, destroying previous data.
create-drop: drop the schema when the SessionFactory is closed explicitly,
typically when the application is stopped. -->

            <property name="hibernate.connection.url"
                  value="jdbc:mysql://localhost:3306/timekeep?
createDatabaseIfNotExist=true" />
            <!-- the address with port of your mySql service, followed by the
name of DB that is created if not exist -->

            <property name="hibernate.connection.username" value="root" />
            <property name="hibernate.connection.password" value="" />
            <!-- username and password of your mySql connection -->

            <property name="hibernate.show_sql" value="false" />
            <!-- show_sql=true permits to view queries in console -->

            <property name="hibernate.format_sql" value="false" />
            <!-- format_sql=true permits to view formatted queries in console
-->

</properties>
```

```
</persistence-unit>
</persistence>
```

Questo è un file di configurazione che permette a Hibernate di configurare una serie di parametri che sono commentati per capire meglio il funzionamento.

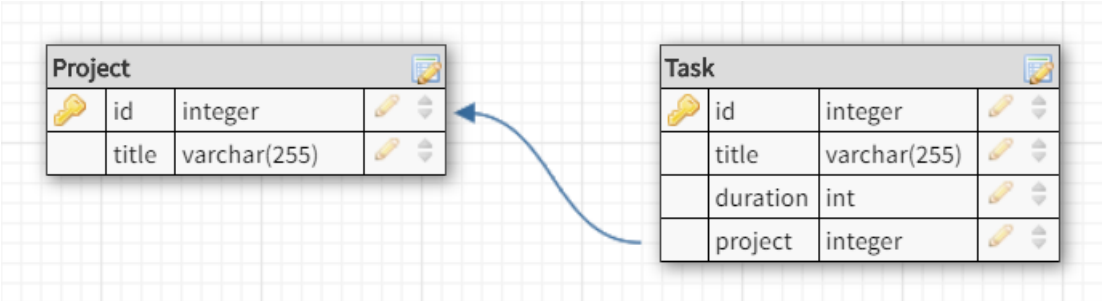
In particolare è necessario dare un nome alla persistence-unit. In questo caso ho inserito “mysql-pu”. È necessario anche fornire una lista delle classi che saranno persistite, inserendo il nome delle classi tra i tag <class>*</class>, il dialect che indica con quale database saranno persistiti i dati (altri dialetti sono disponibili qui: https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/dialect/package-summary.html), l’url per la connessione al Database e le credenziali di accesso.
Nel valore della proprierà `hibernate.connection.url` è stato aggiunta la dicitura `?createDatabaseIfNotExist=true` che, come dice il nome, crea il database se non presente.

Per motivi di testing, ho inserito come proprietà `hibernate.hbm2ddl.auto` il valore `create` che distrugge eventuali dati presenti sul database ogni volta che il programma si avvia.
In fase di produzione è necessario utilizzare un differente valore, come ad esempio `update`.

Ovviamente è necessario cambiare i dati dei nomi delle classi da persistere, l’url del database a cui connettersi e le credenziali di accesso, se diverse da quanto indicato.

- Creazione Task Entity

Andiamo adesso a creare una seconda entità in modo che venga mappata sul database nel modo riportato in figura:



La prima cosa da fare è inserire nei file persistence.xml il nome della classe tra i tag <class></class>, come è stato fatto per la classe Project.

È necessario quindi che la tabella Task abbia una foreign key. Jpa ci permette di fare questo creando un attributo Project sulla classe Task e andando ad annotare l’attributo Project come @ManyToOne. L’annotazione @ManyToOne ci permette di specificare a JPA che deve esserci una colonna di tipo foreign key sulla tabella Task.

Creiamo quindi la classe Task:

```
@Entity
public class Task{

	private Long id;
	private String title;
	private Project project;
	private int duration;
```

```
	@Id
	@GeneratedValue
	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getTitle() {
		return title;
	}
	public void setTitle(String title) {
		this.title = title;
	}

	@ManyToOne
	public Project getProject() {
		return project;
	}

	public void setProject(Project project) {
		this.project = project;
	}

	public int getDuration() {
		return duration;
	}

	public void setDuration(int duration) {
		this.duration = duration;
	}
}
```

Aggiungiamo tra i tag <class></class> nei file persistence.xml la classe Task.

Dato che molte parti del codice di Task sono uguali, in particolare il metodo equals, gli attributi id e title e i relativi getter e setter, è possibile utilizzare una classe astratta che rende il codice più pulito. Per JPA andremo ad utilizzare l'annotazione `@MappedSuperclass` per poter mappature gli attributi ereditati in modo correto, senza utilizzare una tabella separata.

Per una corretta comparazione degli oggetti di tipo Task è necessario, anche qui, utilizzare la Business key equality e testare il tutto.

Questo è il risultato delle 3 classi: `BasicEntity` entità base che contiene gli attributi e i metodi in comune, `Project` senza alcun metodo o attributo e `Task` con attributi duration e Progetto associato (e i relativi getter/setter)

```
@MappedSuperclass
public abstract class BasicEntity {

	private Long id;
	private String title;

	@Id
	@GeneratedValue
```

```
	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getTitle() {
		return title;
	}
	public void setTitle(String title) {
		this.title = title;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((title == null) ? 0 : title.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		BasicEntity other = (BasicEntity) obj;
		if (title == null) {
			if (other.title != null)
				return false;
		} else if (!title.equals(other.title)) {
			return false;
		}
		return true;
	}
}

@Entity
public class Project extends BasicEntity {

}

@Entity
public class Task extends BasicEntity{

	private Project project;
	private int duration;

	@ManyToOne
	public Project getProject() {
		return project;
	}
```

```
	public void setProject(Project project) {
		this.project = project;
	}

	public int getDuration() {
		return duration;
	}

	public void setDuration(int duration) {
		this.duration = duration;
	}
}
```

- Business Logic (ProjectStatistic)

Creiamo adesso la business logic, ovvero la parte logica applicativa che rende operativa l'applicazione. Per semplicità, verrà implementata solo una piccola logica che riguarda una piccola statistica sui progetti e i task assegnati.

Utilizzando la strategia TDD, andiamo a creare una classe di test chiamata ProjectStatisticTest.

Ricordiamoci di configurare il plugin `maven-failsafe-plugin`.

Facciamo in modo che nella classe ProjectStatistic ci sia un metodo che conti quanti progetti sono persistiti nel database ma, dato che dobbiamo creare test di unità è necessario fare un mock della classe che si occuperà di gestire la comunicazione con il database ovvero il Dao. In particolare la classe projectDao.

Andiamo a creare come attributi della classe di test i seguenti:

```
	private ProjectStatistic statistic;
	private ProjectDaoInterface projectDaoInterface;
	private List<Project> projects;
```

Questo perché questi oggetti verranno riutilizzati per ogni test.

Di volta in volta, utilizziamo il QuickFix di Eclipse per creare le classi o i metodi non ancora dichiarati. In questo caso ci sarà la necessità di creare la classe `ProjectStatistic` e l'interfaccia `ProjectDaoInterface`.

Andiamo ad effettuare il setup della classe di test mockando l'interfaccia ProjectDaoInterface. E andiamo a configurare il comportamento della chiamata projectDaoInterface.getAll() che, una volta implementata, dovrà restituirce tutti i Projects persistiti sul database.

```
	@Before
	public void setUp() throws Exception {
		projects = new ArrayList<Project>();
		projectDaoInterface = mock(ProjectDaoInterface.class);
		when(projectDaoInterface.getAll()).thenReturn(projects);
		statistic = new ProjectStatistic(projectDaoInterface);
	}
```

E andiamo quindi a testare, utilizzando il TDD, il comportamento su zero, e successivamente su uno o più progetti.

Il risultato sarà il seguente:

```
private void assertNumberOfProject(int expected) {
	int numberOfProject = statistic.getNumberOfProject();
	assertEquals(expected, numberOfProject);
}

@Test
public void testNoProject() {
	assertNumberOfProject(0);
}

@Test
public void testSingleProject() {
	projects.add(new Project());
	assertNumberOfProject(1);
}

@Test
public void testMultipleProjects() {
	projects.add(new Project());
	projects.add(new Project());
	assertNumberOfProject(2);
}
```

Sviluppando così un metodo sulla classe ProjectStatistic simile a questo:

```
public int getNumberOfProject() {
	List<Project> projects = projectDaoInterface.getAll();

	return projects.size();
}
```

Continuiamo con il test di un metodo che ci restituirà la durata totale di un progetto. È necessario fare quindi il mock del metodo che si occuperà di prendere i task associati ad un progetto.

Aggiungiamo nel metodo setUp() il seguente:

```
tasks = new ArrayList<Task>();


when(projectDaoInterface.getTasks(any(Project.class))).thenReturn(tasks);
```

Utilizziamo nuovamente una funzione per evitare duplicazione di codice:

```
private void assertTotalDuration(Project project, int expected) {
	int averageDuration = statistic.getTotalDuration(project);
	assertEquals(expected, averageDuration);
}
```

Ed andiamo a testare il metodo per zero, uno o più task associati ad un progetto:

```
@Test
public void testGetTotalDurationGivenProjectWithoutTask() {
	Project tempProject = new Project();

	assertTotalDuration(tempProject, 0);
```

```
}

@Test
public void testGetTotalDurationGivenProjectWithOneTask() {
	Project tempProject = new Project();

	Task tempTask1 = new Task();
	tempTask1.setDuration(30);

	tasks.add(tempTask1);

	assertTotalDuration(tempProject, 30);
}

@Test
public void testGetTotalDurationGivenProject() {
	Project tempProject = new Project();
	Task tempTask1 = new Task();
	tempTask1.setDuration(30);

	Task tempTask2 = new Task();
	tempTask2.setDuration(20);

	tasks.add(tempTask1);
	tasks.add(tempTask2);

	assertTotalDuration(tempProject, 50);
}
```

Dovremmo sviluppare un metodo sulla classe ProjectStatistic simile a questo:

```
public int getTotalDuration(Project tempProject) {
	List<Task> tasks = projectDaoInterface.getTasks(tempProject);
	int sum = 0;
	for (Task task : tasks) {
		sum += task.getDuration();
	}
	return sum;
}
```

- Business Logic (TaskStatistic)

La stessa cosa è possibile farla con una statistica riguardante i task. Creiamo quindi la classe di test `TaskStatisticTest`.

Implementiamo il metodo setUp() per effettuare il mock del metodo getAll() che sarà presente nella classe TaskStatistic.

```
private TaskStatistic statistic;
private TaskDaoInterface taskDaoInterface;
private List<Task> tasks;

@Before
public void setUp() throws Exception {
	tasks = new ArrayList<Task>();
	taskDaoInterface = mock(TaskDaoInterface.class);
```

```
        when(taskDaoInterface.getAll()).thenReturn(tasks);

        statistic = new TaskStatistic(taskDaoInterface);
    }
```

Adesso la presenza di zero, uno o più task.

```
    private void assertNumberOfTask(int expected) {
        int numberOfTask = statistic.getNumberOfTask();
        assertEquals(expected, numberOfTask);
    }

    @Test
    public void testNoTask() {
        assertNumberOfTask(0);
    }

    @Test
    public void testSingleTask() {
        tasks.add(new Task());
        assertNumberOfTask(1);
    }

    @Test
    public void testMultipleTasks() {
        tasks.add(new Task());
        tasks.add(new Task());
        assertNumberOfTask(2);
    }
```

Testando il metodo getNumberOfTask() con il seguente sviluppo:

```
    public int getNumberOfTask() {
        List<Task> tasks = taskDaoInterface.getAll();

        return tasks.size();
    }
```

Andiamo a sviluppare un metodo che esegua la somma delle durate dei task.

```
    private void assertTotalDuration(int expected) {
        int durationTotal = statistic.getTotalDuration();
        assertEquals(expected, durationTotal);
    }

    @Test
    public void testSumDurationNoTask() {
        assertTotalDuration(0);
    }

    @Test
    public void testSumDurationOneTask() {
        Task tempTask = new Task();
        tempTask.setDuration(20);
        tasks.add(tempTask);

        assertTotalDuration(20);
```

```
}

@Test
public void testSumDurationMultipleTasks() {
	Task tempTask1 = new Task();
	tempTask1.setDuration(20);
	tasks.add(tempTask1);

	Task tempTask2 = new Task();
	tempTask2.setDuration(30);
	tasks.add(tempTask2);

	assertTotalDuration(50);
}
```

Il risultato del metodo sarà il seguente:

```
public int getTotalDuration() {
	List<Task> tasks = taskDaoInterface.getAll();
	int sum = 0;
	for (Task task : tasks) {
		sum += task.getDuration();
	}
	return sum;
}
```

Per rendere più corposa la statistica, inseriamo un metodo che effettua la media dei tempi di Task.

```
private void assertAverageDuration(int expected) {
	int averageDurarion = statistic.getAverageDuration();
	assertEquals(expected, averageDurarion);
}

@Test
public void testAverageDurarionNoTask() {
	assertAverageDuration(0);
}

@Test
public void testAverageDurarionOneTask() {
	Task tempTask = new Task();
	tempTask.setDuration(20);
	tasks.add(tempTask);

	assertAverageDuration(20);
}

@Test
public void testAverageDurarionMultipleTasks() {
	Task tempTask1 = new Task();
	tempTask1.setDuration(20);
	tasks.add(tempTask1);

	Task tempTask2 = new Task();
	tempTask2.setDuration(30);
	tasks.add(tempTask2);
```

```
		assertAverageDuration(25);
	}
```

Questo sarà il risultato del medoto da sviluppare:

```
	public int getAverageDuration() {
		int numberOfTask = taskDaoInterface.getAll().size();
		if (numberOfTask != 0) {
			return getTotalDuration() / numberOfTask;
		}
		return 0;
	}
```

- Creazione Dao di Project e testing

Adesso creiamo il Dao per gestire la tabella Project utilizzando il TDD.

Creiamo quindi la classe `ProjectDaoTest`, ed creiamo un metodo setUp() che ci permetterà di riutilizzare il codice.

```
	private ProjectDao projectDao;
	private EntityManager entityManager;
	TypedQuery<Project> query;

	@SuppressWarnings("unchecked")
	@Before
	public void setUp() throws Exception {
		entityManager = mock(EntityManager.class);
		projectDao = new ProjectDao(entityManager);
		query = (TypedQuery<Project>) mock(TypedQuery.class);
		when(entityManager.createQuery(anyString(),
eq(Project.class))).thenReturn(query);
	}
```

Con il quick fix di Eclipse, creiamo la classe `ProjectDao`, facciamo in modo che estenda l'interfaccia `ProjectDaoInterface` e facciamo in modo che venga creato il seguente costruttore:

```
protected EntityManager entityManager;

public ProjectDao(EntityManager entityManager) {
	this.entityManager = entityManager;
}
```

Il risultato dello sviluppo sarà il seguente:

```
public class ProjectDao implements ProjectDaoInterface {

	private EntityManager entityManager;

	public ProjectDao(EntityManager entityManager) {
		this.entityManager = entityManager;
	}
}
```

Andiamo adesso a creare una sorta di interfaccia che possa rendere trasparente il layer DBMS, ma che possa farci comunicare con la tabella Project. Creiamo quindi i metodi save (per salvare un oggetto Project sulla tabella), getAll (per avere tutti gli oggetti di tipo Project della tabella), findById (per ricevere in output un oggetto di tipo Project, dato il suo identificativo), update (per

aggiornare lo stato di un oggetto Project già persistito, o crearne uno nuovo in caso non esista sul database).

Andiamo ad effettuare il test per il metodo save().

```
	@Test
	public void testSave() {
		EntityTransaction transaction = mock(EntityTransaction.class);
		when(entityManager.getTransaction()).thenReturn(transaction);

		projectDao.save(new Project());

		verify(entityManager).persist(any(Project.class));
		verify(transaction).begin();
		verify(transaction).commit();
	}
```

Aiutandoci con i quick fix, implementiamo il metodo save() ricordandoci di inizializzare e committare la transazione.

```
	public void save(Project project) {
		entityManager.getTransaction().begin();
		entityManager.persist(project);
		entityManager.getTransaction().commit();
	}
```

In parallelo andiamo a creare la classe di test `ProjectDaoIntegrationTest` per testare i metodi sviluppati, ricordandoci per ogni test di cancellare le tabelle Project e Task e cancellare le risorse.

Adesso è necessario effettuare un integration test ed è quindi necessario quindi inserire i plugin `maven-surefire-plugin` nel file pom.xml.

```
public class ProjectDaoIntegrationTest {

	private static final String PERSISTENCE_UNIT_NAME = "mysql-pu";
	private static EntityManagerFactory entityManagerFactory;
	private EntityManager entityManager;
	private ProjectDao projectDao;

	private Project project1;
	private Project project2;

	@BeforeClass
	public static void setUpClass() {
		entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
	}

	@Before
	public void setUp() throws Exception {
		entityManager = entityManagerFactory.createEntityManager();

		// make sure to drop the Project table for testing
		entityManager.getTransaction().begin();
		entityManager.createNativeQuery("delete from
Project").executeUpdate();
```

```
            entityManager.getTransaction().commit();

            projectDao = new ProjectDao(entityManager);

            project1 = new Project();
            project1.setTitle("First project");
            project1.setDescription("This is my first project, hi!");

            project2 = new Project();
            project2.setTitle("Second project");
            project2.setDescription("This is my second project, wow!");
      }

      @After
      public void tearDown() {
            project1 = null;
            project2 = null;
            entityManager.close();
      }

      @AfterClass
      public static void tearDownClass() {
            entityManagerFactory.close();
      }

}
```

Ed andiamo a testare il metodo save() della classe ProjectDao.

```
      @Test
      public void testSave() {
            projectDao.save(project1);

      // Clear Hibernate's cache to make sure data is retrieved from the store
            entityManager.clear();

            assertEquals(project1, entityManager.createQuery("from Project
where id =:id", Project.class)
                        .setParameter("id",
project1.getId()).getSingleResult());
      }
```

Ogni volta che si testa il contenuto del database, è importante cancellare la cache di Hibernate con `entityManager.clear();` altrimenti si rischia di testare il contenuto della cache di Hibernate.

Adesso procediamo con lo sviluppo del metodo getAll(). Utilizzando il TDD, andiamo a creare i test nei 3 casi: nel caso la funzione non restituisca nessun Project, ne restituisca uno e nel caso ne resituisca 2:

```
@Test
      public void testGetAllNoProjects() {
            List<Project> pList = new ArrayList<Project>();
            when(query.getResultList()).thenReturn(pList);

            List<Project> returnedProjects = projectDao.getAll();
```

```
		assertEquals(0, returnedProjects.size());
	}

	@Test
	public void testgetAllOneProject() {
		List<Project> pList = new ArrayList<Project>();
		pList.add(new Project());

		when(query.getResultList()).thenReturn(pList);

		List<Project> returnedProjects = projectDao.getAll();

		assertEquals(1, returnedProjects.size());
	}

	@Test
	public void testgetAllMoreThanOneProjects() {
		List<Project> pList = new ArrayList<Project>();
		pList.add(new Project());
		pList.add(new Project());
		when(query.getResultList()).thenReturn(pList);

		List<Project> returnedProjects = projectDao.getAll();

		assertEquals(2, returnedProjects.size());
	}
```

Andiamo così a implementare il metodo della classe `ProjectDao`.

```
public List<Project> getAll() {
		return entityManager.createQuery("select p from Project p",
Project.class).getResultList();
```

E andiamo a testarlo anche sulla rispettiva classe di integration test:

```
	@Test
	public void testEmptyGetAll() {
		assertTrue(projectDao.getAll().size() == 0);
	}

	@Test
	public void testOneGetAll() {
		projectDao.save(project1);

		// Clear Hibernate's cache to make sure data is retrieved from the
store
		entityManager.clear();

		assertEquals(project1, projectDao.getAll().get(0));
		assertTrue(projectDao.getAll().size() == 1);
	}

	@Test
	public void testMultipleGetAll() {
		projectDao.save(project1);
		projectDao.save(project2);
```

```
		// Clear Hibernate’s cache to make sure data is retrieved from the
store
		entityManager.clear();

		assertTrue(projectDao.getAll().size() == 2);
	}
```

Adesso creiamo il metodo findById per ricercare un progetto dato il suo identificativo:
Prima gli Unit test nei casi non trovi nessun risultato o oppure lo trovi:

```
@Test
	public void testFindByIdNoProject() {
		when(entityManager.find(eq(Project.class), anyLong())).thenReturn(null);

		Project returnedProject = projectDao.findById((long) 0);

		assertNull(returnedProject);
	}

	@Test
	public void testFindByIdOneProject() {
		Project tempProject = new Project();
		when(entityManager.find(eq(Project.class),
anyLong())).thenReturn(tempProject);

		Project returnedProject = projectDao.findById((long) 0);

		assertEquals(returnedProject, tempProject);
	}
```

Sviluppiamo l’implementazione

```
	public Project findById(Long id) {
		return entityManager.find(Project.class, id);
	}
```

Ed effettuiamo i due integration test:

```
	@Test
	public void testEmptyFindbyId() {
		assertNull(projectDao.findById((long) 34214342));
	}

	@Test
	public void testNotEmptyFindbyId() {
		projectDao.save(project1);

		// Clear Hibernate’s cache to make sure data is retrieved from the
store
		entityManager.clear();

		assertEquals(project1, projectDao.findById(project1.getId()));
	}
```

Come ultimo passo, definiamo il metodo getTask che ci permetterà di avere tutti i task associati ad un progetto.
Per fare questo è stato usato un metodo usiliario per effettuare il mock della Query.

```
	private TypedQuery<Task> mockQuery() {
		TypedQuery<Task> queryTask = (TypedQuery<Task>)
mock(TypedQuery.class);
		when(entityManager.createQuery(anyString(),
eq(Task.class))).thenReturn(queryTask);

		TypedQuery<Task> queryTaskParameter = (TypedQuery<Task>)
mock(TypedQuery.class);
		when(queryTask.setParameter(eq("project_id"),
anyObject())).thenReturn(queryTaskParameter);
		return queryTaskParameter;
	}
```

Ed ho effettuato due Unit test: con zero, uno o più task.

```
	@Test
	public void testGetTaskNoTask() {
		TypedQuery<Task> queryTaskParameter = mockQuery();

		List<Task> tList = new ArrayList<Task>();
		when(queryTaskParameter.getResultList()).thenReturn(tList);

		List<Task> taskList = projectDao.getTasks(new Project());

		assertEquals(0, taskList.size());
	}

	@Test
	public void testGetTaskOneTask() {
		TypedQuery<Task> queryTaskParameter = mockQuery();

		List<Task> tList = new ArrayList<Task>();
		tList.add(new Task());

		when(queryTaskParameter.getResultList()).thenReturn(tList);

		List<Task> taskList = projectDao.getTasks(new Project());

		assertEquals(1, taskList.size());
	}

	@Test
	public void testGetTaskMoreThanOneTask() {
		TypedQuery<Task> queryTaskParameter = mockQuery();

		List<Task> tList = new ArrayList<Task>();
		tList.add(new Task());
		tList.add(new Task());

		when(queryTaskParameter.getResultList()).thenReturn(tList);

		List<Task> taskList = projectDao.getTasks(new Project());
```

```
		assertEquals(2, taskList.size());
	}
```

Implementiamo il getTasks secondo la logica TDD

```
	public List<Task> getTasks(Project tempProject) {
		return entityManager.createQuery("select t from Task t where
project_id		= :project_id", Task.class).setParameter("project_id",
		tempProject.getId()).getResultList();
	}
```

Ed effettuiamo gli Integration test opportuni.

```
	@Test
		private void testGetTasksWithNoTask() {
		projectDao.save(project1);

	// Clear Hibernate’s cache to make sure data is retrieved from the store
		entityManager.clear();

		assertEquals(0, projectDao.getTasks(project1).size());
	}

	private void assertGetTasksWithOneTask() {
		projectDao.save(project1);

		Task tempTask = new Task();
		tempTask.setProject(project1);
		TaskDao taskDao = new TaskDao(entityManager);
		taskDao.save(tempTask);

	// Clear Hibernate’s cache to make sure data is retrieved from the store
		entityManager.clear();

		assertEquals(tempTask, projectDao.getTasks(project1).get(0));
	}
```

Con l’utilizzo dell’EntityManager, che ci alleggerisce notevolmente il lavoro, è possibile svolgere tutte le seguenti operazioni attraverso l’uso dei metodi persist(), find() e merge(), oltre che effettuare delle query sul database utilizzando quello che viene chiamato JPQL (Java Persistence Query Language).
Il linguaggio JPQL ci permette di effettuare delle query indipendentemente da quale layer DBMS stiamo utilizzando.
Esiste anche la possibilità di effettuare delle query native con il metodo createNativeQuery(), ma in questi casi si potrebbe perdere la flessibilità offerta da JPA di essere indipendente dal tipo di DBMS utilizzato.

- Creazione Dao di Task e testing

Nello stesso modo, è possibile creare il DAO di Task, inserendo anche un metodo per ritrovare il progetto associato ad un determinato Task.
Creiamo quindi la classe di test `TaskDaoTest` e implementiamo i test di Unità del metodo `getProjectByTaskId(Long id)`.

```
@Test
	public void testGetProjectByTaskIdNoProject() {
		when(entityManager.find(eq(Task.class),
anyLong())).thenReturn(null);

		Project tempProject = taskDao.getProjectByTaskId((long) 0);

		assertNull(tempProject);
	}

	@Test
	public void testGetProjectByTaskIdWithProject() {
		Task tempTask = mock(Task.class);
		when(entityManager.find(eq(Task.class),
anyLong())).thenReturn(tempTask);
		Project tempProject = new Project();
		when(tempTask.getProject()).thenReturn(tempProject);

		Project returnedProject = taskDao.getProjectByTaskId((long) 0);

		assertEquals(tempProject, returnedProject);
	}

public Project getProjectByTaskId(Long id) {
	Task tempTask = entityManager.find(Task.class, id);
	if (tempTask!=null) {
		return tempTask.getProject();
	}
	return null;
}
```

Nuovamente è possibile testare con dei test di integrazione il corretto funzionamento del Dao utilizzando la stessa filosofia di Project, ma andando a controllare anche il progetto associato.

Creiamo quindi un metodo per fare il setUp della classe test:

```
private static final String PERSISTENCE_UNIT_NAME = "mysql-pu";
	private static EntityManagerFactory entityManagerFactory;
	private EntityManager entityManager;
	private TaskDao taskDao;

	private Project project_another;
	private Task task1;
	private Task task2;

	@BeforeClass
	public static void setUpClass() {
		entityManagerFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
	}

	@Before
	public void setUp() throws Exception {
		entityManager = entityManagerFactory.createEntityManager();

		// make sure to drop the Task table for testing
```

```
            entityManager.getTransaction().begin();
            entityManager.createNativeQuery("delete from
Task").executeUpdate();
            entityManager.getTransaction().commit();

            project_another = new Project();
            project_another.setTitle("Another project");
            project_another.setDescription("Another exciting project!");
            entityManager.persist(project_another);

            task1 = new Task();
            task1.setTitle("First task");
            task1.setDescription("This is my first task, hi!");
            task1.setDuration(20);
            task1.setProject(project_another);

            task2 = new Task();
            task2.setTitle("Second task");
            task2.setDescription("This is my second task, wow!");
            task2.setDuration(30);
            task1.setProject(project_another);

            taskDao = new TaskDao(entityManager);
      }
```

E andiamo a liberare le risorse:

```
      @After
      public void tearDown() {
            task1 = null;
            task2 = null;
            entityManager.close();
      }

      @AfterClass
      public static void tearDownClass() {
            entityManagerFactory.close();
      }
```

Adesso andiamo a svolgere tutti i test dei metodi, come il test del metodo save:

```
      @Test
      public void testSave() {
            taskDao.save(task1);

      // Clear Hibernate's cache to make sure data is retrieved from the store
            entityManager.clear();

            assertEquals(task1, entityManager.createQuery("from Task where id
=:id", Task.class).setParameter("id", task1.getId()).getSingleResult());
      }
```

Il test del metodo getAll():

```
      @Test
      public void testEmptyGetAll() {
            assertTrue(taskDao.getAll().size() == 0);
```

```
}

@Test
public void testOneGetAll() {
	taskDao.save(task1);

// Clear Hibernate’s cache to make sure data is retrieved from the store
	entityManager.clear();

	assertEquals(task1, taskDao.getAll().get(0));
	assertTrue(taskDao.getAll().size() == 1);
}

@Test
public void testMultipleGetAll() {
	taskDao.save(task1);
	taskDao.save(task2);

// Clear Hibernate’s cache to make sure data is retrieved from the store
	entityManager.clear();

	assertTrue(taskDao.getAll().size() == 2);
}
```

Il test del metodo FindById():

```
@Test
public void testEmptyFindbyId() {
	assertNull(taskDao.findById((long) -1));
}

@Test
public void testNotEmptyFindbyId() {
	taskDao.save(task1);

// Clear Hibernate’s cache to make sure data is retrieved from the store
	entityManager.clear();

	assertEquals(task1, taskDao.findById(task1.getId()));
}
```

Infine, testiamo la possibilità di richiedere il Progetto dato un identificativo di Task non esistente.

```
@Test
public void testGetProjectByTaskIdWithNonExistingId() {
	assertNull(taskDao.getProjectByTaskId((long) -1));
}
```

E il get del Progetto dato un identificativo esistente:

```
@Test
public void testGetProjectByTaskIdWithExistingId() {
	taskDao.save(task1);

// Clear Hibernate’s cache to make sure data is retrieved from the store
```

```
            entityManager.clear();


assertEquals(task1.getProject(),taskDao.getProjectByTaskId(task1.getId()));
      }
```

Ulteriore database Relazionale (PostgreSQL)

Per aggiungere un secondo database relazionale è necessario inserire innanzitutto la dipendenza nel file pom.xml.

```
            <!-- JDBC (Java DataBase Connectivity) PostgreSQL -->
            <dependency>
                  <groupId>org.postgresql</groupId>
                  <artifactId>postgresql</artifactId>
                  <version>9.3-1102-jdbc41</version>
            </dependency>
```

Successivamente è necessario creare una seconda unità di persistenza nei file persistence.xml

```
<persistence-unit name="postgresql-pu" transaction-type="RESOURCE_LOCAL">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <!-- The provider specifies the JPA implementation that you want to use
            (Hibernate) -->

      <class>com.garritano.keepchronos.model.Project</class>
      <class>com.garritano.keepchronos.model.Task</class>

      <properties>
            <property name="javax.persistence.jdbc.driver"
                  value="org.postgresql.Driver" />
            <property name="javax.persistence.jdbc.url"
                  value="jdbc:postgresql://localhost:5432/timekeep?
createDatabaseIfNotExist=true" />
            <!-- the address with port of your postgresql service -->

            <property name="javax.persistence.jdbc.user"
                  value="postgres" />
            <property name="javax.persistence.jdbc.password" value="" />
            <!-- username and password of your postgresql connection -->

            <property name="hibernate.dialect"
                  value="org.hibernate.dialect.PostgreSQL95Dialect" />
            <!-- The dialect specifies the language of your DB (PostgreSQL) -->
            <property name="hibernate.hbm2ddl.auto" value="create" />
            <!-- Possible values for hbm2ddl:
validate: validate the schema, makes no changes to the database.
update: update the schema.
create:creates the schema, destroying previous data.
create-drop: drop theschema when the SessionFactory is closed explicitly,
typically when the application is stopped. -->

            <property name="hibernate.show_sql" value="false" />
            <!-- show_sql=true permits to view queries in console -->
```

```
            <property name="hibernate.format_sql" value="false" />
            <!-- format_sql=true permits to view formatted queries in console
-->
      </properties>

</persistence-unit>
```

Anche in questo caso è necessario inserire il quale implementazione di Jpa utilizzare (Hibernate), fare una lista delle classi da persistere, inserire quale driver jdbc usare, inserire l'url di accesso, le credenziali, il dialetto che verrà utilizzato, impostiamo la configurazione di Hibernate in modo che cancelli e ricrei sempre il database ogni volta che si avvia l'applicazione e mettiamo a false la configurazione che ci permette di vedere quali query vengono eseguite da Hibernate sul database.

Adesso aggiungiamo il container PostgreSQL sul file docker-compose.yml.

```
services:
  postgres:
    image: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: ""
      POSTGRES_DB: timekeep
      PGDATA: /data/postgres
    volumes:
       - /data/postgres:/data/postgres
    ports:
      - "5432:5432"
    networks:
      - postgres

networks:
  postgres:
    driver: bridge
```

Adesso effettuiamo il refactor di tutti i test di integrazione creati per poterli sfruttare anche con il secondo database relazionale.

Andiamo quindi a modificare la classe di test `ProjectDaoIntegrationTest`.

Facciamo in modo che i singoli test possano mettere in vita il proprio EntityManagerFactory, così da poter scegliere liberamente quale persistence-unit utilizzare. Questo vale anche per gli oggetti che prima erano creati nel setup del singolo test.

Andiamo quindi ad eliminare il metodo `setUpClass`, e riscriviamo il metodo `setUp` così da poter cancellare le tabelle ogni volta che si effettua un singolo test.

```
      private static EntityManagerFactory entityManagerFactory;
      private EntityManager entityManager;
      private ProjectDao projectDao;

      private Project project1;
      private Project project2;

      private void deleteTable(String persistenceUnit) {
```

```
            entityManagerFactory =
Persistence.createEntityManagerFactory(persistenceUnit);
            entityManager = entityManagerFactory.createEntityManager();
            // make sure to drop the Task table for testing
            entityManager.getTransaction().begin();
            entityManager.createNativeQuery("delete from
Task").executeUpdate();
            entityManager.getTransaction().commit();

            // make sure to drop the Project table for testing
            entityManager.getTransaction().begin();
            entityManager.createNativeQuery("delete from
Project").executeUpdate();
            entityManager.getTransaction().commit();
            entityManager.close();
            entityManagerFactory.close();
      }

      @Before
      public void setUp() throws Exception {
            deleteTable("mysql-pu");
            deleteTable("postgresql-pu");
      }
```

Andiamo adesso a riscrivere i singoli test. Utilizziamo sempre una funzione utility per evitare di duplicare il codice. Riscriviamo il test del metodo save() così:

```
private void assertSave(String persistenceUnit) {
            entityManagerFactory =
Persistence.createEntityManagerFactory(persistenceUnit);
            entityManager = entityManagerFactory.createEntityManager();
            projectDao = new ProjectDao(entityManager);
            project1 = new Project();
            project1.setTitle("First project");
            project1.setDescription("This is my first project, hi!");
            projectDao.save(project1);

      //Clear Hibernate's cache to make sure data is retrieved from the store
            entityManager.clear();

            assertEquals(project1, entityManager.createQuery("from Project
where id =:id", Project.class)
.setParameter("id",project1.getId()).getSingleResult());
            entityManager.close();
            entityManagerFactory.close();
      }

      @Test
      public void testSave() {
            assertSave("mysql-pu");
            assertSave("postgresql-pu");
      }
```

Con questo sviluppo è possibile utilizzare un EntityManager diverso per ogni database, inoltre tutti gli oggetti del model vengono ricreati per evitare che i valori Id generati automaticamente da Hibernate, possano creare qualche forma di dipendenza.

Eliminiamo anche i metodi che ci permettevano di elimianare le risorse, come `tearDown()` e `tearDownClass()`.

Adesso andiamo ad effettuare lo stesso procedimento per tutte i metodi della classe di test. Sia per ProjectDaoIntegrationTest, sia che per TaskDaoIntegrationTest.

Il risultato dovrà essere simile a quello che è linkato qui per ProjectDaoIntegrationTest (https://github.com/andryxxx/KeepChronos/blob/master/keepchronos/database/src/test/java/com/garritano/keepchronos/dao/ProjectDaoIntegrationTest.java) e qui per TaskDaoIntegrationTest (https://github.com/andryxxx/KeepChronos/blob/master/keepchronos/database/src/test/java/com/garritano/keepchronos/dao/TaskDaoIntegrationTest.java)

Progetto Maven Multimodulo

Per disaccoppiare maggiormente le varie parti del progetto, in particolare la parte di Unit Test e la la parte di Integration Test, ho effettuato una divisione in più moduli Maven.
Un modulo padre che conterrà le dipendenze comuni a tutti i moduli figlio, una parte che conterrà il model, la business logic e le interfacce per i Dao, infine una parte che conterrà i Dao.
Verrà poi aggiunto un ulteriore modulo per effettuare un report aggregato del plugin Jacoco.

Andiamo quindi a creare il progetto padre che abbia un packaging di tipo Pom e inseriamo le dipendenze `hibernate-jpa-2.1-api`, `hibernate-core`, e `junit` e configuriamo il plugin Jacoco.

Creiamo un modulo figlio con package Java e impostiamo come dipendenze solo `mockito-all`.
Inseriamo successivamente la parte di interfacce Dao, le classi nel model e le classi della Business logic.

Creiamo un ulteriore modulo figlio con package di tipo Java contenente come dipendenza i driver per comunicare con i database `mysql-connector-java` e `postgresql`. Inseriamo un'ulteriore dipendenza, ovvero l'artefatto del primo modulo figlio.
Nel mio caso sarà il seguente:

```
<dependency>
	<groupId>${project.groupId}</groupId>
	<artifactId>app</artifactId>
	<version>${project.version}</version>
</dependency>
```

È necessario che tra i tag `<artifactId>app</artifactId>` ci sia il nome dell'artefatto del primo modulo figlio.

Infine creiamo un ultimo modulo figlio con package di tipo Pom che dovrà contenere come dipendenze gli artefatti degli altri moduli figli con `<scope>compile</scope>` e per produrre un report jacoco aggregato, si dovrà configurare il plugin Jacoco inserendo come goal il seguente: `<goal>report-aggregate</goal>`

Il risultato dovrà essere simile a questo:

```
<build>
	<plugins>
		<plugin>
```

```
                  <groupId>org.jacoco</groupId>
                  <artifactId>jacoco-maven-plugin</artifactId>
                  <executions>
                        <execution>
                              <id>report-aggregate</id>
                              <phase>verify</phase>
                              <goals>
                                    <goal>report-aggregate</goal>
                              </goals>
                              <configuration>

<dataFileIncludes>**/*.exec</dataFileIncludes>
                              </configuration>
                        </execution>
                  </executions>
            </plugin>
      </plugins>
</build>
```