Question 1 :

Au départ, nous avons fortement sous-estimé les points par rapport aux histoires. La mise en place de la structure du projet avait pris plus de temps que prévu et donc tous les binômes n'ont pas pu travailler sur leurs tâches simultanément dès le début.

À partir de la deuxième itération, nous avons réévalué le temps que prendrait le développement des tâches. Nous avions alors une vision plus réaliste du temps de travail en prenant la moyenne de l'estimation du temps de travail de chaque membre.

À la fin de l'itération, nous avons remarqué que certaines tâches ont été surestimées et d'autres sous-estimées ce qui a créé une sorte d'équilibre dans la gestion du temps.

Le *Planning Poker* (que nous avons utilisé lors des itérations) était l'une des actions qui nous ont été les plus utiles durant le projet pour estimer le nombre de points d'une tâche. Ceci était utile car chacun estimait la valeur d'une tâche selon ses compétences et si nous devions développer un nouveau projet avec plusieurs différentes personnes, nous recommanderions d'utiliser le *Planning Poker* au départ du projet.

Avec l'expérience acquise durant le projet, nous avons une vision plus définie de certaines tâches qui reviendront dans de futurs projets et donc une idée plus claire du temps que prendraient ces tâches.

Pour terminer, si nous devions commencer un nouveau projet, nous essayerions de minimiser au plus possible le refactoring du code qui s'est avéré être une tâche chronophage.

Question 2 :

À travers l'usage du TDD, qui permet de s'assurer que nos fonctionnalités implémentées donnent de bons résultats en validant les tests, ce qui aide à garantir de la qualité finale du produit. Ensuite en ayant recours à différents design patterns qui s'assurent que notre code est bien structuré et facilite la maintenance et la lisibilité du projet. Nous avons aussi utilisé des plugins comme checkstyle, PMD et findBugs. Ceux-ci nous garantissent que le code écrit respecte les différentes normes et conventions de codage et apportent aussi une meilleure lisibilité au code.

Concernant la méthode XP, elle a bel et bien joué un rôle dans l'assurance de la qualité du code et du produit. En effet, vu que nous travaillons en binômes et qu'on intervertissait les binômes pour la même tâche, on avait ainsi plusieurs regards sur le projet, ce qui permettait alors à chacun d'apporter sa contribution en améliorant le code ou corrigeant des bugs rencontrés.

Question 3 :

Les différents updates sont une conséquence logique de l'implémentation de l'observateur-observé. En effet, lorsqu'un observé réalise une action, ce dernier notifie l'observateur et exécute la méthode "update" de l'observateur. Ne sachant quelle action est exécutée, il est nécessaire d'avoir un certain "repère" afin que l'observateur puisse savoir comment réagir. D'où l'utilisation des states dans notre cas.

Les states représentent les différents états que peut avoir un observé lorsqu'il notifie ses observateurs et ces derniers exécutent certaines actions selon cet état.

En effet, l'attribut state ne sert qu'aux méthodes "update" des observateurs. La solution actuelle fonctionne parfaitement, mais comme l'attribut "state" n'est pas utilisé dans le modèle, mais uniquement dans les méthodes update, ce ne serait pas nécessaire de stocker la variable en attribut.

Une autre solution serait d'implémenter notre propre interface Observable et d'y créer notre propre méthode "notifyObservers" qui prendrait en paramètres le "state" (au lieu de créer un attribut) et les objets qui sont actuellement passés.

Par exemple, au lieu d'avoir ce code :

```
public void addTagToProject(Project project, Tag tag)
    throws DatabaseException, ConnectionFailedException {
  this.tagHandler.addTagToProject(project, tag);
  setState(State.TAG_CREATED);
  notifyObservers(project);
}
```

Nous aurons ce code :

```
public void addTagToProject(Project project, Tag tag)
    throws DatabaseException, ConnectionFailedException {
  this.tagHandler.addTagToProject(project, tag);
  notifyObservers(project, State.TAG_CREATED);
}
```

Question 4 :

Le problème qui peut se poser avec les méthodes getTags, getTasks, getSubProjects, getCollaborators, …, c'est qu'on peut faire un traitement sur les objets retournés par les getters, depuis un contexte extérieur à la classe Project.

On pourrait par exemple faire appel à getCollaborators() puis modifier la liste ou faire un traitement sur la liste, depuis l'extérieur de la classe Project. Alors que si on voulait ajouter un collaborateur depuis la classe Project, il ne suffirait pas d'update la liste des collaborateurs, mais il faudrait aussi ajouter le projet à la liste des projets de l'utilisateur en question. On remarque qu'en faisant cela, on violerait le principe d'encapsulation et qu'on crée des erreurs.

On pourrait régler le problème en retournant une unmodifiableList, de telle sorte à ce qu'on ne puisse plus modifier l'état interne d'un objet grâce aux getters. De plus il est plus clair pour un programmeur en voyant une unmodifiableList, que celle-ci n'est pas destinée à ajouter un collaborateur.

Une autre solution est d'effectuer une défensive copy dans les getters. De cette manière, la liste retournée possède des copies des objets ayant des références différentes dans la mémoire.

Les solutions proposées ne changeraient pas significativement notre code. Nous avons besoin du principe des getter au sein des contrôleurs pour pouvoir accéder aux éléments qui nous intéressent et pour pouvoir les afficher correctement dans la vue. Lors de l'affichage des projets et sous-projets par exemple, il faut pouvoir demander à un projet quels sont ses collaborateurs pour les afficher. Dans ces cas, nous n'aurions donc pas de problèmes vu qu'on ne modifie pas les listes.

Question 5 :

La responsabilité de la classe *User* est de stocker les informations concernant les projets de l'utilisateur connecté ainsi que ses données personnelles telles que le nom d'utilisateur, email, etc.

Pour la méthode findSubProject, son rôle permet de trouver si un sous-projet d'un projet parent contient le projet qu'on souhaite trouver. Il est vrai que l'emplacement de cette méthode pouvait être placée dans la classe *Project* avec bien évidemment les modifications qui vont avec, comme le montre le screen ci-dessous :

```java
/**
 * Finds a subproject in project's tree structure.
 *
 * @param projectToFind the project to find
 * @return the project if found, null otherwise
 */
public Project findSubProject(Project projectToFind) {
  for (Project subProject : this.getSubProjects()) {
    if (subProject.equals(projectToFind)) {
      return subProject;
    }
    return subProject.findSubProject(projectToFind);
  }
  return null;
}
```