Commande Git :

Working Area

* clone : clone a repository into a new directory.
* init : Create an empty Git Repository or reinitialize an existing one.

work on current change :

* add : Add file contents to the index.
* mv : Move or rename a file, a directory, or a symlink.
* reset : Reset current HEAD to the specified state.
* rm : Remove files from the working tree and from the index.

examine the History and state :

* bisect : Use binary search to find the commit that introduced a bug
* grep : Print lines matching a pattern
* log : show commit logs.
* show : show various types of objects.
* status : show the working tree status.

Grow, mark and tweak your History :

* branch : List, create, or delete branches.
* checkout : switch branches or restore working tree files.
* commit : Record changes to the repository.
* diff : show changes between commits, commit and working tree, etc.
* merge : Join two or more development histories together.
* rebase : Forward-port local commits to the updated upstream head
* tag : Create, list, delete or verify a tag object signed with GPG.

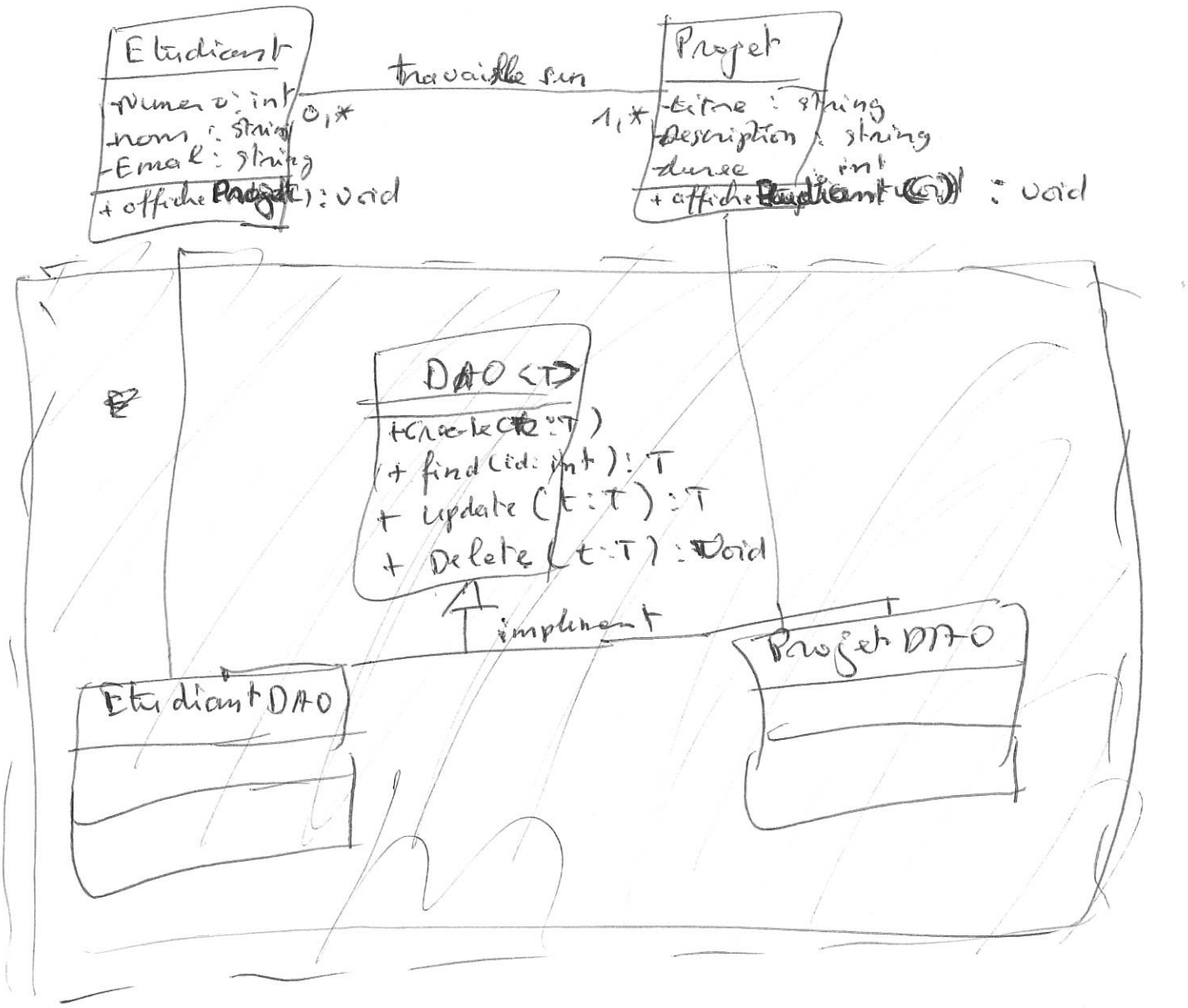
Collaborate :

* Fetch : Download objects and refs from another repository.
* Pull : Fetch from and integrate with another repository or local branch.
* Push : Update remote refs along with associated objects.

Projet :

* git status : voir pas encore enregistrer
* git add (nom ou lien) : ajouter au dossier GIT.
* git commit -m "Titre" : enregistrer.
* si pas co : git remote add origin "lien".
* git remote -v : voir dossier (origine)
* git push : Upload Projet

1. Diagramme UML



2.

```
public class Etudiant {

    private int numero;
    private String nom;
    private String email;

    private ArrayList<Projet>
        listProjet;

    public Etudiant (int numero, String nom, String email){
        this.numero = numero;
        this.nom = nom;
        this.email = email;
    }
```

~~public String~~

```
public int getNumero (){
    return this.numero;
}
// definition des getters de nom et email, ListProjet

public void setNumero (int numero) {
    this.numero = numero
}
// definition des setters de nom et email, ListProjet
```

~~public void~~

```
public String toString ( ) {
    return "numero" + this.numero +
           "nom" + this.nom +
           "Email" + this.Email ;
}
}
```

3. De même pour la classe projet.

4.

* Fonctionnement du pattern DAO

Il s'agit d'ajouter des objets ayant la responsabilité de l'accès au SGBD (opération de Create, Read, update et delete (CRUD))

On a une Interface DAO qui contient les signatures des methodes CRUD.

Pour chaque objet metier persistant on créé un objet DAO correspondant qui implimente l'interface DAO.

* Interrêt du pattern DAO

Il permet de separer le code de persistance du code metier. Ainsi l'application ne dependra plus du code de persistance qui est du code technique.

5. Classes DAO necessaires.

```
public class EtudiantDAO extends DAO<Etudiant>{

    @Override
    public Etudiant create(Etudiant obj) {}

    @Override
    public Etudiant find(int numero) {}

    @Override
    public Etudiant update(Etudiant obj) {}

    @Override
    public void delete(Etudiant obj) {}

}
```

```
public class ProjetDAO extends DAO<Projet> {

    @Override
    public Projet create(Projet obj) {}

    @Override
    public Projet find(String titre) {}

    @Override
    public Projet update(Projet obj) {}

    @Override
    public void delete(Projet obj) {}

}
```

6. JDBC : Java DataBase Connectivity est une spécification qui permet d'interfacer une application JAVA avec une SGBD.

Le driver : permet d'établir la communication avec un SGBD spécifique.

7.
```
@Override
public Etudiant create(Etudiant obj) {
    try {
        PreparedStatement prepare = connect.prepareStatement(
            "INSERT INTO Etudiant (numero, nom, email)
                VALUES (?, ?, ?)");
        prepare.setInt(1, obj.getNumero);
        prepare.setString(2, obj.getNom);
        prepare.setString(3, obj.getEmail);
        int result = prepare.executeUpdate();
        assert result == 1;
    } catch (SQLException e) { e.printStackTrace(); }
    return obj;
```

8.

```
@Override public Etudiant find (int numero) {
    Etudiant e = new Etudiant();
    try {
        PreparedStatement prepare = connect.prepareStatement(
          "select * from etudiant where numero = ? ");
        prepare.setInt(1, numero);
        ResultSet result = prepare.executeQuery();
        if (result.first()) {
            e = new Etudiant(
                result.getInt("numero"),
                result.getString("nom"),
                result.getString("email") );
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return e;
}
```

9.

```
Etudiant e1 = new Etudiant (1, "IP", "ip@yahoo.fr");
Etudiant e2 = new Etudiant (2, "IV", "iv@yahoo.fr");
Projet p = new Projet ("t1", "desc", 6 );

p.getListEtudiant().add (e1);
p.getListEtudiant().add (e2);
e1.getListProjet().add (p);
e2.getListProjet().add (p);
EtudiantDAO DAO<Etudiant> etudiantDao = new EtudiantDAO
                                        ();
    etudiantDao.create (e1);
    etudiantDao.create (e2);
```

MIN15114 - Programmation, GL, preuve
Examen (jan. 2016)

Stéphane Lopes Zoubida Kedad

Durée : 2h - Documents autorisés

Exercice 1 (Persistance avec JDBC et le pattern DAO)
Dans cet exercice, vous allez développer, avec JDBC et le pattern DAO, la couche de persistance d'une application de gestion de projets d'étudiants.

Un *étudiant* est identifié par son *numéro d'étudiant* et possède un *nom* et un *email*. Un *projet* possède un *titre*, une *description* et une *durée* (en mois). Un étudiant peut travailler sur plusieurs projets (au moins un) et chaque projet peut être attribué à plusieurs étudiants (zéro ou plus).

L'application doit permettre :
- pour chaque projet, d'afficher les étudiants impliqués,
- pour chaque étudiant, d'afficher les projets sur lesquels il travaille.

1. Donnez un diagramme de classes UML qui modélise cet énoncé (uniquement le domaine sans la couche de persistance).
2. Donnez l'implémentation Java de la classe `Etudiant`. L'affichage des caractéristiques de l'étudiant se fera avec la méthode `toString`.
3. Faites de même pour la classe `Projet`.
4. Expliquez le fonctionnement et l'intérêt du pattern DAO.

Dans la suite, on suppose l'existence de la classe abstraite `DAO<T>` vue en cours. En particulier, vous supposerez que la connexion au SGBD est déjà établie et que les tables sont présentes dans le SGBD.

5. Donnez le squelette (déclaration et signature des méthodes) des classes DAO nécessaires.
6. À quoi sert la spécification JDBC et quel est le rôle du driver ?
7. Donnez l'implémentation de la méthode `EtudiantDAO.create` qui rend persistant un étudiant.
8. Donnez l'implémentation de la méthode `EtudiantDAO.find` qui recherche un étudiant à partir de son identifiant.
9. Donnez l'extrait de code qui crée deux étudiants, les ajoute à un projet et rend les objets persistants.
10. Donnez l'extrait de code qui récupère les deux étudiants à partir de la BD et affiche leurs caractéristiques.
11. Expliquez le fonctionnement et l'intérêt du pattern FABRIQUE ABSTRAITE.
12. Donnez le code de la classe `DaoJdbcFactory` qui implémente le pattern FABRIQUE pour la création des DAO.
13. Donnez le code de la classe `DaoAbstractFactory` qui implémente le pattern FABRIQUE ABSTRAITE pour la création des DAO.
14. Quels changements faut il apporter au code qui utilise ces DAO ?
15. Donnez un diagramme de classes UML qui reprend l'ensemble des classes créées et leurs relations.

Exercice 2 (Persistance avec JPA)
Pour cet exercice, la persistance sera assurée par la spécification JPA, l'ORM Hibernate et le SGBD Derby. L'énoncé du problème est le même que pour l'exercice précédent (*étudiant* et *projet*). L'association entre *étudiant* et *projet* devra être bi directionnelle.

1. Expliquez le lien entre Derby, JDBC, Hibernate et JPA.
2. Donnez l'élément `persistence unit` du fichier XML `persistence.xml` dans le contexte de cet exercice.
3. Donnez l'implémentation de la classe `Etudiant` intégrant les annotations JPA.
4. Donnez l'implémentation de la classe `Projet` intégrant les annotations JPA.
5. Donnez l'extrait de code qui crée deux étudiants, les ajoute à un projet et rend les objets persistants.
6. Donnez l'extrait de code qui récupère les deux étudiants à partir de la BD et affiche leurs caractéristiques.
7. Affichez la liste des étudiants dont l'email se termine par `@ens.uvsq.fr` en interrogeant la BD avec JPQL.