[SOEN343] PROJECT MILESTONE 3

Code Smells & Refactoring: Cabra

Laura BERNHARDT – Guillaume FORTIN – Magalie PERON – Derek RINGUETTE – Nicolas TRICHEUX

Project Cabra

Laura BERNHARDT - #7118112
Guillaume FORTIN - #6325300
Magalie PERON - #7129173
Derek RINGUETTE - #1947834
Nicolas TRICHEUX - #7129491

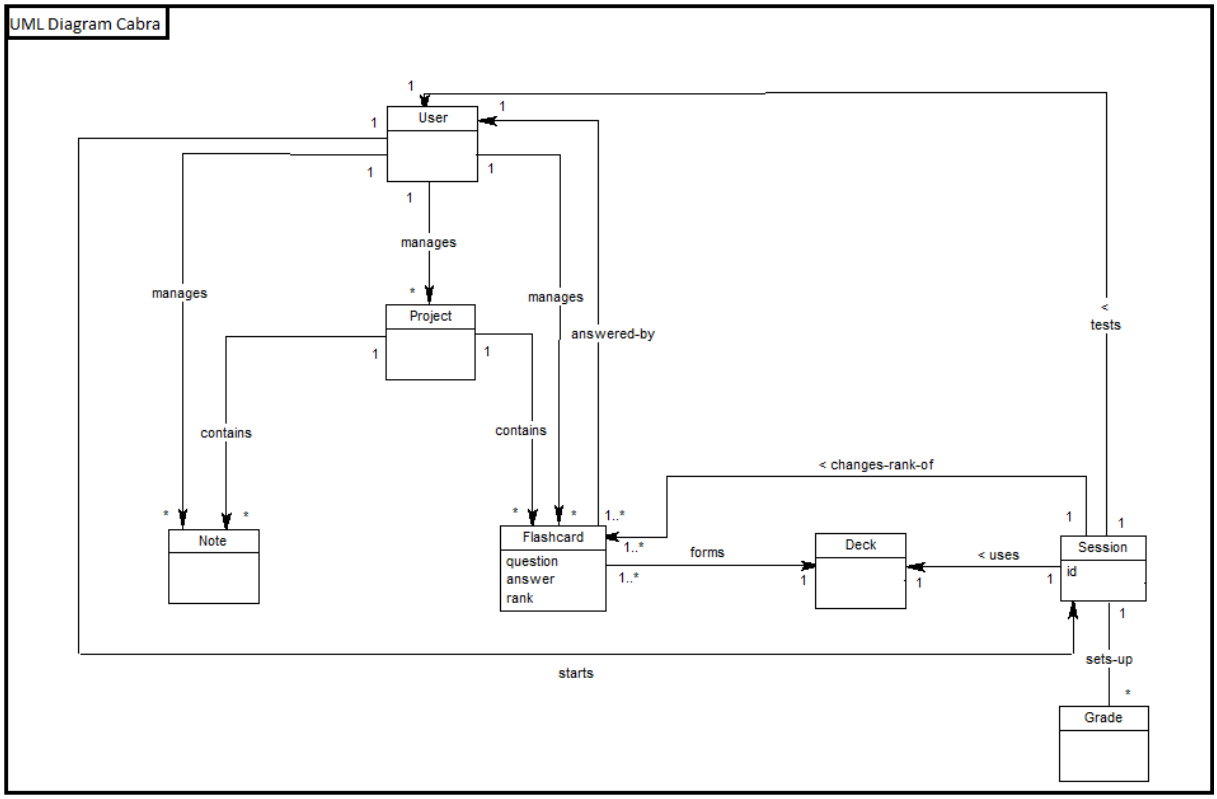
Project summary

Cabra is a free open source program meant to enhance your studying skills. It revolves around the concept of flashcards. These come in the format of a question and an answer or multiple answers, which allow you to classify how well you understand the subject (know it, sort of, don't know). The program is only limited by the amount of cards or projects you create, which can be categorize by lesson and further grouped into parent subjects. For each project (set of cards) the program shows you statistics on how well you know it and shows you your progression into learning the new material.

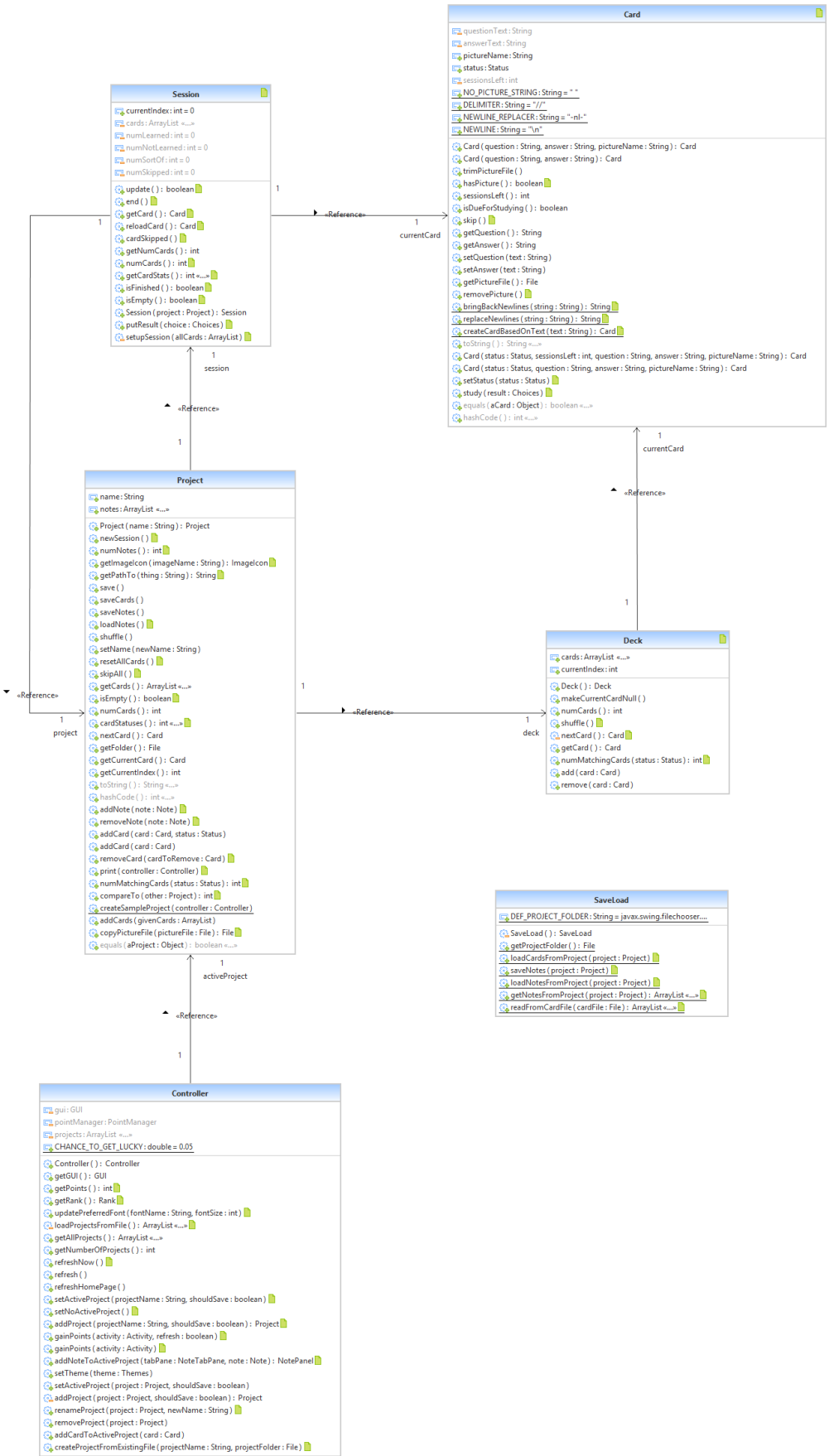
This is an interesting project because it was conceptualized and is currently being developed by a single high school student. The most recent version is very popular (several thousand downloads) and it is cross platform (multi-os, mobile and browser based). To some extent teachers can even make their own study cards based on the material they want the students to learn and share it with them using a simple passcode. Moreover, the source code is very organized in terms of comments. This way, the algorithms being implemented are easier to understand and could be a great help to us in the refactoring phase of the project.

Class diagram of actual system

Here is the conceptual diagram from Milestone 2:



This is the UML diagram of classes of interest (we focus on our domain: study). A pdf of this diagram is available in root folder.



The UML diagram shows us that the central class of the application's architecture is the Project class. This class contains methods to add and remove Notes as well as Flashcards to their respective collections. The Flashcard object, unlike the Notes object, contains an intermediary class (Deck) between itself and Project. Deck is a simple class which is instantiated immediately and stays instantiated as long as the project exists. A Deck acts in a similar manner as a collection object, meaning it supplies the methods related to adding and removing flashcards as well as shuffling and retrieving the next card during a study session. A Session, which is another class in the UML diagram, acts as a quiz in order to test the user on their knowledge of the flashcard material. During a study session, the user answers questions and specifies whether he got it right, wrong or sort of right. The Session class also contains the ability to skip the current flashcard. As we can see through the architectural diagram, all information regarding grades for a certain session is saved inside the Session object itself.

Next up, we have added the Controller and SaveLoad classes to our architectural focus. The Controller class acts as an intermediary between the view, which includes all the aspects related to the interface the user interact with, and the model, which encapsulates all the classes used to represent the data and concepts being manipulated. The vast majority of its methods contain calls to existing methods stored in either the Project class, or other classes related to GUI which we have scoped out of focus. The SaveLoad class supplies the application with the functions related to saving all progress made in regards to creating a project as well as the methods which load these previously saved states.

After comparing our diagram of conceptual classes to the UML diagram of actual relevant architecture, we can see that there is really not much of a difference between the two. The only major difference is that our conceptual architecture consisted of a Grade class. In our design, the Grade class would be instantiated by the Session class every time a new session was started in order to record and keep track of all the users' progress through a session as he or she is answering flashcards. As mentioned previously, the current system is designed to keep all of this information within the Session object. In our opinion, this information (information regarding how many flashcards were answered right, how many were answered wrong, how many were skipped, etc.) should be stored in a different class. We believe that the Session object should only focus on functionality associated with cycling through and retrieving flashcards. This would increase the cohesion of the Session object. We think that the current architecture design in respect to keeping track of grades can be greatly improved upon.

The second major difference between our conceptual diagram and the actual architecture is how the Deck object is utilized. In our conceptual architecture, the Deck class was directly used by the Session class. Its instantiation was for the sole reason of being used in conjunction with Session. In the actual design, the deck class is used as a collection of flashcards. Therefore, existence of a single flashcard object implies the existence of the Deck object. We think this is another aspect of the architecture which can be improved. A Deck of flashcards, when comparing it to its real-world counterpart, should contain persistent data from the point in time before a study session to the point in time when that study session is complete. We believe that the existence of another class, such as a Deck which is used with one specific paired Session would improve the application's design and make for a more coherent structure.

Reverse engineering tool

In order to help us have a good perspective of the Cabra architecture, we wanted to use a reverse engineering tool. First, we tried to find a tool for NetBeans but the market of NetBeans is very small compared to Eclipse market. So, finally, we used an Eclipse tool : UML Lab. It's a great util developed by Yatta Solutions which has lots of features. First of all, you can create your model and class diagrams with it, selecting the classes you want to have in the diagram. It shows you only these classes and links between them, if any. In the diagram itself, you are allowed (by a simple button click) to show parents, childs or any composition/aggregation links with other classes that the class has. Changing code in the project changes directly the diagram without regenerate it. Finally, one of the powerful tool that UML Lab has is refactoring tool. Indeed, you can refactor some code in the diagram itself and it changes directly the related code. We used this tool to do the class diagram. It allowed us to see first refactorings just looking for relation between the classes, like Inappropriate Intimacy between Project/Session for example.

Relations between 2 classes of interest

Session.java

```
public class Session extends Object{
  private Project project; //the project we're studying for

    public Session(Project project){
      this.project = project;
      project.setSession(this);
      cards = new ArrayList<Card>();
      setupSession(project.getCards());
  }

  public void end()
        {
      project.setSession(null);
  }
}
```

Project.java

```
public class Project implements Comparable<Project>{
  private Session session = null; //if a study session is going on, it's here


  public void newSession(){  // creates a new session  for this project
      do{
         setSession(new Session(this));
      }
```

```
    while(getSession().isEmpty());
  }
   public void addCard(Card card,Status status){
    card.setStatus(status);
    deck.add(card);
          //if the card has a picture, move the picture over here
    if(card.hasPicture()){
      File copiedFile = copyPictureFile(card.getPictureFile());
      card.setPictureName(copiedFile.getAbsolutePath());
      //and now trim the card's picture file... we won't need the full path any more
      card.trimPictureFile();
    }
    saveCards();
     if(session != null){  //since there's a new card, notify the session
      session.update();
    }
  }
}
```

Code smells & Refactoring

There are several types of code smells in Cabra project. Some of them are easy to detect and fix. Others are more complex.

First of all, there are some dead code in Card, Note, Project and Session classes. Both of them contains commented code, even whole functions (such as removeCard or decreaseCurrentIndex in Session class). This code smell can be fix simply in remove this code.

Another code smell which is easy to fix is Long Method. We can see examples of it in SaveLoad and Controller classes. In SaveLoad, loadCardsFromProject and getNotesFromProject are refactorable. In both of these functions, the developer wrote comments to explain the code. In order to improve the clarity of the code and to remove this code smells, we can use the Extract Method each time the developer put a comment. For example, in getNotesFromProject we can extract 4 methods : getFiles, readNote, manageIncompatibleKindOfNote, addNoteToNotesList. In the Controller class, almost all methods are too long but especially the constructor that is so long that the code is very unclear. The method to fix this is Extract Method too.

Other code smells are very interesting because they imply architecture design, coupling and cohesion. There is Inappropriate Intimacy between Project and Session. Indeed, the Project class has an instance of Session (set in SetSession which is called in the constructor of Session) and the session has an instance of the project (parameter of the constructor). However, the only reason of that is because the session wants the list of cards of the project to get the cards studied for this session. To reduce coupling and increase cohesion, instead of set the project in session's constructor and get the cards with getter in project, we can pass directly the deck which contains manage the list of the cards to the session constructor. When this is the end of the session, the StudyPanel calls end() function of current session. This function just sets the session instance to null in project. Instead, StudyPanel can call a function end in the Project

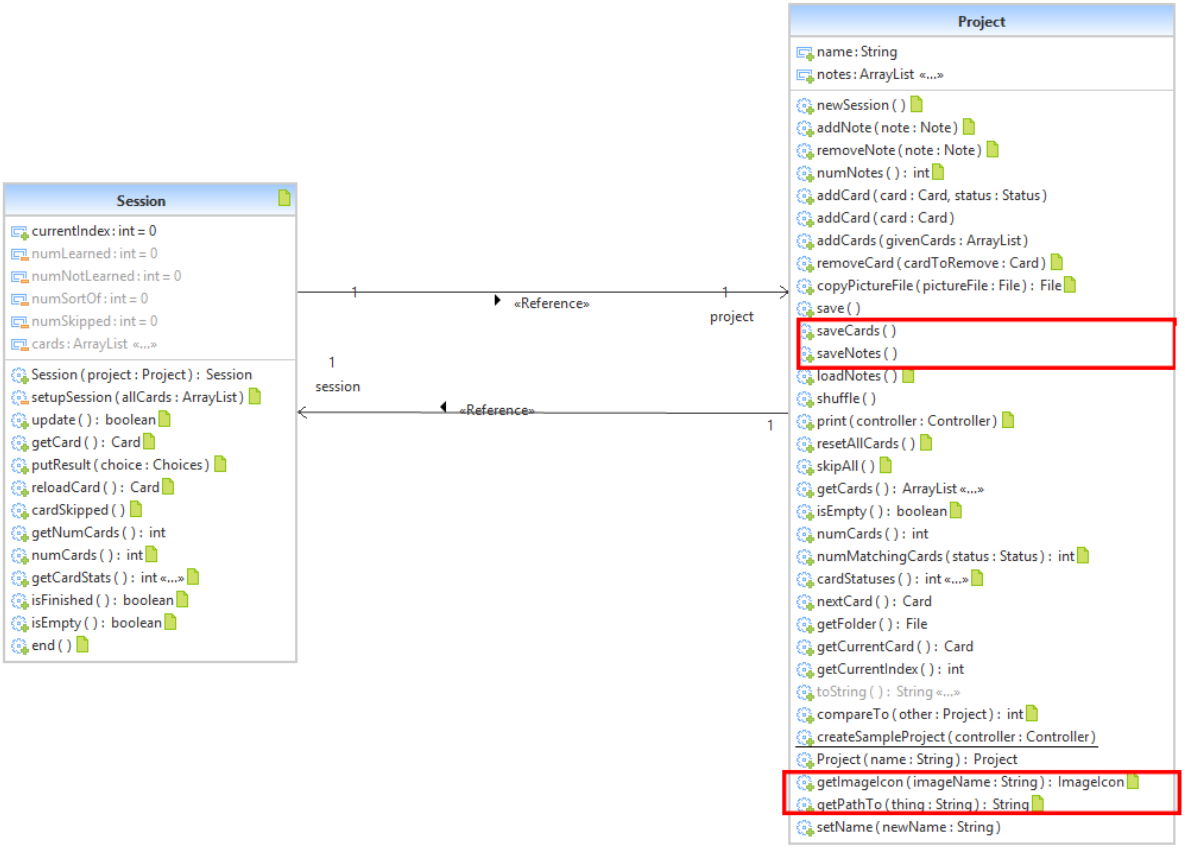
class that set directly instance of session to null. This way, the project has a session instance but the session doesn't deal with project anymore.

Some classes that we focus on have the Divergent Change code smells. Indeed, they have responsibilities that other classes should have. In order to increase cohesion, that each class have only one responsibility (that is the main concept of Object Oriented Programming). It concerns Card, Project and Controller classes. In Card class, they are functions that deal with picture: trimPictureFile, getPictureName, getPictureFile, setPictureName, removePicture. However, the Card has to deal only with question and answer. It can have a picture but all functions that get the picture from file and so on should not be in Card class but an util class like PictureManager, which will be called by Card. So, we can use the Extract Class to do the class PictureManager and Move Methods dealing with Picture into this class. ReplaceNewLines, BringBackNewLines and constant attributes of class Card shouldn't be there because it deals with string stuff. The code smells is also divergent change. Extract a class StringUtils for example and move these methods into this class will increase cohesion. All of these changes are fixing another code smell : large class. Indeed, Card class has too much responsibilities and is too large. Combining extract class and move methods refactorings should increase cohesion and increase reusability and clarity of the code.

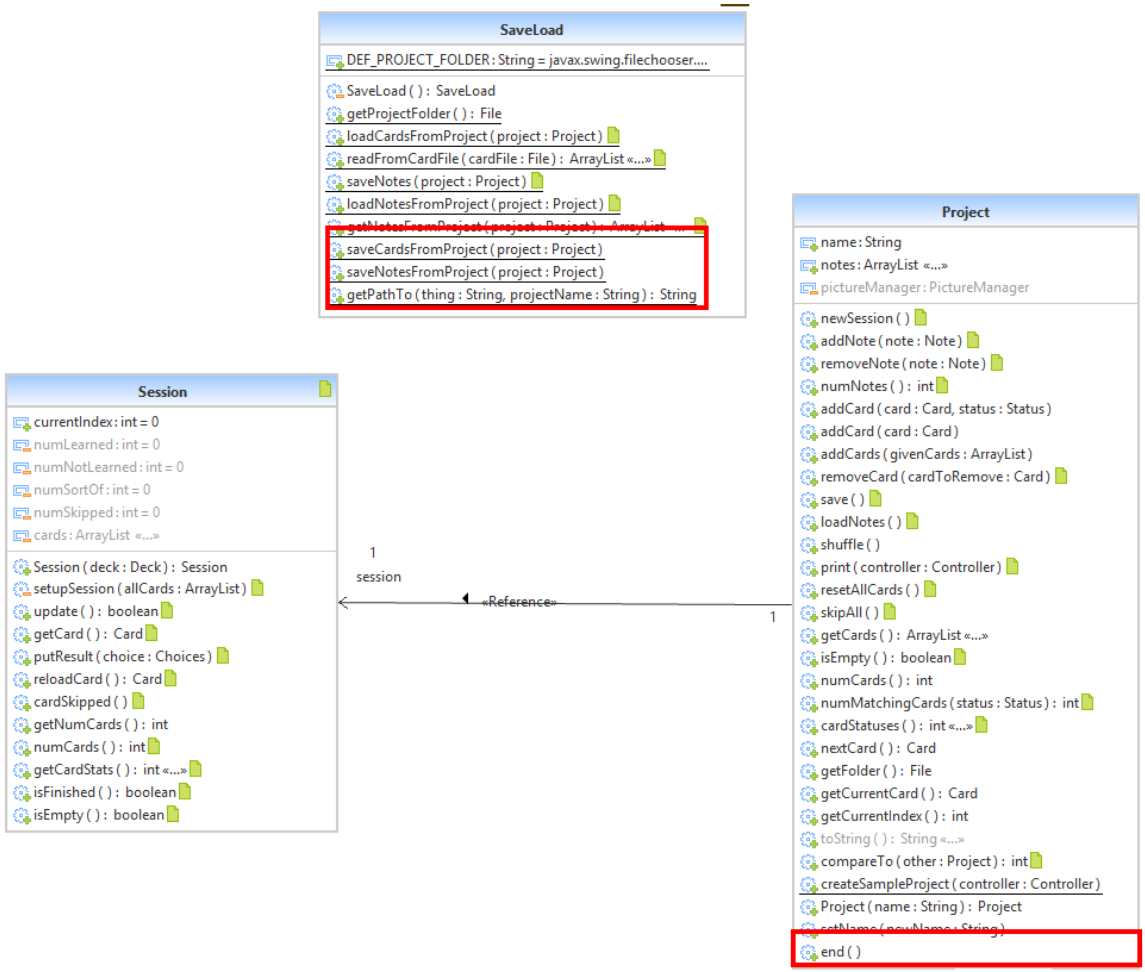
Divergent change exists in Project class too. The main purpose of this class is to manage cards of one project, containing in a deck and starts/ends sessions of study for this project. However, it contains functions dealing with picture (copyPictureFile), icon (getImageIcon), save/load (getPathTo, saveCards, saveNotes). copyPictureFile doesn't need Extract class refactoring because it can be moved in PictureManager class that we extract from Card. Likewise, getImageIcon can be moved in ImageManager and finally save/load functions should be moved in SaveLoad class, which is a class precisely managing the save and load of a project. All these changes are logical and increase cohesion of the Project class. All calls of these functions should be changed. Now, static class SaveLoad is called instead of Project and we pass in parameter to those functions the name of the project or the project itself when needed. With this architecture, it has only one responsibility and combining with previous refactorings fixing Inappropriate Intimacy and Dead Code, it fixes another code smell in this class : Large Class.

Controller is the class linking user interactions with the "real" study code. It's interesting because this is the class which call lots of functions of the classes we've seen previously. Maybe that's why there are some divergent changes in this class. Indeed, because it's the link between GUI and study algorithm, the developer did a Large Class and put in one place all code calling the algorithm according to events. However, in some cases such as addProject, removeProject and loadProjectsFromFile, it seems to be in the wrong place because of their purpose. For example, addProject and removeProject create and remove a directory for the project itself. However, it should be logical that it'd be in SaveLoad class (that deals with all events about saving/loading a project, find it in the files explorer, and so on). So, first we should move these methods into SaveLoad class. Then, we should rename the class SaveLoad because now it create and remove a class too. For example, SaveLoad can be renamed as…… . Finally, we can move loadProjectsFromFile that searches itself all projects created by user in the old class SaveLoad renamed ……. These refactorings increase cohesion of the Controller class but reduce its size (not a Large Class anymore). It removes too the feature envy with SaveLoad class because instead of calling Controller functions to call SaveLoad functions, all the related code is directly moved into the SaveLoad class.

There is an example of refactorings that we can do with just two classes : Session and Project. The class diagram below represents the classes before the refactorings.



We can see the Inappropriate Intimacy between those two classes and looking at the Project class, we can see that there are Large Class and Divergent change (methods' name) code smells too. If we do the refactorings suggested before, the diagram class becomes as below:



Code of one suggested refactoring

Suggested Refactoring: Using deck in the session.

The Deck class extends objects and is essentially an array list of Cards. Currently the deck is used by the project to maintain all of it's cards. When you time comes to study the project makes a session which then gets all the cards back from the project and stores them in an independent array list of Cards. Rather than doing this we would consider passing the whole deck during the session creation and use a local deck to manipulate the needed Cards of the project deck during the study session. This will give a more logical structure to the current code closer to our conceptual diagram and eliminate some duplicate code.

Code example:

Deck.java

```
public class Deck extends Object{
  private ArrayList<Card> cards; //flash cards of owner project
  private Card currentCard = null;
  private int currentIndex;
```

```
  public Card getCard(){
    if(numCards() == 0)
      return null; //no cards here
    currentCard = nextCard()
    return currentCard;
  }
  public int getCurrentIndex(){
    return currentIndex;
  }
  public int numCards(){
    return cards.size();
  }
...
```

Session.java

```
public class Session extends Object{
  ...
  private int currentIndex = 0; //how many cards we have studied
  private ArrayList<Card> cards; //cards we'll study
  private Card currentCard; //current card we're studying
  …
  public Session(Project project){
    …
    cards = new ArrayList<Card>();
    setupSession(project.getCards());
  }
  public Card getCard(){
    if(currentIndex >= numCards()){
      return null;
    }
    currentCard = cards.get(currentIndex);
    currentIndex++;
    return currentCard;
  }
  public int getCurrentIndex(){
    return currentIndex;
  }
  public int getNumCards(){
    return cards.size();   }
```