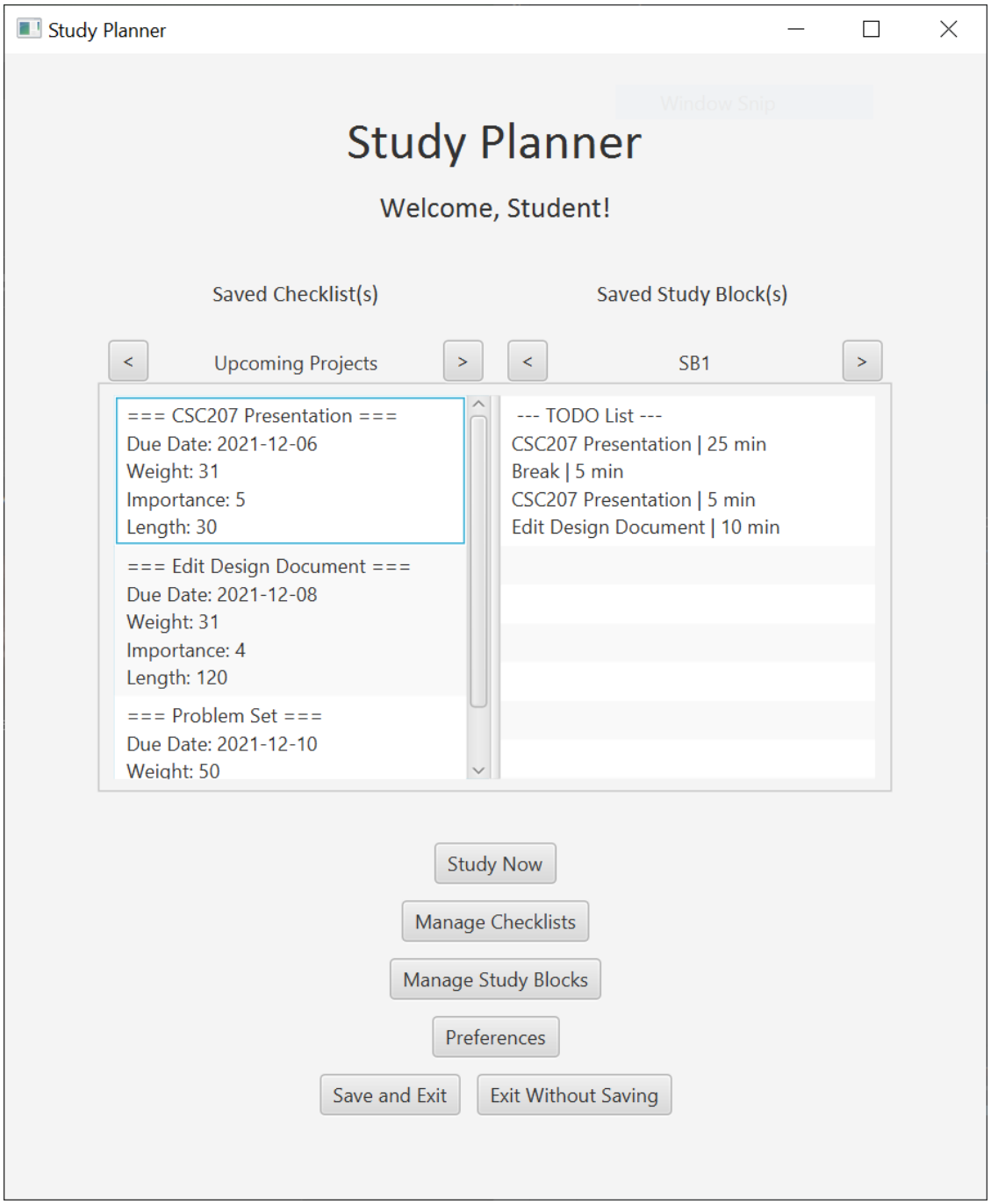
STUDY PLANNER



Group Dinosaurs | CSC207

SPECIFICATION

StudyPlanner allows a student to track various Checklists of Tasks, and to organize their active study and break time into study blocks. Students can create Tasks of different weights (what each Task is worth as a percentage of their grade), importance (a 5-point system of assigning priority), due dates, and lengths (time to complete). Students can also create one or multiple Checklists, which automatically sort Tasks according to the student's preferred sorting method. Tasks can be marked as complete or incomplete and once they are incomplete, they are moved from the Checklist to a list of completed Tasks.

Once the Student has added some Tasks, they can create a StudyBlock that will give them a detailed breakdown of how to organize their time while completing the Tasks. StudyBlocks can also be created from a convenient Study Now interface that allows the user to quickly create a new block from one or more Checklists. The StudyBlock assigns active time and break time until time runs out or all Tasks are completed according to a user-defined preference. Students can also save Checklists and Tasks to their device before exiting the app. For convenience, StudyBlocks are exportable in a common calendar format.

FUNCTIONALITY

HOW TO USE

To begin, run MainGUI located in /Controllers.

1. Create a Checklist. This can be done by clicking the "Manage Checklists" button on the main menu and then "Create New Checklist."
2. Add a Task. Cycle to the Checklist you want to edit by using the "<" and ">" buttons. Then click "Add Task" and input the relevant information.
3. Create a study block. Return to the main menu and then click on "Manage Study Blocks". From here, click on "Create New Study Block" and select which Checklist(s) you'd like to study with. Select how long you'd like to study for and how you'd like the study block to be sorted.
4. Study!

IMPROVEMENTS

In Phase 2 we improved the functionality of the program in a variety of ways. We added automatic, seamless loading of saved Checklists, Tasks, and StudyBlocks, rather than requiring the user to use buttons to select and load the files manually. We implemented the remaining buttons that had not yet been coded in phase 1 like the delete buttons for Checklists, Tasks and StudyBlocks. These buttons also delete the files from the directory, rather than keeping them there indefinitely. Our app can now change the way Tasks are sorted within a Checklist based on due date, importance, length or weight. We changed the way importance and weight is set, using sliders instead of having users enter a value.

LIMITATIONS

There are a few areas where we did not achieve our goals in functionality. The biggest failure was our inability to have the views update automatically when Tasks, Checklists, or StudyBlocks are added, deleted, or changed. The change is registering within the program, but the user must navigate or hit a button to update the ListView. We struggled for a long time, trying various ways of implementing this functionality since it is key to fulfilling our users' expectations of how a GUI should work, but we were mostly unable to achieve our goal. Curiously, it proved easy to implement in the instance of sorting existing Checklists by different priorities. Selecting different priorities from the drop-down menu immediately displays the correctly sorted list. It is likely that with some more time and JavaFX experience, we could implement this functionality, as it remains a high priority for the group.

MAJOR DESIGN DECISIONS

OVERVIEW

Leading up to Phase 1, our group rushed to implement a somewhat functional JavaFX GUI using our pre-existing code established in Phase 0. Although much of the functionality was implemented successfully, the design process was sloppy, and our program began to accumulate code smells (an example of technical debt discussed in lecture). Violations of clean architecture cropped up in our controllers package and the functionality was inconsiderate to a diverse range of users; there was substantial room for improvement. As a result, only a small portion of Phase 2 was dedicated to finishing up the functionality whereas the focus was to go back and clean up any poorly implemented features from Phase 1. Phase 2 major design decisions were made to abide by Clean Architecture and SOLID design principles, while also making the program more accessible for different users and any future programmers who might look at our code. In this section, we will cover some major design decisions which we agreed upon throughout the semester and provide a brief description of the thought processes behind them.

DATA ACCESS INTERFACE

The DataAccessInterface was implemented after Phase 1 to address the Clean Architecture violation between the Data class and all controllers. Before, we had a Data class which stored all of our local memory (Checklists, StudyBlocks, study methods etc.) and was instantiated when the program started. The controller methods would directly manipulate this outer layer class and did not demonstrate proper adherence to Clean Architecture and Dependency Inversion Principle. In order to address this, DataAccessInterface was implemented so the controllers could call methods of this interface instead of accessing the data directly (Figure 2.0). You can read more about this in the Clean Architecture and Dependency Inversion sections.

```
@FXML
protected void createNewChecklist(ActionEvent actionEvent) {
    Checklist newChecklist = new Checklist(checklistNameField.getText());
    Data.checklistList.add(newChecklist);
    Data.addToChecklistList(newChecklist);
```

Figure 3.0 Change from pull request #61 showing changes made after implementing the DataAccessInterface. Now, this method in MainGUIController manipulates the information in DataAccess through interface methods instead of directly calling upon the class. (https://github.com/CSC207-UofT/course-project-dinosaurs/pull/61/files)

SEAMLESS LOADING

Seamless loading was implemented to uphold two universal principles of design: simple and intuitive use and low physical effort. In Phase 1, our main menu displayed two buttons “Load Checklists” and “Load StudyBlocks” which would open up the file explorer and allow the user to select files to import from their local hard drive. We decided this could be implemented more efficiently; these buttons required manual effort to go comb through the user’s local storage and took up too much time if the user had to go through this process every time they accessed this program. The seamless loading pull request changed it so that “saving” the program would store all local memory into respective folders, which would then be automatically loaded into the program on startup. This allows for the user to save time on startup and jump right back into using their study planner. See our accessibility report for more information.
(https://github.com/CSC207-UofT/course-project-dinosaurs/pull/81)

SCHEDULABLE INTERFACE AND EXPORTING

Having the option to export a study block was decided upon following Phase 0. This decision was made in an effort to integrate our program more smoothly into our intended user’s life and give them more flexibility in the way they want to use StudyBlocks. Our study blocks implement a Schedulable interface which contains an export function capable of downloading the information into an .ics file which is readily stored in the user’s current directory. The ics file format is compatible with Android and IOS software; modern day computers and phones have built in calendar functionality which we are able to interact with because of this interface. Any user who actively uses their calendar functionality can now incorporate our program seamlessly into their day-to-day scheduling.
(https://github.com/CSC207-UofT/course-project-dinosaurs/pull/50)

PRIORITIES, CUSTOM STUDY METHODS AND TEMPCREATOR

Having different priorities for a Checklist was a decision we implemented during Phase 0. We realized that different students have different systems for managing their studying, and our program should be able to implement as many of these styles as possible. In the preliminary planning stages, we had all Checklists sorted in order of due date with no option to organize. One of our team members suggested that some students may tackle their to-do lists in a different order, and thus we implemented options to sort by length, weight and relative importance.

In the preliminary stages, we had hardcoded three study methods (pomodoro, ultamidium and desktime) and gave the user the option to choose between them. After revisiting this implementation, we realized it did not provide much flexibility in terms of functionality, so we created the custom study method which allows the user to pick the duration of their active study time and break time. If a user does not change to a custom study method, we decided there should be a default study method to abide by simple and intuitive use. The default method chosen was the most widely recognized among the group members (pomodoro).

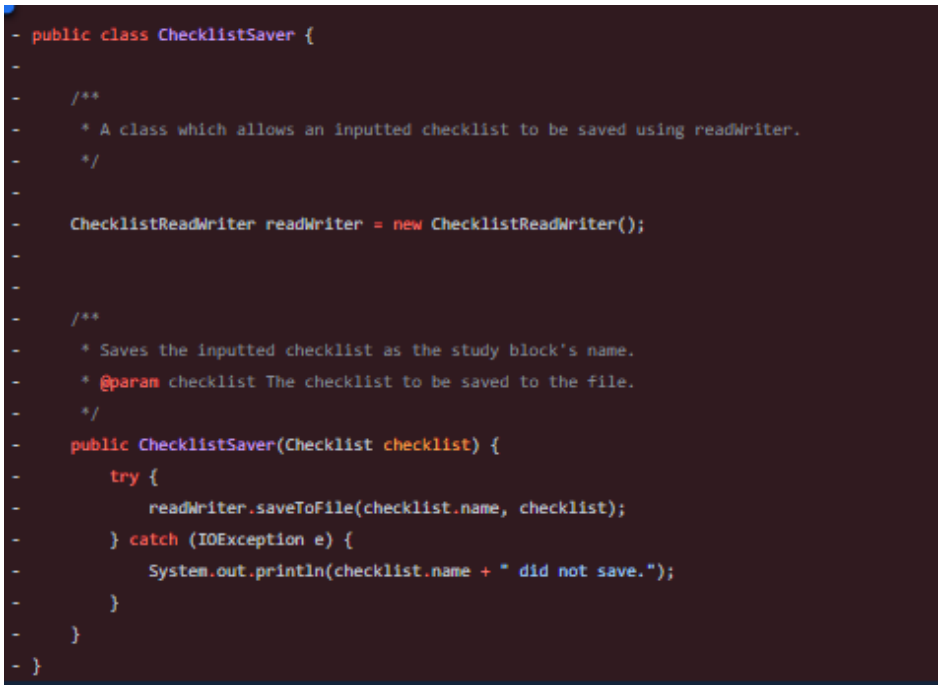
TempCreator was birthed following Phase 1. At the time, our implementation consisted of a singular Checklist which contained all the student's Tasks. The issue with this implementation was that we were designing our program in terms of our personal preferences and not keeping in mind the intended user base. Our new implementation now allows for multiple Checklists and uses the TempCreator class which can take in one or more of these Checklists and create a new Checklist from it. This way, students can make a study block out of whichever Checklists they want -- whether it be a singular Checklist for a specific day of the week or Checklists for 2 out of 5 of their courses. This implementation accommodates our own studying styles, while also providing flexibility for others.

All of these decisions were implemented in an effort to abide by the flexibility in use principle which allows users to use our program in a way that is tailored to their own needs. Since our program's intended user is any student, we recognized that a wide spectrum of preferences and abilities should be accommodated by our program.

REFACTORING

As mentioned in the overview, numerous code smells cropped up as we rushed to implement a working version of our GUI. Missing javadoc, multi-line methods and unnecessary classes were all ignored in this preliminary phase, but we slowly went through all these changes and made the code much cleaner in an effort to facilitate the understanding of any programmer who may look through our code.

One of the major refactoring steps was the removal of ChecklistSaver (Figure 2.1) and StudyBlockSaver. These were initially implemented in order to follow suit of the Data Serialization example provided in lecture. However, with the seamless loading pull request, we realized that the implementation of these classes was redundant as we could call the functions directly in the controller. Moreover, these classes did not allow for flexibility in specifying the directory where a file could be stored -- a feature which was needed for seamless loading. As such we were able to get rid of both of these classes and simplified their roles into helper functions.



```
- public class ChecklistSaver {
-
-     /**
-      * A class which allows an inputted checklist to be saved using readWriter.
-      */
-
-     ChecklistReadWriter readWriter = new ChecklistReadWriter();
-
-
-     /**
-      * Saves the inputted checklist as the study block's name.
-      * @param checklist The checklist to be saved to the file.
-      */
-     public ChecklistSaver(Checklist checklist) {
-         try {
-             readWriter.saveToFile(checklist.name, checklist);
-         } catch (IOException e) {
-             System.out.println(checklist.name + " did not save.");
-         }
-     }
- }
```

Figure 4.0 A deprecated ChecklistSaver class from Phase 1. The use of this class did not allow for specifying which directory to save the Checklist to, which became an issue when trying to save to folders created by our program.
(https://github.com/CSC207-UofT/course-project-dinosaurs/pull/65)

The use of helper functions within our program is a work in progress. The intention behind their use is to eliminate long method code smells and to make certain calls much cleaner. This was conducted in the saveAllButton as seen below. Before, all delete and save functions were written directly within this method and there were no functions being called. We realized that we could instead rewrite the bulk of this code as helper functions which could make the saveAllButton method much easier to understand (Figure 4.1).

```
@FXML
protected void saveAllButton(ActionEvent actionEvent) throws IOException {
    ChecklistReadWriter checklistReadWriter = new ChecklistReadWriter();
    if (createChecklistFolder()){
        System.out.println("Checklist folder created!");
    }


    StudyBlockReadWriter studyBlockReadWriter = new StudyBlockReadWriter();
    if (createStudyBlockFolder()){
        System.out.println("Study Block folder created!");
    }

    File checklistDir = new File(System.getProperty("user.dir") + "\\Checklists\\");
    File[] checklistFileList = checklistDir.listFiles();
    if (checklistFileList != null) {
        for (File file : checklistFileList) {
            if (file.delete()) {
                System.out.println(file.getName() + "was deleted successfully.");
            } else {
                System.out.println("Failed to overwrite files.");
            }
        }
    }
```

```
@FXML
protected void saveAllButton(ActionEvent actionEvent) throws IOException {
    if (createChecklistFolder()){
        System.out.println("Checklist folder created!");
    }
    if (createStudyBlockFolder()){
        System.out.println("Study Block folder created!");
    }

    deleteAllChecklists();
    deleteAllStudyBlocks();
    saveAllChecklists();
    saveAllStudyBlocks();
```

Figure 4.1 Reorganization of the save all button in MainGUIController. A) The deprecated implementation of this method displaying just one of the method calls needed. This function was moved into a separate helper function named deleteAllChecklists() seen in B).
(https://github.com/CSC207-UofT/course-project-dinosaurs/pull/75)

However, this style of refactoring is a work in progress and there are several methods which exceed 10 lines of code. Unfortunately, we did not have enough time to clean up all methods and there are still areas which slowly accumulated additional lines and could now benefit from the creation of helper functions. For example, the loadAll method began with only loading in the Checklists (which initially had less than 10 lines) but adding the serialization of study blocks and preferences slowly bloated this method call (Figure 2.3). Now, it serves as an example of one method call which could benefit from refactoring. Future developers should move forward trying to uphold this clean coding style and address any code smells as they crop up.

```
private void loadAll() throws IOException, ClassNotFoundException{

    if (checklistFolderExists()) {
        ChecklistReadWriter checklistReadWriter = new ChecklistReadWriter();
        File checklistDir = new File( pathname: System.getProperty("user.dir") + "//Checklists");
        File[] checklists = checklistDir.listFiles();
        if (checklists != null) {
            for (File file : checklists) {

                // Deserializes files.
                Checklist checklist = checklistReadWriter.readFromFile(file.getPath());
                Data.addToChecklistList(checklist);
            }
        }
    }
    if (studyBlockFolderExists()) {
        StudyBlockReadWriter studyBlockReadWriter = new StudyBlockReadWriter();
        File studyBlockDir = new File( pathname: System.getProperty("user.dir") + "//StudyBlocks");
        File[] studyBlocks = studyBlockDir.listFiles();
        if (studyBlocks != null) {
            for (File file : studyBlocks) {

                // Deserializes files.
                StudyBlock studyBlock = studyBlockReadWriter.readFromFile(file.getPath());
                Data.addToStudyBlockList(studyBlock);
```

Figure 4.2 Our current implementation of loadAll(). As you can see, this code is long and there are similarities between the two loading functions. At the very least it could benefit from creating loadChecklists() and loadStudyBlocks() helper functions in order to condense this method. It could also possibly be combined into one parameterized helper function.

One other example of a code smell which could potentially be refactored is the repetition of code between methods. Since our program runs on JavaFX, many of the buttons resulting in a shift in scene or window popup require the same sequence of events in order to set the scene. As of right now, we have this whole sequence written out for every button, but we believe that this could potentially be made into a helper function which is passed in the fxml file name.

CLEAN ARCHITECTURE

We fully implemented our specification during this phase. Due to our prior adherence to Clean Architecture, we were able to work independently in many different areas of the program simultaneously without disrupting the overall function of the program or needing to refactor dependent code. This shows that our interfaces and layers are well-defined, with low coupling and high cohesion between and within classes. We also combed our program for any instances where inner layer code mentioned anything declared in an outer layer.

Going into Phase 1, we created a Data class as a gateway to our storage system for when the program was running. This class contained all the Checklists, StudyBlocks and StudyMethods which were being used or manipulated in the program. If the user chose to save their work, it was the information contained in the Data class that was written to disk. While functional and convenient, we decided this implementation did not abide by clean architecture. We had a gateway Data class (existing in frameworks and drivers) being manipulated by our GUI controllers which is clearly violating the inner-outer dependency rule. After careful study of the lecture notes, we realized we could follow the dependency inversion principle and create an interface use case that our controllers could depend on. In this pull request, a DataAccessInterface was created in the UseCase packages which our Data class (now renamed to DataAccess) implements. A static DataAccess class is instantiated when the program starts up and all controller calls now use these interface methods to manipulate the DataAccess class which allows us to abide by Clean Architecture.

We placed a high priority on ensuring our inner layers were never dependent on outer layers. For example, we can consider creating a new Task. The interface of our GUI is always separate from the Controller that it depends on. The Controller only interacts with our Task and Checklist entities through the TaskManager use case class (in this example, addTaskHelper() and addTask()). After creating the Task, the TaskManager is used again to add it to a Checklist in the DataAccess class. Once in DataAccess, the GUI Controllers can access the Checklist and Task to pass them to the display, or they can be saved to disk when the program exits. This example demonstrates that data from user inputs flows inwards to our entities and then back out to our controllers and drivers, without any skipping of layers.

Finally, we addressed a number of smaller outstanding Clean Architecture violations during Phase 2. We added new use case classes to manager Study Blocks and Study Methods. With the current implementation these managers are relatively barebones, but they better define the boundary between the Controllers and Entities. Our GUI Controller classes had some instances where they were accessing Entities directly, which we addressed as we fleshed out the GUI implementation.

SOLID PRINCIPLES

SINGLE RESPONSIBILITY PRINCIPLE:

Our classes obey the single responsibility principle because each class is responsible to only one actor; this is best demonstrated in our entity classes. For example, the manipulation of data in Checklist is only influenced by the TaskManager class which is only called upon by the controllers. The controllers are then directly manipulated by the user's input (button clicking). Similarly, the StudyBlock class is manipulated by the StudyBlockManager class which is then called upon by the controllers. Since all information has a unique route throughout our program, the classes exhibit the single responsibility principle, and no class is responsible to multiple actors.

It is important to note that we do have some large controller classes that could be viewed as violating the Single Responsibility Principle or suffering from low cohesion, such as MainGUIController and ChecklistManagerController. These classes include GUI navigation methods, various presenter methods to update the UI, and controller methods that interact with the rest of the program, as well as saving and loading data from disk. Each of these areas could hypothetically be responsible to different actors. Part of this complexity is explained by the MVC view model necessitating a combined presenter/controller. We also have limited JavaFX experience and organized our controller classes by which scenes they were responsible for, including any pop-up windows owned by those scenes. These large classes could be seen as having a single reason to change (the scene they are responsible for showing and updating), or as being responsible to multiple actors (UI/UX, database management, controller and program logic).

OPEN/CLOSED PRINCIPLE:

The Open/Closed principle is shown by the classes that instantiate our interfaces: StudyBlock implements Schedulable, ReadWriter is implemented by Checklist, Preferences and StudyBlock ReadWriter and DataAccess implements DataAccessInterface. This is because they are closed for modification however, they have a lot of scope for implementation elsewhere to improve upon functionality. These classes are closed for modification as they are using all the methods from the above-mentioned interfaces but open for extension without the source code being modified and hence they satisfy the Open/Closed Principle.

Moreover, the managers for Study Block, Task, Preferences and the various GUI controllers, are all examples of this principle as well because they do not add directly to the classes they are managing, just control/organize them, making the entities Task and StudyBlock closed for modification but open for extension through the manager classes.

LISKOV SUBSTITUTION PRINCIPLE:

The Liskov substitution principle states that a superclass must be able to replace all the objects from the subclass however, our program has no abstract classes hence our program does not employ the Liskov Substitution Principle.

Given more time, the BlockManagerController, ChecklistManagerController and StudyNowController could all be made subclasses of a ManagerController abstract class perhaps with methods that they all have in common like openPopUp and changeSceneToMenuButton so the Liskov Substitution Principle could apply to the program.

INTERFACE SEGREGATION PRINCIPLE:

In our program we have used smaller interfaces and implemented or extended them rather than adding methods that are unnecessary or unused by the classes implementing the interfaces. For example, the ReadWriter interface only has two functions, centered around data persistence and file manipulation. All classes implementing this interface belong in the infrastructure package and only deal with this interaction between local storage and program memory. ReadWriter adheres to this because both methods are mutually relevant to one another. Should a client want to implement a ReadWriter class, there is no scenario where they would want to save a serialized file but not access it later (read it). Vice versa, there is no scenario where a client would want to read a pre-existing serialized object into our program (since all the relevant serialized objects are saved by our program) and so the only functions implemented are the necessary ones.

Schedulable is another interface that our program uses. It is also short and clear. Regardless of the type of the event, a calendar needs to be created, as does an event and it will need to be written up as an ICS file. The interface is then extended in other parts of the program to export a study block, to update the calendar etc hence all the methods are relevant to clients.

The DataAccessInterface follows the ISP as all the methods from it are used in the DataAccess class hence clients will only know of the methods which it needs. However, the DataAccessInterface is quite large so it could be split into smaller interfaces if the target audience is no longer students as then there may not be a need for study blocks or study methods. For the purposes of our specification, the ISP is satisfied because no methods are unused by the class implementing the interface.

DEPENDENCY INVERSION

The Dependency Inversion principle states that inner layers should not depend on outer layers - this was violated in Phase 1 when we had our Data class being manipulated by the controllers of the program. Our DataAccessInterface was implemented to follow this principle. As per the Dependency Inversion principle, our outer layer (DataAccess) implements an inner layer interface (DataAccessInterface) which is then manipulated by the layer directly above the frameworks and drivers (MainGUIController, StudyNowController, etc.) In an effort to circumnavigate this issue, the DataAccessInterface (a UseCase interface) is now being manipulated by the controllers which ultimately governs the manipulation of information stored in DataAccess. (https://github.com/CSC207-UofT/course-project-dinosaurs/pull/61)

DESIGN PATTERNS

IMPLEMENTED PATTERNS

In the GUI, we did not implement any design patterns ourselves. However, we do use Observers in the case of ObservableLists that update the various ListViews throughout the program.

In phase one, iterable was implemented in our Checklist class, and we rely on its implementation in a number of ArrayLists that store instances of Checklists and StudyBlocks. The iterator design pattern was used for the iteration through the Tasks in the Checklist. It allowed us to move through the Tasks without revealing the underlying representations. It provided us with a code that is cleaner, easier to use, easier to test, and adheres to the clean architecture principles through its implementation.

In phase 2, we implemented the Singleton design pattern for each of the constants associated with making Tasks (weight, importance, due date and length). Weight, importance, due date and length each got Singleton classes in the constants package. We decided to use this design pattern as there was only one instance of these objects and we wanted lazy initialisation.

PATTERNS CONSIDERED

We considered using the Memento design pattern for Checklists or Tasks so the user could undo their creation however we ultimately decided it is unnecessary since the delete Checklist and delete Task functions had a clean implementation. Memento is implemented to produce snapshots of the state of an object so reversion to previous states is possible or getters/ setters of an object violate its encapsulation however, our program does not violate encapsulation and we do not require reversion to previous states as the delete buttons perform that function hence we reached the conclusion as a group that memento is unnecessary for our project.

We also initially considered using the builder design pattern for the StudyBlock creation, however, we have decided against it as it would not make the StudyBlock creation more efficient. The builder design pattern is used to separate the object construction and object representation to allow the production of more than one object representation through the use of the same construction process. The StudyBlocks created all have very similar output representations, therefore the builder pattern would have not given us a more efficient construction pattern.

POTENTIAL PATTERNS FOR THE FUTURE:

Command could be implemented in the StudyMethod class to schedule study and break times and log what the student is doing during that scheduled study block. The StudytimeStartCommand will implement command and start study time. The same thing can be done to end it or start and end break times in StudyBlock. It can also be used to change study methods during the session or record the one currently being employed.

Alternatively, when students make a request i.e. add a Task in this case, the command class will call the execute method and store the details of the Task. These details like weightage, estimated time to complete, due date, etc. will be passed into client as parameters. Finally the receiver will create the Task. This way Tasks will be created cleanly and stored in TaskManager.

The Factory design pattern could be used for Checklist and StudyBlock read writer classes as ChecklistReadWriter and StudyBlockReadWriter are very similarly written save for what they are saving. Instead of having separate classes like this, we could create ReadWriter superclasses with subclasses for the type of object they are saving.

Although these design patterns were considered, we have chosen not to implement them due to current time and scope limitations. We have had to make the decision to favor implementing and improving our current work rather than working on the design pattern.

USE OF GITHUB

PHASE 1

During phase 1, we switched our GitHub branching naming pattern from using our usernames to descriptions of the area and feature we were implementing on that branch. However, we did not use a single style convention, so while the names were more descriptive, they were not unified. We also began to push and pull branches between various users before merging them. During Phase 0, the complexity of the program was low enough that we were satisfied by reviewing a branch's code on GitHub before merging. However for Phase 1, especially after the GUI was rolled out, pulling branches between one another allowed us to better review and test them before they were merged into main. We currently do not use any of GitHub's notification features, instead we rely primarily on our group WhatsApp where it is easier to reach each other quickly.

PHASE 2

During phase 2, we continued to name branches based on the feature we were implementing or editing. We also made better use of pull requests as we began to create branches based on different features we wanted to implement. This allowed for multiple group members to push commits to one branch and test out respective code before pushing it to main. This also allowed for all group members to work simultaneously on different branches; much of Phase 1 was spent waiting for one member to finish a feature which was dramatically slowing our progress. In Phase 2, this was not an issue as we simply worked on our respective features and then addressed any conflicts within the GitHub web editor, as was intended by the developers. Expanding on advice from our TA, issue tracking was improved within GitHub and many of our pull requests now include descriptions of what was changed, and what still needs to be implemented. This helped group members who were not involved in a feature's implementation still take part in the discussion and contribute meaningful ideas on how the program should be designed. (Examples: https://github.com/CSC207-UofT/course-project-dinosaurs/pull/65, https://github.com/CSC207-UofT/course-project-dinosaurs/pull/61)

CODE STYLE

PACKAGING STRATEGIES

We packaged our project according to clean architecture to ensure we were adhering to it by packaging based on the layers of clean architecture. With all entities in one package, all use cases in one package and so on. This strategy also enabled us to easily check our imports for each class to gauge whether it was accessing or dependent on any classes in a way that violated the principles of Clean Architecture. Additionally, we packaged our tests so that they are grouped with other classes that test the certain package name. For example, our “Entities” package contains the classes; Checklist, StudyMethod, StudyBlock, and Task. The package used to test these classes is called “EntitiesTests” and it contains the following classes: ChecklistTest, StudyMethodTest, StudyBlockTest, and TaskTest. By packaging our classes this way, we are able to have an organized view of each class and their respective tests.

CODING STYLES AND DOCUMENTATION

We fixed the warnings in our code which helped refactor certain methods along with removing unused methods. Every class has enough documentation at the top of the code to explain what it does. This documentation includes the parameters, attributes and the summary of the class. Additionally, every method has a javadoc which explains its description, parameters, and what it returns. For example, in StudyBlock there is a helper function which uses a javadoc to explain its parameters, function, and how it relates to the main function.

TESTING

We try to write test cases in parallel with writing the code because it helps us debug when needed. Our tests are grouped into classes that test the same packages. We have; ControllerTests, EntitiesTests, HelperFunctionTests, InfrastructureTests, UseCasesTests. These tests try to test every aspect of the element in order to check for usage.

CONTROLLER TESTS

We faced the inability to thoroughly test the controllers through test cases as they require information to be inputted from the GUI and they relied on JavaFX button functions. We manually tested the controllers in the program through creating, adding, deleting, and reorganizing the priorities of the Tasks, Checklists, and StudyBlocks. However, we were able to test the helper functions, therefore testing the existence and the creation of the Checklist and StudyBlock folders along with the deletion of those folders.

ENTITIES TESTS

These tests were mainly used to test the getters and setters of the entities. StudyMethodTest, TaskTest and ChecklistTest used set() and get() to check if the entities were being created correctly. However, StudyBlockTest is mainly used to test the functions inside the entity. StudyBlock has a function that breaks up a given StudyBlock length into its respective subblocks and a function that fills up those subblocks. These had to be tested most thoroughly because they contained the most possible edge cases. The main conflict was creating a StudyBlock when the Checklist length was more than the StudyBlock length, or when Tasks overlapped in a subblock. These edge cases were carefully tested to see how the method assignTasks() can handle them.

HELPER FUNCTION TESTS

These tests were used to ensure the proper functionality of the due date, the importance, the length, and the weight comparators which are used for Task organizing and prioritizing. The DueDateComparatorTest tests the due dates to verify they are being sorted correctly. The ImportanceTest compares the importance of the Tasks to ensure they are being sorted correctly. The LengthTest tests the length of different Tasks to check that they are being sorted correctly. The WeightTest compares and tests the weight of the Tasks to verify they are being sorted correctly.

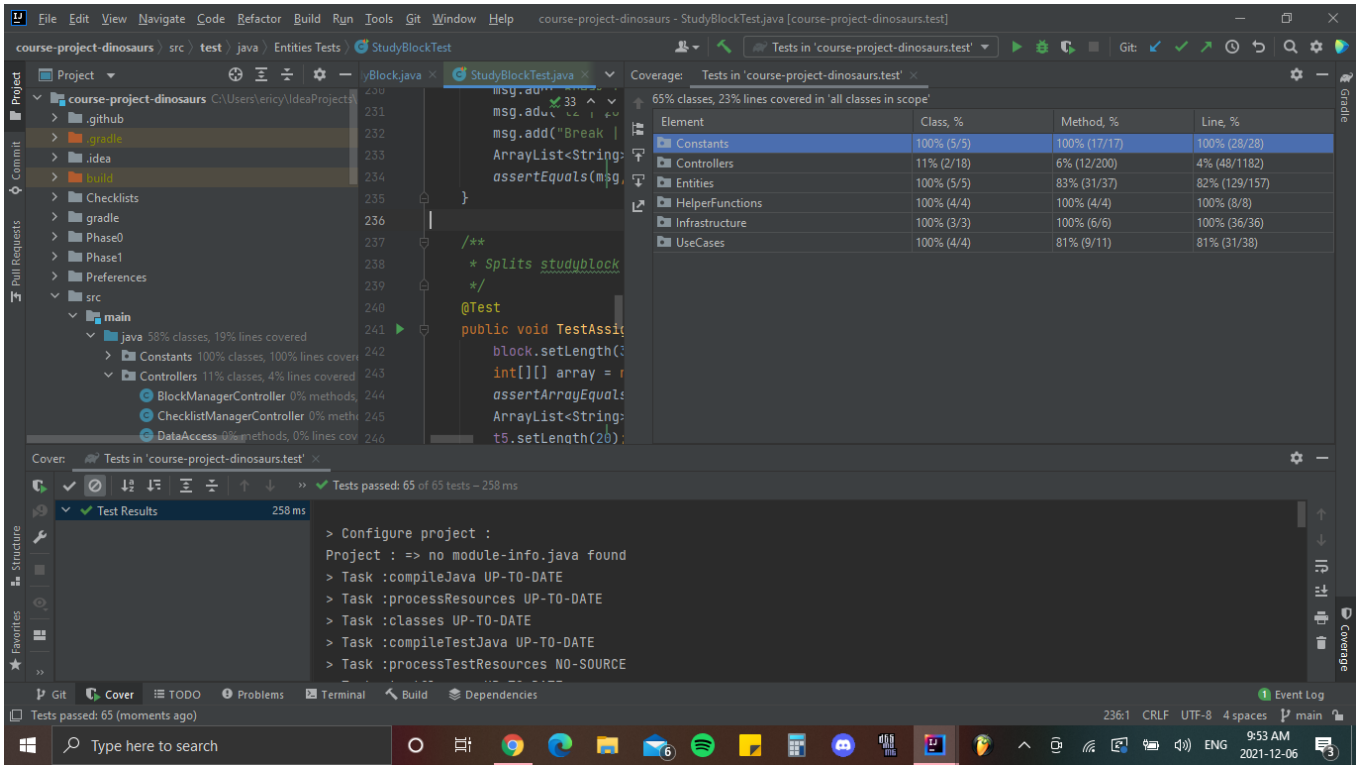
INFRASTRUCTURE TESTS

Our infrastructure tests were used to test if an object can be saved and read. We had ChecklistReadWriterTest and StudyBlockReadWriterTest inside our package which each tested their respective class. In ChecklistReadWriterTest we created a Checklist with multiple Tasks added to it and checked if our program can save one and two Checklists. In StudyBlockReadWriterTest we created a Checklist with multiple Tasks added to it and then created a StudyBlock from that Checklist, we then checked if our program could save one and two StudyBlocks.

USE CASES TESTS

We created different tests for each of the preferences manager, StudyBlock manager, Task manager, and temp creator. The PreferencesManagerTest tests ensure that the study method helper operates properly. The StudyBlockManagerTest, ensures the StudyBlock manager returns a valid StudyBlock when called. The TaskManagerTest ensures that Tasks are correctly created, ordered, and assigned as complete or incomplete. The TempCreatorTest tests the ability to create a temporary Checklist from different Checklists.

Figure 3.0 A screenshot of our test coverage across all classes and methods.



ACCESSIBILITY REPORT

EQUITABLE USE

The foundation of our program was implemented in a way which accommodates students of all kinds. Studying is a universal experience among high school, undergraduate and graduate students so our program was designed with identical means of use for all users (e.g. every student has Tasks with due dates and percentages). This open-concept design makes the program appealing to students from all disciplines. The program currently exists within the user's local device, ensuring that personal information is protected from outside users.

FLEXIBILITY IN USE

Our program was designed for any student to use so we had to alter much of our implementation to accommodate a wide range of preferences and abilities. Some examples of this include the option for Checklists to be reorganized based on due date, length, importance or weight. Our study block is able to be divided up in whatever time intervals the user desires; this can be changed in the preferences menu. Finally, we allow for multiple Checklists to be used when creating a study block. This allows for students to customize how they'd like to structure their studying, whether it be one or multiple Checklists.

SIMPLE AND INTUITIVE USE

The seamless loading function was implemented in an effort to abide by this principle; it allows for users to quickly return to an instance of their program without any additional complexity. However, we were not able to update some of our GUI elements automatically which can be confusing if the user expects their data manipulation to be reflected immediately. Furthermore, the sequence of using our program (Checklist, Task, StudyBlock) may be confusing to an outsider who has not read our README file. In the future, we would like to add the option to gain feedback from our users and identify areas where our design was unintuitive or difficult to navigate.

PERCEPTIBLE INFORMATION

Our program utilizes ListViews to directly present the relevant information back to the users as it is being manipulated. Our program does not use a colorful layout, it is mainly black, white, and gray, therefore making the contrast and the visual view very simple to avoid overwhelming the users. Adding some of the JavaFX properties that allow the integration of accessibility devices with our program would increase their compatibility and therefore lessen the limitation for users with limited perceptual abilities.

TOLERANCE FOR ERROR

Our program has an exception handler that opens a “Something Went Wrong!” pop-up window when an IOException or ClassNotFoundException is thrown (usually in the instance of a file error). This was an improvement from the initial design (a print statement in the terminal) but can be expanded upon in the future to be more descriptive for troubleshooting. If a user accidentally completes a Task, they have the option to view their history and revert this Task back to incomplete. Our saving function allows the user to save any work that they have put into our program.

LOW PHYSICAL EFFORT

The seamless loading function adheres to this principle. Before, the user had to manually dig through their hard drive in order to load in study blocks, Checklists and study methods. With seamless loading, this is all done automatically, and the user can immediately continue to work with whatever data they had stored previously. In the future, we can provide an option for the user to edit the information stored instead of deleting and recreating any Tasks, Checklists or study blocks. This can help minimize fatigue from using our program.

SIZE AND SPACE FOR APPROACH AND USE

Our program is currently only formatted in a rigid window; opening fullscreen does not reformat the buttons which may hinder some user’s ability to see our program. In the future, we should work to accommodate different viewing modes which allow for more flexible use in standing, sitting, close or far away. All the buttons are implemented relative to this rigid view and are all properly formatted within the user’s line of sight.

TARGET USERS

Our program is most likely to be marketed towards students as it was designed to help organize the student's Tasks. However, our program was designed with the intention of accessing a wide variety of students, whether it be high school, undergrad or graduate. The information stored in our program is largely ubiquitous across all students. For example, every student will be assigned Tasks or assessments which all have due dates and percentage weights. Every student is likely to have multiple Tasks at once and will need to devote time to completing these Tasks (which can be done through our study block function). We initially wanted to investigate integrating timetable information from Quercus into our app, but that was beyond the scope of this course, and we decided to keep our target user non-specific to a university.

NON-TARGET USERS

While much of the program was designed for ubiquity amongst students, it was still designed with the concept of a student user. For example, someone who is not in school may still find use for our program, though many of the features may be irrelevant (e.g., most non-school Tasks do not have a weight). If we take an office worker, for example, they may still have deadlines (due dates) to meet and can still estimate the relative importance and length of their Tasks. In terms of accessibility support, our program is extremely limited in its capacity. For example, there are no text-to-speech functionalities within our program which excludes blind people entirely from using the study planner. JavaFX allows for accessibility support through the implementation of specific features (e.g., screenwriting) and should be seriously considered in future development.

PROGRESS REPORT

WHAT WORKED WELL?

We had strong communication among the group members which worked well in organizing the group meetings, as well as updating one another about the progress of our project. Our communication especially on the group WhatsApp worked very well as it was easy to get in touch with and discuss the project with any group member, this was crucial when trying to integrate multiple branches simultaneously into main. With respect to the project, the GUI is something we are all quite proud of. Although there are small limitations here and there (e.g., no automatic refresh), we spent a lot of time ironing out bugs and formatting the menus in order to present a clean output to the user.

WHAT WE WORKED ON DURING PHASE 2

Pooja:

Added a remove Task method in the Task manager and used that to code the delete Task button. Also worked on the delete Checklist button and created classes for weight and importance sliders. Made Singleton classes for the constants used to create Tasks.

Significant pull request: https://github.com/CSC207-UofT/course-project-dinosaurs/pull/62

This pull request is for the delete Checklist and Task buttons. It is significant because without these methods our program would not perform the way we intended it to and it is an example of GitHub use for communication as another member commented on my pull request and I edited it accordingly and merged with main later.

Zoya:

Linked the StudyBlock creation, display, and export methods with the GUI. Modified some of the GUI appearance in the StudyNow view and the BlockManager view where StudyBlocks are created and I constructed different appearances for the different ways of creating the StudyBlock. Created a StudyBlockManager UseCase to allow Controllers to create a StudyBlock without breaking clean architecture.

Significant Pull request: https://github.com/CSC207-UofT/course-project-dinosaurs/pull/67/files

This pull request contains most of the work done on the GUI regarding the StudyBlock creation, export, and modification. This is an important contribution to the project because it links our work to the GUI to allow the user to create StudyBlocks. It also portrays the different ways the user can create a StudyBlock either using one Checklist, more than one Checklist, or all Checklists.

Jeb:

Implemented various remaining buttons in the GUI, such as sorting Checklists by different priorities, adding Sliders to the AddTask popup, updating and saving StudyMethod, marking Tasks complete and viewing previously completed Tasks, as well as tests and helpers where appropriate.

Significant Pull Request: https://github.com/CSC207-UofT/course-project-dinosaurs/pull/79

This pull request finally implemented our final entity: StudyMethod. It contains the creation of a new UI that displays the currently selected StudyMethod and allows the user to create a custom active/break time setup. It created a new PreferencesManager class that currently only has a helper to instantiate new StudyMethods but could be expanded to cover any other user preferences in future updates, like selecting a folder for exporting .ics files, changing the username or password, and more. Finally, this pull request made StudyMethod a serializable data type so that user's preferences would be saved after they closed the program.

Ken:

Added a helper function to the assignTasks() that reduces the time it takes to create a study block. This meant rewriting StudyBlock to use the helper function. StudyBlockTests was updated to use an existing Checklist and to use the new implementation of StudyBlock. The javadoc was finished for both StudyBlock and StudyBlockTest.

Significant Pull Request: https://github.com/CSC207-UofT/course-project-dinosaurs/pull/82

The pull request above shows the final StudyBlock class with the helper function. StudyBlock takes in the other entities and returns a filled StudyBlock, which the rest of the code uses to display to the user. This pull request was crucial to the project because most of the program uses StudyBlock to display what the todo list is for the user.

Paridhi:

Learned GUI and then worked along on StudyBlock GUI implementation. Worked on the delete button for StudyBlock, made changes to a few pop-up windows.
Initially a delete popup was implemented, which was then modified. And the code was shortened to avoid redundancy.

Significant Pull Request: https://github.com/CSC207-UofT/course-project-dinosaurs/pull/67/files

Initially the code was redundant. So, I modified some parts of it to make it shorter. There was also a delete now pop-up, which was then modified. This pull request is significant as it demonstrates the changes we have made throughout the code in StudyBlock.

Eric:

I mainly worked on revamping the loading function to make it seamless, this involved doing research into directory access as well as working with the java.io object File. The idea was to access all files in our created folders and extract them into our program within the initialize function. I also worked on the DataAccessInterface and implemented this for clean architecture adherence. Finally, I increased test coverage for every class and tried to hit as many methods as possible.

Significant Pull Requests:

1. Seamless Loading: https://github.com/CSC207-UofT/course-project-dinosaurs/pull/65
2. DataAccessInterface: https://github.com/CSC207-UofT/course-project-dinosaurs/pull/61

The seamless loading pull request demonstrates a restructuring of the file system and how the program demonstrated data persistence. The data access interface pull request was created to abide by clean architecture and includes a restructuring of the controller and outer layers.

SCOPE FOR EXPANSION AND LIMITATIONS

SCOPE FOR EXPANSION

- If we were to expand the scope of our project, given more time, we could implement a local calendar that allows users to input their lecture and tutorial times so the app serves as a study app that also tracks when the student is busy with university related stuff. They can add club meetings and office hours as well.

- We could potentially add a reminders function which notifies the user of when their deadlines are fast approaching e.g., 1/2 hours before the deadline or lets users pick how often they want reminders/notifications, so they are notified in a way that works best for them. Many apps offer push notifications, and this functionality would do just that.

- We could also potentially add a timer to the program that is displayed when the person is studying or working on a specific StudyBlock, this enables the user to follow their wanted study method more accurately rather than loosely follow the time blocks. This also makes the studying more efficient as following techniques like pomodoro increase productivity and effectiveness of studying.

- The GUI currently has no design patterns implemented to make its code simpler, reformatting the GUI classes and methods in order to incorporate the decorator design pattern would help make the GUI cleaner.

LIMITATIONS

- Our current program does not allow multiple users who share a device to have their own different planners. Moreover, the user cannot access their planner anywhere but the device it is found on. Therefore, we could implement a remote access system to enable the user to access the study planner from different devices and not be limited to one system. This would make the program more usable and flexible to fit all the user's potential needs.

- Our current program is only accessible through a computer and not other devices. This would limit the user's ability to flexibly add Tasks, create Checklists, and make StudyBlocks from their phone or tablets.

- Our current GUI window cannot be scaled up or down, also, the font cannot be changed if someone finds it too small or too big. Incorporating reformatting options for the GUI window would make our current program more convenient for the user.

- This program is mostly geared towards students, others could use it, however as they are not our main audience, some of the program options could seem less useful for them such as StudyBlocks.

- Currently, the program does not have refresh functions implemented properly, therefore making the program less efficient and adding a nuisance of manually refreshing every time. Implementing refresh functions could increase the efficiency of the program and bring it closer to being more accessible from a perceptible standpoint.

- Under our current program, once Tasks are made, their length must be assumed by the person and that is a limitation as no one knows the accurate time it would take to complete a Task. To face this problem, a dynamic Task timing could be implemented, one that allows the user to change the projected length of the Task easily, especially during StudyBlock creation.

- Currently, when users create Tasks, there is no way of editing them. They can only be deleted, not modified. This is slightly inflexible as people often get extensions so due dates could change and people often underestimate the time it takes to do a Task hence length may need editing as well. To deal with this, an edit Task button could be added once users go to manage Checklists, allowing students to edit the selected Task.