Tee Jun Jie Ivan - Project Portfolio

PROJECT: Code Camp X

Overview

Code Camp X (CCX) is for those who prefer to use a desktop app to manage the administrative tasks of running a coding camp. More importantly, CCX is optimized for those who prefer to work with a Command Line Interface (CLI) while still having the benefits of a Graphical User Interface (GUI). If you can type fast, CCX can allow you to manage your camp's administrative tasks faster than traditional GUI apps.

Summary of contributions

- **Major enhancement**: Implemented Code Base for `Assignment` and `Progress`
  - What it does: This allows the user to add `Assignments`, with a name and deadline into the program. This also allowed for future extension where a `Progress` object can be created to tie a `Student` and `Assignment` together.
  - Justification: This allows the users to track all the `Assignments` that is in the camp and will help in the implementation of tracking student progress in the subsequent extension.
  - Highlights:
    - An in-depth analysis was required to determine if `Progress` was really required since it takes up a lot of time to successfully add another `AddressBookGeneric` into the program, while taking into account the UI, the storage the commands, parsers, etc. However, the group decided that by splitting the logic of `Assignment` and `Progress`, this will allow for better usability and testability. This was discussed within the Developer's Guide as well.
    - Another highlight is that I needed to implement a special form of ID, which I call `CompositeID`, just for `Progress` object, since `Progress` objects are identified by 2 IDs, studentID and assignmentID. Due to this, I needed to implement many methods in `ModelManager` in order to allow for model handling via this `compositeID`.
- **Major enhancement**: Implemented all **Undo/Redo-able** `Assign` & `Un-assign` commands
  - What it does: Allow user to assign/un-assign a `Student`/`Teacher`/`Assignment` to a `Course` and allow for them to undo/redo those commands
  - Justification:
    i. This allows the user add/remove a student/teacher/assignment to and from a course, allowing for them to better manage the their coding camp.
    ii. Should they have executed the command wrongly, they are able to undo or redo the commands they have just executed while returning to the **correct** previous state.
  - Highlights: This was one of the most challenging portions of the program due to these reasons:

1. It ties in many of the functionality and classes implemented. For example, the Assign/Un-assign commands need to make sure that entity linking is **correct** and that `Progress` objects added/removed **correctly**. As such, it is very dependent on `EdgeManager` and `ProgressManager` for correct implementation.
2. Many design alternatives were considered. Initially, the logic of assigning the edges was within the `Assign`/`Un-assign` command itself, but I refactored everything out to the `ModelManager` and subsequently `EdgeManager` to ensure that we better follow design principle of `Single-Responsibility Principle`.
3. Pre-processing of Command was **vital** since our program works via IDs. The method is crucial in making sure that the entities are in a `consistent state` before executing the assign command. `Consistent state` means that either the two targeting objects have each other's IDs or they do not. There should never be the case where one party has the other's ID but not vice versa.
4. Allow for Undo/Redo-ability into the **correct** state
    a. Generally, when undo/redo is called, we just have to execute the opposite command.
    b. However, we realized that it was not so trivial when we discovered a huge bug where undo-ing an un-assign command does not put back the specific `Progress` objects that were removed.
    c. As such, special considerations had to be implemented to ensure that these `Assign`/`Un-assign` commands can work correctly.
    d. These considerations are included in the Developer's Guide as well.

- **Minor enhancement**: Implemented ProgressManager to allow for successful Creation/Deletion of `Progress objects
- **Minor enhancement**: Implemented Assign/Un-assign portion of EdgeManager
- **Minor enhancement**: Implemented Done/Undone Commands to mark Progress objects as `Done`/`Undone`
- **Code contributed**: [Functional code]
- **Other contributions**:
    - Project management:
        - Managed releases `v1.2` (1 release) on GitHub
        - Managed Deadlines and Deliverable
    - Enhancements to existing features:
        - Allow for additions and removal of `Assigment` object entities
    - Documentation:
        - Complete documentation of `Model`, `Entity Linking` and `Student Progress Management` in Developer's Guide
        - Added `Assignment` and `Done`/`Undone` Commands into User Guide
    - Community:
        - PRs reviewed (with non-trivial review comments): #91, #111, #188

- Contributed to forum discussions (examples: 1)
- Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3)

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

Assignment

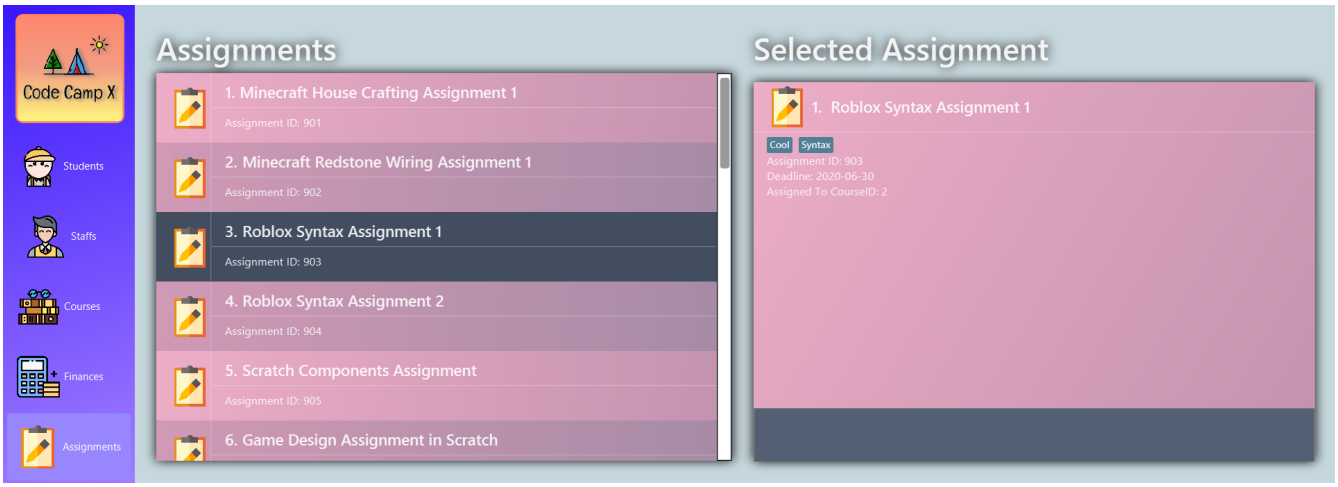


Figure 1. List of Assignments. Shows details of selected Assignment.

Add an assignment: `add-assignment`

Add a new assignment to the list of all assignments
Format: `n/NAME dl/DEADLINE [t/TAG]…`
Example: * `add-assignment n/Adversarial Search Assignment 2 dl/2020-12-30 t/AI t/Difficult`

NOTE	`DEADLINE` must be in `YYYY - MM - DD` or `YYY - MM - DD` format else an error will be thrown.

Delete an assignment: `delete-assignment`

Format: `delete-assignment ID`
Example:

- `delete-assignment 16100`

- Deletes the item at the specified `ID`. The ID refers to the ID number shown in the displayed item list
- The ID **must be a positive integer** 16100, 25200, 52500, …

NOTE	Be reminded that when you delete an assignment, the assignment will be deleted from every course as well. Furthermore, the progress of the assignment will also be removed from every course of any student and every student of any course. For example, a student that was only assigned to a course that contains only this assignment will have no assignment left after this assignment is deleted.

Edit an assignment: `edit-assignment`

Format: `edit-assignment ID [n/NAME] dl/DEADLINE t/TAGS]`
Example:

- `edit-assignment 16100 n/Edit Python Assignment 1`

- Edits the assignment at the specified `ID`. The ID refers to the ID number shown in the displayed course list panel
- The ID **must be a positive integer** 16100, 2520, 52500, …
- At least one of the optional fields must be provided
- Existing values will be updated to the input values
- When editing tags, the existing tags of the item will be removed i.e adding of tags is not cumulative
- You can remove all the assignment's tags by typing `t/` without specifying any tags after it

Locating items by name: `find-assignment`

Finds assignments whose names contain any of the given keywords.
Format: `find-assignment KEYWORD [MORE_KEYWORDS]…`

- The search is case insensitive. e.g `java` will match `Java`
- The order of the keywords does not matter. e.g. `Java Assignment` will match `Assignment Java`
- Only the name is searched
- Only full words will be matched e.g. `Java` will not match `Javascript`
- Items matching at least one keyword will be returned (i.e. `OR` search). e.g. `Java Assignment` will return `Java Course`, `Java Code` and `Python Assignment`

Examples:

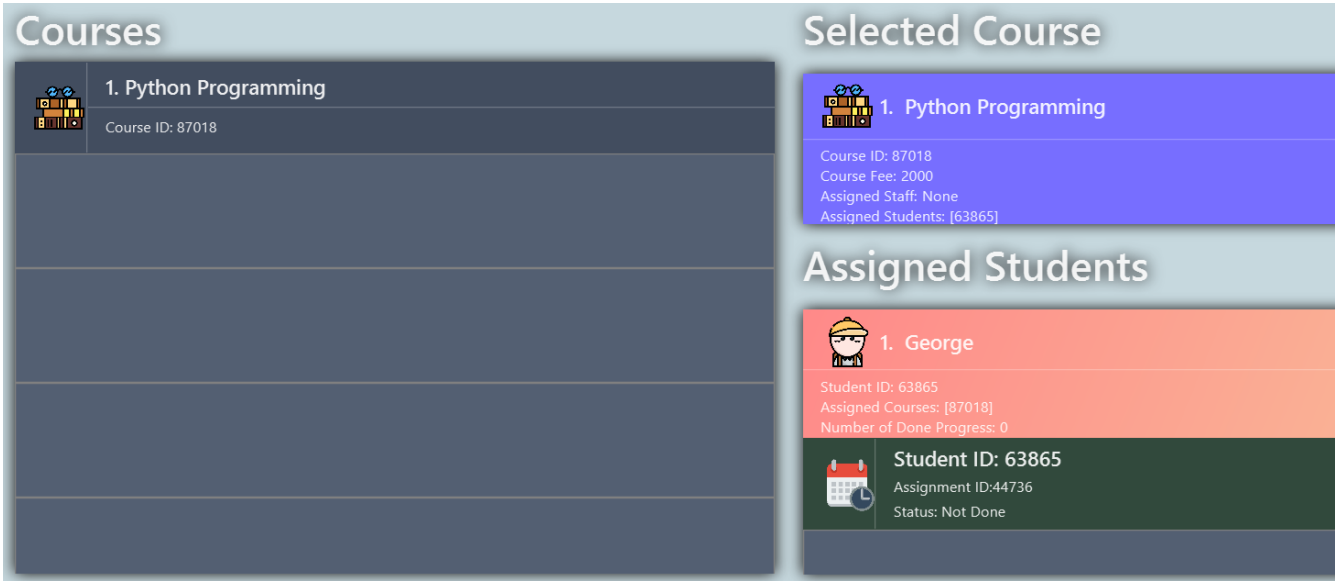
- `find-assignment java`
  Returns `java` and `Java Assignment`

Mark a Student's Assignment as Done/Undone

Mark a student's assignment as Done.
Format: `done aid/ASSIGNMENTID sid/STUDENTID`
Example: `done aid/829 sid/21`

Mark a student's assignment as Undone.
Format: `undone aid/ASSIGNMENTID sid/STUDENTID`
Example: `undone aid/829 sid/21`

Illustration:



Notice that George has initially not completed Assignment 44736.

After `done aid/44736 sid/63865`:

Now observe 3 updates:

1. Success message as displayed in message box

2. The `status` of the Assignment has been marked as done
3. The `Number of Done Progress` has been incremented by 1

Contributions to the Developer Guide

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

Model component [Tee Jun Jie Ivan]

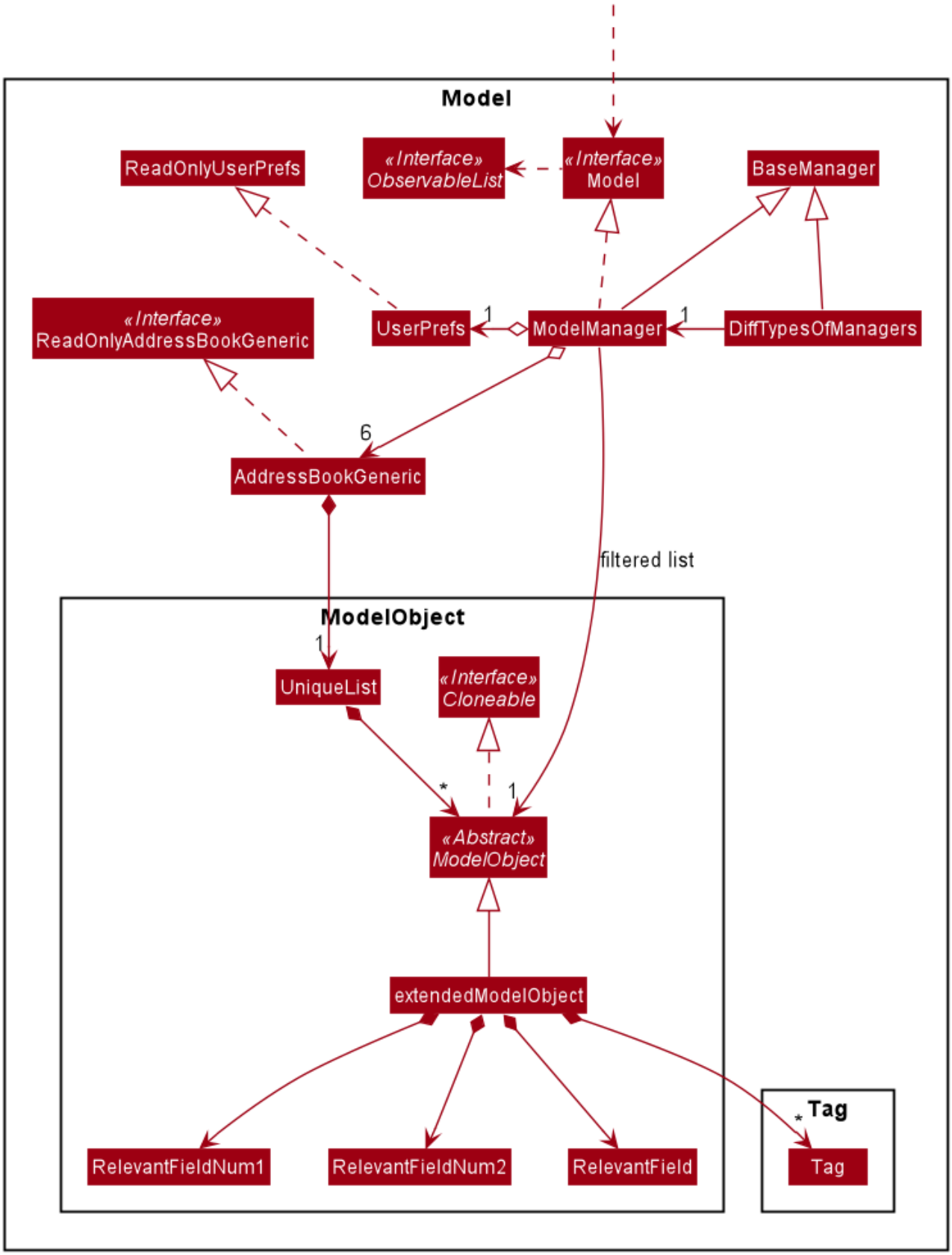


Figure 2. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the 6 AddressBookGeneric<K extends ModelObject>, each of which holds a different type of ModelObject. The 6 types are namely
    1. Student
    2. Course
    3. Staff
    4. Assignment
    5. Progress
    6. Finance
- exposes an unmodifiable `ObservableList<K extends ModelObject>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other components.

Below is an example of the different types of RelevantFields that can be tied to an Assignment.

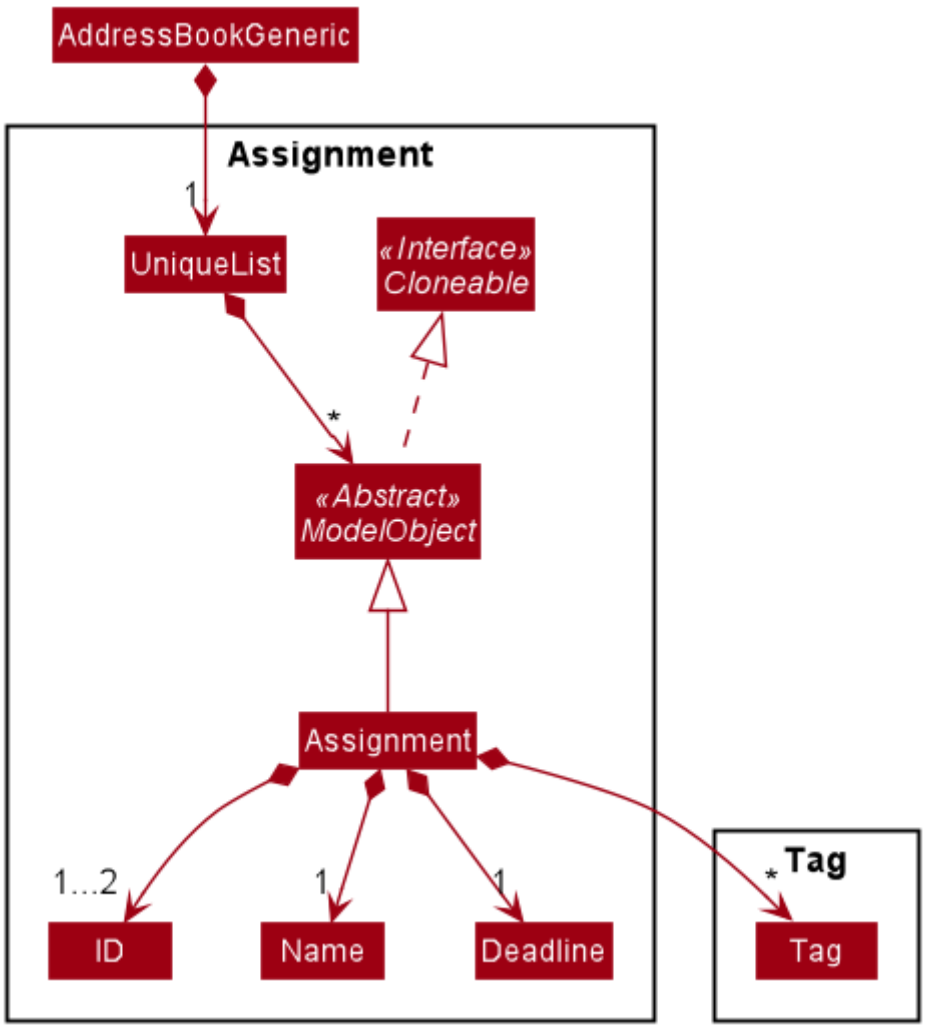


Figure 3. Class Diagram of Assignment

NOTE	The AddressBookGeneric in the diagram above is actually an AddressBookGeneric<Assignment>. The AddressBookGeneric has been made to accept any class that extends ModelObject. This cannot be shown due to limitations in PlantUML.

Notable Implementations in Model

1. By making use of `Generics` and `Polymorphism`, the group has made it such that `AddressBookGeneric<K extends ModelObject>` can hold any class that extends from ModelObject

   Benefits

   a. Allows for code optimization by having reusable code. There is significant decrease in workload when code can be reused for each others' benefit instead of having duplicated code.

   b. Allows for extension easily for future features. Future features that involve creating new AddressBooks can be developed very quickly and allow for faster development of future features.

2. All `ModelObjects` implement `Cloneable` so as to allow for Defensive Programming more easily.

   a. Please refer to `Step 2 of Section 2.2.2` for the team's rationale behind having ModelObject implement Cloneable.

3. All Non-Crud Commands such as `Assign`/`Un-assign`/`Done` are handled in `DiffTypesOfManagers` such as `EdgeManager` or `ProgressManager` instead of having all implementations being done in `ModelManager`

   Benefits

   a. Easier implementation since lower level implementations can be abstracted away

   b. More decoupling which will lead to be better testability and easier debugging

Entity Linking - Assigning/Un-Assigning an Entity to/from a Course [Tee Jun Jie Ivan]

In order to allow the tracking of the students/assignments/teachers that are assigned to a course and vice versa, this required us to implement a structure which allowed us to obtain information from the aforementioned objects, without causing any circular referencing errors.

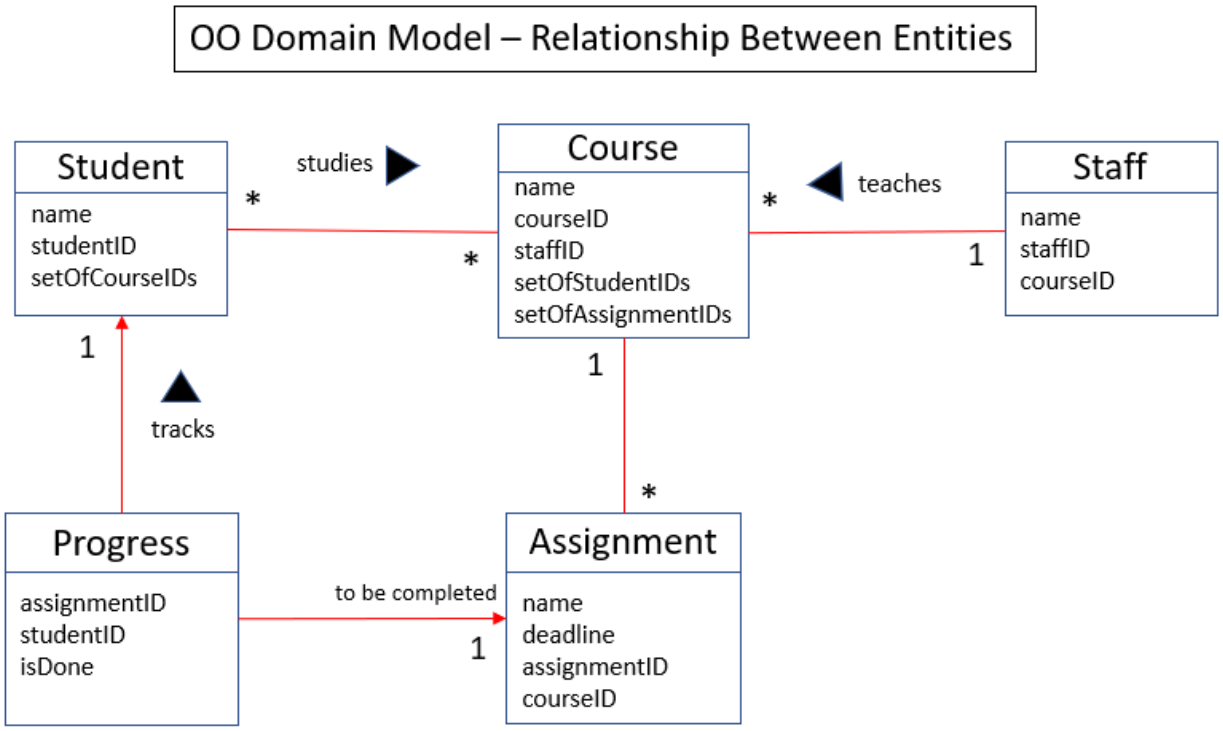


Figure 4. Relationship between Entities

The group came up with the structure above where we centralize most links around the `Course` object so as for easier management of entity links.

IMPORTANT	Notice that `Student` does **not** hold a `Course`, but a set of `CourseIDs`. Hence, there should be an association between `Student` and `ID` instead of `Student` and `Course`. However, the group found it much more **intuitive** to think of the associations to be from 2 modelObjects rather than to and from IDs. Every non-directed association between 2 objects ensures that both objects have each other's ID.

The only exception is `Progress` objects which are created via a composite ID of `studentID` and `assignmentID`. A more detailed explanation of Progress Management is explained in Student Progress Management [Tee Jun Jie Ivan].

Entity Linking is managed exclusively by `Edge Manager`

- Ensures that links are maintained/removed properly during assign, un-assign, delete commands
  API : `EdgeManager.java`

Execution of Assign/Un-assign Command [Tee Jun Jie Ivan]

For the actual execution of an assign/un-assign command, 2 main steps are performed.

1. Pre-process the targeted entities to ensure consistent state - Via `PreprocessUndoCommand` method call

2. Add/Remove both object's ID into/from each other - Handled by `EdgeManager`

Step 1: Preprocess Entities

<u>Rationale</u>

Firstly, a `pre-processing step` must be performed before executing an undo-able assign/un-assign command to ensure that all entity links are in correct state before command execution. This means that either

1. Both targeted objects have each other's IDs or
2. They do not

There should be no instance where Course has an Assignment/Student/Staff's ID but they do not have the Course's ID or vice versa.

<u>Current Implementation</u>

Below is an activity diagram showing the pre-processing performed for assign commands. The diagram can be generalized for un-assign commands by checking if the course contains X and vice versa in the second stage instead.

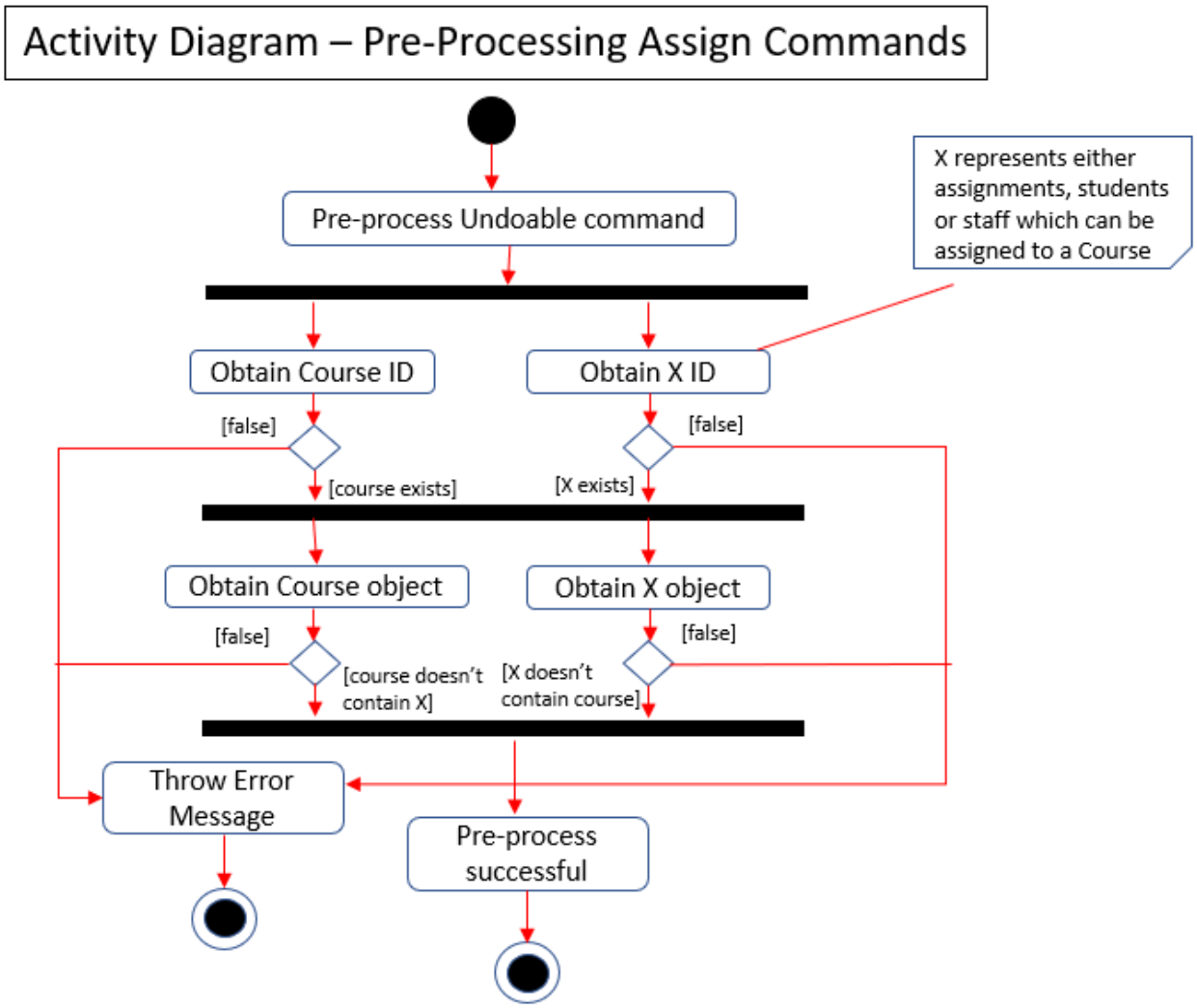


Figure 5. Activity Diagram of Pre-processing for Assign/Un-assign commands

Notice that there are 2 main exit points in the activity diagram.

1. The success case is straightforward and will lead to a the program continuing to execute the actual assign/un-assign command.
2. For the failure case, should any of the conditions fail, this means that either that the
    - specified objects does `not exist`,
    - both entities are `already assigned` to each other or,
    - most importantly, that the model is in an `inconsistent state` where one entity is assigned to the other but not vice versa.

Step 2: Assign IDs via EdgeManager

<u>Rationale</u>

After the necessary checks have been performed, respective IDs need to be added to the targeted course and targeted object in order to ensure correct and consistent assigning of objects.

<u>Current Implementation</u>

Below is a sequence diagram of how EdgeManager adds the IDs to the two objects involved.

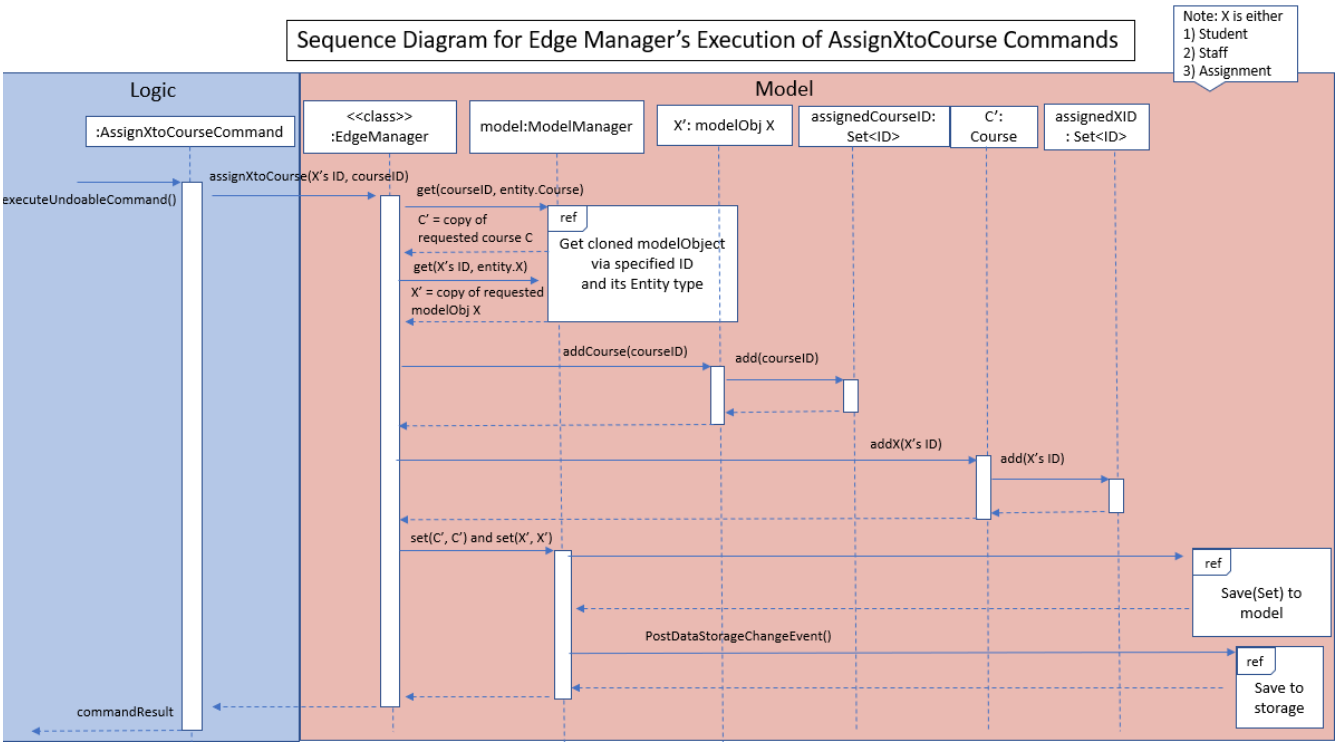


Figure 6. Sequence Diagram of EdgeManager's AssignXtoCourse commands

The flow of an assign command is as follows:

1. Obtain a copy of the requested `modelObjects` from `ModelManager`
    a. One of which must be a `Course` modelObject, the other being either a `Student`/`Staff`/`Assignment` modelObject

2. For object X', add the `courseID` into the assignedCourseIDs in X`
    a. For assignments, every assignment can only be assigned to at most **one course**. Hence, it'll hold just an assignedCourseID instead of a set of Course IDs.
3. For Course C', add `X's ID` into assignedXIDs in C'
    a. For courses, every course may only have at most **one teacher**. Hence, course will have an assignedTeacherID instead of a set of Teacher IDs.
4. Update the 2 affected models by executing `set(targetObj, editedObj)` in modelManager for both changed modelObjects
5. Update the storage by running postDataStorageChangeEvent() - ref to `Figure 2` to understand how storage save is triggered

NOTE	This sequence diagram can be generalized for `un-assign command` as well. Instead of adding IDs, un-assign will remove each other's ID from the respective objects.

<u>Design Considerations</u>

1. Manipulating the `actual` modelObjects by having modelManager#get() return referenced variable of the actual `modelObjectTags`.

    Pros

    a. Simpler implementation since any changes to the model will change the actual object directly

    b. Faster execution since any changes is done to the actual modelObject

    Cons

    a. Might allow for unintentional changes to the actual modelObject

2. (Current Implementation) Manipulate a `cloned` modelObject by having modelManager#get() return a copy of actual modelObject.

    Pros

    a. Prevents unintentional modifications of the shared object

    Cons

    a. Extra processing required. For example, a method is required to replace the old modelObject with the new model object. Also, requires all ModelObject classes to implement a clone() method.

    b. More memory intensive and can hurt overall program performance.

Overall, the second option was chosen since the program is very dependent on maintaining a `consistent state`, where either 2 modelObjects have each other's ID or they do not. Some performance can be sacrificed in order to ensure that the links between objects cannot be modified by mistake.

Summary of Entity Linking

Overall, in order to ensure successful entity linking, the role that `EdgeManager` plays is crucial. The table below shows the method calls made to `EdgeManager` during an `Assign`/`Un-assign` command.

Command	Method Call within the Command	EdgeManager method call
AssignStudentToCourse	PreprocessUndoableCommand() Is executed for all Commands to ensure consistent state	assignStudentToCourse()
AssignTeacherToCourse		assignTeacherToCourse()
AssignAssignmentToCourse		assignAssignmentToCourse()
Un-assignStudentFromCourse		unassignStudentFromCourse()
Un-assignTeacherFromCourse		unassignTeacherFromCourse()
Un-assignAssignmentFromCourse		unassignAssignmentFromCourse()

Figure 7. Table Summary of EdgeManager's involvement during Assign/Un-assign Commands

Student Progress Management [Tee Jun Jie Ivan]

The `Progress` of students is managed exclusively by the `ProgressManager` class.

API : `ProgressManager.java`

Student Progress Creation/Removal

<u>Rationale</u>

New `Progress` objects must be created in 2 main scenarios.

1. If a `Student` has been added to a `Course`, the `Student` will need to complete all `Assignments` that have already been assigned to the `Course`.
2. If an `Assignment` has been added to a `Course`, all `Students` currently taking the `Course` must now complete that `Assignment`.

The rationale is similar when un-assigning either `Student` or `Assignment` from a `Course`. `Progress` objects need to be removed instead.

<u>Current Implementation</u>

Below is a sequence diagram illustrating how the ProgressManager adds Progress objects into the ProgressAddressBook when a `AssignStudentToCourse` command is run.

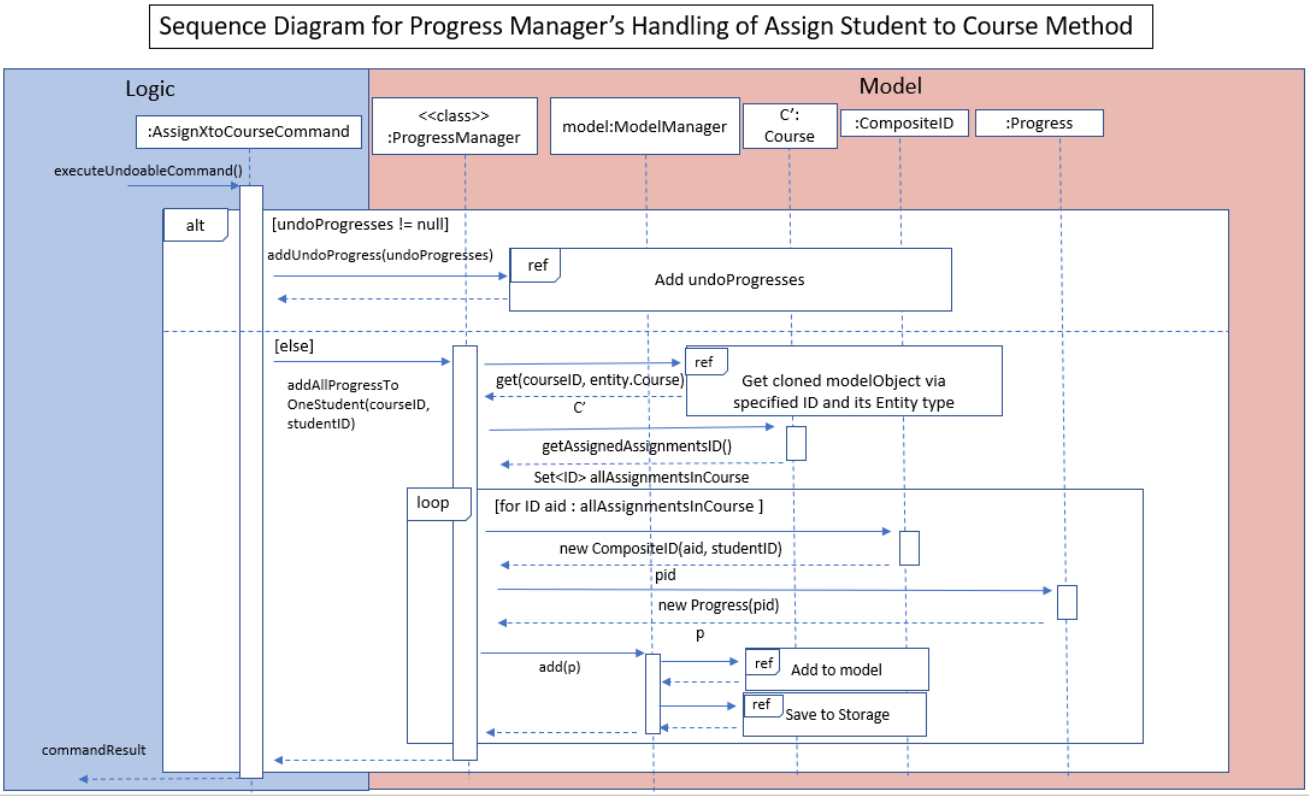


Figure 8. Activity Diagram for creating Progress objects when assigning a Student to a Course

The flow of adding Progress objects in a AssignStudentToCourse is as follows

1. If there are `Progress` objects that were previously in ProgressAddressBook but were removed due to an `Un-assign/Delete` command, add those `Progress` objects back in.
    a. This only occurs when undo-ing an `Un-assign/Delete` command. Refer to Opposite command for assign/ un-assign [Tee Jun Jie Ivan] for more information.
2. Else, i.e. if this is an entirely new AssignStudentToCourse command,
    a. Obtain **C'**, a copy of the targeted `Course`
    b. From **C'**, obtain all assignmentIDs that are assigned to it.
    c. For each of the obtained assignmentIDs, create a new CompositeID of assignmentID and studentID
    d. Create a `Progress` object using each of the `CompositeID`
    e. Finally, add these `Progress` objects into the `ProgressAddressBook`.

Notes

- The flow is largely the same for AssignAssignmentToCourse! Instead, we want every `Student` that is currently studying in course **C'** to complete the particular `Assignment`. Hence, rather than obtain all assignmentIDs, we will need to obtain all `studentIDs` that are studying **C'**, and create a `Progress` object so that we can track whether those students have completed the assignment.
- The flow is also largely the same for UnassignCommands! Instead, we are looking to remove Progress objects rather than adding them back in.

The table below shows the summary of method calls from `ProgressManager` for each variant of `Assign`/`Un-assign` command.

Command	ProgressManager method call	
	no UndoProgresses	have UndoProgresses
AssignStudentToCourse	addAllProgressToOneStudent()	addUndoProgress()
AssignTeacherToCourse	X	
AssignAssignmentToCourse	addOneProgressToAllStudent()	addUndoProgress()
Un-assignStudentFromCourse	removeAllProgressFromOneStudent	X
Un-assignTeacherFromCourse	X	
Un-assignAssignmentFromCourse	removeOneProgressFromAllStudents()	X

Figure 9. Table Summary of ProgressManager method calls() for Assign/Un-assign Commands

Marking Progress as Done/Undone

<u>Rationale</u>

When a `Student` finishes an `Assignment` that is allocated to him, you want to be able to mark his work as `Done`.

Similarly, if an `Assignment` has been mistakenly marked as `Done` or is actually `Undone`, you want to be able to mark the `Assignment` as `Undone`.

<u>Current Implementation</u>

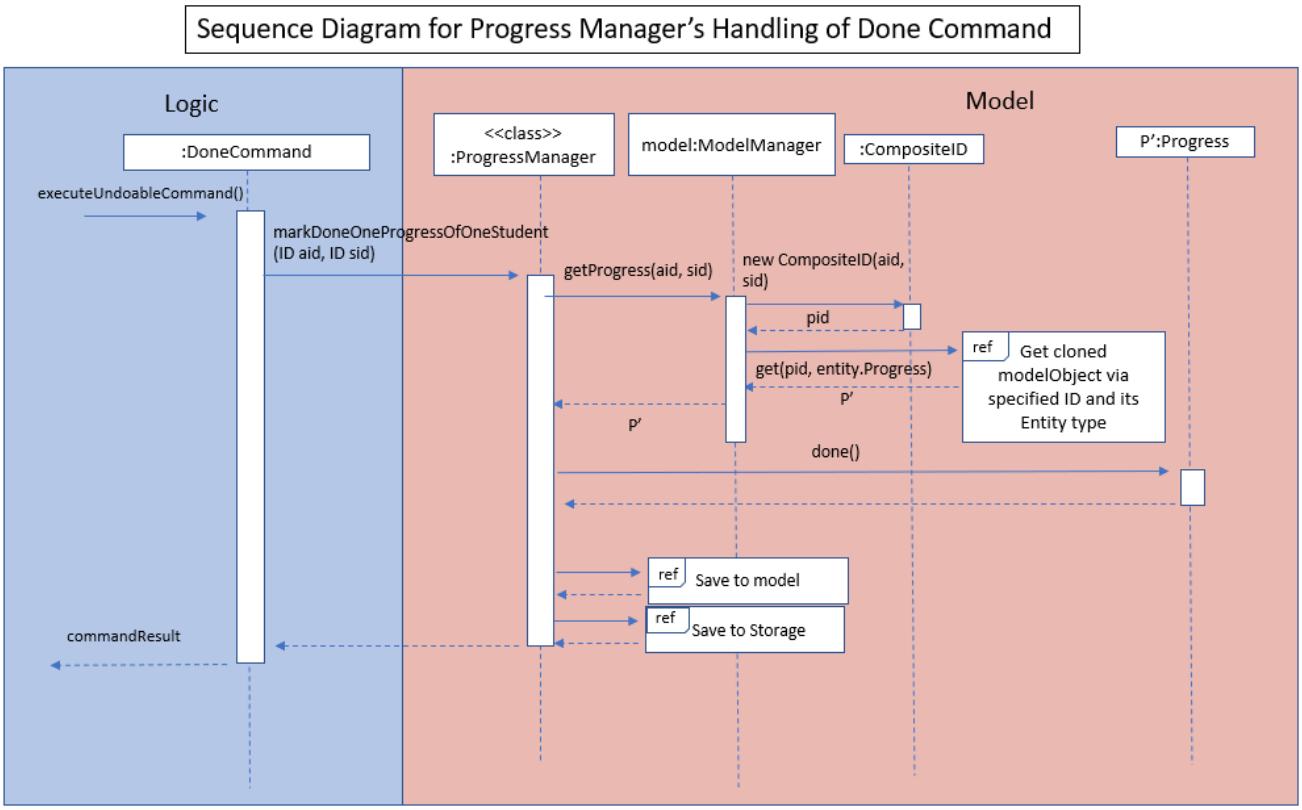


Figure 10. Sequence Diagram for Progress Manager's Handling of Done Command

The implementation of `Done` command is very straight-forward. Only 2 parameter is needed - assignmentID (aid) and studentID (sid).

1. Create the CompositeID of `Progress` objects using the aid and sid
2. Using this newly created compositeID, call modelManager#get(ID id, entity.type)
    a. This returns a `copy` of the `Progress` object - P'.
3. Execute `P'#done()`
    a. If `P'` is already done, an exception will be thrown
    b. Else, `P'` will be set its internal boolean isDone to be true.
4. Save the edited `Progress` object to model
5. Save the edited `Progress` object to storage

With this, the implementation of `Undone` command is about the same, we just have to call `P'#undone()` instead.

<u>Design Consideration</u>

There is 1 main considerations when implementing `Progress` objects.

1. Is there a need to separate `Assignment` and `Progress`?
    1. Implementation 1 - Have `Assignment` hold 1 `StudentID` and a 1 `isDone` boolean

        Pros

        a. Simple to implement

        Cons

        a. Memory-intensive since every student can have up to N number of `Assignments`
        b. Suppose that a field in the `Assignment` needs to be updated, the program needs to loop through every single `Assignment` object to update that particular field, resulting in high computational costs.
        c. With an additional link from `Assignment` to `Student`, it will be require more work to maintain the correctness of the linking.
            i. In `Section 2.3`, the decided implementation was to **centralize all links around `Course`**.
            ii. Hence, if another type of link was to be introduced, another `manager` will need to be implemented.

    2. Implementation 2 (Current Implementation) - Separate the logic of `Assignment` and `Progress`. `Assignments` just need to hold its ID, name and deadline while `Progress` will handle whether a `Student` has completed that `Assignment` or not.

        Pros

        a. Intuitive and simple to understand
            i. In-line with Object Oriented Programming since it can be modelled as a real world

object.

ii. As most people have been through school, they can understand that when given a homework/assignment in school, there is actually only **1** `Assignment` that **every** `Student` has to complete. This idea is basically what we have implemented.

b. Solves the first disadvantage of Implementation 1. Any time the details of the `Assignment` is changed, the details will be automatically changed for all `Progress` objects.

c. Works well with current implementation of AddressBookGeneric which has `getters` and `setters` via `ID` since every `Progress` object will have its own `ID`

Cons

a. Also very memory intensive

Opposite command for assign/ un-assign [Tee Jun Jie Ivan]

Generating of opposite commands for assign and un-assign commands is very intuitive. The opposite of assign is un-assign and vice versa.

The **tricky** part comes after you un-assign a Student/Assignment from a Course and have removed the affected `Progress` objects. When you want to undo the un-assign command, you need to add back those particular `Progress` objects which were just removed instead of adding new **undone** `Progress objects`. This is because those removed `Progress` objects may or may not be `done`.

This is achieved by 3 simple, additional steps.

1. When pre-processing an un-assign command as per **`Step 1 of Section 2.2.2`**, you'll need to assign all `Progress` objects that are about to be removed to a variable.
2. When `GenerateOppositeCommand` is called, via an overloaded constructor, you will need to instantiate a new `Assign` using the `Progress` objects that you have saved:

```
public AssignAssignmentToCourseCommand(AssignDescriptor assignDescriptor, Set
<Progress> undoProgresses)
```

   a. This allows the opposite command to add back the removed `Progress` objects

3. Finally, when `executeUndoableCommand` is executed, seeing that the undoProgresses is not null, the `Assign` Command will add those `Progress` objects back. Please see the activity diagram below for a better understanding of when the Undo Progress will be added back in.

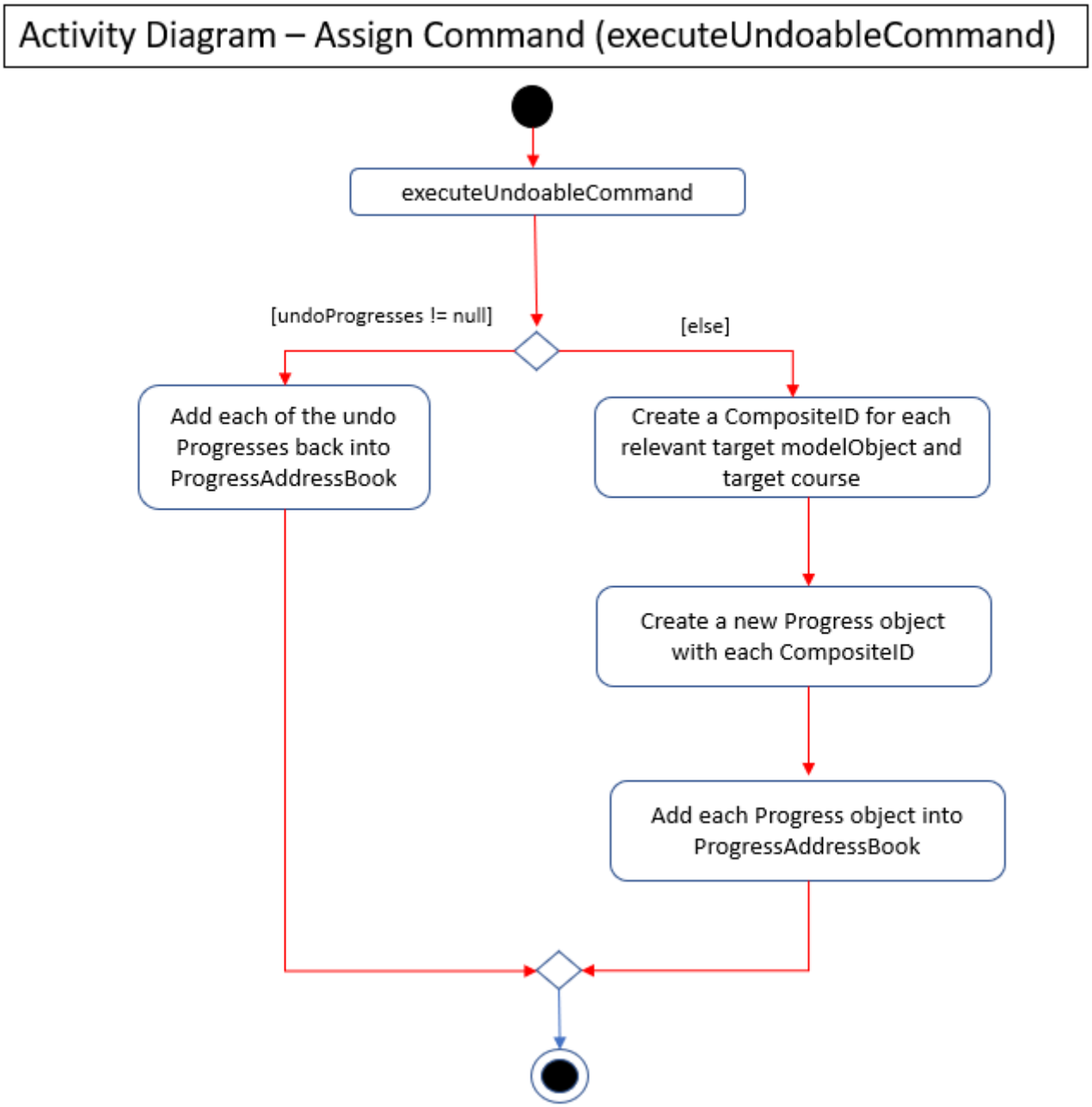


Figure 11. Activity Diagram of executing executeUndoableCommand of Assign Commands

This results in the **correct** `Progress` objects, which may or may not be `Done`, to be added back in instead of completely new `Progress` objects that are all `Undone`.

You can also notice that this is a faster implementation since we do not need to re-create a `CompositeID` and the actual `Progress` object itself when we are just adding back the UndoProgresses.