Revamp Reslife

CSCI 4448 Project Part 6

Jason Hill
Aubree Lytwyn
Shane Sarnac
Eric Thompson

Completed Use Cases

ID	Description	Priority
USR-008	As an RA, I need to complete a resident's check-in upon moving to my floor.	High
USR-009	As an RA, I must complete the checkout process when a resident leaves my floor.	High
USR-012	As a CA, I must be able to check out an item to a Resident.	High
USR-013	As a CA, I must be able to check in an item after a resident returns the item.	High
USR-014	As a CA, I must be able to issue a temporary key to a resident.	High
USR-015	As a CA, I must be able to reinstate the original key once the temporary key is returned.	High
USR-022	As a Resident, I need to be able to request a temporary key to be made when I am at the front desk.	High
USR-023	As a Resident, I need to be able to return a temporary key that I checked out.	High
USR-026	As a Resident, I need to be able to modify my Room Condition Form during my approved move-in period.	High
USR-027	As a Resident, I need to submit my Room Condition Form when I have completed it.	High

Incomplete Use Cases

ID	Description	Priority
USR-001	As an RA, I must be able to modify the Room Condition Forms for any room.	Medium
USR-002	As an RA, I must be able to confirm that the Room Condition Forms are complete	Medium
USR-003	As an RA, I want to view the Room Condition Forms completed by the resident.	Medium
USR-004	As an RA, I want to view my residents' roommate agreements	Medium
USR-005	As an RA, I want to be able to modify my residents' roommate agreements	Low
USR-006	As an RA, I would like to modify resident 121's.	Medium
USR-007	As an RA, I would like to view the status of my residents' 121's	Medium

USR-010	As an RA, I would like to be able to email my residents.	Medium
USR-011	As an RA, I would like to view a roster of all my residents	Medium

USR-016	As a CA, I need to be able to make keys for residents moving into the building	Medium
USR-017	As a CA, I need to be able to remove access from residents moving away from the building	Medium
USR-018	As a CA, I need to be able to make a key for a resident moving between rooms within the building.	Medium
USR-019	As a CA, I need to be able to remove access from a key once a resident has completed a move between rooms within the building.	Medium
USR-020	As a Resident, I can view a list of items that I can	Low

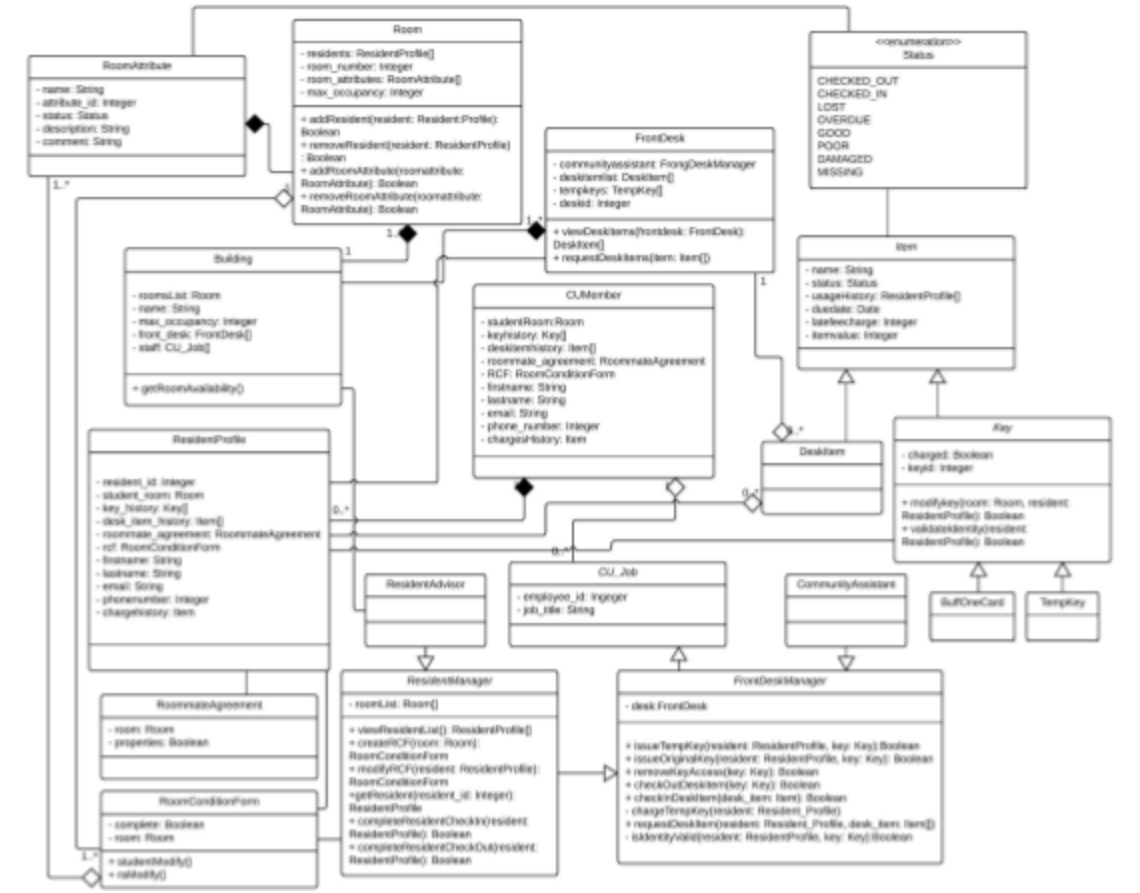
	check out from the Front Desk in my building	
USR-021	As a Resident, I want to be able to request a temporary key to be made when I am not at the desk.	Low

USR-024	As a Resident, I want to be able to request that an item at the front desk be checked out to me.	Medium
USR-025	As a Resident, I want to be able to see what desk items I have checked out to me and when I must turn the item back in by.	Low

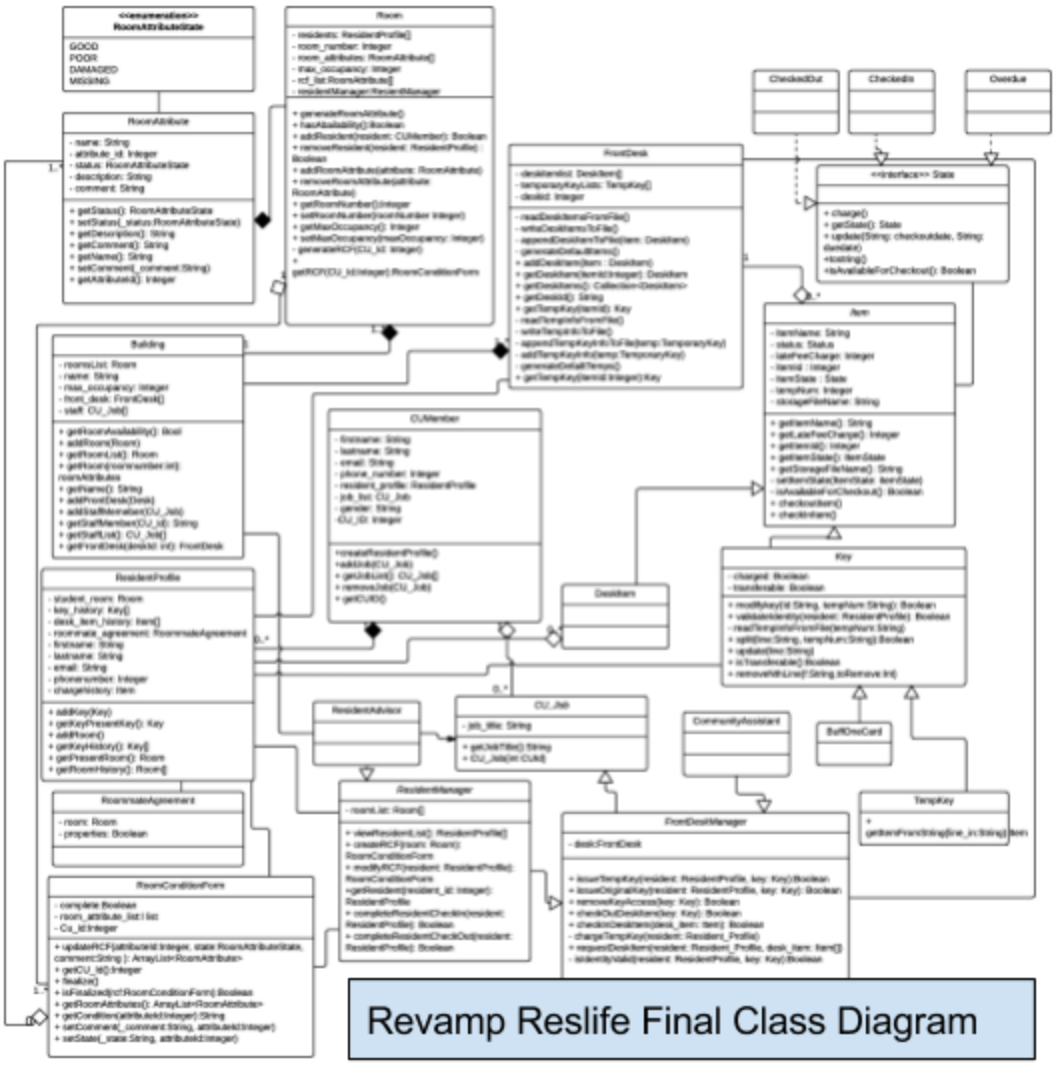
USR-028	As a Resident, I need to be able to complete my Roommate Agreement when I have a roommate.	Medium
USR-029	As a Resident, I would like to request a room move from my current room to a specific room type in a residence hall that I prefer.	Low

Class Diagram Evolution

Part 2 class Diagram



Revamp Reslife Class Diagram From Part 2

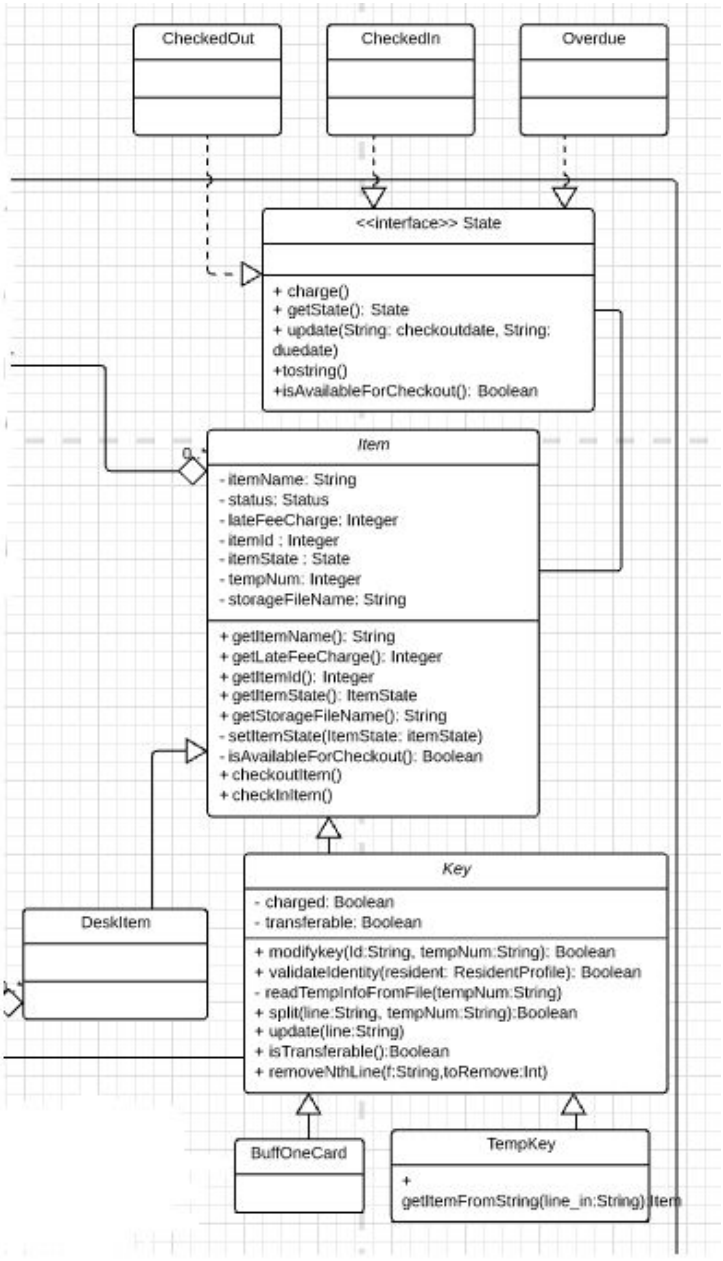

Revamp Reslife Final Class Diagram

If nothing else, working on this project highlighted the importance of developing quality class diagrams prior to coding. As evidenced by the progression between the two diagrams above, the class diagram remained dynamic throughout the project, as attributes and methods were added or modified as needed by the implementation. For instance, the *RoomConditionForm* class required the addition of a boolean variable *finalized* and corresponding getter method *isFinalized* to meet the requirements outlined by the use case for completing a Room Condition Form. Other modifications included altering the relationship between the *ResidentProfile* and *Room* classes by removing the list of resident profiles from the *Room* class, adding methods to the *ItemState* interface for returning a string denoting the state implementing the interface and whether the item can be checked out, and separating *ItemState* from *RoomAttributeState* for better comprehension.

Despite these changes, the overall structure of the class diagram changed very little. This demonstrates how the large amount of preparation we put in early aided us in developing the code later. The sheer complexity of our project guaranteed that changes would need to be made; however, the classes representing the key components of the Residence Life organization persisted throughout the project, properly enveloping the attributes and methods necessary to interact with those components, such as tracking *FrontDesk* objects in the *Building* class based on where a given desk resides. These well-thought-out components of the class diagram helped to create placeholder classes as the application was under development and provided a means to reference methods and attributes that were needed for the selected use cases.

Design Patterns

As we analyzed the relationships between the various documents used in the Residence Life organization, particularly the logs used to record the state of desk items and keys issued to residents, it became clear that the State design pattern would provide the means to encapsulate these relationships. The classes represented in the class diagram below represent our implementation of the State design pattern. Namely, we utilize the *ItemState* interface to define the methods performed in the *CheckedIn*, *CheckedOut*, and *Overdue* item states. These item states are then stored in the abstract class *Item*, giving common states for *DeskItem* and *Key* objects.



Conclusions

One thing that Revamp ResLife has taught the group is that designing the system never ends. While implementing our design we ran across many issues in our class diagram. For example, our original class diagram was designed so that a “Room” object did not have a “Building” attribute. We thought it would be sufficient for each building to hold a list of rooms. As we continued to design the system it became apparent that it would be very useful to be able to query the room object with a “getBuilding()” method to show where a room was.

We found that consistent restructuring of the class diagram was a necessity. The full complexity of the system only became apparent during the implementation. Left to our own devices, the individual coders had a tendency to create an increasing amount of coupling between classes. I think we all left with a greater understanding of why it is important to design the system to increase cohesion and reduce coupling. We ran into several situations where our poor designs meant that fixing one aspect of a view or a controller meant that changes had to propagate through a half dozen different classes.

If we could do it all over again, we would start by designing the fundamental classes carefully without getting distracted by secondary functionality. One problem with our approach was that we attempted to write out a thorough plan for 20+ different classes, and it turned out that it was too much complexity for us to figure out in our heads.

At its core, our system keeps track of users, and models the relationship between rooms that are occupied and unoccupied. If we had designed these 5-10 foundation classes from the start, it would be much easier to extend functionality later. As it was, we got lost in the weeds worrying about how to model the relationships between buildings and front desks, and how to inherit job functionality between RAs, CAs, and residents, or how desk items, keys, and temporary keys should be arranged under the “Item” umbrella. The lesson here is that you need to bite off smaller chunks of the problem. You can’t try to take it all on at once.

We also appreciated static type checking in Java because it allowed us to reason about the interactions between classes/functions. We could write functions in one class and be perfectly confident that its inputs and outputs would not create issues in other functions/classes because we knew the return types and parameter types. The debugging process is much faster when you can eliminate non-deterministic typing issues as the source of the bug.

Speaking of debugging, it seemed that our project was getting to the threshold where writing unit tests would save us time as we continued to extend the functionality. We made no attempt to write unit tests for our functions/classes, and in the short term that saved us a lot of time. If our project stayed at the same size it is, it might not be worth writing tests. But in reality we probably only built out 10% of the code base that would be necessary to release our project as a real product. As the scale of our project continued to grow we would appreciate unit tests more and more.