A Simplified Multiple Listing Service (MLS) System

Authors:

Mengya Song

Rui Min

Xixi Gao

Yang Lan

York University

EECS2030 Assignment 3

Date: July 25th, 2021

Table of contents

1. Introduction

The aim of the project is to design and develop a simple residential Multiple Listing Service (MLS) system with object-oriented principles in Java. The system should be able to create, read, update, and delete MLS records (CRUD operations). For simplification, txt files are used for storing records. This report consists of the following parts: MLS package, model package and additional resource package.

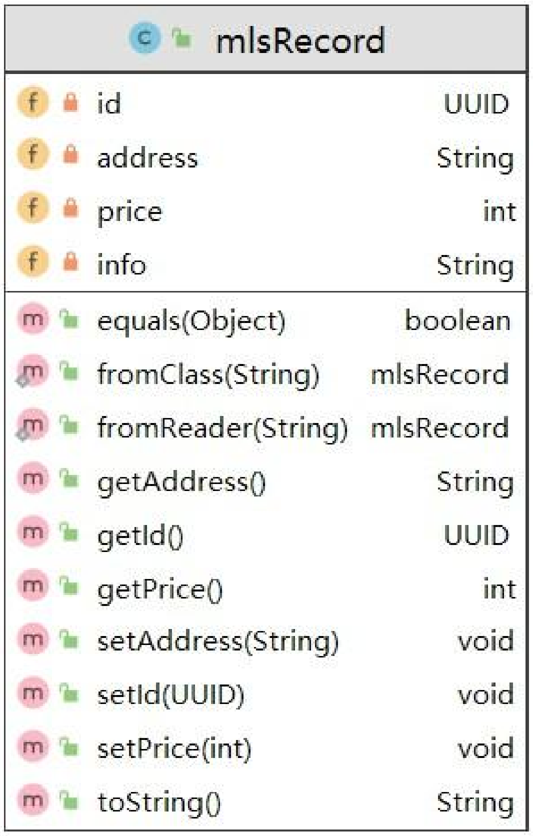
The MLS package is the body of the operational MLS classes, including mlsRecord for MLS object creation and handling, and RecBook for record txt file read/write and access cache management.

1) The mlsRecord class will bundle general property objects' complete information, and also provide a uniformed String representation of their unique attributes.
2) The RecBook class will provide multiton instances for sample testing and regular usage, with their own access cache and weight diminishing system depending on access behaviors. RecBook will also keep .txt records for any valid mlsRecord inputs, implementing record read/write/update/delete functionalities, and alteration of multiton attributes.

The model package is built for setting the hierarchy of different types of properties. Among the name list in the assignment file, "Lockers and storage", "Multi-generational", "Parking spaces" are designed as interfaces; "High-value homes", "Multi-family homes" and "New Constructions" are embedded inside classes as attributes; "Condominiums", "Freeholds" are written as abstract classes representing corresponding ownership types; and all other names are regarded as concrete classes. All concrete classes can be instantiated with builders.

The resource package contains mls.txt and mlsSample.txt for potential file manipulation and MLS system testing demonstration.

2. MLS package



mlsRecord	
id	UUID
address	String
price	int
info	String
equals(Object)	boolean
fromClass(String)	mlsRecord
fromReader(String)	mlsRecord
getAddress()	String
getId()	UUID
getPrice()	int
setAddress(String)	void
setId(UUID)	void
setPrice(int)	void
toString()	String

2.1 mlsRecord (object)

The purpose of mlsRecord class is to help construct an object to represent a real estate property's characteristics, including the following 4 attributes:

- UUID id of the entry
- Address
- Price
- Other information of the property, including parking space, multi-generation, construction date etc.

Construction of these attributes is by utilising uniformed overridden *toString()* methods in model package classes, or by reading from existing txt record file, with help of its own builder pattern:

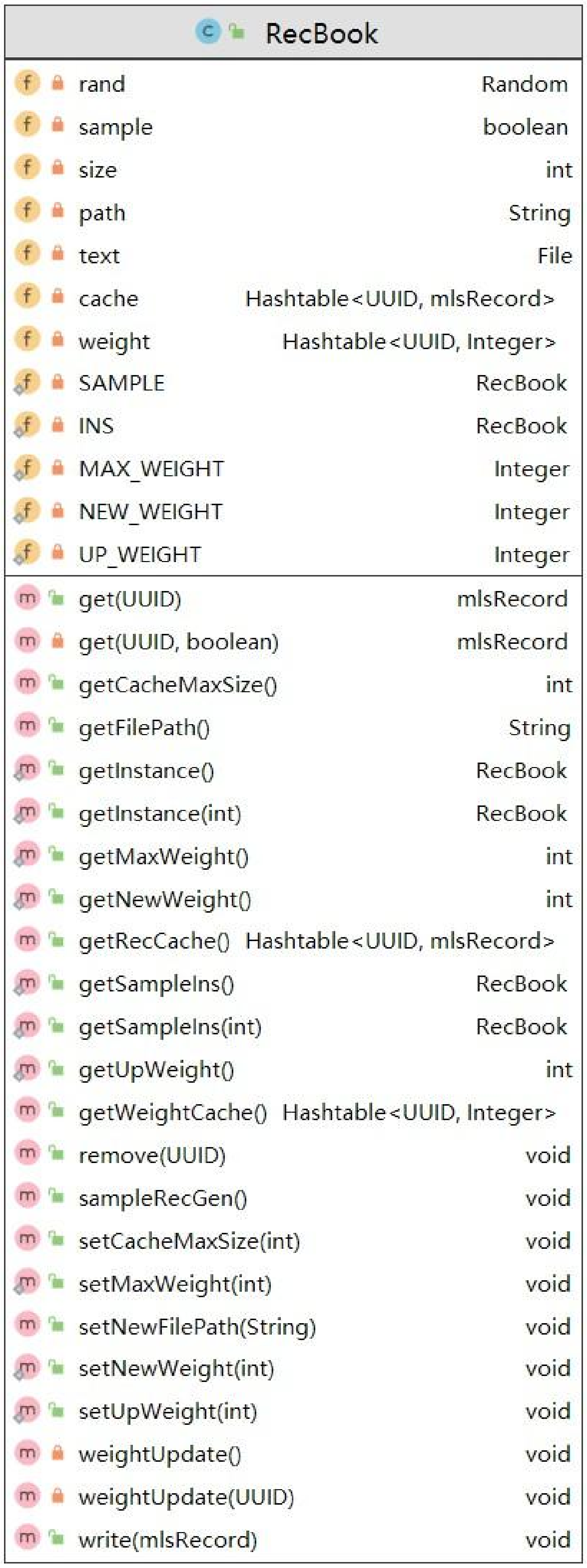
a) *fromClass(String)* is used for property *toString()* inputs, that splits raw string representation of a property object into above fields and then uses builder pattern to complete the process.
b) *fromReader(String)* is used for record txt file read record input to construct mlsRecord object. All file entries are uniform in format as "UUID && Address && Price && Info".

Overridden Object methods like *toString()* and *equals(Object)* realise uniform string representation and reality comparison of mlsRecords:

a) *toString()* creates above "UUID && Address && Price && Info" as string representation of a mlsRecord object. The result of this method and function of *fromReader(String)* method are interconvertible.
b) *equals(Object)* helps to compare if 2 mlsRecords are the same, by checking whether all 4 fields of them equal each other. This function partially reflects the reality check of 2 properties.

Additional getters and setters are present for id, address and price read/change.

2.2 RecBook (Multiton, record r/w)



RecBook	
rand	Random
sample	boolean
size	int
path	String
text	File
cache	Hashtable<UUID, mlsRecord>
weight	Hashtable<UUID, Integer>
SAMPLE	RecBook
INS	RecBook
MAX_WEIGHT	Integer
NEW_WEIGHT	Integer
UP_WEIGHT	Integer
get(UUID)	mlsRecord
get(UUID, boolean)	mlsRecord
getCacheMaxSize()	int
getFilePath()	String
getInstance()	RecBook
getInstance(int)	RecBook
getMaxWeight()	int
getNewWeight()	int
getRecCache()	Hashtable<UUID, mlsRecord>
getSampleIns()	RecBook
getSampleIns(int)	RecBook
getUpWeight()	int
getWeightCache()	Hashtable<UUID, Integer>
remove(UUID)	void
sampleRecGen()	void
setCacheMaxSize(int)	void
setMaxWeight(int)	void
setNewFilePath(String)	void
setNewWeight(int)	void
setUpWeight(int)	void
weightUpdate()	void
weightUpdate(UUID)	void
write(mlsRecord)	void

2.2.1 Multiton instance access

RecBook adopts multiton design and provides 2 instances: *INS* and *SAMPLE*, with the same functionality but different operable txt files and randomiser.

Initialisation of RecBook Singleton/Doubleton instances. Users can initialise and get an instance of RecBook by invoking *getSampleIns(int)* or *getInstance(int)*, where integer input determines the limit of cached entries. Or instead they just call *getSampleIns()* or *getInstance()* for default instances with 64(sample) or 128(formal) cache limit.

Later calls of above 4 getters will not affect the initialized cache limit, except for direct cache setter call *setCacheMaxSize(int)* to only increase this limit.

Additional setters and getters for cache and weight related parameters are present. Note that due to simplicity, the new cache size can only be larger than the original value.

2.2.2 Cache access & weight system

As mentioned above, cache access happens when users invoke *get(UUID)* and a valid record is found, either in existing cache entries, or record txt file. Instead of a queued cache, here we demonstrate a behavior-dependent cache-synchronised weight system. Rules are as follows:

1) Whenever a valid entry is found in the record file, not only instance's mlsRecord cache will update this new entry, its weight cache will also get a new entry with the same UUID key and *NEW_WEIGHT* value.
2) Whenever a cached entry is requested, that entry's weight will increase by *UP_WEIGHT* and up to *MAX_WEIGHT*.
3) All valid entry retrieval will decrease the weight of irrelevant entries by 1.
4) When invoking *remove(UUID)*, corresponding entry in mlsRecord cache and weight cache will be removed. No updates on other entries' weight.

Additional setters and getters for weight related parameters allow changes to the weight system. *MAX_WEIGHT* can only be larger than the original value, and new *NEW_WEIGHT* should always be smaller than the current *MAX_WEIGHT* value.

2.2.3 Record file read/write/update functions(2-6)

Essential record manipulation methods available for a RecBook instance includes:

a) *get(UUID)* method is used to retrieve a mlsRecord if it is either in cache or existing mls(Sample).txt record file. If the record is in cache, return directly, else try to search line by line in record for a match, then use the mlsRecord constructor to build and return a proper mlsRecord object. If no record is found, return null. A hidden *get(UUID, boolean)* method is called, where boolean input controls only the cache put/retrieve functionality.

b) *write(mlsRecord)* method is used to directly write a mls object into record text. It also provides a duplication check. If the given mlsRecord *equals* one of the existing entries in record txt, no update is needed. If the given mlsRecord has a fresh UUID, or some record-duplicate UUID but different field values, append or replace existing record for the new entry.

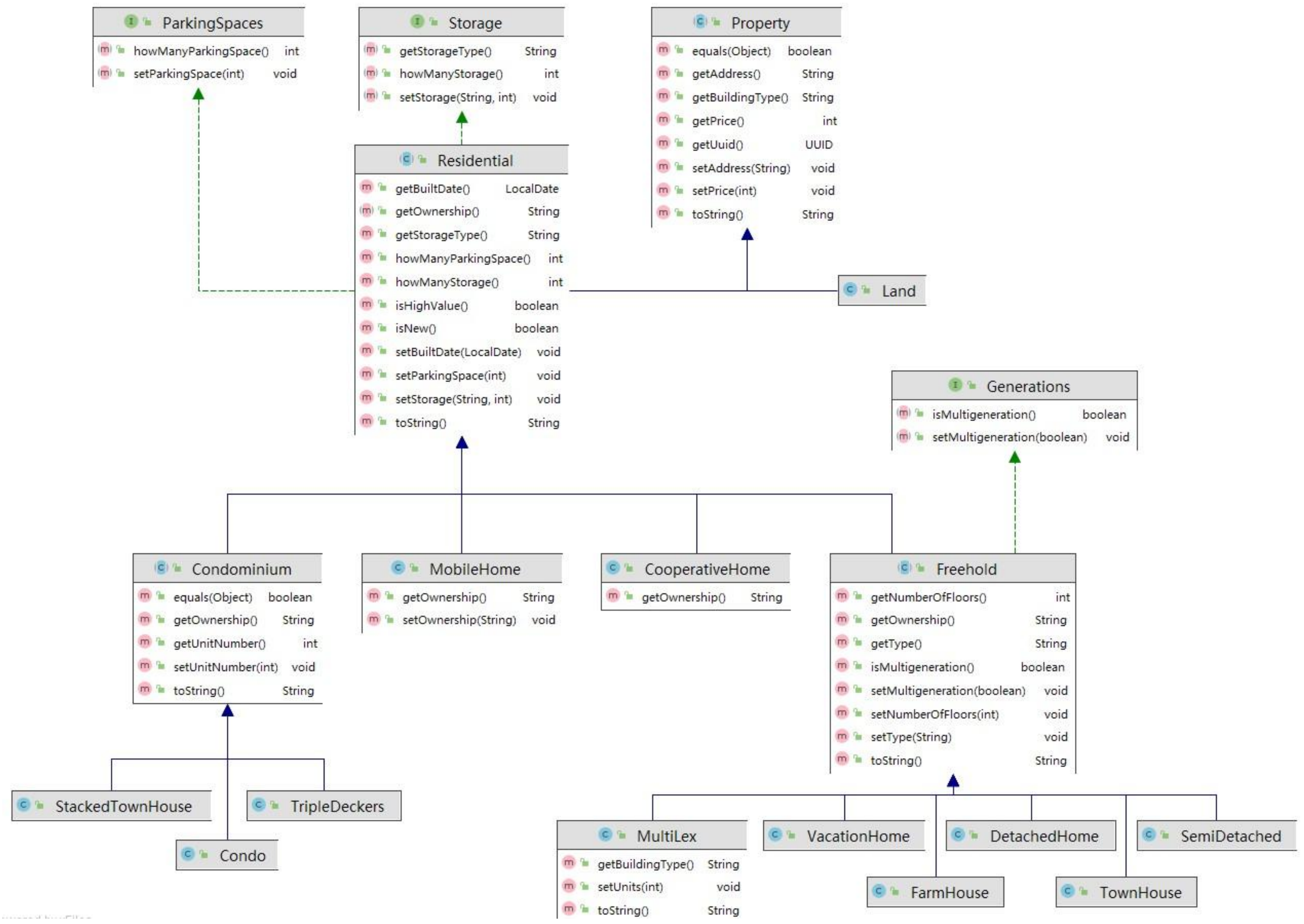
c) *remove(UUID)* method provides the ability to remove a record corresponding to the designated UUID. It also removes any corresponding entry in the instance cache or weight system. If no such entry exists, do nothing.

d) *setNewFilePath(String)* method can relocate instance specific record txt file to a designated file path.

Equipped with above methods, users can easily manipulate property data and quickly access history search. File management and weighted cache system tests are present in “RecBookTest” class.

3. Model Package

Below is the java class hierarchy of property classes:



3.1 Abstract Classes

3.1.1 Property Class

Property is the only level 1 class of the whole model package. It is an abstract class representing the idea "Property", which contains both residential properties and other types. It extends java.lang.Object, and has all other classes as its descendants.

This class declares three new private attributes: uuid, address, and price. These are attributes common to all model package classes, and descendants manipulate the private attributes through public constructors, setters and getters. All concrete descendants should be instantiated with the three attributes in the beginning.

getBuildingType() method is declared to return the current class's name in simple format.

There are two overridden methods discussed as follows:

1) *equals(Object)*: For simplification, two properties are regarded as the same property if they have the same address attribute.
2) *toString()*: It is mandatory to start with below format because MLS package needs the format for database(.txt file) manipulation:

```
String.format("%s{uuid=%s, address=%s, price=%d}",
    this.getBuildingType(), this.getUuid(),
    this.getAddress(), this.getPrice());
```

JUnit4 testing for the current class is done. The testing strategy is to test all new or overridden methods. Because abstract classes can not instantiate, two concrete subclasses are utilised as run-time types to facilitate the process. No error found.

3.1.2 Residential Class

This is an abstract class that specifies the common attribute and methods of all subclass objects extending it. Since it is an abstraction with partial definition of a residential property, this class cannot be instantiated.

The API of this class contains a field, a constructor and eleven methods.

The field *REFER_PRICE* is the reference value used in the method *isHighValue()* where the price of the calling Residential object is compared with *REFER_PRICE***.** If the object's price is higher, the method will return true; otherwise, false. This method is designed to facilitate MLS to manipulate the high value residential properties. The system can easily filter all high value homes and play with them via this method.

The *isNew()* method will return a boolean value indicating whether an object of type Residential or its descendants is a new construction. It achieves this intent via calculating the time difference, measured in year, between the buildDate attribute of the calling object and current date when it is called. If the difference is more than 5 years, it will return false; otherwise, true. This is designed to meet the professor's minimum requirement that our MLS should consider **new constructions.** Taking advantage of this method, our MLS can easily find all records that are new constructions and apply any operations on them.

The overridden *toString()* method is intended to generate a string representation that contains all details of the calling residential property object. If called by any subclass object, in the body of this method another method defined in Residential class, *getOwnership()*, will be called. Since *getOwnership()* is an abstract method, its definition is completed by the subclasses. Hence, the behavior of *getOwnership()* is determined in the **run-time** corresponding to the **run-time type** of the object by **polymorphism** and **dynamic dispatch**. Different residential objects calling the *getOwnership()* method will return different strings that describe the calling object's type of ownership in our simplified hierarchy of properties.

For the details and rest methods defined in this class or inherited from its parent class "Property", please consult the java documentation.

Tests:
In **ResidentialTest.java,** 6 separate tests are used to test the methods related to built date, parking space, storage, high value, ownership, and building type. In the other 3 tests, the builders of some subclasses are tested along with the *toString()* method. All tests are passed.

3.1.3 Freehold Class

Freehold is the subclass of Residential, it inherited all the methods from its superclass. Meanwhile, it is the superclass of DetachedHome, FarmHouse, FreeHold, MultiLex, SemiDetached, TownHouse, VacationHome. To make it impossible to be initialized, FreeHold is an abstract class.

FreeHold is an object with private new added attributes type, numberOfFloors, and isMutigeneraion. There are constructor public Freehold(UUID uuid, String address, int price) and methods to get or set all these three attributes.

FreeHold implements the interface Generation, abstract *isMultigeneration()* and *setMutigeneration(boolean)* methods are implemented inside FreeHold class.

Finally, The *toString()* method is overridden to generate a string representation that contains all details of the calling FreeHold object, it uses super.toString to simplify the coding.

All Implemented Interfaces:
ParkingSpaces, Storage, Generation

Direct Subclasses:
DetachedHome, FarmHouse, FreeHold, MultiLex, SemiDetached, TownHouse, VacationHome

Test cases:
In FreeHoldTest.java, we used 5 junit tests to test the methods related to multi-generation and it's new getter and setter methods. In the last test, we test the builders of some subclasses and *toString()* method together.

3.1.4 Condominium Class

Condominium is one of the level 3 classes. It is an abstract class representing the idea "Condominium", which contains all building types with condominium ownership. It extends the Residential abstract class, and has three concrete subclasses.

This class declares one new private attribute: unitNumber, which represents the unit number of the condominium type property. This is the attribute common to all its subclasses, and subclasses manipulate the private attribute through public setters and getters. Its only constructor is inherited from superclass.

There are three overridden Object methods

1) *getOwnership()*: It is an overridden method to return the string "Condominium", which indicates the ownership type.
2) *equals()*: For simplification, two condominium type properties are regarded equal if they have the same address and unitNumber attributes.
3) *toString()*: it is mandatory to start with the format defined in Property class because the MLS package needs the format for database(.txt file) construction.

JUnit4 testing for the current class is done. The testing strategy is to test all new or overridden methods. Because abstract classes can not instantiate, all its concrete subclasses are utilised as run-time types to facilitate the process.

3.2 Interfaces

3.2.1 ParkingSpaces Interface

This interface describes what methods an object must have to deal with parking spaces. It contains two methods:

1. *howManyParkingSpace()***;**
   This is a getter returning an int value of the number of parking spaces.
2. *setParkingSpace(int howMany)***;**
   This is a setter that sets the number of parking spaces belonging to the object.

Implementing classes: Residential and its subclasses.

3.2.2 Storage Interface

It is implemented by the Residential class. So any concrete subclass object can use the methods declared in the interface and implemented in the Residential class.

Two getters and one setter "*int howManyStorage()*", "*String getStorageType()*" and "*void setStorage(String, int)*".

3.2.3 Generations Interface

Generations is an interface to represent how many generations of a family are living in one placeGeneration Interface has two methods:

1) *isMultigeneration()*: return boolean value of whether the object is multi-generational.
2) *setMutigeneration(boolean)*: set the value of the *isMultigeneration()* of the specific object.

Since FreeHold is the superclass of other classes, all its subclasses inherit the implemented method of Generation.

3.3 Concrete Classes

3.3.1 Land Class

This class defines a simple subclass of “Property”. It contains only inherited (UUID, address, price) fields, representing an empty land property with no onsite realty.

3.3.2 CooperativeHome Class

This class defines a subclass of “Residential”. It is a basic example of property ownership in mid-late 20th century, that except for obvious fields inherited from “Residential” abstract class, no additional attributes are required.

3.3.3 MobileHome Class

This class also defines a subclass of “Residential”. It is also a basic residential unit like cooperative homes. But in reality it does have unique ownership, and doesn’t require a fixed address.

3.3.4 MultiLex Class

This is a concrete class defining the template of a multilex property in our MLS. There is a nested inner class Builder in it that assembles the object of the MultiLex class. As its instance represents a multilex home, the constructor in this class will call the *setType()* inherited from the Freehold class to initialize its type as **“**Multi-Family”.

There is a private int attribute defined in this class which denotes how many units are there in a multiplex house. There is a *getBuildingType()* method overriding the same method defined in Property class, which returns specific multiplex types corresponding to the number of units the multiplex object has. The *toString()* method inherited from Freehold is also overridden here to add the class specific information — the number of units of the calling Multilex object. Please find more details in the java documentation in the project files.

Tests for this class is covered by ResidentialTest and FreeholdTest

3.3.5 DetachedHome Class

This is a concrete class defining the template of a detached home property in our MLS. There is a nested inner class Builder in it that assembles the object of DetachedHome class. Please find more details in the java documentation in the project files.

Since there is no class specified method and attribute, tests for this class are completed by ResidentialTest and FreeholdTest.

3.3.6 SemiDetached Class

This is a concrete class defining the template of a semi detached home property in our MLS. There is a nested inner class Builder in it that assembles the object of SemiDetached class.

Since there is no class specified method and attributes, tests for this class are completed in ResidentialTest and FreeholdTest.

3.3.7 FarmHouse Class

This is a concrete class defining the template of a FarmHouse property in our MLS. There is a nested inner class Builder in it that assembles the object of FarmHouse class.

Since there is no class specified method and attributes, tests for this class are covered by ResidentialTest and FreeholdTest.

3.3.8 TownHouse Class

This is a concrete class defining the template of a TownHouse property in our MLS. There is a nested inner class Builder in it that assembles the object of TownHouse class.

Since there is no class specified method and attributes, tests for this class are covered by ResidentialTest and FreeholdTest.

3.3.9 VacationHome Class

This is a concrete class defining the template of a VactionHome property in our MLS. There is a nested inner class Builder in it that assembles the object of the VactionHome class.

Since there is no class specified method and attributes, tests for this class are covered by ResidentialTest and FreeholdTest.

3.3.10 Condo Class

Condo is one of the level 4 classes. It is a concrete class representing the building type "condo" of the property. It extends the Condominium abstract class, and has no subclasses. There is a nested "Builder" class inside it.

This class declares no attribute and its only constructor is inherited from superclass, with uuid, address and price as mandatory fields. There are no overridden methods.

The concrete class is designed with "Builder Pattern" to facilitate its construction. Inside the nested Builder class, uuid, address and price are mandatory attributes. And unitNumber, howManyParks, howManyLockerStorage, storageType and builtDate are optional attributes. "*public Condo build()*" method helps build Condo objects with its attributes specified or by default.

JUnit4 testing for the current class is done. The testing strategy is to test normal constructor and the builder pattern. Further, the inherited equals() method is also tested. No error found.

3.3.11 TripleDeckers Class

TripleDeckers is one of the level 4 classes. It is a concrete class representing the building type "triple deckers" of the property. It extends the Condominium abstract class, and has no subclasses.

All its contents are similar to the Condo class. Please refer to the Condo class section.

JUnit4 testing for the current class is done. The testing strategy is to test normal constructor and the builder pattern. Further, the inherited equals() method is also tested. No error found.

3.3.12 StackedTownHouse Class

StackedTownHouse is one of the level 4 classes. It is a concrete class representing the building type "stacked townhouse" of the property. It extends the Condominium abstract class, and has no subclasses.

All its contents are similar to the Condo class. Please refer to the Condo class section.

JUnit4 testing for the current class is done. The testing strategy is to test normal constructor and the builder pattern. Further, the inherited equals() method is also tested. No error found.

4. Conclusions

In this project, a simple MLS system is built based on our MLS and model package. The system can create, read, update, and delete MLS records stored in txt files under the “resources” folder. A RecBook test is presented to demonstrate mentioned multiton and file operations. Basic tests on functionality such as file manipulation and multiton instance weighted cache system are passed. Manual deletion of mlsSample.txt content prior to testing is required for a pass in test.

The hierarchy of our property classes is designed according to discussion and online searching. In the real world, by looking into Realtor.ca, we found most multilex homes, detached homes and so on are Freehold homes in terms of ownership, and most condo, stacked townhouse, triple decker belong to Condominiums. Therefore, we made the classification of residential properties by ownerships in level 3 of our hierarchy, assuming the properties represented by the level 4 classes have fixed type of ownerships, for simplicity. That is, in our design of hierarchy, we ignored the rare cases of unconventional ownerships such as a townhouse being a Condominium or a triple decker being a Freehold.

All properties required are considered. Practical use of encapsulation, inheritance, abstraction, and polymorphism is displayed in the model package. Both UML diagrams and Java documentation are created under the “docs" folder. All these lead to our best practices of object-orientated principles in Java.

5. References

1. Guide to UUID in Java
2. JavaDoc BufferedReader
3. JavaDoc BufferedWriter
4. JavaDoc Files
5. How to Read and Write Text File in Java
6. Delete a line from a file using java