UNIVERSITÉ Concordia UNIVERSITY

Little Computer People

SOEN-6971

PROJECT AND REPORT

PROJECT REPORT

NAME	ID
URMIL KANSARA	40037747
TEJAS SADRANI	40041572
VAIBHAV JAIN	40046891

1. Abstract

Design patterns is a tricky concept, a software practice that is found to solve known problems and can be used in many applications having the same intent. However it is tricky to figure out the intents as there is always a scope to use more than one design pattern for a same problem. Here is an explanatory project work by building a simulation game, to give brief idea of all the design patterns that can be used in a single application with the main intent for organizing code thus improving software design, for a better future maintainability and scalability.

2. Introduction

Little computer people (LCP) is a simulation game where it represents fully furnished house on a computer. There is no winning condition and it shows the sideways view of the inside of a 3-story house. In this game, a person does the task that is carried out in daily routine of an individual like cooking, sleeping, bathing, eating, playing, office, etc. After the game starts an animated character will move and roam around the house to perform the daily routine activities described above. Players can interact with this animated character and command him to do the particular task like watching TV, opening fridge etc. The house in which person moves and perform the tasks is shown in single screen in 2D. There is only one person and there can be pets based on the requirement of the game.

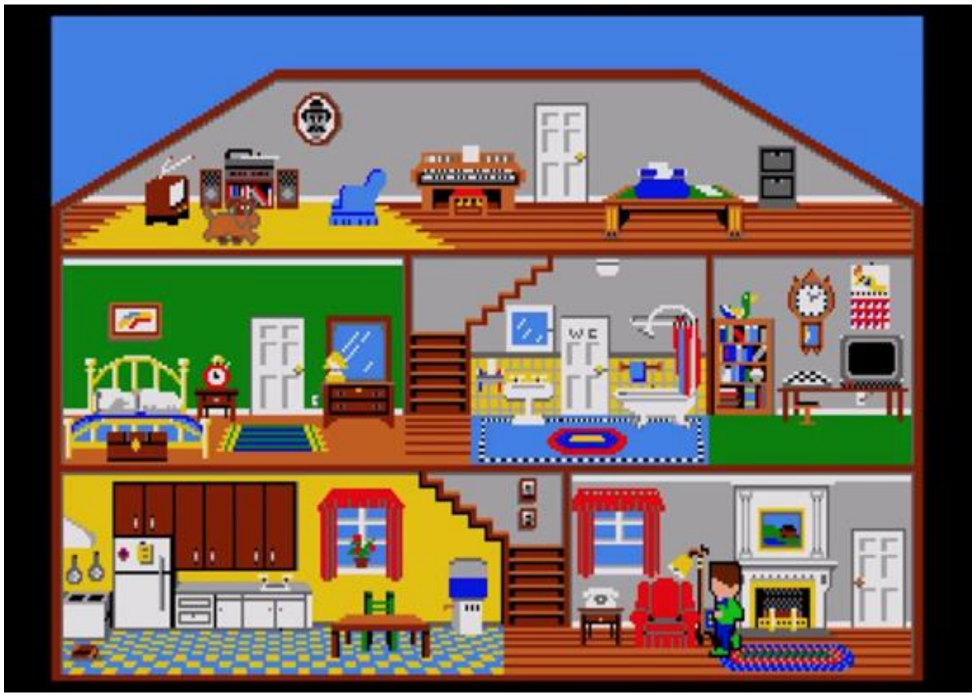


Figure 1

3. Motivation

The main idea behind developing this game is to implement all the design patterns in an agile way. Design patterns makes it easier to reuse successful designs and architectures by providing solutions to the common design problems. However, figuring out their applicability's depends on many factors may be related to the issue or the project we are building.

This project aims at figuring out the DP's in an articulate fashion, that can be used along with each other in conjunction to produce a better software design that considers many factors, mainly increasing abstraction for users and clients for whom the application is built.

4. Methodology

The simulation game under consideration is build by using core java concepts with design patterns to make a better implementation of the code. We have used java swings basic UI to render the simulation of persons movement based on various factors(a).

We build our code based on simple notion, a house is a composite object that carries various things inside out hence composite pattern. The composite pattern describes a group of objects that is treated the same way as a single instance of the same type of object(b). Hence a house has floors, floors has rooms and rooms have another furniture objects. A composite gives us a notion of house as a whole object.

Coming to building and constructing the house where simulation takes place on specific time intervals according to the documentation(a), here factory design pattern is used. Where a factory is a class that creates whole structure of composites and provides that to the client, leaving the burden of object creation to the system and not on the clients. Thus, clients just need to ask for an object of a specific kind and factory will deliver this object on the safe hands.

Singleton pattern is used where factory is used, a singleton DP is used when exactly one object is needed to coordinate actions across the system. So in here a factory object is such an object that all the clients can refer and can be used to make different objects of composite types. Thus for us a factory is a singleton object.

Observer design pattern is used for the runners, be it tester or a UI runner. In here, a observer which is essentially a client checks for addition for composites and if the composites are build one over the other or not. It is a push and pull model where clients push and pings the system if one object is added inside the other or not and there is when system pings back or notifies the client for the addition if it took place. Based on this client acts accordingly.

With an aim to build a simplistic version of a LCP simulation game, we implemented a Model View Controller (MVC) architectural design model along with Observer pattern. It was an effort to use extreme programming effort with iterative software development approach to make a modular design and deliver several working coherent modules in small increments or builds.

Model View Controller
Model View Controller design aims at decomposing an interactive system into three components, namely: Model, View and Controller.

Model - The model represents data and the rules that govern access to and updates of this data. In enterprise software, a model often serves as a software approximation of a real world process.

View - The view renders the contents of a model. It specifies exactly how the model data should be presented. If the model data changes, the view must update its presentation as needed. This can be achieved by using a push model, in which the view registers itself with the model for change notifications, or a pull model, in which the view is responsible for calling the model when it needs to retrieve the most current data.

Controller - The controller translates the user's interactions with the view into actions that the model will perform. In a stand-alone GUI client, user interactions could be button clicks or menu selections. Depending on the context, a controller may also select a new view -- for example, an action on a particular click event may result in rendering a completely new view.

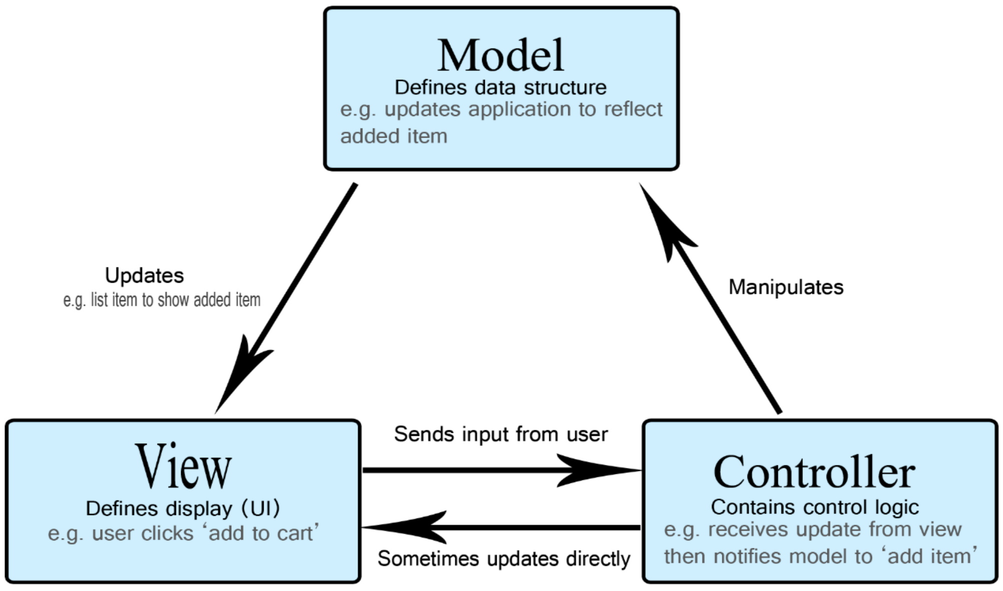

Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Figure 2

5. Design Decisions

As any challenging applications requires attention on design decisions, same for this, now and then there have been assumptions and decisions made for the code to be much cleaner and simpler for client's use.

Composite pattern

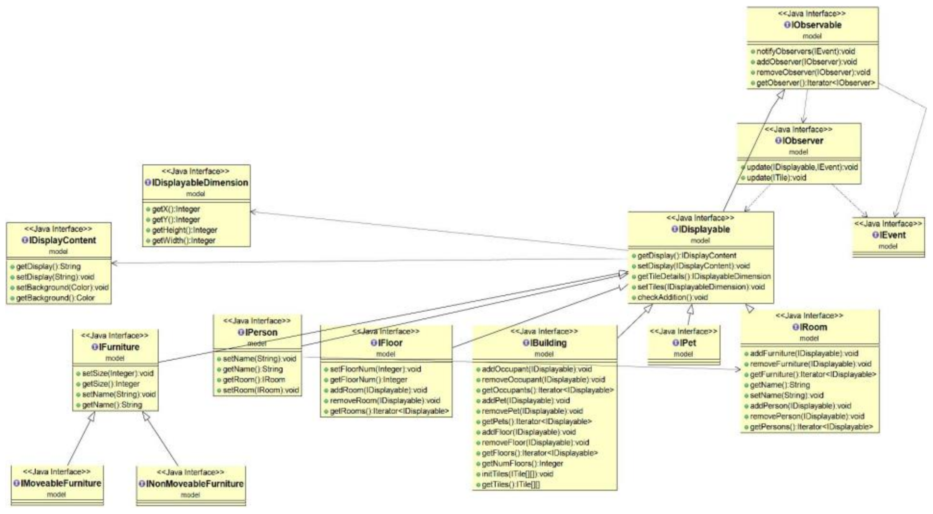
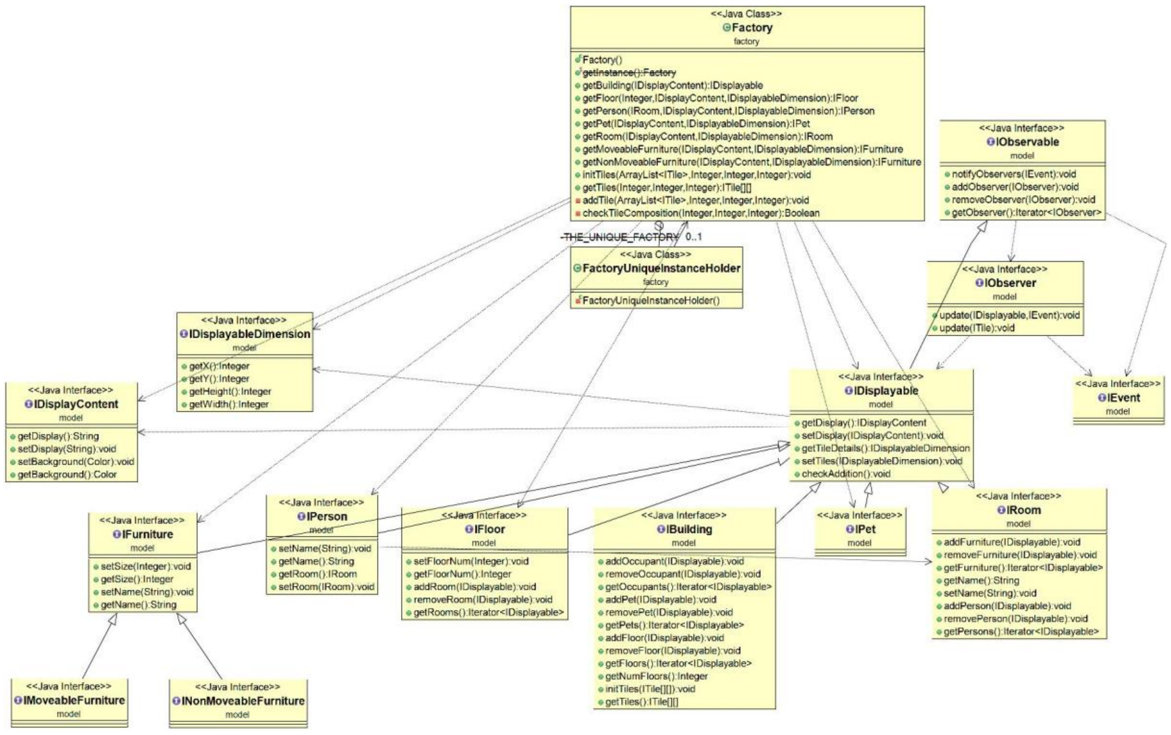

<<Java Interface>>
IObservable
model
notifyObservers(IEvent):void
addObserver(IObserver):void
removeObserver(IObserver):void
getObserver():Iterator<IObserver>
<<Java Interface>>
IObserver
model
update(IDisplayable,IEvent):void
update(ITile):void
<<Java Interface>>
IDisplayableDimension
model
getX():Integer
getY():Integer
getHeight():Integer
getWidth():Integer
<<Java Interface>>
IDisplayContent
model
getDisplay():String
setDisplay(String):void
setBackground(Color):void
getBackground():Color
<<Java Interface>>
IDisplayable
model
getDisplay():IDisplayContent
setDisplay(IDisplayContent):void
getTileDetails():IDisplayableDimension
setTiles(IDisplayableDimension):void
checkAddition():void
<<Java Interface>>
IEvent
model
<<Java Interface>>
IFurniture
model
setSize(Integer):void
getSize():Integer
setName(String):void
getName():String
<<Java Interface>>
IPerson
model
setName(String):void
getName():String
getRoom():IRoom
setRoom(IRoom):void
<<Java Interface>>
IFloor
model
setFloorNum(Integer):void
getFloorNum():Integer
addRoom(IDisplayable):void
removeRoom(IDisplayable):void
getRooms():Iterator<IDisplayable>
<<Java Interface>>
IBuilding
model
addOccupant(IDisplayable):void
removeOccupant(IDisplayable):void
getOccupants():Iterator<IDisplayable>
addPet(IDisplayable):void
removePet(IDisplayable):void
getPets():Iterator<IDisplayable>
addFloor(IDisplayable):void
removeFloor(IDisplayable):void
getFloors():Iterator<IDisplayable>
getNumFloors():Integer
initTiles(ITile[][]):void
getTiles():ITile[][]
<<Java Interface>>
IPet
model
<<Java Interface>>
IRoom
model
addFurniture(IDisplayable):void
removeFurniture(IDisplayable):void
getFurniture():Iterator<IDisplayable>
getName():String
setName(String):void
addPerson(IDisplayable):void
removePerson(IDisplayable):void
getPersons():Iterator<IDisplayable>
<<Java Interface>>
IMoveableFurniture
model
<<Java Interface>>
INonMoveableFurniture
model

Figure 3

As can be seen in figure 3, here there is one for all object named IDisplayable. IDisplayable is the composite object which can represent any object of the building. Here composite pattern is used for the raw objects or the model data objects in use. Viz. There is a building that comprises of rooms, person and pets, and rooms in turn comprises of furniture's of various kinds bifurcated into two categories namely, moveable and unmovable. All these objects are composites of type IDisplayable.

Factory Pattern and Singleton Pattern



<<Java Class>>
Factory
factory
Factory()
~~getInstance():Factory~~
getBuilding(IDisplayContent):IDisplayable
getFloor(Integer,IDisplayContent,IDisplayableDimension):IFloor
getPerson(IRoom,IDisplayContent,IDisplayableDimension):IPerson
getPet(IDisplayContent,IDisplayableDimension):IPet
getRoom(IDisplayContent,IDisplayableDimension):IRoom
getMoveableFurniture(IDisplayContent,IDisplayableDimension):IFurniture
getNonMoveableFurniture(IDisplayContent,IDisplayableDimension):IFurniture
initTiles(ArrayList<ITile>,Integer,Integer,Integer):void
getTiles(Integer,Integer,Integer):ITile[][]
addTile(ArrayList<ITile>,Integer,Integer,Integer):void
checkTileComposition(Integer,Integer,Integer):Boolean
~~-THE_UNIQUE_FACTORY~~ 0..1
<<Java Class>>
FactoryUniqueInstanceHolder
factory
FactoryUniqueInstanceHolder()
<<Java Interface>>
IObservable
model
notifyObservers(IEvent):void
addObserver(IObserver):void
removeObserver(IObserver):void
getObserver():Iterator<IObserver>
<<Java Interface>>
IObserver
model
update(IDisplayable,IEvent):void
update(ITile):void
<<Java Interface>>
IEvent
model
<<Java Interface>>
IDisplayableDimension
model
getX():Integer
getY():Integer
getHeight():Integer
getWidth():Integer
<<Java Interface>>
IDisplayContent
model
getDisplay():String
setDisplay(String):void
setBackground(Color):void
getBackground():Color
<<Java Interface>>
IDisplayable
model
getDisplay():IDisplayContent
setDisplay(IDisplayContent):void
getTileDetails():IDisplayableDimension
setTiles(IDisplayableDimension):void
checkAddition():void
<<Java Interface>>
IFurniture
model
setSize(Integer):void
getSize():Integer
setName(String):void
getName():String
<<Java Interface>>
IPerson
model
setName(String):void
getName():String
getRoom():IRoom
setRoom(IRoom):void
<<Java Interface>>
IFloor
model
setFloorNum(Integer):void
getFloorNum():Integer
addRoom(IDisplayable):void
removeRoom(IDisplayable):void
getRooms():Iterator<IDisplayable>
<<Java Interface>>
IBuilding
model
addOccupant(IDisplayable):void
removeOccupant(IDisplayable):void
getOccupants():Iterator<IDisplayable>
addPet(IDisplayable):void
removePet(IDisplayable):void
getPets():Iterator<IDisplayable>
addFloor(IDisplayable):void
removeFloor(IDisplayable):void
getFloors():Iterator<IDisplayable>
getNumFloors():Integer
initTiles(ITile[][]):void
getTiles():ITile[][]
<<Java Interface>>
IPet
model
<<Java Interface>>
IRoom
model
addFurniture(IDisplayable):void
removeFurniture(IDisplayable):void
getFurniture():Iterator<IDisplayable>
getName():String
setName(String):void
addPerson(IDisplayable):void
removePerson(IDisplayable):void
getPersons():Iterator<IDisplayable>
<<Java Interface>>
IMoveableFurniture
model
<<Java Interface>>
INonMoveableFurniture
model

Figure 4

Here, factory pattern is used in order to create instances of all other classes that represent entities in LCP such as floor, room, building, person etc. This class that acts as the factory class to instantiate other classes should not be instantiated itself twice. To fulfil that requirement, we used singleton pattern and restricted the instantiation of factory class at the time of initialization phase. This ensures the central control for creating instances in a execution environment and also results in efficient memory utilisation and garbage collection.

Factory class is responsible for creation of composite objects so that a level of indirection is made and all client knows is the api factory provides for creation of raw objects. Singleton pattern is used within Factory, so that clients can refer to only single object of factory class, because it is never required for any class to have more than one instance of factory. All client does is requests a static class factory by calling its method and factory in the end gives the object that client requires based on the permission.

MVC Runner

The game runner here is multithreaded as we have followed MVC architecture. View and models are the threads in this game. We are using Observer pattern to integrate this architecture. Logical aspect of the game is written in controller for example when a person moves it updates the model and model notifies the observers about the change.

As the figure 4 goes, this is how MVC was implemented. Various **Models** (Displayable,LCPDriver,DisplayDimension, etc) manages the behavior and data of the application domain. Once it gets a change state query request from the View (LCPView) that are registered to the model, they respond to instructions to change the state from controller (LCPController etc).

• Here we have built an event-driven system where the model notifies observers (usually views) which have been registered to the models, whenever there is change in information or state, so that they can react

Views on the other hand renders the model into a form suitable for visualization or interaction, in a form of UI (user interface). If the model data changes, the view must update its presentation as needed.
• Here we have developed a push and pull model where view registers itself with the model for the change notifications, thus following observer pattern.

Controllers are designed to handle input and initiate a response based on the event by making calls on appropriate model objects.Thus accept various input from the user and instruct the model to perform operations. • The controller translates the stimulated interactions with the view it is associated with, into actions that the model will perform that may use some additional/changed data gathered in a user-interactive view.
• Controller is also responsible for invoking new views upon conditions.

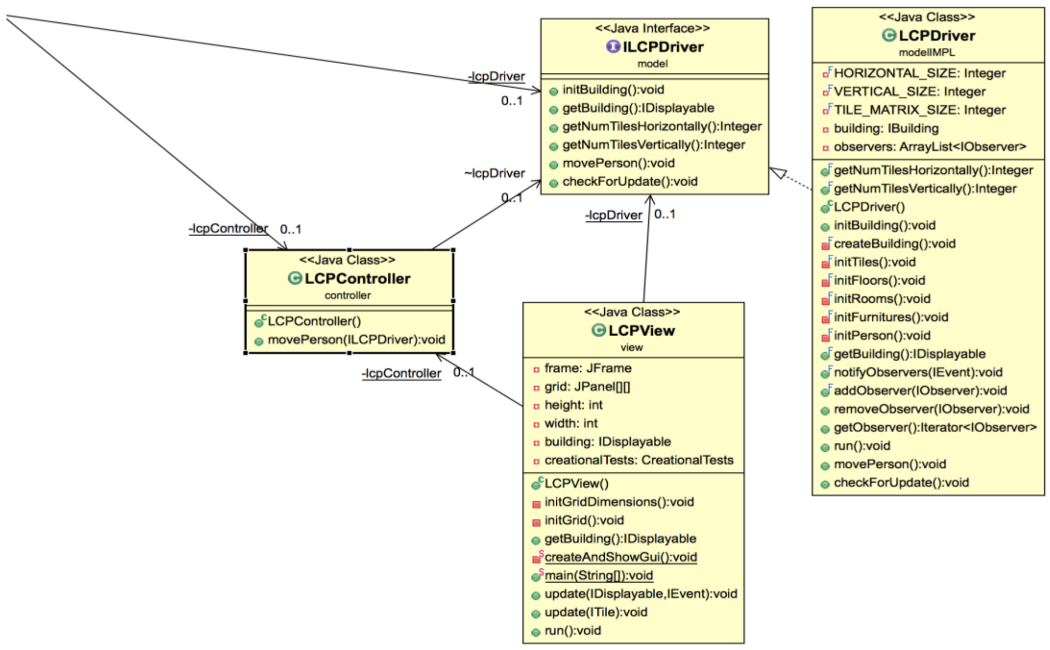


Figure 5

6. Results

Here two test runners were created, one follows simple JUnit test case named "*CreationalTest.java*", where tests for creation of the whole building along with its object is tested upon (as seen in figure 8). Second one is "*ButtonGrid.java*", which is based on swing UI (as seen in the figure 6 and 7).

In swing UI, there are partition of colors based on the rooms and floors. So far for understanding purpose we only have created and decorated one room with all kinds of object according to figure 1. A person based on multithreading of MVC can move from one spot to the other, however the checks for the object penetration are yet to be figured out in the next builds.

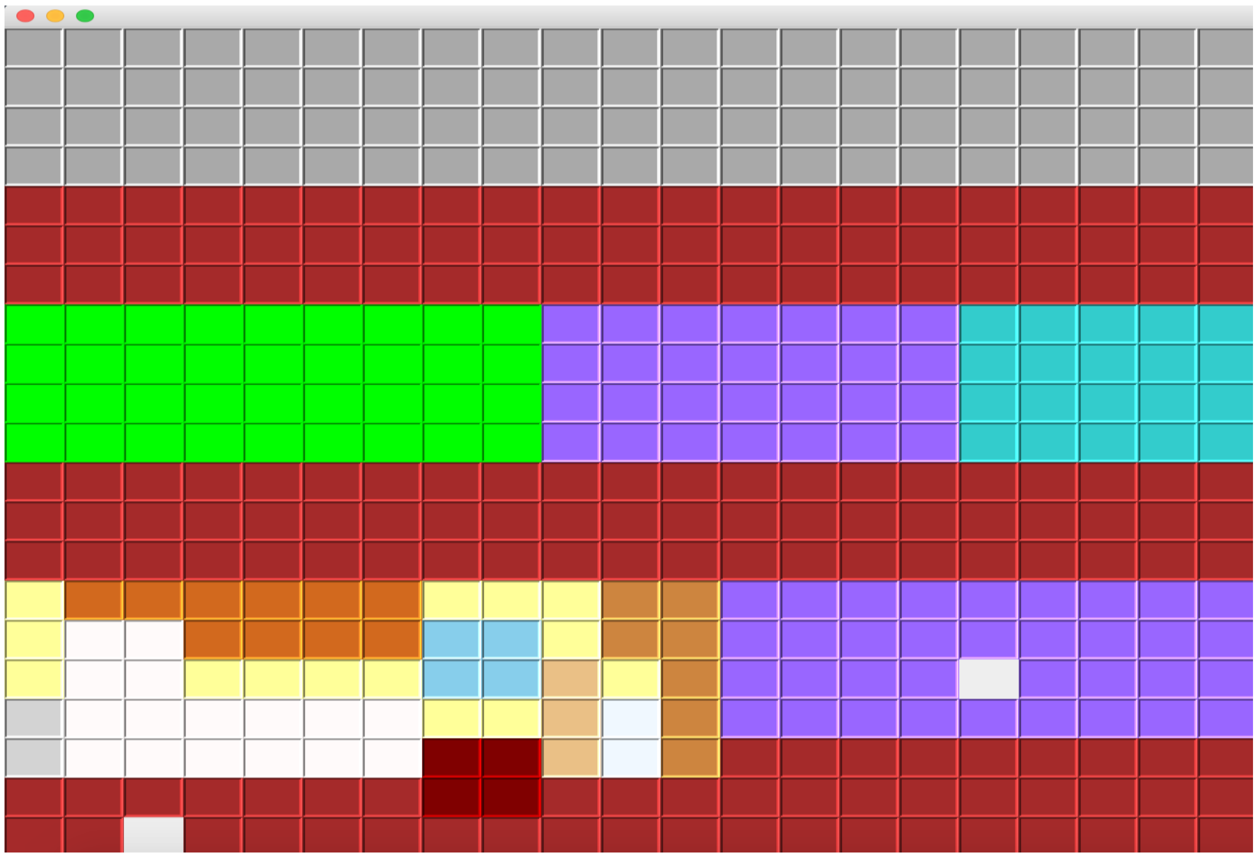


Figure 6 - Room 5 on the bottom is kitchen with utilities of kitchen as in figure 1

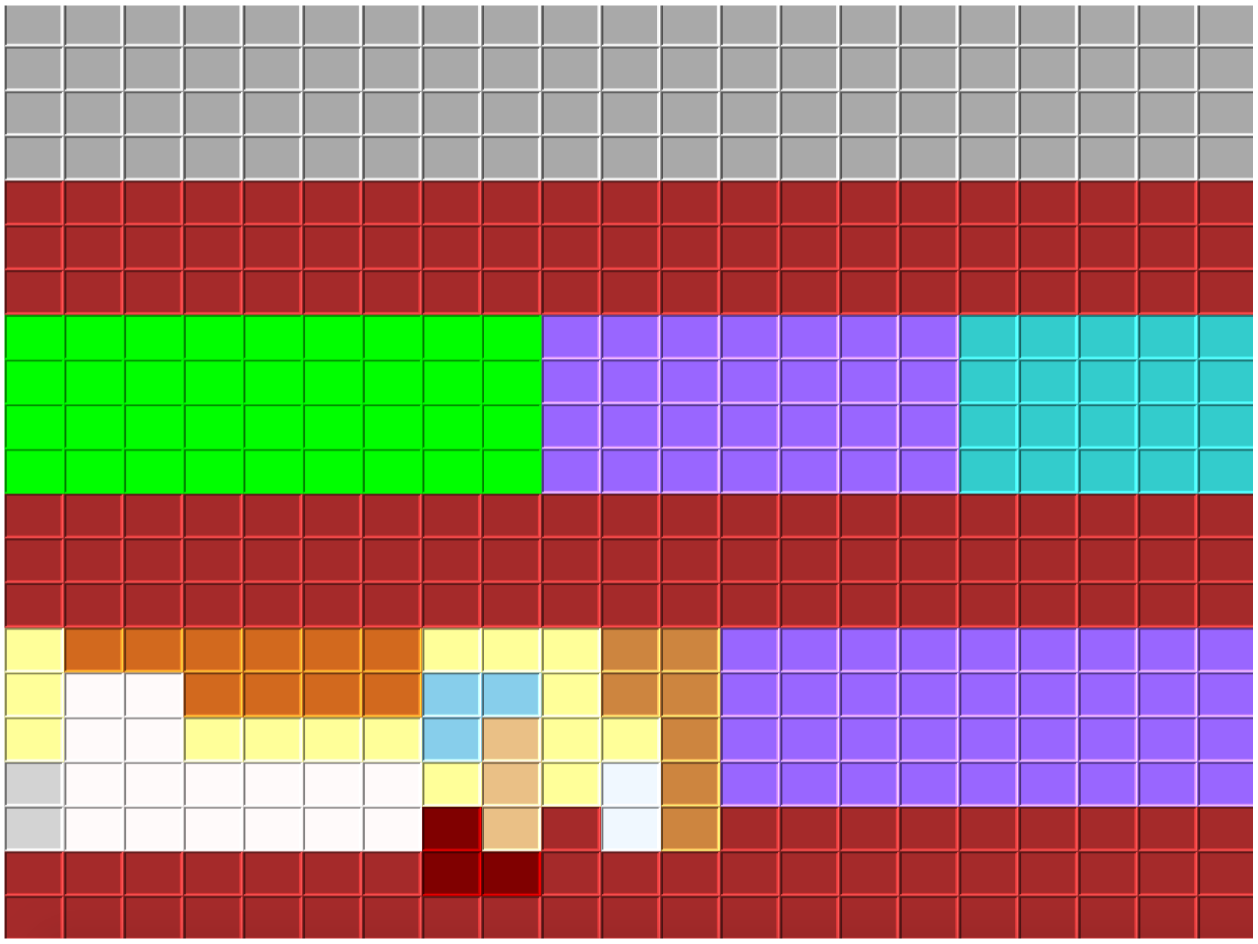
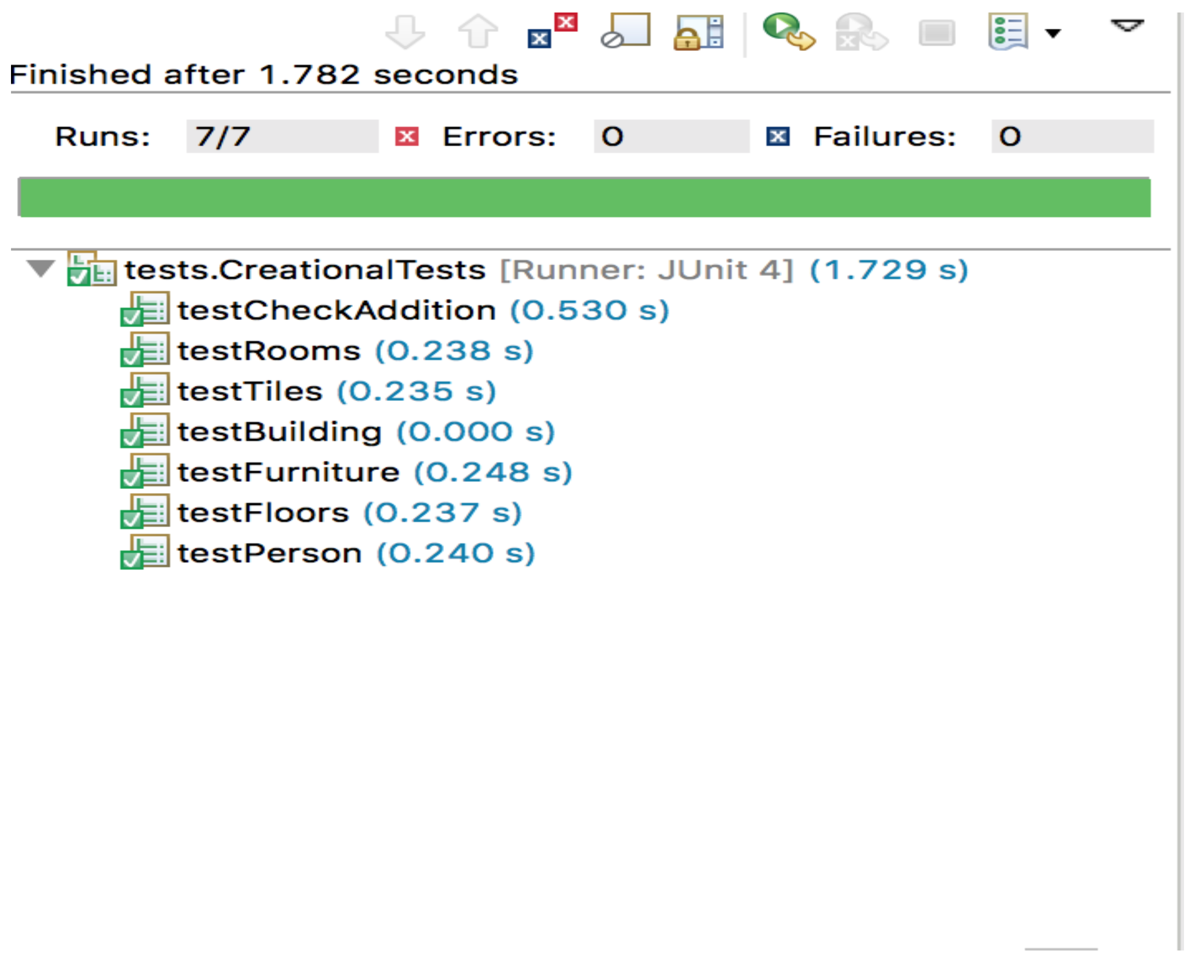


Figure 7 - Person which has skin color moves till the fridge but moves over the table



```
Finished after 1.782 seconds

Runs: 7/7    Errors: 0    Failures: 0

tests.CreationalTests [Runner: JUnit 4] (1.729 s)
    testCheckAddition (0.530 s)
    testRooms (0.238 s)
    testTiles (0.235 s)
    testBuilding (0.000 s)
    testFurniture (0.248 s)
    testFloors (0.237 s)
    testPerson (0.240 s)
```

Figure 8

7. Future Work

A comprehensive simulation system can be built with strategies of person to move from one room to the other in a seamless manner. Here a decorator can be used where a furniture for instance, can be decorated as penetrable or impenetrable. Based on that a movement of object can differ and may require to make several decisions on which tile it should go if there is an impenetrable object on its way.

Based on the decorations, display can also be tweaked in a way if person can go around the table or through the table for example. Assumptions for screen sizes can be made more dynamic based on the screen size that client offers. UI can be improvised using images instead of colors over JPanels. Last but not the least, code requires a lot of refactoring that needs to be done for dead code, linguistic antipatterns.

8. Conclusion

It is always interesting to build something, but it adds more weight to it if it is done in more articulate fashion and then is when using design patterns helps your code look much more cleaner. Not only cleaner but a client will be happiest to have a code built with less burden on his shoulders. A design patterns has many intents but the main intent that every design pattern focuses on is to build a code that helps client use the system seamlessly.

A explanatory work for using various design patterns with Little Computer People simulation platform is a good example to look and learn design patterns at its core. Much future work is needed to be done for the former statement to be true, but in the end design patterns are a good coding practice that can be used along any project based on its intent and problem under consideration.

9. References

a. https://en.wikipedia.org/wiki/Little_Computer_People
b. https://en.wikipedia.org/wiki/Composite_pattern
c. https://en.wikipedia.org/wiki/Little_Computer_People