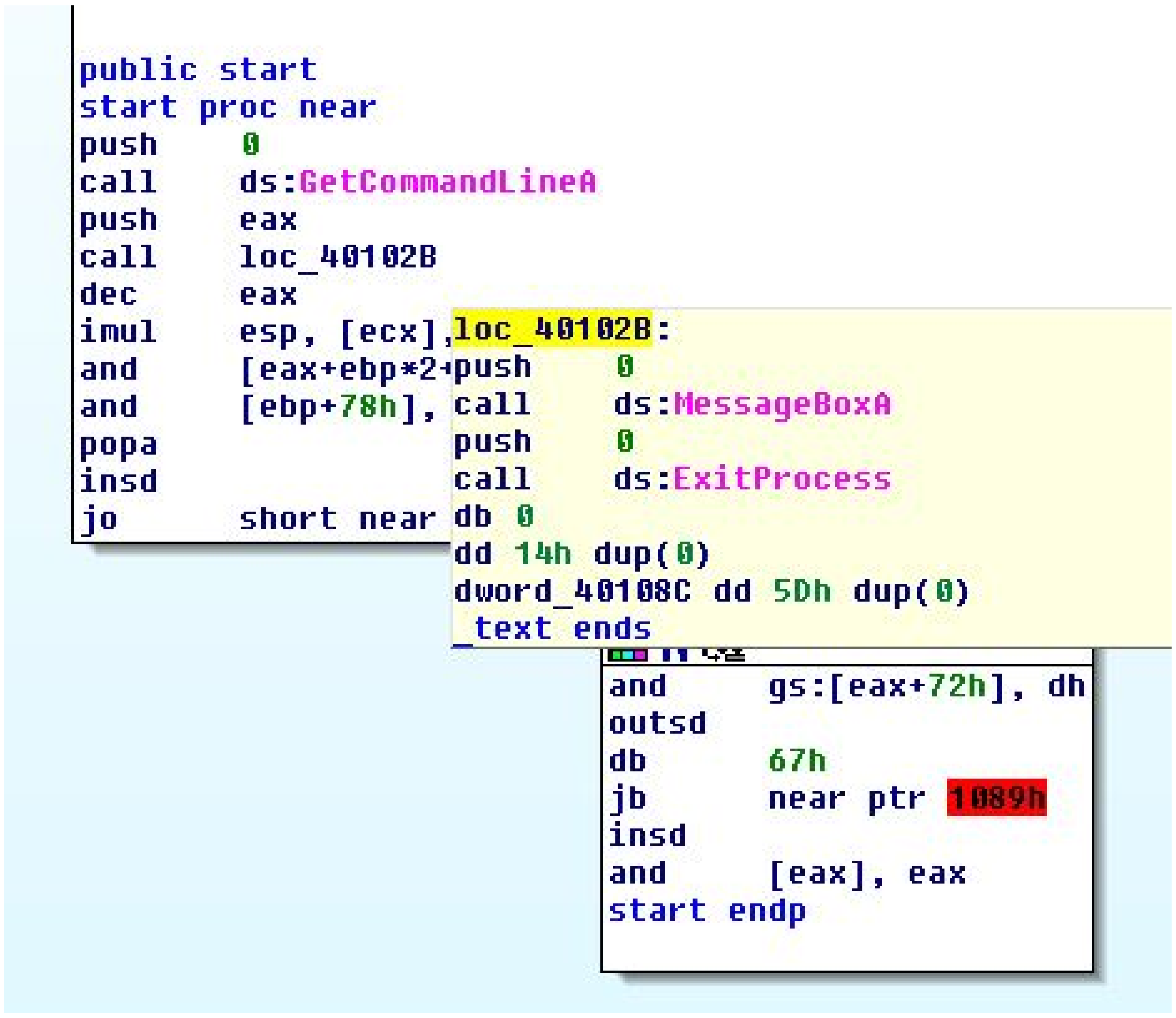
COS 121 - Code Motivation 2016-10-04

Connor Armand Du Plooy : u16169532



```
public start
start proc near
push    0
call    ds:GetCommandLineA
push    eax
call    loc_40102B
dec     eax
imul    esp, [ecx],
and     [eax+ebp*2+
and     [ebp+78h],
popa
insd
jo      short near
```

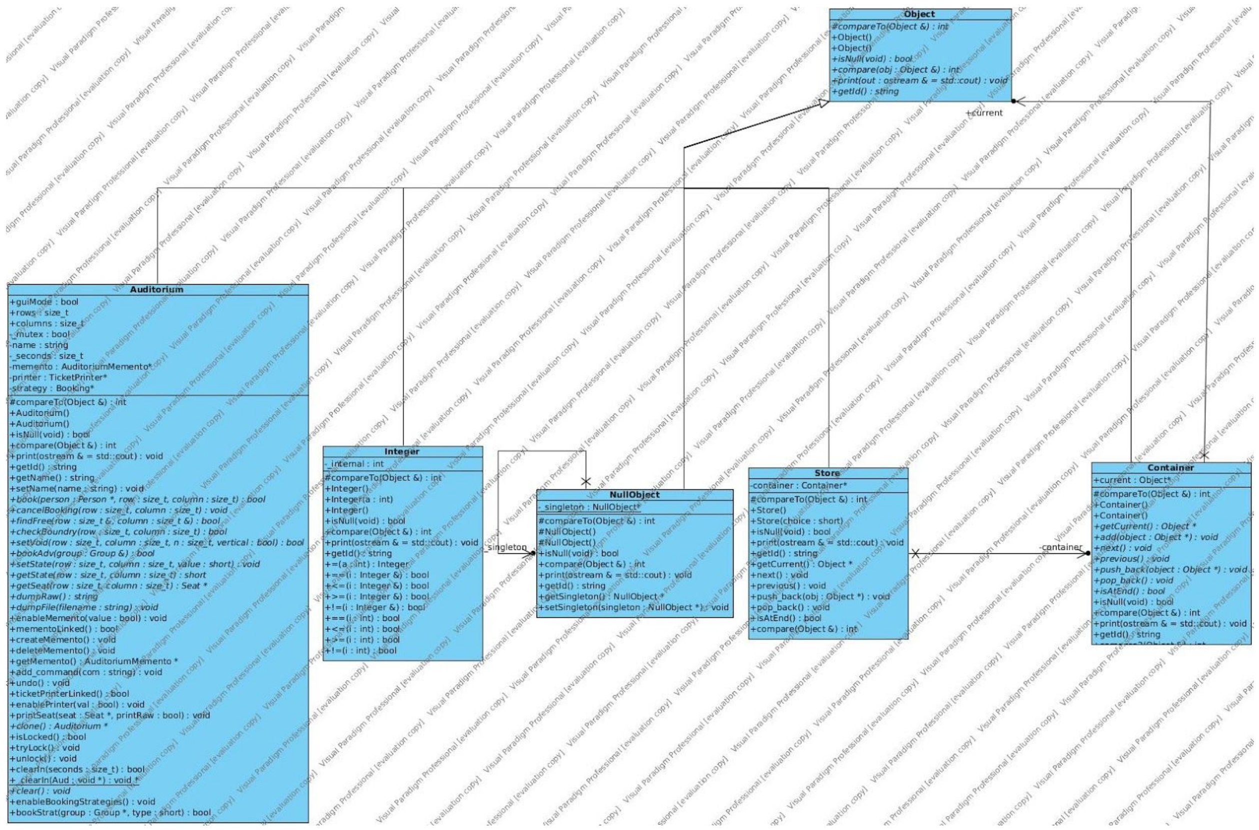
```
loc_40102B:
push    0
call    ds:MessageBoxA
push    0
call    ds:ExitProcess
db 0
dd 14h dup(0)
dword_40108C dd 5Dh dup(0)
_text ends
```

```
and     gs:[eax+72h], dh
outsd
db      67h
jb      near ptr 1089h
insd
and     [eax], eax
start endp
```

# Index

## Subsystem 1 : Object Hierarchy

(Subsystem1.jpg in the documentation directory)
I apologise for the watermarks , I had a working version of visual paradigm and when I was ready to export everything it stopped to accept my login credentials ; forget your password was of no help.



Object
#compareTo(Object &) : int
+Object()
+Object()
+isNull(void) : bool
+compare(obj : Object &) : int
+print(out : ostream & = std::cout) : void
+getId() : string
+current
Auditorium
+guiMode : bool
+rows : size_t
+columns : size_t
-_mutex : bool
-name : string
-_seconds : size_t
-memento : AuditoriumMemento*
-printer : TicketPrinter*
-strategy : Booking*
#compareTo(Object &) : int
+Auditorium()
+Auditorium()
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string
+getName() : string
+setName(name : string) : void
+book(person : Person *, row : size_t, column : size_t) : bool
+cancelBooking(row : size_t, column : size_t) : void
+findFree(row : size_t &, column : size_t &) : bool
+checkBoundry(row : size_t, column : size_t) : bool
+setVoid(row : size_t, column : size_t, n : size_t, vertical : bool) : bool
+bookAdv(group : Group &) : bool
+setState(row : size_t, column : size_t, value : short) : void
+getState(row : size_t, column : size_t) : short
+getSeat(row : size_t, column : size_t) : Seat *
+dumpRaw() : string
+dumpFile(filename : string) : void
+enableMemento(value : bool) : void
+mementoLinked() : bool
+createMemento() : void
+deleteMemento() : void
+getMemento() : AuditoriumMemento *
+add_command(com : string) : void
+undo() : void
+ticketPrinterLinked() : bool
+enablePrinter(val : bool) : void
+printSeat(seat : Seat *, printRaw : bool) : void
+clone() : Auditorium *
+isLocked() : bool
+tryLock() : void
+unlock() : void
+clearIn(seconds : size_t) : bool
+_clearIn(Aud : void *) : void *
+clear() : void
+enableBookingStrategies() : void
+bookStrat(group : Group *, type : short) : bool
Integer
-_internal : int
#compareTo(Object &) : int
+Integer()
+Integer(a : int)
+Integer()
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string
+=(a : int) : Integer
+==(i : Integer &) : bool
+<=(i : Integer &) : bool
+>=(i : Integer &) : bool
+!=(i : Integer &) : bool
+==(i : int) : bool
+<=(i : int) : bool
+>=(i : int) : bool
+!=(i : int) : bool
-_singleton
NullObject
-_singleton : NullObject*
#compareTo(Object &) : int
#NullObject()
#NullObject()
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string
+getSingleton() : NullObject *
+setSingleton(singleton : NullObject *) : void
Store
-container : Container*
#compareTo(Object &) : int
+Store()
+Store(choice : short)
+isNull(void) : bool
+print(ostream & = std::cout) : void
+getId() : string
+getCurrent() : Object *
+next() : void
+previous() : void
+push_back(obj : Object *) : void
+pop_back() : void
+isAtEnd() : bool
+compare(Object &) : int
-container
Container
+current : Object*
#compareTo(Object &) : int
+Container()
+Container()
+getCurrent() : Object *
+add(object : Object *) : void
+next() : void
+previous() : void
+push_back(object : Object *) : void
+pop_back() : void
+isAtEnd() : bool
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string

Subsystem 1 forms the base for the entire project because almost all the other classes inherit from it. This is done so that the classes can be as generic as possible.

Design patterns implemented for this subsystem ;

- Adapter
- Singleton
- Iterator
- Factory

The table below maps classes from the specification to my source.

| Practical Specification | Actual Source |
|---|---|

| | |
|---|---|
| Object | Object |
| NullObject | NullObject |
| Integer | Integer |
| Auditorium | Auditorium |
| Container | Container |
| Abstract Iterator | Store |

## Object

The object class is the super/parent class of all other classes in the specification. It has a few virtual functions which are required by all child classes of itself. The Object class is also used to be able to implement containers, iterators and factories of all the other subclasses because it is the one generic interface present in all of the aforementioned classes.

The object class per se plays a large role in all of the design patterns used for the project but it does not form part of any concrete class ( it is not a direct participant in any of the patterns ).

## NullObject

The NullObject is part of the implementation for a singleton. There can be only one (or zero ) instance(s) of the NullObject at any given moment. Another class NullPerson inherits from NullObject , this class is not a singleton.

The NullObject is used primarily inside larger , more complex memory structures such as lists , matrixes and vectors to represent the absence of an element. This is where the isNull() method comes in handy. All subclasses of object implement the isNull function but only the NullObject returns true on this call.

## Integer

The Integer class is used primarily for debugging purposes. It has absolutely no implementation in the release version of the project.

## Auditorium

The Auditorium class is an abstract class exposing two subclasses FixedSizeAuditorium and DynamicSizeAuditorium both of which use a FixedSizeMatrix in the background. This class is used to represent different types of auditoriums as well as bind People pointers to Seats , the internal class used by FixedSizeAuditorium. It also exposes a few functions which allow people to book seats in more advanced and diverse ways.

## Container

The object class is the super/parent class of all larger classes in the specification , it forces subclasses to implement next() , forward() and end() functions.

The iterators ( Store ) for the container classes should be reset once something is added or removed from the container. This is due to the diversity of the different types of containers. This class makes handling large groups of Objects elementary.

## Store ( Abstract Iterator )

The store class acts as an interface to the adapter class Container , it is used to handle groups of Objects in a uniform fashion.

## Design Patterns

Factory : The Auditorium Modeller class contains the functionality to construct Auditoriums. The Auditorium Modeller thus acts as the Creator and Concrete Creator. The concrete product is an Auditorium pointer which can be a FixedSizeAuditorium or a DynamicSizeAuditorium ( which's implementation might be omitted because it can be emulated by using shorts to represent state ). The factory method was chosen because it provides an easy way save templates of auditoriums. The Store class also uses the factory method to instantiate any type of container.

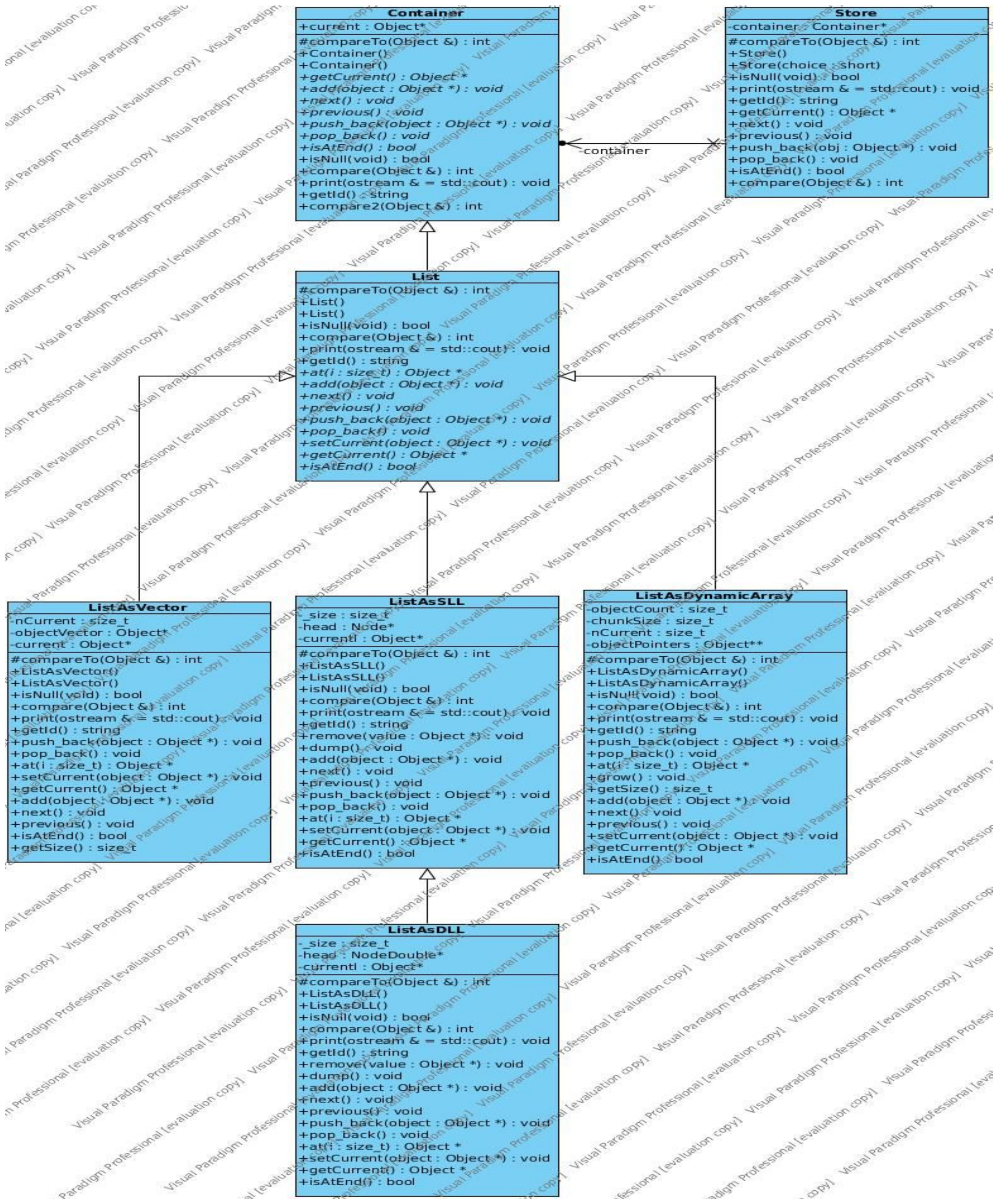
Iterator : The Store class serves as an easy way to it iterate over any kind of Container object. In essence this means that Lists and Matrixes can all be handled as if they were the same data type. The structure of the iterator pattern inside this subsystem is slightly more complicated , Container acts as the iterator itself but the iterating functions are delegated to its two subclasses List and Matrix. That means that the list and matrix subclasses are the concrete iterators. The Store class serves as a concrete aggregate. The iterator pattern was used to simplify the process of working with larger data structures.

Singleton : The NullObject can only have one instance of itself at any given moment. It provides an accessor method so the class itself can make sure there is an instance of itself and if there isn't create it. The singleton pattern is used because we don't need to instantiate more than one of the NullObjects , we only need to save pointers to one instance of it.

Adapter : The adapter pattern works as a wrapper. It uses delegation to accomplish this goal. Container acts as the adapter while it's subclasses List and Matrix acts as adaptees. The target in this case is the Store class. In other words , the Container class adapts either a List or Matrix subclass to interface with the Seat class. The adapter pattern was used so that changes could be made to container and it's subclasses without breaking the entire implementation.

No libraries were used in this subsystem, everything could be implemented without them.

## Subsystem 2: List Hierarchy



Subsystem 2 is used to implement efficient storage data structures , it consists primarily of ListAsSLL , ListAsDLL , ListAsDynamicArray and ListAsVector which all are different methods of storing Objects ( the base class ).
There are better images inside the documentation directory. (Subsystem3.jpg)

## Design Patterns

### Iterator

The list hierarchy is slightly more complex than just that, it is very closely bound to the matrix hierarchy and iteration through it is accomplished via aggregation and delegation. There is another class named Store which can create any one of the aforementioned containers ( list or matrix ). In this case the iterator is the concrete class which you want to iterate , the aggregate is the store class itself. It should be noted that the store class can also be used to insert/remove items but that the iterator itself needs to be reset should this functionality be used.

### Factory

The factory pattern is implemented inside the Store class to create iterators and/or containers. The creator is the store class itself while the product would be any child class of container. This was done to have one universal method of traversing larger data structures.

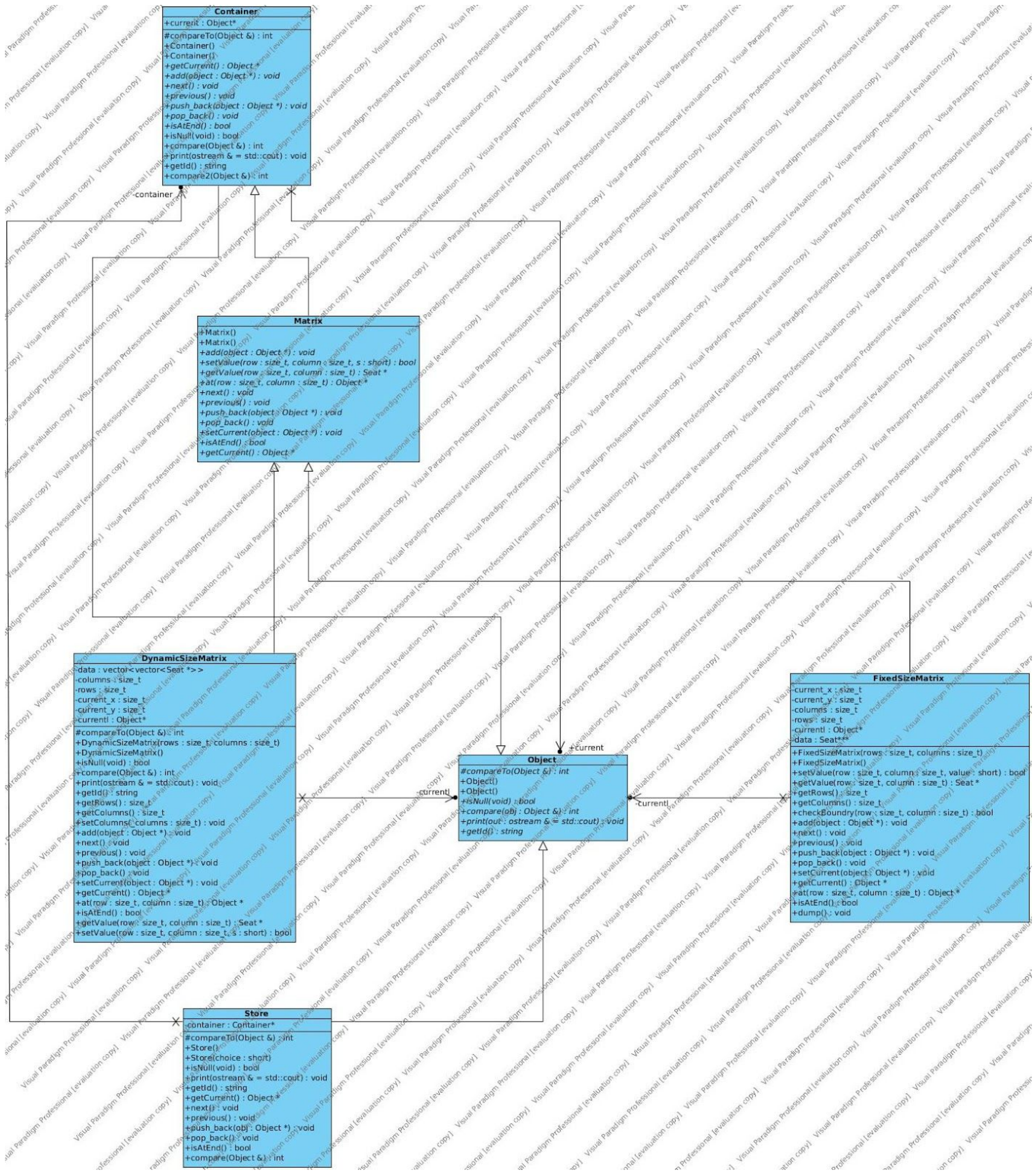
### Decorator

The decorator pattern isn’t implemented fully in subsystem two because it wasn’t necessary. The ListAsVector is used as an extra component to Auditorium and it’s subclasses ( albeit via inheritance and internal functions ) it is used to implement memento and the printer. Auditoriums store pointers to Printers and Mementos which can be enabled at any time during program execution. The pattern is used to optimize the auditorium classes.

## Adapter

The adapter pattern is implemented using delegation and inside child classes of Container. The adaptee is the container class which exposes an interface to the target , the adapter is implemented using delegation i.o.w all subclasses of container implement a standard set of functions to move around in the data structure. Again , the adapter pattern is implemented to create a universal interface to larger data structures and remove the burden of having to know how to use the aforementioned structure’s internal implementation.

## Subsystem 3: Matrix Hierarchy



Container
+current : Object*
#compareTo(Object &) : int
+Container()
+Container()
+getCurrent() : Object *
+add(object : Object *) : void
+next() : void
+previous() : void
+push_back(object : Object *) : void
+pop_back() : void
+isAtEnd() : bool
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string
+compare2(Object &) : int
-container
Matrix
+Matrix()
+Matrix()
+add(object : Object *) : void
+setValue(row : size_t, column : size_t, s : short) : bool
+getValue(row : size_t, column : size_t) : Seat *
+at(row : size_t, column : size_t) : Object *
+next() : void
+previous() : void
+push_back(object : Object *) : void
+pop_back() : void
+setCurrent(object : Object *) : void
+isAtEnd() : bool
+getCurrent() : Object *
DynamicSizeMatrix
-data : vector<vector<Seat *>>
-columns : size_t
-rows : size_t
-current_x : size_t
-current_y : size_t
-currentl : Object*
#compareTo(Object &) : int
+DynamicSizeMatrix(rows : size_t, columns : size_t)
+DynamicSizeMatrix()
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string
+getRows() : size_t
+getColumns() : size_t
+setColumns(_columns : size_t) : void
+add(object : Object *) : void
+next() : void
+previous() : void
+push_back(object : Object *) : void
+pop_back() : void
+setCurrent(object : Object *) : void
+getCurrent() : Object *
+at(row : size_t, column : size_t) : Object *
+isAtEnd() : bool
+getValue(row : size_t, column : size_t) : Seat *
+setValue(row : size_t, column : size_t, s : short) : bool
-current
-currentl
Object
#compareTo(Object &) : int
+Object()
+Object()
+isNull(void) : bool
+compare(obj : Object &) : int
+print(out : ostream & = std::cout) : void
+getId() : string
-currentl
FixedSizeMatrix
-current_x : size_t
-current_y : size_t
-columns : size_t
-rows : size_t
-currentl : Object*
-data : Seat***
+FixedSizeMatrix(rows : size_t, columns : size_t)
+FixedSizeMatrix()
+setValue(row : size_t, column : size_t, value : short) : bool
+getValue(row : size_t, column : size_t) : Seat *
+getRows() : size_t
+getColumns() : size_t
+checkBoundry(row : size_t, column : size_t) : bool
+add(object : Object *) : void
+next() : void
+previous() : void
+push_back(object : Object *) : void
+pop_back() : void
+setCurrent(object : Object *) : void
+getCurrent() : Object *
+at(row : size_t, column : size_t) : Object *
+isAtEnd() : bool
+dump() : void
Store
-container : Container*
#compareTo(Object &) : int
+Store()
+Store(choice : short)
+isNull(void) : bool
+print(ostream & = std::cout) : void
+getId() : string
+getCurrent() : Object *
+next() : void
+previous() : void
+push_back(obj : Object *) : void
+pop_back() : void
+isAtEnd() : bool
+compare(Object &) : int

See documentation/Subsystem3.jpg for a higher quality image.

## Design Patterns

Subsystem 3 is used to implement efficient storage data structures , it consists primarily of FixedSizeMatrix , DynamicSizeMatrix and FlexiMatrix which all are different methods of storing Objects ( the base class ). It is very similiar to subsystem 2 although raw iteration of the data structures differ slightly.

## Iterator

The matrix hierarchy is slightly more complex than just that, it is very closely bound to the list hierarchy and iteration through it is accomplished via aggregation and delegation. There is another class named Store which can create any one of the aforementioned containers ( list or matrix ). In this case the iterator is the concrete class which you want to iterate , the aggregate is the store class itself. It should be noted that the store class can also be used to insert/remove items but that the iterator itself needs to be reset should this functionality be used.Again , the iterator pattern is implemented to create a universal interface to larger data structures and remove the burden of having to know how to use the aforementioned structure’s internal implementation.

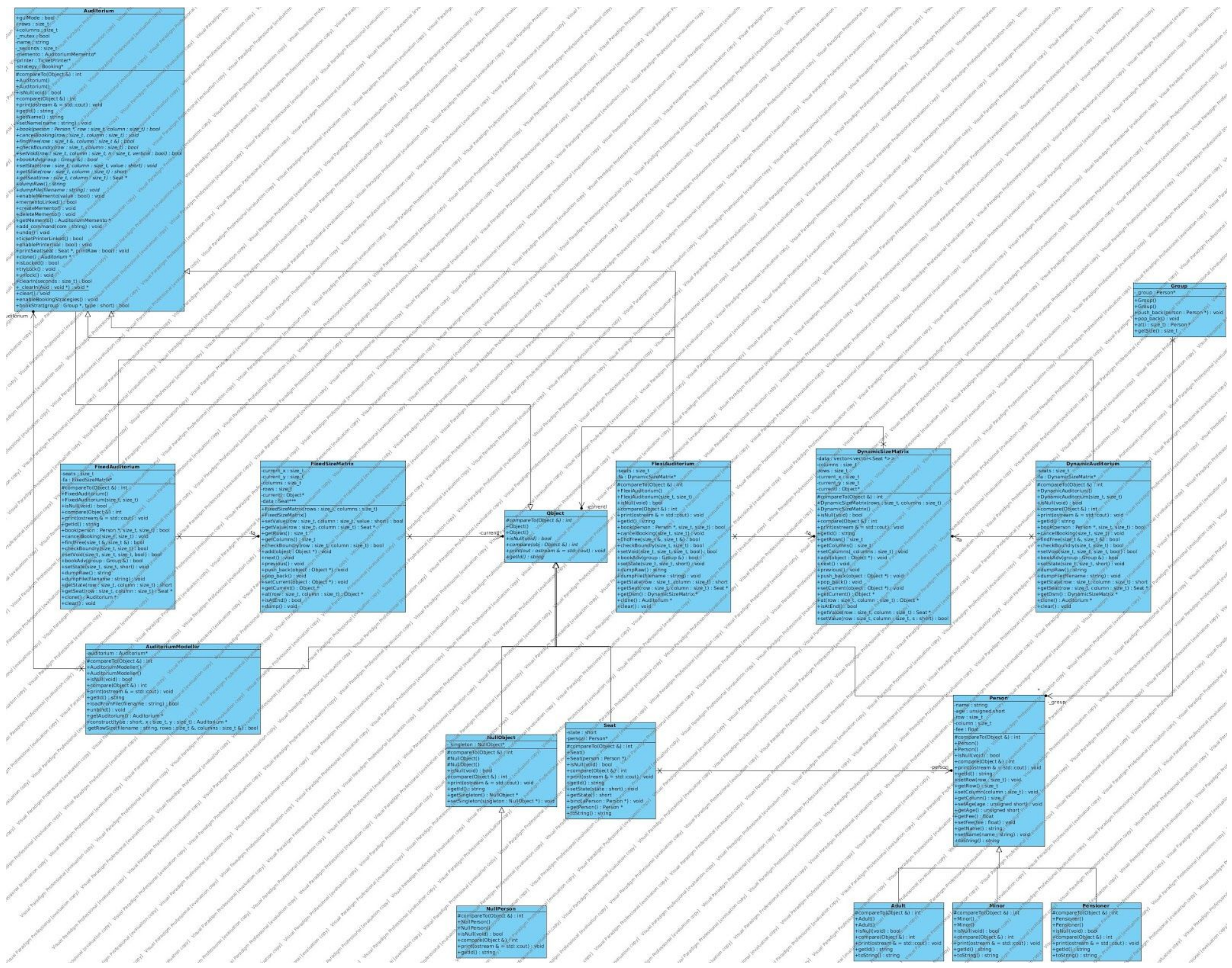
## Factory

The factory pattern is implemented inside the Store class to create iterators and/or containers. The creator is the store class itself while the product would be any child class of container. This was done to have one universal method of traversing larger data structures.

## Template

The template pattern is used internally to implement the universal iterators , all child classes of Container have to implement their own internal ( standardized ) functions to move around in the data structure. The pattern was used to implement iterator and for no other reason.

## Subsystem 4: Auditorium Modeller



Higher quality image is in documentation/Subsystem4.jpg

The Auditorium Modeller is used to create different types of auditoriums , it primarily functions as a factory! It also has the added functionality of being able to load auditoriums from files ( think of this function as loading templates from files ).

## Iterator

There is no concrete  iterator implementation inside the auditoriums , it can be exposed via Container but for the most part is not needed. The booking functions iterate via container's interface to complete it's functions.

## Factory

The factory method for the auditorium method is implemented using a private auditorium pointer and a get() function. The modeller can be called with a short parameter to specify the kind of auditorium that should be created. Modeller is thus the creator while the product is an instance of auditorium. The factory method was used to make it easier to manage the different kinds of auditoriums.

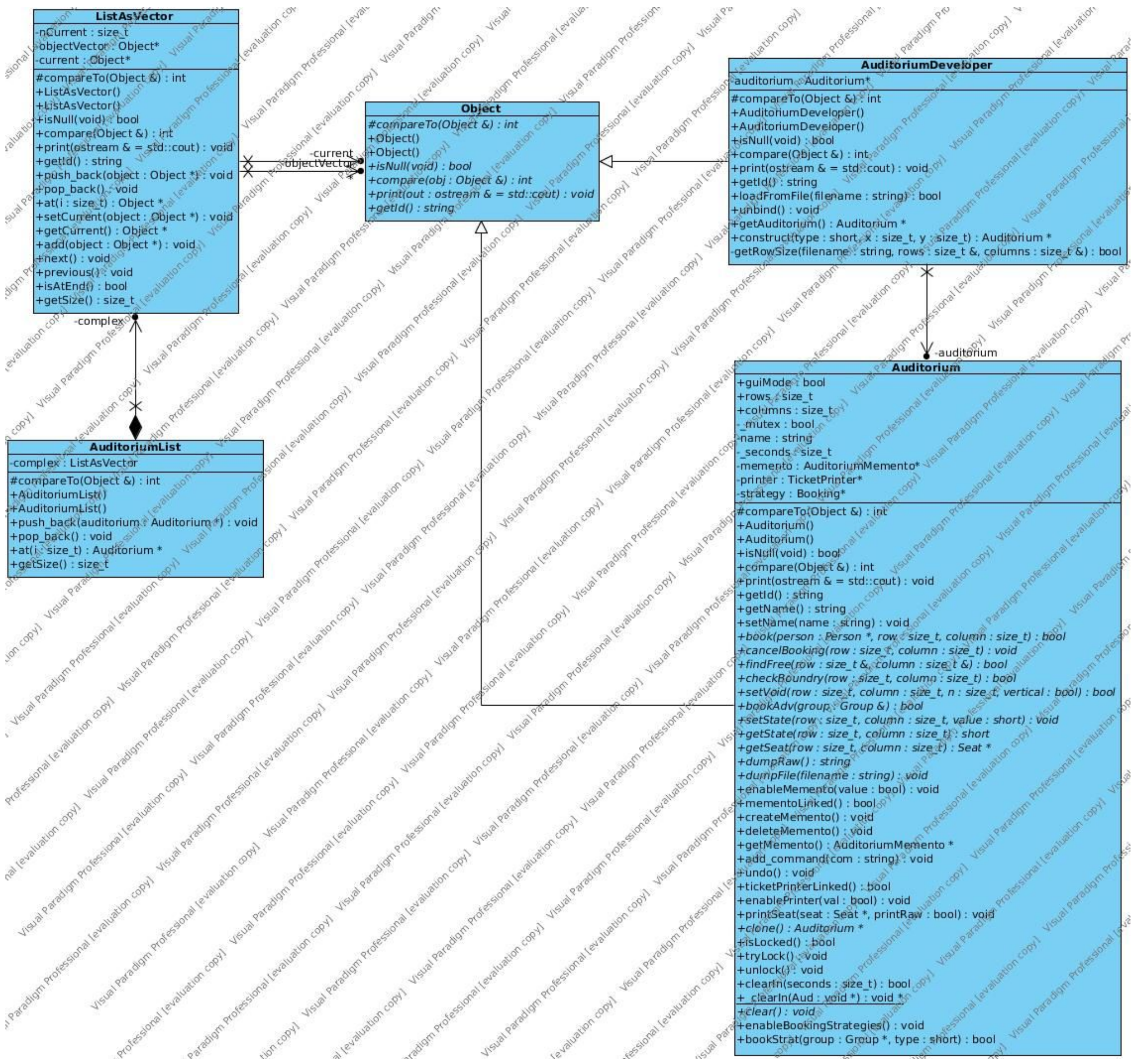
## Template

The template method is used inside the subclasses of auditorium , the Auditorium class defines virtual functions for booking and cancelling seats but the function's implementation are called from within the concrete classes of auditorium. They may differ slightly , eg a fixed size auditorium does not have any seats with a state of SEAT_VOID which means I can omit checks for it. The template method was used to minimise the amount of code I need to write in order to have working auditoriums.

## Prototype

All of the auditoriums utilise the prototype method to create copies of themselves. This is useful once operations on the data structures become too complex to work with just one instance of the object - or if we need to implement memento for example. The client is the modeller which can get a clone of the auditorium , while the concrete prototype is the auditorium clone itself.

## Subsystem 5: Auditorium Developer


ListAsVector
-nCurrent : size_t
-objectVector : Object*
-current : Object*
#compareTo(Object &) : int
+ListAsVector()
+ListAsVector()
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string
+push_back(object : Object *) : void
+pop_back() : void
+at(i : size_t) : Object *
+setCurrent(object : Object *) : void
+getCurrent() : Object *
+add(object : Object *) : void
+next() : void
+previous() : void
+isAtEnd() : bool
+getSize() : size_t
-current
-objectVector
-complex
Object
#compareTo(Object &) : int
+Object()
+Object()
+isNull(void) : bool
+compare(obj : Object &) : int
+print(out : ostream & = std::cout) : void
+getId() : string
AuditoriumDeveloper
-auditorium : Auditorium*
#compareTo(Object &) : int
+AuditoriumDeveloper()
+AuditoriumDeveloper()
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string
+loadFromFile(filename : string) : bool
+unbind() : void
+getAuditorium() : Auditorium *
+construct(type : short, x : size_t, y : size_t) : Auditorium *
-getRowSize(filename : string, rows : size_t &, columns : size_t &) : bool
-auditorium
AuditoriumList
-complex : ListAsVector
#compareTo(Object &) : int
+AuditoriumList()
+AuditoriumList()
+push_back(auditorium : Auditorium *) : void
+pop_back() : void
+at(i : size_t) : Auditorium *
+getSize() : size_t
Auditorium
+guiMode : bool
+rows : size_t
+columns : size_t
-_mutex : bool
-name : string
-_seconds : size_t
-memento : AuditoriumMemento*
-printer : TicketPrinter*
-strategy : Booking*
#compareTo(Object &) : int
+Auditorium()
+Auditorium()
+isNull(void) : bool
+compare(Object &) : int
+print(ostream & = std::cout) : void
+getId() : string
+getName() : string
+setName(name : string) : void
+book(person : Person *, row : size_t, column : size_t) : bool
+cancelBooking(row : size_t, column : size_t) : void
+findFree(row : size_t &, column : size_t &) : bool
+checkBoundry(row : size_t, column : size_t) : bool
+setVoid(row : size_t, column : size_t, n : size_t, vertical : bool) : bool
+bookAdv(group : Group &) : bool
+setState(row : size_t, column : size_t, value : short) : void
+getState(row : size_t, column : size_t) : short
+getSeat(row : size_t, column : size_t) : Seat *
+dumpRaw() : string
+dumpFile(filename : string) : void
+enableMemento(value : bool) : void
+mementoLinked() : bool
+createMemento() : void
+deleteMemento() : void
+getMemento() : AuditoriumMemento *
+add_command(com : string) : void
+undo() : void
+ticketPrinterLinked() : bool
+enablePrinter(val : bool) : void
+printSeat(seat : Seat *, printRaw : bool) : void
+clone() : Auditorium *
+isLocked() : bool
+tryLock() : void
+unlock() : void
+clearIn(seconds : size_t) : bool
+_clearIn(Aud : void *) : void *
+clear() : void
+enableBookingStrategies() : void
+bookStrat(group : Group *, type : short) : bool

Higher quality image in documentation/Subsystem5.jpg

Subsystem 5 , Auditorium Developer , is used to rapidly create and copy auditoriums. It is not linked directly to the reservation system as specified in the assignment specification.

### Builder

The builder pattern is implemented as a case statement which constructs different kinds of auditoriums based on a short which is passed to a function. The pattern was used to create different kinds of auditoriums with the same function. The concrete builder is the developer class while the product is the instance of the auditorium that is created.

### Factory

The factory pattern is implemented , but wrapped , by the builder pattern. Once again the concrete creator is the Developer while the auditorium instance is the concrete pattern. The design pattern is used to define an easy to use and easy to change interface for creating auditorium classes.

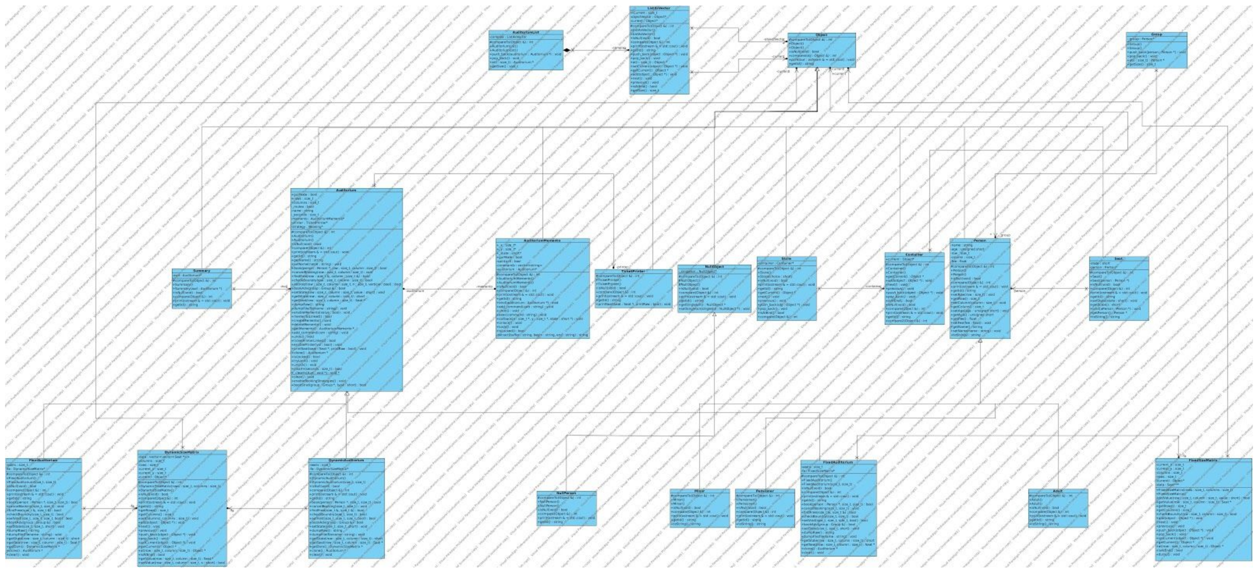
### State

The state pattern is not implemented using an Object Orientated approach but rather a procedural one. Auditoriums will remain relatively small and thus using a for loop to parse them and return a variable representing state will be elementary. This was done to not waste memory or processing time.

### Memento

The memento pattern is not directly associated with auditorium developer in my project but rather the auditorium class itself ( as more of a decorator ). It will be discussed in subsystem six.

## Subsystem 6: Reservation



Higher quality image is in documentation/Subsystem6.jpg

The reservation subsystem is not implemented as a class nor is it functions. The subsystem instead resides as smaller implementations within previous classes. I will provide a table mapping the specification functions to internal implementations.

| Practical Specification | Internal Application |
|---|---|
| Memento | AuditoriumMemento |
| Strategy | Auditorium delegates to subclasses. |
| Decorator | TicketPrinter and AuditoriumMemento |
| Command | AuditoriumMemento |

## Strategy

The strategy pattern is implemented ( as a lot of my project ) using #define constants , internally declared as a short. Each of these constant represent a different kind of strategy. In other words the strategy pattern isn't implemented using classes/objects but rather using procedural programming techniques and Object Orientated techniques. The context is the auditorium instance , the strategies are the define constants. The pattern was implemented this way as to not suffer performance penalty from Object Orientated overhead but also have it's ease of use. Subclasses internally handle booking strategies ( object orientated ) but the strategies themselves are not classes ( procedural ).

## State

The state pattern is not implemented using an Object Orientated approach but rather a procedural one. Auditoriums will remain relatively small and thus using a for loop to parse them and return a variable representing state will be elementary. This was done to not waste memory or processing time.

## Decorator

This is where all my magic happens. The decorator pattern is used extensively in this practical by leveraging pointers. The main functionality of the entire practical resides inside the auditorium subclasses. These classes can be extended by the decorator pattern to print tickets upon a booking or/and manage internal mementos.  I have no decorator class per se , it is all implemented/enabled by calling functions of the component/concrete component Auditorium. The concrete decorators include AuditoriumMemento and TicketPrinter.
The decorator pattern was used as to allow the easy extension and modification of the auditorium classes without editing them directly.

## Memento

As stated above memento is internally implemented using a decorator. The AuditoriumMemento class will be discussed here. The memento itself is AuditoriumMemento while the Originator is the auditorium class itself. The implementation was done using STL vectors for memory and CPU performance. Auditoriums and memento's reference each other internally , whenever a booking is made and a memento is linked to an auditorium a std::string representing a command is passed to an STL vector contained withing AuditoriumMemento. The undo function pops from the vector , and calls the functions associated with the specific operation via it's internal Auditorium pointer. All of this is handled by my system and the programmer need only specify whether he/she wants the feature enabled - to lower memory usage it is disabled by default.

## Command

The command pattern is implemented internally between Auditoriums and AuditoriumMemento. See the above explanation if you want to know how it is implemented.
The receiver object is the auditorium , the invoker is the AuditoriumMemento ( after undo has been called ) , the client is also the AuditoriumMemento. Concrete commands are represented and stored inside an STL vector as std::string. The command pattern was used because it exposes an interface to working with the core object of the project ( Auditorium ). In it's current

state it can even be used as a small scripting language for the project to automate certain aspects of managing an auditorium.

## Subsystem 7: User Interface

The user interface subsystem is simply a GUI implementation of the core classes which have already been written, the subsystem is written largely as specified in the assignment but I do not agree with a few things. There is no synchronous operations between the GUI and CLI , in fact you cannot passively switch between the two. In a normal working environment if you need a GUI and CLI implementation it is done by implementing a library which can then be used by either the  GUI/CLI.

My project has a GUI implementation ( more of a demo really ) to illustrate that I can build GUI applications , however the implementation is minimalistic. The CLI is more complete than the GUI interface but the point of the project is illustrating good design patterns and programming skill.

### Bridge

If a bridge pattern is used it will be implemented inside the main of the CLI application , so that the concrete implementation of the CLI is called when no command line parameters are passed and the GUI implementation is called if --gui is passed to the CLI application. The abstraction can thus be the main function ( or rather it's parameters namely argv and argc ) while the concrete implementations are just the different functions called on each conditional.
The bridge pattern is applied here so that we can separate abstractions from concrete implementations.

### Observer

The observer pattern is not implemented in my project because of the reasons introduced at the beginning of subsystem 7. I don't believe it to be good practise , save a few unique cases.
No large projects use this method ( updating the CLI and GUI as if they were the same process - they shouldn't be even existing in the same thread ). In any case if I were to implement this kind of functionality I would do it using UNIX sockets , where the CLI interface is a server and controller sending state changes across the socket to the listener/GUI which updates variables based on this message.  As a reference I'd like to add Iserni's answer to a question on StackExchange.

## Libraries

I used Qt to create my GUI application , referencing the makefile we can see the following libraries being linked.

**-lQt5Widgets -L/usr/X11R6/lib64 -lQt5Gui -lQt5Core -lGL -lpthread**

Qt5Widgets, Qt5Gui and Qt5Core are all part of Qt's internal workings. GL is the openGl library for graphics which is probably used to create forms and load images. pThread is a C library which allows you to use threads in your application. This is done due to how GUI applications work internally , there needs to be a thread monitoring the application to detect when it isn't responding to internal/external events and signals. I chose Qt because it allows you to easily export to other systems.

## Finalisation

The finished project looks as follows : (Final.jpg in the documentation directory for a better view)