OOP – Final assignment

Yedidya Even chen - 207404997

1) General explanation:
    a. The system is centered around a **database** that holds all the information.
    There is a collection of **cities** – each city holds the **hotels** that are in it, and each hotel holds a collection of its rooms.
    There is also a collection of **guests**. A guest can see, make, and cancel reservations.
    Each guest has a **search** object, that holds the search parameters and does the actual filtering and searching. The search asks a relevant hotel for suggestions of rooms that match the search parameters. Each hotel has a public method that gets a request to reserve a room, with the following parameters: the room number, check in and check out dates, and payment information (like credit card info). The hotel attempts to request payment for the reservation. If the payment is successful, the rooms are reserved, and the hotel returns a **reservation** that can be sent to the guest. Cancellations work in a similar but inverse way – the hotel attempts to refund and then frees up the rooms.
    The interaction with the program – user input, output, and all the different actions – is handled by the abstract **controller** class. None of the methods of the database classes have any user interaction (printing or getting input). All the information is passed as method parameters. This allows the database to be used by different programs or interfaces and to easily change the UI.

    The controller class – what actions can a user do?
    Making a reservation:
    The first step is to choose a city. The **search** looks within the requested city – filtering out irrelevant hotels based on the parameters (stars, price range, amenities). Then the search "asks" each hotel for a list of relevant rooms. Every hotel that has relevant
    The hotel is responsible for suggesting the rooms – this allows, potentially, for a hotel to decide which rooms to suggest based on what is convenient for itself (for instance: giving the most expensive rooms in the price range; suggesting rooms that have an opening on the specific dates in order to keep other rooms more available; or suggesting rooms in specific areas).
    This also allows the user to request a specific number of rooms for a group of people, as long as it is logical - for example, there must be at least as many rooms as people. The hotel can implement its own internal logic to assign rooms in a configuration that works. The hotel might be able to provide exactly the number of rooms that was requested, and in that case will suggest a different configuration for the same number of guests.
    The search parameters remain as they were in between searches, allowing the user to search with the same parameters without having to input the same ones each time.
    When a guest makes a reservation, they can choose how to pay, out of the payment options that the hotel supports. Once the payment information is given, the hotel is passed that information and asked to reserve a room. The hotel creates a **receipt** with the payment information – this is when the actual transaction will happen. Once the payment is confirmed, the hotel reserves the room and returns the **reservation** to the guest. Each guest keeps a collection of its reservations.

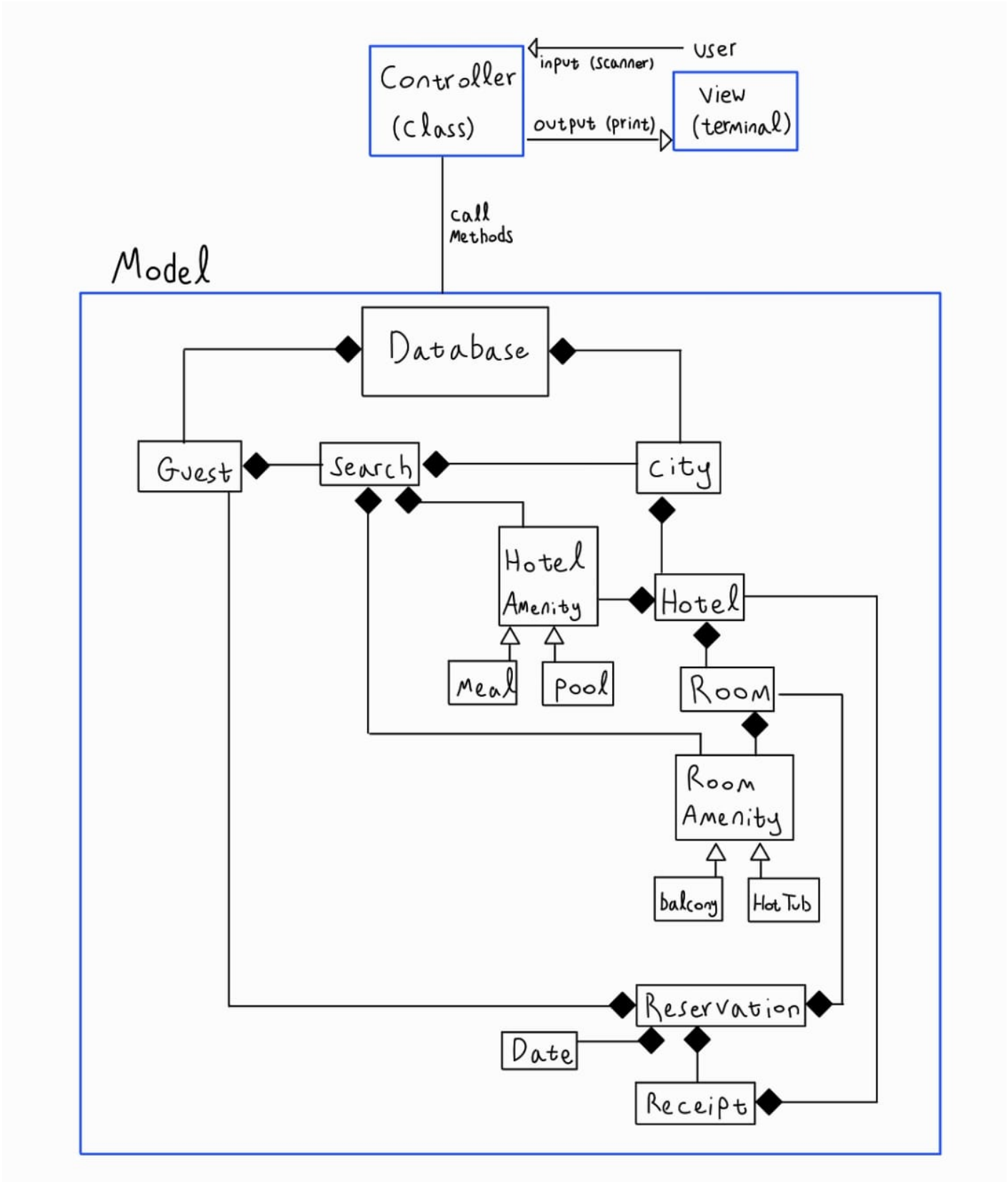
    Cancelling a reservation:
    The list of reservations is printed, and the guest chooses a reservation to cancel. Refund

options (if any are available) are offered. As with making a reservation, the guest asks the hotel to cancel the reservation, and passes the refund information. The hotel refunds the payment and cancels the reservation, freeing up the room on the relevant dates. The guest receives confirmation and deletes the reservation from its collection.
Having the hotel be responsible for the refund allows it to choose to refund a certain amount, or only a certain amount of time before the reservation, etc.

Creating a new guest, modifying information:
To log in, the user can see the guest list or search by ID.
The user can create new guests, log out and log in to a different guest, and change their guest info.

b. Some of the primary considerations while designing and implementing the system:
    i. Code flexibility: allowing for changes in the future without breaking the code and writing code that can work on different systems. None of the "model" classes have any input (for example, scanner) or output (for example, printing to console). This allows the same code to run on any system or application, if we have a working controller. For this reason, if a method gets bad input or can't do what is requested of it, it throws an exception. Each controller or façade can handle the exceptions differently.
    Hotels, rooms, and users all implement interfaces – this allows to easily add new types in the future. (for example, adding an Airbnb as a separate accommodation type).
    ii. Exception handling: what might happen that is unexpected, what the user might do. This is also part of code flexibility – it allows the methods to work with different controllers. (For example, if a method prints the error message directly, some controllers might not know what to do with it. The exceptions make it more universal).
    iii. Scalability – Runtime & Space complexity: In a practical use, a system like this would have thousands of cities, hundreds of thousands of hotels, and in each hotel hundreds of rooms. In addition, the system might have millions of users booking rooms at the same time. The way that the searches work – filtering by city, then by relevant hotels, and then asking each relevant hotel for suggestions – minimizes checking redundant rooms and minimizes runtime and memory usage. In addition, having the hotel suggest the rooms allows a hotel to temporarily "reserve" the room that it's suggesting, so that it won't suggest the same room to two users.
    Implementing interfaces also benefits scalability, because it allows us to expand the code in more ways.
    iv. Encapsulation: the user does not need to know how or why, just what. In fact, the only public method in the Controller class is the menu; everything else is private and accessible only through the menu according to the user inputs. Each method has a clear and single job (or uses helper methods). This makes the code more readable and makes debugging easier.
    v. User experience: building the façade (in this project, the controller class) in a way that makes the process convenient and intuitive for the user. Simple menus, clear and simple instructions for input, simple and readable output.

c. Describing the relationship between the classes:
The project in general has an MVC structure, and the controller is a façade. The database is a singleton. Creating rooms is done in a factory pattern. The search class utilizes a strategy pattern for sorting. The MyDate class is implemented with flyweight and an iterator. Each client is also an observer. The general logic of the reservation process uses a chain of responsibility pattern. The city – hotel – room classes are a composite structure. The composition and relation between classes is described below:



2) Modifiers in Java:
   a. **Private:** only accessible within the class. If we want to access, modify, or use, we need to use getters and setters.
   Protected: "private" to all classes except classes that extend this class.
   Public: accessible from all classes in the project.
   Final: cannot be changed once it is created.
   Static: "belongs" to the class instead of an instance of the class. Is called as itself, not on an instance.

   b. Examples:
   In general, "make everything private unless there is a reason not to".
   All fields are **private** and use getters and setters. This gives us control over changes and makes the code more flexible – we could add more classes and methods and they won't be able to "mess up" the fields that we have in unexpected ways. Most of the methods in each class (the helper functions) are private.
   The methods that are used by the controller are **public** – like the addReservation method in Client. Or methods that need to be called by other classes – like the addDate method in Room.
   The general Sort class is extended by specific sort classes (SortByStars, SortByPrice). It holds an array of the types of Sort that are created. This array is **protected** – accessible only by the classes that extend it.
   For setting constants – assigning numbers to the payment types, or for the Boolean "allow hyphen" (whether names can be hyphenated or not) – we can set them as **final**. Things that we might want to change for debugging, but not during runtime. Or when we want to assign a name to a value, for more convenient use.

```
public static final int BANK_TRANSFER = 1,
        CREDIT_CARD = 2, BIT_PAY = 3, MAX_YEAR = 3000;
```

   Counters (for example counting how many reservations and receipts were made so that each one has a unique number), or methods that we want to use without an object (the entire "functions" class, or the controller, for example) – are set as **static**. That way they can be accessed without an instance, and the counters keep track accurately.

```
public class Receipt {
    private static int count = 0;
```

3) Abstract Classes, Interfaces, Inheritance, and Polymorphism:
    a. **Abstract classes:** an abstract class is a class that we cannot create any instance of. It is useful as a "blueprint" for other classes that will extend it. For example, the "Room amenity" is abstract because we don't want an option to create a room amenity that isn't a specific type. It is extended by the specific room amenity types. This allows us to create methods that will be inherited by all the subclasses and allows us to have a collection of room amenities of different types in the same collection. Each room holds a list of the amenities it has. They are all of different concrete types, but they are all also an abstract "Room Amenity" type, so they can be on the same list. On the other hand, "hotel amenity" isn't abstract because this gives us the option to create a generic amenity – for example, an event that the hotel is hosting – without creating a class for it.
    Another reason to create an abstract class is a class that functions as a library – a collection of helper functions. However, an abstract class is meant to be overridden; So if we want a library but don't want to override it, a better option is to declare the class final and not provide a constructor. The "Functions" class is like this. There is no reason to create an instance of the class – all its methods are static. Similarly, we have no need of an instance of the Controller class, so it is final.
    Interfaces: the guest class implements the "Subscriber" interface. This allows us to have a collection of "subscribers" and update them about things, and potentially have more subscribers of other types in the same collection. The guest class also implements the "User" interface. We may want to add different types of users and having them implement the interface will allow us to refer to them in the same way. Similarly, there are interfaces for accommodations (hotels) and for rooms.
    Inheritance: the specific amenities extend the parent classes, allowing us to refer to them as the same type and giving them all access to the methods in the super class. Similarly, the general Sort class is extended by specific sort classes (SortByStars, SortByPrice).
    Polymorphism: abstract classes, interfaces, and inheritance are all examples of polymorphisms. They allow us to refer to different specific types as a single parent type. This is done with almost all object types in the database.

    b. The benefits of these concepts in OOP:
    Abstract classes allow us to define a "type" of class, so that a family of different classes can all share the same methods. This allows code reuse and gives consistency in that family of classes. They also allow us to refer to the different classes as the same type – an example of polymorphism
    Interfaces: every class that implements an interface must implement the defined methods. This allows us to refer to all the classes that implement the interface as the same "type", so we can perform the same actions on a variety of objects. This is another example of polymorphism.
    There are similarities between abstract classes and interfaces; the technical differences are:
        i. An abstract class can have concrete methods and an interface cannot. This is an advantage of the abstract class, as it allows us to reuse code.
        ii. An interface can be implemented by more than one class. This is an advantage for the interface, because it allows us to refer to a class in more than one way. For example, "Guest" implements the User and Subscriber interfaces, so we can refer to either one as needed.

Inheritance is a core part of object-oriented programming. It allows for efficient code reuse and polymorphisms and gives the code a structure that makes it easier to understand and maintain.
These concepts are used throughout this project, they help make the code more efficient and more readable and maintainable.

4) Design Patterns:
   a. Singleton:
      i. Definition and role: A creational design pattern. Useful when we need there to be exactly one instance of an object type – like a database. This means that all the information and changes will be on the same database (so, for example, there is no chance of 2 reservations being made for the same room at the same time).
      ii. Implementation: the constructor is made private. Instead, we keep a static copy of a database, and access is given by a method that first checks if a copy exists and returns it:

```
private static Database data = null;
private Database(){
}
public static Database data(){
    if(data==null){
        data = new Database();
    }
    return data;
}
```

      iii. Benefits: gives us control over access and use of the class.
      This design pattern is only relevant in the specific cases when we want exactly one concrete instance. In other cases, like with the controller, having an abstract class with static methods is a better solution.
      iv. Code reuse and maintenance: the singleton means that we can write the code for the specific use, and we don't need to consider how different instances might interact. This also benefits the maintainability, because it makes the system more reliable.

   b. Factory:
      i. Definition and role: A creational design pattern. An object or method used for creating other objects. Useful when we need to create many similar objects, or when we don't know in advance which specific type of object we'll need to create.
      ii. Implementation: when initializing the database, we create many hotels and rooms without specifying each one.
      iii. Benefits: good for code reuse and convenient, and therefore also beneficial for maintenance.

   c. Strategy:
      i. Definition and role: A behavioral design pattern. When we want more than one implementation of the same general thing (for example, a search algorithm) but we want the different kinds to be interchangeable and we want an option to add more types. A "family" of methods or algorithms.

ii. Implementation: the user can decide how to sort the hotel options – by price, by stars, etc. This is the same general action – sorting the array of relevant hotels – but it happens slightly differently in each one. So, we have a general "Sort" class, which is extended by the specific sort classes (SortByStars, SortByPrice). During the search process, the "sort" method is called, and it handles the specific type of sort that should happen. There is no need to write a different option for each type of sort in the Search class.

```
public void sort(String parameter){
    SortInterface sort = new Sort();
    switch (parameter){
        case "stars" -> sort = new SortByStars();
        case "price" -> sort = new SortByPrice();
    }
    sort.sort(this.hotels);
}
```

example 2: the guest can decide how to pay, if the hotel has different options. There are different methods for each type of payment, but these are invisible to the user. Only the "menu" method is called by the main class, and the decision of which method to use to get the payment information happens privately based on the user's input.

```
String[] payInfo = getPaymentInfo(Integer.parseInt(input), guest,
hotel);

private static String[] getPaymentInfo(int paymentOption, Guest guest,
Hotel hotel)
        throws PaymentErrorException {
    String[] s;
    switch (paymentOption){
        case BANK_TRANSFER -> s = payBankTransfer(guest, hotel);
        case CREDIT_CARD ->  s = payCreditCard(guest, hotel);
        case BIT_PAY -> s = payBitPay(guest, hotel);
        default -> throw new PaymentErrorException("no such payment
                                                    option");
    }
    return s;
}

private static String[] payCreditCard(Guest guest, Hotel hotel){
...
```

iii. Benefits: this makes the code more flexible. We can easily add more sort options without making major changes to the code. This also is an example of encapsulation – the user doesn't need to know if the sort is happening by different algorithms or how it is implemented. The user just chooses the sort parameter. This uses polymorphism – we refer to the sort options as a general "Sort" type. However, this may not always be the correct approach. It might be simpler to find a solution that doesn't require creating extra classes. Or in some cases we might want the user to have more control over the process.

iv. Code reuse and maintenance: there is a potential for code reuse, if we had methods that we wanted to write in the general Sort class and have them be used by all the specific sort classes. This also helps maintenance – if the code is shared, we only need to fix it once.

d. Façade:
   i. Definition and role: A structural design pattern. Having a UI that separates the user from the working methods. This lets the user see what they need to, without the UI being confusing or giving access to things we would rather keep hidden.
   ii. Implementation: the "controller" class acts as the façade. The user has no interaction with any other class. It is the only class that has input or output.
   iii. Benefit: keeps the UI simple and keeps the program secure – a user can't do things that we don't offer.
   Disadvantage: limits what the user can do and requires writing the working methods in a certain way (sometimes it's easier to have the output and input directly in each method).
   iv. Code reuse and maintenance: lets us add, change or remove parts of the code (or all of it) and as long as the façade works in the same way, there is no effect on the user.

e. Iterator:
   i. Definition and role: A behavioral design pattern. Methods to iterate through a collection, without exposing how it works or giving access to its private fields.
   ii. Implementation: in the 'MyDate' class, we have the 'next' method that gives the next date:

```
public MyDate next(){
    int day = this.day==31 ? 1 : this.day+1;
    int month = day==1 ? (this.month==12 ? 1 : this.month+1) :
              this.month;
    int year = (day==1 && month==1) ? this.year+1 : this.year;
    return date(year, month, day);
}
```

   iii. Benefit: This is useful for figuring out what the next date is – it puts the responsibility on the date class instead of on each method that needs to know. This allows code reuse – we write the method once and it is used in multiple places. This also makes maintenance easier, and is an example of encapsulation. This is not always necessary – sometimes a collection might be simple to traverse, or we might want the user to have more control over the exact traversal method.
   iv. Code reuse and maintenance: since getting the "next" date is not a straightforward step, having an iterator saves us the trouble of writing the same method in every place that we need to traverse dates. This is also good for maintenance – we only need to fix one method, instead of checking that it works in every place that we iterate over dates.

f. Observer:
   i. Definition and role: when we want a way to "push" information to multiple users, instead of it being necessary for them to continuously check.
   ii. Implementation: each user implements the 'subscriber' interface, allowing us to treat them all as observers and push information to them.

```
public interface Subscriber {
    void notify(String message);
```

```
public class Guest extends Client implements Subscriber {
```

   iii. Benefits: allows for more convenient user interaction and allows us to push the information to all users without needing to plan for each type separately.
   iv. Code reuse and maintenance: allows us to reuse the sender's code for sending information for all user types, so we don't need to write it separately for each type. This makes maintenance easier since there is only one implementation. The sender only needs a method to notify a "subscriber". It doesn't matter if the subscriber is a guest of a specific type or something new entirely.

g. Decorator:
   i. Definition and role: A structural design pattern. An object or method that allows us to dynamically add features or behaviors to objects, in a variety of configurations.
   ii. Implementation: we can add and remove different features in a room, and we don't need to create a different one each time. We don't have a "room with balcony" or a "room with hot tub and mini fridge". We have a single type of room, and we can add or remove amenities as needed.
   iii. Benefits: Code reuse and maintenance. Reduces the amount of code we need to write and allows for easier change and maintenance of the code.

h. Model – view – controller:
   i. Definition and role: A structural/architectural design pattern. The main part of the code – the logic – is the "model". This part is hidden from the user. The user interacts with the "view" (output) and the "controller" (input). The controller manipulates the model.
   ii. Implementation: the aptly named 'Controller' class is the controller. It is the only class that gets input from the user and sets the output. It calls the necessary methods from the rest of the classes – the "model". The terminal acts as the "view".
   iii. Benefit: this also acts as a façade, making the user's experience much easier. Since the methods in the model don't take direct user input, they can be used in other UI's or applications. We just need to add an appropriate controller. Drawback: this puts constraints on the methods themselves – all the information needs to be passed as parameters. We can't count on user input inside the method.
   iv. Code reuse and maintenance: since the model is generic, it can be used with different UI's (for example, the same database logic can be used by a website and

a mobile app). This is also good for maintenance – we can change things without the user noticing, and it splits the code into clear areas of responsibility.

i. Flyweight:
   i. Definition and role: A structural design pattern. If we have many objects that are identical, we can have many references to the same instance instead of many duplicates of the same object.
   ii. Implementation: the "MyDate" class: we'll potentially have hundreds of thousands of rooms, and each room stores the dates that it is reserved on. It is very likely that we'll be using the same date many times. So, instead of having a new object for each date in each room, the "MyDate" class uses the flyweight method to keep only one instance of each specific date. The class keeps a static collection of the existing dates. The constructor is private, and there is a public method that returns the requested date. It first checks if the date exists, and only creates a new one if necessary.

```
private MyDate(int year, int month, int day){
    if(!dates.containsKey(year)){
        dates.put(year, new MyDate[12][31]);
    }
    this.year = year;
    this.month = month;
    this.day = day;
    dates.get(year)[month-1][day-1] = this;
}
public static MyDate date(int year, int month, int day){
    if(!dates.containsKey(year)){
        dates.put(year, new MyDate[12][31]);
    }
    if(dates.get(year)[month-1][day-1]!=null){
        return dates.get(year)[month-1][day-1];
    }
    else dates.get(year)[month-1][day-1] = new MyDate(year, month, day);
    return dates.get(year)[month-1][day-1];
}
```

the dates are saved in such a way that checking if a date exists happens in an efficient way, so runtime is unaffected.

   iii. Benefit: reduces memory usage. Drawbacks: may increase runtime and is only relevant if we are likely to create the same object many times.

j. Builder:
   i. Definition and role: A creational design pattern. Allows us to build an object step by step, instead of having multiple different constructors.
   ii. Implementation: The process of creating a new hotel. Rooms and amenities can be added and removed freely and separately.
   iii. Benefits: allows for a simpler, shorter code and gives flexibility. The shorter code also makes maintenance easier.

k. Composite:
    i. Definition and role: A structural design pattern. Creating objects with a tree hierarchy – each object contains other objects.
    ii. Implementation: in this project, a city contains hotels which contain rooms which have amenities.
    iii. Benefits: gives the code a structure that is easy to understand and makes the building process more intuitive. These also make the maintenance easier. It also allows for polymorphisms, because we can have hotels that are different (but still the same class) instead of having different classes for each type of hotel.

l. Chain of responsibility:
    i. Definition and role: A behavioral design pattern. Requests get passed along a chain of commands. Each step can complete the request, pass it on, or change it and pass it on.
    ii. Implementation: the reservation process "travels" from the controller to the hotel, which creates a reservation with the guest. Inside each class, multiple methods are used.
    iii. Benefits: splitting the process into steps makes it easier to code, debug, modify, and maintain. This also allows code reuse because some of the methods can be used in different processes.

5) Generics and collections:
    a. Generics are useful because they allow us to write code once and reuse it with different data types. The HashMap and ArrayList data structures in Java are implemented with generics. This allows us to create collections of almost any type, using the same structure so we don't have to write a new one for each data type.
    b. Generics were essential in this application for using the HashMap and ArrayList data structures.

6) Exception handling:
    a. Each method is as robust as possible; there are checks to avoid runtime exceptions and bad inputs, and the internal logic should avoid exceptions (for example, an exception will be thrown if a room is being reserved on a date that is already taken – but the reserve method in the hotel has checks to avoid suggesting a room that is reserved on the requested dates). Ideally, the methods can handle any input they get – the controller should make sure of that. At every stage where there still might be a problem, the method throws an exception that is meant to be caught by the controller class. Exceptions are also a way to communicate between methods – for instance, the payment methods can throw an exception that gives information about what went wrong. This allows the controller to let the user know what happened even if the error happened in a different class. The controller is designed to avoid exceptions too – it has checks and loops to handle bad inputs. If an exception is thrown, the controller prints a relevant message and goes back to the previous menu.

b. A specific example of exception handling is in the controller method that makes a reservation. If for some reason the hotel suggested a room that is taken, an exception will be thrown. Or there might be a problem with the payment. The method needs to handle the exception, printing an error message that tells the user what went wrong and letting the user try again without the program crashing.

7) Code optimization and efficiency:
   a. Methods and strategies to optimize code performance and efficiency:
      i. When searching for relevant room options, we filter in a top-down approach – eliminating first by city, then eliminating hotels based on star rating and price range that are quick to check. That way, we can remove many irrelevant options efficiently. Then, each hotel provides room suggestions. This is also done efficiently, because the hotel doesn't need to check all the rooms – once the requested amount is reached, it can return them without checking all the other rooms.
      ii. The "MyDate" class utilizes the flyweight pattern, minimizing memory usage. The tradeoff is runtime. This is minimized by using a HashMap to store each year (for fast lookup time) and in each year the dates are stored in a 2D array. Considering that in a full-scale system, in all probability there will be at least one room reserved on each day of the year, this makes very good use of memory. The HashMap of years gives a constant runtime (especially considering that the number of years is virtually constant and low), and the array gives a constant lookup time as well.
      iii. In general, HashMaps are used as much as possible, minimizing search times. In places where the collection is small or not searched often, ArrayLists can be used. If it makes sense for the collection to have an order – for example, the list of dates in a room – an ArrayList is more convenient than a HashMap. It's a trade-off with efficiency.

   b. In general, a more readable code will be easier to maintain. However, if the code is readable at the expense of flexibility, this will make maintenance harder if any changes are necessary. For example, using generics can make the code more complex, because we need to account for more options. Writing code for a specific data type is much easier but will require duplicating the code if support of a new data type is required.
   Readability and ease of maintenance do not guarantee efficiency; in fact, making the code as efficient as possible means that the code must be designed for a very specific purpose, and this makes the code difficult to change or maintain. A very efficient code will also probably be less readable, because it might have multiple actions condensed into the same method or use specific data structures that are more efficient but difficult to use. Spreading things out makes it easier to understand the code but requires more memory and possibly runtime.
   Focusing only on efficiency can make a code unreadable, or too specific and therefore not capable of scalability, upgrading, or maintenance. But if the code is unlikely to need to be changed, this might be the correct approach.
   Code that is designed only for maintenance would need to be extremely modular and generic. Too much of this makes the code inefficient and unreadable.

A code that is written mainly for readability might have drawbacks in efficiency, but code that is unreadable is also difficult to maintain, because mistakes are more likely to happen and harder to fix.
On most modern systems (especially if we're running Java or some other high-level OOP language), the memory is not such a scarce resource that we need to code for extreme memory efficiency, and the runtime is unlikely to be greatly affected by small changes in implementation. If we write code that runs asymptotically longer, this is different; but if the runtime is asymptotically equivalent, we are unlikely to see a big difference. Code should be readable, but this is easier to fix with good commenting, as opposed to maintainability.
In conclusion, any one of the attributes will come at the expense of the others. Sometimes two of them will work well together at the expense of the third. It is all a matter of what the priorities are in the project. In general, it seems that a rule of thumb could prioritize maintainability first, then asymptotic performance, then readability, and finally "tweaking" the code for optimal performance.

8) Testing and debugging:
    a. The general approach is to test as you code. Each method that is written is tested to make sure it works as intended before moving on to the next method. And of course, testing how the methods work together. Testing more often minimizes the chances of a bug that will require rewriting the code. The testing can happen with a specific testing class, or by experimenting in the console. Pretending to be an "annoying" user is a great way to catch bugs that might have slipped through the tests.
    b. The size and complexity of the project presents a challenge: Having many classes and methods that interact with each other raises the chances of bugs. The way to overcome this challenge is to plan ahead as much as possible, so that the general outline is clear even if the code isn't yet written. Knowing in advance what the method signatures are is a big advantage. If we know what each method should get and return and that doesn't change, then fixing the actual internal implementation is more manageable.
    c. Some of the design patterns give the project a clearer structure (like builder, composite, or MVC), and this makes the code more readable and easier to write, test, and debug. Some of them present restrictions and demands (like flyweight or façade). They require that the code be written in a more specific way, and this makes the code more complex and therefore harder to write, test, and debug.