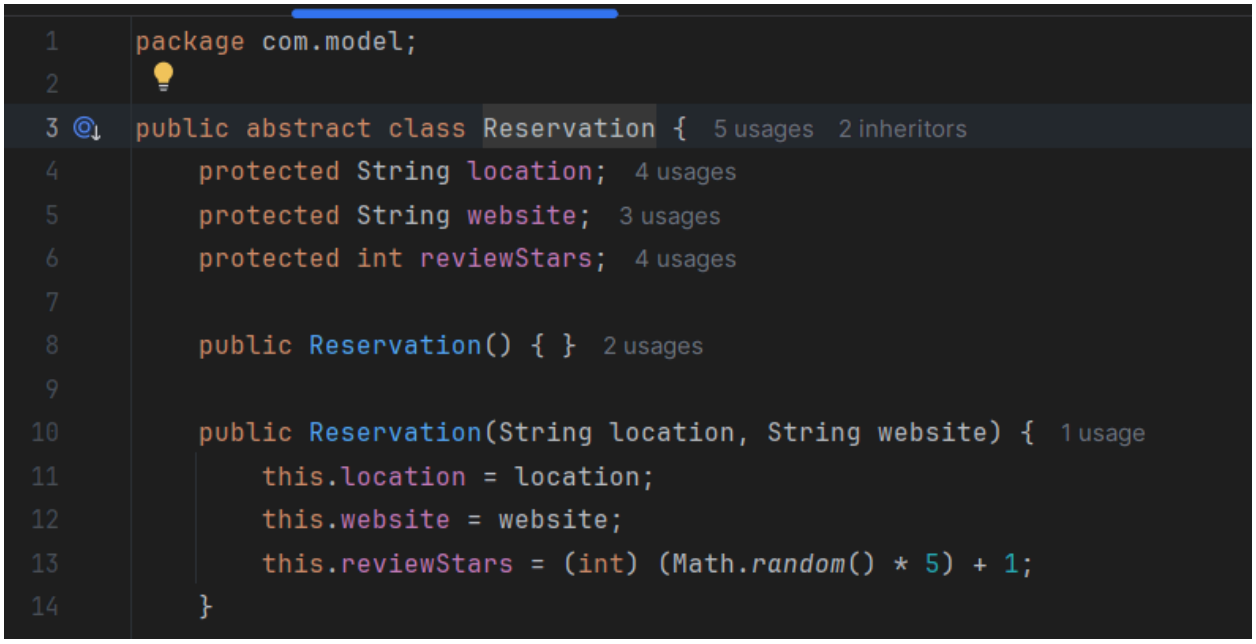
Decorator Pattern: Reservation class -> Dining reservation, Accommodation reservation

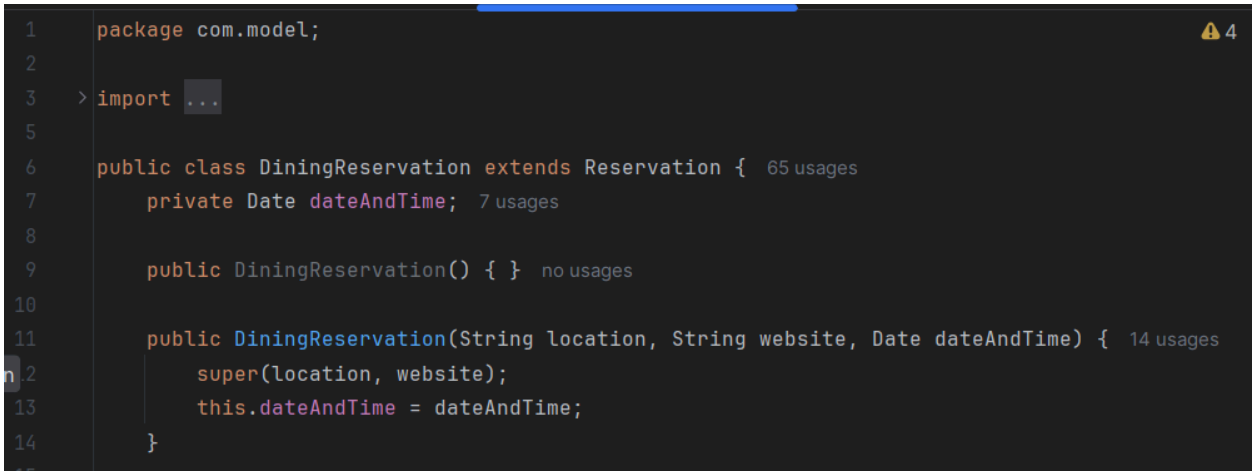


```java
package com.model;

public abstract class Reservation {  5 usages  2 inheritors
    protected String location;  4 usages
    protected String website;  3 usages
    protected int reviewStars;  4 usages

    public Reservation() { }  2 usages

    public Reservation(String location, String website) {  1 usage
        this.location = location;
        this.website = website;
        this.reviewStars = (int) (Math.random() * 5) + 1;
    }
```



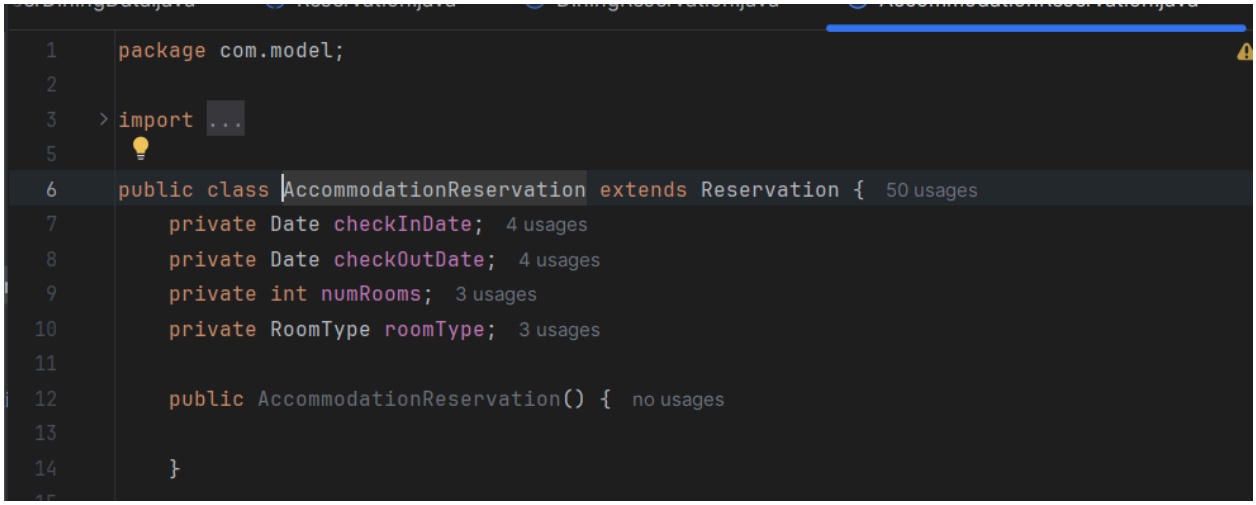
```java
package com.model;  4

import ...

public class DiningReservation extends Reservation {  65 usages
    private Date dateAndTime;  7 usages

    public DiningReservation() { }  no usages

    public DiningReservation(String location, String website, Date dateAndTime) {  14 usages
        super(location, website);
        this.dateAndTime = dateAndTime;
    }
```



```java
package com.model;

import ...

public class AccommodationReservation extends Reservation {  50 usages
    private Date checkInDate;  4 usages
    private Date checkOutDate;  4 usages
    private int numRooms;  3 usages
    private RoomType roomType;  3 usages

    public AccommodationReservation() {  no usages

    }
```

The code shown above showcases the decorator pattern. We implemented a base Reservation Decorator class that contains the basic information all reservation datatypes should hold: location, website, and reviewsStars. And that class gets extended by both the AccommodationReservation and DiningReservation classes. Each of those classes needs different attributes while keeping the common attributes. Therefore, using a decorator pattern works best with this situation as DiningReservation gets to have a dateAndTime attribute and the AccomodationReservation needs checkInDate, checkOutDate, numRooms, and roomType. By wrapping these different classes on top of the base Reservation class, we get to utilize the benefits of a Decorator pattern. And if we ever need to expand our app to include other types of reservations, it can be easily done so by creating additional reservation wrapper classes.

Observer Pattern

```java
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @
    View view = inflater.inflate(R.layout.fragment_community, container, attachToRoot: false);

    // ViewModel initialization
    communityViewModel = new ViewModelProvider( owner: this).get(CommunityViewModel.class);

    // Set up RecyclerView
    RecyclerView recyclerView = view.findViewById(R.id.recyclerView);
    recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));
    final PostAdapter adapter = new PostAdapter();
    recyclerView.setAdapter(adapter);

    // Observe posts data
    communityViewModel.getPosts().observe(getViewLifecycleOwner(), adapter::submitList);

    // Set up FloatingActionButton
    FloatingActionButton fabAddButton = view.findViewById(R.id.fabAddButton);
    fabAddButton.setOnClickListener(v -> openCreatePostDialog());

    return view;
}
```

```java
private void openCreatePostDialog() {  1 usage
    CreatePostDialog dialog = new CreatePostDialog(post -> communityViewModel.addPost(post));
    dialog.show(getParentFragmentManager(),  tag: "CreatePostDialog");
}
```

```java
public class CommunityViewModel extends ViewModel {  3 usages
    //temporary just to make view functional
    private final MutableLiveData<List<Post>> posts = new MutableLiveData<>(new ArrayList<>());

    public CommunityViewModel() {  no usages
        // Populate with default values
        initializeDefaultPosts();
    }

    public LiveData<List<Post>> getPosts() {  1 usage
        return posts;
    }

    public void addPost(Post post) {  1 usage
        List<Post> currentPosts = posts.getValue();
        if (currentPosts != null) {
            currentPosts.add(post);
            posts.setValue(currentPosts);
        }
    }
```

We have the post class to be observable, this way we can have the most updated version of the posts shown in the view. Whenever the viewModel notices that the view has taken in a new post, which is when the user decides to create a new post in the community board, it knows to update the post class and show the most recent version of it. The connection between the Community view class, Community viewModel, and post class is showcased by line 39 in the first screenshot. The viewModel handles the update as that's where we put the work of actually updating the view. This is shown directly through the getPosts() method as it gives the view the most updated version of the posts.