JAVA WEB DEVELOPMENT

LESSON

Understanding SOLID Principles:

Single Responsibility

SOLID

As *a small reminder, in* **SOLID** there are five basic principles which help to create good (or solid) software architecture. SOLID is an acronym where:-

S stands for **SRP** (**Single responsibility principle**)

O stands for **OCP** (Open closed principle)

L stands for **LSP** (Liskov substitution principle)

I stand for **ISP** (Interface segregation principle)

D stands for **DIP** (Dependency inversion principle)

A CLASS SHOULD HAVE ONE, AND ONLY ONE, REASON TO CHANGE.

Robert C. Martin

SINGLE RESPONSIBILITY PRINCIPLE

Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

The classes you write, should not be a swiss army knife. They should do one thing, and to that one thing well.

WHY IS THAT THIS PRINCIPLE IS REQUIRED?

- Imagine designing classes with more than one responsibility/ implementing more than one functionality. There's no one stopping you to do this. But imagine the amount of dependency your class can create within itself in the due course of the development time. So when you are asked to change a certain functionality, you are not really sure how it would impact the other functionalities implemented in the class. The change might or might not impact other features, but you really can't take risk, especially in production applications. So you end up testing all the dependent features.

CORRECT WAY

- You might say, we have automated tests, and the number of tests to be checked are low, but imagine the impact over time. These kind of changes get accumulate owing to the viscosity of the code making it really fragile and rigid.

- One way to correct the violation of SRP is to decompose the class functionalities into different classes, each of which confirms to SRP.

BENEFITS OF THE SINGLE RESPONSIBILITY PRINCIPLE

- It makes your software easier to implement and prevents unexpected side-effects of future changes.

A SIMPLE QUESTION TO VALIDATE YOUR DESIGN

- *Unfortunately, following the single responsibility principle sounds a lot easier than it often is.*
- **Problem:** If you build your software over a longer period and if you need to adapt it to changing requirements, it might seem like the easiest and fastest approach is adding a method or functionality to your existing code instead of writing a new class or component. But that often results in classes with more than responsibility and makes it more and more difficult to maintain the software.

ADVICE/SOLUTION

- You can avoid these problems by asking a simple question before you make any changes: What is the responsibility of your **class/component/microservice**?
- If your answer includes the word "**and**", you're most likely breaking the single responsibility principle. Then it's better take a step back and rethink your current approach. There is most likely a better way to implement it.

SHOPPINGLIST

As we can see "ToDoListApplication" provides many of functionalities like printing information, asking for user input, saving domain objects in database and etc

Functionality:

1. Print menu
2. Add task
3. Find task

WHAT WE CAN DO?

Correct answer will be - decompose!

DECOMPOSE


TODOLIST APPLICATION


TODOLIST APPLICATION
VIEW
SERVICE
DATABASE

REFACTORING

DECOMPOSE

- In **ToDoListApplication** class we can found **createTask** method. This method must be decomposed to smaller parts where responsibility of each part should be entirely encapsulated by the class.

User Input and object creation

```
private static void createTask(Map<Long, Task> tasks) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter task name: ");
    String name = scanner.nextLine();
    System.out.println("Enter task description: ");
    String description = scanner.nextLine();

    Task task = new Task();
    task.setName(name);
    task.setDescription(description);

    if (task.getName() == null) {
        throw new IllegalArgumentException("Task name must be not null.");
    }

    task.setId(TASK_ID_SEQUENCE);
    tasks.put(TASK_ID_SEQUENCE, task);
    TASK_ID_SEQUENCE++;

    System.out.println("Task created, id:  " + task.getId());
}
```

Validation

Saving to collection (database)

SRP

- First of all what we can do - move the task collection to the another “layer” that called “database access layer”

```
public static void main(String[] args) {
    Map<Long, Task> tasks = new HashMap<>();
    while (true) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("1. Create task");
            System.out.println("2. Find task by id");
            System.out.println("3. Exit");
            int userInput = scanner.nextInt();
            switch (userInput) {
                case 1:
                    createTask(tasks);
                    break;
                case 2:
                    findTask(tasks);
                    break;
                case 3:
                    return;
            }
        } catch (Exception e) {
            System.out.println("Error! Please try again.");
        }
    }
}
```

DATABASE ACCESS LAYER

- As we can see now database absolutely separated from main menu.
- Database must provide simple CRUD operations (without business logic)

```
public class TaskInMemoryRepository {

    private Long TASK_ID_SEQUENCE = 0L;
    private Map<Long, Task> tasks = new HashMap<>();

    public Task insert(Task task) {
        task.setId(TASK_ID_SEQUENCE);
        tasks.put(TASK_ID_SEQUENCE, task);
        TASK_ID_SEQUENCE++;
        return task;
    }

    public Task findTaskById(Long id) {
        return tasks.get(id);
    }
}
```

BUSINESS LAYER

- As we can see now **TaskService** provides simple functionality which can be easy **tested (or not?)**.
- Services (**Business Layer**) can provide functionality that includes business logic.

```
public class TaskService {

    private TaskInMemoryRepository repository = new TaskInMemoryRepository();

    public Long createTask(Task task) {
        if (task == null) {
            throw new IllegalArgumentException("Task must be not null.");
        }
        if (task.getName() == null) {
            throw new IllegalArgumentException("Task name must be not null.");
        }
        Task createdTask = repository.insert(task);
        return createdTask.getId();
    }
}
```

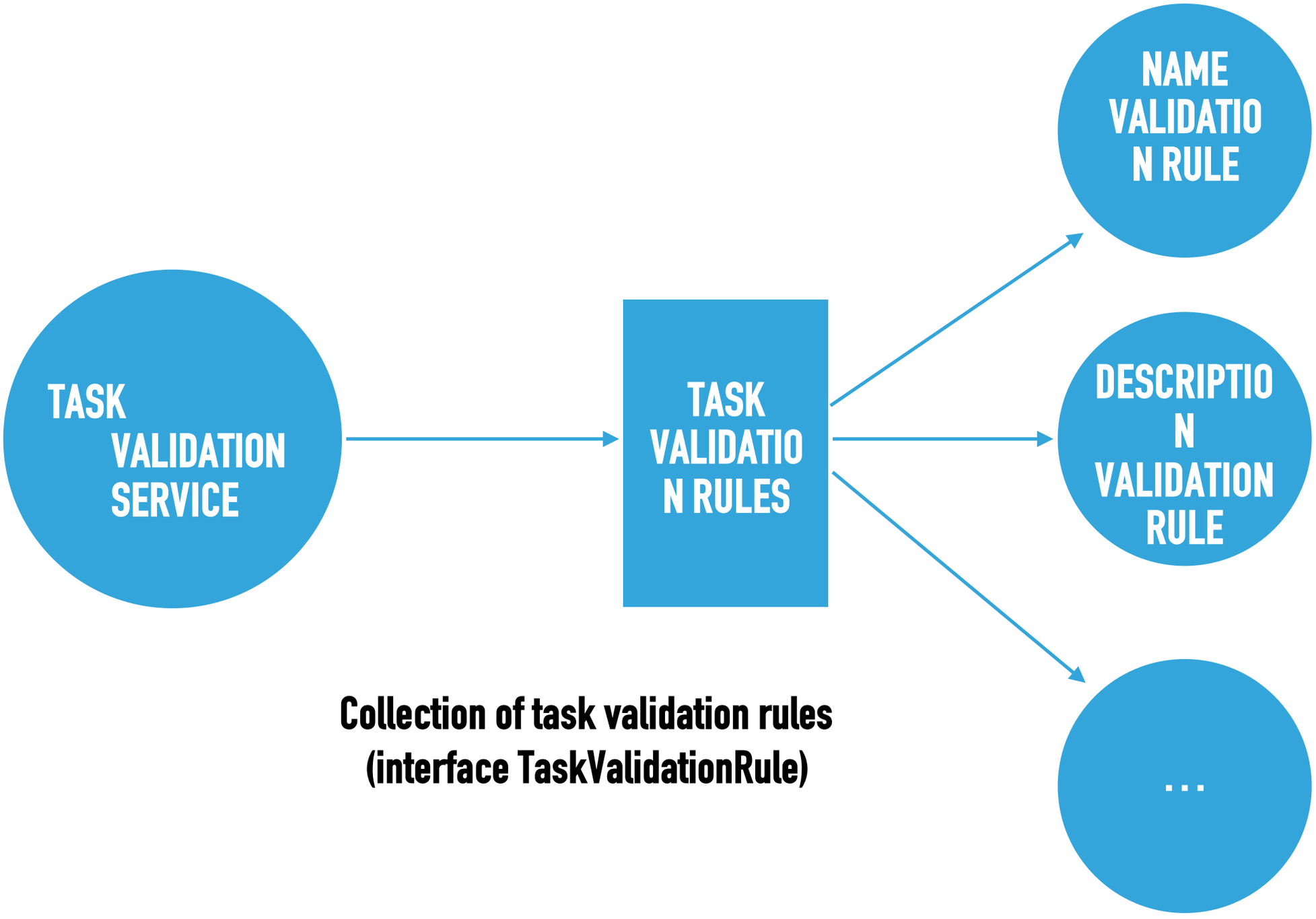
```java
public class TaskService {

    private TaskInMemoryRepository repository = new TaskInMemoryRepository();

    public Long createTask(Task task) {
        if (task == null) {
            throw new IllegalArgumentException("Task must be not null.");
        }
        if (task.getName() == null) {
            throw new IllegalArgumentException("Task name must be not null.");
        }
        Task createdTask = repository.insert(task);
        return createdTask.getId();
    }
}
```

Validation

TaskValidationRule implementation



Collection of task validation rules (interface TaskValidationRule)

EXAMPLE

```java
public interface TaskValidationRule {

    void validate(Task task);

    default void checkNotNull(Task task) {
        if (task == null) {
            throw new TaskValidationException("Task must be not null");
        }
    }
}
```

Implementation

```java
public class TaskNameValidationRule implements TaskValidationRule {

    @Override
    public void validate(Task task) {
        checkNotNull(task);
        if (task.getName() == null) {
            throw new TaskValidationException("Task name must be not null.");
        }
    }
}
```

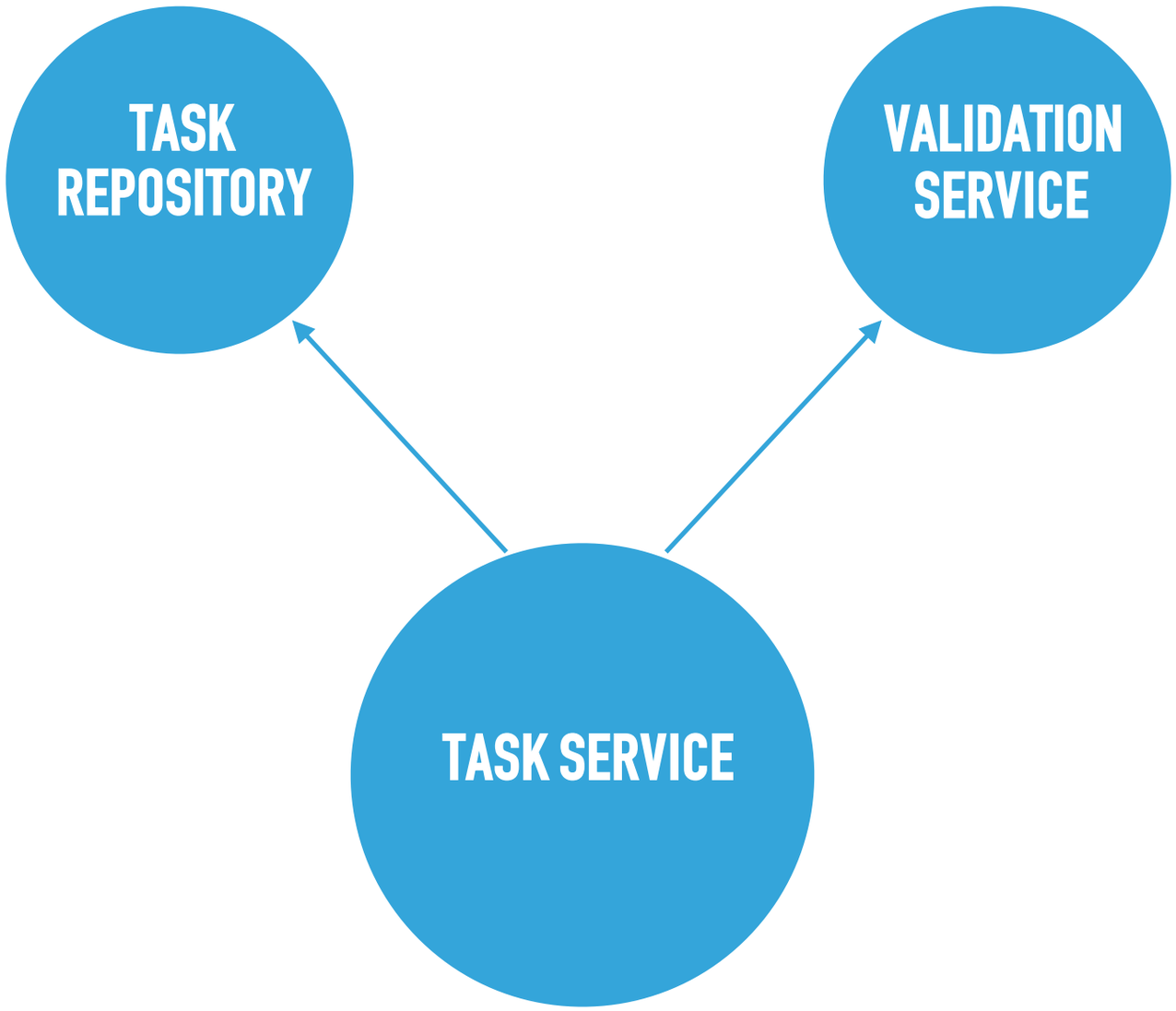
```
class TaskValidationService {

    private Set<TaskValidationRule> validationRules = new HashSet<>();

    public TaskValidationService() {
        validationRules.add(new TaskNameValidationRule());
    }

    public void validate(Task task) {
        validationRules.forEach(s ->s.validate(task));
    }

}
```


TASK REPOSITORY
VALIDATION SERVICE
TASK SERVICE

TaskService after refactoring

```
public class TaskService {

    private TaskInMemoryRepository repository = new TaskInMemoryRepository();
    private TaskValidationService validationService = new TaskValidationService();

    public Long createTask(Task task) {
        validationService.validate(task);
        Task createdTask = repository.insert(task);
        return createdTask.getId();
    }
}
```

VIEW LAYER

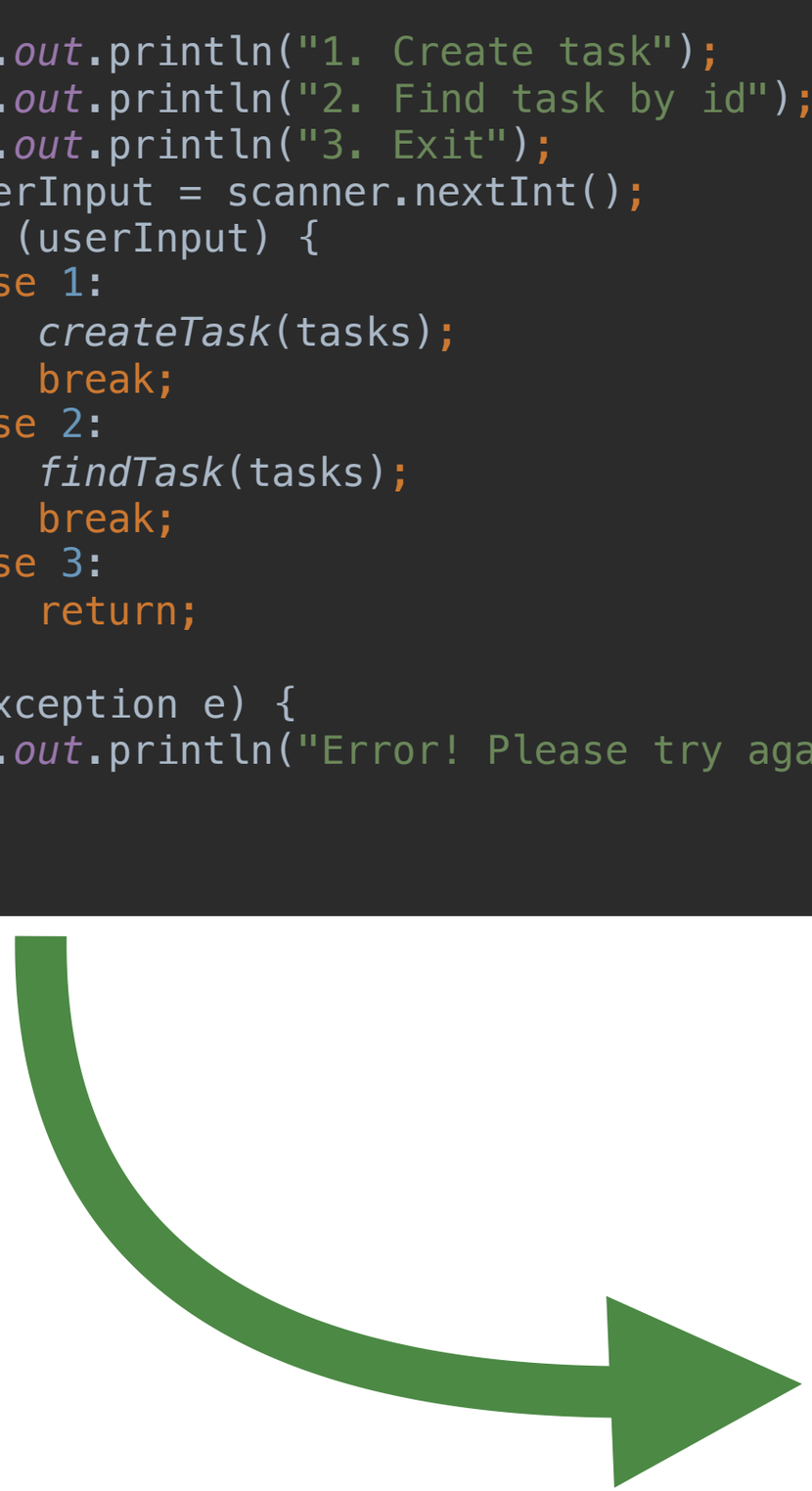
- And last part of decomposing our application is “View Layer”. All functionality that includes “user inputs/views” can be moved to this layer.

```
class ConsoleUI {

    private TaskService taskService = new TaskService();

    public void execute() {
        while (true) {
            Scanner scanner = new Scanner(System.in);
            try {
                System.out.println("1. Create task");
                System.out.println("2. Find task by id");
                System.out.println("3. Exit");
                int userInput = scanner.nextInt();
                switch (userInput) {
                    case 1:
                        createTask();
                        break;
                    case 2:
                        findTask();
                        break;
                    case 3:
                        return;
                }
            } catch (Exception e) {
                System.out.println("Error! Please try again.");
            }
        }
    }
```

```java
public static void main(String[] args) {
    Map<Long, Task> tasks = new HashMap<>();
    while (true) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("1. Create task");
            System.out.println("2. Find task by id");
            System.out.println("3. Exit");
            int userInput = scanner.nextInt();
            switch (userInput) {
                case 1:
                    createTask(tasks);
                    break;
                case 2:
                    findTask(tasks);
                    break;
                case 3:
                    return;
            }
        } catch (Exception e) {
            System.out.println("Error! Please try again.");
        }
    }
}
```



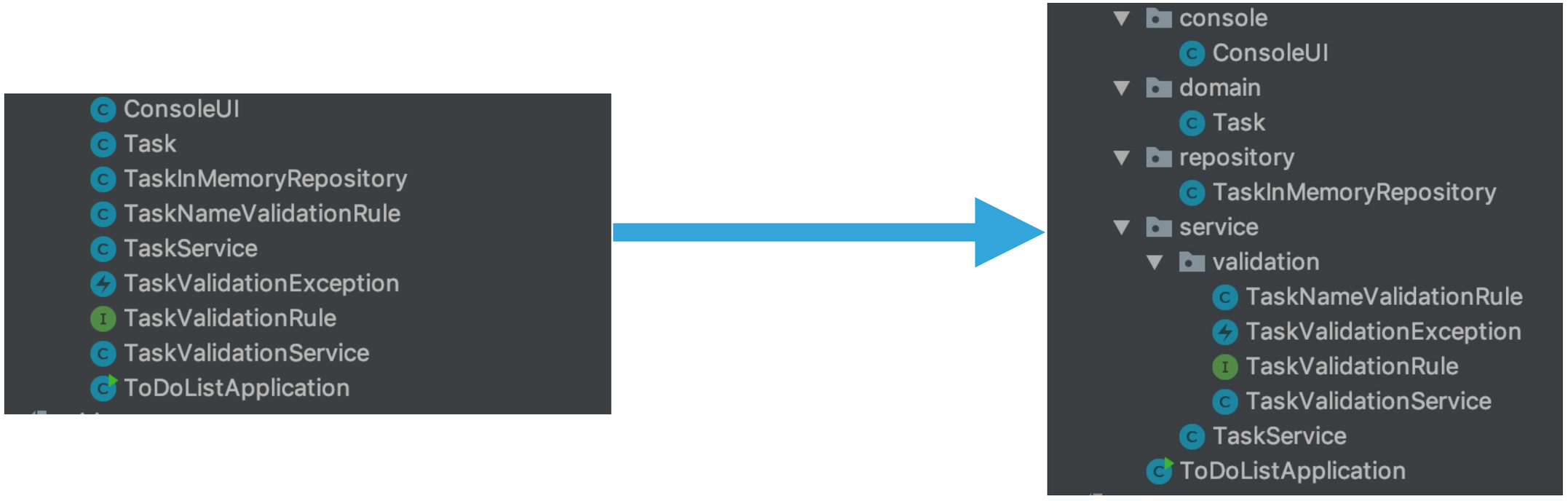
```java
public class ToDoListApplication {

    public static void main(String[] args) {
        ConsoleUI consoleUI = new ConsoleUI();
        consoleUI.execute();
    }
}
```

REPACKAGING

REPACKAGING

- Here we can see that all classes are pooled together in a **“todolist”** list package. This makes it complicated to find a specific class needed. Therefore, classes must be allocated to different packages with clear and understandable names



FINISH

- Now every part have responsibility over a single part of the functionality provided by the software, and that responsibility entirely encapsulated by the classes

REFERENCES

- https://www.javacodegeeks.com/2011/11/solid-single-responsibility-principle.html
- https://dzone.com/articles/single-responsibility-principle-in-object-oriented
- https://springframework.guru/principles-of-object-oriented-design/single-responsibility-principle/
- https://hackernoon.com/you-dont-understand-the-single-responsibility-principle-abfdd005b137
- https://codeburst.io/understanding-solid-principles-single-responsibility-b7c7ec0bf80
- https://hackernoon.com/solid-principles-made-easy-67b1246bcdf
- https://stackify.com/solid-design-principles/