AVISI

$ whoami_

Dirk Groot (46)

Software Architect at Avisi

25+ years of software engineering

Arnhem

Avisi?

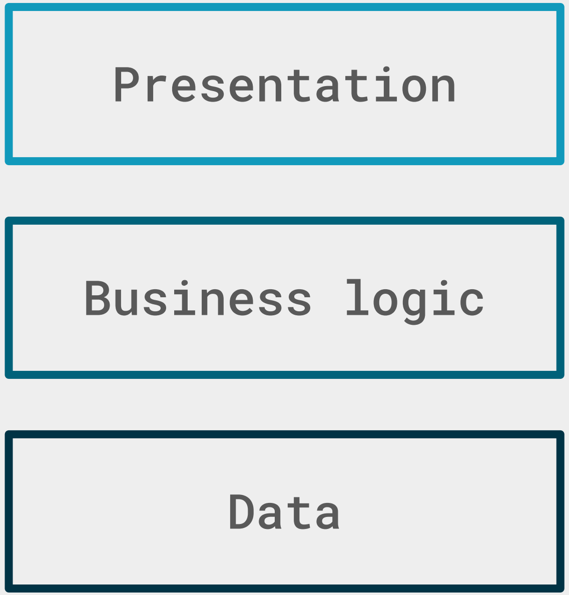
Business Logic in Code.

Pitfalls, Principles and Practices

Architecture.

Architecture.

Often used:
The n-tier architecture



AVISI

Architecture / data

```kotlin
@Entity
data class TodoList(
   @Id val id: UUID,
   var name: String
) {

   @OneToMany(mappedBy = "todoList", cascade = [CascadeType.ALL])
   val items: MutableList<TodoItem> = mutableListOf()

}

@Repository
interface TodoListRepository : CrudRepository<TodoList, UUID>
```

Architecture / data

```kotlin
@Entity
data class TodoItem(
   @Id val id: UUID,
   @ManyToOne
   @JoinColumn(name = "todolist_id", nullable = false)
   val todoList: TodoList,
   var title: String,
   var dueDate: LocalDateTime? = null,
   var done: Boolean = false,
)

@Repository
interface TodoItemRepository : CrudRepository<TodoItem, UUID> {
   fun findByTodoListId(id: UUID): List<TodoItem>
}
```

Architecture / business logic

```kotlin
class TodoListService(val repository: TodoListRepository) {

    fun createTodoList(name: String): TodoList {
        require(name.isNotBlank()) { "Name cannot be blank" }

        val todoList = TodoList(UUID.randomUUID(), name)
        repository.save(todoList)
        return todoList
    }

}
```

Architecture / business logic

```
class TodoListService(val repository: TodoListRepository) {

   fun removeTodoList(todoListToRemove: TodoList) {
       if (todoListToRemove.items.any { !it.done }) {
           throw TodoListNotRemovableException()
       }
       repository.delete(todoListToRemove)
   }

}
```

Architecture / business logic

```
class TodoItemService(/*...*/) {

   fun create(id: UUID, description: String, dueDate: LocalDateTime?): TodoItem {
      if (dueDate != null && dueDate.isBefore(LocalDateTime.now())) {
          throw InvalidDueDateException()
      }

      val todoList = todoListRepository.findById(id).orElseThrow { TodoListNotFoundException() }
      val todoItem = TodoItem(id, todoList, description, dueDate)

      todoList.items.add(todoItem)
      todoListRepository.save(todoList)

      return todoItem
   }

}
```

AVISI

Architecture / business logic

```
class TodoItemService(/*...*/) {

   fun updateDescription(id: UUID, description: String) {
      val todoItem = todoItemRepository.findById(id).orElseThrow { TodoItemNotFoundException() }

      if (todoItem.done) throw IllegalStateException("Cannot update description if todo item is done")
      todoItem.description = description
      todoItemRepository.save(todoItem)
   }

}
```

Architecture / presentation

```
@RestController
@RequestMapping("/api/v1/todo/list")
class TodoListResource(private val todoListService: TodoListService) {

   @PostMapping
   fun create(@RequestParam name: String): ResponseEntity<TodoListRestModel> {
       val todoList = todoListService.createTodoList(name)
       return ResponseEntity.status(HttpStatus.CREATED)
           .body(TodoListRestModel.from(todoList))
   }

}
```

Code review.

Code review / duplication

```kotlin
class TodoListService(val repository: TodoListRepository) {

    fun createTodoList(name: String): TodoList {
        require(name.isNotBlank()) { "Name cannot be blank" }
        // ...
    }

    fun renameTodoList(id: UUID, name: String): TodoList {
        require(name.isNotBlank()) { "Name cannot be blank" }
        // ...
    }

}
```

Code review / public mutable state

```kotlin
@Entity
data class TodoItem(
   @Id val id: UUID,
   @ManyToOne
   @JoinColumn(name = "todolist_id", nullable = false)
   val todoList: TodoList,
   var title: String,                  // !!!
   var dueDate: LocalDateTime? = null, // !!!
   var done: Boolean = false,          // !!!
)
```

AVISI

Code review / bug

```kotlin
class TodoItemService(/*...*/) {

    fun create(id: UUID, description: String, dueDate: LocalDateTime?): TodoItem {
        if (dueDate != null && dueDate.isBefore(LocalDateTime.now()))
            throw InvalidDueDateException()

        val todoList = todoListService.findById(id)
            .orElseThrow { TodoListNotFoundException() }
        val todoItem = TodoItem(id, todoList, description, dueDate)

        todoList.items.add(todoItem)
        todoListRepository.save(todoList)

        return todoItem
    }

}
```

```kotlin
@Entity
data class TodoItem(
   @Id val id: UUID,
   @ManyToOne
   @JoinColumn(name = "todolist_id", nullable = false)
   val todoList: TodoList,
   var title: String,
   var dueDate: LocalDateTime? = null,
   var done: Boolean = false,
)
```

AVISI

Code review / bug

```
class TodoItemService(/*...*/) {

   fun create(id: UUID, description: String, dueDate: LocalDateTime?): TodoItem {
      // ...
      // This is the ID of a todo list, but should be a new ID for the todo item
      val todoItem = TodoItem(id, todoList, description, dueDate)

      todoList.items.add(todoItem)
      todoListRepository.save(todoList)

      return todoItem
   }

}
```

```
@Entity
data class TodoItem(
   @Id val id: UUID,
   @ManyToOne
   @JoinColumn(name = "todolist_id", nullable = false)
   val todoList: TodoList,
   var title: String,
   var dueDate: LocalDateTime? = null,
   var done: Boolean = false,
)
```

Smells.

AVISI

Smells.

Primitive obsession

Definition

Using primitives instead of small objects for simple tasks

Consequences

- Defensive programming: Many validations in many places
- Duplication
- Bugs

Smells.

Object orgy /
Inappropriate intimacy

Definition

Classes expose their “private parts” without constraints

Consequences

- Maintaining invariants is scattered across the code base
- Defensive programming: Inputs cannot be trusted
- Bugs

AVISI

Smells.

Low cohesion / high coupling

Definition

Related code is scattered across the code base. Changes are scattered, not isolated.

Consequences

- Code is hard to understand
- Shotgun surgery

AVISI

Principles and practices.

Principle.

“Make illegal states unrepresentable”

Yaron Minsky / Scott Wlaschin

Design types that make it **impossible** to write **compiling** code that introduces an illegal state.

Or: Prefer compile-time validation over runtime validation.

AVISI

Principle.

Use encapsulation

Make classes **exclusively** responsible for maintaining **their own** invariants.

Don’t create public methods that enable collaborators to violate those invariants.

Practice / value object

```kotlin
data class Name(private val value: String) {
   init {
       require(value.isNotBlank()) { "Name cannot be blank" }
       require(value.lines().size == 1) { "Name must have exactly one line" }
   }
}

data class Description(private val value: String) {
   init {
       require(value.isNotBlank()) { "Description cannot be blank" }
   }
}
```

Practice / value object

```
class TodoList(val id: TodoListID, name: Name) {
   fun rename(newName: Name) {
       name = newName
   }
}

val list = TodoList(TodoListID.create(), Name("My todo list"))

list.rename(Description("Description\nwith multiple lines")) // Does not compile
```

Practice / encapsulation

```kotlin
class TodoList(val id: TodoListID, name: Name) {
   private val items = mutableListOf<TodoItem>()
   var name = name
       private set

   val todoItems get() = items.toList()

   fun rename(newName: Name) { name = newName }

   fun addItem(title: Description): TodoItem {
       val item = TodoItem.create(id, title)
       items.add(item)
       return item
   }

   fun canBeDeleted() = items.all { it.state is TodoItem.Done }
}
```

Practice / encapsulation using sum types

```
class TodoItem(val id: TodoItemID, val todoListID: TodoListID, description: Description) {
   var description: Description = description
       private set
   var state: State = Todo()
       private set

   sealed interface State

   inner class Todo : State {
       fun updateDescription(newDescription: Description) { description = newDescription }
       fun markAsDone() { state = Done }
   }
   data object Done : State
}
```

Practice / encapsulation using sum types

```
@PostMapping("/{todoListID}/items/{todoItemID}/update-description")
fun updateItemDescription(
   @PathVariable todoListID: String, @PathVariable todoItemID: String, @RequestParam title: String
): ResponseEntity<Void> {
   val item = findItem(todoListID, todoItemID) ?: return notFound().build()

   // Pattern matching
   return when (val state = item.state) {
      TodoItem.Done -> badRequest().build()
      is TodoItem.Todo -> {
          state.updateDescription(Description(title))
          todoListRepository.updateItem(item)
          noContent().build()
      }
   }
}
```

Practice / encapsulation using sum types

```
return when (val state = item.state) {
   is TodoItem.Todo -> badRequest().build()
   TodoItem.Done -> {
       state.updateDescription(Description(title)) // Does not compile
       todoListRepository.updateItem(item)
       noContent().build()
   }
}
```

Conclusion.

N-tier architecture?

Presentation

👍🏽

Business logic

This is the domain model. Don’t just use services. Also use domain objects.

Data

Data structures in this layer **do not** represent the domain model, but the **data model**! Use this layer only for persistence.

Impact.

Smells

- Not very harmful at first sight
- Big negative impact on maintainability, especially when complexity grows

Principles and best practices

- Easy to apply
- Big positive impact on maintainability, especially when complexity grows

Further reading.

Blog: Designing with types: Making illegal states unrepresentable - Scott Wlaschin

Book: Domain Modeling Made Functional - Scott Wlaschin

Blog: Types + Properties = Software: designing with types - Mark Seemann