Instituto Superior de Engenharia de Lisboa

LEIC

Técnicas de Desenvolvimento de Software

Repetição do Teste Final de Época de Recurso, Inverno de 2021/2022, 17 fevereiro 2022

Nome: Número:

Grupo I

Considerando os temas tratados nas aulas da disciplina, responda às perguntas seguintes assinalando de forma inequívoca a opção correta. Não responda arbitrariamente: cada resposta incorreta desconta 1/3 da cotação da pergunta ao total obtido no grupo. A cotação total do grupo é 8 valores e todas as perguntas têm a mesma cotação.

1. Dadas as seguintes definições:

```
interface F { fun oper(): Int = -1 }
class A: F { override fun oper() = 1 }
class B(val f: F): F { override fun oper() = f.oper()+2 }
val C = object :F { override fun oper() = 0 }
```

A execução do seguinte troço de código:

```
val fx: List<F> = listOf(A(), B(A()), C)
val res = fx.map { it.oper() }
```

- ◯ Produz erro de compilação
- ◯ Coloca em **res** uma lista de inteiros com os valores: `[1, 1, 0]`
- ◯ Coloca em **res** uma lista de inteiros com os valores: `[1, 3, 0]`
- ◯ Instância 3 objetos do tipo **A**

2. Dadas as seguintes definições:

```
typealias F = () -> Int
fun A():F = { 1 }
fun B(f: F) = { -> f()+2 }
val C = { 0 }
```

2.1. Os tipos das funções **A** e **B** são, respectivamente:

- ◯ `() -> Int` e `(F) -> F`
- ◯ `() -> ()->Int` e `(()->Int) -> F`
- ◯ `() -> F` e `(()->Int) -> Int`
- ◯ Nenhuma das outras opções

2.2. A execução do seguinte troço de código:

```
val fx: List<F> = listOf(A(), B(A()), C)
val res = fx.map { it() }
```

- ◯ Produz erro de compilação
- ◯ Coloca em **fx** e em **res** uma lista de funções
- ◯ Coloca em **res** uma lista de inteiros com os valores: `[1, 3, 0]`
- ◯ Chama a função **A** 3 vezes

3. Dada uma classe **C** qualquer, para adicionar uma nova funcionalidade foram consideradas as alternativas:
   a) Alterar a classe **C** acrescentando-lhe uma nova *função interna*
   b) Definir uma *função extensão* do tipo **C**

   Perante estas alternativas qual das afirmações é verdadeira

   ◯ A função interna pode aceder às propriedades públicas da classe **C** e a função extensão não pode.
   ◯ As duas alternativas são totalmente equivalentes.
   ◯ A função interna é a alternativa correta se for necessário que as chamadas sejam polimórficas.
   ◯ Os modificadores *override* e *open* podem ser aplicados nas duas alternativas.

4. Considerando duas implementações da funcão **checkProgrammingCourses**

```
fun checkProgrammingCourses(courses: List<String>): String = when {
   "PG" !in courses -> "Falta PG"
   "AED" !in courses -> "Falta AED"
   "TDS" !in courses -> "Falta TDS"
   else -> "OK: PG,AED,TDS"
}
```

```
fun checkProgrammingCourses(courses: List<String>): String =
   checkCourses(courses, listOf("PG","AED","TDS"))

fun checkCourses(courses: List<String>, required: List<String>): String =
   required.firstOrNull { it !in courses } ?.let { "Falta $it" }
     ?: required.joinToString(separator= ",", prefix= "OK: ")
```

   qual das seguintes frases é <u>falsa</u>

   ◯ A primeira implementação, apesar de violar o princípio DRY, é fácil de compreender.
   ◯ A segunda implementação pode ser pior do que a primeira de acordo com o princípio KISS.
   ◯ Dadas as vantagens e os inconvenientes, é sempre preferível usar a primeira implementação.
   ◯ A segunda implementação é uma solução mais escalável/adaptável para outros cenários.

5. Identifique a frase verdadeira relativamente à utilização de encapsulamento

   ◯ Para ter encapsulamento basta ter classes com funções públicas e as propriedades privadas.
   ◯ O encapsulamento contribui para garantir a validade do estado privado.
   ◯ A ocultação de informação (*information hiding*) e o encapsulamento são técnicas muito distintas.
   ◯ Não é possível usar encapsulamento sem usar exceções.

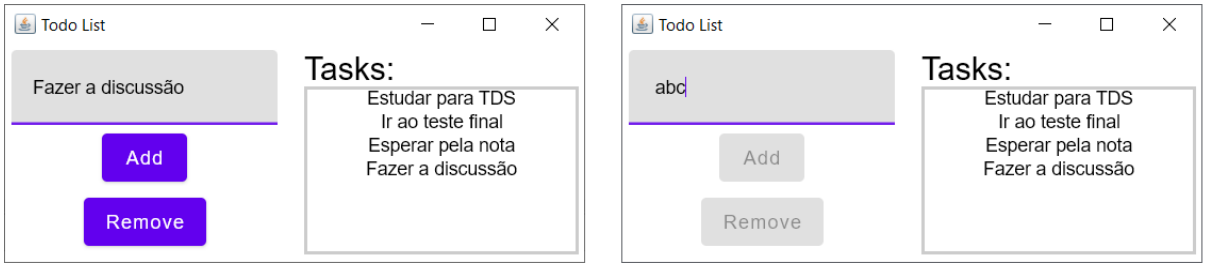
6. O princípio SRP é um dos principais no desenvolvimento de software porque:
   ◯ Permite que sejam realizadas mais verificações em tempo de compilação.
   ◯ Estabelece critérios para dividir o software em várias peças.
   ◯ Evita a repetição da mesma ideia em diferentes partes do código.
   ◯ É suficiente para garantir que a dimensão de cada peça de software não é demasiado grande.

Grupo II

Pretende-se desenvolver uma aplicação para editar uma lista de tarefas a realizar. A aplicação permite adicionar ou remover tarefas da lista. Para simplificar, cada tarefa tem apenas um nome, mas sujeito às seguintes regras: não pode começar ou terminar com espaço e tem comprimento compreendido entre 4 e 40 caracteres. A função `main` da aplicação é a que se apresenta de seguida.

```
fun main() = application {
  val state = WindowState(width= 450.dp, height= 200.dp)
  val todo = remember { mutableStateOf( TodoList() ) }
  Window( onCloseRequest= ::exitApplication, state= state, title= "Todo List" ) {
    Row(Modifier.padding(5.dp)) {
      TaskNameEditor(
        onAdd= { t:Task -> todo.value = todo.value.add(t) },
        onRemove= { t:Task -> todo.value = todo.value.remove(t) }
      )
      StringListView(title= "Tasks:", values= todo.value.tasks.map { it.name })
} } }
```

A figura que se segue apresenta a janela da aplicação em dois possíveis estados: Quando o nome da tarefa em edição cumpre as regras (com botões ativos) ou quando não cumpre as regras (com botões inativos).



1. [2] Tendo em atenção a função `main`, defina os tipos **imutáveis** `Task` e `TodoList`, que representam a tarefa e a lista de tarefas, respetivamente. As instâncias de `TodoList` contêm tarefas (`tasks`) e cada tarefa tem um nome (`name`). Certifique-se que é impossível instanciar tarefas com nomes inválidos (i.e. é lançada `IllegalArgumentException`) e que existe uma função que permite verificar se uma *string* é admissível como nome de uma tarefa.

2. [2] Estenda o tipo `TodoList` com as operações `add` e `remove`, chamadas na função `main`, para adicionar e remover tarefas produzindo uma nova lista de tarefas, tendo em atenção que não podem existir tarefas repetidas. Valorizam-se as soluções que não alterem a definição de `TodoList` realizada no ponto anterior.

3. [2] Crie testes automáticos para verificação da correção das operações `add` e `remove`. Para que a resposta não se torne demasiado exaustiva, implemente apenas dois testes: uma utilização válida de uma operação e uma utilização inválida da outra operação.

4. [2] Implemente o *Composable* `StringListView`, usado na função `main`, para apresentação de uma lista de *strings*, com aspeto semelhante ao da parte direita das janelas da figura.

5. [2] Implemente o *Composable* `TaskNameEditor`, com estado interno (*stateful*) e com aspeto semelhante à parte esquerda das janelas da figura, que permita a edição do nome da tarefa (usando um `TextField`) para adicionar ou remover à lista de tarefas, quando clicado o botão "Add" ou "Remove". Os botões só estão ativos (`enabled`) quando o texto atual cumprir as regras dos nomes das tarefas.

6. [2] Descreva as alterações necessárias para que a lista de tarefas seja carregada do ficheiro *Todo.txt* quando iniciada a aplicação e para que a lista seja posteriormente gravada no mesmo ficheiro quando terminar a aplicação. Considere a pré-existência das funções: `fun readFromFile(fileName: String): TodoList` e `fun TodoList.writeToFile(fileName: String): Unit`

Duração: 90 minutos
ISEL, 17 de Fevereiro de 2022