Teste de Software Aplicado ao Desenvolvimento

Abordagem:

- Criada a classe com a função final: `convert_romano`.
- Criada a classe de teste `RomanoTest` e implementado um teste que falha `test_romano01` (fase vermelha).
- Corrigida a implementação para fazer o teste funcionar (fase verde).
- Código refatorado: implementadas as etapas intermediarias.

Para rodar os testes, executar a seguinte linha de comando:

```
python -m unittest tests_romano.py
```

Qual(is) critério(s) de teste utilizei na elaboração do conjunto de teste?
Usamos a **Técnica Funcional**:
"O teste funcional também é conhecido como teste caixa preta pelo fato de tratar o software como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída".
Usamos o critério **Particionamento em Classes de Equivalência**: Consideramos os domínios de entrada e de saída do programa para estabelecer as classes de equivalência.
Referencias: aurimrv.gitbook.io, Copeland, 2004

De acordo com o apresentado na situação problema, definimos as regras seguintes regras:

- **Regra 1**: Um número romano válido deve ser representado por sete diferentes símbolos, listados na tabela a seguir:

Símbolo	Valor
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

- **Regra 2**: Algarismos de menor ou igual valor à direita são somados ao algarismo de maior valor;
- **Regra 3**: Algarismos de menor valor à esquerda são subtraídos do algarismo de maior valor.
- **Regra 4**: Nenhum símbolo pode ser repetido lado a lado por mais de 3 vezes.

A estratégia foi:

- Testar todas as classes inválidas, ou seja, que não respeitam uma das 4 regras.
- As regras 1 e 4 condicionam o valor da entrada. Então particionamos o domínio da entrada com os 3 primeiros registros da tabela.
- Já as regras 2 e 3 estão verificáveis na saída do algoritmo e não na entrada, ou seja, podemos constatar o respeito delas apenas para os testes de classe válida. Para validar essas 2 regras, particionamos o domínio de saída com os registros 4 e 5 (Copeland, 2004) Exemplo: O registro 5 tem que respeitar a regra 2 por causa do **I** depois do **X**. Já o registro 4 tem que respeitar a regra 3 por causa do **I** antes do **V**.
- Para os testes de classe válida, vamos criar o menor número possível de casos que cubram todas elas.

Número romano	**Regra 1**	**Regra 2**	**Regra 3**	**Regra 4**	**Resultado Esperado**	**Resultado Obtido**
XXXXOVI	inválida	X	X	inválida	inválido	inválido
CCCCII	válida	X	X	inválida	inválido	inválido
UIJ	inválida	X	X	válida	inválido	inválido
CCXIV	válida	X	válida	válida	214	214
XXII	válida	válida	X	válida	22	22
LIV	válida	válida	válida	válida	54	54
MDIV	válida	válida	válida	válida	1504	1504
MMDIX	válida	válida	válida	válida	2509	2509

Quantos testes foram necessários para satisfazê-los?

- 3 testes de classes inválidas.
- 5 testes de classes válidas misturando os 7 símbolos permitidos e aplicando as duas regras, **Regra 2** e **Regra 3**.

Qual o tamanho da solução proposta da primeira versão até a última em termos de linhas de código?
Aparecem 44 linhas de código sem os comentários e as quebras de linha.
Como o **TDD** guia a implementação, ele geralmente ajuda a se manter o mínimo exigido e contribuindo para um código mais limpo e coeso.

Você ficou satisfeito com a solução desenvolvida em termos de sua qualidade? Por quê?
Sim, pois foi implementado um teste para cada uma das classes inválidas e mapeado o conjunto o mais representativo possível das combinações de classes válidas. Em um projeto maior, o conjunto de testes de classe válida tende a ser o menor possível, dado que esse tempo de implementação é um recurso precioso. Novos testes dificilmente garantiriam um código com menos erros pois estes não abordariam nenhuma outra classe.

Na sua opinião, quais são as principais vantagens e desvantagens no uso do TDD até aqui?

- As vantagens são que a implementação fica mais guiada e tende a reduzir os defeitos de omissão e de comissão.

- As desvantagens são que consome mais tempo e a implementação também fica menos flexível, pois algumas funções internas do código nem sempre são previstas ao iniciar a implementação do projeto.

Como não foi solicitado, as funções intermediarias `is_romano_rule_1`, `is_romano_rule_2` e `is_romano` não foram testadas. Em um projeto maior, o ideal é implementar testes unitários para todas as funções, a fim de conseguir a maior taxa de cobertura possível.