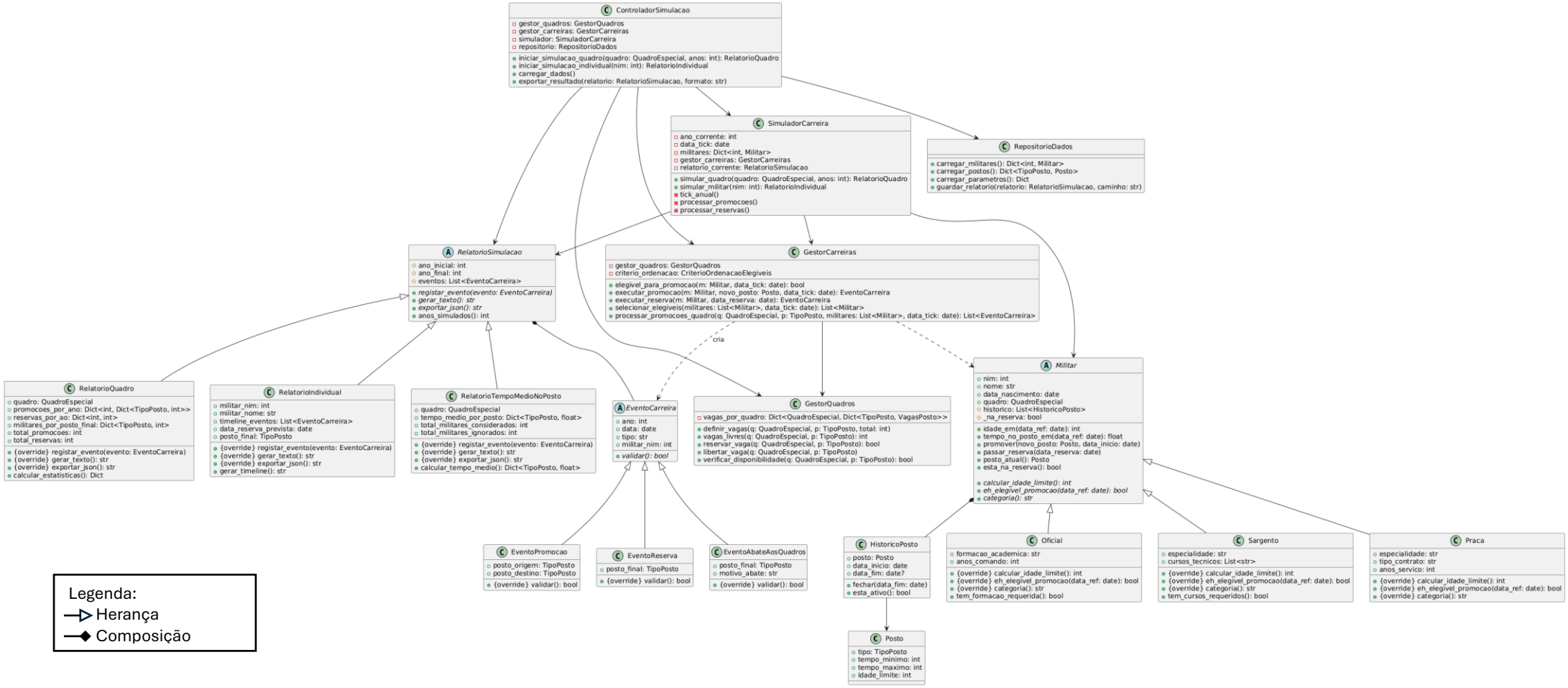
UNIVERSIDADE ABERTA
www.uab.pt

Licenciatura em Eng. Informática
Programação por Objetos 25/26

Proposta de projeto

Simulador de Carreiras

Diagrama de classes



Protótipos em Python

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass
from datetime import date
from typing import List, Optional

@dataclass
class Posto:
    """Objeto simples que representa um posto militar."""
    nome: str
    idade_limite: int

@dataclass
class HistoricoPosto:
    """Elemento do histórico (COMPOSIÇÃO dentro de Militar)."""
    posto: Posto
    data_inicio: date
    data_fim: Optional[date] = None

    def fechar(self, quando: date) -> None:
        """Fecha o período num posto."""
        self.data_fim = quando

class Militar(ABC):
    """Superclasse abstrata. Define o CONTRATO que as subclasses têm de implementar."""
    def __init__(self, nim: int, nome: str, data_nascimento: date) -> None:
        self.nim = nim
        self.nome = nome
        self.data_nascimento = data_nascimento
        # COMPOSIÇÃO: o Militar "tem" um histórico de postos
        self.historico: List[HistoricoPosto] = []

    def idade_em(self, ref: date) -> int:
        """Calcula a idade do militar (em anos) numa dada data de referência."""
        return ref.year - self.data_nascimento.year

    def posto_atual(self) -> Optional[Posto]:
        """Devolve o posto atual, se existir algum registado no histórico."""
        return self.historico[-1].posto if self.historico else None

    def promover(self, novo_posto: Posto, quando: date) -> None:
        """Regista mudança de posto no histórico (usa a composição)."""
        if self.historico:
            self.historico[-1].fechar(quando)
        self.historico.append(HistoricoPosto(novo_posto, quando))

    # CONTRATO: métodos que TODAS as subclasses devem concretizar
    @abstractmethod
    def calcular_idade_limite(self) -> int:
        """Devolve a idade-limite (em anos) para este militar. A implementação concreta deve calcular a idade-limite em função
        do posto atual e da categoria (por exemplo, Oficial ou Sargento). Pré-condição: o militar tem um posto atual válido."""
        ...

    @abstractmethod
    def categoria(self) -> str:
        """Identifica a categoria do militar (por exemplo, 'Oficial' ou 'Sargento'). Este valor é usado pelo código cliente para tratar objetos Militar de
forma polimórfica, sem testar explicitamente o tipo concreto com isinstance ou comparações de classe."""
        ...


class Oficial(Militar):
    """Subclasse concreta que HERDA de Militar."""
    # OVERRIDE: concretiza o contrato definir idade limite
    def calcular_idade_limite(self) -> int:
        posto = self.posto_atual()
        return posto.idade_limite if posto else 60

    # OVERRIDE: concretiza o contrato definir categoria
    def categoria(self) -> str:
        return "Oficial"

class Sargento(Militar):
    """Outra subclasse concreta."""

    def calcular_idade_limite(self) -> int:  # override
        return 58

    def categoria(self) -> str:  # override
        return "Sargento"

# EXEMPLOS DE CRIAÇÃO E USO
if __name__ == "__main__":
    hoje = date(2030, 1, 1)

    # Criar alguns postos
    aspirante = Posto("Aspirante", idade_limite=56)
    alferes = Posto("Alferes", idade_limite=60)
    sargento2 = Posto("2º Sargento", idade_limite=60)

    # HERANÇA: Oficial e Sargento são ambos Militar
    m1: Militar = Oficial(1001, "Ana", date(1990, 5, 10))
    m2: Militar = Sargento(1002, "Bruno", date(1988, 7, 20))

    # COMPOSIÇÃO: adicionar elementos ao histórico
    m1.promover(aspirante, date(2020, 1, 1))
    m1.promover(alferes, date(2025, 1, 1))
    m2.promover(sargento2, date(2022, 1, 1))

    # POLIMORFISMO + OVERRIDE:
    # Chamadas passam sempre pelo mesmo contrato de Militar,
    # mas cada subclasse responde de forma específica.
    for m in (m1, m2):
        print(f"{m.nome} ({m.categoria()})")
        print("  Posto atual:", m.posto_atual().nome)
        print("  Idade em 2030:", m.idade_em(hoje))
        print("  Idade limite:", m.calcular_idade_limite())
        print()
```

Cenários de Evolução

Cenário A: Violação do Princípio Abertas/Fechadas

Descrição: Um dia podemos querer introduzir uma forma de ordenação dos militares elegíveis (por exemplo, deixar de ser só antiguidade e passar a uma ponderação do mérito).

Para implementar isto na forma atual, teríamos de modificar o método de seleção/ordenação em GestorCarreiras, introduzindo novos ramos (if por categoria/quadro), violando o princípio abertas/fechadas. Uma forma de mitigar esta violação do princípio abertas/fechadas seria extrair a lógica de ordenação dos elegíveis para uma estratégia configurável (por exemplo, uma classe ou função de "política de ordenação"), permitindo acrescentar novos critérios sem alterar o código interno do GestorCarreiras.

Avaliação de plausibilidade: ALTA – Em sistemas de gestão de carreiras militares, as regras de ordenação para promoções nem sempre são estáticas e variam entre categorias.

Cenário B: Respeito do Princípio Abertas/Fechadas

Descrição: Um dia podemos querer começar a simular a carreira de um militar da categoria de Praças.

Para implementar isto na forma atual, basta concretizar a subclasse Praca (já prevista na hierarquia Militar), definindo os seus atributos específicos e as implementações de calcular_idade_limite() e eh_elegivel_promocao(). O restante código continua a trabalhar apenas com o tipo base Militar (listas de militares, seleção de elegíveis, relatórios), sem qualquer alteração às superclasses.

Avaliação de plausibilidade: ALTA – É muito provável que um simulador de carreiras militares venha a ser estendido às Praças, pois são essenciais para o dimensionamento global dos efetivos e para o planeamento a médio prazo.

Cenário A: Violação do Princípio de Substituição de Liskov (LSP)

Descrição: Um dia podemos querer introduzir um relatório RelatorioTempoMedioNoPosto, como subclasse de RelatorioSimulacao, que calcula o tempo médio de permanência em cada posto.

Para introduzir esta métrica, a implementação do relatório passa a assumir que todos os Militar têm o HistoricoPosto completo desde o início da carreira. Sempre que, ao processar os eventos da simulação ou ao gerar o texto, o relatório deteta um militar com histórico incompleto, lança uma exceção ou aborta o cálculo, tratando esse caso como um estado inválido do sistema. No entanto, o contrato original de RelatorioSimulacao não exigia histórico completo e funcionava de forma robusta com dados parciais. Desta forma, por passar a depender de uma pré-condição mais forte, quebra-se a substituibilidade: o código passa a poder falhar inesperadamente quando recebe um RelatorioTempoMedioNoPosto, apesar de usar a mesma interface.

Plausibilidade: ALTA – Em contextos de planeamento de efetivos e gestão de carreiras é muito comum querer introduzir métricas mais sofisticadas, como o tempo médio de permanência no posto, a partir de dados históricos.

Cenário B: Respeito do Princípio de Substituição de Liskov (LSP)

Descrição: Um dia podemos querer introduzir um novo tipo de evento, EventoAbateAosQuadros, como subclasse de EventoCarreira, para representar a situação em que o militar é abatido aos quadros.

Esta subclasse acrescenta apenas campos específicos (por exemplo, motivo_abate e posto_final) e implementa validar() de forma semelhante às outras subclasses, verificando apenas a consistência interna dos seus próprios dados (datas válidas, posto final definido). Os relatórios existentes que apenas contabilizam promoções e reservas continuam a funcionar, pois tratam a lista de EventoCarreira de forma genérica e podem simplesmente ignorar eventos de tipo "ABATE_AOS_QUADROS" ou tratá-los como mais um tipo terminal de carreira. Desta forma, qualquer código que opera sobre EventoCarreira pode receber instâncias de EventoAbateAosQuadros sem falhas inesperadas, preservando a substituibilidade e respeitando o LSP.

Plausibilidade: ALTA – Em sistemas de gestão de carreiras militares é comum distinguir entre passagem à reserva, e outras saídas resultantes da atrição no decorrer do serviço, o que é designado por abate aos quadros.

Tabela de Decisões

Situação de Dúvida	Justificação	Aplicação do Princípio Abertas/Fechadas
Usar uma classe abstrata **Militar** com subclasses Oficial, Sargento e Praca, em vez de um único Militar com campo "categoria"?	O estado e os métodos de carreira (NIM, quadro, histórico, promoção, reserva) são comuns; as diferenças estatutárias ficam nas subclasses através de override. Evita if categoria espalhados.	**RESPEITA** – O **Cenário B** (passar a simular Praças) faz-se apenas concretizando a subclasse Praca, sem alterar Militar nem o simulador.
Definir eh_elegivel_promocao() como abstrato em Militar ou centralizar toda a lógica em GestorCarreiras com ramos por categoria/quadro?	Cada categoria conhece as suas próprias regras legais (idade limite, tempos mínimos/máximos), que variam entre Oficiais e Sargentos e Praças. GestorCarreiras foca-se só em vagas e ordenação, chamando o método polimórfico.	**RESPEITA** – Novas regras para uma categoria obrigam apenas a mexer na respetiva subclasse de Militar; a infraestrutura (GestorCarreiras, SimuladorCarreira) permanece fechada.
Guardar apenas postoAtual em Militar ou usar **composição** Militar *-- HistoricoPosto com lista de períodos?	O histórico é um componente natural do Militar, mas com ciclo de vida dependente dele. HistoricoPosto concentra datas de início/fim e permite calcular tempos e trajetórias sem encher Militar de campos.	**RESPEITA** – Novos relatórios ou estatísticas sobre "tempo em cada posto" usam o histórico sem alterar Militar; acrescentam apenas operações de leitura/cálculo sobre a lista de HistoricoPosto.
Manter RelatorioQuadro e RelatorioIndividual independentes ou introduzir superclasse abstrata **RelatorioSimulacao** com métodos abstratos e override?	RelatorioSimulacao define o contrato comum (registar_evento, gerar_texto, exportar_json) e reaproveita a composição com EventoCarreira. Cada relatório só especializa a forma de agregar e apresentar.	**RESPEITA** – Novos relatórios (por tipo de promoção, por idade de saída, etc.) nascem como novas subclasses de RelatorioSimulacao, sem alterações ao simulador, que continua a depender apenas da abstração.
Introduzir uma superclasse abstrata **EventoCarreira** com duas subclasses (EventoPromocao e EventoReserva), em vez de um único EventoCarreira com campo "tipo".	O significado e os dados relevantes de uma promoção (posto de origem/destino, impacto em vagas) são diferentes de uma passagem à reserva (fecho de carreira, impacto definitivo no quadro). Separar em subclasses permite encapsular validações e efeitos específicos sem encher EventoCarreira de if tipo, mantendo cada classe com invariantes mais simples.	**RESPEITA** – Novos tipos de evento (reclassificação, regressão, passagem à disponibilidade, etc.) podem ser acrescentados como novas subclasses de EventoCarreira, sem alterar GestorCarreiras nem RelatorioSimulacao, que continuam a depender apenas da abstração.
Calcular estatísticas diretamente sobre listas de Militar ou usar **composição** RelatorioSimulacao *-- EventoCarreira e trabalhar sobre eventos?	EventoCarreira regista de forma genérica promoções e passagens à reserva, independente da categoria. Os relatórios apenas consomem estes eventos, sem ter de conhecer a estrutura interna de Militar.	**RESPEITA** – Novos tipos de evento ou relatórios específicos podem ser introduzidos por extensão (novas subclasses de EventoCarreira/RelatorioSimulacao). A única futura violação prevista está na forma de ordenação dos elegíveis em GestorCarreiras (Cenário A).

Situação de Dúvida	Justificação	Aplicação do LSP / Ação Tomada
Como introduzir RelatorioTempoMedioNoPosto sem violar o LSP?	A métrica de tempo médio exige mais dados, e existe de quebrar a sustentabilidade por haver um reforço de pré-condição face a RelatorioSimulacao	Calcular a média apenas para militares com dados suficientes e sinalizar/ignorar os restantes, sem reforçar as pré-condições da superclasse.
O relatório pode assumir HistoricoPosto completo para todos os militares?	RelatorioSimulacao funciona com históricos parciais (snapshots, testes); exigir completude reforça a pré-condição.	Manter o contrato flexível; se for necessário histórico completo, validar isso numa fase anterior (carregamento/controlo), não na subclasse.
Onde tratar os dados históricos adicionais (BD/ficheiros) usados nas estatísticas?	Se o relatório fizer I/O nos métodos, introduz falhas externas numa hierarquia que era puramente em memória	Carregar dados via RepositorioDados/fábricas antes da simulação; os relatórios operam apenas sobre o estado já carregado em memória
Como introduzir EventoAbateAosQuadros na hierarquia EventoCarreira?	É um novo tipo de fim de carreira; relatórios antigos não o conhecem e não devem falhar por causa disso	Definir EventoAbateAosQuadros como subclasse com campos e validar() próprios; relatórios genéricos podem tratá-lo como EventoCarreira normal ou ignorá-lo explicitamente

Lista de testes (U-Unitários e de I-Integração)

ID	Tipo	Resumo	Como permite testar um princípio SOLID
TU1	U	RelatorioTempoMedioNoPosto com militares com HistoricoPosto incompleto não lança exceção e ignora/sinaliza esses casos	Testa LSP – o novo relatório continua a aceitar os mesmos estados que RelatorioSimulacao, sem reforçar pré-condições
TU2	U	RelatorioTempoMedioNoPosto usado através de uma referência RelatorioSimulacao gera texto válido sem código especial	Testa LSP – valida que a subclasse pode ser usada como um RelatorioSimulacao genérico, sem if de tipo nem adaptações no cliente
TU3	U	RelatorioTempoMedioNoPosto.registar_evento() processa EventoPromocao e EventoReserva mantendo as mesmas invariantes de RelatorioSimulacao (eventos contados, anos simulados) e apenas acrescenta estatísticas de tempo médio.	Testa LSP/OCP – garante que a subclasse estende o comportamento dos relatórios (enriquece com tempo médio) sem alterar o contrato base nem quebrar código existente
TU4	U	EventoAbateAosQuadros.validar() verifica apenas a consistência interna (datas, motivo, posto_final) tal como outros eventos	Testa LSP – confirma que o novo subtipo de EventoCarreira respeita o mesmo contrato de validação, sem novas falhas ou efeitos laterais
TI1	I	ControladorSimulacao executa a simulação primeiro com RelatorioQuadro e depois com RelatorioTempoMedioNoPosto, sem alterações ao controlador	Testa OCP + LSP – demonstra que é possível adicionar um novo tipo de relatório estendendo RelatorioSimulacao sem modificar o controlador e mantendo a substituibilidade pela mesma interface
TI2	I	Sistema com vários militares (promoções, reservas e abates aos quadros) gera relatórios existentes sem falhas nem branches específicos	Testa LSP – valida que EventoAbateAosQuadros se integra na hierarquia EventoCarreira sem quebrar relatórios antigos nem exigir if por tipo de evento

Relatório de evolução

Incoerência identificada: A análise dos cenários de evolução mostrou que a introdução de RelatorioTempoMedioNoPosto como subclasse de RelatorioSimulacao podia violar o LSP. Enquanto RelatorioQuadro e RelatorioIndividual funcionam bem com históricos parciais (apenas parte da carreira, dados de teste, etc.), a primeira ideia para RelatorioTempoMedioNoPosto assumia que todos os Militar tinham HistoricoPosto completo. Com dados incompletos, o relatório podia falhar o cálculo ou lançar exceções em gerar_texto(), reforçando pré-condições face à superclasse e quebrando a substituibilidade.

Consequência no projeto: Se isso não fosse tratado, teríamos:

1. Código cliente que usa RelatorioSimulacao a falhar inesperadamente quando recebe RelatorioTempoMedioNoPosto;
2. Testes frágeis, obrigados a fabricar históricos "perfeitos" só para esta subclasse;
3. Uma hierarquia onde uma subclasse exige mais do que a superclasse, contrariando o LSP.

Decisão tomada (refatoração): Para respeitar o LSP sem reescrever o sistema todo de dados, decidiu-se:

1. RelatorioTempoMedioNoPosto não lança exceções por causa de históricos incompletos;
2. Militares sem dados suficientes são contados como "ignorados" nas estatísticas, mas o relatório é sempre gerado;
3. O contrato de RelatorioSimulacao é mantido: registar_evento() + gerar_texto() funcionam com os mesmos eventos e estados que nas outras subclasses.

Princípios de design envolvidos

LSP (principal)

- ✓ Não há pré-condições reforçadas: o novo relatório aceita os mesmos dados que os restantes;
- ✓ Não há pós-condições enfraquecidas: continua a devolver sempre um texto de relatório válido;
- ✓ O código cliente pode tratar RelatorioTempoMedioNoPosto como um RelatorioSimulacao genérico, sem if por tipo.

OCP (secundário)

- ✓ Foi possível acrescentar um novo relatório com métricas de tempo médio sem alterar RelatorioSimulacao nem o SimuladorCarreira;
- ✓ O comportamento base é mantido, apenas enriquecido com estatísticas adicionais.

DIP (implícito)

O relatório trabalha sobre abstrações de domínio (EventoCarreira, HistoricoPosto), deixando o carregamento/limpeza de dados para outras camadas, o que facilita testes e respeita a separação de responsabilidades.

Código de dois testes

TU1 – RelatorioTempoMedioNoPosto com militares com HistoricoPosto incompleto

```
import unittest
from datetime import date
from projeto.dominio import Posto, HistoricoPosto, TipoPosto, QuadroEspecial
from projeto.relatorios import RelatorioQuadro, RelatorioTempoMedioNoPosto
from projeto.eventos import EventoPromocao

class TestRelatorioTempoMedioNoPostoTU1(unittest.TestCase):
    def test_relatorios_substituiveis_com_historico_incompleto(self):
        """TU1 - RelatorioTempoMedioNoPosto com histórico incompleto não reforça
        pré-condições face a RelatorioSimulacao."""

        # Histórico "completo" para o militar 1
        posto_tenente = Posto(TipoPosto.TENENTE, tempo_minimo=3, tempo_maximo=8,
idade_limite=60)
        posto_capitao = Posto(TipoPosto.CAPITAO, tempo_minimo=4, tempo_maximo=10,
idade_limite=62)

        historico_m1 = [
            HistoricoPosto(posto_tenente, date(2020, 1, 1), date(2023, 1, 1)),
            HistoricoPosto(posto_capitao, date(2023, 1, 2), None),
        ]

        # Histórico "incompleto/defeituoso" para o militar 2
        historico_m2 = [
            HistoricoPosto(posto_tenente, None, None),  # datas em falta de propósito
        ]

        historicos_por_nim = {
            1: historico_m1,
            2: historico_m2,
        }

        eventos = [
            EventoPromocao(militar_nim=1, posto_origem=TipoPosto.TENENTE,
posto_destino=TipoPosto.CAPITAO),
            EventoPromocao(militar_nim=2, posto_origem=TipoPosto.TENENTE,
posto_destino=TipoPosto.CAPITAO),
        ]

        # Baseline: RelatorioQuadro (tipo já existente na hierarquia)
        relatorio_quadro = RelatorioQuadro(
            quadro=QuadroEspecial.INFANTARIA,
            ano_inicial=2020,
            ano_final=2024,
        )
        for e in eventos:
            relatorio_quadro.registar_evento(e)
        texto_quadro = relatorio_quadro.gerar_texto()
        self.assertIn("total", texto_quadro.lower())

        # Novo subtipo: RelatorioTempoMedioNoPosto
        relatorio_tempo_medio = RelatorioTempoMedioNoPosto(
            quadro=QuadroEspecial.INFANTARIA,
            historicos_por_nim=historicos_por_nim,
            ano_inicial=2020,
            ano_final=2024,
        )

        # Act - usar o mesmo padrão de utilização que para qualquer RelatorioSimulacao
        for e in eventos:
            relatorio_tempo_medio.registar_evento(e)

        # Não deve lançar exceção ao gerar o texto, apesar do histórico incompleto
        texto_tempo_medio = relatorio_tempo_medio.gerar_texto()

        # Invariantes: continua a haver um relatório de texto válido
        self.assertIsInstance(texto_tempo_medio, str)
        # E a lógica interna deve conseguir distinguir pelo menos um "ignorado"
        self.assertGreaterEqual(relatorio_tempo_medio.total_militares_ignorados, 1)
```

TI1 – Mesmo fluxo com RelatorioQuadro e RelatorioTempoMedioNoPosto

```
import unittest
from datetime import date

from projeto.dominio import (Posto,HistoricoPosto,TipoPosto,QuadroEspecial,Oficial)
from projeto.relatorios import RelatorioQuadro, RelatorioTempoMedioNoPosto
from projeto.gestao import GestorQuadros, GestorCarreiras, CriterioOrdenacaoElegiveisPorDefeito
from projeto.simulacao import SimuladorCarreira


class TestSimulacaoComRelatoriosTI1(unittest.TestCase):
    def test_mesmo_fluxo_com_dois_tipos_de_relatorio(self):
        """TI1 – O mesmo fluxo de simulação (SimuladorCarreira) funciona com RelatorioQuadro e com
RelatorioTempoMedioNoPosto"""
        # ==== Dados mínimos de militares para a simulação ====
        posto_tenente = Posto(TipoPosto.TENENTE, tempo_minimo=3, tempo_maximo=8, idade_limite=60)
        posto_capitao = Posto(TipoPosto.CAPITAO, tempo_minimo=4, tempo_maximo=10, idade_limite=62)

        historico_m1 = [
            HistoricoPosto(posto_tenente, date(2018, 1, 1), date(2022, 12, 31)),
            HistoricoPosto(posto_capitao, date(2023, 1, 1), None),
        ]

        historico_m2 = [HistoricoPosto(posto_tenente, date(2020, 1, 1), None),]

        m1 = Oficial(nim=1, nome="Oficial 1",d ata_nascimento=date(1985, 5, 10),
quadro=QuadroEspecial.INFANTARIA, historico=historico_m1,
        )

        m2 = Oficial(nim=2, nome="Oficial 2", data_nascimento=date(1987, 8, 20),
quadro=QuadroEspecial.INFANTARIA, historico=historico_m2,)

        militares = {1: m1, 2: m2}
        # ==== Infraestrutura mínima de gestão / simulação ====
        gestor_quadros = GestorQuadros()
        criterio = CriterioOrdenacaoElegiveisPorDefeito()
        gestor_carreiras = GestorCarreiras(gestor_quadros=gestor_quadros,
                                           criterio_ordenacao=criterio)

        # --- 1) Simulação com RelatorioQuadro (baseline) ---
        relatorio_quadro = RelatorioQuadro(quadro=QuadroEspecial.INFANTARIA, ano_inicial=2024,
ano_final=2026,)

        simulador_quadro = SimuladorCarreira(ano_corrente=2024, militares=militares,
gestor_carreiras=gestor_carreiras, relatorio_corrente=relatorio_quadro,)

        relatorio_resultado_quadro = simulador_quadro.simular_quadro(
            quadro=QuadroEspecial.INFANTARIA,
            anos=2,
        )
        texto_quadro = relatorio_resultado_quadro.gerar_texto().lower()

        self.assertIn("promocoes", texto_quadro)
        self.assertIn("reservas", texto_quadro)

        # --- 2) Mesma simulação com RelatorioTempoMedioNoPosto (evolução) ---
        relatorio_tempo_medio = RelatorioTempoMedioNoPosto(quadro=QuadroEspecial.INFANTARIA,
historicos_por_nim={1: historico_m1, 2: historico_m2}, ano_inicial=2024, ano_final=2026,)

        simulador_tempo_medio = SimuladorCarreira(ano_corrente=2024, militares=militares,
gestor_carreiras=gestor_carreiras, relatorio_corrente=relatorio_tempo_medio,)

        relatorio_resultado_tempo_medio = simulador_tempo_medio.simular_quadro(
            quadro=QuadroEspecial.INFANTARIA,
            anos=2,
        )
        texto_tempo_medio = relatorio_resultado_tempo_medio.gerar_texto().lower()

        # Mesmo contrato base: continua a falar de promoções/reservas
        self.assertIn("promocoes", texto_tempo_medio)
        self.assertIn("reservas", texto_tempo_medio)

        # Extensão (OCP) + LSP: acrescenta info de tempo médio sem quebrar o fluxo
        self.assertIn("tempo medio", texto_tempo_medio)
```