UNIVERSIDADE ABERTA
www.uab.pt

Licenciatura em Eng. Informática
Programação por Objetos 25/26

Proposta de projeto

Simulador de Carreiras

Diagrama UML

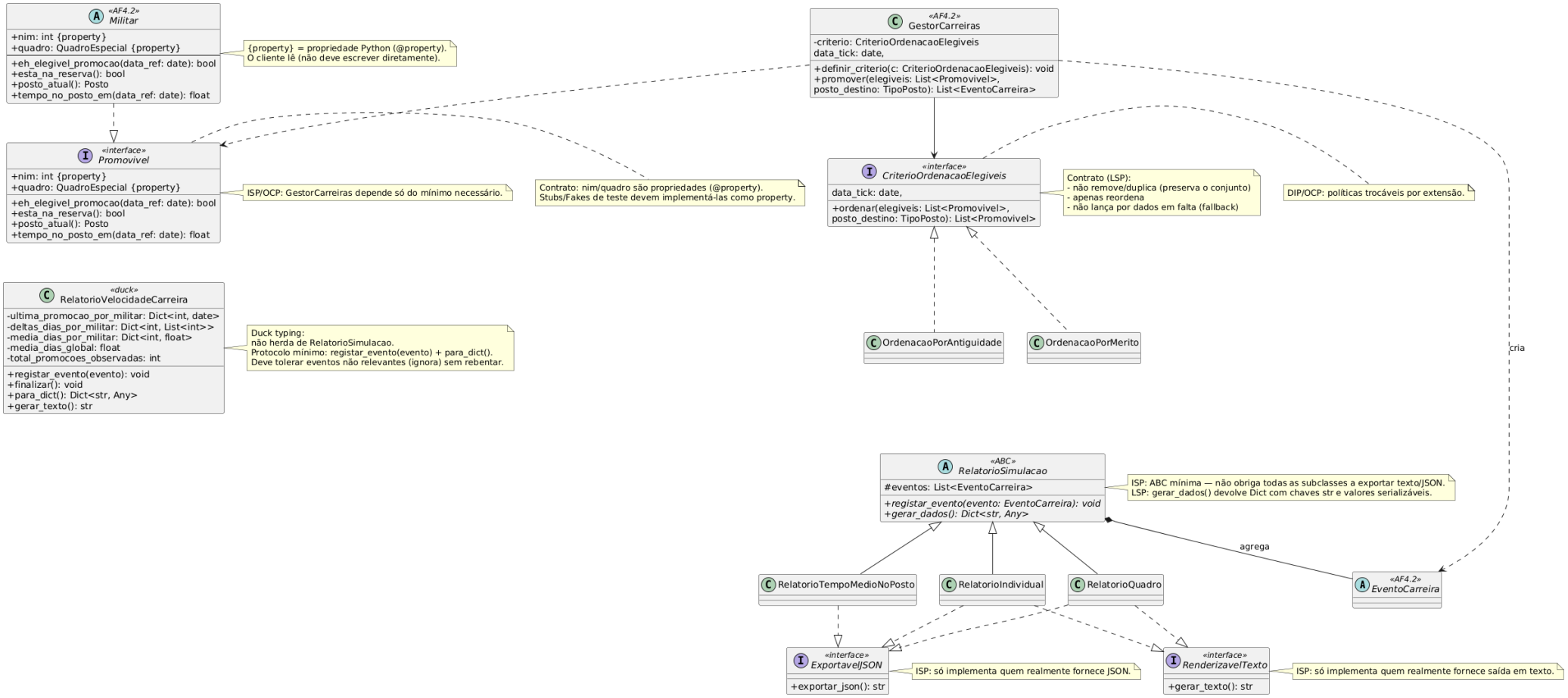

Tabela de decisões e síntese crítica

Tabela de decisões polimórficas principais

Comportamento	Tipo de Polimorfismo	Justificação	Princípio SOLID	Impacto Prático	Cenário de Falha
Seleção de elegíveis	ABC formal, através de Interface formal	O motor usa só o "mínimo" para decidir elegibilidade	ISP + DIP	Motor mais simples; testes com stubs fáceis; permite "fantasmas" sem mexer na hierarquia	Promovível incompleto (falta método / propriedade), falha na seleção; mitigação: validação + testes de integração
Ordenação de elegíveis	ABC formal, através de Strategy por interface	Evolução vs AF4.2: deixa de alterar GestorCarreiras quando muda o regulamento; acrescenta-se uma classe	DIP + LSP	Mudanças normativas isoladas; menos if; manutenção localizada na estratégia	Critério inválido (estratégia filtra/duplica ou ordena por chave errada) → promoções erradas; mitigação: testes de contrato da estratégia (não alterar conjunto, só ordem)
Exportação do relatório	ABC formal, através de Interfaces de capacidade	Não forçar todos os relatórios a exportar tudo como estava na AF4.2	ISP + LSP	Relatórios mais focados; menos métodos "a mais"; integrações só onde faz sentido	Cliente assume capacidade (chama texto/JSON num relatório que não suporta) → erro; mitigação: verificação de capability / fallback no controlador/exportador
Relatório experimental de Velocidade da Carreira	Duck typing	Protótipo rápido sem herdar / contrato formal	OCP	Iteração rápida de métricas; útil para estudos temporários	Programador esquece método/shape (evento sem tipo/data/nim), gera erro em runtime; mitigação: testes específicos + validação no registar_evento

Cenários de rutura (O código destes cenários foi omitido face ao apresentado em AF5.1):

Cenário 1: Violação LSP por exceções/semântica "surpresa" na ordenação de elegíveis

Problema: Evolução de ordenação por antiguidade para mérito, criação de uma estratégia que "faz mais do que ordenar" ou lança exceções quando faltam avaliações. Rutura: GestorCarreiras deixa de poder trocar estratégias de forma segura porque a nova estratégia introduz falhas e comportamentos inesperados (quebra LSP). Prevenção com contrato formal (ABC/interface) e regra de comportamento.

Cenário 2: Violação ISP por "RelatorioSimulacao" monolítico

Problema: ABC de relatório obriga todas as subclasses a exportar texto e JSON, mesmo quando um relatório é só estatístico/estrutural. Rutura: As subclasses ficam poluídas com implementações artificiais, aumentando ruído, risco e manutenção (violação ISP). Solução por segregação (ABC mínima + capacidades opcionais):

Cenário 3: Risco de Duck Typing em relatórios experimentais

Problema: programador cria um relatório "rápido" com registar_evento, mas esquece invariantes mínimas ou assume eventos com estrutura diferente. Rutura: Evento não previsto (ou sem campos esperados), o relatório falha em runtime e quebra a simulação/relato. Mitigação por testes de integração (contrato informal "proativo"):

Análise:

O design polimórfico atual previne rutura nos pontos críticos e mitiga riscos nas zonas experimentais. Na ordenação, a introdução de CriterioOrdenacaoElegiveis impõe um contrato claro ("só reordena; não filtra/duplica nem falha por dados em falta"), evitando que a passagem de antiguidade para mérito volte a introduzir semântica surpresa ou exceções que quebrem o GestorCarreiras (LSP), o que é confirmado pelo teste que garante que o conjunto de elegíveis se mantém. Nos relatórios, a redução de RelatorioSimulacao a uma ABC mínima e a separação de exportações por capacidades (RenderizavelTexto, ExportavelJSON) elimina a pressão de uma interface monolítica, evitando implementações "fake" e respeitando ISP/LSP. Por fim, o duck typing (ex.: RelatorioVelocidadeCarreira) é assumido como flexível, mas o risco de falhas em runtime pode ser mitigado por testes de integração que verificam o protocolo mínimo e a tolerância a eventos do core.

Código Python Implementado – ABC e Implementações

```python
from __future__ import annotations
from abc import ABC, abstractmethod
from datetime import date
from typing import Any, Dict, List

# ===== ABC DEFINIDA NA AF5.1 =====
# CriterioOrdenacaoElegiveis, para mudança de ordenação por antiguidade -> mérito, sem modificar o motor).
class CriterioOrdenacaoElegiveis(ABC):
    """ABC para ordenação de elegíveis. Respeita LSP. Contrato: Deve sempre retornar list, não pode filtrar/duplicar elegíveis e não deve lançar exceções por dados em falta (degradação graciosa)."""

    @abstractmethod
    def ordenar(self, elegiveis: List[Dict[str, Any]], data_tick: date, posto_destino: Any) -> List[Dict[str, Any]]:
        """Reordena elegíveis (mesmos elementos, ordem diferente). Nunca retorna None."""
        pass

# ===== IMPLEMENTAÇÕES CONCRETAS DA ABC =====
class OrdenacaoPorAntiguidade(CriterioOrdenacaoElegiveis):
    """Implementação formal AF5.1: ordenação por antiguidade (anos_no_posto)."""

    def ordenar(self, elegiveis: List[Dict[str, Any]], data_tick: date, posto_destino: Any) -> List[Dict[str, Any]]:
        return sorted(elegiveis, key=lambda m: float(m.get("anos_no_posto", 0.0)), reverse=True)

class OrdenacaoPorMerito(CriterioOrdenacaoElegiveis):
    """Implementação formal AF5.1: ordenação por mérito (fallback None -> 0.0). Evita o cenário de rutura: estratégia não pode rebentar se faltar avaliação."""

    def ordenar(self, elegiveis: List[Dict[str, Any]], data_tick: date, posto_destino: Any) -> List[Dict[str, Any]]:
        return sorted(elegiveis, key=lambda m: float(m.get("merito") or 0.0), reverse=True)

# ===== CLASSE DUCK TYPING INDEPENDENTE =====
# Protocolo informal definido na AF5.1 - Implementação por duck typing (relatório experimental):
# Shape mínimo do evento (dict): {"tipo": "PROMOCAO", "militar_nim": int, "data": date}
class RelatorioVelocidadeCarreira:
    def __init__(self) -> None:
        self._ultima: Dict[int, date] = {}
        self._deltas: Dict[int, List[int]] = {}

    def registar_evento(self, evento) -> None:
        if getattr(evento, "get", None) and evento.get("tipo") != "PROMOCAO":
            return
        nim = evento.get("militar_nim") if getattr(evento, "get", None) else None
        d = evento.get("data") if getattr(evento, "get", None) else None
        if nim is None or d is None:
            return  # degradação graciosa (não rebenta)
        if nim in self._ultima:
            self._deltas.setdefault(nim, []).append((d - self._ultima[nim]).days)
        self._ultima[nim] = d

    def media_global_dias(self) -> float:
        vals = [x for lst in self._deltas.values() for x in lst]
        return sum(vals) / len(vals) if vals else 0.0

# ===== FUNÇÃO GENÉRICA QUE USA POLIMORFISMO =====
def processar_promocoes(elegiveis: List[Dict[str, Any]], criterio, relatorio,
                        data_tick: date, posto_destino: Any) -> Dict[str, Any]:
    """Função polimórfica que aceita qualquer objeto com métodos esperados. Demonstra:
    - Polimorfismo formal (Strategy): criterio com ordenar(...)
    - Duck typing defensivo: relatorio com registar_evento(...) (hasattr/callable)"""

    ordenados = criterio.ordenar(elegiveis, data_tick, posto_destino) \
        if hasattr(criterio, "ordenar") and callable(criterio.ordenar) else list(elegiveis)

    if hasattr(relatorio, "registar_evento") and callable(relatorio.registar_evento):
        for m in ordenados:
            relatorio.registar_evento({"tipo": "PROMOCAO", "militar_nim": m.get("nim"), "data": data_tick})

    return {
        "ordenados_nim": [m.get("nim") for m in ordenados],
        "media_global_dias": relatorio.media_global_dias() if hasattr(relatorio, "media_global_dias") else None,
    }
```

Testes polimórficos

```python
import unittest

def elegiveis_base():
    return [
        {"nim": 101, "anos_no_posto": 3.0, "merito": 10.0},
        {"nim": 202, "anos_no_posto": 1.0, "merito": None},
        {"nim": 303, "anos_no_posto": 2.0, "merito": 20.0},
    ]

# ===== TESTES UNITÁRIOS =====
class TestCriteriosOrdenacao_Unitarios(unittest.TestCase):
    """Testes de unidade para estratégias formais (ABC) — validam métodos individuais."""

    def test_ordenar_quando_antiguidade_entao_ordem_desc_por_anos_no_posto(self):
        """Verifica que OrdenacaoPorAntiguidade.ordenar() ordena por tempo no posto (desc)."""
        crit = OrdenacaoPorAntiguidade()
        out = crit.ordenar(elegiveis_base(), date(2025, 1, 1), "SARG-AJUD")
        self.assertEqual([m["nim"] for m in out], [101, 303, 202],
                         "Antiguidade deve ordenar por anos_no_posto (desc) de forma determinística.")

    def test_ordenar_quando_merito_none_entao_fallback_zero_sem_excecao(self):
        """Verifica que OrdenacaoPorMerito.ordenar() degrada graciosamente quando merito=None."""
        crit = OrdenacaoPorMerito()
        out = crit.ordenar(elegiveis_base(), date(2025, 1, 1), "SARG-AJUD")
        self.assertEqual([m["nim"] for m in out], [303, 101, 202],
                         "Merito=None deve degradar para 0.0 e manter ordenação válida (sem exceções).")

class TestRelatorioDuck_Unitarios(unittest.TestCase):
    """Testes de unidade para duck typing — verificação de protocolo informal AF5.1."""

    def test_registar_evento_quando_nao_promocao_entao_ignora(self):
        """Verifica que RelatorioVelocidadeCarreira ignora eventos que não sejam PROMOCAO."""
        r = RelatorioVelocidadeCarreira()
        r.registar_evento({"tipo": "RESERVA", "militar_nim": 101, "data": date(2025, 1, 1)})
        self.assertEqual(r.media_global_dias(), 0.0,
                         "Relatório duck deve ignorar eventos não-PROMOCAO sem alterar métricas.")

    def test_registar_evento_quando_duas_promocoes_entao_delta_e_media_calculados(self):
        """Verifica que duas promoções do mesmo militar geram delta (dias) e média global."""
        r = RelatorioVelocidadeCarreira()
        r.registar_evento({"tipo": "PROMOCAO", "militar_nim": 101, "data": date(2025, 1, 1)})
        r.registar_evento({"tipo": "PROMOCAO", "militar_nim": 101, "data": date(2025, 1, 11)})
        self.assertEqual(r.media_global_dias(), 10.0,
                         "Duas promoções do mesmo militar devem produzir delta e média global consistente.")

# ==== TESTES DE INTEGRAÇÃO ====
class TestIntegracaoPolimorfismo(unittest.TestCase):
    """Testes de integração — validam colaboração Strategy (ABC) + função genérica + relatório (duck)."""

    def test_processar_promocoes_com_criterio_formal(self):
        """Testa função polimórfica com implementações formais completas (ABC)."""
        crit = OrdenacaoPorAntiguidade()
        rel = RelatorioVelocidadeCarreira()
        resultado = processar_promocoes(elegiveis_base(), crit, rel, date(2025, 1, 1), "SARG-AJUD")
        self.assertIn("ordenados_nim", resultado, "Função deve devolver ordem final observável.")
        self.assertEqual(resultado["ordenados_nim"], [101, 303, 202],
                         "Fluxo completo deve refletir a estratégia selecionada.")

    def test_substituibilidade_lsp_quando_troca_criterios_entao_mesmo_contrato(self):
        """Valida que diferentes estratégias são intercambiáveis sem quebrar o cliente."""
        # Valida prevenção do cenário de rutura 1 identificado na AF5.1
        # (Estratégia não pode filtrar/duplicar nem rebentar por dados em falta)
        elegiveis = elegiveis_base()
        criterios = [OrdenacaoPorAntiguidade(), OrdenacaoPorMerito()]
        for c in criterios:
            with self.subTest(criterio=type(c).__name__):
                out = c.ordenar(elegiveis, date(2025, 1, 1), "SARG-AJUD")
                self.assertEqual(len(out), len(elegiveis),
                                 "LSP: estratégia não pode remover nem criar elementos (tamanho preservado).")
                self.assertEqual({m["nim"] for m in out}, {m["nim"] for m in elegiveis},
                                 "LSP: estratégia deve manter o mesmo conjunto (só reordena).")

    def test_cenario_rutura_duck_evento_incompleto_nao_quebra(self):
        """Testa cenário de rutura AF5.1: protocolo informal pode falhar em runtime.
        Aqui validamos a mitigação: relatório duck degrada graciosamente com evento incompleto."""
        # Valida prevenção do cenário de rutura 3 identificado na AF5.1
        rel = RelatorioVelocidadeCarreira()
        try:
            rel.registar_evento({"tipo": "PROMOCAO", "militar_nim": 101})  # falta "data"
        except Exception as exc:
            self.fail(f"Relatório duck não pode rebentar; deve degradar graciosamente. Erro: {exc}")

        self.assertEqual(rel.media_global_dias(), 0.0,
                         "Sem data, o evento deve ser ignorado e as métricas manter-se consistentes.")
```

Relatório de evolução

Incoerência observada

A implementação prática do SCM evidenciou a tensão central entre a flexibilidade do **duck typing** e a segurança dos contratos formais por **ABC**, exatamente como antecipado na AF5.1. O relatório experimental RelatorioVelocidadeCarreira (duck typing) confirmou o trade-off planeado:

- **Vantagem observada:** integração imediata no pipeline de simulação sem herdar de RelatorioSimulacao nem implementar "capabilities" formais; basta existir registar_evento(evento).
- **Risco confirmado:** a ausência/alteração do *shape* do evento (ex.: falta de data) só é detetada em runtime, podendo gerar falhas ou métricas inválidas.
- **Contradição:** o sistema quer aceitar relatórios rápidos e evolutivos, mas simultaneamente exige estabilidade para não comprometer resultados da simulação.

Esta tensão tornou-se evidente no teste de integração **test_cenario_rutura_duck_evento_incompleto_nao_quebra()**, onde o relatório duck recebe um evento incompleto e, embora não colapse, tem de ignorar o input para manter coerência.

Consequência prática

A incoerência implicou três consequências diretas no código e nos testes:

- **Verificação defensiva** no protocolo informal: o registar_evento usa getattr/get para tolerar eventos incompletos;
- **Degradação graciosa**: em vez de exceções, o comportamento é ignorar/retornar valores neutros (mantendo métricas consistentes);
- **Dupla camada de validação**: testes unitários para as estratégias formais (ABC) e testes de integração para o duck typing.

Na prática, isto aumenta a complexidade do "código de cola" (validações e defaults), mas permite manter relatórios experimentais sem bloquear evolução.

Limite aceite

Aceitei conscientemente que relatórios experimentais podem ser incompletos, mas defini condições claras:

- **Testes de integração obrigatórios** para qualquer relatório duck: o teste test_cenario_rutura_duck_evento_incompleto_nao_quebra() valida que o relatório não compromete o fluxo da simulação.
- **Degradação documentada**: se faltam campos, o evento é ignorado (não há métricas falsas nem crash).
- **Segregação do risco**: duck typing fica confinado a funcionalidades não-críticas (relatórios adicionais/experimentais), não ao motor de promoções.

Em contrapartida, para comportamentos críticos (ordenação), a rigidez foi não negociável: a ABC CriterioOrdenacaoElegiveis garante que trocar estratégias não altera o conjunto de elegíveis nem falha por dados em falta.

Princípio de design envolvido

Esta decisão equilibra explicitamente:

- **OCP:** novas políticas de ordenação entram por extensão (OrdenacaoPorMerito como nova classe), sem alterar o motor; relatórios duck permitem extensões rápidas sem mexer no core.
- **LSP:** as estratégias formais são substituíveis sem "surpresas"; o teste **test_substituibilidade_lsp_quando_troca_criterios_entao_mesmo_contrato()** confirma que nenhuma estratégia filtra/duplica elegíveis.
- **ISP:** o relatório duck implementa apenas registar_evento, não sendo forçado a exportar/formatar/gerar texto se isso não for necessário.

O custo aceite é a perda de verificação formal no duck typing, compensada por mitigação via testes e validação defensiva. O resultado prático é um núcleo estável (ABC + testes LSP) e uma periferia extensível (duck typing), com risco controlado.

Validação das decisões da AF5.1

Na AF5.1 defini quatro pontos de polimorfismo no SCM para permitir evolução sem "engordar" o GestorCarreiras com if/elif por regulamento, categoria ou exceções: (1) seleção de elegíveis via Promovivel, (2) ordenação via Strategy (CriterioOrdenacaoElegiveis), (3) relatórios formais com ABC mínima (RelatorioSimulacao) e capabilities segregadas (ExportavelJSON/RenderizavelTexto) e (4) relatórios experimentais por duck typing (RelatorioVelocidadeCarreira). Na AF5.2, a implementação e a bateria de testes permitiram ligar estas decisões a resultados concretos: menor acoplamento entre camadas, maior facilidade de testar o core isoladamente e maior previsibilidade nos pontos críticos, sem impedir experimentação.
A Strategy formal para ordenação foi a decisão mais sólida. Na prática, tornou a mudança "antiguidade → mérito" uma extensão por nova classe, sem alterar o motor (OCP/DIP), exatamente o cenário de evolução problemático identificado na AF4.2/AF5.1. Mais importante, os testes validaram LSP ao garantir invariantes comportamentais: qualquer critério só reordena, não remove/duplica elegíveis e não falha por dados em falta. Ou seja, o contrato deixou de ser apenas "assinatura" e passou a ser "comportamento observável". O fallback implementado é evidência direta de prevenção da rutura (avaliações em falta causarem exceções e quebra do fluxo):

```
return sorted(elegiveis, key=lambda m: float(m.get("merito") or 0.0), reverse=True)
```

Este ponto mostrou-se adequado porque o impacto de erro é alto (promoções erradas ou bloqueio da simulação) e o custo de formalização é baixo.
A introdução de Promovivel, apesar de não se encontrar exposta aqui a sua implementação, também se revelou adequada, pois GestorCarreiras passou a depender apenas do mínimo necessário (ISP/DIP), reduzindo dependência da hierarquia Militar e do seu estado completo (histórico, reserva, datas, etc.). Isto traduziu-se em maior testabilidade, pudendo-se criar stubs/fantasmas para simular elegíveis e validar o comportamento do motor sem depender de carregamento de dados ou regras específicas por categoria. Existe necessidade de documentar exatamente o que é obrigatório (ex.: nim, quadro, posto_atual, eh_elegivel_promocao) para evitar stubs "quase compatíveis" que geram falhas artificiais e confundem diagnóstico, como por exemplo:

```
class Promovivel(Protocol):
    @property
    def nim(self) -> int: ...
    def quadro(self) -> Any: ...
    def eh_elegivel_promocao(self, data_ref: date) -> bool: ...
    def posto_atual(self) -> Any: ...
```

A decisão que exigiu mais cuidado foi a aplicação de ISP nos relatórios formais. Ao separar capabilities, evitou-se o problema clássico da AF4.2 (classes obrigadas a implementar métodos irrelevantes, produzindo "fake methods" como return "" só para cumprir contrato). Em AF5.2, isso tornou as classes mais limpas e a manutenção mais óbvia ("cada relatório faz o que promete"). No entanto, apareceu um ajuste inevitável: o cliente (controlador/exportador) não pode assumir que todo o relatório exporta JSON ou gera texto. O consumo teve de ser defensivo, com fallback para gerar_dados() quando a capability não existe, sob pena de criar um novo cenário de rutura ("cliente chama método inexistente"):

```
if isinstance(rel, ExportavelJSON): return rel.exportar_json()
return json.dumps(rel.gerar_dados())
```

Este ajuste mostrou que ISP melhora o design, mas transfere responsabilidade para quem consome as interfaces.
Por fim, o duck typing confirmou o trade-off planeado: é excelente para relatórios experimentais (flexibilidade e extensão rápida sem tocar no core), mas expõe fragilidade ao "shape" do evento em runtime. A mitigação prática foi degradação graciosa + testes de integração (o protocolo informal passa a ser "contrato executável" via testes, como demonstrado em test_cenario_rutura_duck_evento_incompleto_nao_quebra).

Em síntese, a AF5.2 validou que ABC/interfaces devem proteger o núcleo (seleção/ordenação) pela exigência de previsibilidade e substituibilidade, enquanto o duck typing é útil fora do core para inovação/experiência, desde que o risco seja controlado por validação defensiva e testes de integração.

Reflexão sobre princípios SOLID

A AF5.2 permitiu observar que os princípios SOLID não são "regras absolutas", mas ferramentas a aplicar onde o custo do erro é maior. No SCM, isto ficou claro ao distinguir o núcleo (promoções/seleção/ordenação) das extensões (relatórios e protótipos).

OCP — Aberto/Fechado (extensibilidade sem modificar o core)

No contexto do SCM, o OCP foi aplicado sobretudo na ordenação de elegíveis. A mudança de "antiguidade" para "mérito" (cenário evolutivo identificado na AF4.2/AF5.1) deixou de exigir alterações no GestorCarreiras, fazendo com que a evolução passasse a ser feita por extensão, criando uma nova estratégia.
Tensão prática: aplicar OCP em excesso pode gerar "classe por tudo". O limite aceite foi: abstrair apenas onde a variabilidade é real e recorrente (políticas de ordenação e formatos de exportação), evitando criar interfaces desnecessárias para comportamentos estáveis.

LSP — Substituição de Liskov (trocar implementações sem "surpresas")

No SCM, LSP foi o princípio mais sensível porque falhas aqui não são "apenas erros", podem gerar promoções erradas. A AF5.2 mostrou que LSP não se garante só com assinatura: exige invariantes comportamentais verificáveis ("só reordena; não filtra/duplica; não rebenta com dados em falta").
A lógica de fallback foi uma medida direta para preservar substituibilidade:

```
key=lambda m: float(m.get("merito") or 0.0)    # evita exceção quando falta avaliação
```

Tensão prática: tornar LSP "real" aumentou o número de testes e obrigou a documentar melhor o contrato (o que a estratégia pode e não pode fazer). Em troca, o motor ficou mais previsível e menos frágil a mudanças de regras.

ISP — Segregação de Interfaces (depender só do mínimo necessário)

O ISP foi aplicado em dois pontos:

1. Promovivel: o GestorCarreiras depende do mínimo para decidir elegibilidade/ordenar, reduzindo acoplamento à hierarquia Militar (e facilitando stubs).
2. Relatórios por capabilities: separar ExportavelJSON e RenderizavelTexto evitou obrigar todos os relatórios a implementar tudo (evitou "métodos fake").

Exemplo de consumo correto (o cliente não assume capabilities):

```
if isinstance(rel, ExportavelJSON):
    return rel.exportar_json()
return json.dumps(rel.gerar_dados())
```

Tensão prática: ISP melhora clareza, mas desloca responsabilidade para o cliente (tem de verificar capabilities). O risco é cair em excesso de interfaces pequenas e fragmentação. O limite aceite foi introduzir capabilities apenas quando havia subclasses que claramente não precisavam do método.

Síntese das tensões (princípios vs praticidade)

A AF5.2 confirmou um padrão: OCP e ISP aumentam flexibilidade e reduzem acoplamento, mas podem aumentar complexidade conceptual (mais classes/contratos). LSP aumenta segurança, mas exige esforço adicional (invariantes + testes). A decisão prática foi aplicar rigor (ABC + testes LSP) no core e aceitar flexibilidade controlada (duck typing + mitigação) nas extensões, equilibrando qualidade arquitetural com pragmatismo de implementação.

Conclusão consolidada

O balanço entre conceção (AF5.1) e implementação (AF5.2) foi positivo: as decisões de desenho deixaram de ser apenas "boas intenções" e passaram a produzir efeitos observáveis no código e nos testes. A AF5.2 confirmou que identificar, na AF5.1, pontos reais de variabilidade (seleção/ordenação de elegíveis, exportação/representação de relatórios e relatórios experimentais) foi essencial para evitar crescimento do núcleo por condicionais e para reduzir acoplamento entre camadas.

A arquitetura polimórfica planeada ficou validada em dois sentidos. Primeiro, no núcleo (promoções/ordenação), a formalização por contratos (interfaces/ABC) e invariantes testáveis reforçou a substituibilidade (LSP) e permitiu evolução por extensão (OCP), mantendo o motor dependente apenas do mínimo necessário (ISP/DIP). Segundo, na periferia, o duck typing mostrou-se viável como "válvula de escape" para protótipos e relatórios experimentais, desde que acompanhado por degradação graciosa e testes de integração que funcionam como contrato executável.

As principais lições aprendidas foram: em Python, a segurança do polimorfismo não vem só de "tipos", vem de contratos comportamentais e testes; aplicar ISP melhora clareza, mas obriga o cliente a consumir capabilities com disciplina (não assumir métodos); duck typing é poderoso, mas deve ser confinado a zonas não-críticas e com mitigação explícita. Com isto, o sistema fica preparado para evoluções futuras (novos critérios regulamentares, novos formatos de exportação, novos relatórios), com menor risco de regressões e com um caminho claro: estender por novas classes onde o impacto é alto; experimentar por protocolos informais onde o impacto é baixo.