UNIVERSIDADE ABERTA
www.uab.pt

Licenciatura em Eng. Informática
Programação por Objetos 25/26

Proposta de projeto

Simulador de Carreiras

Diagrama UML

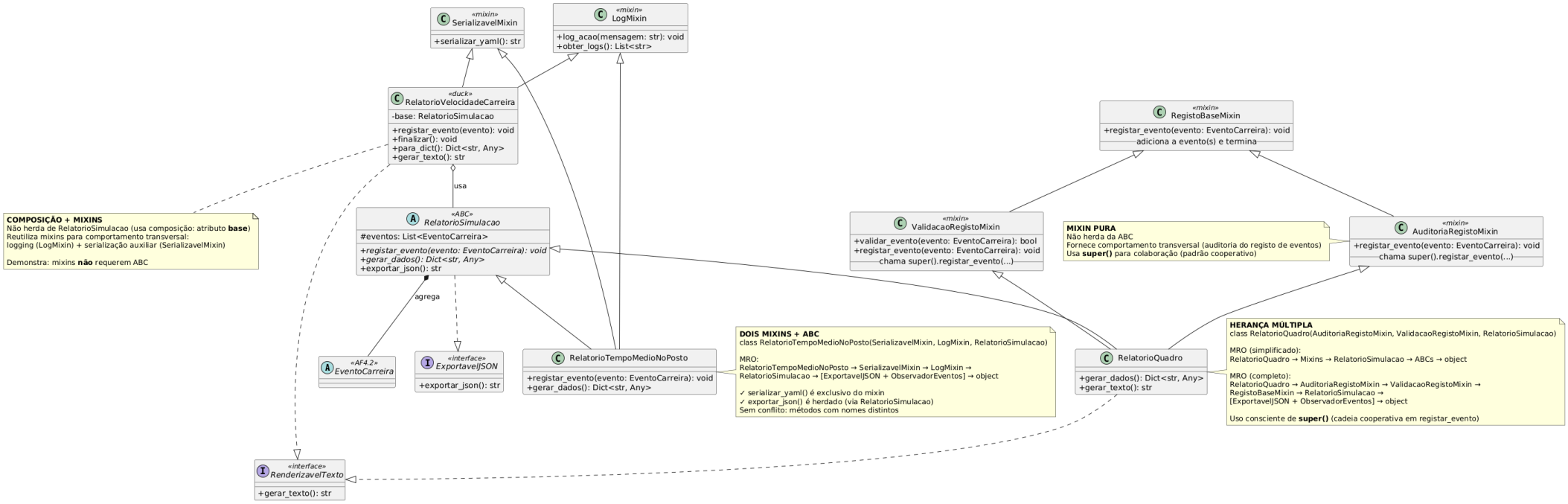

Tabela de decisões de reutilização

Funcionalidade	Cenário evolutivo	Tipo	Relação com AF5.2	Princípio SOLID em tensão	Trade-off aceite	MRO prevista + análise
Rastreabilidade da simulação (debug de promoções/relatórios)	Motor cresce (regras, exceções, casos-limite) e é preciso registar passos / decisões	LogMixin	Independente da ABC; complementa RelatorioSimulacao sem alterar o contrato	SRP vs DRY	Aceita-se acrescentar logging transversal para evitar duplicação e ganhar rastreabilidade	RelatorioTempoMedioNoPosto → SerializavelMixin → LogMixin → RelatorioSimulacao → object Análise: LogMixin cedo para registar antes de serializar/gerar dados
Exportação alternativa (YAML além de JSON)	Surge necessidade de partilhar resultados com ferramentas externas / integrações além do JSON	SerializavelMixin	Alternativa / complemento a ExportavelJSON (capacidade opcional)	OCP vs ISP	Aceita-se coexistência de formatos, mantendo cada formato como capacidade separada	RelatorioTempoMedioNoPosto → SerializavelMixin → LogMixin → RelatorioSimulacao → object Cuidado: YAML deve derivar de gerar_dados() para evitar "duas verdades"
Registo robusto de eventos (validação + auditoria)	Eventos de carreira passam a exigir validação forte e auditoria sem duplicar código em cada relatório	Herança múltipla com mixins (AuditoriaRegistoMixin + ValidacaoRegistoMixin)	Usa a ABC como contrato; mixins adicionam passos ao registar_evento()	LSP vs Complexidade	Aceita-se complexidade de MRO para pipeline extensível; cada mixin chama super()	RelatorioQuadro → AuditoriaRegistoMixin → ValidacaoRegistoMixin → RegistoBaseMixin → RelatorioSimulacao → object Cuidado: se um mixin não chamar super(), corta a cadeia
Relatório experimental fora do contrato formal	Necessidade de um relatório novo (RelatorioVelocidadeCarreira) rapidamente, sem engordar a ABC e sem obrigar todos a suportar o mesmo contrato	Composição + mixins (LogMixin + SerializavelMixin)	Não herda de RelatorioSimulacao; **usa** base: RelatorioSimulacao quando necessário	LSP vs Flexibilidade	Aceita-se não ser substituível por relatório formal; ganha-se liberdade de evolução	RelatorioVelocidadeCarreira → LogMixin → SerializavelMixin → object Cuidado: não é substituível por RelatorioSimulacao (exige protocolo mínimo)

Análise detalhada da MRO + Cenários problemáticos

Cenário Complexo: RelatorioQuadro com padrão diamante (mixins cooperativos)

```
class RelatorioSimulacao:
    def gerar_dados(self): pass

class ValidacaoRegistoMixin:
    def registar_evento(self, e): return super().registar_evento(e)

class AuditoriaRegistoMixin:
    def registar_evento(self, e): return super().registar_evento(e)

class RegistoBaseMixin:
    def registar_evento(self, e): self.eventos.append(e)  # termina (agrega)

class RelatorioQuadro(ValidacaoRegistoMixin, AuditoriaRegistoMixin, RegistoBaseMixin, RelatorioSimulacao):
    def __init__(self): self.eventos=[]
    def gerar_dados(self): return {"n": len(self.eventos)}
```

Análise da MRO com mro():
MRO calculada: RelatorioQuadro → ValidacaoRegistoMixin → AuditoriaRegistoMixin → RegistoBaseMixin (base comum do diamante (aparece uma única vez)) → RelatorioSimulacao → object
Problemas identificados:

- Diamante (base comum): ambos os mixins chamam super(), convergindo em RegistoBaseMixin.
- Ordem crítica: trocar ValidacaoRegistoMixin e AuditoriaRegistoMixin muda a sequência.
- Corte de cadeia: se um mixin não fizer super(), o evento pode nunca ser agregado.

Solução planeada:

- Regra: qualquer mixin que override registar_evento chama sempre super().registar_evento(e).
- Diamante controlado: RegistoBaseMixin aparece uma vez na MRO (C3), garantindo agregação única.
- Teste: validar RelatorioQuadro.__mro__ e que eventos e aud mudam como esperado.

Cenários evolutivos problemáticos identificados:

1. Explosão de mixins no "pipeline" de registar_evento() (relatórios)
Cenário: SCM evolui e queremos acrescentar aos relatórios: AuditoriaRegistoMixin (quem/porquê do evento), MetricasMixin (KPIs), CacheMixin (evitar recomputar gerar_dados()), além de LogMixin e ValidacaoRegistoMixin.
Problema: responsabilidades sobrepostas (observabilidade/telemetria) → a MRO fica difícil de prever e o registar_evento() vira uma "pipeline" frágil:

```
class RelatorioQuadro(AuditoriaRegistoMixin, MetricasMixin, CacheMixin, LogMixin, ValidacaoRegistoMixin,
RelatorioSimulacao): ...
```

Sinal de Alerta: >3 mixins no mesmo domínio conceptual (logs/auditoria/métricas/cache) a intercetar o mesmo método.
Solução preferível: Composição (ex.: lista de handlers de evento) / padrão Observer / plugins ("RelatorioSimulacao notifica observadores").

2. Diamante "incontrolável" ao misturar 2+ hierarquias abstratas
Cenário: Surge a necessidade de um relatório "completo" que seja simultaneamente: um RelatorioSimulacao (ABC do domínio), "persistível" (ex.: escrever para BD/ficheiro) via uma nova ABC Persistivel, e ainda com LogMixin + SerializavelMixin.
Problema: Duas hierarquias convergentes + métodos com o mesmo nome (ex.: exportar() / guardar() / gerar_dados()), criando diamantes e MRO ambígua: RelatorioSimulacao + Persistivel → RelatorioQuadroCompleto
Sinal de Alerta: 2+ ABCs independentes a convergir numa classe (cada uma com "métodos centrais").
Solução preferível: Composição: o relatório tem um repositorio/persistencia (serviço), não é persistência (DIP: injeta interface).

3. Cadeias longas de super() (incluindo __init__)
Cenário: Mixins passam a precisar de configuração/estado: ConfiguravelMixin(__init__(config, ...)), MetricasMixin(__init__(stats, ...)), LogMixin(__init__(logger, ...)) em GestorCarreiras ou em RelatorioQuadro.
Problema: Um mixin que não chama super().__init__() (ou não propaga **kwargs) quebra toda a cadeia; bugs aparecem "longe" da causa.
Sinal de Alerta: Erros de inicialização difíceis de depurar + muitos mixins com estado próprio.
Solução preferível: Injeção por composição (atributos logger, metrics, cache) ou Builder/fábrica para montar o objeto, mantendo mixins preferencialmente stateless.

Síntese reflexiva consolidada:

Análise de Coexistência ABC + Mixins

No SCM, as ABC da AF5.2 (ex.: RelatorioSimulacao e capacidades opcionais como ExportavelJSON/RenderizavelTexto) definem contratos de domínio: o mínimo necessário para o motor e para os consumidores dos resultados. Os mixins (LogMixin, SerializavelMixin, ValidacaoRegistoMixin, AuditoriaRegistoMixin) acrescentam comportamentos transversais sem alterar esse contrato. A sinergia principal é manter a ABC pequena (ISP) e estender por composição de capacidades (OCP). O ponto de maior risco é a coexistência de mixins que interceptam o mesmo método (registar_evento): cria-se um "pipeline" onde a ordem na MRO e o uso de super() passam a ser parte do design (diamante controlado).

Tensões entre Princípios SOLID

A tensão mais evidente é SRP vs DRY: ao usar LogMixin, um relatório ganha responsabilidade extra (observabilidade), mas evita duplicação e facilita depuração e testes. Outra tensão é ISP vs herança múltipla: a classe final "engorda", mas o objetivo é que isso resulte de capacidades realmente necessárias, mantendo o contrato base simples e evitando obrigar todos os relatórios a suportar exportações/formatos que não usam. Em termos de KISS, a herança múltipla é aceite apenas enquanto a MRO for explicável e previsível; quando a complexidade ultrapassa o benefício, a composição torna-se preferível.

Decisões de design e limites

A escolha foi: ABC para o núcleo do domínio ("o que o relatório é e garante"), mixins para funcionalidades ortogonais (logging, serialização alternativa, passos adicionais de registo) e composição quando a substituibilidade não é desejada ou quando a hierarquia fica confusa (ex.: RelatorioVelocidadeCarreira usa base: RelatorioSimulacao, não herda). Critérios para candidatas a mixin: comportamento transversal, pequeno, reutilizável, pouco acoplado ao estado interno. Sinais de alerta para migrar para composição: >3 mixins a intercetar o mesmo método, necessidade de estado pesado/__init__ em vários mixins, e dificuldade em justificar a MRO num diagrama simples.

Preparação para implementação

A estratégia de implementação será cooperative multiple inheritance: qualquer mixin que override registar_evento() chama sempre super().registar_evento(...), e existe um "ponto de fecho" (base de registo) que garante a agregação final. A validação da MRO na AF6.2 será feita com inspeção de Classe.__mro__ e testes que confirmem a sequência (evento agregado + validação/auditoria/logs executados). A expectativa é um custo moderado de complexidade (ordem/MRO) em troca de maior reutilização e extensibilidade, com compromisso explícito de refatorar para composição se os sinais de alerta aparecerem.