Refatoração do Arcabouço de Modelos Probabilísticos ToPS

Renato Cordeiro Ferreira

Monografia do Trabalho de Conclusão de Curso
apresentado à disciplina
MAC0499
(Trabalho de Formatura Supervisionado)

Supervisor: Prof. Dr. Alan Mitchell Durham

São Paulo, Novembro de 2015

Conteúdo

Capítulo 1

Introdução

O código de um programa sofre modificações ao longo do tempo. Ao fazer mudanças para cumprir objetivos de curto-prazo, ou criar alterações sem um entendimento completo do design do programa, o código começa a perder sua estrutura - tornando-se mais difícil e custoso de se manter (Fowler **et al.** , 1999).

Para combater a perda de qualidade no código, podemos utilizar a técnica de **refatoração**, que apresenta duas definições complementares, segundo o contexto (Fowler **et al.** , 1999):

- **Refatoração** (substantivo):
  Uma mudança feita na estrutura interna de um programa para fazê-lo mais fácil de entender ou mais barato de modificar, sem alterar o seu comportamento observável.

- **Refatorar** (verbo):
  Reestruturar um programa por meio da aplicação de uma série de refatorações, sem alterar o seu comportamento observável.

Dessa maneira, fazer a refatoração de um programa significa refatorar o seu código - o que, de forma geral, requer aplicar várias refatorações sobre ele (Fowler **et al.** , 1999).

Segundo a definição apresentada, refatorar implica necessariamente em não modificar o comportamento do programa. Entretanto, realizar estas alterações demanda tempo, que poderia ser utilizado para a implementação de novas funcionalidades. Apesar desse aparente desperdício, existe uma série de vantagens que minimizam o custo da refatoração a longo prazo:

- **Design**
  Conforme descrito no início da seção, modificações contínuas no código tendem a prejudicar a estrutura do programa do qual ele faz parte. Em geral, isso é causado pela presença de **duplicações** - trechos de código que realizam a mesma tarefa, em partes diferentes do programa. Para fazer uma alteração, torna-se necessário modificar muitos lugares, o que aumenta a chance de introduzir erros. Sem uma forma de documentação, também é possível que alguns trechos fiquem sem ser modificados, causando inconsistências. Utilizando as técnicas de refatoração, é possível eliminar duplicações, simplificando o design e separando melhor as responsabilidades entre diferentes componentes lógicos do programa.

- **Compreensão**
  A programação pode ser vista como a arte de expressar soluções de problemas para que o computador possa executar essas soluções. (Stroustrup , 2014). Entretanto, o código-fonte também é utilizado por outros desenvolvedores (ou pelo seu próprio criador) para modificar o programa e fazer correções até meses mais tarde. Nesses casos, o programador depende quase exclusivamente do código e de sua documentação para entender como o programa funciona. A refatoração, então, tem o papel de tornar o código mais legível, uma vez que o deixa mais estruturado. Programar torna-se, então, um modo de transmitir o significado da solução implementada, e não apenas implementar essa solução (Fowler **et al.** , 1999).

- **Erros**
  Ao deixar o código mais estruturado e compreensível, torna-se mais simples encontrar erros no código. Indiretamente, portanto, a refatoração diminui tanto a quantidade de erros presentes no programa quanto a chance de introdução novos erros.

- **Velocidade**
  Conforme descrito nos itens anteriores, um design pouco claro, com excesso de duplicações, faz com que realizar uma modificação torne-se um trabalho igualmente replicado. A chance mais alta de introduzir novos erros e a dificuldade em encontrar os antigos também aumenta o tempo gasto com correções. Ao diminuir ambos os problemas, a refatoração pode ser usada como instrumento para acelerar a velocidade de programação. A longo prazo, esses benefícios compensam o tempo investido ao refatorar, invalidando um dos principais argumentos contra o uso da refatoração em projetos de sistemas.

Tendo em vista as vantagens de realizar uma refatoração, surge naturalmente a pergunta: quais características presentes em um programa evidenciam a necessidade de refatorá-lo? No livro *Refactoring: Improving the Design of Existing Code*, Kent Beck e Martin Fowler, definem o conceito de **mau cheiro de código** (*code bad smells*) - estruturas no código-fonte de um programa que sugerem a possibilidade de refatoração (Fowler **et al.** , 1999). A lista de maus cheiros proposta por ambos está sumarizada na Tabela 1.1

1.1 ToPS

1.1.1 Arquitetura

1.1.2 Organização e distribuição

Tabela 1.1: *Maus cheiros de código*

Capítulo 2

Objetivos

Neste trabalho, tínhamos por objetivo refatorar o arcabouço de modelos probabilístico ToPS, realizando modificações em quatro níveis:

- **Código**
  Padronização do estilo de código utilizado no sistema, tendo por base as recomendações do Google C++ Style Guide. Modernização da implementação de classes e algoritmos, removendo a biblioteca boost e utilizando os recursos das novas versões 11 e 14 do C++. Remoção de todos os *warnings* de compilação em seu nível máximo. Compatibilidade total com os compiladores g++ e clang++.

- **Arquitetura**
  Modificação da hierarquia de classes de modelos probabilísticos. Criação de novos métodos e mudança do nome dos antigos para auto-documentação. Separação entre dados utilizados pelos algoritmos e classes que contêm os algoritmos

- **Componentes**
  Separação da linguagem de descrição de modelos e da biblioteca de modelos probabilísticos, tornando a segunda independente da primeira.

- **Repositório**
  Uso de um repositório de armazenamento de código com recursos mais modernos (GitHub). Criação de uma nova política de contribuição para o projeto (baseado em métodos ágeis). Integração com ferramentas de compilação automática e teste. Integração com ferramentas de coleta de métricas de qualidade do código.

Realizamos alterações no arcabouço em todos os níveis, paralelamente. Para começar, entretanto, foi necessário configurar o novo repositório e trazer o código do repositório antigo. Antes de fazer modificações, implementamos testes de unidade para os modelos probabilísticos, de forma a manter os resultados previamente produzidos e evitar a introdução de novos erros (Fowler **et al.** , 1999).

No Capítulo 3, descreveremos concretamente os recursos utilizados para realizar a refatoração, segundo os quatro níveis descritos anteriormente.

Capítulo 3

Metodologia

Para explicar os recursos utilizados na refatoração do ToPS, utilizaremos a divisão de tarefas proposta no Capítulo 2. Embora esta separação seja clara, podemos apresentá-la segundo várias abordagens (*bottom-up*, *top-down*, etc). Nas próximas seções, trataremos das alterações em **ordem crescente de complexidade** - cuja configuração também é uma boa aproximação da ordem cronológica de modificações. Embora algumas mudanças aparentem não ter ligação entre si, as escolhas feitas em um nível de abstração têm consequência de maior ou menor grau nas decisões tomadas nos outros níveis. A ordenação escolhida para a explicação, entretanto, minimiza essas dependências, definindo e utilizando novos conceitos de forma incremental. Deixamos, por fim, para o Capítulo 4 uma discussão mais abrangente das vantagens e desvantagens das escolhas feitas nas próximas seções.

3.1 Repositório

3.2 Código

3.3 Componentes

3.4 Arquitetura

Capítulo 4

Análises

Capítulo 5

Conclusões e considerações futuras

Bibliografia

Fowler et al. (1999) Martin Fowler, Kent Beck, John Brant, William Opdyke e Don Roberts. Refactoring: Improving the design of existing code. **Xtemp01**, páginas 1–337. doi: 10.1007/s10071-009-0219-y. Citado na pág. 1, 2, 3

Stroustrup (2014) Bjarne Stroustrup. **Programming: Principles and Practice Using C++ (2nd Edition)**. Addison-Wesley. Citado na pág. 2