# 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

# Agradecimentos

Um trabalho como este não poderia ser feito sem o apoio e auxílio das pessoas que me acompanharam neste último ano. Mais do que isso, seria impossível tê-lo produzido sem aqueles que me forneceram o conhecimento e experiências necessários para chegar onde estou. Muitas vezes, no curso do BCC, a disciplina de trabalho de formatura supervisionado não é a última feita pelos alunos. No meu caso, porém, este projeto marca a conclusão de uma fase - o sonho que eu realizei, quatro anos atrás, ao entrar na melhor universidade do país. Aproveito, então, essa seção para agradecer a todos que participaram dessa jornada. Não tenho, infelizmente, espaço suficiente para escrever para todos que mereceriam. Deixo então, aqui registradas, minhas palavras para os que estiveram ao meu lado nos momentos mais marcantes. Aos outros, peço desculpas pela injustiça, mas fico com a certeza de que saberão quão importante foi a sua participação nesta empreitada.

Agradeço, primeiramente, a meus pais, Alex Ferreira e Elisabete Cordeiro Silva Ferreira, que sempre apoiaram a realização dos meus sonhos. Agradeço por sempre estarem ao meu lado, desde as portas do vestibular até a apresentação do TCC, dando, para mim, o melhor presente que eu poderia pedir: sua preocupação e sua presença.

Agradeço aos meus colegas de turma, Lucas Dário, Karina Awoki, Victor Portella, Ruan Costa, Gervásio Santos, Gabriel Guilhoto, Mateus Rodrigues, Yara Gouffon, Luciana Abud, Renato Massao, Vinícius Vendramini, Helena Victoretti, Ludmila Ferreira, Vinícius Silva, Jonath Martins, Carlos Eduardo Elmadijian, Adnan Dekagi, Evandro Sanches, Lucas Hiroshi, Rafael Raposa, Pedro Rodrigues e Renan Fichberg, pelos momentos que compartilhamos ao longo desses anos de graduação. Agradeço pela amizade e por toda ajuda, sem a qual jamais poderia concluir este trabalho.

Agradeço aos meus amigos do colégio, Nathalie Rosa, Mariana Almeida, Lucas Silva, Stefânia Medeiros e Matheus de Oliveira, por provarem que a amizade pode sobreviver à distância. Agradeço por compartilharem comigo minhas alegrias, e por me ajudarem nos meus momentos de tristeza.

Agradeço ao meu colega de pesquisa e amigo Ígor Bonadio, que participou como colaborador deste trabalho. Agradeço por todas as conversas e bons ensinamentos que pudemos compartilhar.

Agradeço ao meu supervisor, Prof. Alan Durham, por ter me orientado nesse trabalho e ter feito com que eu sempre tomasse as decisões corretas. Agradeço pelos três anos de parceria, desde a minha iniciação científica até a futura pós-graduação.

Agradeço a todos os professores que, ao longo dos anos, contribuíram com minha formação.

Agradeço, por fim, a Deus, pelo conforto que encontrei nos momentos de dificuldade. Agradeço pela fé e pela força de vontade, que me guiam, juntas, pelo caminho correto.

# Resumo

O ToPS (*Toolkit for Probabilistic Models of Sequences*) é um arcabouço em C++ utilizado para treinamento e inferência de modelos probabilísticos usados para descrição de sequências finitas de símbolos. Possui uma série de aplicativos e uma linguagem de especificação, que permite configurar os modelos sem conhecimento prévio de programação. Neste projeto, propomo-nos a realizar uma refatoração do arcabouço, visando melhorá-lo em três aspectos: compreensibilidade, mutabilidade e extensibilidade. Para cumprir com esse objetivo, realizamos diversas alterações em quatro camadas do sistema: código, arquitetura, componentes e repositório. Definimos uma série de processos e verificações para contribuição, e montamos um ambiente com ferramentas que facilitam o desenvolvimento. Internamente, padronizamos a formatação do código-fonte, e removemos todos os problemas de compilação. No design, desacoplamos a linguagem da estrutura interna, e propusemos um novo padrão de arquitetura para organizar o acesso às funcionalidades dos modelos. Por meio dessas mudanças, buscamos aproximar o ToPS dos três atributos de qualidade desejados, seguindo os princípios da programação orientada a objetos (SOLID). Esperamos, assim, facilitar o uso do arcabouço, tanto na criação de novas aplicações quanto no uso em futuras pesquisas.

**Palavras-chave:** ToPS, refatoração, modelos probabilísticos.

# Capítulo 1

# Introdução

## 1.1 Motivação

A Bioinformática é uma área de pesquisa interdisciplinar que utiliza as técnicas da Ciência da Computação para o estudo de fenômenos biológicos. Nas últimas décadas, a criação de novas tecnologias de sequenciamento permitiu a determinação de sequências DNA, RNA e proteínas para um número cada vez maior de espécies. Para lidar com a grande quantidade de dados produzida, foram desenvolvidos programas de computador que fazem buscas por similaridades e categorização, permitindo a automatização de análises manuais e a criação de novas possibilidades de estudo para a biologia molecular moderna.

Uma das tarefas exploradas dentro do campo da Bioinformática é a **predição de genes**. Os **genes** são regiões do DNA de um organismo que são transcritas para RNA. Os RNAs, por sua vez, são traduzidos para proteínas (RNAs mensageiros), ou assumem funções reguladores e estruturais dentro das células (RNAs não codificantes). Indiretamente, então, os genes influenciam o comportamento celular, servindo como repositório para a informação que codifica os RNAs. Chamamos de **predição de genes *in silico*** o uso de programas de computador para encontrar genes, e damos o nome de **preditores de genes** aos programas que realizam essa tarefa.

Os preditores de genes do tipo ***ab initio*** recebem como entrada apenas a sequência de DNA de um certo organismo. Utilizando Modelos Ocultos de Markov Generalizados (*Generalized Hidden Markov Models*, GHMMs) ou Campos Aleatórios Condicionais (*Conditional Random Fields*, CRFs), esses preditores apresentam uma arquitetura de modelos probabilísticos que é capaz de descrever as diferentes regiões de um gene. Com base nessa estrutura, os programas funcionam em duas etapas: o **treinamento**, no qual estimam-se os parâmetros dos modelos; e a **predição**, na qual encontram-se as regiões da sequência de DNA com maior probabilidade de serem genes.

Atualmente, existem vários preditores de genes disponíveis de forma aberta para uso em pesquisas: Genscan (Stanke e Waack, 2003), SNAP (Korf, 2004), Augustus (Stanke e Waack, 2003)), CRAIG (DeCaprio et al., 2007), Conrad (Bernal et al., 2007), dentre outros. Entretanto, a maioria deles apresenta uma série de problemas que dificultam a sua utilização (Kashiwabara, 2012):

- Poucos programas aproveitam-se da modularização natural dos modelos probabilísticos para melhorar a modelagem do seu código;
- Muitos sistemas só permitem a modificação de parâmetros e da arquitetura de modelos via alterações no código-fonte, dificultando o acesso a usuários sem conhecimento de programação;

- Alguns sistemas não oferecem algoritmos de treinamento, ou não facilitam o treinamento para organismos fora do conjunto para o qual o preditor originalmente foi desenvolvido; e
- Poucos programas aproveitam-se da capacidade de paralelismo das máquinas mais modernas.

O **MYOP** (*Make Your Own Predictor*) é um sistema gerador de preditores de genes que propõe-se a resolver esses problemas. Desenvolvido na tese de doutorado de André Kashiwabara (Kashiwabara, 2012), visa facilitar a construção de novas arquiteturas de modelos probabilísticos sem exigir que os usuários tenham conhecimento prévio de programação. Ao mesmo tempo, busca manter o máximo de performance possível, aproveitando-se do paralelismo e de melhorias algorítmicas. Para explorar essas otimizações, o MYOP foi construído sobre o **ToPS** (*Toolkit of Probabilistic Models of Sequences*), arcabouço que implementa algoritmos de treinamento e inferência de modelos probabilísticos que descrevem sequências finitas de símbolos (Kashiwabara et al., 2013). Essa separação em dois sistemas permite que o ToPS seja reutilizado em outros campos de estudo (como processamento de linguagem natural), enquanto o MYOP fornece uma interface específica para os usuários interessados em predição de genes.

A primeira versão do ToPS, publicada em 2013 na revista PLOS Computational Biology, apresenta 8 modelos probabilísticos que podem ser utilizados em aplicações de Bioinformática (Tabela 1.1a). Atualmente, o sistema continua sendo ativamente desenvolvido, com 5 novos modelos total ou parcialmente implementados como parte das teses de mestrado e doutorado dos alunos Vitor Onuich (Onuchic, 2012), Rafael Mathias (Mathias, 2015) e Ígor Bonadio (Bonadio, 2013) (Tabela 1.1b). Além dessas adições, o Modelo de Covariância (*Covariance Model*, CM), descrito no livro de Durbin et al., está entre as extensões planejadas para o sistema.

Por conta da variedade de modelos disponíveis e do número de contribuidores que alteraram o ToPS, o sistema ganhou cada vez mais complexidade. A curva de aprendizado para novos desenvolvedores aumentou, dificultando a tarefa de estender e modificar o arcabouço. Tal rigidez aumenta a chance de introduzir erros, e prejudica o reúso de código nos novos modelos que estão sendo implementados. Esses problemas tendem a se tornar mais críticos com a publicação do MYOP, quando o ToPS passará a ter, indiretamente, mais usuários. É essencial, portanto, que o arcabouço possa receber manutenções, correções e adições mais facilmente, sem que as mudanças feitas nele prejudiquem o seu funcionamento. Resolver esse problema, e portanto manter o impacto que a qualidade do ToPS e do MYOP trouxeram para a Bioinformática, é a principal motivação deste trabalho.

| Modelos probabilísticos já publicados |
|---|
| Processo Independente e Identicamente Distribuído |
| Cadeia de Markov de Alcance Variável |
| Cadeia de Markov Não Homogênea |
| Modelo Oculto de Markov |
| Modelo Oculto de Markov Pareado |
| Modelo Oculto de Markov para Perfis de Sequências |
| Modelo Oculto de Markov Generalizado |
| Ponderamento Baseado na Similaridade de Sequência |

**(a)** *Modelos publicados na primeira versão do ToPS (Kashiwabara et al., 2013)*

| Modelos a serem implementados |
|---|
| Modelo de Covariância |

| Modelos não publicados |
|---|
| Modelo Oculto de Markov Pareado Generalizado |
| Modelo Oculto de Markov Sensível ao Contexto |
| Modelo Oculto de Markov Sensível ao Contexto para Perfis de Sequências |
| Campo Aleatório Condicional de Cadeias Lineares |
| Campo Aleatório Condicional Semi Markoviano |

**(b)** *Modelos implementados por Vitor Onuchic, Rafael Mathias e Ígor Bonadio.*

**Tabela 1.1:** *Modelos probabilísticos do ToPS*

## 1.2 Objetivos

Neste trabalho, temos como objetivo propor e implementar uma série de modificações para o ToPS, com o intuito de melhorar a capacidade de extensão, modificação e compreensão a longo prazo do sistema. Para identificar os pontos que podem ser aperfeiçoados, realizaremos uma análise do arcabouço, revisando desde detalhes da escrita do código até o modelo de contribuição atualmente adotado. A fim de avaliar concretamente o impacto dessas mudanças, faremos uma pequena revisão dos problemas de design que podem ser encontrados em um programa, posteriormente indicando os benefícios de resolvê-los. Com isso, buscaremos mostrar que as alterações feitas no sistema realmente tiveram o impacto desejado: a melhoria da qualidade do ToPS.

Como compreender e modificar um sistema grande e complexo é uma tarefa árdua, contaremos com a colaboração do aluno de doutorado Ígor Bonadio, também do grupo do professor Alan Durham, para ajudar nas discussões e implementações feitas ao longo deste trabalho. Essa parceria é importante para que possamos aprender sobre o sistema com um desenvolvedor que já trabalhou diretamente nele (implementando o modelo de Campo Aleatório Condicional de Cadeias Lineares, *Linear Chain Conditional Random Field*, LCCRF) (Bonadio, 2013). Além disso, será possível obtermos *feedbacks* sobre os benefícios das alterações feitas no código, dado que o aluno Ígor usará a nova versão do arcabouço para as extensões do ToPS propostas na sua tese (adicionando o modelo de Campo Aleatório Condicional Semi Markoviano, *Semi Markov Conditional Random Field*, SemiCRF). Ao longo do texto, deixaremos explícitas todas contribuições feitas pelo aluno Ígor.

No próximo capítulo (Capítulo 2), apresentaremos os atributos desejáveis que formam um bom programa. Introduziremos a ideia de **refatoração**, e discutiremos os benefícios de utilizá-la para melhorar o design do código de um sistema. Em seguida, definiremos o conceito de **mau cheiro**, e o utilizaremos para examinar os detalhes da implementação do ToPS que evidenciam a necessidade de refatorá-lo (Capítulo 3). Finalmente, apresentaremos as mudanças realizadas no arcabouço e as suas respectivas consequências (Capítulo 4), concluindo com um panorama geral dos resultados deste trabalho e as possíveis extensões que podem ser feitas a partir dele (Capítulo 6).

# Capítulo 2

# Qualidade de Software

Ao analisar um sistema, podemos definir uma série de atributos desejáveis que indicam a sua boa qualidade. Em particular, as características relativas ao **design de código** são conhecidas por terem um impacto direto na criação de um bom programa (Suryanarayana et al., 2014). As principais propriedades que se enquadram nessa categoria estão listadas na Tabela 2.1

Todo programa não trivial sofre modificações ao longo do tempo. Quanto mais transformações são feitas, mais o sistema tende a afastar-se dos seus atributos desejáveis. Ao fazer mudanças para cumprir objetivos de curto-prazo, ou criar alterações sem o entendimento completo do design do sistema, o código começa a perder sua estrutura, tornando-se mais difícil e custoso de manter (Fowler e Beck, 1999). Para combater essa perda de qualidade, podemos utilizar a técnica de **refatoração**: conceito que pode ser definido de formas ligeiramente distintas, segundo o contexto de utilização da palavra:

| Atributos de qualidade | Definição |
|---|---|
| Compreensibilidade | Facilidade com que um fragmento de design pode ser compreendido. |
| Mutabilidade | Facilidade com que um fragmento de design pode ser modificado (sem efeitos em cascata) quando uma funcionalidade existente é alterada. |
| Extensibilidade | Facilidade com que um fragmento de design pode ser melhorado ou estendido (sem efeitos em cascata) para suportar novas funcionalidades. |
| Reusabilidade | Facilidade com que um fragmento de design pode ser utilizado no contexto de um problema diferente daquele para o qual o fragmento de design foi originalmente desenvolvido. |
| Testabilidade | Facilidade com que um fragmento de design suporta a detecção de defeitos dentro dele via testes. |
| Confiabilidade | Extensão na qual um fragmento de design suporta a correta concepção da funcionalidade e ajuda a guardar contra a introdução de problemas em tempo de execução. |

**Tabela 2.1:** *Atributos de qualidade e suas definições (Suryanarayana et al., 2014). Os atributo são definidos em termos de* ***fragmentos de design****: abstrações simples (classes, interfaces, classes abstratas, etc.) ou conjunto de abstrações interrelacionadas (hierarquias de herança, grafos de dependência, etc.).*

- **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.

Em ambos os casos, **refatorar** implica necessariamente em **não modificar** o comportamento existente no programa. Realizar qualquer mudança, porém, demanda tempo, que poderia ser utilizado para a implementação de novas funcionalidades. A primeira vista, portanto, refatorar pode parecer um desperdício. Entretanto, ao analisar mais profundamente o caráter dessas alterações, é possível notar uma série de vantagens a longo prazo, que são responsáveis por minimizar esse custo (Fowler e Beck, 1999). Dentre esses benefícios, podemos citar:

- **Design**
  Conforme descrito no início da seção, modificações contínuas num sistema tendem a prejudicar a sua estrutura. Em geral, isso é causado pela presença de **duplicações** - pedaços de código que realizam a mesma tarefa, em partes diferentes do programa. Para alterar a funcionalidade implementada por esses fragmentos, é preciso modificar vários lugares, aumentando a chance de introduzir erros. Além disso, sem uma boa forma de documentação, é possível que alguns dos trechos sejam esquecidos, causando inconsistências. Utilizando as técnicas de refatoração, é possível eliminar essas duplicações, simplificando o design e separando melhor as responsabilidades entre diferentes componentes lógicos do programa.

- **Compreensão**
  De forma geral, o código-fonte de um sistema não é utilizado apenas uma vez, sendo modificado pelo seu próprio criador ou por outros desenvolvedores para adicionar funcionalidades e correções. Essas mudanças, quando feitas, podem ter até meses de intervalos entre si. Nessa situação, o programador depende quase exclusivamente do próprio código e da sua documentação para voltar a compreender o funcionamento daquela parte do programa. A refatoração, nesse contexto, tem o papel de deixar o código mais legível, mantendo-o sempre estruturado. Programar, então, torna-se mais que simplesmente *solucionar um problema*: passa a ser o modo de *comunicar o significado dessa solução*.

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

- **Velocidade**
  Um design pouco claro, com excesso de duplicações, faz com que modificar um sistema seja um trabalho igualmente replicado. A chance de introduzir erros e a dificuldade de encontrar os antigos amplia o tempo gasto com correções. Ao diminuir esses problemas, a refatoração pode ser usada como instrumento para acelerar a velocidade de programação, compensando, a longo prazo, o tempo investido ao refatorar.

| Princípios de design | Descrição |
|---|---|
| Abstração | Separação de entidades por **redução** e **generalização**. Redução é a eliminação de detalhes desnecessários. Generalização é a identificação e especificação das características comuns e importantes. |
| Encapsulamento | **Separação de responsabilidades** e **ocultamento de informação** por técnicas como ocultamento de detalhes de implementação e ocultamento de variações. |
| Modularização | Criação de **abstrações coesas e fracamente acopladas** por meio de técnicas como localização e decomposição. |
| Hierarquia | Criação de uma **organização hierarquica de abstrações** por técnicas como classificação, generalização, substituibilidade e ordenação. |

**Tabela 2.2:** *Princípios de design, utilizados no esquema de classificação de maus cheiros apresentado em Suryanarayana et al., 2014 e baseados no modelo de objetos de Booch, 2004.*

Tendo em vista as vantagens de realizar uma refatoração, ainda é necessário identificar quais características presentes em um programa evidenciam a necessidade de refatorá-lo. De forma geral, se um sistema atende a todos os atributos de qualidade, não é necessário modificá-lo. Contudo, para termos uma forma concreta de analisar o código de um programa qualquer, Fowler e Beck criaram o conceito de **mau cheiro** (*bad smell*): estruturas presentes no código-fonte de um programa que sugerem a possibilidade de refatoração.

Neste trabalho, utilizaremos o sistema de classificação de maus cheiros proposto por Suryanarayana et al., que define maus cheiros como sendo "estruturas que indicam a violação de princípios fundamentais do design (Tabela 2.2), impactando negativamente na qualidade do código". Como não encontramos uma versão oficial do texto em português, criamos uma tradução do esquema proposto no livro para ser utilizada neste trabalho. Nosso sistema de nomeação, apresentado no Diagrama 2.1, visa se aproximar ao máximo da praticidade do original. A identificação de cada mau cheiro será composta de duas partes: um substantivo, que é o nome do princípio de design violado; e um adjetivo ou locução adjetiva, que descreve a violação. Adicionalmente, apresentamos no Diagrama 2.1 uma descrição de como cada mau cheiro se expressa no código de um programa.

No Capítulo 3, utilizaremos o conceito de mau cheiro para analisar os problemas presentes na arquitetura do ToPS.

## Abstração

**Abstração não encontrada**

Este mau cheiro ocorre quando aglomerados de dados e strings são usadas no lugar de classes e interfaces

**Abstração desnecessária**

Este mau cheiro ocorre quando uma abstração é introduzida no design de um programa sem necessidade (podendo, assim, ser evitada).

**Abstração imperativa**

Este mau cheiro surge quando uma operação ou algoritmo é transformada em uma classe.

**Abstração não utilizada**

Este mau cheiro surge quando uma abstração não é diretalmente utilizada / alcançável.

**Abstração incompleta**

Este mau cheiro surge quando uma abstração não suporta completamente métodos complementares ou interrelacionados a si.

**Abstração duplicada**

Este mau cheiro surge quando duas ou mais abstrações têm o mesmo nome, a mesma implementação ou ambos.

**Abstração multifacetada**

Este mau cheiro surge quando uma abstração possui mais de uma responsabilidade atribuída a si.

## Modularização

**Modularização quebrada**

Este mau cheiro surge quando dados e/ou métodos que idealmente deveriam estar localizados em uma única abstração estão espalhados entre várias abstrações.

**Modularização com dependência cíclica**

Este mau cheiro surge quando duas ou mais abstrações dependem uma da outra, direta ou indiretamente (criando um acoplamento entre as abstrações)

**Modularização insuficiente**

Este mau cheiro surge quando uma abstração não foi completamente decomposta, e essa decomposição poderia reduzir o seu tamanho, complexidade de implementação, ou ambos.

**Modularização centralizadora**

Este mau cheiro surge quando uma abstração tem dependências (de entrada e de saída) com um grande número de abstrações.

## Encapsulamento

**Encapsulamento deficiente**

Este mau cheiro ocorre qunado o acesso a um ou mais membros de uma abstração é mais permissivo do que o realmente necessário.

**Encapsulamento não explorado**

Este mau cheiro surge quando o código do cliente precisa utilizar checagem explícita de tipos (switch-case/if-else) no lugar de explorar a variação de tipos em uma hierarquia.

**Encapsulamento com vazamento**

Este mau cheiro surge quando uma abstração "expõe" ou "vaza" seus detalhes de implementação em sua interface pública.

**Encapsulamento não encontrado**

Este mau cheiro ocorre quando variações em tipos ou hiearquias não são encapsuladas.

## Hierarquia

**Hierarquia não encontrada**

Este mau cheiro surge quando usam-se "tags" e condicionais para administrar a variação de comportamento, sendo que a variação poderia ser encapsulada por uma hierarquia.

**Hierarquia profunda**

Este mau cheiro surge quando uma hierarquia de herança é excessivametne profunda.

**Hierarquia rebelde**

Este mau cheiro surge quando um subtipo rejeita métodos do(s) seu(s) supertipo(s).

**Hierarquia desnecessária**

Este mau cheiro surge quando toda uma hierarquia de herança aplicada sem necessidade no contexto específico.

**Hierarquia quebrada**

Este mau cheiro surge quando um supertipo e seus subtipos não compartilham a relação conceitual "é-um", quebrando a substituição.

**Hierarquia não fatorada**

Este mau cheiro surge quando há duplicação desnecessária entre tipos de uma hierarquia.

**Hierarquia de múltiplos caminhos**

Este mau cheiro surge quando um subtipo herda direta e indiretamente de um supertipo, criando a caminhos dispensáveis na herança.

**Hierarquia larga**

Este mau cheiro surge quando uma hierarquia de herança é "muito larga", indicando que podem faltar abstrações intermediárias.

**Hierarquia especulativa**

Este mau cheiro surge quando um ou mais tipos em uma hierarquai são fornecidos de forma especulativa (baseados em requisitos imaginados, não reais).

**Hierarquia cíclica**

Este mau cheiro surge quando um supertipo depende de um de seus subtipos.

**Diagrama 2.1:** *Maus cheiros de design (Suryanarayana et al., 2014) e suas descrições. O nome de cada mau cheiro é composto de duas partes: o princípio de design violado (1ª palavra), e um adjetivo ou locução adjetiva que descreve a violação (restante).*

# Capítulo 3

# ToPS

Neste capítulo, temos como objetivo apresentar os detalhes de implementação do ToPS, focando nos aspectos técnicos que podem ser alterados para melhorar a qualidade do sistema. Aqui, quando nos referirmos ao nome "ToPS", estaremos tratando da versão publicada na revista PLOS Computational Biology (Kashiwabara et al., 2013), que implementa os modelos listados na Tabela 1.1a. No restante do trabalho, usaremos essa versão como base para as propostas de modificação do sistema.

Para organizarmos melhor as análises feitas ao longo do texto, dividiremos nossas discussões em quatro categorias: **código**, **arquitetura**, **componentes**, e **repositório**. Buscamos, assim, facilitar o pensamento nas diferentes camadas do sistema, que, embora interrelacionadas, apresentam particularidades pouco relevantes fora do seu próprio escopo. Para este capítulo, optaremos por apresentar os detalhes de funcionamento do ToPS começando pelo seu código (Seção 3.1). Em seguida, trataremos da arquitetura do sistema (Seção 3.2) e da sua divisão em diferentes componentes lógicas (Seção 3.3). Por fim, encerraremos com uma discussão sobre o modelo de distribuição e contribuição atualmente utilizado no projeto (Seção 3.4). No final de cada seção, ressaltaremos os pontos de melhoria que serão posteriormente trabalhados no Capítulo 4.

## 3.1 Código

Conforme descrito no Capítulo 1, o ToPS é um sistema originalmente construído como base para o preditor de genes MYOP, contendo modelos probabilísticos utilizados para descrição de sequências finitas de símbolos. Como os algoritmos implementados pelo ToPS são computacionalmente custosos (Durbin et al., 1998, Kashiwabara, 2012, Onuchic, 2012, Bonadio, 2013, Mathias, 2015), seus autores optaram por desenvolvê-lo na linguagem C++, que apresenta uma série de vantagens para programas voltados a realizar processamento intensivo, tais como:

- Compilação direta para código objeto executável (sem máquinas virtuais ou *bytecode*);
- Possibilidade de modelagem de problemas utilizando vários paradigmas de programação (procedural, funcional e, principalmente, orientação a objetos) (Stroustrup, 2014);
- Metaprogramação (para criação de abstrações de alto nível e reutilização de código) em tempo de compilação (Meyers, 2005);
- Biblioteca padrão com implementações otimizadas (Meyers, 2001);

- Ampla disponibilidade de bibliotecas abertas com foco em performance[1]; e
- Facilidade para uso de concorrência de forma nativa (Williams, 2012) ou por bibliotecas[2].

O código do ToPS usa a biblioteca de templates Boost[3] para estender a capacidade de programação orientada a objetos do C++98 (versão do C++ padronizada em 1998). Os **templates** são um recurso nativo do C++ que pode ser usado para a criação de tipos e funções parametrizadas, servindo como uma forma geral de meteprogramação (Stroustrup, 2013). Dentre as classes oferecidas pela biblioteca Boost, os **ponteiros inteligentes** (*smart pointers*) estão entre os mais usadas pelo ToPS, pois utilizam a técnica RAII (*Resource Acquisition Is Initialization*) para alocar memória nos seus construtores e liberá-la nos seus destrutores, atuando como um método simplificado de coleta de lixo (*garbage collection*) (Meyers, 2005).

Para a compilação, o ToPS usa a ferramenta CMake[4], que tem ganhado popularidade frente a alternativas como Makefiles ou a suite de ferramentas Autotools[5] da GNU. O grande diferencial do CMake é permitir a geração de scripts de compilação (Makefiles, projetos para XCode, projeto para Visual Studio, etc.) segundo a plataforma no qual está sendo utilizado.

### Melhorias

Em termos de escrita, o código-fonte do ToPS possui pouca padronização. Por ter sido desenvolvido, inicialmente, por apenas uma pessoa (Kashiwabara, 2012) e depois ter sido modificado em vários trabalhos independentes (Onuchic, 2012, Bonadio, 2013, Mathias, 2015), cada parte do código adquiriu o estilo particular do seu desenvolvedor. Embora esses estilos não sejam muito diferentes e os programadores consigam lidar com essas pequenas diferenças, a falta de uma formatação única tende a desestimular as refatorações e a diminuir a velocidade com que correções são feitas (Beck, 1999), tornando mais difícil manter consistência e robustez do programa.

Com relação ao tempo de compilação, o ToPS leva cerca de 8 min para compilar num processador Intel® Core™ i7-4510U CPU de 2.00GHz. Essa demora pode ser explicada por dois fatores:

- a geração de código para todos os templates da biblioteca Boost; e
- as otimizações de compilação, que demandam análises extras sobre as estruturas de representação intermediária do compilador para gerar um código objeto mais rápido (Appel, 2004).

Como resultado, o compilador produz um total de 68 *warnings*, a maioria por problemas de conversão entre números com e sem sinal. A presença deles, especialmente em número tão alto, prejudica o desenvolvimento, dado que, com o tempo, os desenvolvedores tendem a ignorá-los (Sutter e Alexandrescu, 2004), deixando passar avisos sobre problemas mais graves. Compilar sem *warnings* ou desativá-los localizadamente é a melhor forma de evitar esse problema.

Em termos de dependências, a biblioteca Boost já não traz mais tantos benefícios. O lançamento da nova versão do C++ em 2011 (C++11) incluiu na biblioteca padrão de templates (*Standard Template Library*, STL) praticamente todos os recursos da `Boost` usados pelo ToPS. Na compilação, a flexibilidade do CMake é pouco explorada. Assim, remover ambas as dependências pode facilitar o uso do ToPS em seus programas clientes, além de simplificar o processo de instalação.

[1] **A list of open source C++ libraries** n.d.
[2] **OpenMP** n.d.
[3] **Boost C++ Libraries** n.d.
[4] **CMake** n.d.
[5] **Automake** n.d.

## 3.2 Arquitetura

Estruturalmente, o ToPS pode ser classificado como um **arcabouço**, que segundo Ralph Johnson (Johnson, 1997) é um "projeto de larga escala que descreve como um programa é decomposto em um conjunto de objetos que interagem entre si". Ainda segundo o autor, um arcabouço é, de forma geral, "representado por uma série de classes abstratas e o modo como suas instâncias interagem". Diferentemente de uma biblioteca, que costuma ter uso mais genérico, um arcabouço fornece um conjunto de classes que ajuda a resolver problemas de um dado domínio específico. A **arquitetura** na qual essas classes estão organizadas especifica como elas serão usadas pelas **aplicações** do arcabouço, e serve para ajudar a resolver os problemas desse dado domínio. No ToPS, o domínio de aplicação se refere ao treinamento e inferência de modelos probabilísticos que descrevem sequências de símbolos. O MYOP, ao usar esses modelos para realizar predição de genes, pode ser classificado como uma aplicação do ToPS,

O ToPS utiliza orientação a objetos para representar internamente os modelos descritos na Tabela 1.1. Além desse núcleo, implementa uma série de **aplicativos** que permitem o acesso, em linha de comando, às ações que podem ser realizadas sobre esses modelos (treinamento, inferência, cálculo de probabilidades, etc.). Para criação e armazenamento dos modelos entre execuções dos programas, o ToPS possui uma **linguagem de especificação** (Kashiwabara et al., 2013), que facilita o uso do sistema por usuários não programadores. Para ler e construir uma representação dos componentes dessa linguagem internamente, o ToPS tem um *parser* gerado pelas ferramentas Flex[6] e Bison[7], que convertem o texto dos arquivos em uma árvore sintática abstrata (*Abstract Syntax Tree*, AST) manipulável pelo sistema. Para modelar esse fluxo de criação e representação dos modelos, o ToPS organiza sua arquitetura em torno de 3 hierarquia de classes:

- A **hierarquia de modelos probabilísticos** ou **hierarquia principal** (Diagrama 3.1), que contém sob uma mesma interface (`tops::ProbabilisticModel`) todas as classes concretas que representam os modelos probabilísticos. Abaixo dessa interface, outras classes abstratas intermediárias separam os modelos em categorias, conforme os métodos que implementam:
  - **Modelos fatoráveis** (`tops::FactorableModel`), em que a probabilidade de uma sequência pode ser dada pelo produto das probabilidades de uma partição da sequência;
  - **Modelos fatoráveis não homogêneos** (`tops::InhomogeneusFactorableModel`), modelos fatoráveis cuja probabilidade de uma mesma subsequência pode variar segundo a posição em que ela se inicia.
  - **Modelos decodificáveis** (`tops::DecodableModel`), que possuem estados cuja emissão, transição e duração são descritos via outros modelos da hierarquia. Por conta disso, a hierarquia principal segue o padrão de projeto **Composite** (Gamma et al., 1995).
  - **Modelos decodificáveis pareados** (`tops::PairDecodableModel`), modelos decodificáveis que fazem alinhamento de pares de sequências (e, portante, recebem duas sequências em seus métodos).
  - **Decoradores de modelos** (`tops::ProbabilisticModelDecorator`), modelos que estendem funcionalidades de outros modelos já existentes, segundo o padrão **Decorador** (Gamma et al., 1995).

[6] **Bison - GNU Project - Free Software Foundation** n.d.
[7] **Flex: The Fast Lexical Analyzer** n.d.

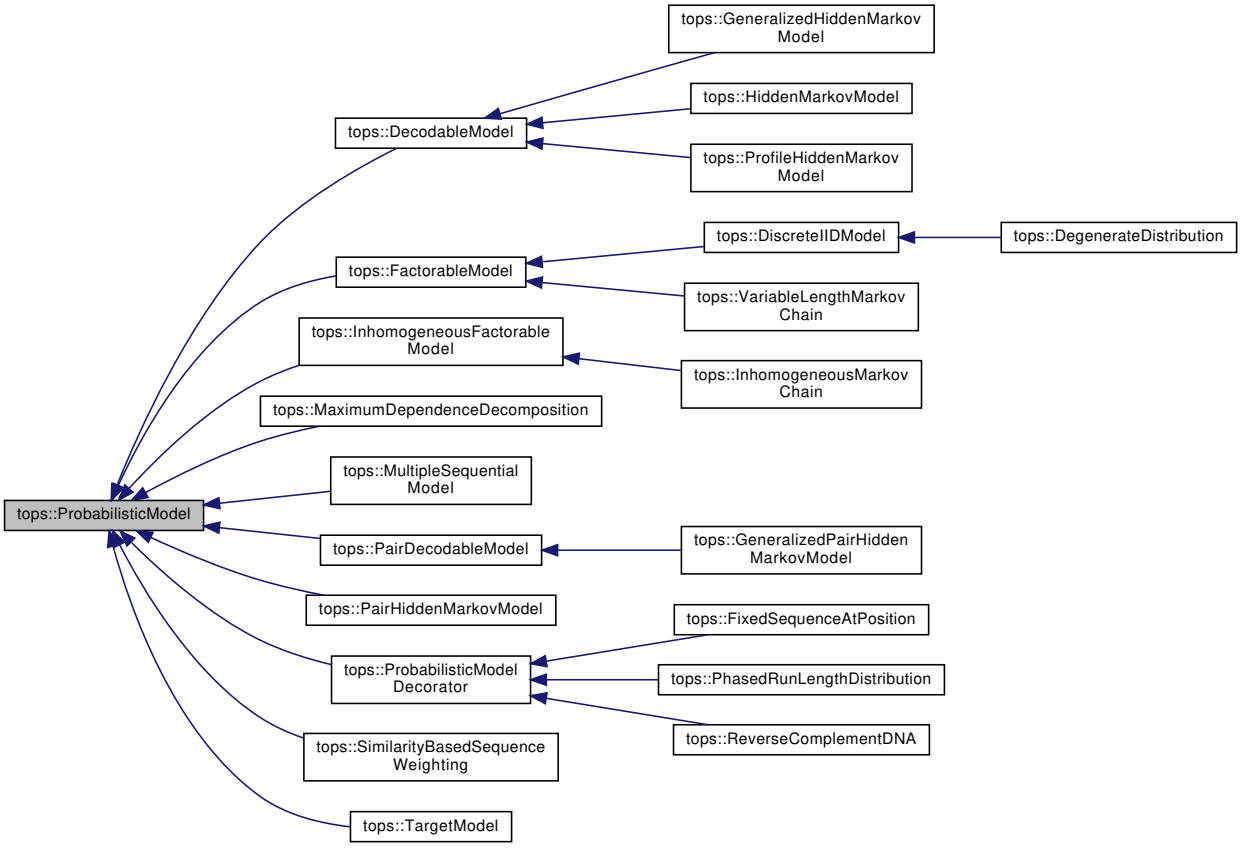



**Diagrama 3.1:** *Hierarquia de modelos probabilísticos do ToPS, gerada pela ferramenta de documentação Doxygen. As caixas brancas representam classes, e as flechas azuis denotam herança. O diagrama pode ser vista com mais detalhes em http://tops.sourceforge.net/doc/html/index.html*

- A **hierarquia de nós da AST** (Diagrama 3.2), que define uma interface para os elementos da linguagem de especificação ToPS (`tops::lang::ASTNode`). A linguagem do ToPS, feita para ser simples, busca se aproximar da descrição matemática dos modelos (Kashiwabara, 2012). Por esse motivo, a classes da AST representam poucas categorias de elementos presentes na linguagem, tais como parâmetros dos modelos, valores simples (inteiros, ponto flutuante) e agregados (listas) que podem ser atribuídas a esses parâmetros.

- A **hierarquia de fábricas de modelos probabilísticos** (Diagrama 3.3), cuja única interface (`tops::ProbabilisticModelCreator`) contém diversos **métodos fábrica** (Gamma et al., 1995) para gerar modelos a partir de parâmetros da AST. As classes que fazem parte dessa hierarquia constroem modelos simples (como `tops::HiddenMarkovModelCreator`) e criam modelos treinados (como `tops::TrainHMMBaumWelch`) por algum algoritmo de treinamento . Outras classes, ainda, realizam tarefas específicas para alguns aplicativos (como selecionar o melhor entre dois modelos, em `tops::AkaikeInformationCriteria`).

## Melhorias

Para encontrarmos os possíveis pontos de melhoria no design de classes do ToPS, vamos utilizar o conceito de maus cheiros, descrito em detalhes no Capítulo 2. A fim de facilitar o entendimento, vamos analisar cada hierarquia separadamente, deixando para a próxima seção (Seção 3.3) a análise dos relacionamentos entre elas.

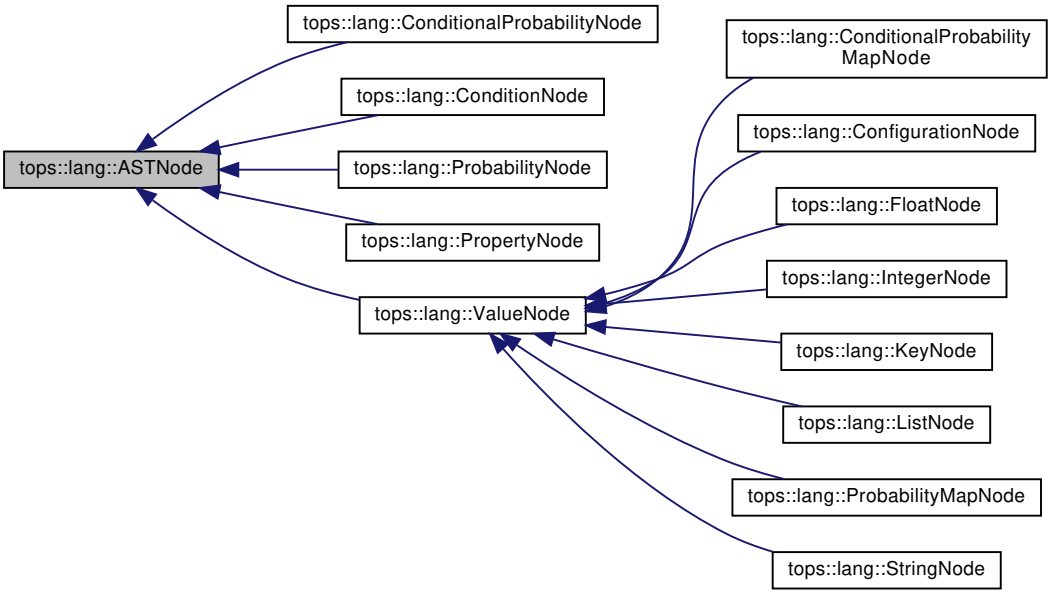



**Diagrama 3.2:** *Hierarquia de nós da AST da linguagem de especificação do ToPS, gerada pela ferramenta de documentação Doxygen. O diagrama pode ser vista com mais detalhes em http://tops.sourceforge.net/doc/html/index.html*

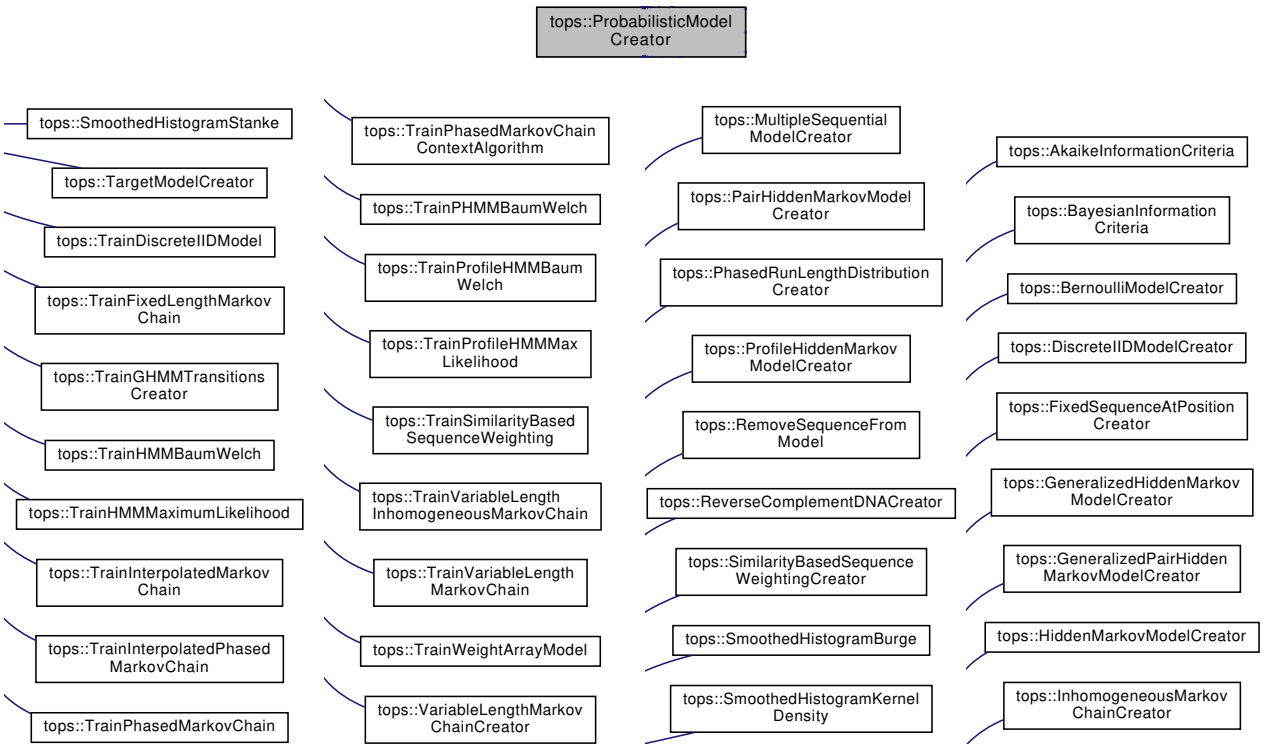



**Diagrama 3.3:** *Hierarquia de fábricas de modelos probabilísticos do ToPS, gerada pela ferramenta de documentação Doxygen. O diagrama pode ser vista com mais detalhes em http://tops.sourceforge.net/doc/html/index.html*

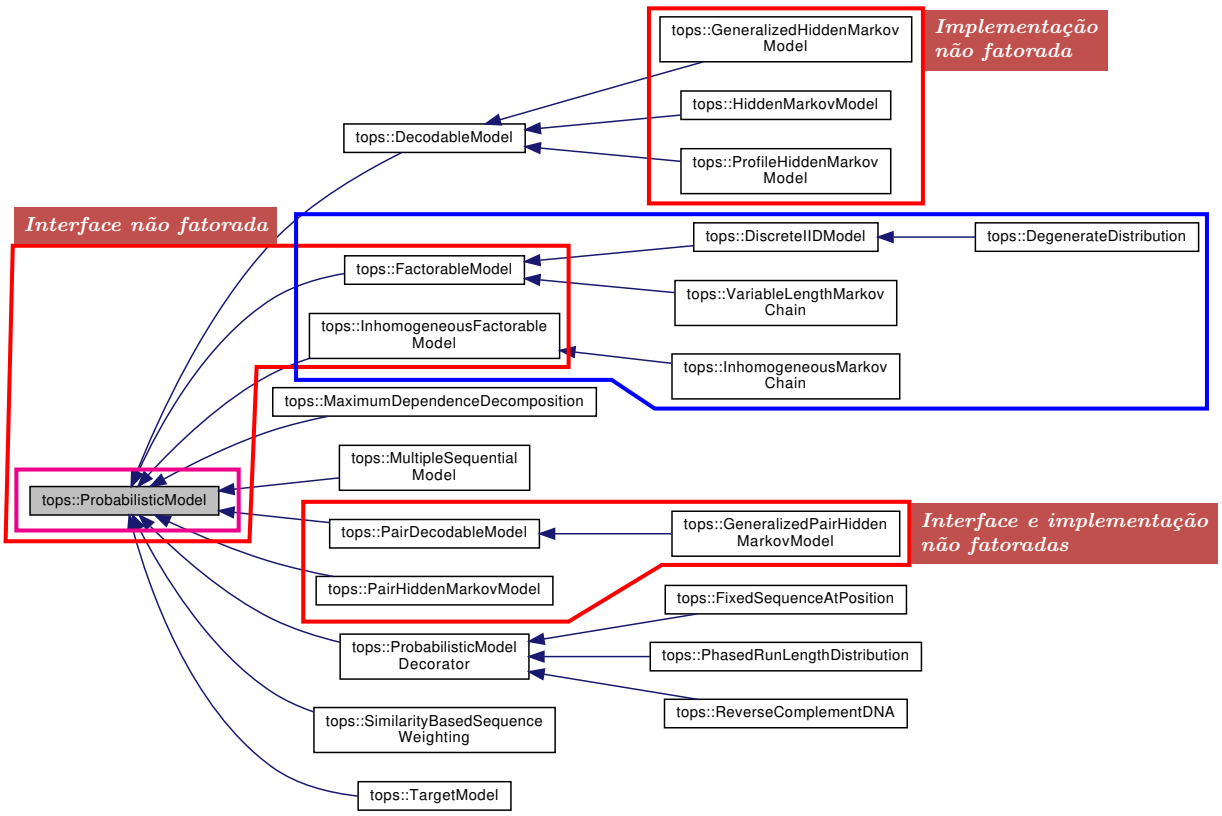



**Diagrama 3.4:** ***Maus cheiros de design na hierarquia de Modelos Probabilísticos:*** *Analisando os métodos da hierarquia de modelos probabilísticos, identificamos a presença de 3 maus cheiros:* ***Hierarquia não fatorada*** ■ *(expresso em 3 configurações), que indica a presença de duplicações desnecessárias de tipos;* ***Hierarquia desnecessária*** ■*, que indica a existência de caminhos que não acrescentam ganhos à hierarquia; e* ***Abstração multifacetada*** ■*, que indica que as principais abstrações possuem múltiplas responsabilidades.*

- **Hierarquia de modelos probabilísticos**

  Ao analisar os métodos da hierarquia de modelos probabilísticos do ToPS (Diagrama 3.1), é possível encontrar duplicações em diversos pontos:

  1. As interfaces `tops::ProbabilisticModel`, `tops::FactorableModel` e `tops::InhomogeneusFactorableModel` não possuem quase nenhum método de diferença;
  2. As classes `tops::HiddenMarkovModel`, `tops::ProfileHiddenMarkovModel` e `tops::GeneralizedHiddenMarkovModel` tem vários algoritmos com implementações similares;
  3. A classe `tops::PairHiddenMarkovModel` é similar à classe `tops::GeneralizedPairHiddenMarkovModel`, mas não herda de `tops::PairDecodableModel`.

  Essas duplicações são características de uma **hierarquia não fatorada** (Diagrama 2.1), expressa respectivamente nas suas três principais formas (Suryanarayana et al., 2014): *interface não fatorada*, *implementação não fatorada* e *interface e implementação não fatoradas*.

  Por conta das interfaces do Item 1 serem muito similares, criar uma definição mais abrangente de `tops::ProbabilisticModel` possibilitaria que todas as classes filhas de `tops::FactorableModel` e `tops::InhomogeneusFactorableModel` herdassem diretamente dessa classe abstrata. Logo, ambas as sub-hierarquias com raízes nessas interfaces formam exemplos de **hierarquias desnecessárias** (Diagrama 2.1).

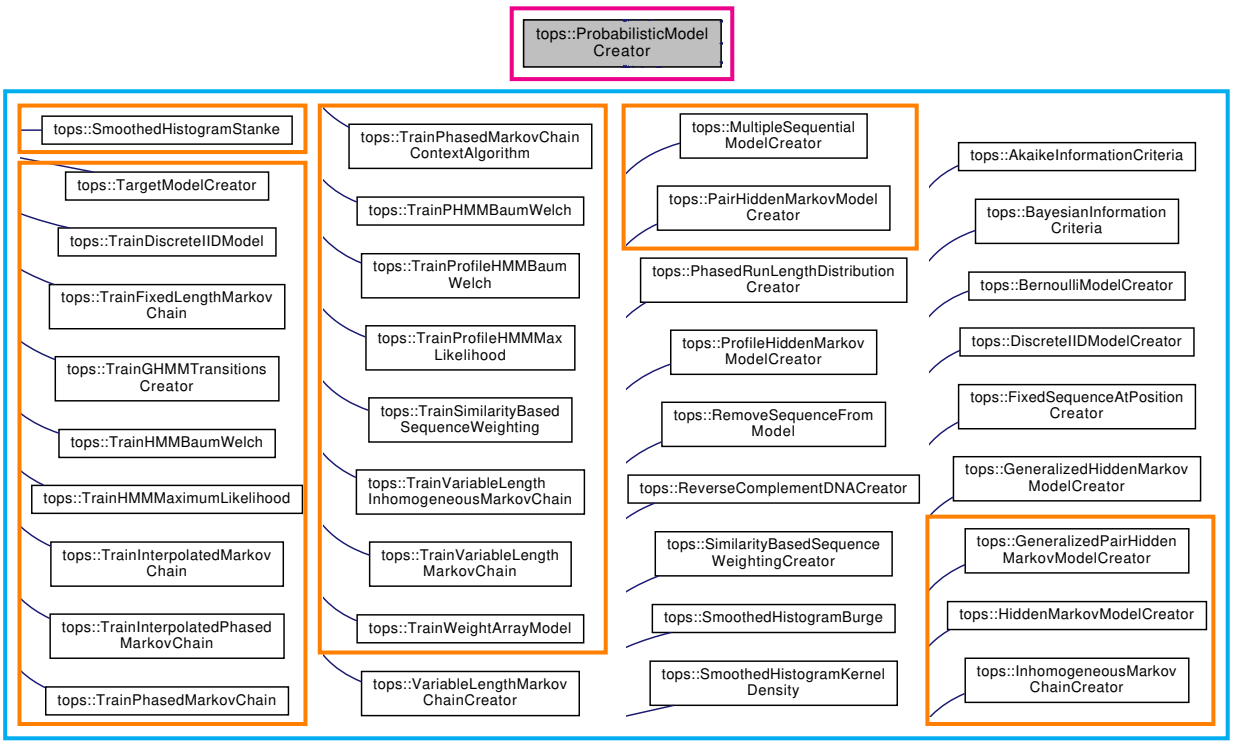



**Diagrama 3.5:** ***Maus cheiros de design na hierarquia fábricas de Modelos Probabilísticos:*** *As fábricas de modelos probabilísticos apresentam 3 maus cheiros:* ***Abstração multifacetada*** ■*, que indica que a abstração da classe possui múltiplas responsabilidades;* ***Hierarquia rebelde*** ■*, que indica a existência de métodos que deveriam pertencer a outras abstrações; e* ***Abstração imperativa*** ■*, que corresponde a operações que foram transformadas em classes.*

Por fim, a interface `tops::ProbabilisticModel` possui métodos para registrar e converter o **alfabeto** das sequências usadas pelos modelos probabilísticos. Esse alfabeto, definido nos arquivos de configuração dos modelos, é um elemento característico da linguagem do ToPS. Desse modo, a interface principal da hierarquia de modelos probabilísticos é um exemplo de **abstração multifacetada** (Diagrama 2.1): uma abstração que possui mais de uma responsabilidade (lidar com elementos da linguagem e dos modelos).

Os maus cheiros presentes nessa hierarquia estão resumidos no Diagrama 3.4.

- **Hierarquia de fábricas de modelos probabilísticos**

  Ao analisar a interface principal da hierarquia de fábricas de modelos probabilísticos (Diagrama 3.3), nota-se de imediato que os métodos dessa interface apresentam um acoplamento entre a linguagem do ToPS e os modelos probabilísticos. Esse acomplamento é natural, dada a modelagem escolhida para gerar os modelos dentro do arcabouço. Entretanto, isso faz com que a interface `tops::ProbabilisticModelCreator` torna-se uma **abstração multifacetada** (Diagrama 2.1), pois ela é responsável por validar os parâmetros passados na linguagem de especificação e por executar os algoritmos que geram modelos probabilísticos.

  Com relação às classes concretas da hierarquia, é possível notar que muitas delas não sobrescrevem todos os métodos presentes na interface. Isso faz com que elas herdem uma implementação que lança uma exceção de "método não implementado". Tal comportamento caracteriza um caso típico de **hierarquia rebelde** (Diagrama 2.1), em que um tipo rejeita os métodos do seu supertipo. Esse mau cheiro é notável por sermos capazes de separar as classes concretas

da hierarquia em 3 categorias:

1. Classes para geração de modelos com parâmetros pré-definidos (ex: `tops::HiddenMarkovModelCreator`);
2. Classes que realizam treinamento (ex: `tops::TrainHMMBaumWelch`);
3. Classes que selecionam o melhor modelo segundo um critério de informação (ex: `tops::AkaikeInformationCriteria`).

As classes do Item 2, em especial, são uma forma de **abstração imperativa** (Diagrama 2.1), pois os algoritmos de treinamento dos modelos probabilísticos foram transformados em classes, com pouco ou nenhum ganho para o design do sistema.

Os maus cheiros presentes nessa hierarquia estão sumarizados no Diagrama 3.5.

- **Hierarquia de nós da AST**

  A árvore sintática abstrata do ToPS possui uma estrutura simples, condizente com a maneira tradicional de construção de ASTs (Appel, 2004), que possui uma única superclasse e diversas subclasses que representam as expressões e comandos da linguagem abstrata. Como cada classe representa um único elemento ou conjunto de elementos da linguagem do ToPS, a hierarquia não apresenta duplicações ou maus cheiros visíveis. Logo, ela só precisará ser alterada caso a linguagem do ToPS (ou a forma como ela é processada) também seja mudada.

## 3.3 Componentes

A separação das classes do ToPS em torno de hierarquias revela que o sistema possui duas grandes componentes lógicas: a linguagem de especificação de modelos e a biblioteca de modelos probabilísticos. Na linguagem, definem-se os parâmetros que influenciam a criação dos modelos, com quais tipos de dados eles utilizarão. Os modelos já existentes, por sua vez, podem ser representados novamente pela linguagem, permitindo que eles sejam lidos e usados posteriormente sem serem retreinados.

### Melhorias

Atualmente, existem diversas linguagens de marcação (JSON[8], XML[9], YAML[10], etc.) que passaram a ser utilizadas para transferência e descrição de dados em vários tipos de aplicações (web, desktop, etc.). Paralelamente, as linguagens de script (como Perl[11], Python[12], Ruby[13], R[14], dentre outras) tornam-se cada vez mais populares, pois possuem atributos diferentes das linguagens compiladas que facilitam a programação (tipagem dinâmica, reflexão, etc). Ao tornar os modelos probabilísticos independentes da sua especificação, será possível usar o ToPS com qualquer linguagem (de marcação ou programação) compatível com o C++, facilitando a adoção do arcabouço como opção para criação de outros tipos de programas.

[8] **JSON** n.d.
[9] **Extensible Markup Language (XML)** n.d.
[10] **The Official YAML Web Site** n.d.
[11] **Perl Programming Language** n.d.
[12] **Python Programming Language** n.d.
[13] **Ruby Programming Language** n.d.
[14] **R: The R Project for Statistical Computing** n.d.

## 3.4 Repositório

O código-fonte do ToPS, em sua publicação na revista PLOS Computational Biology (Kashiwabara et al., 2013), foi disponibilizado no site SourceForge.net: repositório online de código utilizado por vários projetos de software livre bem sucedidos (como o reprodutor de mídias VLC e a biblioteca Java de mapeamento objeto-relacional Hibernate). O SourceForge foi um dos pioneiros em armazenamento de código online, oferecendo suporte para projetos com sistema de controle de versão centralizados (Apache Subversion[15]) e posteriormente estendendo seus serviços para sistemas de controle de versão distribuídos (Git[16] e Mercurial[17]). Paralelamente, o site também funciona como provedor de domínios, cedendo espaço para que os projetos armazenem suas próprias páginas web estáticas. Esse recurso é usado pelo ToPS (e por muitos outros programas) para manter informações de contato, tutoriais e documentações acessíveis online. As páginas também ajudam a popularizar as ferramentas, permitindo que elas sejam encontradas por mecanismos de busca. O site do ToPS e suas informações podem ser acessadas em http://tops.sourceforge.net/.

### Melhorias

Embora o SourceForge ainda seja muito utilizado, outros serviços ganharam mais popularidade com o tempo. O GitHub, em especial, foi um dos primeiros a oferecer armazenamento de código para projetos que usam Git[18], ganhando um número crescente de usuários conforme o programa `git` tornou-se quase padrão para controle de versão de novos projetos de software livre. Com a recente desativação do Google Code, ainda mais projetos estão migrando para o GitHub. Toda essa visibilidade somada às ferramentas oferecidas pelo site tornam o GitHub o local mais atrativo projetos que usam Git, inclusive o ToPS.

No que diz respeito ao processo de desenvolvimento, o fato do ToPS ter crescido pelas contribuições individuais de trabalhos de mestrado e doutorado fizeram com que as adições feitas não tivessem um processo bem definido. Sem a imposição de padrões claros de qualidade (testes, revisões, tolerância a falhas, etc.) a serem seguidos antes de incluir mudanças na versão de produção do ToPS, acelerou-se a degradação do código do arcabouço. Assim, criaram-se a maioria dos problemas que podem ser vistos nesse capítulo, espalhados entre as diferentes camadas do sistema. Impor novas regras para contribuições, seguindo práticas como as propostas em **Extreme Programming Explained: Embrace Change** (Beck, 1999) pode ajudar a manter a qualidade do arcabouço conforme ele for expandido e modificado ao longo do tempo.

[15] **Apache Subversion** n.d.
[16] **Git** n.d.
[17] **Mercurial** n.d.
[18] **Git** n.d.

# Capítulo 4

# Refatoração

No Capítulo 3, discutimos os detalhes de implementação do ToPS, identificando pontos de melhoria em quatro camadas do sistema: código, arquitetura, componentes e repositório, No Capítulo 2, definimos o conceito de refatoração, analisando os benefícios de utilizá-la para preservar a qualidade do código de um programa. Neste capítulo, descreveremos uma série de refatorações aplicadas no ToPS, tomando por base as ideias construídas nos dois capítulos anteriores. As modificações apresentadas nesta parte do texto têm como objetivo corrigir os problemas já detectados no arcabouço. Por fim, no Capítulo 5, analisaremos o impacto dessas mudanças na capacidade de extensão, modificação e compreensão do ToPS - que consiste na principal contribuição deste trabalho.

Para facilitar o entendimento das modificações feitas durante a refatoração, utilizaremos novamente a divisão em categorias proposta no Capítulo 3. Dessa vez, porém, optamos por tratar das camadas do sistema segundo a *complexidade das alterações*. Começaremos, assim, pelo repositório (Seção 4.2), tratando do novo destino remoto para armazenamento das contribuições e do processo adotado para submetê-las. Em seguida, falaremos do código (Seção 4.3), com as correções e padronizações feitas sobre ele. Na parte estrutural, trataremos das componentes do arcabouço (Seção 4.4) e de como propusemos diminuir as dependências entre elas. Finalmente, entraremos na arquitetura do sistema (Seção 4.5), e em como buscamos corrigir os maus cheiros encontrados nas hierarquias de classes. Essa ordenação, além de explicar as modificações de forma incremental, também serve de boa aproximação para a ordem cronológica da realização das refatorações. Antes de entrarmos a fundo nas mudanças feitas no ToPS, dedicaremos um pequeno espaço para discutir nossa metodologia de trabalho (Seção 4.1): a abordagem utilizada para refatorarmos o arcabouço. As escolhas mostradas nessa seção, feitas antes de alterarmos as primeiras linhas de código, influenciaram diretamente as decisões tomamos no decorrer da refatoração.

Como trataremos de duas versões do ToPS ao longo deste capítulo, utilizaremos os termos "ToPS 1.0", "ToPS original", "versão publicada do ToPS" e variantes para nos referirmos à versão do ToPS publicada na revista PLOS Computational Biology (Kashiwabara et al., 2013) e discutida no Capítulo 3. Usaremos as expressões "ToPS 2.0", "novo ToPS", "versão refatorada do ToPS" e similares para a versão produzida neste trabalho.

## 4.1 Metodologia

Tradicionalmente, refatorar um sistema implica em modificar diretamente o código já existente. Ao analisar o ToPS, porém, podemos ver duas características que desestimulam essa abordagem:

- **Falta de testes**
  Para garantir que as mudanças feitas no código de um programa não alterem o seu comportamento, é essencial termos algum método prático que nos dê evidências de que esse comportamento se mantém. Os **testes de unidade**, nesse contexto, são um instrumento poderoso para checar se trechos pequenos de código (chamados de *unidades*, geralmente na escala de funções ou métodos) estão produzindo os resultados esperados para diferentes cenários de entradas. Por serem automatizados, é fácil rodar toda a bateria de testes para cada pequena modificação, assegurando um avanço incremental sem que comportamento testado se altere. Por conta disso, os testes de unidade são essenciais para se fazer uma boa refatoração (Fowler e Beck, 1999). Na versão publicada do ToPS, porém, não há nenhum tipo de teste, o que faz com que não tenhamos uma maneira de certificar que os algoritmos dos modelos probabilísticos manterão os seus resultados. Para começarmos a modificar o arcabouço, precisamos implementar alguns testes para esses algoritmos, do modo a ganharmos segurança para alterá-los.

- **Design modular**
  O design de organização dos modelos probabilísticos do ToPS - expresso na arquitetura do sistema por meio de uma *Composite* (Seção 3.2) - é um dos grandes diferenciais do arcabouço em termos de reúso de código. Essa modularização, inerente a esse padrão de design, oferece uma grande oportunidade ao alterarmos o sistema: ao invés de modificar muitos modelos ao mesmo tempo, podemos desconectar alguns deles da hierarquia de classes e deixá-los para serem refatorados posteriormente - sem prejudicar o funcionamento do restante do arcabouço. Assim, não precisamos implementar muitos testes antes de começar a refatoração, e com menos modelos fica mais simples experimentarmos novas opções de arquitetura no arcabouço.

Tendo em vista essas duas particularidades do ToPS, decidimos fazer a refatoração seguindo uma metodologia **incremental**. Ao invés de alterarmos o sistema inteiro - o que tomaria muito mais tempo e reduziria a nossa flexibilidade de experimentação - restringimos nossas modificações para um subconjuntos dos modelos probabilísticos implementados no arcabouço. Preferimos, então, focar inicialmente nos modelos utilizados pelo MYOP para predição de genes (Tabela 4.1). Com esse recorte, pudemos receber a ajuda do doutorando Ígor Bonadio, que implementou as primeiras baterias de testes de unidade usando o arcabouço Google Test[1]. Esses testes foram, posteriormente, adicionados no novo repositório do ToPS, conforme descrito na Seção 4.2.

| Modelos refatorados | Sigla |
|---|---|
| Processo Independente e Identicamente Distribuído | IID |
| Cadeia de Markov de Alcance Variável | VMLC |
| Cadeia de Markov Não Homogênea | IMC |
| Modelo Oculto de Markov | HMM |
| Modelo Oculto de Markov Generalizado | GHMM |
| Ponderamento Baseado na Similaridade de Sequências | SBSW |

**Tabela 4.1:** ***Modelos refatorados****. Para reduzir o tempo necessário para aplicar refatorações no sistema, e assim facilitar experimentos no design do código, escolhemos refatorar neste trabalho apenas um subconjunto dos modelos já implementados no ToPS,*

[1] **Google Test** n.d.

## 4.2 Repositório

Para começar a refatoração do ToPS, resolvemos configurar, antes de tudo, o novo repositório remoto para o qual enviaríamos as modificações feitas no código do sistema. Levando em conta a popularidade do GitHub frente ao SourceForge (conforme discutido na Seção 3.4), resolvemos utilizar os recursos administrativos disponíveis no site para organizarmos melhor nosso novo local de contribuição. Criamos, então, a organização ToPS, na qual ficariam todos os projetos diretamente relacionados ao arcabouço. Ao criar uma organização, passamos a não depender mais da conta individual de nenhum dos desenvolvedores, permitindo que todos os envolvidos adicionem novos contribuidores aos projetos, criem repositórios privados para armazenar código não publicado e aceitem *pull requests*[2].

O primeiro repositório da organização, chamado tops-refactoring, foi criado como um *fork*[3] do repositório original do ToPS, de modo a preservar o histórico de *commits* anterior à refatoração. Em vez de utilizarmos o código existente, porém, seguimos a ideia de fazer alterações incrementais (descrita na Seção 4.1), criando um primeiro *commit* que não continha nenhum código. A partir de então, dedicamos nossos esforços a portar o conteúdo da versão antiga para a nova, começando pelos arquivos de configuração e documentação. Antes de começarmos a mexer com o código em si, aproveitamos o repositório vazio para introduzir um novo conjunto de regras para contribuições. As diretrizes, disponíveis no arquivo `CONTRIBUTING.md` do repositório, padroniza duas formas de contribuição ao projeto:

- **Novas funcionalidades e correções**
  Para contribuir, o desenvolvedor que não faz parte do projeto deve usar um sistema de *forks* para criar sua própria versão do repositório. Nele, deverá fazer *commits* que adicionam a nova funcionalidade ou correção em uma *branch* cujo nome especifique o que está sendo implementado. Adicionar testes de unidade para o novo código é extremamente recomendável, seguindo o conjunto de práticas da metodologia de Programação Extrema (Beck, 1999). Para os desenvolvedores que fazem parte do projeto, apenas a etapa de criar um *fork* pode ser omitida, e a *branch* pode ser feita diretamente no repositório do projeto. Ao concluir sua implementação, o contribuidor deve fazer um *pull request*, que só será aceito após a revisão de algum membro do projeto (que não seja ele mesmo). Esse sistema de **revisão em pares** visa manter a qualidade do código e verificar se o contribuidor seguiu todas as recomendações do projeto, ajudando a manter o código do ToPS sempre com as características desejadas.

- **Triagem de erros**
  Para reportar erros e defeitos no arcabouço, o contribuidor deve criar uma *issue* com o rótulo *bug* utilizando o *issue tracker* do GitHub. Esse mesmo mecanismo pode ser usado para propor funcionalidades, refatorações e melhorias. exigindo-se apenas que a *issue* seja classificada em uma das categorias pré-definidas pelos desenvolvedores do ToPS. Neste trabalho, utilizamos esse sistema para marcar tarefas relativas à refatoração. Para facilitar a visualização dessas *issues*, utilizamos uma integração com o GitHub chamada waffle.io, que as representa na forma de um *kanban*[4]. As *issues* podem ser submetidas de forma equivalente por ambos os sites.

[2] Mecanismo oficial do GitHub para contribuições de usuários não-membros de um projeto.
[3] Vide nota 1
[4] Quadro para anotação e acompanhamento de tarefas, tipicamente usado em metodologias ágeis

Com essas diretrizes, conseguimos especificar qual a abordagem correta para modificar o ToPS. A revisão em pares, em particular, é um instrumento muito útil para reforçar a verificação de código, e serve como método educacional para novos desenvolvedores. Apesar dessas qualidades, tal revisão possui natureza não automatizada, podendo ser esquecida e, nesse caso, permitir a introdução de erros que quebrem o sistema. Para combater esse problema, foi criado o conceito de **integração contínua**, que requer que a cada *commit* no repositório, toda a aplicação seja compilada e a bateria de testes, executada (Humble e Farley, 2010). Caso não haja erros, a nova versão já pode ser disponibilizada para uso, com a garantia de que funcionará tanto quanto a anterior.

Para nos aproveitarmos dos benefícios da integração contínua no ToPS, recorremos a dois serviços que se integram com o GitHub, e que são gratuitos para projetos de software livre:

- **Travis CI**
  O site Travis permite fazer todo o processo de integração contínua usando apenas o arquivo de configuração `.travis.yml`. Especificando apenas alguns comandos de shell, o Travis monta uma máquina virtual com Ubuntu ou OSX, instala todas as dependências necessárias para a compilação, e roda um programa que faz verificações sobre o código. No caso do ToPS, o Travis executa toda a bateria de testes de unidade, utilizando os compiladores gcc e clang, uma vez em cada sistema operacional. Além de rodar nos *commits* submetidos ao repositório, o Travis também faz verificações em *pull requests*, avisando pelo GitHub caso os testes não passem. Com esse sistema, temos uma forma de ajudar o revisto do *pull request*, facilitando que ele verifique que as novas modificações não causarão problemas no sistema.

- **Coveralls**
  Além de rodar os testes, outra informação relevante para a manutenção de um sistema é a **cobertura**: percentual de linhas de código que são exercitadas pelos testes de unidade. Essa estatística é relevante pois pode revelar modificações muito grandes que não foram testadas - algo que, de forma geral, é indesejável. O site Coveralls fornece uma maneira automática de coletar essas estatísticas a partir do Travis, e pode impedir que a execução seja bem sucedida caso a cobertura caia abaixo de um certo limiar. No ToPS, decidimos por não usar o recurso de falha, pois como o sistema não tinha testes, a cobertura inicial era baixa. Ainda assim, configuramos essa integração contínua, de modo a ajudar os revisores de *pull request* a analisarem a testabilidade do código submetido para o repositório.

Uma vez configurado todo o ambiente e definido o processo de contribuição, pudemos finalmente nos concentrar nos problemas presentes no código, descritos em detalhes na próxima seção.

## 4.3 Código

Para atingir as melhorias do código propostas na Seção 3.1, decidimos realizar alterações no ToPS em três frentes:

1. Substituição do CMake por outra ferramenta para compilação;
2. Remoção de todos os *warnings* de compilação;
3. Padronização da formatação dos arquivos de código do ToPS.

Para cumprir com o Item 1, passamos a utilizar o All-in-One Makefile[5] (desenvolvido pelo autor deste trabalho como projeto paralelo de software livre) para realizar a compilação e gestão de dependências do ToPS. O AIO Makefile fornece, de forma praticamente automática, diversas funcionalidades úteis para a compilação do arcabouço, tais como a geração de múltiplos executáveis, a produção de bibliotecas dinâmicas e a compilação seguida de execução de testes de unidade. Por depender apenas do GNU Make - programa padrão na maioria dos sistemas Linux e facilmente instalável nos baseados em BSD (como o OSX da Apple) - o AIO Makefile oferece uma boa alternativa ao CMake, facilitando a compilação e instalação do ToPS em quase qualquer plataforma.

Para resolver o problema do Item 2, corrigimos os *warnings* em todos os trechos de código apontados pelos compiladores gcc e clang. O *warning* mais frequente - de comparação entre tipos inteiros com e sem sinal - foi facilmente resolvido (na maioria dos casos) com a troca da variável com sinal (do tipo `int`) por uma sem sinal (do tipo `unsigned int`). Para estender ainda mais a política de remoção de *warnings*, aumentamos o nível de avisos de ambos os compiladores para o máximo (usando as *flags* `-Wall -Wextra -Wpedantic`) e adicionamos uma *flag* extra (`-Werror`) para indicar que todos os *warnings* devem ser tratados como erros. Essa prática, aliada com a presença de um sistema de integração contínua, é útil para melhorar a confiabilidade do código do ToPS, forçando os programadores a seguirem todas as recomendações feitas pelos compiladores.

Para efetivar o Item 3, resolvemos seguir o padrão do *Google C++ Style Guide*[6], que oferece o programa python `cpplint` para fazer a **análise estática do código** a ser verificado. Considerando apenas as informações disponíveis em tempo de compilação, o *script* lista todos os locais que não estão seguindo o estilo recomendado. Por usarmos o AIO Makefile, pudemos nos beneficiar da integração nativa entre as duas ferramentas: com apenas um comando, é possível rodar o `cpplint` para todos os executáveis e bibliotecas do projeto. Como o AIO Makefile também cria arquivos C++ seguindo o estilo da Google, fica ainda mais fácil incentivar os contribuidores a utilizarem esse padrão. Com a formatação unificada, podemos por fim colocar em prática a posse coletiva de código, em que todos os desenvolvedores podem modificar qualquer parte do sistema sem se preocupar com quem originalmente a implementou (Beck, 1999).

Dentre as propostas de melhoria indicadas na Seção 3.1, apenas uma não foi citada na lista acima: a substituição da biblioteca Boost. Esse trabalho foi realizado pelo aluno de doutorado Ígor Bonadio, que, ao portar os modelos probabilísticos do repositório antigo para o novo (como descrito na Seção 4.2). trocou as implementações da Boost pelas alternativas[7] da biblioteca padrão de *templates* (STL) do C++11. Só por retirarmos essa dependência, o tempo de compilação necessário para gerar os modelos caiu drasticamente. Utilizando um processador Intel® Core™ i7-4510U CPU de 2.00GHz, na versão publicada do ToPS (com a Boost) leva de cerca de 3,5 min para compilar os modelos listados da Tabela 4.1. Na versão refatorada (usando C++11/14), esse tempo caiu para aproximadamente 1 min - 71,4% a menos que o original. Ambos os testes foram realizados com o nível máximo de otimização de compilação (`-O3`) ativado.

Concluídas as correções no código e definidas as ferramentas a serem utilizados no projeto, pudemos começar, finalmente, com a refatoração das classes e hierarquias do ToPS. Em toda essa etapa, tentamos seguir os padrões de contribuição estabelecidos nessas duas últimas seções, testando a solidez dos novos processos adotados pelo ToPS.

[5] **All-in-One Makefile** n.d.
[6] **Google C++ Style Guide** n.d.
[7] **C++ Reference** n.d.

## 4.4 Componentes

Para conseguirmos desacoplar as duas componentes do ToPS citadas na Seção 3.3, decidimos criar dois *namespaces* distintos para separar explicitamente o código de cada parte do arcabouço:

- `tops::lang`, para lidar com as estruturas de representação da linguagem do ToPS;
- `tops::model`, para lidar com as classes que representam os modelos probabilísticos.

Os *namespaces* são um recurso da linguagem C++ que visa reunir, num mesmo escopo, um conjunto de funcionalidades relacionadas (Stroustrup, 2013). São tipicamente usados para criar **módulos reutilizáveis**, que podem ser pré-compilados na forma de **bibliotecas** estáticas ou dinâmicas. Essas bibliotecas podem ser, então, distribuídas e instaladas, sendo posteriormente ligadas aos programas que as utilizam na última fase da compilação (Appel, 2004). Para nos aproveitarmos dessas vantagens em termos de organização de código e distribuição, decidimos criar uma biblioteca para cada componente do ToPS, usando o AIO Makefile para gerá-las e, posteriormente, ligá-las nos aplicativos do sistema. Dessa maneira, o ToPS fornecerá a suas aplicações duas alternativas de utilização: uma API (*Application Programming Interface*) que pode usada dentro do código-fonte; e programas de shell, que podem ser acessados via chamadas de sistema.

Dentre as melhorias propostas na Seção 3.3, desejávamos, ainda, fazer com que as classes dos modelos probabilísticos não tivessem conhecimento das estruturas da linguagem de especificação. Para cumprir com esse objetivo, o aluno Ígor Bonadio, ao portar o código do repositório antigo o novo (Seção 4.2), alterou os algoritmos dos modelos para que usem como alfabeto de sequências apenas números inteiros não negativos ($\mathbb{Z}^+$). Embora essa mudança pareça uma limitação, deixamos para que a componente da linguagem mapeie o alfabeto especificado pelo usuário para o alfabeto padrão - o que pode ser feito trivialmente dado que o ToPS lida apenas com alfabetos finitos. O restante das adaptações feitas nos modelos está descrita na Seção 4.5.

Infelizmente, devido à complexidade das modificações feitas na biblioteca de modelos probabilísticos, não houve tempo hábil para portar as classes que representariam (internamente) a linguagem de especificação. Nossa ideia original era criar uma hierarquia de `structs` que determinariam os parâmetros de configuração e treinamento dos modelos. Essas *structs* seriam, então, preenchidas pelo interpretador da linguagem, que contaria com um *parser* (gerado automaticamente em C++ com ferramentas como o FlexC++[8] e o BisonC++[9]) para ler os arquivos de configuração. A ideia de reimplementar a linguagem diretamente, porém, foi descartada. Considerando que o ToPS será estendido (num futuro próximo) com dois modelos de Campos Aleatórios Condicionais (Tabela 1.1b), a linguagem do ToPS também precisará dar suporte à criação de **funções** (que fazem parte da especificação desses modelos). Entretanto, a linguagem não possui, atualmente, essa funcionalidade, que não é simples de implementar. Buscando achar uma solução que já atenda a esse novo requisito, começamos a fazer experimentos com a linguagem ChaiScript[10], que possui sintaxe parecida com a antiga linguagem do ToPS e que foi criada para ser integrada a programas C++. Embora nossos protótipos sejam promissores, ainda precisamos decidir alguns detalhes da ligação entre o interpretador do ChaiScript e o ToPS. Sendo assim, o restante da implementação da linguagem do ToPS ficará como trabalho futuro deste projeto (Capítulo 6).

[8] **FlexC++** n.d.
[9] **BisonC++** n.d.
[10] **ChaiScript - Easy to use scripting for C++.** N.d.

## 4.5 Arquitetura

Para decidir quais modificações seriam feitas na arquitetura da biblioteca de modelos probabilísticos, definimos duas metas a serem alcançadas durante esta etapa da refatoração:

- Melhorar a **usabilidade** das classes da biblioteca, criando uma API mais simples para acesso aos modelos probabilísticos;
- Melhorar a **extensibilidade** das hierarquias, facilitando a criação de novos modelos e a adição de novas funcionalidades nos modelos já existentes.

Como primeira medida tomada para alcançar esses objetivos, buscamos terminar de desacoplar a biblioteca de modelos da linguagem do ToPS. Como citado na Seção 4.4, os algoritmos existentes nos modelos foram alterados para considerarem um único tipo de alfabeto, possibilitando a remoção dos métodos que administravam esses símbolos. Sem essa responsabilidade extra na classe abstrata `tops::ProbabilisticModel`, eliminou-se o mau cheiro de **abstração multifacetada**.

Embora essa padronização tenha corrigido o problema na hierarquia de modelos (Diagrama 3.4), o mesmo mau cheiro pode ser encontrado, de forma mais explícita, na hierarquia de fábricas (Diagrama 3.5), cujos métodos herdados da classe abstrata `tops::ProbabilisticModelCreator` validam os parâmetros passados pela linguagem de especificação. Em vez de tentarmos arrumar esse defeito, porém, optamos por remover a fonte do problema, eliminando a própria hierarquia de fábricas. Como discutido na Seção 3.2, esse conjunto de classes apresentava mais dois maus cheiros: formava uma **hierarquia rebelde**, e possuía várias **abstrações imperativas**. Somados à existência do acoplamento, ficava claro que essa hierarquia não era a melhor solução para criar modelos, ainda que fosse uma implementação do padrão Fábrica Abstrata (Gamma et al., 1995).

Para as classes da hierarquia de fábricas que criavam algum modelo (explicitamente) a partir dos parâmetros dos seus métodos, criamos métodos estáticos `make` dentro das classes dos modelos correspondentes. Para as classes que faziam treinamento, agimos de forma similar, movendo os seus algoritmos para métodos estáticos cujos nomes começavam com o prefixo `train`. Os critérios de seleção não foram refatorados nesse momento, sendo deixados para uma reavalização durante a reimplementação dos aplicativos do ToPS.

Com a eliminação de uma das hierarquias do sistema, tornou-se mais fácil testar outras possibilidades de reúso de código entre os modelos. Na hierarquia de modelos probabilísticos (agora, a principal do sistema), ainda restavam dois maus cheiros de design: uma sub-**hierarquia desnecessária**, e vários casos de **hierarquia não fatorada**. Para resolver o primeiro mau cheiro, resolvemos remover as classes abstratas `tops::FactorableModel` e `tops::InhomogeneusFactorableModel`, fazendo com que todos os modelos que herdavam delas passassem a herdar da nova e atualizada interface principal da hierarquia: `tops::model::ProbabilisticModel`. Essa modificação também foi suficiente para corrigirmos a interface não fatorada presente entre essas três classes - um dos casos do segundo mau cheiro da hierarquia. A implementação não fatorada presente entre as classes `tops::HiddenMarkovModel` e `tops::GeneralizedHiddenMarkovModel` foi resolvida movendo os trechos de código em comum para a nova superclasse abstrata da qual ambos passaram a herdar: `tops::model::DecodableModel`. Uma última classe abstrata também foi criada para os decoradores de modelos, com o nome `tops::model::ProbabilisticModelDecorator`. A hierarquia resultante pode ser vista no Diagrama 4.1.

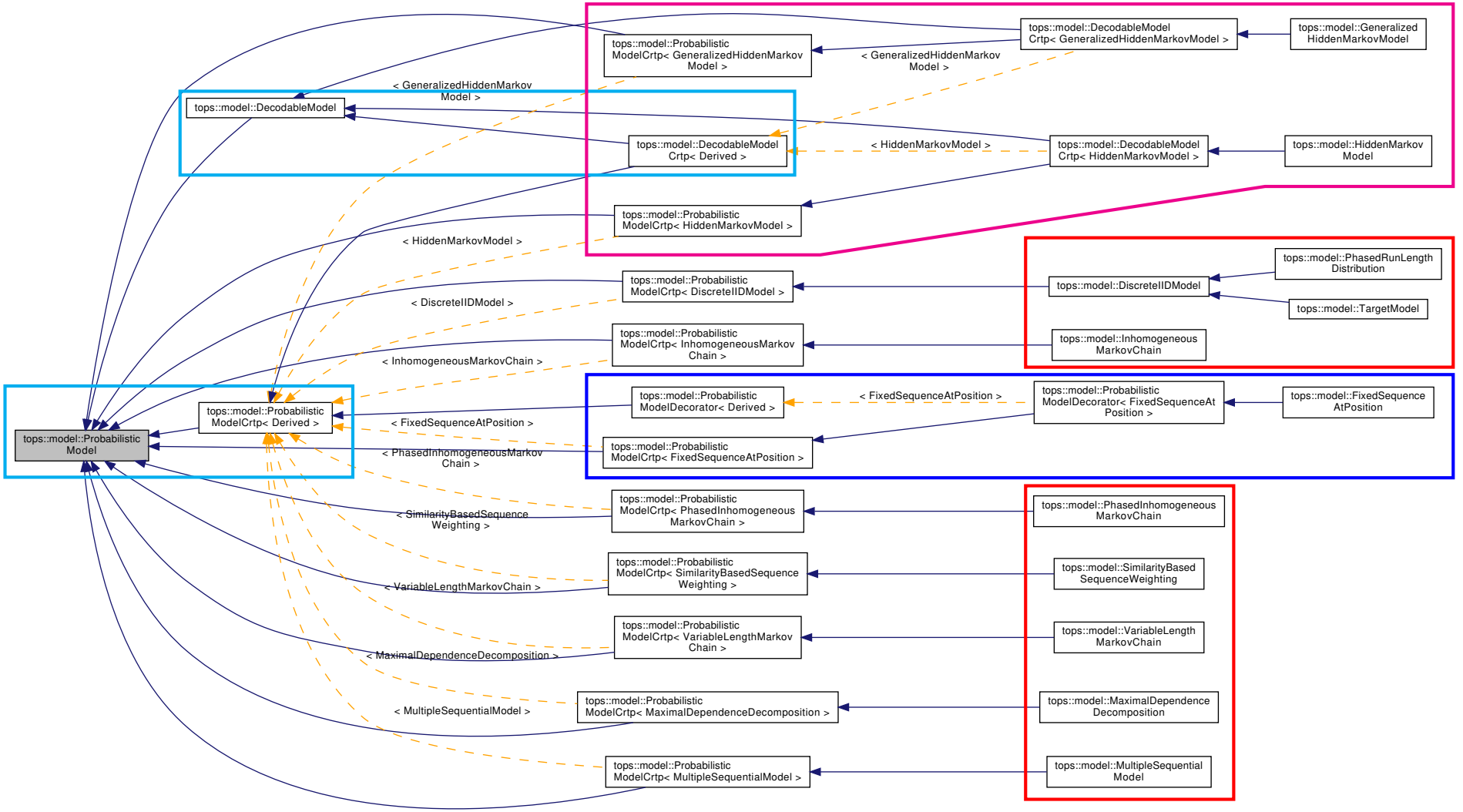



**Diagrama 4.1:** ***Refatoração da hierarquia de Modelos Probabilísticos:*** *Todos os* ***modelos probabilísticos*** ■ *herdam da classe abstrata* `tops::model::ProbabilisticModel`. ***Modelos decodificáveis*** ■ *e* ***decoradores de modelos*** ■ *possuem suas próprias interfaces, As principais abstrações da hierarquia são implementadas utilizando o* ***Curiously Recurring Template Pattern*** ■ *- idioma do C++ que permite reutilizar a implementação de métodos similares (como os métodos fábrica dos front-ends) por meio de templates.*

Dentre os maus cheiros citados anteriormente, não apresentamos a refatoração para o caso de interface e implementação não fatoradas. Conforme descrito na Seção 4.1, apenas os modelos presentes na Tabela 4.1 foram selecionados para serem refatorados. Como esse caso do mau cheiro envolvia as classes `tops::PairDecodableModel`, `tops::PairHiddenMarkovModel` e `tops::GeneralizedHiddenMarkovModel`, essa modelagem não foi considerada.

### 4.5.1 Arquitetura de *front-ends*

Paralelamente aos nossos esforços para corrigir os maus cheiros presentes na arquitetura do ToPS, buscamos resolver outros tipos de problemas encontrados na hierarquia de modelos probabilísticos. Um dos diferenciais do sistema em termos de otimização são os **métodos com cache**: funções de classe que reutilizam o resultado de algoritmos custosos armazenando-os em vetores de memória para acesso rápido. Em parte das funcionalidades presentes nos modelos, podemos encontrar uma lógica parecida, inserida em **algoritmos de programação dinâmica** que salvam, em matrizes, valores parciais usados em seus cálculos. Na modelagem orientada a objetos utilizada pela versão publicada do ToPS, esses vetores e matrizes são armazenados como atributos das classes dos modelos, o que impossibilita o seu reúso caso haja uma execução com entrada diferente da que gerou os dados armazenados. Essa modelagem também impede que os algoritmos sejam executados de forma concorrente (com múltiplas *threads* em ambiente de memória compartilhada), por conta das condições de corrida que poderiam ser geradas no acesso às estruturas de dados.

Uma forma natural de resolver o problema dos caches seria utilizar objetos intermediários para armazenar os resultados produzidos para cada entrada ou *thread*. Essa abordagem, embora seja interessante por sua simplicidade, exigiria a criação de parâmetros extras em todos os métodos com suporte à execução memorizada. Além disso, também facilitaria a ocorrência de erros, pois deixa para o cliente a tarefa de alocar, preservar e liberar uma estrutura de dados de uso interno (que sequer deveria ser visível ao usuário).

Para permitir um design flexível que lide com os caches, e ainda fornecer uma administração de recursos de forma automática e encapsulada, desenvolvemos ao longo deste trabalho a **arquitetura de *front-ends***: padrão que se propõe a estender a ideia simplista do parágrafo anterior e uniformizar a forma como métodos com e sem cache são acessados nas classes dos modelos probabilísticos. Segundo nossa proposta, podemos dividir os objetos presentes na nossa arquitetura em dois conjuntos, segundo suas propriedades:

- ***Front-ends***, que possuem um ou mais métodos semanticamente relacionados que podem ser executados no *back-end*. Um objeto *front-end* não possui nenhum algoritmo complexo internamente: apenas delega a execução para um dado objeto *back-end*, passando a si próprio como primeiro parâmetro da chamada de método (e reencaminhando o restante dos parâmetros recebidos na sua própria chamada). O *front-end* pode possuir, dentro de si, algum tipo de cache definido pela classe do *back-end*. Caso o método executado use esse cache, o algoritmo presente no *back-end* poderá acessá-lo via um *getter* no *front-end*, sendo capaz de lê-lo e alterá-lo. O *front-end* pode ser criado diretamente como uma classe concreta ou ser definido genericamente por uma interface, cujas implementações concretas podem apresentar pequenas variações entre si (por exemplo, ter ou não cache). Nesse caso, o *back-end* fica responsável por selecionar qual o método a ser rodado.

- ***Back-ends***, que implementam algoritmos relacionados à lógica de negócio do sistema. Os métodos do objeto *back-end* devem possuir, obrigatoriamente, um primeiro parâmetro que seja uma referência ou ponteiro para uma classe concreta *front-end*. Caso dois métodos sobrecarregados (com mesmo nome) do *back-end* difiram apenas nesse primeiro, o *back-end* deve assegurar que o método correto seja executado - o que pode ser feito usando mecanismos da própria linguagem de programação, como polimorfismo.

Antes de chegarmos a essa proposta de padrão, realizamos uma série de experimentos para definir o relacionamento entre os objetos que se tornariam, por fim, os *back-ends* e *front-ends*. Essas investigações foram feitos, em particular, para testar os recursos do C++ que poderiam ser usados para implementar essa arquitetura. Considerando a complexidade do ToPS e o grande número de linhas de código que precisam ser alteradas para fazer alterações nas suas hierarquias, tentamos criar métodos que reduzissem a quantidade de código a ser alterada - tanto para a refatoração, quanto para eventuais manutenções e extensões feitas no sistema. Ao longo de nossos experimentos, construímos um protótipo para exemplificar a ideia da arquitetura de *front-ends*, que pode ser acessado em: https://github.com/topsframework/tops-architecture.

### 4.5.2 *Front-ends* do ToPS

Como parte da implementação do protótipo de arquitetura, criamos uma série de metafunções (Meyers, 2005) e macros para encapsular a lógica de delegação dos métodos dos *front-ends* para os seus respectivos *back-ends*. Por meio de metaprogramação, reduzimos essa tarefa repetitiva a poucas linhas de código, inseridas com macros nas classes dos *front-ends*. Com essa facilidade, decidimos agrupar os métodos da hierarquia de modelos em seis *front-ends*, segundo as suas funcionalidades: **avaliadores**, **geradores**, **calculadores**, **rotuladores**, **treinadores** e **serializadores**. Como cada *front-end* possui suas particularidades, decidimos discutir sobre eles individualmente. Um resumo esquemático da arquitetura de *front-ends* do ToPS pode ser visto na Diagrama 4.2.

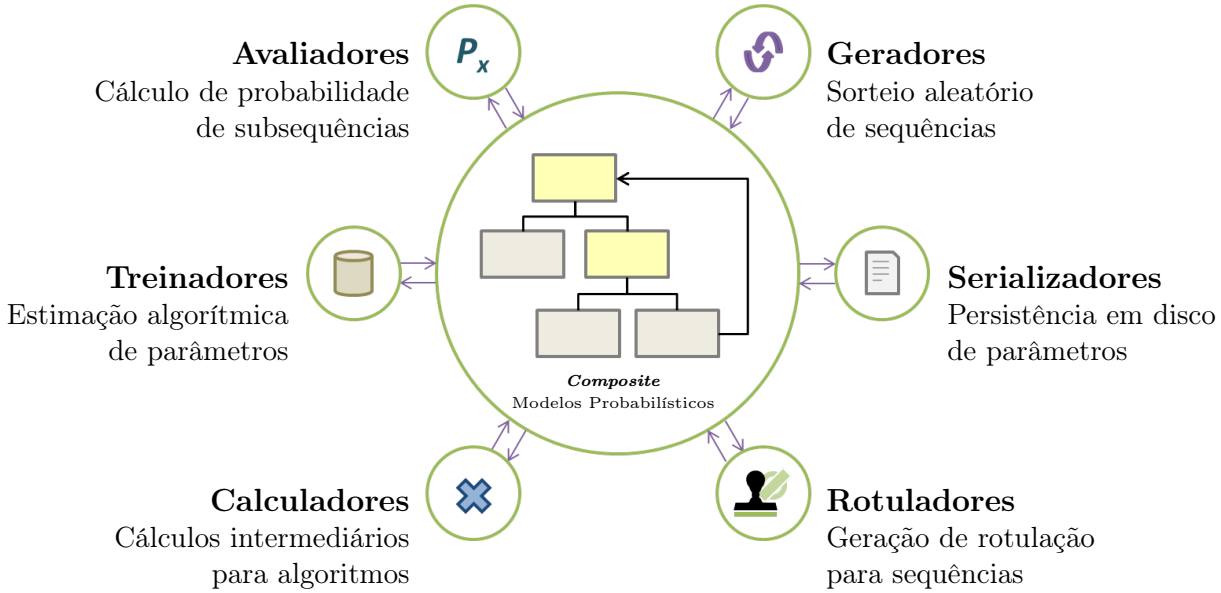



**Diagrama 4.2:** ***Arquitetura de front-ends no ToPS****: As classes da composite (Gamma et al., 1995) de modelos probabilísticos (back-ends) têm seus algoritmos acessados por meio de classes auxiliares (front-ends), que delegam a execução e armazenam dados para os modelos.*

### Avaliadores

O *front-end* de avaliadores foi o primeiro a ser implementado dentro da nova arquitetura do ToPS. Sua função consiste em calcular a probabilidade de subsequências de uma dada sequência, usando os métodos `evaluateSequence` e `evaluateSymbol` presentes nos modelos probabilísticos. Como ambos os métodos possuem duas versões - com e sem cache - definimos esse *front-end* pela interface `tops::model::Evaluator`, que possui duas implementações concretas:

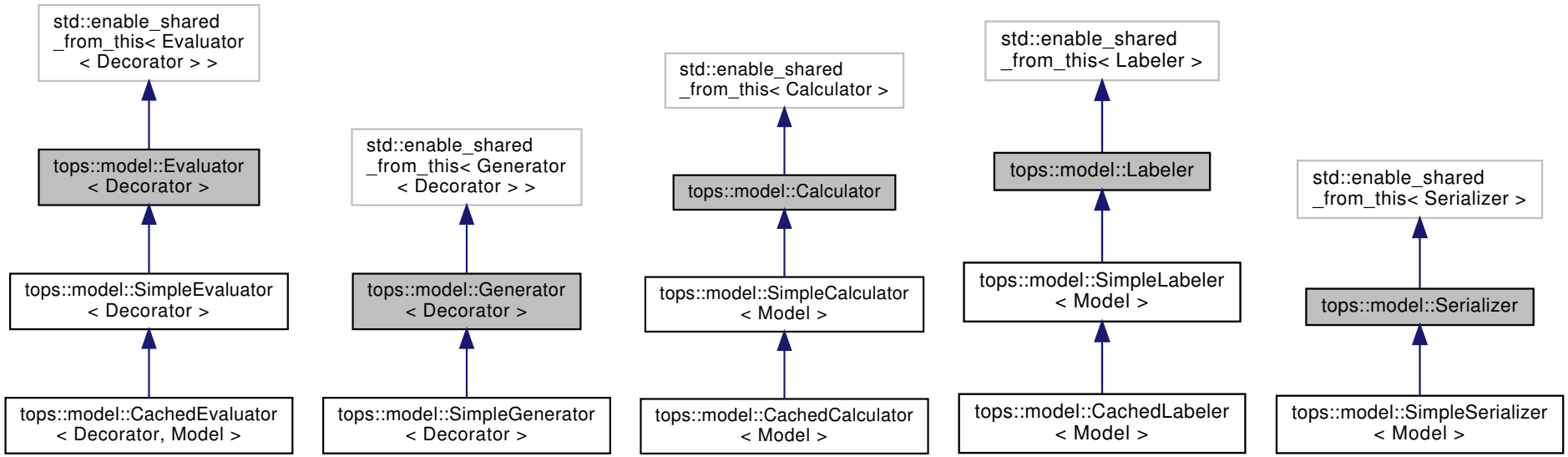
- `tops::model::SimpleEvaluator`, que faz a avaliação sem cache; e
- `tops::model::CachedEvaluator`, que faz a avaliação de *todas* as subsequências da sequência alvo na primeira chamada de um dos métodos *evaluate* - realizando uma busca em memória (com tempo constante) no restante das vezes.

Como todos os avaliadores precisam, necessariamente, de uma sequência para ser avaliada, os objetos da hierarquia de avaliadores (Diagrama 4.3a) recebem uma sequência em seus construtores. Todos os métodos executados no *front-end*, bem como o cache gerado na implementação memorizada, são relacionados à essa sequência. De forma análoga, todos os avaliadores requerem um *back-end* para o qual possam delegar os seus métodos. Por isso, também recebem no construtor um parâmetro extra: um ponteiro para modelo probabilístico.

Como desejamos saber, em tempo de compilação, qual será o tipo do cache armazenado em `tops::model::CachedEvaluator`, as classes concretas da hierarquia são parametrizadas pelo tipo do modelo (acessível na forma de um parâmetro de *templates*). Se o modelo contiver o tipo `Cache` dentro da sua classe, o avaliador com memorização será capaz de gerar o objeto apropriado. A raiz da hierarquia, por sua vez, não precisa desse *template*, pois não usa nenhuma informação do tipo de modelo - permitindo que todos avaliadores sejam usados pela mesma interface.

Considerando a duplicação existente ao instanciar um avaliador (é necessário passar a classe de um modelo como *template*, e um ponteiro para objeto dessa classe no construtor), nota-se que a construção desse *front-end* não é tão trivial quanto possível. Para garantir o avaliador seja instanciado corretamente, adicionamos um Método Fábrica (Gamma et al., 1995) na hierarquia de modelos probabilísticos. Esse método, nomeado `standardEvaluator`, é puramente virtual na interface `tops::model::ProbabilisticModel`, e recebe como parâmetro dois argumentos: a sequência a ser avaliada, e uma *flag* booleana que indica se o avaliador deve ser cacheado ou não. Como ambas as classes do `front-end` implementam a mesma interface, o método fábrica pode retornar simplesmente um ponteiro para `tops::model::Evaluator`. Com esse design, conseguimos melhorar usabilidade da API dos modelos, tanto escondendo as implementações concretas dos usuários (que não precisam se preocupar com elas) quanto fazendo com que o cliente da classe só dependa do modelo para criar seus *front-ends*.

Como último detalhe da implementação, é importante notar os modelos decodificáveis (Diagrama 4.1) possuem um comportamento extra: eles também podem avaliar sequências de rotulações. Para reutilizar a mesma estrutura citada anteriormente, adicionamos a todas as classes da hierarquia de avaliadores mais um parâmetro de *template*: um **decorador** de sequência. Caso a sequência seja padrão (decorador `Standard`), o avaliador agirá conforme explicado. Caso a sequência seja rotulada (com decorador `Labeling`), os mesmos métodos e classes valem para pares de sequência de observação e de rótulo. Para gerar esses avaliadores, usamos o método fábrica `labelingEvaluator`, que é exclusivo da sub-hierarquia com raiz em `tops::model::DecodableModel`.




**(a)** *Hierarquia de classes dos Avaliadores*

**(b)** *Hierarquia de classes dos Geradores*

**(c)** *Hierarquia de classes dos Calculadores*

**(d)** *Hierarquia de classes dos Rotuladores*

**(e)** *Hierarquia de classes dos Serializadores*

**(f)** *Hierarquia de classes dos Treinadores*

**Diagrama 4.3:** ***Hierarquias de classes dos front-ends do ToPS****: as interfaces raízes de cada hierarquia herdam da classe* `std::enable_shared_from_this` *do C++, necessária para fazer a alocação e cópia dos objetos com ponteiros inteligentes (Meyers, 2005). As versões cacheadas dos front-ends são implementados como subclasses das versões simples, pois também podem ser usadas nos métodos dedicados a elas, se o cache for ignorado.*

### Geradores

Os geradores - segundo *front-end* implementado no ToPS - têm funcionamento extremamente similar ao dos avaliadores. Delegam sua tarefa de gerar sequências para os métodos `drawSequence` e `drawSymbol` presentes nos *back-ends* - de forma análoga aos avaliadores. Diferentemente deles, porém, os métodos de sorteio aos quais delegam os geradores não possuem versão cacheada - o que faz com que a hierarquia de geradores (Diagrama 4.3b) precise ter apenas a interface `tops::model::Generator` e a implementação concreta `tops::model::SimpleGenerator`. Ambas as classes possuem um parâmetro de *template* para o decorador, por conta dos modelos decodificáveis (mais uma vez, como no caso dos avaliadores). Os métodos fábrica que instanciam os geradores são `standardGenerator` e `labelingGenerator`, implementados nos mesmos modelos que os seus métodos análogos nos avaliadores.

Como principal diferencial desse *front-end*, temos que os objetos são criados passando-se um **gerador de números pseudo-aleatórios** (PRNGs) no construtor das classes. O C++, na biblioteca padrão de `templates` (STL) oferece algumas variedades de algoritmos[11] encapsuladas dentro de funtores (Stroustrup, 2013) para produzir esses números. Os métodos `draw` dos `back-ends` usam um *wrapper* presente na biblioteca de modelos para encapsular os contêineres da STL e sortear as probabilidades necessárias para formar novas sequências. Com essa abordagem, os usuários da API podem se aproveitar dos recursos padrão do C++ e para decidir com que algoritmo as sequências serão geradas. Caso prefiram, também, pode simplesmente ignorar esse flexibilidade, e usar o algoritmo de números aleatórios padrão (*mersenne twister*[12]).

### Calculadores

O *front-end* dos calculadores foi implementado pelo aluno de doutorado Ígor Bonadio, que se baseou nos exemplos feitos com os avaliadores e geradores para refatorar essa parte da hierarquia dos modelos probabilísticos. Esse *front-end* delega suas tarefas para os métodos `calculate`, que fazem parte das classes dos *back-ends*. Os calculadores diferem dos avaliadores, em termos de implementação, em deles em apenas três aspectos:

- **Modelos válidos**
  Os calculadores são o primeiro *front-end* a valer, em todos os casos, para apenas uma parte da hierarquia de modelos, sendo gerados pelo método fábrica `calculator` presente nos modelos decodificáveis.

- **Decoradores de sequências**
  Em vez de aceitar sequências padrão ou rotuladas, os calculadores exigem sempre sequências simples (e, portanto, têm um parâmetro a menos em seus *templates*).

- **Direção de cálculo**
  Tanto o *front-end* quanto o *back-end* possuem a *direção de cálculo* no método `calculate`, que é uma classe enumerável (`enum class`) definida **pela classe do *front-end***.

A hierarquia de calculadores, com sua interface, implementação simples e cacheada, pode ser vista no Diagrama 4.3c.

[11] **C++ Reference** n.d.
[12] Vide item acima

### Rotuladores

Os rotuladores, também implementados pelo doutorando Ígor Bonadio, são extremamente similares aos calculadores. O método `labeling`. para o qual o `front-end` delega a execução da sua funcionalidade, é exclusivo dos modelos decodificáveis, possuindo uma versão com e outra sem cache; Esse método possui apenas um parâmetro: o tipo de algoritmo de rotulação a ser executado. Essa escolha é feita utilizando uma classe enumerável (*enum class*), presente na raiz da hierarquia de rotuladores (Diagrama 4.3d): a interface `tops::model::Labeler`.

A maior diferença presente nos rotuladores está no próprio método `labeling`, que é uma **sequência rotulada estimada** (um par composto da sequência alvo com a rotulação produzida pelo modelo, mais a probabilidade dessa rotulação ter sido gerada pelo modelo). A existência de um objeto específico para retornar esses valores, com nomes apropriados para cada campo, é mais um ponto positivo para facilitar a usabilidade da API da biblioteca de modelos probabilístico.

### Treinadores

Dentre os *front-ends* implementados no ToPS, os treinadores são, certamente, os mais diferentes. Sua principal funcionalidade é a de *criar* novos modelos, utilizando um conjunto de **sequências de treinamento** para estimar os parâmetros internos que definem, matematicamente, os modelos probabilísticos. Comparados aos exemplos anteriores, podemos observar duas particularidades que dificultam a implementação desse *front-end*: primeiramente, como os treinadores funcionam como uma fábrica de modelos, seus objetos não podem ser gerados por métodos de instância existentes nos *back-ends*; em segundo lugar, os algoritmos de treinamento compartilham muitos poucos parâmetros entre si, sendo quase impossível criar uma interface genérica de treinador que descreva, com seus métodos, todos essas técnicas. Ambas características devem ser levadas em conta para a implementação do *front-end* dos treinadores.

Para realizar nossa modelagem, considerando as dificuldades citadas acima, criamos primeiramente a classe `tops::model::Trainer`, contendo um único método de nome `train`. Para que esse método possa lidar com vários algoritmos de treinamento sem conhecê-los previamente, adicionamos à sua assinatura um ***template* variádico** (Stroustrup, 2013), permitindo que, a cada aplicação da função, o compilador gere uma versão específica dela para o conjunto de parâmetros passados como argumento. Utilizando a técnica de *perfect forwarding* (Meyers, 2014), podemos encaminhar todos esses parâmetros para o *back-end* que, finalmente, executará o treinamento. Como nosso *back-end* não será um objeto (pois é o que queremos gerar), o *front-end* precisará receber um argumento de *template* que indique quak a classe do modelo a ser treinado. Como a delegação será feita para uma classe, todos algoritmos de treinamento precisam ser colocados dentro de **métodos estáticos**, que serão nomeados, como no *front-end*, de `train`. Para garantir que o mecanismo de polimorfismo sempre distinga entre as várias versões desse método, usaremos a técnica de *tag dispatch*[13], exigindo que o primeiro parâmetro de `train` seja uma *tag* representando o nome do algoritmo. Desse modo, os *back-ends* podem implementar diversos algoritmos de treinamento, sem risco de conflitos, enquanto os *front-ends* se adaptam automaticamente para lidar com eles.

Uma vez compreendido o mecanismo básico de implementação dos treinadores, é necessário explicar, ainda, dois tipos de treinamento que trazem complicações para nossa modelagem final.

[13] **More C++ Idioms - Wikibooks, open books for an open world** n.d.

Primeiramente, uma das características mais importantes dentre os modelos decodificáveis refatorados no ToPS é a presença de classes que representam **estados**: pares ou triplas de modelos probabilísticos que definem a emissão, transição e (opcionalmente) duração de um estado dos modelos probabilísticos. Como essa implementação (típica de um de padrão *Composite* (Gamma et al., 1995)) usa os próprios modelos da hierarquia, os modelos decodificáveis precisam treinar esses estados durante os seus próprios processos de treinamento. Na implementação dos treinadores mostrada anteriormente, o método `train` do *front-end* recebia, na sua lista de argumentos, os parâmetros que configuram o algoritmo de treinamento. Embora essa modelagem seja a mais simples para usar a função (na perspectiva de um usuário programando com a API), ela é problemática para os treinamentos recursivos: não é trivial separar os parâmetros dos treinamentos dos submodelos a partir de um único conjunto de argumentos passados para o método *train* do modelos decodificável.

Num ponto completamente diferente da hierarquia principal do ToPS, alguns modelos simples (isto é, não decodificáveis), apresentam o segundo problema que dificulta nossa modelagem final. Modelos como o `tops::model::DiscreteIIDModel` podem ser criados com parâmetros fixos: definidos pelo usuário de forma explícita. Para criar os modelos dessa maneira, as classes que os representam contam com métodos `make`, que alocam e inicializam um novo modelo construindo-os diretamente. O método `train`, para o caso dos parâmetros fixos, simplesmente copia um modelo, já existente, passado como argumento na chamada do método. A ideia de possuir esse tipo de treinamento (embora pareça, inicialmente, redundante) é permitir a criação de modelos fixos nos estados de modelos decodificáveis - o que é necessário porque, nos treinamentos recursivo, chama-se método `train`. e não `make`, dos submodelos.

Para lidar com o caso base e mais essas duas situações específicas, foi criada a hierarquia de treinadores (Diagrama 4.3f), que contém a classe `tops::model::Trainer` e três implementações concretas. Para podermos reutilizar o método `train` (descrito no início da seção) entre elas, não pudemos contar. porém, com os mecanismos tradicionalmente aplicados em hierarquias de classe. Como o método `train` foi feito usando-se *templates*, a própria especificação do C++[14] impede que a função seja declarada como virtual - o que permitira que ela fosse sobrescrita nas subclasses e, com isso, alterasse a ação do método nas três implementações concretas do *front-end*. Para conseguir esse comportamento, usamos uma abordagem alternativa, modificando a implementação de `train` para usar dois métodos protegidos puramente virtuais: o método booleano `delegate`, e o método fábrica `trainAlt`. Internamente, `train` verifica o valor retornado por `delegate`. Caso ele devolva verdadeiro, chama o *back-end* repassando os argumentos recebidos na aplicação do método. Caso contrário, chama `trainAlt`, que não recebe nenhum parâmetro e deve devolver um ponteiro para modelo probabilístico. Dependendo de como cada subclasse sobrescreve esses métodos, podemos cobrir todos os tipos de treinamento existentes.

As três implementações concretas da hierarquia de treinadores são:

- **Treinador simples** (`tops::model::SimpleTrainer`)
  O treinador simples ignora o método `trainAlt`. Consequentemente, todos os argumentos passados da aplicação de `train` no *front-end* são repassados diretamente para a versão desse método no *back-end*. Para tanto, os treinadores simples sempre retornam o valor verdadeiro nas chamadas ao método `delegate`. Como `trainAlt` nunca deveria ser chamado, mas necessita de uma implementação, ao ser executado lança uma exceção de erro de lógica.

[14] **C++ Reference** n.d.

- **Treinador cacheado** (`tops::model::CachedTrainer`)
  Os treinadores cacheados estendem a classe dos treinadores simples, podendo ser usados da mesma maneira que eles. Contudo, a sua principal função é ajudar a resolver o problema do treinamento recursivo. Para tanto, *armazenam* um conjunto de parâmetros de treinamento, passados no construtor e salvos dentro de uma tupla (`std::tuple`), para serem usados posteriormente. Essa implementação é possível porque a classe dos treinadores cacheados é definida como um *template* variádico, cujos argumentos representam o tipo dos objetos que serão enviados ao algoritmos de treinamento.

  Para criar um modelo treinado com os parâmetros salvos, os treinadores cacheados contam com uma versão sobrescrita do método `train` que **não possui parâmetros**. Ela funciona devolvendo o resultado de `trainAlt`, que, por sua vez, chama o método `train` do *back-end* usando os dados da tupla.

  Com esse técnica de armazenamento de parâmetros, é possível escrever os algoritmos de treinamento recursivos de forma muito simples: no `back-end`, o método `train` pode aceitar como argumento uma lista de treinadores dos submodelos. Assumindo que todos eles sejam cacheados, a função pode usar um simples laço para chamar o método `train` sem argumentos de cada um, treinando seus estados e, então, completando o seu próprio treinamento.

- **Treinador fixo** (`tops::model::FixedTrainer`)
  O treinador fixo, em oposição às outras duas implementações, sobrescreve o método `delegate` para sempre executar `trainAlt` - independentemente dos conjunto de parâmetros na chamada ao método `train`. O método `trainAlt`, por sua vez, chama o método `make`, fazendo uma cópia do modelo base que contém os parâmetros fixados. Esse modelo é passado ao treinador fixo em seu construtor, permitindo que múltiplos modelos sejam gerados a partir desse modelo básico.

Para concluir a implementação da hierarquia de treinadores, adicionamos mais algumas modificações, com objetivo de flexibilizar ainda mais o design das classes:

- **Decorador de modelos**
  Alguns modelos decodificáveis, como as Cadeias Ocultas de Markov, possuem dois tipos de algoritmos de treinamento: os que usam sequências simples, e os que precisam de rotulações. Para possibilitar ambas as modelagens sem duplicação, adicionamos aos *templates* de todas as classes um parâmetro extra para o decorador de modelos - de forma similar ao feito anteriormente nos outros *front-ends*.

- ***Builder* de conjunto de treinamento**
  Na classe abstrata `tops::model::Trainer`, adicionamos métodos para construir, incrementalmente, o conjunto de treinamento. Os sequências ficam armazenados dentro do objeto do *front-end*, e podem ser acessadas pelo *back-end* por meio de um método *getter*. Dessa maneira, os treinadores funcionam como um exemplo de padrão *Builder* (Gamma et al., 1995), que permite a criação de modelos de forma iterativa. Essa abordagem é especialmente útil para o treinamento dos modelos decodificáveis, nos quais o conjunto de treinamento do modelo principal pode ser dividido em partes e usado para treinar os submodelos.

Assim como no caso de outros *front-ends*, desejávamos criar métodos fábrica para os treinadores. Entretanto, como não há um objeto que permita instanciá-los, tivemos de implementá-los como métodos estáticos. Esses métodos fábrica, chamados de `trainers`, possuem três versões, e utilizam o tipo e número de argumentos para decidir qual implementação concreta do *front-end* será criada. Caso não haja argumentos, o treinador gerado é o simples. Passando diversos parâmetros (excluindo a *tag* que identifica o tipo de treinamento), o treinador será cacheado. Por fim, se o argumento for uma instância da classe em que se está chamando o método `trainer`, o treinador será fixo.

Por serem estáticos, os métodos fábrica dos treinadores não podem ser colocados nas classes abstratas `tops::model::ProbabilisticModel` ou `tops::model::DecodableModel` para que a sua implementação seja reutilizada em toda a hierarquia. Como descrito anteriormente, a interface `tops::model::Trainer` depende (como argumento do *template*) do tipo do modelo que será treinado. Como as classes abstratas não conhecem o tipo das suas classes filhas, não podem gerar o treinador adequado, Para evitar a reimplementação desnecessária desses métodos, é necessário usar outro tipo de recurso, que explicaremos em detalhe na Subseção 4.5.3.

### Serializadores

O *front-end* de serialização foi o último a ser implementado na nova arquitetura do ToPS. Seu principal objetivo é permitir que os parâmetros dos modelos sejam persistidos entre execuções consecutivas dos aplicativos do arcabouço. Para tanto, os serializadores devem poder percorrer um conjunto de modelos probabilísticos, e traduzi-los para alguma linguagem desejada.

Para implementarmos o *front-end* de serialização, e permitir que ele persista os modelos em várias linguagens diferentes, resolvemos utilizar o padrão *Visitor* (Gamma et al., 1995) para desacoplar a maneira como um conjunto de objetos é percorrido da ação realizada nesses objetos. A interface `tops::model::Translator` define um único método puramente virtual chamado `translate`. A classe concreta `tops::model::SimpleTranslator` é a responsável por implementá-lo, delegando sua funcionalidade para o equivalente no *back-end*. Na realidade, apenas a classe concreta bastaria, mas foi definida com uma interface por padronização com os outros *front-ends*. Na construção, o serializador recebe um ponteiro para um **tradutor** (que é o *visitor* em si), definido pela interface `tops::model::Translator` e com vários métodos puramente virtuais `translate`, um para cada modelo probabilístico. Quando recebe um modelo, um tradutor que implementa a interface lê os seus parâmetros e os escreve segundo a sintaxe da linguagem para a qual traduz. Os métodos `translate` são chamados na implementação de `serialize` dos modelos probabilísticos. No caso simples, essa implementação possui apenas uma chamada ao método `translate`, passando para ele o ponteiro `this` do objeto. Nos modelos decodificáveis - que são os que tornam a hierarquia de classes uma *composite* - é necessário um passo extra: executar o método `serialize` nos estados (que repassam a chamada para os submodelos contidos dentro deles). Como, de forma geral, os modelos decodificáveis têm ponteiros para modelos quaisquer, o *double dispatch* feito entre o serializador e o tradutor é muito importante, pois permite que o modelo (internamente) informe qual é o seu tipo, e que então o polimorfismo descubra qual método do tradutor deve ser executado.

Com esse último *front-end*, fomos capazes de separar todas as funcionalidades dos modelos segundo a proposta de nova arquitetura. Antes de seguirmos, finalmente, para as considerações sobre os benefícios da refatoração (Capítulo 5), dedicaremos um espaço para comentar sobre as últimas melhorias de reúso feitas na arquitetura como um todo.

### 4.5.3 Hierarquia com *templates*

Ao discutirmos o *front-end* dos treinadores, encontramos uma dificuldade técnica: a classe abstrata `tops::model::Trainer` recebe como parâmetro de *template* o tipo de modelo que será treinado. Por conta disso, implementar o método fábrica `trainer` em `tops::model::ProbabilisticModel` seria inútil, dado que ela não tem conhecimento das suas subclasses.

O problema de saber qual o tipo das subclasses também ocorre em outros casos. No ToPS, utilizamos ponteiros inteligentes (em particular, `std::shared_ptr`) para administrar automaticamente o alocação e liberação de memória dos modelos probabilísticos. Para garantir que a criação sempre use esses pointeiros, definimos métodos `make` (com *template* variádico e uso de *perfect forwarding* (Meyers, 2014)) para instanciá-los. Assim como no parágrafo anterior, a implementação dos métodos `make` varia num único ponto: qual o tipo da classe que está sendo criada. Poderíamos, novamente, reutilizar o código, mas esbarramos, mais uma vez, no mesmo problema.

Para construir os *front-ends* e, posteriormente, permitir a delegação de suas tarefas, os modelos precisam, internamente, passar ponteiros para suas instâncias. Embora tenhamos, nos métodos, o ponteiro `this`, usá-lo não se adequa à nossa estratégia de administração de memórias baseada em ponteiros inteligentes. Para criar um `std::shared_ptr` a partir do ponteiro `this`, é necessário usar a classe `std::enable_shared_from_this`, que fornece o método `std::shared_from_this` para suas subclasses. De modo a evitarmos herança multipla, herdamos dessa classe apenas nas raízes das hierarquias. Nas subclasses, fazemos *cast* para corrigir o tipo do ponteiro inteligente. Para evitar chamar `std::shared_from_this` e fazer o *cast* repetidamente, criamos o método `make_shared`, que encapsula essa lógica. Assim como nos dois casos anteriores, sua implementação difere apenas em apenas um trecho: o tipo da classe.

Para permitirmos, finalmente, o reúso dessas implementações, podemos utilizar o idioma do C++ *Curiously Recurring Template Pattern*[15] (CRTP), que se propõe a adicionar numa classe um argumento de *template* para **o tipo da subclasse** que a está estendendo. Com esse mecanismo, a superclasse gerada na instanciação do *template* pode usar o tipo da subclasse em seus métodos.

No ToPS, resolvemos criar classes com o sufixo `CRTP` para as principais classes abstratas da hirarquia de modelos. Geramos, assim. `tops::model::ProbabilisticModelCrtp` e `tops::model::DecodableModelCrtp`. Em vez dos modelos probabilísticos herdarem diretamente das classes abstratas, passaram a estender as classes `CRTP`, passando a sua própria classe como argumento do `template`. Dessa maneira, pudemos implementar os métodos `serializer`, `make` e `make_shared` apenas uma vez, em cada classe `CRTP` apropriada.

Com esse mecanismo, também fomos capazes de definir nas classes `CRTP` métodos virtuais para os métodos usados pelos front-ends. Antes de usarmos essa técnica, isso era impossível, pois o primeiro argumento dos métodos (a classe das implementações concretas dos *front-ends*) dependiam, de forma geral, do tipo da subclasse. Com a mudança, ao estender a classe `CRTP`, um modelo é obrigado a sobrescrever os métodos específicos (ou pode usar a versão herdada). A vantagem é que, com isso, implementar um novo modelo torna-se uma prática segura, pois a compilação do modelo falhará até que ele cumpra com os requisitos mínimos da sua interface.

Com as modificações feitas para introduzir as classes `CRTP`, chegamos, enfim, à conformação da arquitetura presente no Diagrama 4.1. Deixamos, por fim, para o Capítulo 6, a tarefa de discutir sobre os benefícios da refatoração para o sistema.

[15] **More C++ Idioms - Wikibooks, open books for an open world** n.d.

# Capítulo 5

# Resultados

## 5.1 Design SOLID

Neste trabalho, dedicamos o maior esforço de refatoração para alterar a arquitetura da biblioteca de modelos probabilísticos. Para avaliar o efeito das mudanças propostas na Seção 4.5, podemos analisá-las sob a perspectiva dos **Princípios da Programação Orientada a Objetos** (Martin, 2000), definidos por Robert C. Martin com o acrônimo SOLID (Figura 5.1). Assim, podemos verificar a correspondência entre a nova arquitetura e relação às boas práticas da orientação a objetos.

- **Princípio da Responsabilidade Única**
  Ao eliminarmos o mau cheiro de **hierarquia não fatorada** (Diagrama 3.4), retiramos as duplicações presentes nas classes dos modelos. Ao desassociarmos a linguagem de especificação da arquitetura, separamos a implementação dos algoritmos da construção da linguagem. Com a delegação dos *front-ends* para os *back-ends*, todo o código dos modelos encontra-se definido em apenas uma classe - revelando a forte coesão presente no design do sistema.

- **Princípio do Aberto Fechado**
  A criação de interfaces e de classes abstratas `CRTP` permitiu a fácil adição de novos modelos no arcabouço. O uso intensivo de métodos virtuais em quase todas as implementações presentes na biblioteca de modelos permite que qualquer classe seja estendida e tenha seus comportamentos sobrescritos - mostrando a flexibilidade do design do sistema.

| | | |
|---|---|---|
| S | **Princípio da Responsabilidade Única (SRP)**<br>(Single Responsability Principle, SRP) | Uma classe deve ter um, e apenas um, motivo para mudar |
| O | **Princípio do Aberto Fechado (OCP)**<br>(Open Closed Principle) | Deve ser possível estender o comportamento de uma classe, sem modificá-la |
| L | **Príncipio da Substituição de Liskov (LSP)**<br>(Liskov's Substitution Principle) | Classes derivadas devem ser substituíveis por suas superclasses |
| I | **Princípio da Segregação de Interfaces (ISP)**<br>(Interface Segregation Principle) | Muitas interfaces específicas são melhores que uma interface de propósito geral |
| D | **Princípio da Inversão de Dependências (DIP)**<br>(Dependency Inversion Principle) | Deve-se depender de abstrações, não de implementações |

**Figura 5.1:** *Princípios da Programação Orientada a Objetos (Martin, 2000)*

- **Princípio da Substituição de Liskov**
  O princípio da substituição, criado por Bárbara Liskov, defende que as classes que usam herança deve possuir a relação semântica "é um"entre as entidades do sistema. Na arquitetura, o uso de herança restrito às classes abstratas e suas implementações permite que as subclasses sejam utilizadas como ponteiros ou referências para suas superclasses - salientando o uso genérico dos elementos da arquitetura.

- **Princípio da Segregação de Interfaces**
  A divisão das funcionalidades dos *back-ends* por meio de vários *front-ends* diminui as dependências dos clientes da biblioteca de modelos, que podem utilizar apenas uma parte das ações e dados disponibilizados pelas classes. Na hierarquia principal do sistema, a adição de novos comportamentos via interfaces define, claramente, que tarefas podem ser realizadas por suas subclasses - facilitando o entendimento das relações entre os tipos do arcabouço.

- **Princípio da Inversão de Dependências**
  Por conta do uso de classes abstratas para definir todas as entidades do sistema, todos os tipos da biblioteca de modelos são disponibilizados por meio de suas interfaces. Os usuários têm a liberdade de criar suas próprias implementações, e o sistema é capaz de se adaptar para utilizá-las - o que demonstra a simplicidade de extensão do sistema.

Ao seguirmos os princípios da programação orientada a objetos, criamos uma uma arquitetura robusta e reutilizável, mantendo um dos principais atrativos da versão publicada do ToPS.

## 5.2 Atributos de qualidade

No início deste projeto, traçamos como objetivo melhorar a qualidade do ToPS em três aspectos: compreensibilidade, mutabilidade e extensibilidade. Nesta seção, usaremos as modificações feitas no Capítulo 4 para mostrar como a refatoração afetou esses três atributos.

### 5.2.1 Compreensibilidade

A compreensibilidade é um dos atributos mais importantes para qualquer sistema. Numa definição abrangente, refere-se à capacidade com que uma pessoa pode entender o funcionamento de um dado programa. Tendo isso em mente, podemos analisar essa compreensão sobre diversos pontos de vista. Os mais relevantes, no caso de um arcabouço como o ToPS, são o usuário final e os desenvolvedores do sistema. Nessa seção, faremos nossa discussão sob essas duas perspectivas.

Para os desenvolvedores, as modificações implantadas no ToPS impactaram no entendimento tanto dos detalhes de implementação quanto do processo de contribuição do sistema. Em primeiro lugar, os testes de unidade formam uma excelente documentação dos comportamentos desejados para os modelos do ToPS. Com as padronizações da formatação e a introdução de nomes mais significativos para os tipos presentes no arcabouço, a leitura do código e dos algoritmos tornou-se muito mais simples. Com uma metodologia bem definidas para criação de novas funcionalidades, pela presença de novos espaços online para discussão e com um conjunto de diretrizes claras para submissão de mudanças, colaborar com o arcabouço tornou-se um processo, por si só, educativo. Juntando todos esses pontos positivos, o ToPS se tornou-se muito mais amigável para novos programadores, o que é muito útil para garantir a renovação dos mantenedores do sistema.

Para os usuários, a geração de bibliotecas separadas - cada uma com uma responsabilidade bem definida - facilitou o uso do arcabouço para futuras aplicações que sejam desenvolvidas. Os aplicativos, embora ainda não implementados, serão reformulados para refletir as funcionalidades dos *front-ends* - tornando claras as possibilidades de uso do sistema. Assim como para os desenvolvedores, os testes servem de exemplos de como utilizar a API da biblioteca de modelos probabilísticos. A disponibilização de um novo `README` com informações de instalação, e o uso do *issue tracker* do GitHub facilitam o esclarecimento de dúvidas e a triagem de erros - diminuindo a barreira de comunicação entre os desenvolvedores e os usuários.

### 5.2.2 Mutabilidade

A capacidade de mudar um sistema é essencial para garantir que ele se mantenha vivo, constantemente se adaptando aos novos objetivos daqueles que o usam. Corrigir, otimizar e refatorar são as principais ações realizadas sobre o código existente, e são essenciais para que um programa melhore a cada versão lançada. Tendo em vista a importância dessas tarefas para a continuidade de qualquer sistema, analisaremos como as mudanças feitas no ToPS ajudaram cada uma delas.

Em termos de correção, os testes automatizados são um dos instrumentos mais úteis para os programadores. Se uma funcionalidade existente é mudada, avisam se as alterações modificaram algum comportamento de forma inapropriada. Se um erro é encontrado, fornecem um processo bem definido para tratá-lo (permitindo criar um novo teste que reproduza o problema). Desse maneira, os testes agem como um mecanismo de auto-preservação do sistema, mantendo ou aumentando a segurança da programação a cada falha encontrada.

Em termos de otimização, a redução de duplicações e a separação de responsabilidades ajuda a restringir os pontos que devem receber melhorias de desempenho. Sejam na forma de paralelizações, ou com técnicas específicas para ajudar na geração de código executável, quanto mais localizados os gargalos de processamento, mais fácil é decidir quais ações realizar sobre eles. Com a presença de mais abstrações e o uso intensivo de orientação a objetos, é possível aproveitar melhor o resultado de *profilings* (que são uma das melhores técnicas para achar problemas de performance). Com a facilidade de encontrar e melhorar os trechos de processamento intensivo, pode-se fazer o ToPS lidar com uma quantidade cada vez maior de dados.

Em termos de refatoração, toda a infra-estrutura de código construída mais as ferramentas utilizadas no sistema facilitarão a continuidade deste trabalho. Mais do que uma intervenção que interrompeu o desenvolvimento do arcabouço, a presença de testes e de um design conciso tornou possível refatorar a qualquer momento. O uso de técnicas de programação modernas - como desenvolvimento dirigido a testes - podem ser facilmente utilizadas, quando antes seriam quase impossíveis. Assim, com todos esses fatores, ficou viável incluir a refatoração na lista de práticas utilizadas continuamente no desenvolvimento.

### 5.2.3 Extensibilidade

A extensibilidade se refere à capacidade de adicionar novas funcionalidades a um sistema. É também uma das formas mais diretas de fazer um programa evoluir, atendendo a requisitos que antes não existiam ou não eram esperados. Dentro do ToPS, podemos encontrar diversos pontos de extensão: os modelos, as funcionalidades e as linguagens dentre eles. Nessa seção, discutiremos o impacto da refatoração na extensibilidade de cada uma dessas partes do sistema.

Da parte dos modelos, a criação de interfaces mais genéricas, cujos métodos estão agrupados segundo suas funcionalidades, facilita escolher onde se deseja criar um novo modelo probabilístico. Com os métodos fábricas presentes nas interfaces e em suas implementações `CRTP`, ganha-se automaticamente a capacidade de criar *front-ends*. O uso intensivo de métodos virtuais facilita o desenvolvimento, pois cria erros caso algum método que deveria ser sobrescrito não o seja. Os mecanismos de metaprogramação em *templates* maximizando verificação em tempo de compilação. Essas mudanças serão muito úteis na implementação dos modelos não portados, pois garante que esse trabalho possa ser feito mesmo sem o auxílio dos autores deste projeto.

Da parte das funcionalidades, a criação de macros e metafunções feitas nos experimentos com arquitetura ajudaram a facilitar a implementação dos *front-ends* presentes na nova versão do ToPS. Dependendo de quais modelos forem incluídos na hierarquia - e das características que eles apresentarem - pode ser necessário agrupar certas funcionalidades em novos *front-ends*. Nesse caso, os exemplos já existentes e os mecanismos de reúso devem facilitar o crescimento da arquitetura sem que ela perca o seu formato, e nem que prejudique o código já existente.

Da parte das linguagens, a remoção dos maus cheiros da hierarquia de modelos probabilísticos e a padronização do alfabeto de símbolos ajudaram a desacoplar a definição interna dos modelos da especificação por meio de uma linguagem. A criação de um *visitor* no *front-end* de serialização e de um *builder* no *front-end* de treinamento permite que as linguagens interajam com o arcabouço por meio de interfaces e métodos bem definidos. Combinando ambos os fatores, fica simples estender o ToPS para que ele converse com novas linguagens (sejam elas de especificação ou de programação), o que é muito conveniente para a aceitação do arcabouço em diferentes áreas de estudo.

# Capítulo 6

# Conclusões e Trabalhos Futuros

Quando iniciamos este projeto, o ToPS era um sistema diferenciado. Surgido como base de implementação do preditor de genes MYOP, o arcabouço focava em criar uma biblioteca de modelos probabilísticos com alto reúso de código. Sobre esse núcleo, o sistema contava com uma linguagem de especificação, cuja sintaxe próxima da definição matemática permitia a configuração de modelos sem o uso de uma linguagem de programação. Com esses dois componentes, construía diversos aplicativos: programas executáveis que permitiam fazer treinamento, inferência, simulação e diversas outras tarefas relacionadas aos modelos probabilísticos. Considerando a existência de todos esses recursos, levando em conta os bons resultados obtidos (indiretamente) pelo MYOP usando os algoritmos e sabendo da possibilidade de reúso do arcabouço em outras áreas de estudo, tivemos a motivação necessária para começar este projeto de refatoração.

Ao longo do trabalho, focamos em propor modificações que permitissem melhorar o sistema. A revisão conceitual dos aspectos de qualidade de software nos deu o conhecimento necessário para definir quais os problemas existentes, classificando-os segundo a granularidade em que se encontravam (código, arquitetura, componentes e repositório). Nossas refatorações dedicaram-se, então, a corrigir essas falhas. De modo geral, acreditamos que as modificações feitas neste projeto foram bem sucedidas, e tiveram impacto positivo sobre a capacidade de compreensão, modificação e extensão do ToPS. Dessa maneira, cumprimos com nosso principal objetivo: reestruturar o sistema para que ele mantenha sua excelência e possa ser usado em um número cada vez maior (e diverso) de aplicações.

Por definição de uso, a aplicação da técnica de refatoração é um processo sem fim, devendo ser usada para combater a desestruturação natural que surge ao modificar o código de um sistema. Por conta da complexidade das mudanças e do tamanho do arcabouço, a refatoração do ToPS iniciada por nós ainda precisa ser estendida. Embora tenhamos feito experimentos e testes, não houve tempo hábil, durante este trabalho, para refatorar a componente da linguagem do ToPS. Consequentemente, os aplicativos, que dela dependem, também não foram reimplementados. Terminar ambos é essencial para podermos substituir a versão publicada do ToPS, disponibilizando a nova API e seus programas para serem usados pelas aplicações do arcabouço.

Por conta dos possíveis usos do sistema, otimizações e paralelizações são dois grandes diferenciais que podem permitir o uso do ToPS com uma quantidade cada vez maior de dados. Essa capacidade é especialmente importante para aplicações como o MYOP, que faz inferências sobre genomas completos e usa milhares de sequências para treinamento. Essas duas melhorias foram pouco explorados durante este trabalho, e são uma boa maneira de expandir os possíveis casos de usos do sistema.

Vale lembrar, por fim, que tivemos de fazer um recorte neste trabalho, selecionando apenas alguns modelos - dentre os publicados no artigo do ToPS - para serem implementados na versão refatorada. Para substituirmos completamente o sistema antigo, precisaremos incluir o restante dos modelos publicados, e refatorar os não publicados para que já se encaixem na nova versão do programa. Esta tarefa, porém, poderá ser realizada aos poucos, e será facilitada por conta das melhorias na capacidade de extensão do arcabouço.

Como legado deste projeto, cremos que ficou muito mais fácil utilizar o ToPS em futuras trabalhos desenvolvidos pelo nosso grupo de pesquisas. Dessa maneira, garantimos que o arcabouço continue sendo estendido, tornando-se uma ferramenta cada vez mais atrativa para a construção de aplicações que usam modelos probabilísticos que fazem descrição de sequências de símbolos.

# Bibliografia


[1] **A list of open source C++ libraries**. URL: http://en.cppreference.com/w/cpp/links/libs.

[2] **All-in-One Makefile**. URL: https://github.com/renatocf/make.

[3] **Apache Subversion**. URL: https://subversion.apache.org/.

[4] Andrew Appel. **Modern Compiler Implementation in Java**. 2nd. Cambridge University Press, 2004, p. 513.

[5] **Automake**. URL: https://www.gnu.org/software/automake/.

[6] Kent Beck. **Extreme Programming Explained: Embrace Change**. c. 1999, p. 224.

[7] Axel Bernal et al. «Global discriminative learning for higher-accuracy computational gene prediction.» Em: **PLoS computational biology** 3.3 (mar. de 2007), e54. DOI: 10.1371/journal.pcbi.0030054. URL: http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.0030054.

[8] **Bison - GNU Project - Free Software Foundation**. URL: https://www.gnu.org/software/bison/.

[9] **BisonC++**. URL: https://fbb-git.github.io/bisoncpp/.

[10] Ígor Bonadio. «Desenvolvimento de um arcabouço probabilístico para implementação de campos aleatórios condicionais». Tese de doutoramento. 2013, p. 58. URL: http://igorbonadio.com.br/files/dissertacao-igor.pdf.

[11] Grady Booch. «Object-Oriented Analysis and Design with Applications (3rd Edition)». Em: (jun. de 2004).

[12] **Boost C++ Libraries**. URL: http://www.boost.org/.

[13] **C++ Reference**. URL: http://en.cppreference.com/w/.

[14] **ChaiScript - Easy to use scripting for C++.** URL: http://chaiscript.com/.

[15] **CMake**. URL: https://cmake.org/.

[16] D. DeCaprio et al. «Conrad: Gene prediction using conditional random fields». Em: **Genome Research** 17.9 (jul. de 2007), pp. 1389–1398. DOI: 10.1101/gr.6558107. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1950907%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract.

[17] R Durbin et al. «Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids». Em: **Analysis** (1998), p. 356. DOI: 10.1017/CBO9780511790492. URL: http://eisc.univalle.edu.co/cursos/web/material/750068/1/6368030-Durbin-Et-Al-Biological-Sequence-Analysis-CUP-2002-No-OCR.pdf.

[18] **Extensible Markup Language (XML)**. URL: https://www.w3.org/XML/.

[19] **Flex: The Fast Lexical Analyzer**. URL: http://flex.sourceforge.net/.

[20] **FlexC++**. URL: https://fbb-git.github.io/flexcpp/.

[21] Martin Fowler e Kent Beck. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 1999, pp. 1–337.

[22] Erich Gamma et al. **Design patterns: elements of reusable object-oriented software**. Vol. 47. Addison-Wesley Longman Publishing Co., Inc., 1995, p. 429. DOI: 10.1016/j.artmed.2009.05.004.

[23] **Git**. URL: https://git-scm.com/.

[24] **Google C++ Style Guide**. URL: https://google-styleguide.googlecode.com/svn/trunk/cppguide.html.

[25] **Google Test**. URL: https://github.com/google/googletest.

[26] Jezz Humble e David Farley. **Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation**. 2010, p. 497.

[27] Ralph E. Johnson. «Components, frameworks, patterns». Em: **ACM SIGSOFT Software Engineering Notes** 22.3 (mai. de 1997), pp. 10–17. DOI: 10.1145/258368.258378. URL: http://dl.acm.org/citation.cfm?id=258368.258378.

[28] **JSON**. URL: http://www.json.org/.

[29] André Yoshiaki Kashiwabara. **MYOP/ToPS/SGEval: A computational framework for gene prediction**. Fev. de 2012. URL: http://www.teses.usp.br/teses/disponiveis/45/45134/tde-02042012-184145/.

[30] André Yoshiaki Kashiwabara et al. «ToPS: A Framework to Manipulate Probabilistic Models of Sequence Data». Em: **PLoS Computational Biology** 9.10 (out. de 2013). Ed. por Hilmar Lapp, e1003234. DOI: 10.1371/journal.pcbi.1003234. URL: http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003234.

[31] Ian Korf. «Gene finding in novel genomes.» Em: **BMC bioinformatics** 5 (mai. de 2004), p. 59. DOI: 10.1186/1471-2105-5-59. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=421630%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract.

[32] RC Martin. «Design principles and design patterns». Em: **Object Mentor** (2000). URL: http://scm0329.googlecode.com/svn-history/r78/trunk/book/Principles%7B%5C_%7Dand%7B%5C_%7DPatterns.pdf.

[33] Rafael Mathias. «Arcabouço probabilístico para análise de sequências de RNA». Tese de doutoramento. 2015, p. 76.

[34] **Mercurial**. URL: https://www.mercurial-scm.org/.

[35] Scott Meyers. **Effective C++: 55 Specific Ways to Improve Your Programs and Designs**. 3rd. Addison Wesley Longman Publishing Co., Inc., 2005, p. 321.

[36] Scott Meyers. **Effective Modern C++: 42 Specific Ways to Improve Your Use of C++ 11 and C++ 14**. 1st. O'Reilly Media, 2014, p. 334.

[37] Scott Meyers. **Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library**. 2nd. Addison Wesley Longman Publishing Co., Inc., 2001, p. 198. URL: https://scholar.google.com.br/scholar?hl=pt-BR%7B%5C&%7Dq=effective+stl%7B%5C&%7DbtnG=%7B%5C&%7Dlr=%7B%5C#%7D3.

[38] **More C++ Idioms - Wikibooks, open books for an open world**. URL: https://en.wikibooks.org/wiki/More%7B%5C_%7DC%7B%%7D2B%7B%%7D2B%7B%5C_%7DIdioms.

[39] Vitor Onuchic. «Inovações em técnicas de alinhamentos múltiplos e predições de genes». Tese de doutoramento. 2012, p. 67.

[40] **OpenMP**. URL: http://openmp.org/wp/.

[41] **Perl Programming Language**. URL: https://www.perl.org/.

[42] **Python Programming Language**. URL: https://www.python.org/.

[43] **R: The R Project for Statistical Computing**. URL: https://www.r-project.org/.

[44] **Ruby Programming Language**. URL: https://www.ruby-lang.org/.

[45] M. Stanke e S. Waack. «Gene prediction with a hidden Markov model and a new intron submodel». Em: **Bioinformatics** 19.Suppl 2 (out. de 2003), pp. ii215–ii225. DOI: 10.1093/bioinformatics/btg1080. URL: http://bioinformatics.oxfordjournals.org/content/19/suppl%7B%5C_%7D2/ii215.abstract.

[46] Bjarne Stroustrup. **Programming: Principles and Practice Using C++ (2nd Edition)**. Addison-Wesley, 2014, p. 1312.

[47] Bjarne Stroustrup. **The C++ Programming Language, 4th Edition**. Addison-Wesley, 2013, p. 1368.

[48] Girish Suryanarayana, Ganesh Samarthyam e Tushar Sharma. **Refactoring for Software Design Smells: Managing Technical Debt**. Morgan Kaufmann Publishers Inc., nov. de 2014, p. 244. URL: http://dl.acm.org/citation.cfm?id=2755629.

[49] Herb Sutter e Andrei Alexandrescu. **C++ coding standards: 101 rules, guidelines, and best practices**. Addison-Wesley Professional, 2004, p. 225.

[50] **The Official YAML Web Site**. URL: http://www.yaml.org/.

[51] Anthony Williams. **C++ Concurrency in Action: Practical Multithreading**. 1st. Manning Publications Co., 2012, p. 530.