

ENGENHARIA REVERSA: TRANSFORMADOR DE CÓDIGO C++ PARA MODELO

Guilherme Bolfe¹

Giliardi Schmidt²

Fabio Paulo Basso³

Elder de Macedo Rodrigues⁴

Resumo:

Muitos produtos de software ao passar do tempo ficam obsoletos, devido a constante evolução das necessidades dos usuários e da evolução natural das tecnologias disponíveis, necessitando assim manutenções ou adições de funcionalidades. Para realizar estas alterações no software, se faz necessário entender o código-fonte do sistema: sua estrutura, algoritmos e fluxos de execução. Porém, há situações em que este código-fonte está documentado de maneira inconsistente ou superficial, ou até mesmo não possui nenhum documento de apoio, tornando o entendimento do sistema uma tarefa custosa. O transformador gerado neste trabalho tem a capacidade de extrair as informações estruturais de classes: o nome da mesma, quais são seus atributos e funções, a visibilidade destes atributos e funções e a assinatura (parâmetros de entrada e tipo de retorno) destas funções. É possível citar como vantagem do trabalho realizado que, após abstrair as informações do código-fonte para um modelo, é possível exportá-lo para outras linguagens de programação. Assim, facilitando a migração de aplicações. Além de exportações e migrações, após que o processo de engenharia reversa estar aplicado, podemos utilizar técnicas de verificação e validação nos modelos gerados, garantindo uma melhor qualidade das aplicações.

Palavras-chave: Engenharia Reversa, ANTLr, C++, UML, AST

Modalidade de Participação: Iniciação Científica

ENGENHARIA REVERSA: TRANSFORMADOR DE CÓDIGO C++ PARA MODELO

¹ Aluno de graduação. guilhermebolfe11@gmail.com. Autor principal

² Aluno de Graduação. gili.smdt@gmail.com. Co-autor

³ Docente. fabiobasso@unipampa.edu.br. Orientador

⁴ Docente. elderrodrigues@unipampa.edu.br. Co-orientador

ENGENHARIA REVERSA: TRANSFORMADOR DE CÓDIGO C++ PARA MODELO

1 INTRODUÇÃO

Muitos produtos de software ao passar do tempo ficam obsoletos, devido a constante evolução das necessidades dos usuários e da evolução natural das tecnologias disponíveis, necessitando assim manutenções ou adições de funcionalidades. Para realizar estas alterações no software, se faz necessário entender o código-fonte do sistema: sua estrutura, algoritmos e fluxos de execução. Porém, há situações em que este código-fonte está documentado de maneira inconsistente ou superficial, ou até mesmo não possui nenhum documento de apoio, tornando o entendimento do sistema uma tarefa custosa.

Além da necessidade de alterar o código-fonte do software, há casos onde deseja-se efetuar as tarefas de verificação e validação no sistema. Estas duas tarefas podem necessitar de uma documentação mais abrangente sobre o código-fonte, como a estrutura estática do mesmo (KORSHUNOVA, 2006). O processo de verificação e validação analisa exaustivamente e testa o software para determinar se ele executa suas funções, para garantir que ele realiza suas funções de forma correta e mensurar sua qualidade e confiabilidade (WALLACE, 1989).

Para obter esta documentação faltante, é possível utilizar a engenharia reversa, onde partimos de um código-fonte pronto e o elevamos para um nível mais abstrato, um modelo. O processo de obtenção de representações úteis de alto nível de a partir de código-fonte é chamado engenharia reversa (BRUNELIÈRE, 2014).

Neste artigo apresentaremos uma engenharia reversa, onde extraímos de um código na linguagem C++ as características estruturais estáticas, como classes, atributos de classes e assinaturas de métodos, para um diagrama de classes UML - *Unified Modeling Language*. Esta linguagem de modelagem foi criada por especialistas em modelagem, para consolidar e padronizar o grande número de linguagens de programação orientadas a objetos, por volta do final dos anos 80 e início dos anos 90 (LISBOA, 2008). Este diagrama UML, por fim, é salvo em um arquivo no formato XML.

O processo de engenharia reversa também está presente na migração de aplicações (FLEUREY, 2007), pois após obter um modelo contendo as informações estruturais, é possível exportar código-fonte para diversas linguagens, sendo apenas necessário a criação de um transformador.

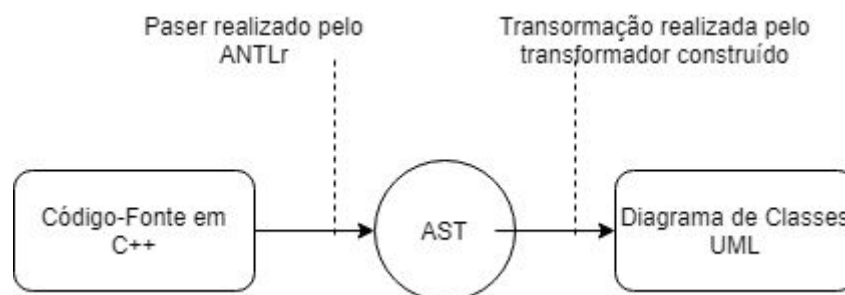
Para conseguir elevar o código a um nível mais alto é necessário criar uma árvore sintática abstrata - AST (*Abstract Syntax Tree*), esta que por sua vez possui diversas formas de ser construída, utilizando diferentes ferramentas. Neste trabalho optamos por utilizar a ferramenta ANTLr (TERENCE, 2018), o qual se mais adequa à nossa necessidade, devido o fato de possuir gramáticas prontas. Após sua construção, a AST pode ser utilizada para inúmeros fins, como, análise de softwares, desenvolvimento de softwares e construção de compiladores (FISCHER, 2007).

2 METODOLOGIA

O processo de transformação do código-fonte para o modelo UML ocorre em duas etapas. A Figura 1 apresenta uma visão geral deste processo. É gerada uma AST utilizando a

ferramenta ANTLr (veja a Figura 3 para um exemplo de AST gerada a partir do código-fonte da Figura 4. Esta AST é gerada com base em uma gramática pré-definida.

Figura 1 - Processo de transformação do código-fonte para modelo.



Fonte: Autor, 2018.

Após isso, o transformador construído navega por esta AST, extraindo informações sobre as classes, atributos e funções destas classes, gerando um diagrama de classes UML.

Com a utilização da ferramenta ANTLr, foi gerado um parser, para a linguagem de programação Java. Para a geração deste parser, utilizou-se uma gramática pronta (SANCHEZ 2018). A partir do parser gerado pelo ANTLr, estendeu-se a classe abstrata *BaseVisitor* (gerada automaticamente pelo ANTLr) e criou-se diversos *Visitors* para extrair as informações necessárias da AST gerada. A Figura 2 apresenta um dos *Visitors* criados, sendo o responsável por extrair informações sobre uma classe.

Assim, o transformador construído neste trabalho aplica os *Visitors* gerados na AST e cria os elementos do diagrama UML com base no tipo de nó visitado.

3 RESULTADOS E DISCUSSÃO

O transformador gerado neste trabalho tem a capacidade de extrair as informações estruturais de classes: o nome da mesma, quais são seus atributos e funções, a visibilidade destes atributos e funções e a assinatura (parâmetros de entrada e tipo de retorno) destas funções.

Para exemplificar o seu funcionamento, considere o que é apresentado nas Figuras 2, 3 e 4. Ao fornecer como entrada o código-fonte da Figura 4 ao parser criado pelo ANTLr, é gerada de forma automatizada a AST da Figura 3. Ao aplicar o *Visitor ClassSpecifierVisitor* ao nó raiz de uma AST, sempre que houver um nó do tipo *classspecifier* (que corresponde à uma classe), o método *visitClassSpecifier(ClassSpecifierContext ctx)* será invocado. Isto ocorre, por exemplo, para o primeiro nó presente na Figura 3 (quando vista de cima para baixo). Para extrair o nome desta classe, basta navegar pelos filhos do nó em questão, como exemplificado na linha 19 da Figura 2. Após isso, são aplicados novos *Visitors* conforme necessário. Por exemplo, é aplicado um novo *Visitor*, que é responsável por extrair as informações de atributos e funções da classe (linhas 29 e 30 da Figura 2).

Figura 2 - Visitor gerado para visitar nós do tipo *classspecifier*.

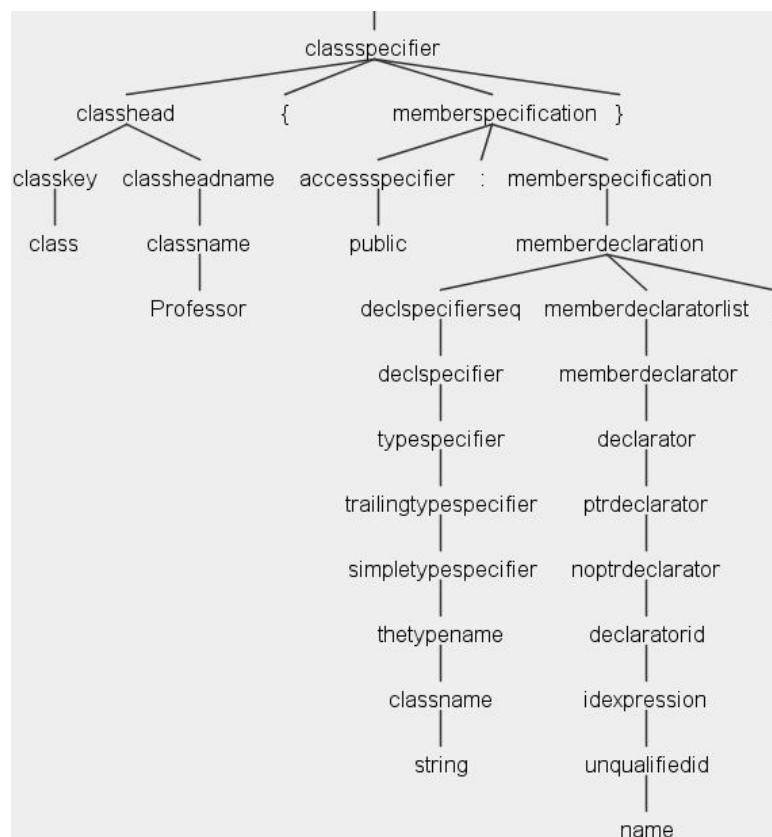
```

9 public class ClassSpecifierVisitor extends BaseVisitor<Class> {
10
11     public ClassSpecifierVisitor(Model model) {
12         super(model);
13     }
14
15     @Override
16     public Class visitClassspecifier(ClassspecifierContext ctx) {
17         Class c = null;
18
19         String name = ctx.classhead().classheadname().getText();
20
21         try {
22             c = super.createClass(name);
23
24             if (ctx.classhead().baseclause() != null) {
25                 ClassNameVisitor classNameVisitor = new ClassNameVisitor(model, c);
26                 ctx.classhead().baseclause().accept(classNameVisitor);
27             }
28
29             MemberSpecificationVisitor memberSpecificationVisitor = new MemberSpecificationVisitor(model, c);
30             ctx.accept(memberSpecificationVisitor);
31
32         } catch (Exception e) {
33             e.printStackTrace();
34         }
35
36         return c;
37     }

```

Fonte: Autor, 2018.

Figura 3 - AST gerada a partir de um código-fonte em C++.



Fonte: Autor, 2018.

Figura 4 - Código-fonte em C++.

```
1 class Professor {  
2     public:  
3         string name;  
4 };
```

Fonte: Autor, 2018.

4 CONSIDERAÇÕES FINAIS

O transformador gerado tem a capacidade de extrair as informações estruturais estáticas de códigos fonte escritos em C++. Porém, sua efetividade não foi testada e avaliada. Ou seja, pode haver situações em que o transformador não consiga extrair as informações corretas, ou até mesmo não funcione.

É possível citar como vantagem do trabalho realizado que, após abstrair as informações do código-fonte para um modelo, é possível exportá-lo para outras linguagens de programação. Assim, facilitando a migração de aplicações. Além de exportações e migrações, após que o processo de engenharia reversa estar aplicado, podemos utilizar técnicas de verificação e validação nos modelos gerados, garantindo uma melhor qualidade das aplicações.

Para trabalhos futuros, após uma bateria de testes para assegurar o funcionamento do transformador em diferentes situações, poderão ser criados geradores de código para diferentes plataformas.

REFERÊNCIAS

E. KORSHUNOVA, M. PETKOVIC, M. G. J. V. DEN BRAND, M. R. MOUSAVI. **CPP2XML: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code**. In: 13TH WORKING CONFERENCE ON REVERSE ENGINEERING, 2006, p.297-298.

D. R. WALLACE, R. U. FUJII. **Software verification and validation: an overview**. 1989.

J. LISBOA F., J. L. BRAGA. **UML: Unified Modeling Language**. 2008.

FLEUREY, FRANCK, BRETON, ERWAN, BAUDRY, BENOIT, NICOLAS, ALAIN, JEQUEL, JEAN-MARC. **Model-driven engineering for software migration in a large industrial context**. In: INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, 2007, p 482-497.

TERENCE P. **ANTLR ANother Tool for Language Recognition**. Disponível em: <<http://www.antlr.org/>>. Acesso em: 03 jul. 2018.

SANCHEZ C. **CPP14 Grammar**. Disponível em: <<https://github.com/antlr/grammars-v4/tree/master/cpp>>. Acesso em: 03 jul. 2018.

G. FISCHER, J. LUSIARDI, J.W. VON GUDENBERG. **Abstract Syntax Trees – and their Role in Model Driven Software Development**. 2007, p 1-6.

BRUNELIÈRE, H. CABOT, J. DUPÉ, G. MADIOT, F. **MoDisco: A model driven reverse engineering framework.** 2014.