UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE GRADUAÇÃO

CONSOLIDANDO *DESIGN BY CONTRACT* ATRAVÉS DO *ASPECTJML WEB*

IGOR VINÍCIUS PINHEIRO CORDEIRO LEÃO

RECIFE
JULHO DE 2017

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

IGOR VINÍCIUS PINHEIRO CORDEIRO LEÃO

CONSOLIDANDO *DESIGN BY CONTRACT* ATRAVÉS DO *ASPECTJML WEB*

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Henrique Emanuel Mostaert Rebêlo

RECIFE
JULHO DE 2017

IGOR VINÍCIUS PINHEIRO CORDEIRO LEÃO

CONSOLIDANDO *DESIGN BY CONTRACT* ATRAVÉS DO *ASPECTJML WEB*

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

_____

Prof. Henrique Emanuel Mostaert Rebêlo

Orientador

_____

Prof. Sérgio Castelo Branco Soares

Avaliador

RECIFE
JULHO DE 2017

Resumo

Com o surgimento da programação orientada a objetos, cresce naturalmente uma preocupação a respeito da confiabilidade dos sistemas escritos neste paradigma, onde por confiabilidade entende-se a combinação de corretude e robustez: a ausência de bugs. O *Design by Contract* (*DbC*) se desenvolve como uma forma de tratar a confiabilidade. Rebêlo et al. criaram o *AspectJML* como uma extensão do *JML* orientada a aspectos, popular linguagem de *DbC Java*, mantendo as qualidades positivas tanto do *JML* quando da orientação a aspectos. Este trabalho, então, se propõe a transformar o *AspectJML* em um serviço de edição, compilação e execução de programas por meio da *WEB*, afirmando a linguagem, tornando possível sua maior disseminação e facilitando o seu uso, em especial para o ensino.

Palavras chaves: *AspectJML*, *Design by Contract*, *DbC*, *JML*, *IDE*, *WEB*

Abstract

With the emergence of object-oriented programming, there is a growing concern about the reliability of the systems written in this paradigm, where reliability is understood as the combination of correctness and robustness: the absence of bugs. Design by Contract (DbC) is developed as a way to treat reliability. Rebelo et al. have created *AspectJML* as an extension of the *JML*, popular Java *DbC* language, retaining the positive qualities from both JML and aspect-oriented programming. This work aims to transform *AspectJML* into a service for editing, compiling and executing *AspectJML* programs through the *WEB*, consolidating the language, making possible its greater dissemination and facilitating its use, especially for teaching.

Keywords: *AspectJML*, *Design by Contract*, *DbC*, *JML*, *IDE*, *WEB*

Lista de ilustrações

Lista de Tabelas

Lista de abreviaturas e siglas

AJML	Aspect JML
DOM	Document Object Model
IDE	Integrated Development Environment
JML	Java Modeling Language
UML	Unified Modeling Language
WEB	World Wide Web

Agradecimentos

Primeiramente, eu gostaria de agradecer a minha família por durante os últimos 23 anos terem compartilhado, entre outras coisas, carinho e atenção. Ao meu irmão Guilherme, por seu desejo de aprender que me fez ensiná-lo coisas que nem eu mesmo sabia, forçando-me a adquirir conhecimento. Aos meus pais que contribuíram de maneira significativa na minha formação.

Àqueles que fizeram os anos de graduação mais felizes e com quem compartilhei boa parte dela: Marcelo Ferrão, Matheus Dornelas e Vinícius Folha. Aos colegas de mesma entrada que se tornaram especiais e que espero não me esquecer: Mariama Oliveira, Fanny Chien, Rafael Nunes, Bruno Soares, Victor Vernilli e Túlio Lages. A quem acabou seguindo outro caminho que não o da ciência da computação, mas que carrego profundo carinho e alguns projetos conjuntamente realizados no começo do curso: Daniel Andrade e Alexandre França. Àqueles em que eu tive a sorte de esbarrar já no final da graduação: Paulo Lieuthier e Miguel Araújo. Aos amigos que fiz na vida, da infância ao ensino médio, e que ainda continuam presentes: Jonathan Coutinho, Pedro Furtado, Filipe Rudrigues, Iohanna Melo, Rayza Cecília, Hanna Matubara, Débora Bezerra e Victor Gomes. Tenho certeza de que com vocês a vida ficou mais fácil e prazerosa.

Aos meus colegas de trabalho, em especial a Airton Sobral por me introduzir ao “mova-se rápido, quebre coisas e aprenda”.

Finalmente, agradeço ao Centro de Informática da UFPE e todos seus funcionários pela dedicação diária em fazer do CIn um espaço de destaque no ensino e pesquisa em Ciência da Computação. Especialmente, agradeço a Henrique Rebêlo, por ter me orientado durante o desenvolvimento deste trabalho, e a tantos outros professores que por vários momentos foram fonte de inspiração.

Índice

1. Introdução

Com o surgimento da programação orientada a objetos, cresce naturalmente uma preocupação a respeito da confiabilidade dos sistemas escritos neste paradigma, onde por confiabilidade entende-se a combinação de corretude e robustez: a ausência de bugs (MAYER, 1992). Essa preocupação é justificada por uma das ideias centrais do paradigma, o reuso. A reutilização de um componente não confiável em larga escala afeta não apenas o próprio componente, mas o funcionamento de vários sistemas. Desta forma, para abordar sistematicamente essa questão, surge a teoria de *Design by Contract* (DbC).

O *Design by Contract* se desenvolve como uma nova forma de tratar a confiabilidade. Ele introduz o conceito de contrato, um conjunto de restrições de comportamentos que devem ser atendidas antes, durante e após a execução de uma determinada rotina, através das pré-condições, declarações de invariante e pós-condições. Se explorarmos um pouco esses conceitos, nas pré-condições temos a definição das propriedades que devem ser asseguradas por quem chama uma determinada rotina para que ela possa ser corretamente executada, a exemplo de um parâmetro de entrada que deve ter um valor válido. Já nas pós-condições temos a definição de um conjunto de restrições que devem ser garantidas pela rotina após a sua execução, a exemplo da real atualização de determinado atributo. Por fim, nas declarações de invariante temos as propriedades que devem ser sempre verdadeiras independente da chamada (MAYER, 1992).

A ideia de fazer a verificação dos contratos em tempo de execução ficou popular nos anos 80 com o *Eiffel* (MEYER, 1992), ganhando ao longo dos anos representatividade por meio de várias outras linguagens de *DbC* como o *PyContract* (WEBB, 2010) e o *JML* (LEAVENS, 2010), sendo esta última uma especificação para Java. Ao passo que essas linguagens se desenvolviam, por outro lado, a literatura reconheceu que o contrato seria um interesse transversal, sendo melhor modelado por meio de aspectos (FELDMAN, 2006). Tomaremos aqui como base o universo Java e, por isso, analisaremos a implementação de contratos modelados como aspectos através da extensão mais famosa de Java ao paradigma, o *AspectJ* (KICZALES, 2001). Modelar contratos com *AspectJ*, entretanto, traz consigo alguns novos problemas.

Um primeiro problema seria o raciocínio modular (REBELO, 2014). O raciocínio modular se caracteriza pela capacidade de tomar decisões a respeito de um módulo

olhando apenas para sua implementação. Por sua própria estrutura, os *adivices* em *AspectJ* são aplicados sem explicitamente serem referenciados por um módulo, afetando o raciocínio modular. O raciocínio modular, entretanto, é um princípio defendido pelo *Design by Contract* (MAYER, 1992) e por isso um impasse é criado. Ao modelar contratos como um interesse transversal através do *AspectJ* acabamos contrariando uma primitiva do *DbC*.

Um segundo problema seria a nível de documentação (REBELO, 2014). Por via de regra, a exemplo de linguagens como *Eiffel* e *JML*, as pré-condições, pós-condições e declarações de invariante são declaradas em, ou próximas, o código que elas estão especificando, aumentando a documentação do sistema. Em *AspectJ* isso não é possível.

Para resolver esses problemas, Rebêlo et al. propuseram o *AspectJML* (REBELO, 2014), uma extensão do *JML* ao paradigma orientado a aspectos, mantendo a separação de interesses e evitando incorrer nos dois problemas acima citados.

1.1. Caracterização do Problema

Se o *AspectJML* consegue modelar a ideia de contrato como interesse transversal ao mesmo tempo que respeita premissas fundamentais do *Design by Contract*, as ferramentas de desenvolvimento para a linguagem ainda são limitadas quando comparadas ao *JML* ou ao próprio *AspectJ.*

Ao que desejar compilar aplicações *AspectJML*, existem algumas opções:

1. Baixar o compilador *AspectJML* e utilizar-se do terminal do sistema operacional para executar tarefas de compilação.
2. Baixar o compilador *AspectJML* e instalar o *Apache Ant*, adicionando uma camada de abstração sobre as chamadas às rotinas de compilação. Nesse caso, torna-se possível integrar tarefas as *Ant* com *IDEs* já estabelecidas, como o *Eclipse*.

Ao integrar tarefas *Ant* com o *Eclipse*, esta opção demonstra-se ser a mais produtiva para o desenvolvedor, por fornecer um ambiente de edição de código junto com o ambiente de compilação, tirando proveito de funcionalidades da *IDE*.

Essa abordagem, porém, possui seus próprios problemas. Dado que a integração com o *Eclipse* ocorre apenas através de tarefas *Ant,* recursos como *destaque de sintaxe* de palavras chaves do *AspectJML* não fazem parte do ambiente de desenvolvimento.

Outro problema é a complexidade para a configuração de um ambiente próprio para o desenvolvimento. Os passos necessários para desenvolver *AspectJML* junto ao *Eclipse* envolvem a resolução dependências como versão do *Java*, *Ant* e Eclipse além

do próprio compilador *AspectJML*. Esse cenário, apesar de factível em computadores pessoais, é um impeditivo quando a intenção é que o *AspectJML* seja utilizado em ambientes corporativos, tais quais laboratórios de pesquisas ou salas de aula em universidades.

Como exemplo da última limitação apresentada, no semestre 2017.1, após atualização do *Eclipse* nos laboratórios do Centro de Informática da Universidade Federal de Pernambuco para a versão mais recente, em aula do professor Henrique Rebêlo foi observado incompatibilidade entre a nova versão da *IDE* e o plug-in necessários para a integração com o *AspectJ,* dependência do *AspectJML*. Na ocasião foi necessário retroceder a versão do Eclipse para manter o ambiente funcionando.

Um segundo exemplo de limitação ocorreu com a professora Ana Cavalcanti da *University of York*, Inglaterra, onde após a atualização da máquina virtual Java, nos laboratórios da universidade, para versão 1.8, houve quebra de compatibilidade com o compilador *AspectJML*.

Como pode ser visto nos dois exemplos acima, vários problemas ocorrem no dia-dia que quem desenvolve utilizando a linguagem. Fica destacado que a disseminação do *AspectJML* em sala de aula é restrita a ambientes de laboratório que tiveram o compilador e ferramentas correlatas previamente instaladas. O processo de configuração desse ambiente quase sempre passa por solicitações ao time de administração de sistemas que, em seu próprio tempo, faz ou não a configuração. Ao mesmo tempo o desenvolvimento da linguagem *Java*, do compilador *AspectJML*, da *IDE* e de plug-ins pode caminhar em velocidades distintas, sendo necessário por vezes solicitações de atualização de versão ou solicitações de diminuição de versão de algum dos softwares envolvidos.

Para se ensinar *AspectJML* de maneira mais orgânica, é possível imaginar um ambiente leve e altamente disponível, onde alunos facilmente consigam compilar e executar suas aplicações sem se preocupar com tarefas transversais como a instalação do compilador ou versão dos plug-ins.

1.2. Objetivos

Este trabalho, então, assume a responsabilidade de estender o *AspectJML* a categoria de serviço, fazendo uma contribuição à linguagem por tornar possível que códigos *AspectJML* sejam compilados e executados remotamente através da *World Wide Web*.

Para atingir o objetivo previamente exposto, fragmenta-o nos seguintes objetivos específicos:

- Revisão da literatura a fim de entender o atual cenário do *Design by Contract* e a contribuição deste trabalho.
- Apresentar uma abordagem para resolução do problema exposto.
- Demonstrar a aplicação desenvolvida, ressaltando quais problemas foram efetivamente resolvidos e quais permanecem em aberto.

1.3. Metodologia

Para atingir os objetivos propostos, o método de pesquisa pode ser dividido como representado na Tabela 1: revisão da literatura, proposta da solução, demonstração da solução.

Etapa 1: Revisão da literatura
• Entender o surgimento do *Design by Contract*
• Investigação a respeito do estado atual do paradigma
Etapa 2: Proposta da solução
• Definição das etapas da solução proposta
Etapa 3: Demonstração da solução
• Aplicação demonstrando a resolução dos problemas encontrados que serviram de motivação para o surgimento deste projeto

Tabela 1: Método de pesquisa
Fonte: Autor

1.4. Organização do trabalho

O trabalho apresenta-se organizado em 5 capítulos, apresentados a seguir, com exclusão da introdução:

- Capítulo 2 – Fundamentação teórica: Neste capítulo são descritos os principais conceitos para o entendimento da abordagem estabelecida neste trabalho.
- Capítulo 3 – Abordagem proposta: Se faz a apresentação da abordagem estabelecida neste trabalho para a resolução dos problemas previamente expostos.
- Capítulo 4 – Demonstração da solução: Se demonstra a solução com base na proposta definida durante o Capítulo 3.
- Capítulo 5 - Conclusões: São apresentadas as conclusões finais ao desenvolvimento deste trabalho, ressaltando contribuições, limitações e trabalhos futuros.

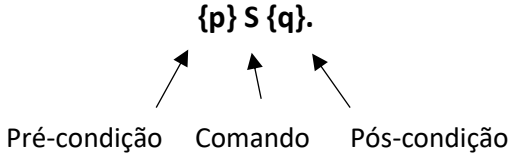
2. Fundamentação teórica

Nesta secção será exibido brevemente o contexto teórico deste trabalho. São tópicos a serem abordados: a confiabilidade de sistemas, os modelos de desenvolvimento e o *Design by Contract* (*DbC*). Desta forma, o principal objetivo deste capítulo é fornecer um entendimento geral do contexto histórico que levou ao surgimento do *AspectJML*. Provido dessa visão torna-se possível entender a conjuntura e as contribuições que aqui são apresentadas.

2.1. Uma breve introdução à confiabilidade de sistemas

Uma das principais preocupações de um projetista de sistemas é ter certeza de que o que está sendo projetado vai satisfatoriamente resolver o problema proposto: ele será tanto correto quanto completo. Essa preocupação é ainda maior no desenvolvimento de sistemas complexos - onde erros são muito mais susceptíveis por conta da sua complexidade inerente, e onde falhas geralmente não são toleradas - ou no desenvolvimento de bibliotecas de software, onde, por meio do reuso, um sistema torna-se subsistema de vários outros (MAYER, 1992).

Dos trabalhos pioneiros a respeito da confiabilidade de sistemas, Hoare, no artigo *An axiomatic Basis for Computer Programming* (HOARE, 1969), lança fundamentos lógico/ matemáticos que visam auxiliar na verificação da corretude de um programa. Para Hoare, essa verificação ocorre por meio da definição de axiomas e regras de inferência, visando encontrar a satisfabilidade de um conjunto de propriedades. Um conceito fundamental da sua teoria é a chamada tripla de Hoare, representando como a execução de parte do código afeta seu estado de execução; um sistema formal para raciocinar programas computacionais. Para um maior entendimento sobre o assunto, observe a expressão a equação a seguir:



Equação 1: Demonstração da tripla de Hoare

Ele indica que a partir de um estado no qual o predicado *p* é verdadeiro, se o programa *S* terminar, ele resultará em um estado em que o predicado *q* também é verdadeiro. *p* é chamado de pré-condição ao passo que *q* é conhecido como uma pós-condição.

Tópico a parte e de complexidade suficiente para um estudo próprio e aprofundado, a lógica de Hoare apresentou-se frutífera para diversas áreas da computação, incluindo o *Design by Contract*, que será abordado posteriormente.

Sete após a publicação do *An axiomatic Basis for Computer Programming* (HOARE, 1969), por meio do livro *A Discipline of Programming* (DIJKSTRA, 1976), houve o amadurecimento dos modelos iniciais da confiabilidade de sistemas. A ideia de pré-condições, invariantes e pós-condições – como predicados sob um estado, torna-se por vez consolidada (JONES, 2003). Estava definida uma base teórica para a área da Semântica Axiomática.

2.2. O modelo: Aplicando um desenvolvimento mais formal

Uma vez que a incipiente área da Semântica Axiomática estava gerando seus primeiros frutos, uma outra área estudava como aplicar essas técnicas formais no ciclo de desenvolvimento de um software. Se a base das técnicas formais estavam, como vimos, bem definidas, em meado dos anos 80 a forma de como aplicar essas técnicas durante o ciclo de desenvolvimento de um software ainda era tópico em aberto. Se fazia necessário encontrar uma maneira de tornar sistemático o uso desse conhecimento.

2.2.1. Sistematizando o processo de desenvolvimento: o formal e o informal

Após uma primeira análise, uma noção detalhada e pragmática a respeito do desenvolvimento de um software é mais escarça do que parece. Desenvolver um sistema computacional pode ser sinônimo de um processo formado de várias fases, tais como a elicitação de requisitos, seguido por uma fase de projeto e uma fase de desenvolvimento *per se*, ao mesmo tempo em que pode ser sinônimo de apenas uma fase codificação, sem muito planejamento, seguido por uma fase de implantação. Essa sistematização do ciclo de desenvolvimento de um sistema computacional é chamada de modelo de desenvolvimento. Vários foram os modelos propostos desde os fundamentos da ciência da computação, tendo o modelo cascata (ROYCE, 1970) se tornado canônico e bastante popular até a década de 90. Sua importância é explicita

de tal forma que ele tem sido profundamente estudado e disseminado na literatura da engenharia de software.

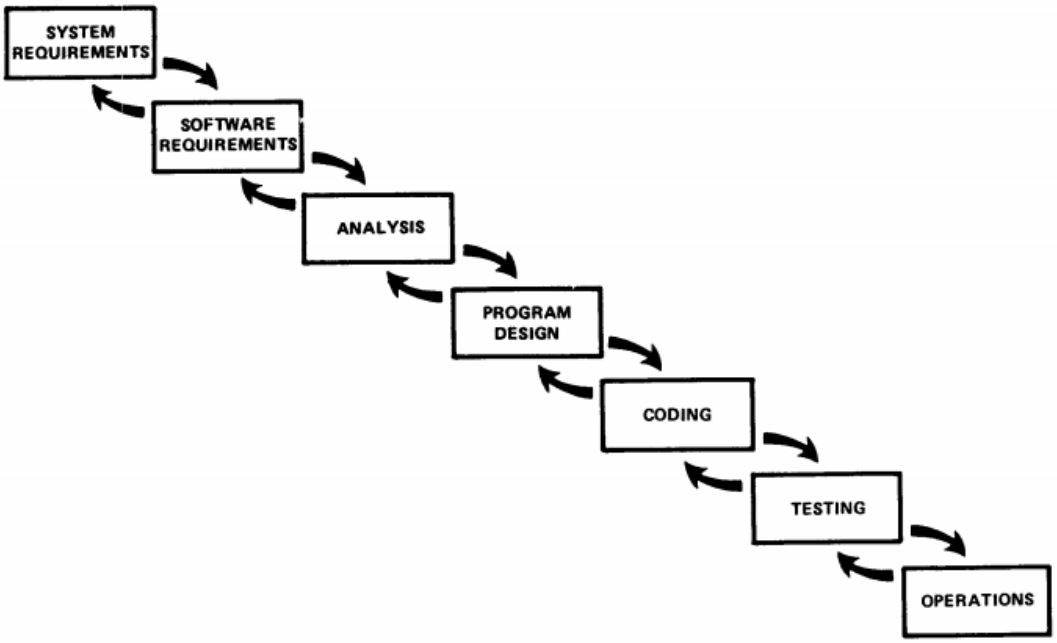


Figura 1: Modelo cascata.
Fonte: MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS. Dr. Winston W. Rovce. 1970

É possível caracterizar o modelo cascata por meio de sete fases:

- Levantamento dos requisitos do sistema
- Requisitos do software
- Design preliminar (ou análise)
- Design detalhado
- Codificação
- Testes
- Operação
- Manutenção

Cada uma dessas fases se comunica estritamente com sua predecessora e sua sucessora, numa espécie de cascata, através de critérios de validação. Sendo assim, aquele que aderir ao modelo cascata para o desenvolvimento de um sistema deve seguir suas fases propostas com o intuito de eventualmente atingir o objetivo final, o software e sua manutenção.

Com o desenvolvimento da Semântica Axiomática e da intenção de produzir sistemas seguindo certos formalismos, foi preciso também sistematizar o processo de criação de softwares que desejam ser verificados por meio de axiomas, tal qual o cascata modela e sistematiza as etapas de um desenvolvimento não formal. Dessas primeiras tentativas temos uma proposta por *Cohen et. Al*, conhecida por Modelo Contratual (BOEHM, 1976).

2.2.2. O modelo contratual:

O modelo contratual é assim chamado por considerar que as etapas durante o processo de desenvolvimento são regidas por um contrato entre duas partes - sempre há um cliente e um fornecedor. A definição de quem é cliente e quem é fornecedor pode variar ao longo do processo. A *tabela 2* exemplifica esse conceito para um modelo de desenvolvimento formado pelas fases de especificação, projeto e implementação.

Fase	**Cliente**	**Fornecedor**
Especificação	Usuário final do software	Engenheiro de Software/ Analista de Sistemas
Projeto	Engenheiro de Software/ Analista de Sistemas	Analista de Projetos
Implementação	Analista de Projetos	Programador

Tabela 2: Exemplificação de contratos entre cliente e fornecedor, duas partes envolvidas no processo
Fonte: Material de ensino do professor Augusto Sampaio

Durante todo o fluxo, em cada fase, há o reconhecimento por parte do *cliente* daquilo realizado pelo *fornecedor*, verificando o cumprimento do contrato previamente estabelecido.

Como característica do modelo, a ideia de contrato como um conjunto de obrigações a serem satisfeitas e respeitadas por ambas as partes envolvidas, cliente e fornecedor, ocorre durante as várias fazes que o modelo propões: análise de requisitos, projeto e implementação.

Na figura 2 é possível observar mais detalhadamente cada uma das fases do modelo contratual, ao mesmo tempo que é possível ter uma visão abrangente sobre o modelo.

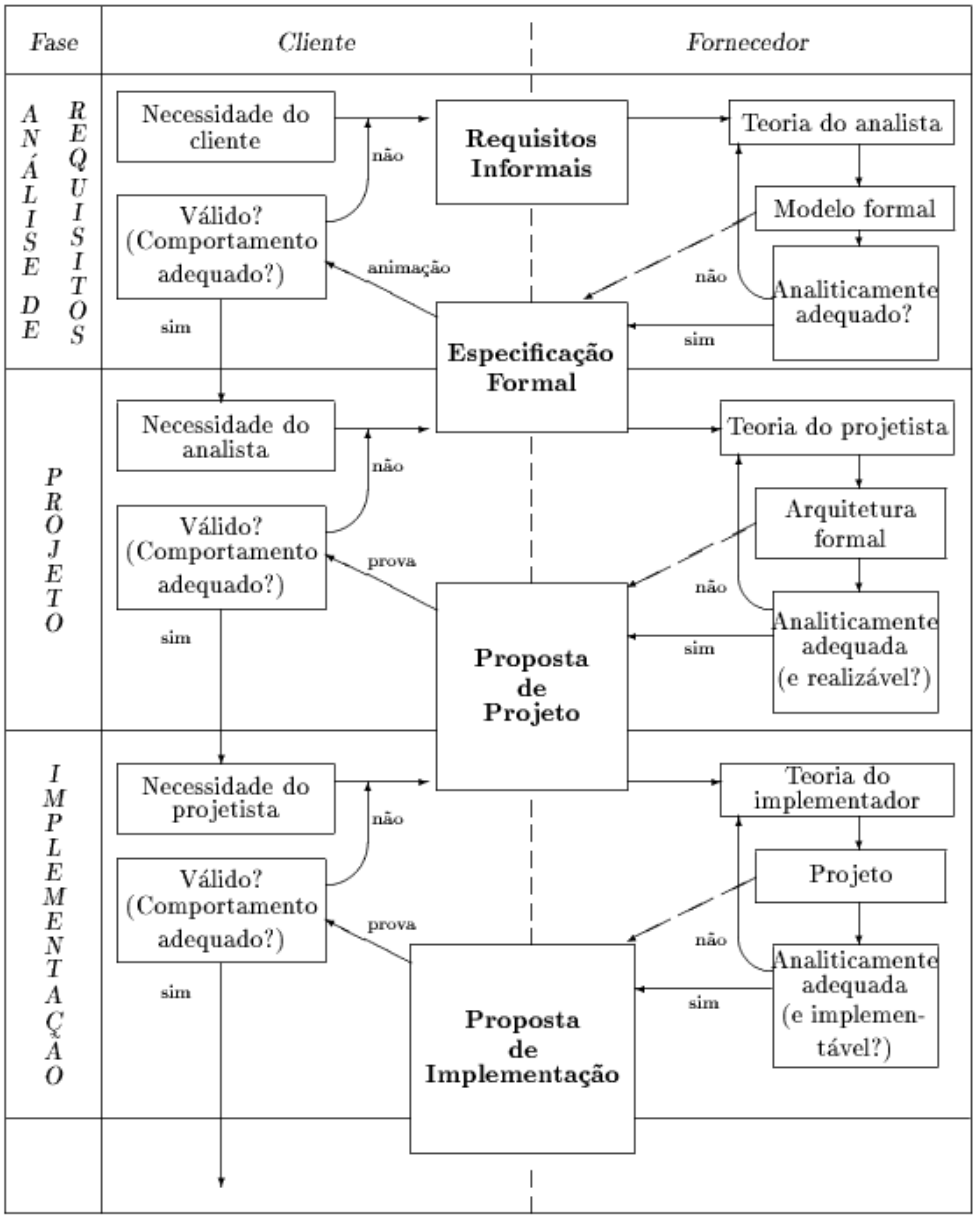


Figura 2: Modelo Contratual
Fonte: Material de ensino do professor Augusto Sampaio

2.3. O *Design by Contract*

O modelo contratual, explorado anteriormente, ainda está longe de ser amplamente utilizado no processo de desenvolvimento por grande parte da indústria. Baseando-se na ideia do *bom o suficiente*, pode ser mais vantajoso aplicar a formalidade necessária apenas durante a fase de implementação, não fazendo uso do conceito de contrato na *análise de requisitos* ou *projeto*, por exemplo.

Assim, seja por sua simplicidade ou não, *o Design by Contract* pode ser visto como uma forma de aplicar as técnicas desenvolvidas por *Floyd*, *Hoare* e *Djikstra*, para garantir confiabilidade, de maneira parcial, durante a fase da codificação do sistema.

Logo em seguida será tanto possível explorar alguns conceitos do *Design By Contract* como também observar que, apesar de restrito a fase de codificação, ele está profundamente influenciado pela Semântica Axiomática.

2.3.1. Assertions: Contratos para software

Se imaginarmos a implementação de uma unidade de rotina computacional, não trivial, muito provavelmente teremos a subdivisão dela em várias outras sub-rotinas.

```
1  def rotina_principal:
2      subrotina_1()
3      subrotina_2()
4      subrotina_3()
5      ...
6      subrotina_n()
7  end
```

Figura 3: Chamadas de sub-rotinas pela rotina principal
Fonte: Autor

```
1  def subrotina_1:
2      .....
3      .....
4  end
```

Figura 4: Sub-rotina chamada por rotina principal
Fonte: Autor

Cada chamada a uma dessas sub-rotinas pode ser mapeada para uma rotina unitária. Sendo assim, sobre a forma de contratos, é possível que o relacionamento entre a rotina que chamou e a rotina chamada obedeça certas restrições em tempo de execução.

```
1   class Pessoa{
2       private int idade;
3
4       // Este método deve garantir que quem o
5       // chamar sempre retornará um valor maior que 0
6       private int getIdade() {
7           return this.idade;
8       }
9
10      // Quem chamar esse método deve passar
11      // um valor maior que 0 como paramêtro
12      private void setIdade(int idade) {
13          this.idade = idade;
14      }
15  }
```

Figura 5: Exemplificando contratos
Fonte: Autor

O ato de uma sub-rotina esperar que a rotina principal atenda determinados predicados durante a chamada é conhecido como pré-condição. Já o fato de esperar que a sub-rotina cumpra seu papel respeitando determinadas condições de saída é chamado de pós-condição. Durante todo o processo algumas propriedades devem se manter, não sendo modificadas. Nesse caso temos a noção de invariantes de classe.

Com forte referência a *Tripla de Hoare*, a principal inovação do paradigma está em fazer a verificação de contratos em tempo de execução, o que se deu pioneiramente por meio da linguagem de programação chamada *Effel* (MEYER, 1992).

Na tabela abaixo é possível observar os deveres e benefícios por diferentes partes ao se trabalhar com a noção de contratos, exemplificando a *figura 5*:

Parte	**Obrigações**	**Benefícios**
Cliente	Utilizar um argumento maior que 0, representando uma determinada idade	Obtém a atualização da idade para pessoa em destaque Não tem necessidade de checar se o parâmetro realmente foi atualizado
Fornecedor	Atualiza a idade para uma determinada pessoa	Não tem necessidade de checar se o parâmetro fornecido pelo cliente é valido

Tabela 3: Obrigações e benefícios através de contratos em referência a Figura 5

2.3.2. JML

JML é uma das principais implementações de *DbC* para o universo Java. Nela é possível expressar requisitos de execução da aplicação através de comentários de anotações (*comment anotations*).

```
1    //@ requires idade > 0;
2    // ...
3    public void setIdade(int idade) {
4        // ....
5    }
```

Figura 6: Notação em JML
Fonte: Autor

Em *JML*, tudo entre /*@ e @*/ será interpretado pelo compilador como pertencente a sintaxe do próprio *JML*, assim como comentários individuais anotados com //@

JML também se utiliza da palavra reservada ***requires*** na definição de pré-condições, obrigações por parte do cliente, e ***ensures*** na definição de pós-condições, obrigações por parte do fornecedor. Caso se faça necessário a verificação do retorno de determinado método, a palavra reservada ***\result*** representa esse valor retornado.

2.3.3. AspectJML como o melhor dos dois mundos

Com o desenvolvimento da programação orientada a aspectos, a literatura reconheceu que a noção de contrato seria um interesse transversal, sendo melhor modelado por meio de aspectos (FELDMAN, 2006).

Um primeiro problema de utilizar o conceito de contrato através do paradigma orientado a aspectos, entretanto, está no raciocínio modular (REBELO, 2014). O raciocínio modular pode ser entendido pela capacidade de o desenvolvedor tomar decisões a respeito de um módulo olhando apenas para sua implementação, aquilo que ele enxerga. Por sua própria estrutura, os *adivices* em *AspectJ* são aplicados sem explicitamente serem referenciados por um módulo, afetando o raciocínio modular. O raciocínio modular, entretanto, é um princípio defendido pelo *Design by Contract* (MAYER, 1992) e por isso um impasse é criado. Ao modelar contratos como um interesse transversal através do *AspectJ* acabamos contrariando uma primitiva do *DbC*.

Um segundo problema seria a nível de documentação (REBELO, 2014). Por via de regra, a exemplo de linguagens como *Eiffel* e *JML*, as pré-condições, pós-condições e declarações de invariante são declaradas em, ou próximas, o código que elas estão especificando, aumentando a documentação do sistema. Em *AspectJ* isso não é possível. O *AspectJMl* resolve, então, esses problemas, trazendo consigo o melhor do *AspectJ* e o melhor do paradigma orientado a aspectos.

3. Abordagem proposta

Dado todo o universo de vantagens possibilitado pelo *Design By Contract*, as ferramentas para o desenvolvimento de aplicações no paradigma são por hora restritas ou deficientes. Considerando o *AspectJML* como o expoente mais promissor do *DbC*, capaz de modelar contratos como interesse transversal sem romper com premissas básicas do *Design by Contract*, a falta de ferramentas para o desenvolvimento de aplicações na linguagem acaba se tornando uma barreira para um uso mais abrangente.

Como já enunciado previamente, este trabalho se propõe, então, a resolver essas limitações. A fim de que o *AspectJML* seja amplamente utilizado sobre quaisquer plataformas, em qualquer região do mundo e com o mínimo esforço, encontrou-se o desenvolvimento de uma *IDE online* como principal maneira de resolver o problema.

3.1. Vantagens de uma plataforma de compilação online

As principais vantagens de se ter uma *IDE online* para o desenvolvimento de aplicações *AspectJML* podem ser condensadas como:

1. Quebra da barreira para desenvolver primeira aplicação na linguagem: Um dos principais problemas presentes do *AspectJML* se dá na dificuldade de instalação do compilador e das dependências necessárias para a execução do projeto. Através de um editor *online*, a criação, a compilação e a execução do código seria possível de forma descomplicada, nas premissas de *Code as a Service*.
2. Facilidade no ensino da linguagem: O *AspectJML* é uma linguagem base para o ensino do *Design by Contract*. Uma das principais queixas de professores que a utilizam em suas aulas é justamente a dificuldade de criação de um ambiente onde os alunos possam fazer uso do compilador. Uma interface *WEB* para edição, compilação e execução de código online sanaria esse problema e possibilitaria uma maior disseminação do paradigma.
3. Maior velocidade de desenvolvimento da linguagem: Ao se utilizar um modelo descentralizado onde cada usuário é responsável por montar seu próprio ambiente de desenvolvimento, correções de bug e adição de novas funcionalidades só são

disseminadas de forma ativa quando o usuário percebe uma nova versão do compilador está disponível, reconfigurando então seu ambiente para essa nova versão. Através de uma interface *WEB* as atualizações podem ser feitas de maneira automática, diminuindo significativamente o trabalho na manutenção de um ambiente de desenvolvimento da linguagem, ao mesmo tempo que garante total compatibilidade entre suas dependências.

3.2. Principais características para um ambiente de desenvolvimento on-line:

A partir das vantagens de uma plataforma de compilação *online* é possível extrair um conjunto de características que são fundamentais para a total resolução dos problemas apresentados:

1. A *IDE* deve estar presente através da *World Wide Web*, sendo disponível para estudantes, entusiastas e profissionais que desejam editar, compilar e executar problemas *AspectJML*.
2. A *IDE* deve mostrar o resultado de compilação do programa de entrada que foi codificado usando a própria *IDE*.
3. A *IDE* deve mostrar o resultado de execução do programa de entrada que foi codificado usando a própria *IDE*.
4. A *IDE* deve possibilitar o *download* do código que foi editado na própria *IDE*.
5. A *IDE* deve possibilitar o *upload* do código que foi baixado por meio da própria *IDE*.
6. A *IDE* deve possibilitar manter uma visão da estrutura do projeto através de uma *TreeView*.
7. A *IDE* deve possibilitar uma visão da estrutura do projeto através do uso de abas.

Em adição a essas funcionalidades fundamentais, algumas outras as complementam e trazem uma maior usabilidade ao projeto:

1. Seleção entre diferentes temas
2. Seleção entre diferentes tamanhos de fonte
3. Possibilidade de esconder ou exibir a *TreeView.*
4. Possibilidade de esconder ou exibir o resultado da compilação e execução.
5. Possibilidade de escolher exibir apenas o resultado da compilação ou apenas o resultado da execução.
6. Possibilidade de baixar tanto o último código no servidor quando o código recém editado pelo usuário que ainda não foi alvo de compilação.

7. Possibilidade de carregar projetos exemplos para finalidade de ensino.
8. Possibilidade de renomear seus arquivos, que são representados tanto por meio de abas quando em uma estrutura de árvore.
9. Possibilidade de nomear e renomear a unidade semântica de todos os arquivos na forma de um projeto.
10. Possibilidade de deletar arquivos e pacotes.
11. Possibilidade facilitar a manipulação de código através de atalhos.

3.2. Definindo os diagramas de caso de uso

Nesta secção serão apresentados alguns diagramas de caso de uso, sendo possível obter uma visão abrangente das funcionalidades previstas pelo *AspectJML Online*. No escopo deste projeto, *UML* foi usado apenas como ferramenta de documentação, nem sempre respeitando o rigor formal da linguagem visual. Além disso, os casos foram subdivididos em figuras distintas por razões didáticas, facilitando a visualização. Apenas serão detalhados, por meio da descrição de casos de uso, os principais.

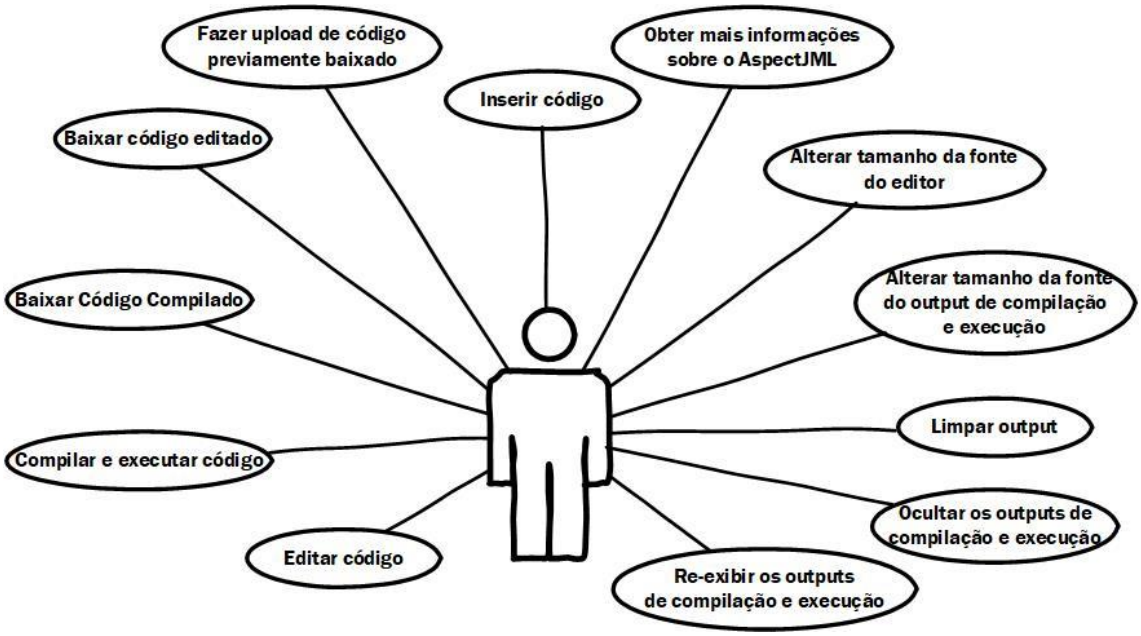


Figura 7: Diagrama de casos de uso, parte 1
Fonte: Autor

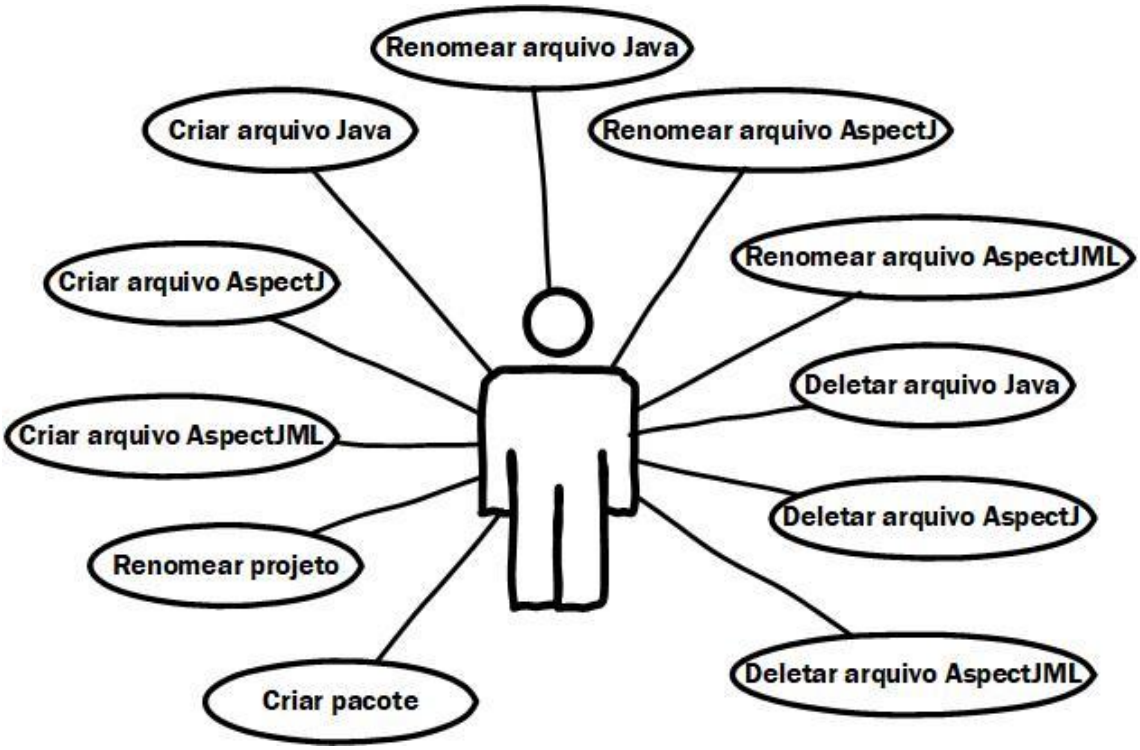


Figura 8: Diagramas de casos de uso, parte 2
Fonte: Autor

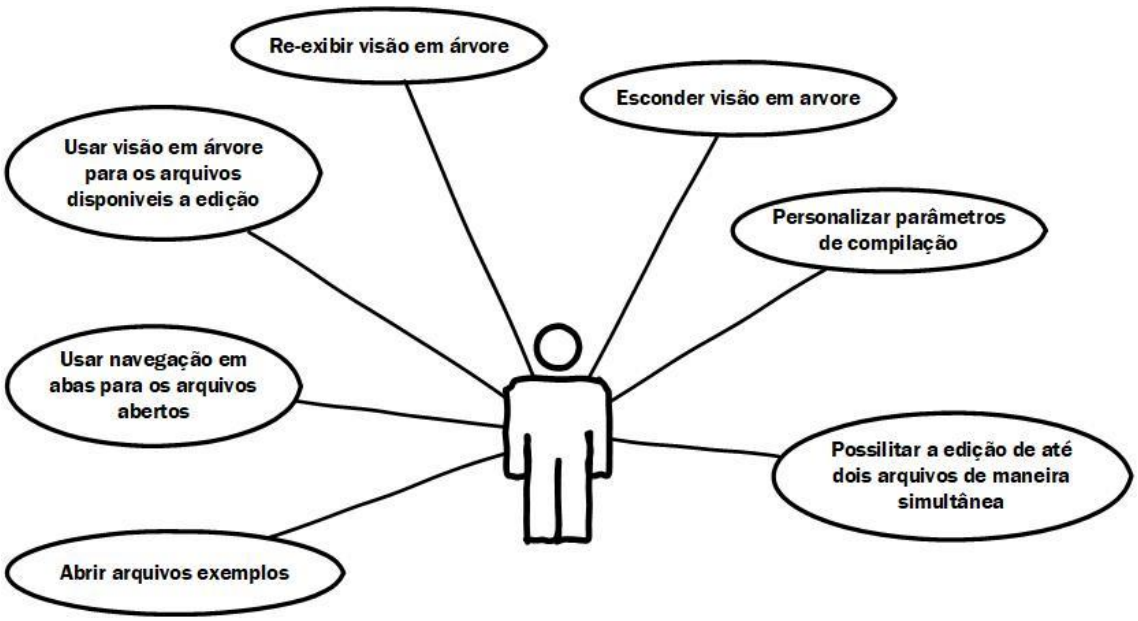


Figura 9: Diagrama de casos de uso, parte 3
Fonte: Autor

Os principais casos de uso podem ser descritos abaixo.

3.2.1. Inserir/ Editar código

Identificação:	[UC001]
Breve descrição:	O usuário acessar o http://aspectjml.online, sendo possível inserir seu próprio código na *IDE*
Pré-condições:	Usuário estar conectado à internet
Ator(es):	Desenvolvedor *AspectJML*
Fluxo principal:	1. O usuário abre seu browser favorito; 2. O usuário acessa o http://aspectjml.online; 3. O usuário clica em criar um novo arquivo (*Java*, *AspectJ* ou *AspectJML*) 4. O usuário insere e edita seu próprio código
Fluxo secundário:	N. A.
Pós-condições:	As alterações do usuário serão mantidas em memória enquanto a página estiver carregada no browser

Tabela 4: Descrição de caso de uso: Inserir/ Editar código

3.2.2. Baixar código

Identificação:	[UC002]
Breve descrição:	O usuário baixa o código editado previamente
Pré-condições:	[UC001]
Ator(es):	Desenvolvedor *AspectJML*
Fluxo principal:	1. O usuário clica no botão de baixar o código editado no browser; 2. Uma cópia de todos os arquivos exibidos na *TreeView* é salva no disco do usuário;
Fluxo secundário:	N. A.
Pós-condições:	Um arquivo *.zip* é baixado pelo navegador

Tabela 5:Descrição de caso de uso: baixar código

3.2.3. Carregar código (upload)

Identificação:	[UC003]
Breve descrição:	O usuário baixa o código editado previamente
Pré-condições:	[UC002]
Ator(es):	Desenvolvedor *AspectJML*

Fluxo principal:	1. O usuário clica no botão carregar um projeto através do menu principal; 2. Ao usuário é solicitado que ele confirme o carregamento de um .zip, deixando claro que todos os arquivos atualmente carregados no browser serão removidos; 3. O usuário seleciona um arquivo .zip previamente baixado através do [UC002];
Fluxo secundário:	1. Se o usuário não confirmar o item principal 2, nenhum arquivo será carregado
Pós-condições:	A interface gráfica é atualizada e os novos arquivos são devidamente carregados na memória do navegador

Tabela 6:Descrição de caso de uso: Carregar código

3.2.4. Compilar/ Executar código

Identificação:	[UC004]
Breve descrição:	O usuário decide compilar e executar o código desenvolvido na *IDE*
Pré-condições:	[UC001]
Ator(es):	Desenvolvedor *AspectJML*
Fluxo principal:	1. O usuário clica em Compilar e Executar Código; 2. O código é compilado e executado;
Fluxo secundário:	N. A.
Pós-condições:	Os resultados tanto da compilação quanto da execução são exibidos em painéis distintos;

Tabela 7: Descrição de caso de uso: Compilar/ Executar código

3.2.5. Edição de arquivos de maneira simultânea

Identificação:	[UC005]
Breve descrição:	O usuário escolhe por exibir, e editar, mais de um arquivo ao mesmo tempo
Pré-condições:	[UC001]
Ator(es):	Desenvolvedor *AspectJML*
Fluxo principal:	1. O usuário seleciona a aba que deseja espelhar; 2. O usuário clica no botão de espelhamento
Fluxo secundário:	N. A.
Pós-condições:	A tela será subdividida de modo que a aba selecionada seja clonada na secção direita da interface gráfica. Após o espelhamento, ao selecionar uma outra aba, a secção esquerda será modificada, enquanto que a secção direita exibirá a aba previamente selecionada.

Tabela 8: Descrição de caso de uso: edição de arquivos de maneira simultânea

3.3. As escolhas de desenvolvimento:

Dado o tamanho do projeto, suas dificuldades inerentes e seus requisitos, foi decido desde o início ter-se uma divisão clara entre *back-end* e *front-end*.

Como a totalidade das interações com o usuário se dá através da interface gráfica, a escolha das tecnologias de *front-end* é crítica para o sucesso ou fracasso deste projeto. Por tanto, foi escolhido *frameworks* robustos e já consolidados a fim trazer velocidade e confiabilidade durante o processo.

3.3.1. Tecnologias front-end:

AngularJS:

Desenvolvido pelo Google, *AngularJS* é um *framework Javascript* que segue o padrão *Model-View-View-Model*, trazendo como principal vantagem a abstração da manipulação do *Document Object Model* (*DOM*), uma árvore hierárquica de objetos a serem interpretados por, por exemplo, o browser.

O AngularJS foi utilizado como principal *framework* no projeto, auxiliando a modelar a complexidade entre e conexão da *View* e dos *Models* que a compõem.

CodeMirror:

Biblioteca Javascript especializada em fornecer uma interface para edição de textos. Ela contém sub-rotinas tanto para manipulação de texto quanto para a exibição dele.

JQuery:

Mais uma biblioteca Javascript capaz de interagir o HTML, a fim de simplificar o uso do Javascript puro.

3.3.2. Tecnologias de back-end

Para o que se refere ao *back-end* da aplicação, duas tecnologias se destacam.

Ruby on Rails:

Ruby foi a linguagem escolhida para o desenvolvimento do *back-end* da aplicação. A principal motivação está no uso de *Rails*, um de seus principais *frameworks* para

aplicações Web. *Rails* possibilita a criação rápida de códigos *WEB* com bastante simplicidade.

Docker:

Uma das principais preocupações durante o processo de desenvolvimento do *AspectJML Online* estava na segurança tanto do servidor quanto do usuário. Eram preocupações, por exemplo:

- O usuário inserir código malicioso capaz de obter o código de outros usuários da plataforma.
- O usuário inserir código malicioso capaz de consumir todos os recursos da máquina e impossibilitar que outros usuários tivessem acesso a plataforma.

Com a ideia de *Containers* trazido pelo universo *Linux*, é possível rodar processos de maneira isolada do sistema de arquivos do host e de outros processos. Junto com a aplicação de políticas de uso de *CPU* e memória, temos um ambiente muito mais seguro para o *AspectJML Online*. Na prática, isso significa que cada submissão de código e cada chamada à rotina de compilação e execução ocorrerá de maneira totalmente isolada de uma outra chamada, não sendo possível uma submissão interagir com o processo da outra ou ver os arquivos gerados pela outra. Dessa forma, é possível atingir o nível de segurança esperado.

Pelo fato do *Docker* ser uma plataforma bastante difundida e estável, ela foi a tecnologia escolhida para aplicar as ideias de containers no escopo deste projeto.

4. Demonstração da solução

4.1. Visão geral

De maneira geral a *IDE* é composta de uma secção central onde o código é editado. Há ainda *Sintaxe Highlight* de palavras chaves do próprio *AspectJML*.

No canto esquerdo é possível encontrar uma *TreeView* contendo os arquivos presentes no editor. Na parte inferior é possível encontrar o resultado da compilação e execução.

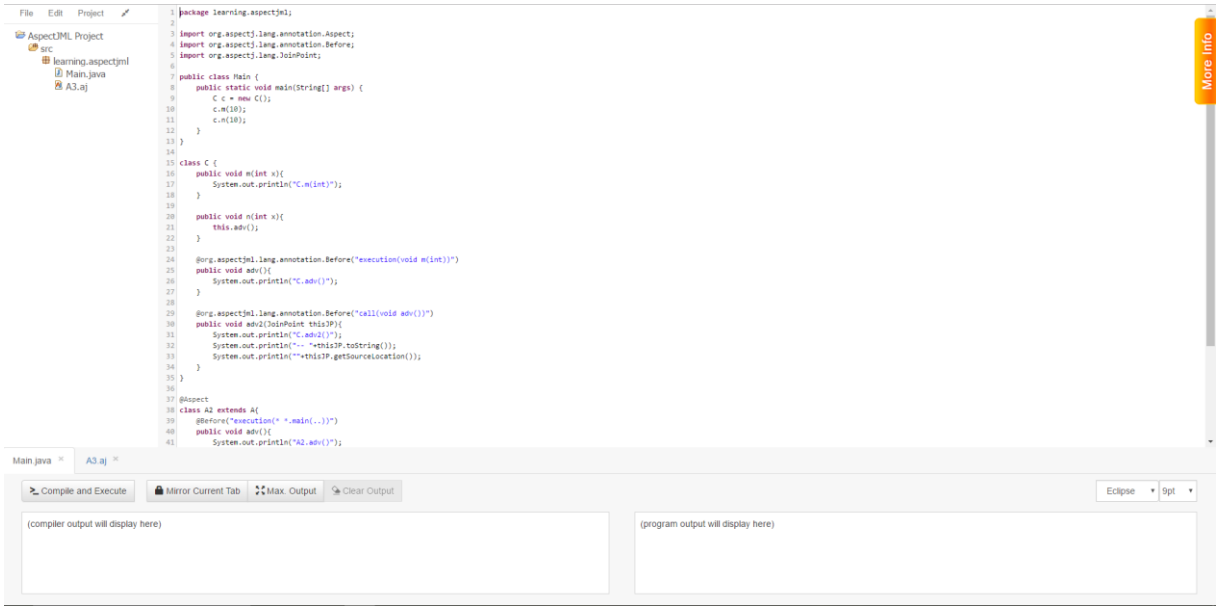


Figura 10: Visão geral da ferramenta desenvolvida

4.2. TreeView

Na *TreeView* é possível selecionar o arquivo a ser manipulado.

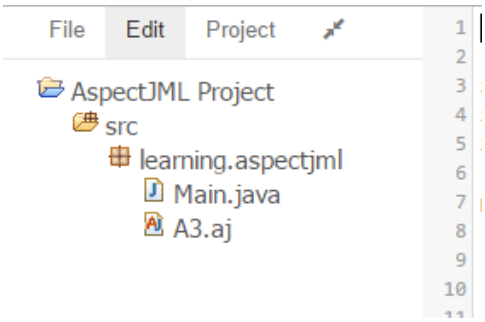


Figura 11: Visão em forma de árvore dos arquivos

Além disso é possível encontrar várias opções, subdividas nos menus *File*, *Edit* e *Project*

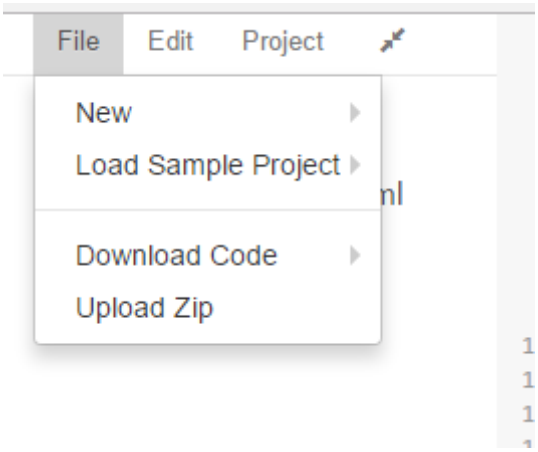


Figura 12: Menu e sub-menu File

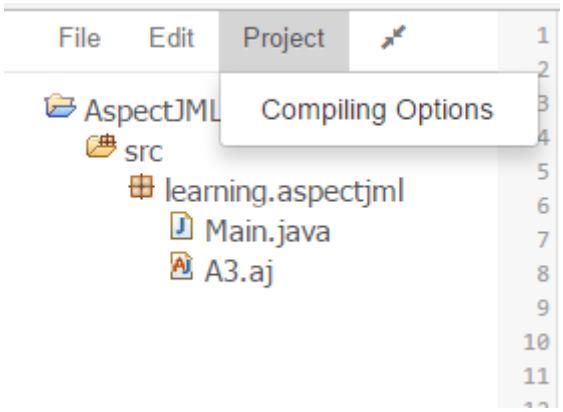


Figura 13: Menu e sub-menu Project

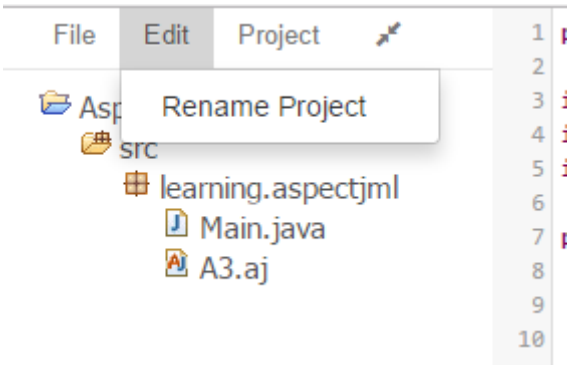


Figura 14: Menu e sub-menu Edit

Também é possível selecionar algumas opções para um arquivo selecionado utilizando-se do botão direito do mouse.

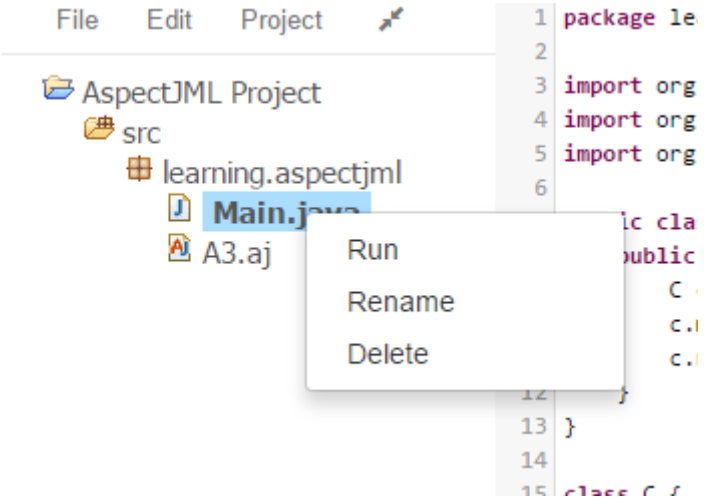


Figura 15: Opções sobre um arquivo na TreeView

4.3. Criação de novos arquivos

Para criar um novo arquivo na *IDE* é possível utilizar-se do menu imediatamente superior à *TreeView*. Uma vez selecionado o arquivo objetivo, é exibido um modal onde o usuário insere dados como o nome do arquivo.

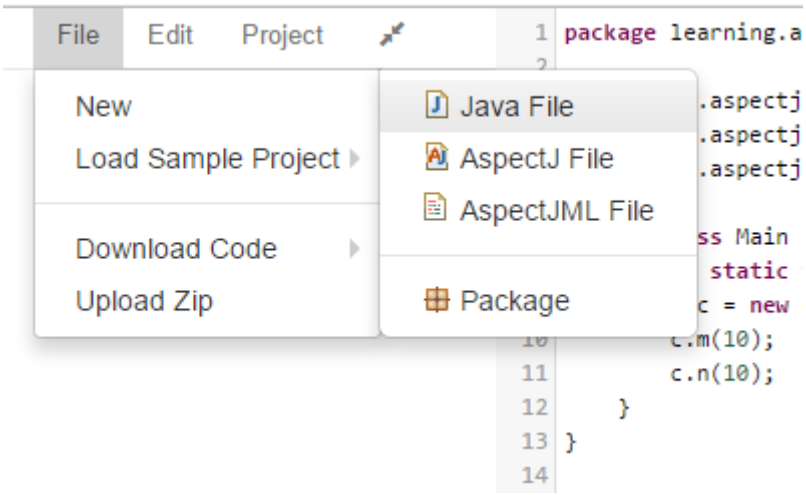


Figura 16: Menu de criação de novos arquivos

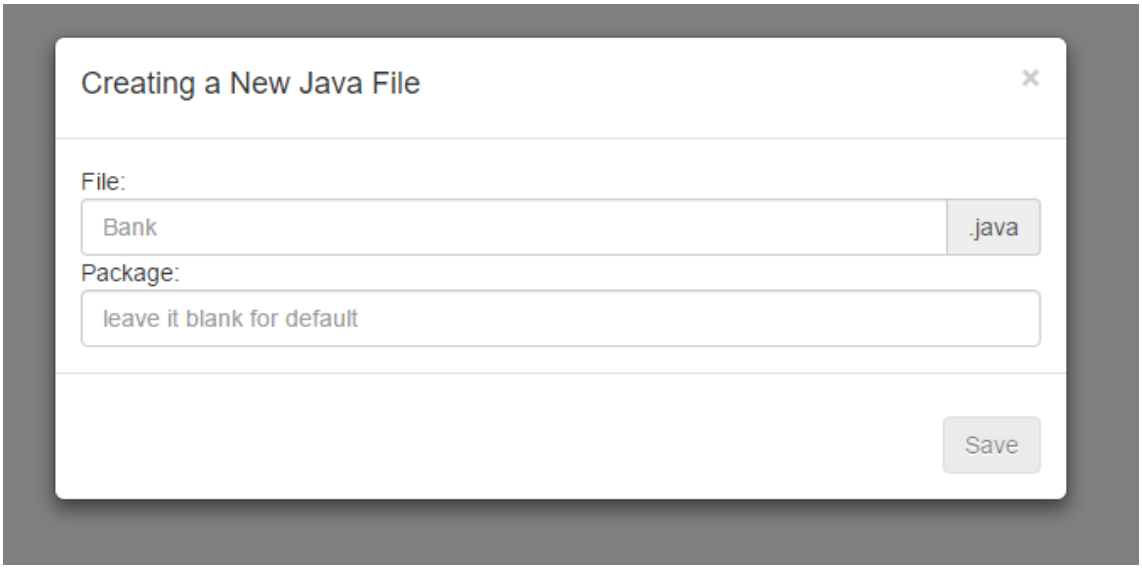


Figura 17: Criação de um novo arquivo Java

4.4. Seleção de fontes

É possível selecionar entre opções pré-estabelecidas de tamanhos de fonte.

A mudança é aplicada tanto para a área de inserção de código quanto para os outputs de compilação e execução

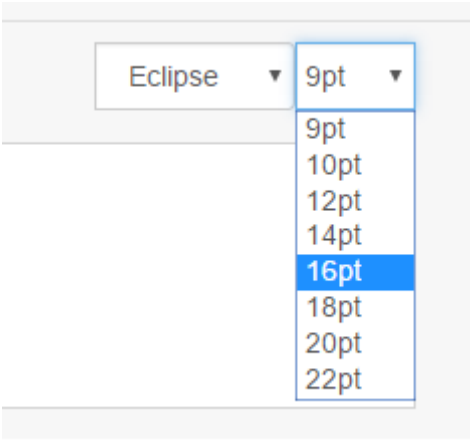


Figura 18: Seleção do tamanho da fonte

4.5. Seleção de temas

É possível selecionar entre opções pré-estabelecidas de temas. São elas *Eclipse*, um tema com uma paleta de cores claras, e *Ambience*, com uma paleta de cores escuras.

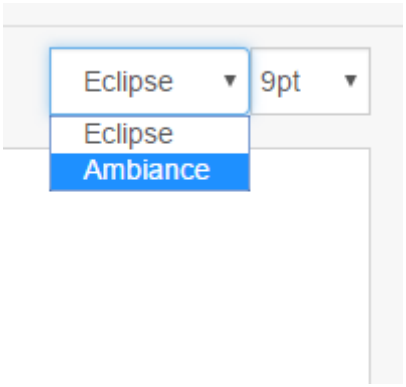


Figura 19: Seleção do tema

```
1  package learning.aspectjml;
2
3  import org.aspectj.lang.annotation.Aspect;
4  import org.aspectj.lang.annotation.Before;
5  import org.aspectj.lang.JoinPoint;
6
7  public class Main {
8      public static void main(String[] args) {
9          C c = new C();
10         c.m(10);
11         c.n(10);
12     }
13 }
14
```

Figura 20: Tema Eclipse

```
1  package learning.aspectjml;
2
3  import org.aspectj.lang.annotation.Aspect;
4  import org.aspectj.lang.annotation.Before;
5  import org.aspectj.lang.JoinPoint;
6
7  public class Main {
8      public static void main(String[] args) {
9          C c = new C();
10         c.m(10);
11         c.n(10);
12     }
13 }
14
```

Figura 21: Tema Ambiance

4.6. Output de compilação e execução

Uma vez que o código é compilado e executado pelo servidor, é possível observar tanto o resultado da compilação quanto da execução.

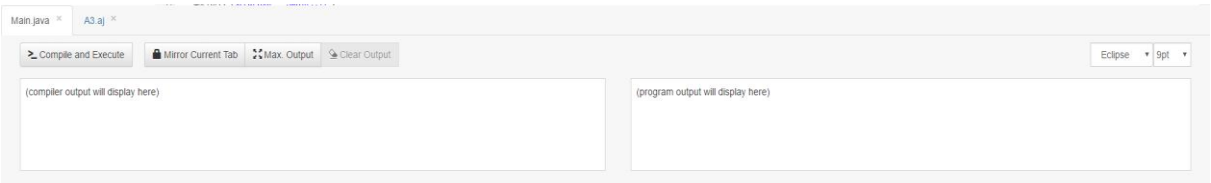


Figura 22: Visão geral da saída de compilação e execução

Além disso é possível selecionar um dos outputs para ocupar tamanho integral da tela.

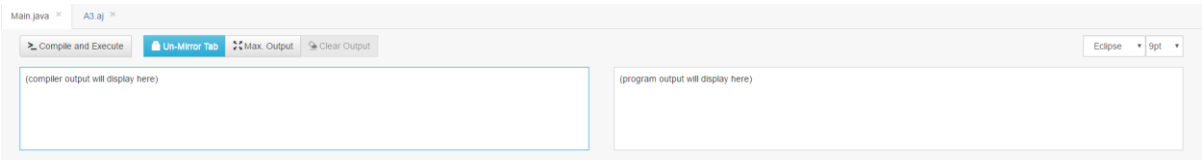


Figura 23: Selecionando a saída de compilação

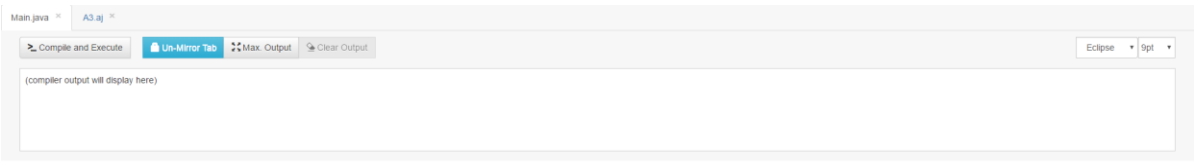


Figura 24: Visualizando apenas a saída de compilação

Uma vez que os resultados da etapa de compilação e execução foram devidamente exibidos para o usuário, é possível remover esses resultados, numa etapa de limpeza. Para isso é utilizado o botão *Clean Output*.

Também pode-se escolher esconder o editor e exibir apenas a secção de output. Isso é possível com o botão *Max. Output*. Uma vez que o *output* é maximizado, o botão adquire uma coloração diferente e seu texto muda para *Min. Output*.

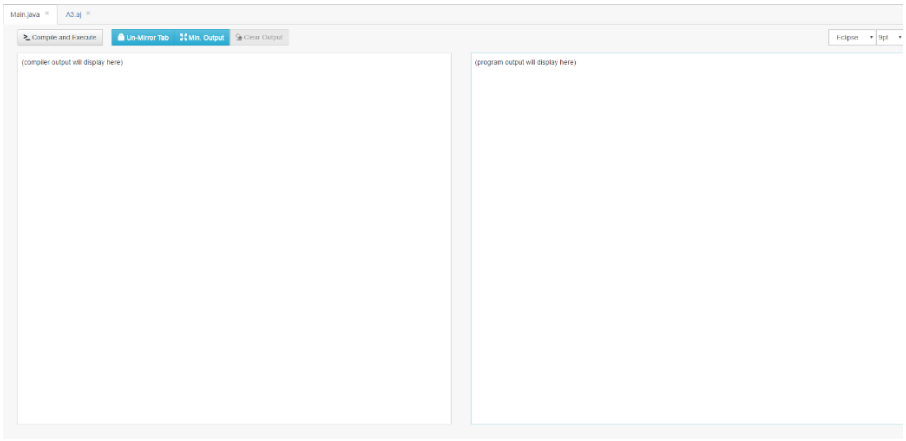


Figura 25: Maximizando secção das saídas de compilação e execução

4.7. Renomeação e deleção

Tanto arquivos quanto pacotes e o próprio projeto podem ser renomeados. Utilizando a *TreeView*, com o botão direito do mouse, ou através do menu, é possível executar tais ações.

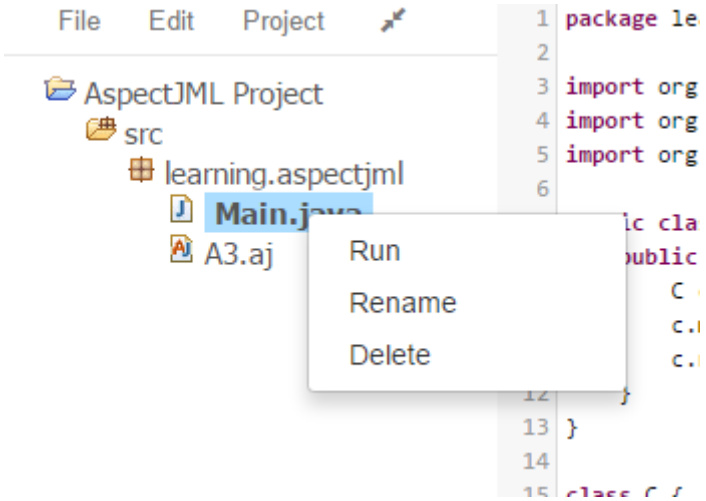


Figura 26: Opções sobre um arquivo na TreeView, selecionando Rename

Figura 27: Diálogo de Raname de um arquivo

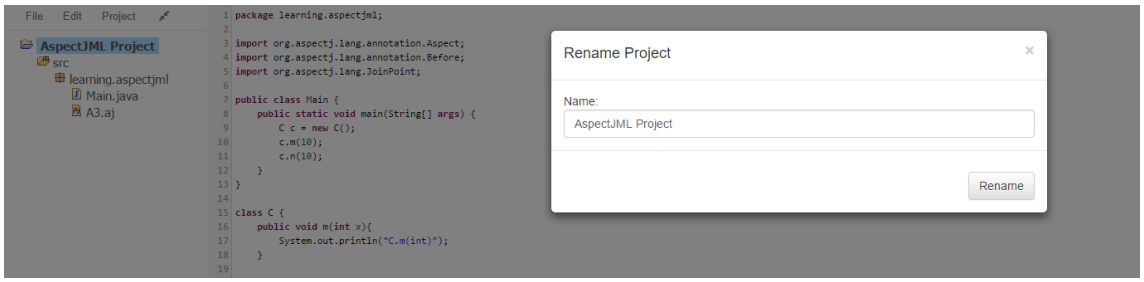


Figura 28: Diálogo de Rename do Projeto

4.8. Download e upload do projeto

Através do menu principal, por meio do sub-menu File, encontra-se a opção de download e upload de arquivos do projeto.

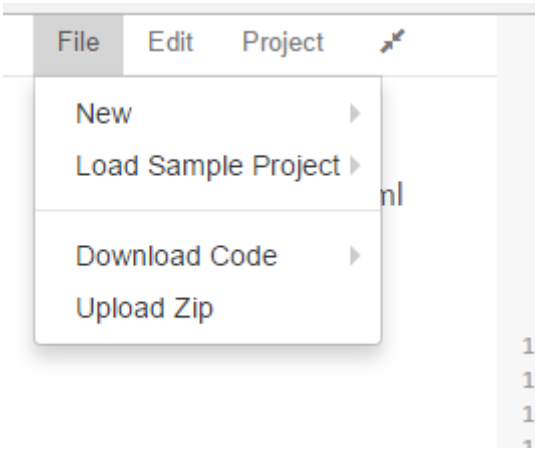


Figura 29: Menu e sub-menu File, a ser utilizado para upload de arquivo

No caso do download existem duas opções disponíveis. A primeira se restringe a baixar exatamente o código que está sendo exibido no browser. A segunda opção, no entanto, baixa os arquivos gerados pelo servidor durante a mais recente compilação. Essa opção só é disponível quando o usuário previamente já compilou e executou seu código.

Uma vez que o arquivo .zip foi baixado com todo o conteúdo e metadados do projeto, é possível, em outro momento ou até mesmo em outro navegador, restaurar a carga de trabalho que foi salva nesse arquivo. Será perguntado se o usuário deseja de fato carregar um novo arquivo, advertindo que os arquivos atuais serão removidos em detrimento dos novos.

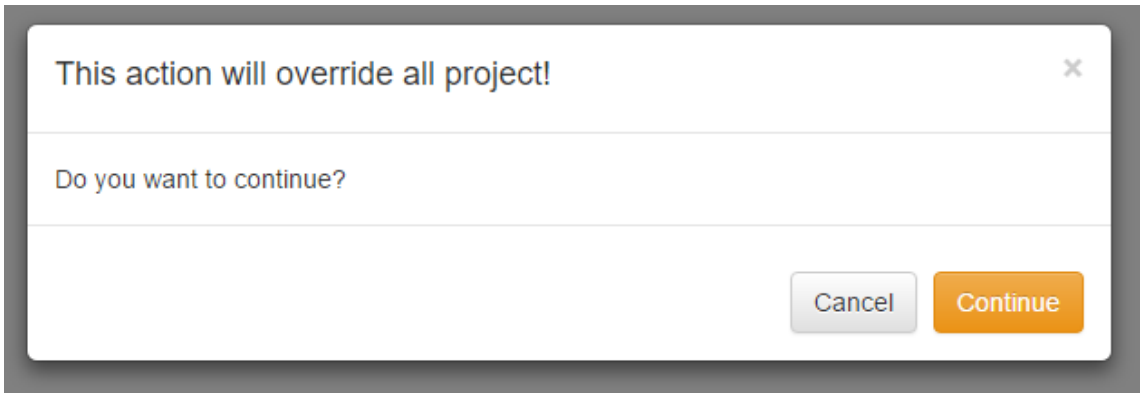


Figura 30: Diálogo para confirmar sobrescrita de arquivos já carregados

4.9. Carregamento de projeto exemplo

Também é possível selecionar um projeto exemplo previamente carregado na IDE. Essa funcionalidade é útil principalmente no uso do *AspectJML* em sala de aula.

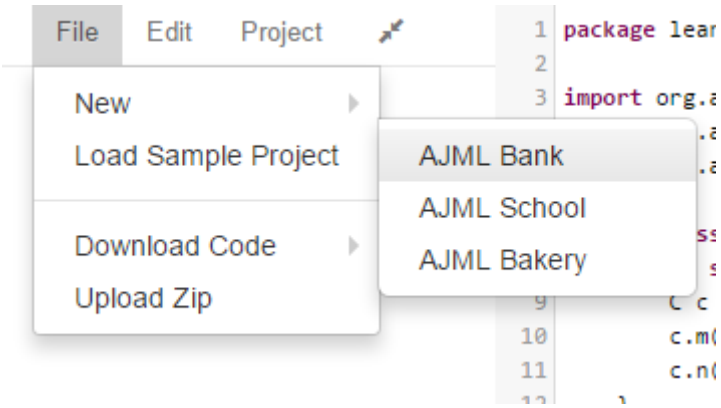


Figura 31: Carregamento de projeto exemplo

4.10. Espelho da aba atual:

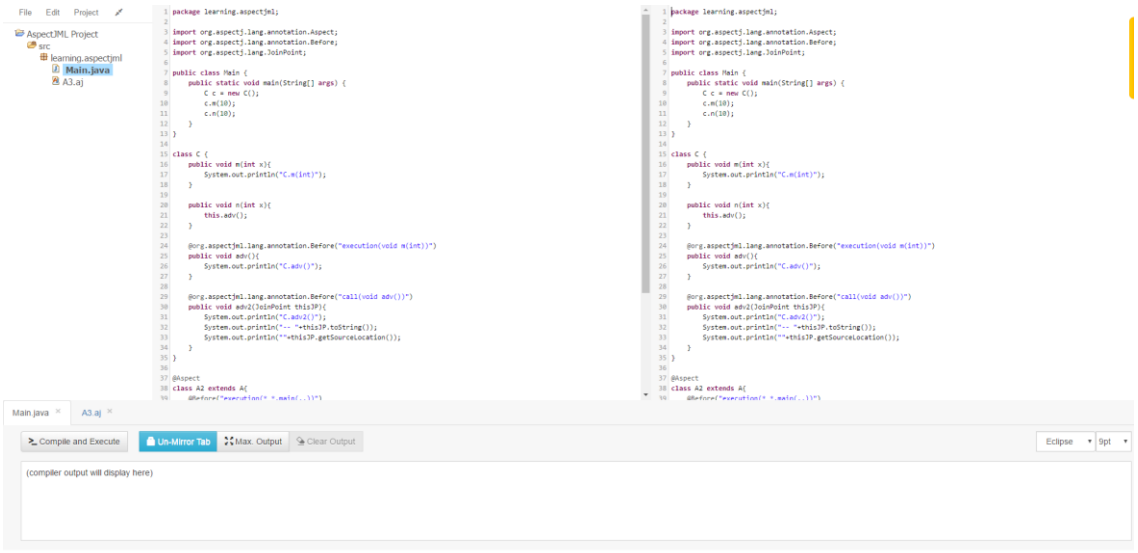


Figura 32: Espelho da aba atual

4.11. Opções de compilação:

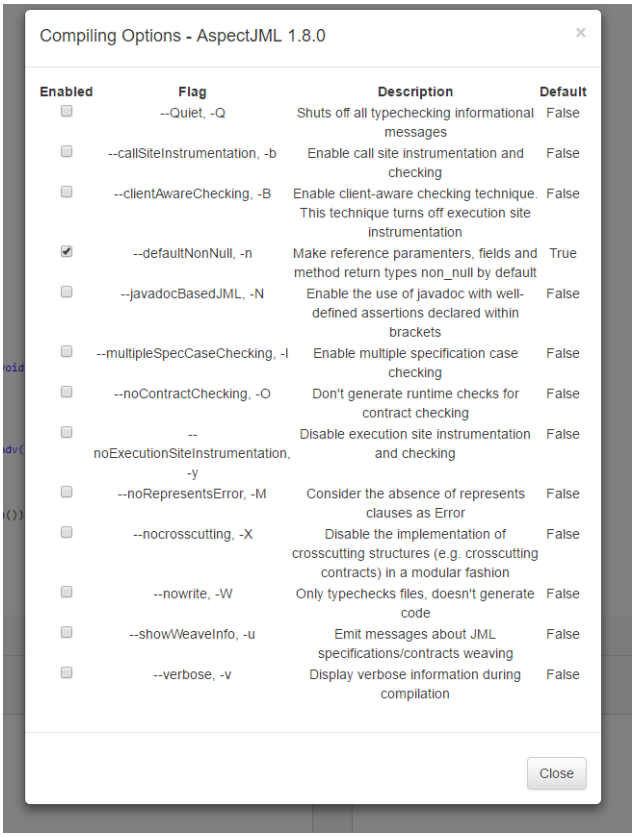


Figura 33: Opções de compilação

5. Conclusões

Neste capítulo são expostas as conclusões obtidas durante todo o desenvolvimento deste projeto. A seguir serão apresentadas as contribuições realizadas por este trabalho, as limitações da solução proposta e possíveis ações necessárias em uma abordagem futura.

5.1. Contribuições do trabalho

De maneira preliminar foi identificado a escassez de ferramentas que possibilitassem o desenvolvimento de aplicações *AspectJML* de maneira orgânica e despreocupada com a configuração do ambiente de desenvolvimento. Este trabalho fornece uma abordagem para o desenvolvimento de aplicações na linguagem. Dessa forma, ressalta-se que, através desse trabalho, tornou-se possível:

- O desenvolvimento de aplicações *AspectJML* através de uma interface *WEB.*
- A abstração do ambiente de desenvolvimento tal qual sistema operacional e versão do compilador por parte do desenvolvedor.
- Criação de uma *IDE* capaz de fornecer recursos previamente não disponíveis, como realce de sintaxe e auto completar, específicos do *AspectJML*.

5.2. Limitações

Apesar de iniciar o que seria uma *IDE online* para o desenvolvimento do *AspectJML*, o projeto desenvolvido ainda não está polido de maneira adequada. Faz-se necessário a correção de bugs e melhora de funcionalidades.

5.3. Considerações finais

Podemos concluir que este projeto atendeu seu principal objetivo. Por meio da plataforma é possível desenvolver *AspectJML* em um ambiente completamente autônomo, disponível com pouquíssima fricção para o usuário final. Isso pode ser observado pela coerência entre a proposta de desenvolvimento e o que foi entregue.

Referências

B. Meyer. Applying “design by contract”. Computer, 25(10):40–51, 1992.

H. Rebelo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. ˆ Zimmerman, M. Cornelio, and T. Th ´ um. AspectJML: Modular specifi- ¨ cation and runtime checking for crosscutting contracts. In Proceedings of the 13th International Conference on Modularity, MODULARITY ’14, pages 157–168, New York, NY, USA, 2014. ACM.

Y. A.Feldman etal. Jose:Aspects forDesignbyContract80-89. IEEE SEFM, 0:80–89, 2006.

G.Kiczales, E.Hilsdale, J.Hugunin, M.Kersten,J.Palm,andW.Griswold. Getting Started with AspectJ. Commun. ACM, 44:59–65, October 2001

Webb, Jason. "A declarative data contract container type for Python." pycontract 0.1.4 (2010). Python Package Index. Web. 5 Dec 2012.

G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: Abehavioral interface specification language forJava. ACMSIGSOFT Software Engineering Notes, 2006.

B. Meyer. Eiffel: The Language. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

Hoare, C.A.R. An Axiomatic Basis for Computer Programming, Communications of the ACM 12, 10, 576-583, October 1969.

E. W. Dijkstra. A Discipline of Programming, Prentice-Hall Inc, Englewood Cliffs, NJ, 1976.

Jones, C. B. The Early Search for Tractable Ways of Reasoning about Programs. IEEE, Annals of the History of Computing 25(2), 26-49 (2003).

Royce, W. Winston: Managing the development of large software systems; in: Proceedings of IEEE Wescon (1970), pp. 382-338

B. Boehm, “Software Engineering”, IEEE Transactions on Computers, vol. C-25, no. 12, Dec. 1976