Reestruturação e aperfeiçoamento de uma ferramenta para detecção de conflitos semânticos de código

João Pedro Henrique Santos Duarte
Orientador: Paulo Henrique Monteiro Borba

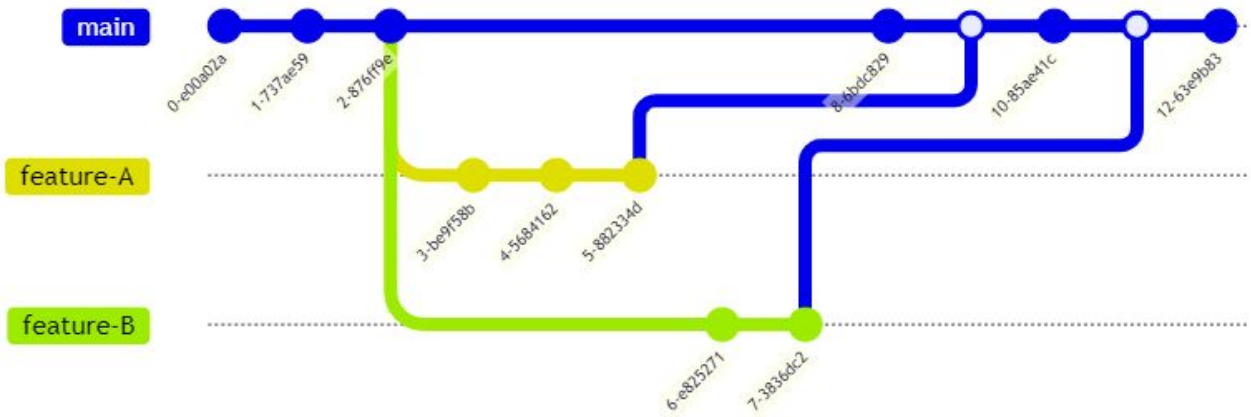
Centro de Informática UFPE
Universidade Federal de Pernambuco

O desenvolvimento moderno de software é um processo **colaborativo**.

Funcionalidades são divididas em várias tarefas

TODO	*DOING*	*DONE*
Importar listagem de alunos de Excel	Formulário de auto-avaliação.	Cadastro manual de alunos.
Refatorar código da visualização de metas		

Como permitir que o desenvolvimento de *software ocorra* de forma independente dentro de um ambiente colaborativo?



Branches e *Merges*

Branches permitem que diferentes desenvolvedores separem-se do tronco principal do projeto e realizem suas tarefas de forma independente, realizando a posterior integração através de *merges*.

```
react-app-demo  my-branch  git merge his-branch
Auto-merging src/colors.txt
CONFLICT (content): Merge conflict in src/colors.txt
Automatic merge failed; fix conflicts and then commit the result.
```

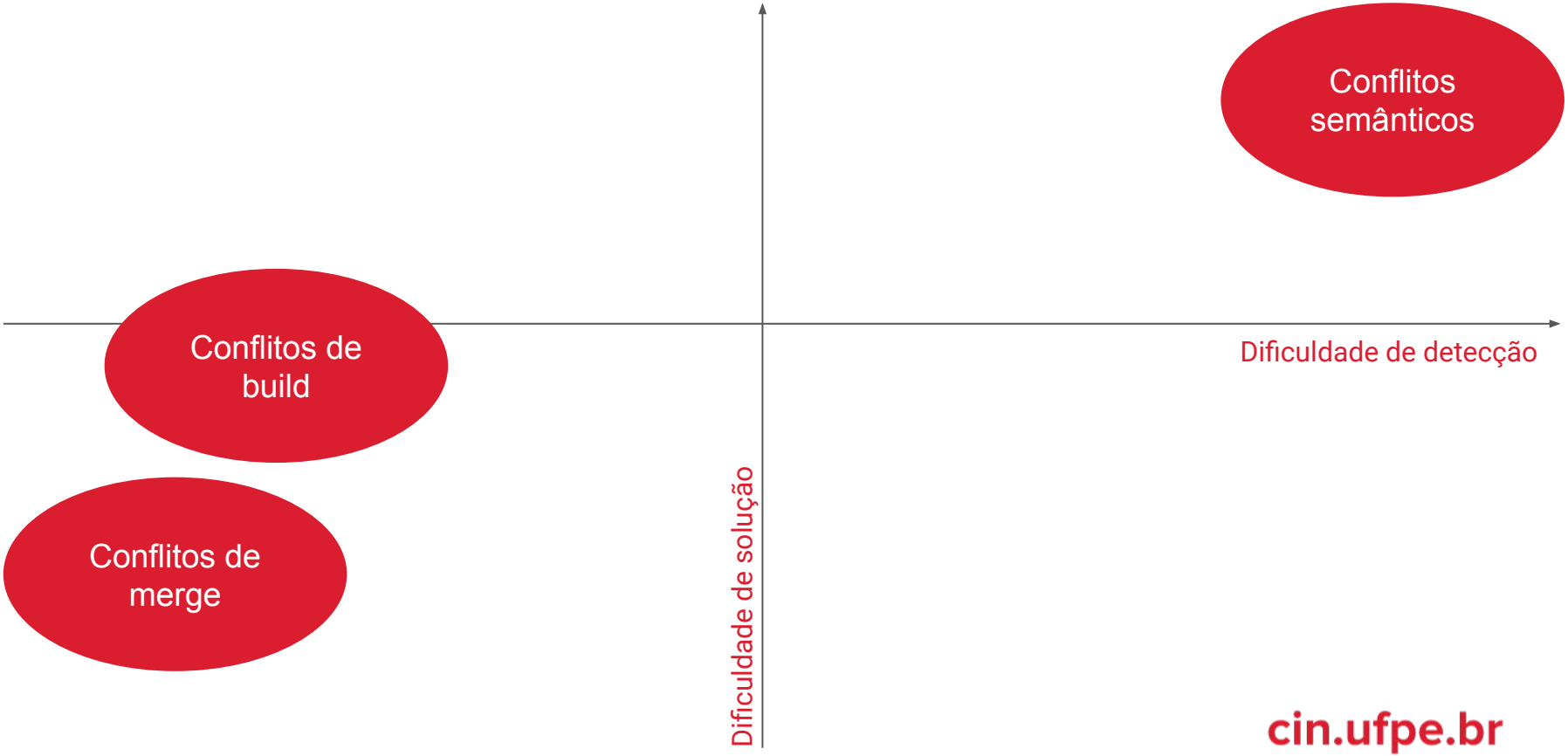
There's always a catch!

Durante um *merge*, podem ocorrer **conflitos de integração causados por diferentes versões do código**. Custam tempo e podem interferir na qualidade do software se não forem detectados antes de atingir o usuário final.

There's always a catch

- Durante um *merge*, podem ocorrer **conflitos de integração** causados por diferentes versões do código.
- Custam tempo e podem interferir na qualidade do *software* se não forem detectados antes de atingir o usuário final.

Conflitos podem aparecer de diversas formas



Conflitos de *merge*

Base

```
class Aluno {
  private UUID codigo;
  private String cpf;
  private String email;

  Aluno(String cpf, String email)
{
    this.codigo = gerarCodigo();
  }
}
```

Left

```
class Aluno {
  private UUID codigo;
  private String cpf;
  private String email;

  Aluno(String cpf, String email)
{
    this.codigo = gerarCodigo();
    validarEmail(email);
  }
}
```

Right

```
class Aluno {
  private UUID codigo;
  private String cpf;
  private String email;

  Aluno(String cpf, String email)
{
    this.codigo = gerarCodigo();
    validarCpf(cpf);
  }
}
```

Merge

```
class Aluno {
  private UUID codigo;
  private String cpf;
  private String email;

  Aluno(String cpf, String email)
{
    this.codigo = gerarCodigo();
    <<<<<<<<<<<<<<
    validarEmail(email);
    ==============
    validarCpf(cpf);
    >>>>>>>>>>>>>>
  }
}
```

Conflitos semânticos em tempo de compilação

- Ocorrem quando a integração das modificações introduzidas pelos desenvolvedores resulta em uma versão **não compilável** do *software*.

```
class GovApiService {
  Aluno fetchAluno (Cpf cpf) {
    (...)
  }
}
```

Left renomeia o método

```
class AlunoController {
  Aluno criarAluno (Cpf cpf) {
    Aluno umAluno = govService.getAluno(cpf)
  }
}
```

Right referencia o nome antigo

- Ao compilarmos: ***CannotFindSymbolError***.
- Detectados e resolvidos com facilidade (mas custam tempo).

Conflitos semânticos em tempo de execução

Ocorrem quando as modificações introduzidas interferem entre si em **tempo de execução**.

Left

```
class Aluno {
  void validarAluno(String email) {
    StringUtils.assertMinLength(email, 15);
    (...)
  }
}
```

Right

```
class Aluno {
  void validarAluno(String email) {
    (...)
    StringUtils.assertMaxLength(email, 10);
  }
}
```

Merge

```
class Aluno {
  void validarAluno(String email) {
    StringUtils.assertMinLength(email, 15);
    (...)
    StringUtils.assertMaxLength(email, 10);
  }
}
```

Entrando na cabeça dos desenvolvedores

- Exige compreender as intenções do desenvolvedor com as modificações introduzidas e verificar como elas interferem entre si;
- Como construir especificações que formalizam as intenções do desenvolvedor ao introduzir suas modificações?

Testes unitários como especificações parciais

Base — **FAIL**

```
class Aluno {
  void validarAluno(String email) {
    (...)
  }
}
```

Left — **FAIL**

```
class Aluno {
  void validarAluno(String email) {
    StringUtils.assertMinLength(email, 15);
    (...)
  }
}
```

Right — **PASS**

```
class Aluno {
  void validarAluno(String email) {
    StringUtils.assertMaxLength(email, 10);
    (...)
  }
}
```

Merge — **FAIL**

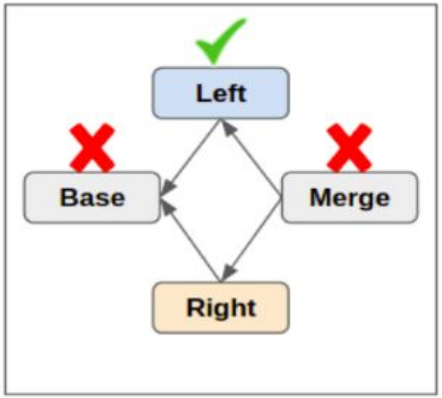
```
class Aluno {
  void validarAluno(String email) {
    StringUtils.assertMinLength(email, 15);
    (...)
    StringUtils.assertMaxLength(email, 10);
  }
}
```

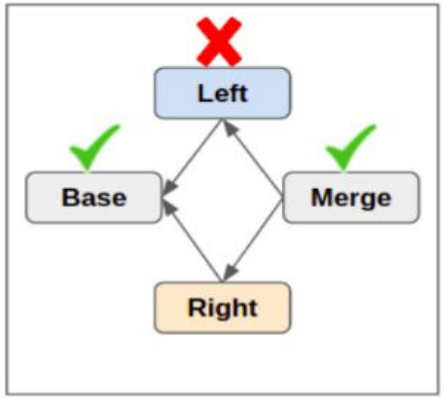
```
Exception e = assertThrows(StringException.class, () -> {
 String email = "12caracteres"; // string com tamanho 12
 Aluno.validarAluno(email);
});
assertTrue(e.getMessage().equals("String não pode ter mais que 10 caracteres");
```

Como detectar interferências nestas especificações?

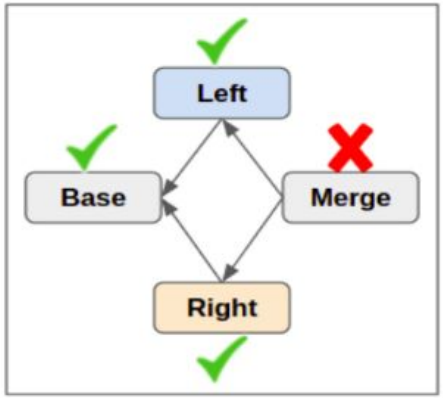
- Analisando as execuções dos casos de testes utilizando critérios heurísticos

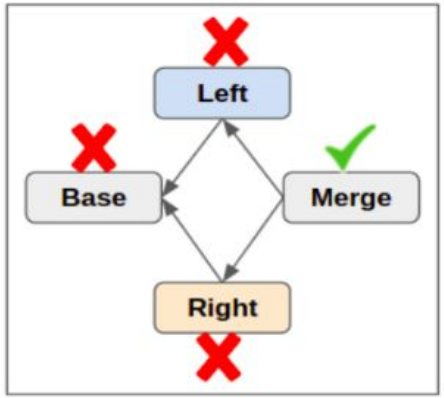
Primeiro critério





Segundo critério





SMAT

Identifying Semantic Merge Conflicts Via Automated Behavior Change Detection

Centro de Informática UFPE

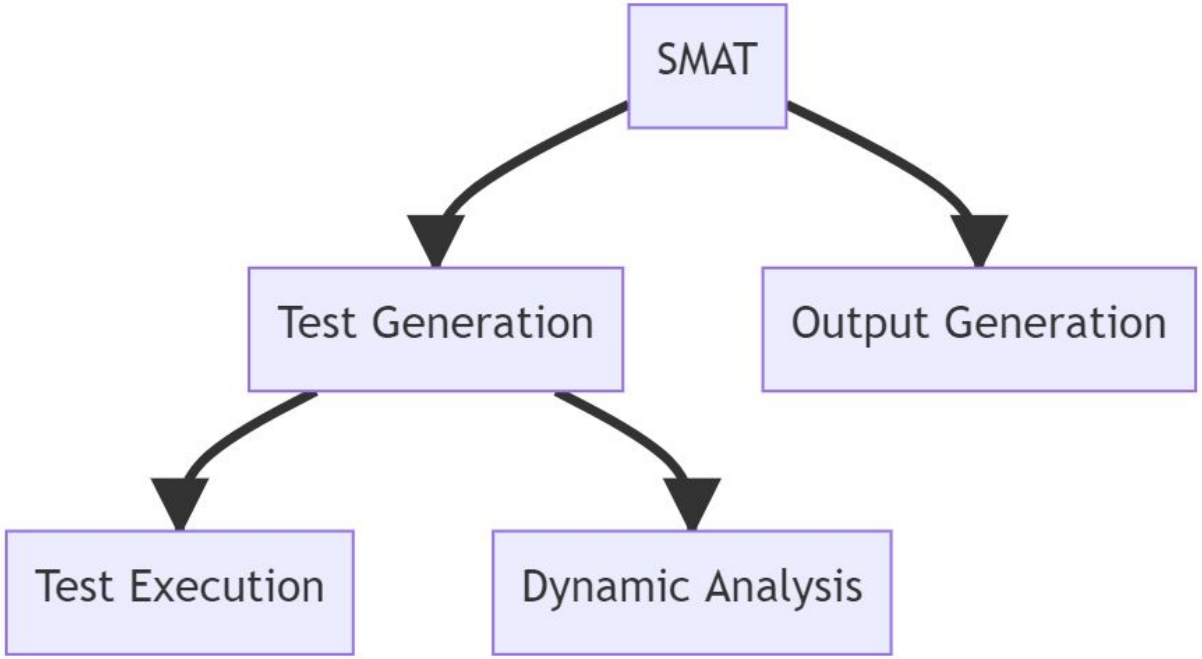
Universidade Federal de Pernambuco

Silva, L. D., Borba, P., Mahmood, W., Berger, T., and Moisakis, J. (2020). **Detecting semantic conflicts via automated behavior change detection**. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 174–184.

Em alto nível

- SMAT recebe:
  - Um cenário de merge
  - Um método alvo onde se quer verificar a existência de conflitos semânticos
- Realiza a execução sequencial de 4 etapas:
  - Geração de suítes de testes para *left* e *right* utilizando ferramentas automáticas.
  - Execução das suítes em cada uma das versões do cenário analisado.
  - Análise heurística do resultado das execuções dos testes.
  - Geração de relatórios a respeito dos cenários analisados.

Diagrama macro-arquitetural de SMAT



Interface pública de SMAT - (Entrada)

- Recebe como entrada um arquivo CSV com diversos cenários de *merge* a serem analisados.
- Campos do CSV
  - Nome do projeto a ser testado
  - Um *booleano* indicando se o cenário deve ser avaliado ou não
  - Seguem 4 colunas com os SHAs a serem analisados (base, left, right e merge)
  - FQCN da classe target
    - podem ser informadas mais de uma classe usando | como separador
  - Assinatura do método a ser verificado
    - métodos que possuem mais de um argumento utilizam | como separador
  - Seguem 4 colunas com os JARs de cada SHA (base, left, right e merge)

As coisas sempre podem melhorar

Identificando pontos de melhoria em SMAT

Centro de Informática UFPE
Universidade Federal de Pernambuco

Tanto para o usuário final

Detecção de falsos positivos

- Um conflito pode ser reportado mesmo que o método alvo não tenha sido executado.

Difícil reprodutibilidade

- Execuções não-determinísticas, dificultando a reprodução de experimentos.

Baixa configurabilidade

- Alterar aspectos simples da aplicação só é possível modificando o código fonte.

Performance em cenários reais

- Apenas um único alvo pode ser analisado por execução.

Quanto para os desenvolvedores

Arquitetural

- Modelo anêmico dificultando a compreensão das entidades e suas relações.
- Acoplamento desnecessário entre diferentes conceitos da aplicação.

Gerência de projeto

- Ausência de informações de tipo.
- Baixa documentação a respeito das decisões tomadas no projeto.

Atacando os pontos de melhoria

Nova interface para SMAT

- Utilizar arquivos JSON para entrada/saída.
  - Permitem representar estruturas complexas mais facilmente do que utilizando CSVs;
- Introduzir a ideia de múltiplos *targets*, as unidades mínimas a serem analisadas: métodos ou propriedades de uma ou mais classes.

```
class Aluno {
  void validarCpf(String cpf) {
    (...)
  }

  void validarEmail(String email) {
    (...)
  }
}
```



```
"targets": {
  "br.ufpe.cin.entidades.Aluno": [
    "validarCpf(String cpf)",
    "validarEmail(String email)"
  ]
}
```

Detectando os alvos envolvidos

- Como verificar quais alvos participam de um conflito?

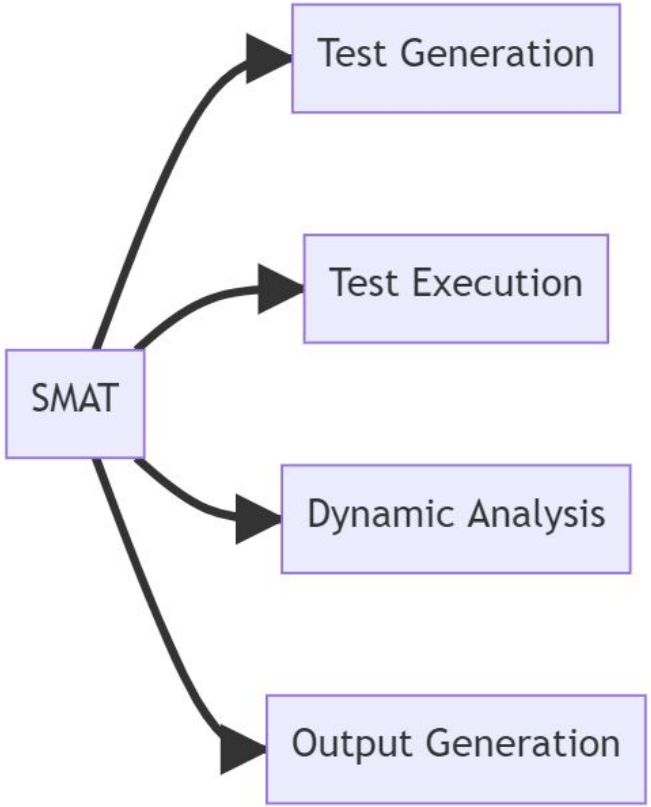
```
class Aluno {
  void validarCpf(String cpf) {
    (...)
  }

  void validarEmail(String email) {
    (...)
  }
}
```

- Estratégias disponíveis:
  - Análise estática do código fonte do teste.
  - Análise dinâmica, utilizando coleta de cobertura de testes.

- Resolve: detecção dos falsos positivos apresentados anteriormente.

Alterações na macro-arquitetura

- Uma classe de controle que realiza a chamada a **módulos de execução**.
- Cada módulo de execução expõe uma interface para seus clientes.
- Módulos podem depender de abstrações das etapas anteriores, mas jamais de etapas posteriores.

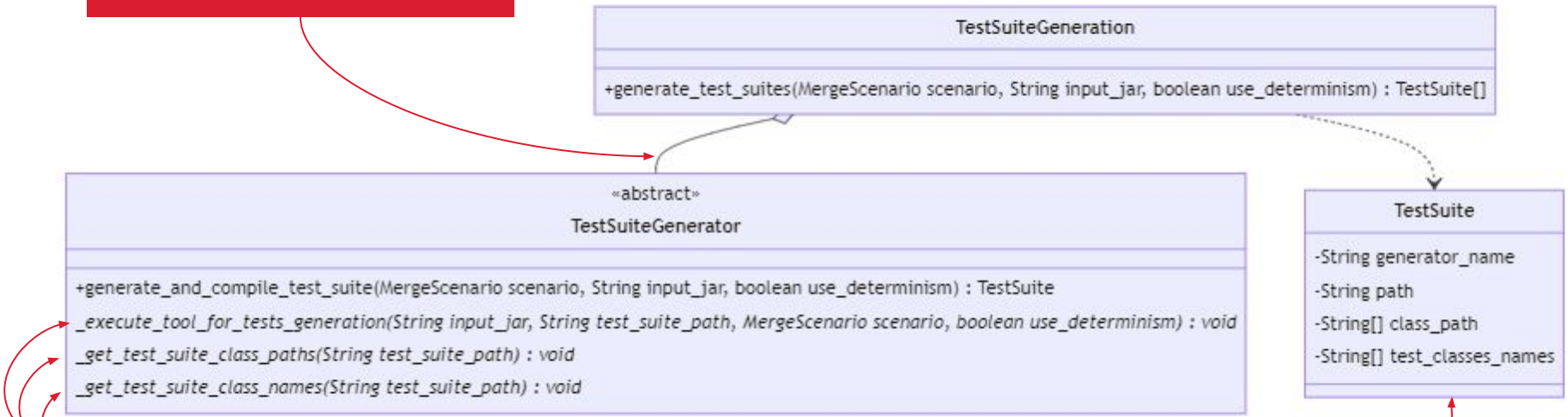


Test Suites Generation

- Geração das suítes de testes para *left* e *right* utilizando Randoop e Evosuite;
- Não estava isolado na arquitetura anterior.
  - Modificar um gerador envolvia conhecer outros aspectos da execução
- Geração não determinística;
  - Execuções sucessivas podem produzir resultados diferentes.
- Não permite determinar em tempo de execução quais geradores serão utilizados nem quanto tempo será gasto durante a etapa de busca.
  - Estas modificações devem ser feitas diretamente no código fonte da aplicação.

Nova arquitetura de *Test Suites Generation*

Geradores utilizados podem ser escolhidos na instanciação



Especificidades das ferramentas abstraídas em *template methods*

Introdução de uma estrutura para os dados de uma suíte

Adicionando determinismo na geração

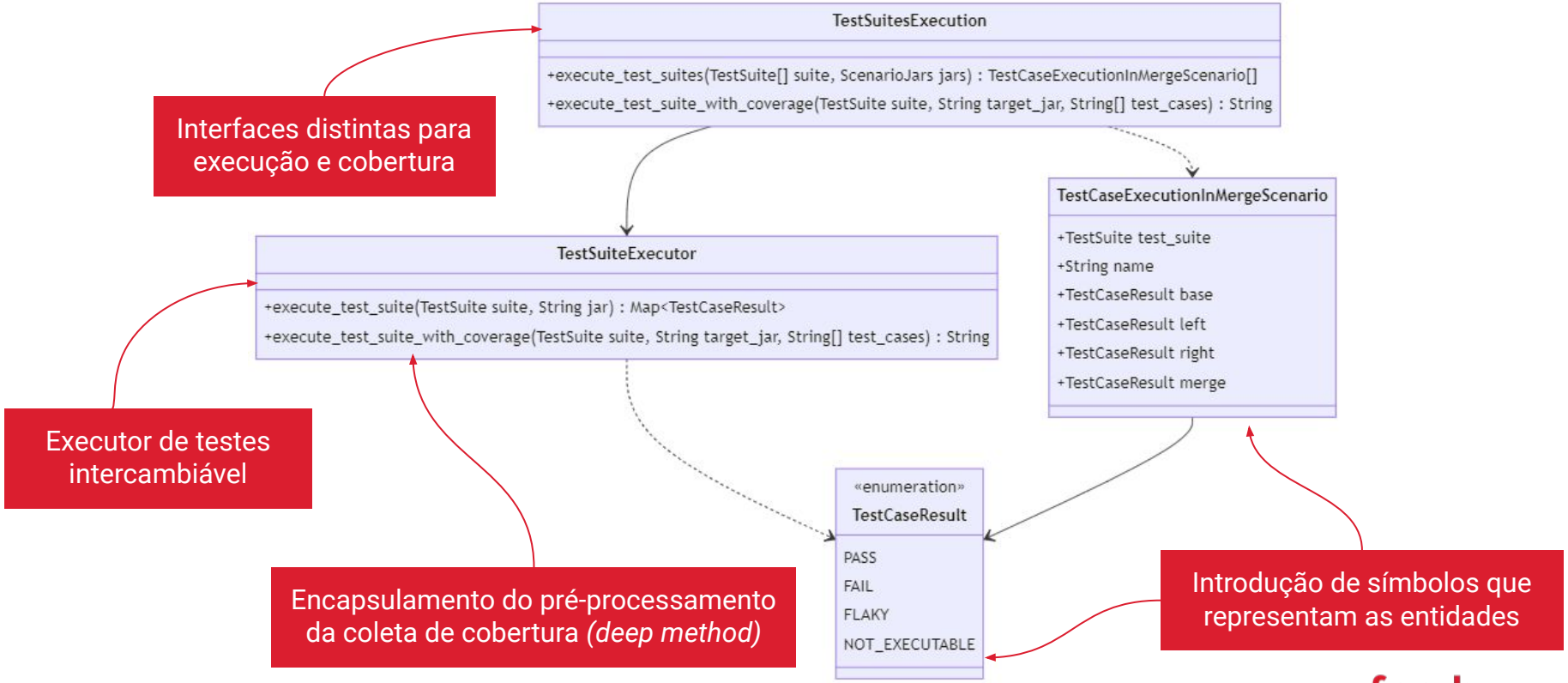
- Tanto Evosuite quanto Randoop podem ter execução determinística.
  - Mesmo que ambos tenham certo grau de aleatoriedade, é possível tornar a execução determinística fornecendo uma *seed*.
- Nova interface de *Test Suites Generation* permite a geração determinística.
  - O usuário pode ativar o determinismo via arquivo de configuração.
  - Uma *seed* pode ser informada via arquivo de configuração.
    - Se nenhuma *seed* for informada, um valor pré-definido será utilizado.

Test Suites Execution

- Execução das suítes geradas em cada uma das versões do cenário;
  - Permite também realizar a coleta de cobertura de código
- Acoplamento com o executor utilizado *(JUnit 4)*
- *Shallow module (Ousterhout, 2018)*
  - Pelo módulo expor um único método, a coleta de cobertura ocorre em todas as execuções, mesmo quando não desejado pelo usuário, resultando em perda de performance;
  - Para realizar a coleta de cobertura de código, o usuário deve, obrigatoriamente, realizar o pré-processamento da versão antes de invocar a execução dos testes *(leaking internals)*.

Ousterhout, J. K. (2018). **A philosophy of software design**, volume 98. Yaknyam Press Palo Alto.

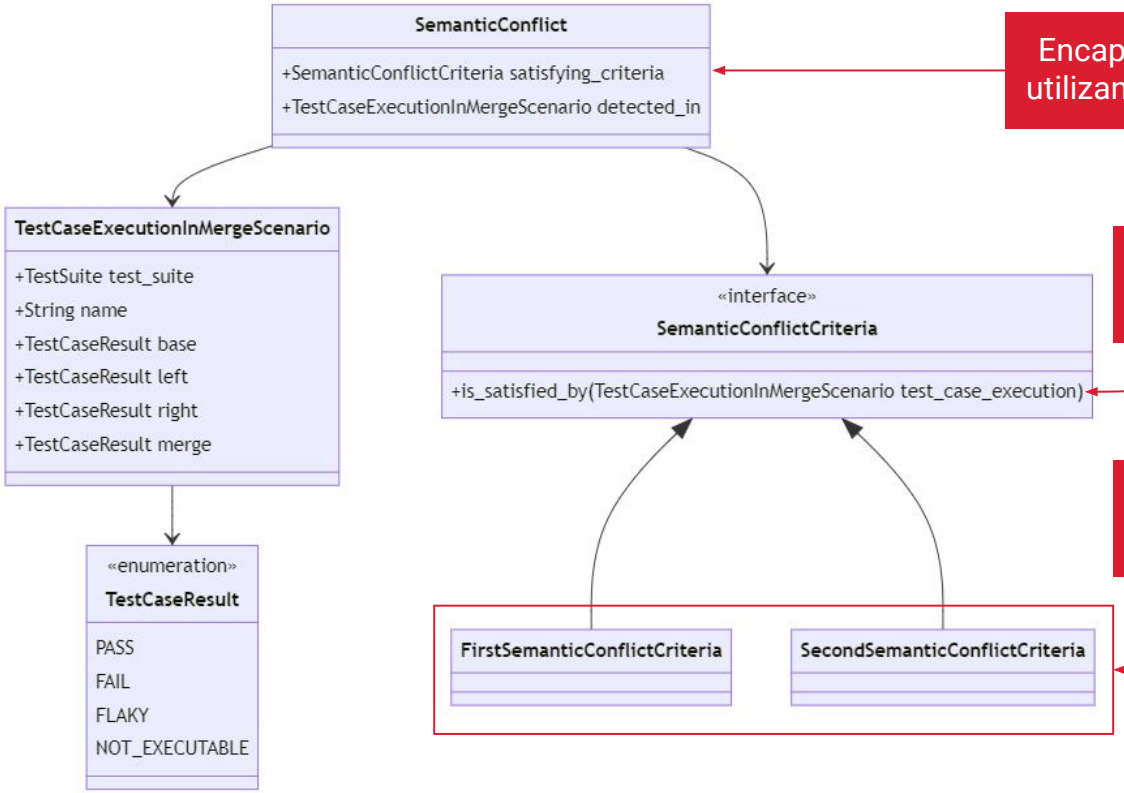
Nova arquitetura de *Test Suites Execution*



Dynamic Analysis

- Avaliação dos resultados de cada teste executado para detectar conflitos semânticos ou meras mudanças de comportamento.
- Conflitos e mudanças de comportamento em um mesmo lugar (God Class)
  - Realizados utilizando operações não-triviais entre conjuntos de testes.
- Métodos com elevado número de parâmetros e abuso de primitivos
- E se quiséssemos modelar a relação entre um caso de teste e um conflito semântico? E a de um caso de teste e uma mudança de comportamento?

Conflitos Semânticos e Casos de Teste



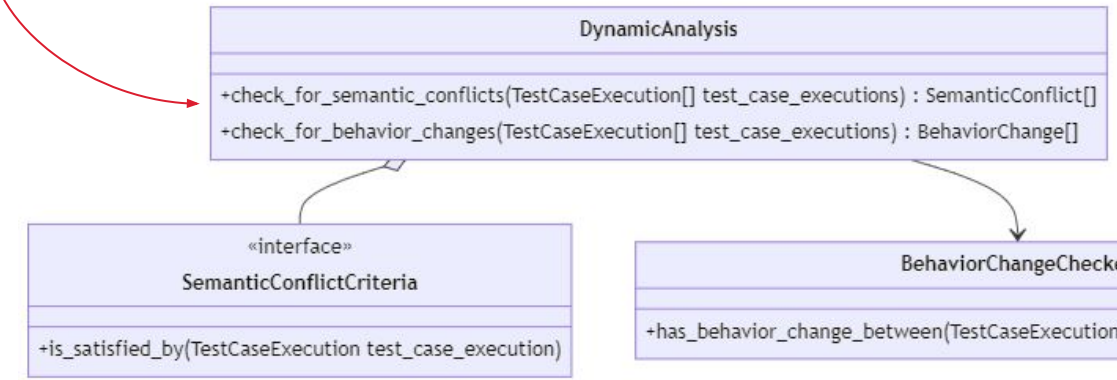
Encapsula um conflito semântico utilizando a linguagem do domínio.

Na análise, queremos saber se algum critério é satisfeito pelo caso de teste

Para adicionar novos critérios, basta implementar uma nova classe.

Nova Arquitetura de *Dynamic Analysis*

Fachada para simplificar o consumo pelo cliente



Modelagem similar à utilizada na detecção de conflitos semânticos.

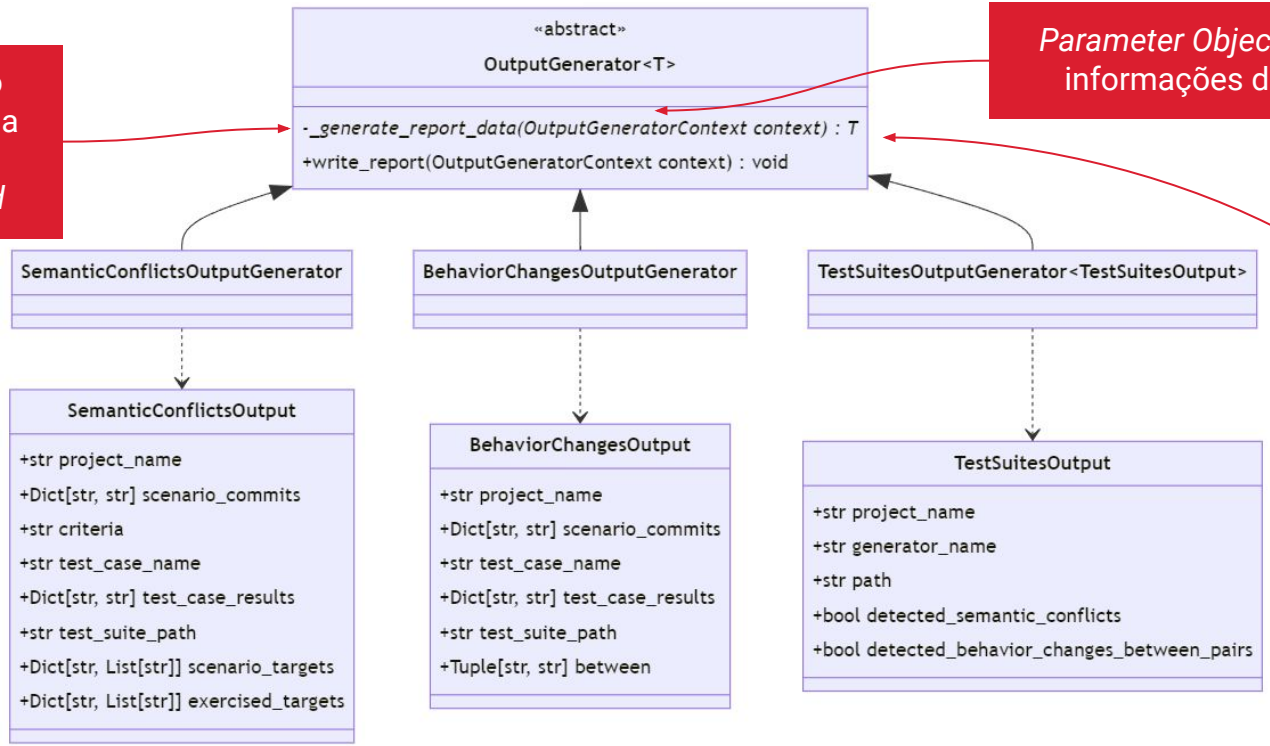
Output Generation

- Construir relatórios contendo os resultados da análise;
- Permitir que o usuário escolha quais relatórios serão gerados;
- Queremos que novos geradores possam ser adicionados com facilidade;
- Um relatório pode precisar acessar informações de diferentes etapas.
    - Como fazer isso mantendo uma interface comum para todos os geradores?

Nova arquitetura de *Output Generation*

Comportamento específico de cada relatório em um *template method*

Parameter Object encapsula as informações de cada etapa



Novo relatório, nova subclasse

Melhorias na gerência do projeto

- Fortalecer o uso de análise estática no projeto.
  - Anotações de tipo adicionadas nas classes modificadas
    - Verificação da validade dos tipos utilizando *mypy*.
    - Adicionar verificação na esteira de integração contínua do projeto.
- Documentação
  - Adicionado um esboço de *Architecture Decision Record (ADR)* apresentando e discutindo em alto nível as decisões arquiteturais de cada módulo do projeto.
  - Melhoria na documentação a respeito da execução e customização da ferramenta.

Discussão

Aperfeiçoamos SMAT para o usuário final

- Capacidade de detectar um subconjunto de falsos positivos;
- Execuções determinísticas;
- Maior configurabilidade;
- Avaliação de mais de um alvo em um mesmo cenário;
- Maior documentação.

E reestruturamos a fim de reduzir o débito técnico

- Refatoração do modelo da aplicação;
- Reorganização dos módulos da aplicação;
- Introdução de uma documentação arquitetural;
- Introdução de verificações estáticas.

Trabalhos Futuros

- Permitir que SMAT seja utilizado com outras linguagens de programação;
  - Depende da evolução de ferramentas geradoras de testes para outras linguagens.
- Atualizar os clientes para que sejam compatíveis com a nova interface;
  - A interface antiga ainda é aceita, mas foi marcada como depreciada.
- Incrementar a cobertura de testes;
  - Introduzir *smoke tests* beneficiando-se das execuções determinísticas da aplicação.
- Avaliar pontos de melhoria na performance da aplicação:
  - Estudar como paralelismo/concorrência pode ser utilizado dentro da ferramenta;

Agradecimentos

Perguntas?