Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Anexo C - Ferramentas para criar testes de mutação

1.1 Introdução

Os teste de mutação serve precisamente para avaliar a qualidade dos testes unitários.

Ferramenta	Seleção	Inserção	Deteção	Relatório
Jester	*Naive*	*Naive*	*Naive*	*AS*
Jumble	Convenção	*NDClassloader*	*EE*	*T*
PIT	Cobertura	*Instrument*	*EE*	*AS*
Java	Manual	*Naive*	*Naive*	*AS*
javaLanche	Cobertura	*Schemata*	*EE*	*P*

Tabela 1.1: Comparação das ferramentas para testes de mutação.

1.2 Análise das ferramentas existentes

Os requisitos para as ferramentas de testes de mutação são os seguintes:

- a ferramenta deve ser capaz de analisar o código coberto por um conjunto de testes unitários e realizar mutações apenas nesta parte do código
- a ferramenta deve suportar a versão 1.6 ou 1.7 do *Java*
- a ferramenta deve suportar *JUnit*
- a ferramenta deve suportar o *Maven* ou *Ant*

Na realidade, existem poucas ferramentas para a realização dos testes de mutação para o código *Java*, e durante a pesquisa foram encontradas as seguintes:

- *Jester* [1]
- *Jumble* [2]
- *PIT* [3]
- *Java* [4]
- *javaLanche* [5]

Nas tabelas 1.1 e 1.2 é apresentada a comparação das ferramentas encontradas.

1.2.0.1 Seleção

Existem muitas maneiras para selecionar testes unitários a serem executados. Tendo por base a tabela 1.1 serão analisadas as vantagens e desvantagens de cada método.

Manual A seleção dos testes unitários é feita manualmente. Vantagens:

- Não há

Ferramenta	*Java 1.6*	*Java 1.7*	*JUnit*	*Ant*	*Maven*
Jester	SIM	NÃO	SIM	NÃO	NÃO
Jumble	SIM	SIM	SIM	SIM	NÃO
PIT	SIM	SIM	SIM	SIM	SIM
Java	SIM	SIM	SIM	NÃO	NÃO
javaLanche	SIM	SIM	SIM	SIM	NÃO

Tabela 1.2: Sistemas suportadas.

Desvantagens:

- Exige trabalho manual
- Apenas pode ser utilizado para determinar a cobertura de classes individuais

Naive A seleção dos testes é automática, mas que exige previa marccação manual. Normalmente todo o conjunto, ou uma grande parte do conjunto dos testes, é executado para cada mutante. Vantagens:

- Não há

Desvantagens:

- É muito lento excepto para os projetos pequenos

Convenção A seleção dos testes é automática e os testes são selecionado de acordo com a sua nomenclatura (baseada no nome do método). Vantagens:

- Mais rápido que a abordagem *Naive*.

Desvantagens:

- Diminui a performance total uma vez que existem vários sítios do código (que sofreram mutação) que *a priori* não são cobertos por testes unitários.

Cobertura A seleção dos testes é automática e os testes são selecionados por cobertura da linha, bloco ou condição. Apenas os testes que exercitem a linha, bloco ou condição que contém o mutante serão executados. Vantagens:

- Apenas os testes que poderiam detetar o mutante serão executados.
- Não executa os testes para os mutantes aplicados ao código que não possui cobertura.
- Fornece uma imagem da cobertura de todo o conjunto dos testes.

Desvantagens:

- Algum *overhead* é necessário para medir a cobertura se um projeto for complexo e tiver poucos testes unitários.

1.2.0.2 Inserção

Naive Mutantes são gerados e são guardados nos ficheiros com as classes que por sua ver são escritos em disco. Nova *JVM* é lançada para cada mutante. Vantagens:

- É um caso de confiança porque trabalha com JVM.
- Os mutantes são activos durante a construção dos estados estáticos.

Desvantagens:

- Lento - para cada mutante é necessário iniciar um nova *JVM*.

Non delegating class loader Mutantes estão em memória e são inseridos em JVM por criação de novo *classloader*. Vantagens:

- Mais rápido que Naive.
- Os mutantes são activos durante a construção dos estados estáticos.

Desvantagens:

- Exige grandes quantidade de espaço *permgen*.

Mutant schmeta É gerada a classe que contém todos os mutantes. Cada mutante é ativado perante a programação prévia. Vantagens:

- Mais rápido que Naive.
- É um caso de confiança porque trabalha com JVM.

Desvantagens:

- Mutantes não são activos durante a construção dos estados estáticos.

Instrumentation api Os mutantes estão em memória e são inseridos no *JVM* usando um *instrumentation api*. Vantagens:

- Mais rápido que Non delegating class loader.

Desvantagens:

- Requer *JVM* com o suporte para *instrumentation api*
- Mutantes não são activos durante a construção dos estados estáticos

1.2.0.3 Deteção

Naive Todos os testes são selecionados para serem executados. Vantagens:

- Não há

Desvantagens:

- Processo demorado

Naive As classes de testes são divididos em casos de testes e são executados até que um destes detete um mutante. Vantagens:

- Potencialmente mais rápido que outras abordagens.

Desvantagens:

- Devisão dos testes de classes pode causar problemas com algumas extenções do *JUint*

1.2.0.4 Relatório com os resultados

Naive Os relatórios das ferramentas podem ser classificados de seguinte maneira:

- T - um relatório simples que contém a descrição das mutações e seus resultados

- P - tem a estrutura exatamente igual a T mas com uma formatação visual, normalmente *html*

- AS - código fonte original ou mutado é anotado com os resultados

1.2.0.5 Conclusão

De todas as ferramentas analisadas, só três poderiam ser utilizadas no projeto da FeedZai por cumprirem os requisitos mínimos, Jumble, *javaLanche* e *PIT*. Destas três só uma possui *plugin* para o *Maven* atualmente utilizado na compilação do produto. Outras duas ferramentas também podem ser integradas no produto, uma vez que o *Maven* permite a execução das tarefas *Ant*.

Na base desta análise e discução com a direção da FeedZai será escolhida a ferramenta de criação dos testes de mutação mais apropriada.

1.3 Bibliografia

1. Jester. http://jester.sourceforge.net/. Acesso em Junho 18, 2013.

2. Jumble. http://jumble.sourceforge.net/. Acesso em Junho 18, 2013.

3. PIT. http://pitest.org/. Acesso em Junho 18, 2013.

4. Java. http://cs.gmu.edu/ offutt/mujava/. Acesso em Junho 18, 2013.

5. javaLanche. http://www.st.cs.uni-saarland.de/mutation/. Acesso em Junho 18, 2013.