Análise de mutantes
Testes de mutação

Criado: ago/2011

Últ. alteração: mar/2013

Tópicos

- Principais técnicas de testes baseados em falhas
- Testes de mutação
- Análise de mutantes vivos
- Escore de mutação
- Operadores de mutação
- Exemplo de ferramenta

Testes baseados em falhas

- Modelo de base:
  - enganos típicos cometidos durante o processo de desenvolvimento
- Exemplos de técnicas
  - Estáticas
    - Semeadura de erros (*Error Seeding*) (Budd, 1981)
    - Análise de Mutantes (Teste de Unidade) (DeMillo et al., 1978)
    - Mutação de Interface (Teste de Integração) (Delamaro et al., 2001)
    - Mutação de modelos de estado (Testes baseados em modelos) (Fabbri et al. 1994)
  - Dinâmicas
    - Injeção de falhas (Arlat et al., 1989)

Hipótese do Programador Competente

- Um dos princípios básicos da técnica: um programador competente escreve programas corretos ou próximo do correto.
  - Assumindo a validade desta hipótese: *bugs* são introduzidos no programa por meio de **pequenos desvios sintáticos** que fazem com que a execução do produto leve a um comportamento incorreto.
  - Portanto, cada mutante tem uma única diferença sintática em relação ao programa original *P*.
  - O testador deve construir um conjunto de teste que mostre que tais modificações criaram produtos que não são corretos (Agrawal et al., 1989).

Efeito de acoplamento

- Segundo princípio da técnica (DeMillo et al., 1978):
  - Falhas *(bugs)* complexas são uma composição de *bugs* simples.
  - Conjuntos de teste que revelam falhas simples são também capazes de revelar falhas complexas (Budd, 1980).

Passos (1)

- Dados um produto que se deseja testar *P* e um conjunto de teste *T*. Os passos de aplicação do Teste de Mutação são:

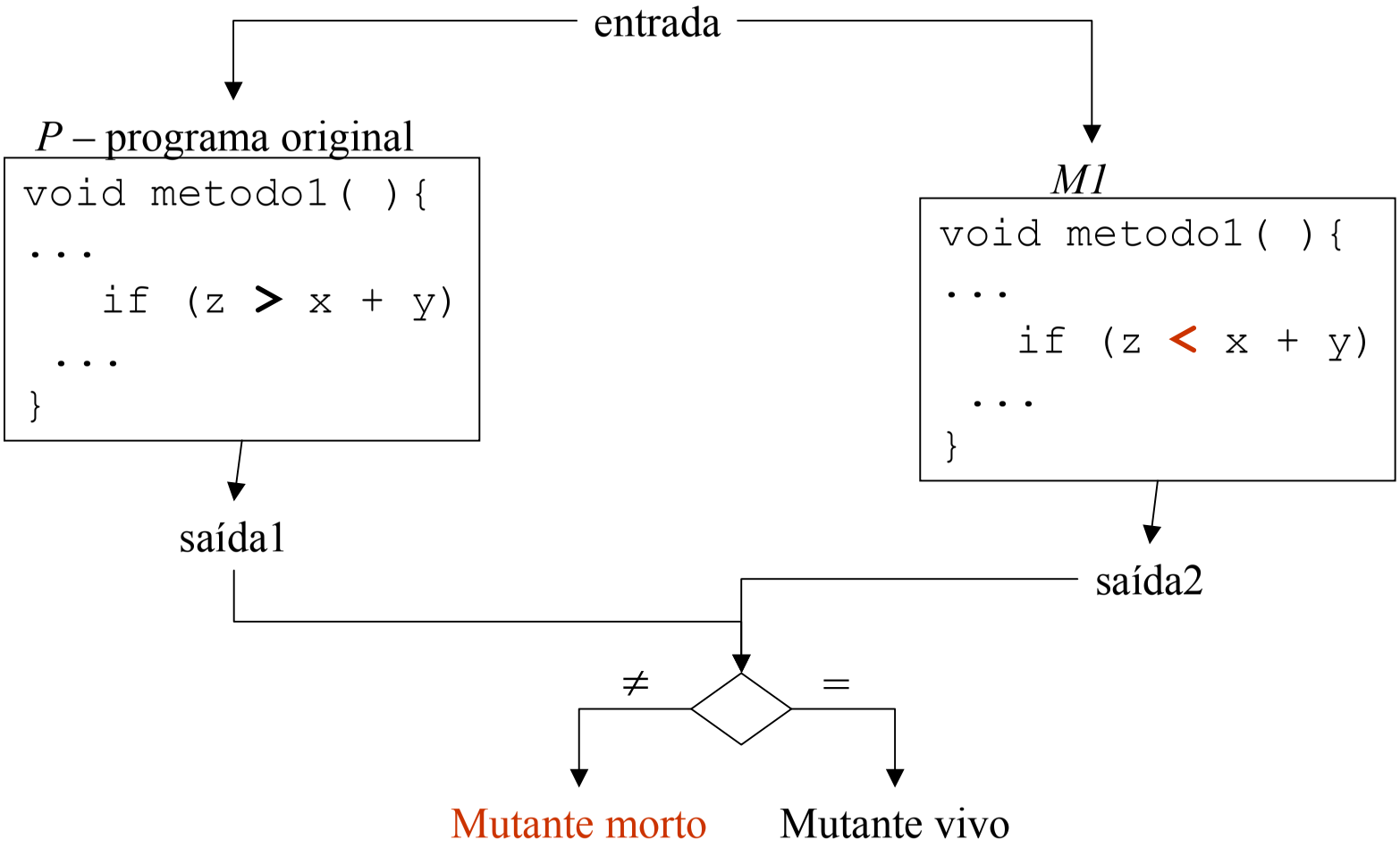
1. Execução do produto original
   - *P* é executado com *T*
   - Se ocorrer um defeito (*failure*), o teste termina.
   - Se nenhum defeito for identificado, *P* ainda pode conter *bugs* que *T* não foi capaz de revelar.
2. Geração dos mutantes
   - P é submetido a um conjunto operadores de mutação que transformam *P* em $M_1, M_2, \ldots, M_n$, denominados **mutantes** de *P*.

Operadores de mutação são regras que representam enganos freqüentes ou desvios sintáticos relacionados com uma determinada linguagem de programação.

Passos (2)

3. Execução dos mutantes
   - Os mutantes são executados com o mesmo conjunto de teste *T*
   - **Mutantes mortos**
     - resultados diferentes de *P*
   - **Mutantes vivos**
     - resultados idênticos ao de *P*
   - ☞ O ideal é ter apenas mutantes mortos!
4. Análise dos mutantes vivos
   - Identificar possível equivalência em relação a P
   - Expor uma fraqueza do conjunto de teste *T*

Esquema geral da técnica



Escore de Mutação

- Medida objetiva para avaliar a adequação de *T* em relação ao Teste de Mutação

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

- Em que:
  - *DM*(*P*,*T*): número de mutantes mortos por *T*;
  - *M*(*P*): total de mutantes gerados;
  - *EM*(*P*): número de mutantes equivalentes a *P*.

Criação de mutantes

- Uso de ***operadores*** de mutação para aplicar as pequenas modificações sintáticas aos programas

- A cada aplicação de um operador de mutação, um novo mutante *M* de *P* é criado

- Exemplo:
  - ROR (*Relational Operator Replacement*): substitui um operador relacional por outro

P – programa original

```
void metodo1( ){
...
   if (z > x + y)
 ...
}
```

ROR

M1

```
void metodo1( ){
...
   if (z < x + y)
 ...
}
```

ROR

M2

```
void metodo1( ){
...
   if (z != x + y)
 ...
}
```

Exemplo de operadores – ferramenta μJava

Operador	Descrição
AOR	Substituição de operador aritmético por outro operador aritmético: unário, binário e de atalho (++, --)
AOI	Inserção de operador aritmético unário / de atalho
AOD	Supressão de operador aritmético unário / de atalho
ROR	Substituição de operador relacional por outro operador relacional
COR	Substituição de operador condicional por outro: &&, ||, &, |, ^
COI	Inserção de operador condicional unário (!)
COD	Supressão de operador condicional unário (!)
SOR	Substituição de operador de deslocamento por outro
LOR	Substituição de operador lógico (bitwise) por outro: &, |, ^
LOI	Inserção de operador lógico unário: ~
LOD	Supressão de operador lógico unário: ~
ASR	Substituição de operador de atribuição (de atalho) por outro: +=, -=, ...

Operadores para programas OO

- Existem várias propostas de operadores para criar falhas em aspectos específicos de OO, tais como:
  - Ocultação de informação (controle de acesso):
    - Operador para modificar o controle de acesso a atributos e métodos
  - Herança
    - Operadores para modificar uso de herança, tais como: suprimir "overriding" de métodos, suprimir "super", ...
  - ...

Exemplo de operadores OO - µJava

Operador	Descrição
IOD	Suprime método sobrescrito
IOR	Troca nome de método sobrescrito
ISI	Insere *super*
JTI	Insere this
OMD	Suprime método sobrecarregado
EAM	Troca método de acesso

Análise de mutantes vivos

- Mutante equivalente
  - Um mutante *M* é dito **equivalente** a *P* se para qualquer dado de entrada $d \in D$, o comportamento de *M* é igual ao de *P*: $M(d) = P(d)$
- Mutante revelador de falha (*fault-reveling)*
  - Um mutante é dito ser **revelador de falha** se para algum caso de teste *t*, tal que $P(t) = M(t)$, conclui-se que $P(t)$ não está de acordo com a especificação, ou seja, a presença de uma falha em *P* foi revelada
  - ☹ Se $t \notin T$ ➔ o que dizer de T?

Limitações

- Principal problema é o grande número de mutantes gerados.
  - Mutantes precisam ser compilados e executados;
  - Mutantes vivos precisam ser analisados devido a possível equivalência.
- Requer bom conhecimento da implementação do produto para facilitar a análise de mutantes vivos:
  - Determinar se um mutante é equivalente ao programa original: problema indecidível

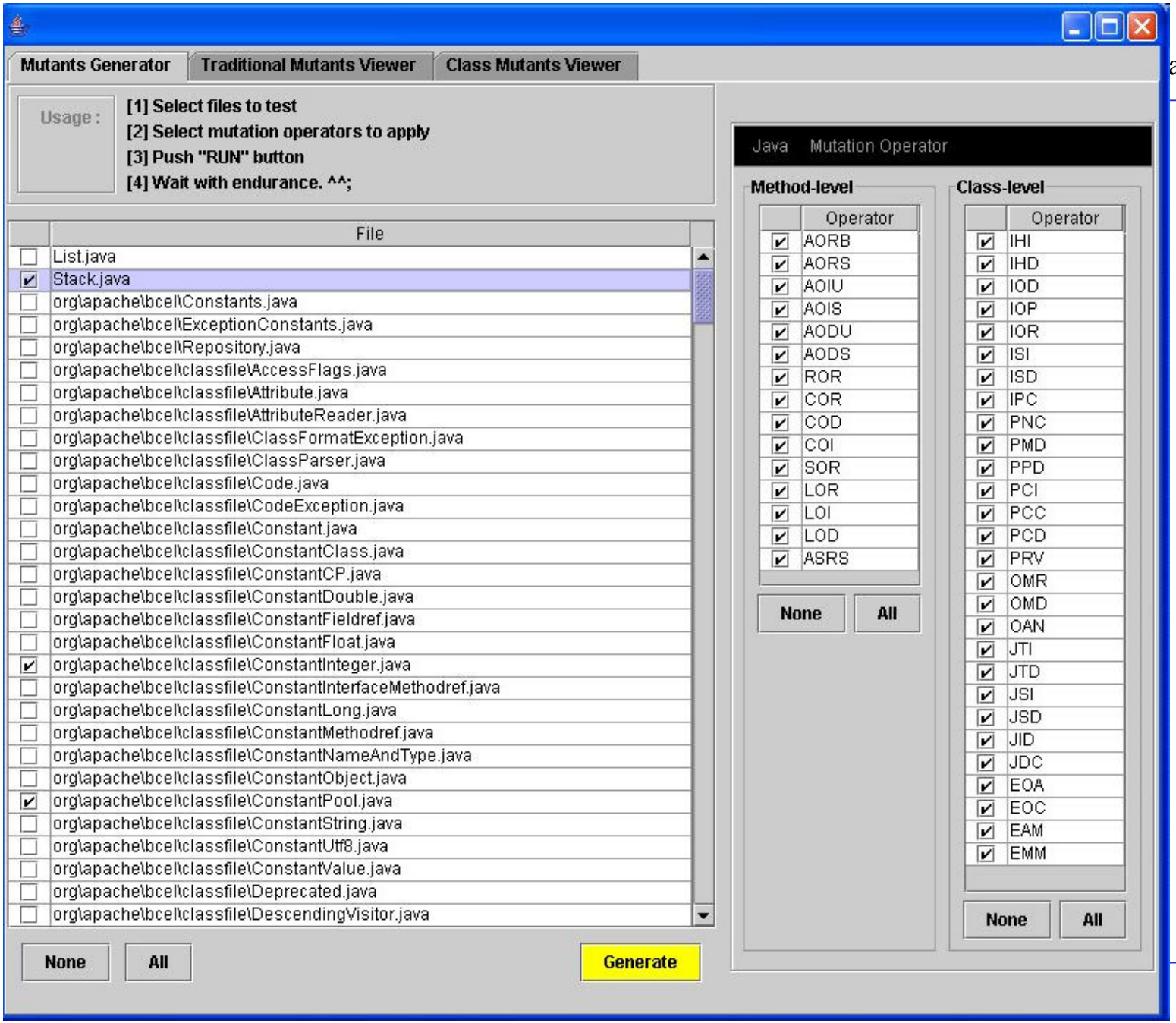
Vantagens

- É fácil de ser estendido para qualquer produto "executável", seja especificação ou implementação.
- É um dos critérios de teste mais eficaz em para revelar falhas.
- É fácil automatizar a geração dos mutantes

Ferramentas

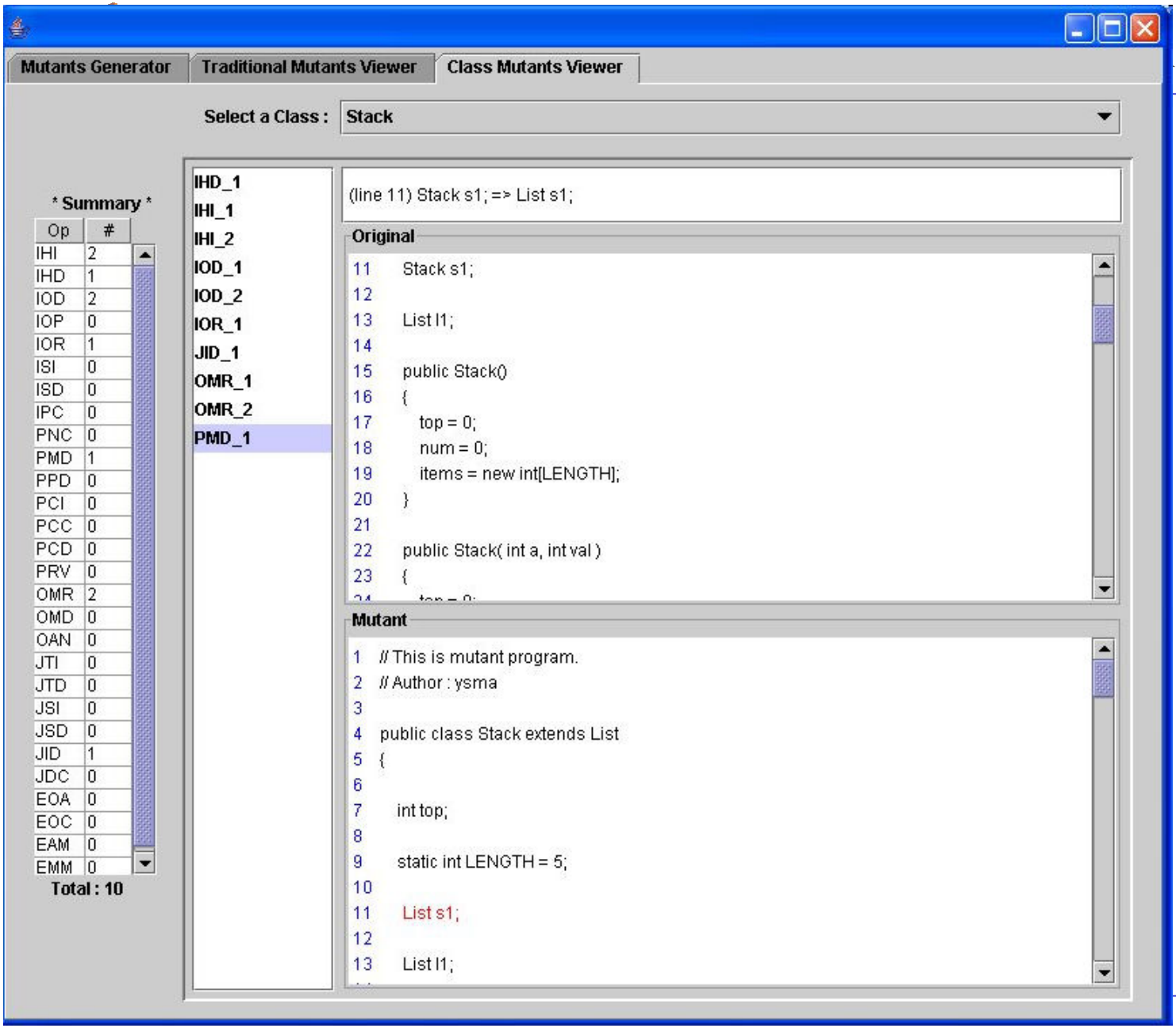
- Mothra (1980): mutação para programas em Fortran77
- Outras ferramentas:
  fonte: http://en.wikipedia.org/wiki/Mutation_testing

➢Jumble: Bytecode based mutation testing tool for Java
➢MuJava: Bytecode based mutation system for Java programs
➢ PIT: Bytecode based mutation testing tool for Java
➢ Jester: Source based mutation testing tool for Java
➢ Heckle: Mutation testing tool for Ruby
➢ Nester: Mutation testing tool for C#
➢Mutagenesis: Mutation testing tool for PHP
➢Proteum: Mutation testing tool for C
➢ ...



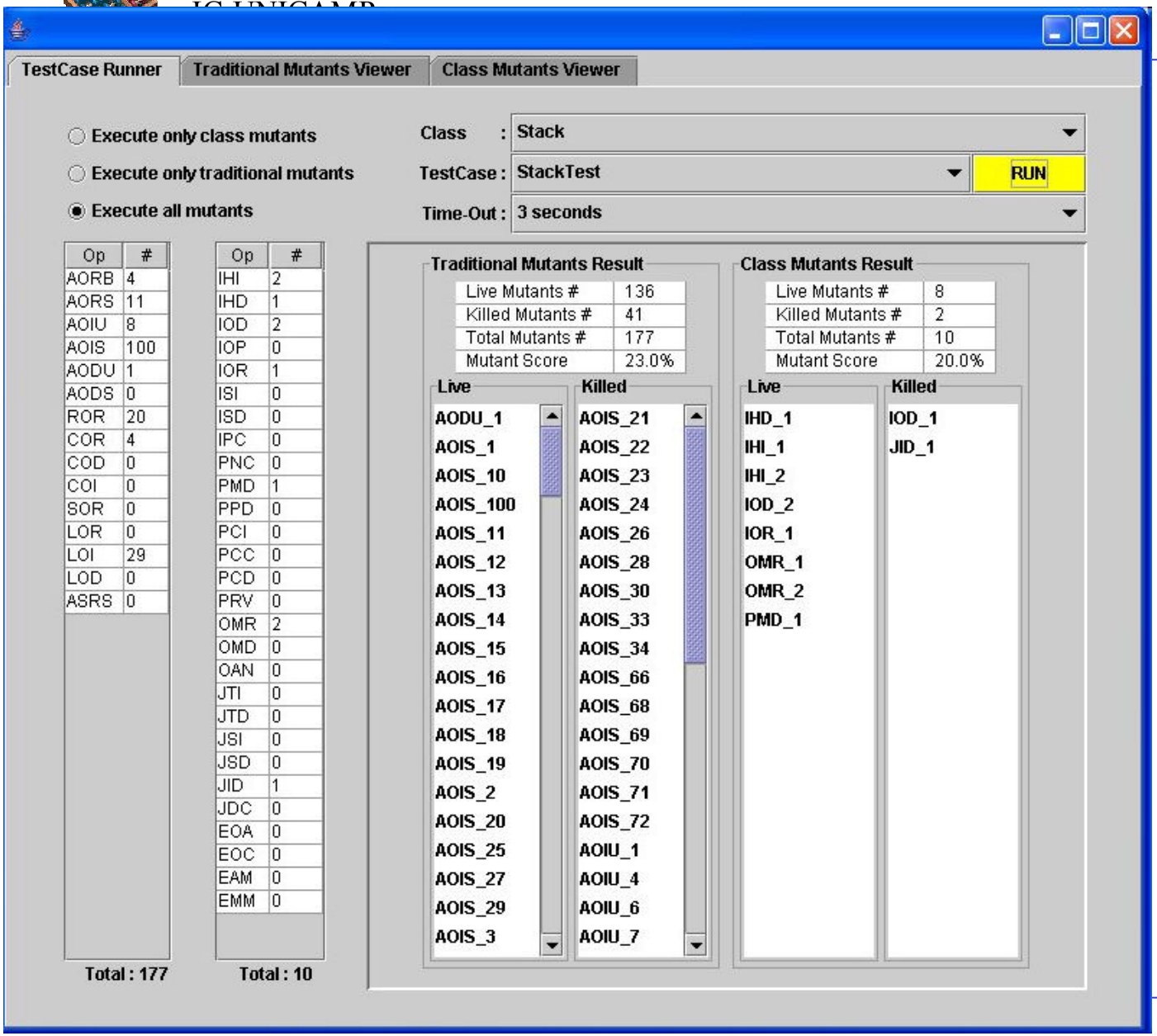
Seleção dos operadores de mutação e das classes

MuJava
Obtido em ago/11 em:
http://cs.gmu.edu/~offutt/mujava/



Visualização dos mutantes

MuJava
Obtido em ago/11 em:
http://cs.gmu.edu/~offutt/mujava/



Resultados dos testes de mutação

MuJava
Obtido em ago/11 em:
http://cs.gmu.edu/~offutt/mujava/

Exercício

```
// Soma elementos nas posições pares de um vetor de inteiros
// Obtém menor elemento em posição ímpar no vetor
1.  public class Exemplo {
2.  public static void main(String args[]) {
3.  int vet[] = {2, 5, 1, 8, 4, 9, 3, 7, 6, 8};

4.  int i;
5.  int somapar = vet[0];
6.  int menorimpar = vet[1];
7.  for (i=1; i<vet.length; i++) {
8.      if (i%2==0){
9.          somapar += vet[i];}
10.     else{
11.         if (vet[i] < menorimpar) {
12.             menorimpar = vet[i];}
        }
        }
13.     System.out.println("Soma dos pares = "+somapar);
14.     System.out.println("Menor nr. impar = "+menorimpar);
15.     if(menorimpar) {
16.     System.out.println("SomaPares/MenorImpar =
"+(somapar/menorimpar)); }
}
}
```

Exercício (cont.)

- Para o código dado:
  - Qual a resposta esperada para o vetor fornecido como entrada de teste?
  - Suponha que foi criado um mutante substituindo, na linha 7, `i=1` por `i=0`. A entrada de teste fornecida mataria esse mutante? Por que?
  - Crie uma entrada que mate esse mutante.
  - Crie uma entrada que não mate esse mutante.

Sumário principais pontos aprendidos