

BS450 Yazılım Test Mühendisliği

# Mutasyon Testi

# Kapsamın Dezavantajları

```
/**
 * Make sure Double.NaN is returned iff n = 0
 */
@Test
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    assertTrue(Double.isNaN(std.getResult()));
    std.increment(1d);
    assertEquals(0d, std.getResult(), 0);
}
```

# Kapsamın Dezavantajları

```
/**
 * Make sure Double.NaN is returned iff n = 0
 */
@Test
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    Double.isNaN(std.getResult());
    std.increment(1d);
    std.getResult();
}
```

# Kapsamın Dezavantajları

```
/**  
 * Make sure Double.NaN is returned iff n = 0  
 */  
@Test  
public void testNaN() {  
    StandardDeviation std = new StandardDeviation();  
    Double.isNaN(std.getResult());  
    std.increment(1d);  
    std.getResult();  
}
```

Kapsama miktarı  
değişmez

# Oracle Problemi

Kodun tamamını **çalıştırmak yeterli değildir**

İşlevsel davranışı **kontrol etmemiz** gerekir

Bu kod aslında **yapması gerekeni yapıyor mu?**

Otomatik oracle: spesifikasyon veya model

Aksi takdirde, manuel oracle tanımlanması gerekir



# Testlerim ne kadar iyi?

**Kapsam** = ne kadar kod çalıştırıldı?

Ama gerçekte ne kadarı **kontrol ediliyor**? 🤔

Maalesef hataların nerede olduğunu bilmiyoruz... 😭

Ama geçmişte yaptığımız hataları biliyoruz! 😎

# Hatalardan Öğrenmek

# Hatalardan Öğrenmek

**Anahtar fikir:** Daha önceki hatalardan ders alarak bunların tekrar olmasını **önlemek**



# Hatalardan Öğrenmek

**Anahtar fikir:** Daha önceki hatalardan ders alarak bunların tekrar olmasını **önlemek**

**Anahtar teknik:** Daha önceki hataları **simüle edin** ve ortaya çıkan kusurların bulunup bulunmadığını görün

# Hatalardan Öğrenmek

**Anahtar fikir:** Daha önceki hatalardan ders alarak bunların tekrar olmasını **önlemek**

**Anahtar teknik:** Daha önceki hataları **simüle edin** ve ortaya çıkan kusurların bulunup bulunmadığını görün

**Hata tabanlı (fault-based) test** veya **Mutasyon Testi** adı verilir

# Mutasyon Kapsamı

Kaynak kodun **mutasyonlarının** eklenmiş olduğu kopyalarını oluşturun.

Ör., “<” ifadesini “>” ile değiştirin veya “+” ifadesini “-” ile.

En fazla sayıda mutanti **öldürebilecek (kill) testi (seti)** bulun

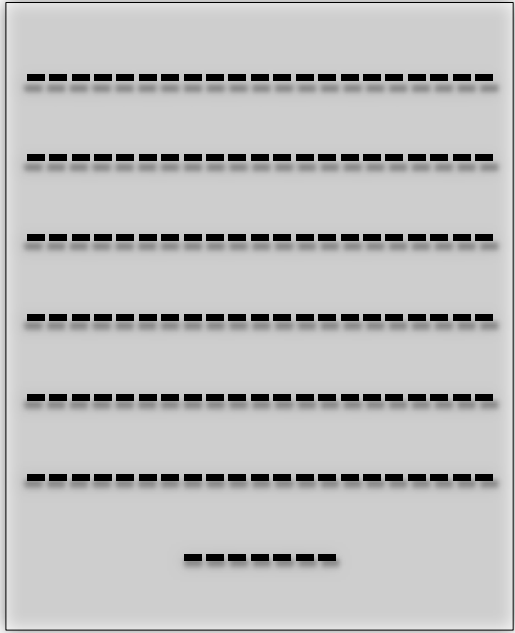
Ör., Testin **hem** orijinal program hem de mutasyona uğramış program üzerinde yürütülmesi **farklı çıktılara** yol açmalıdır.

Test 1

Test 2

Test 3

Original Program



Operators

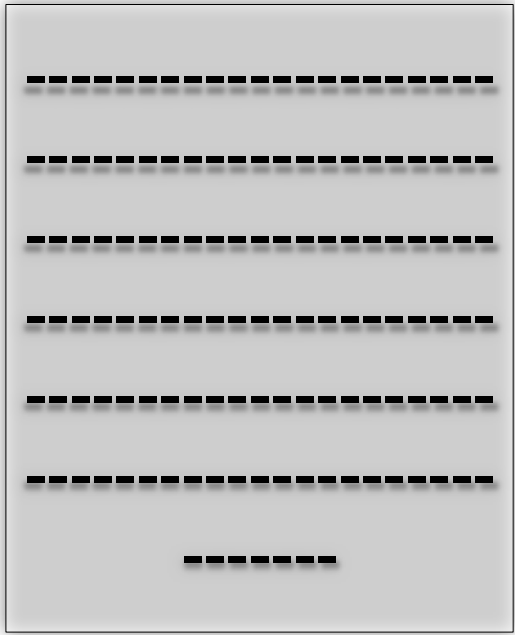


Test 1

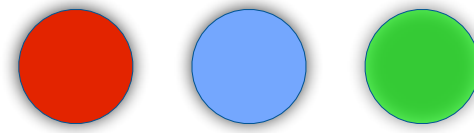
Test 2

Test 3

Original Program



Operators

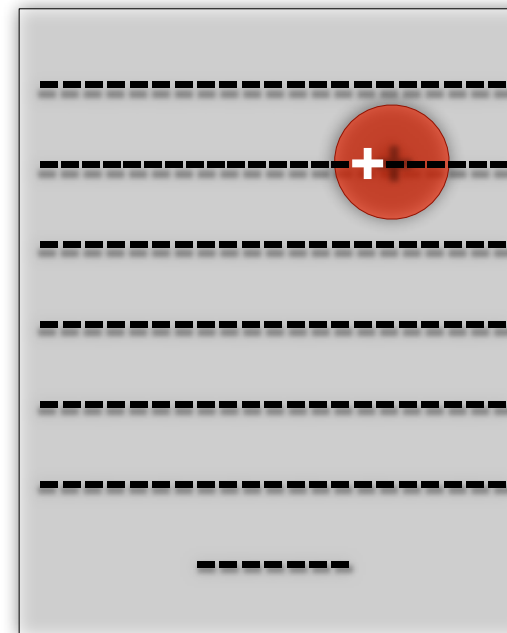
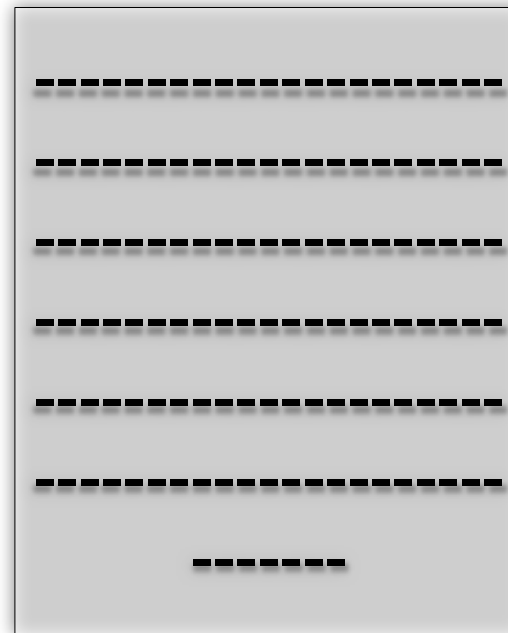


Test 1

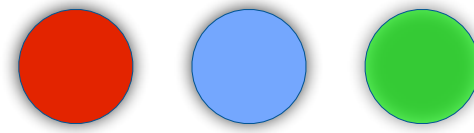
Test 2

Test 3

Original Program



Operators

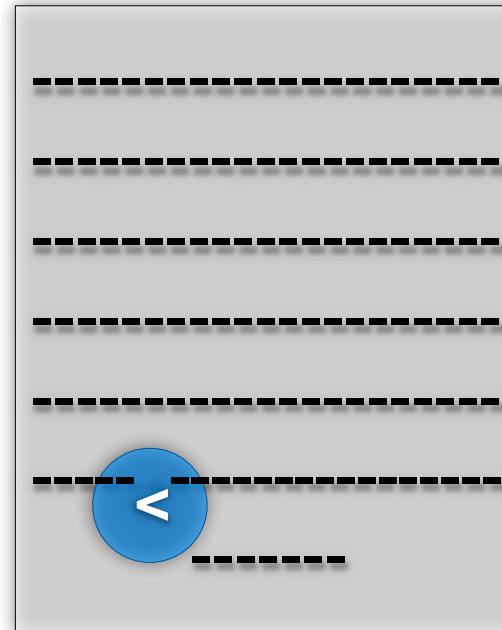
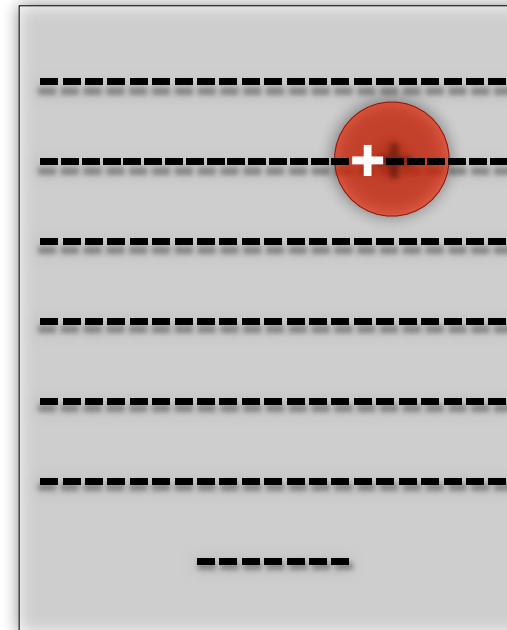
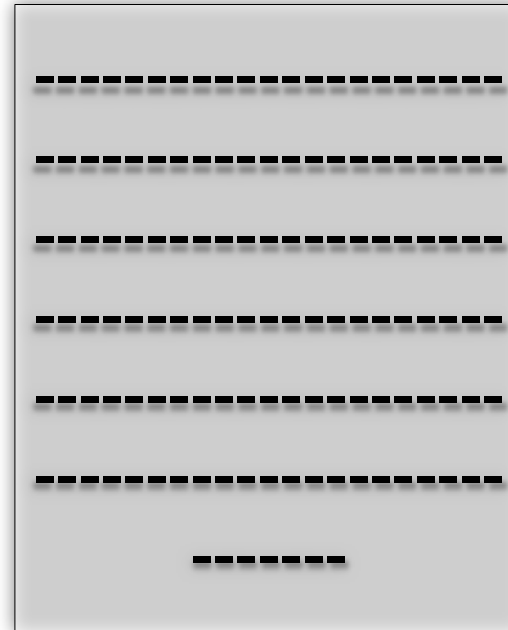


Test 1

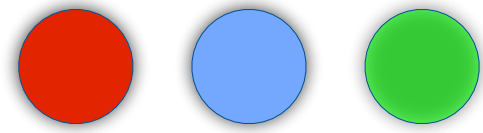
Test 2

Test 3

Original Program



# Operators

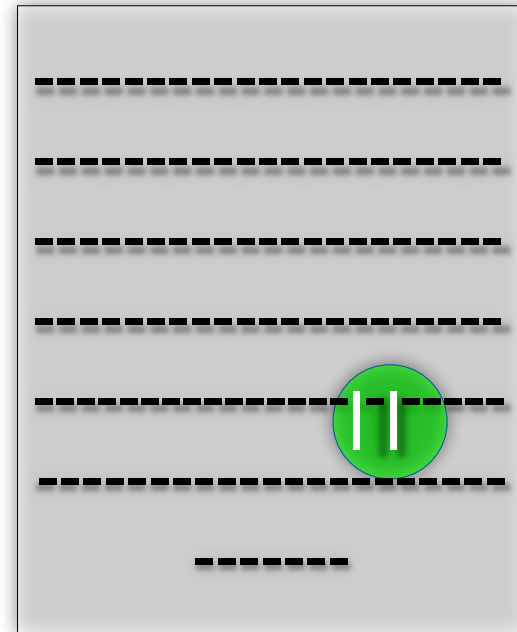
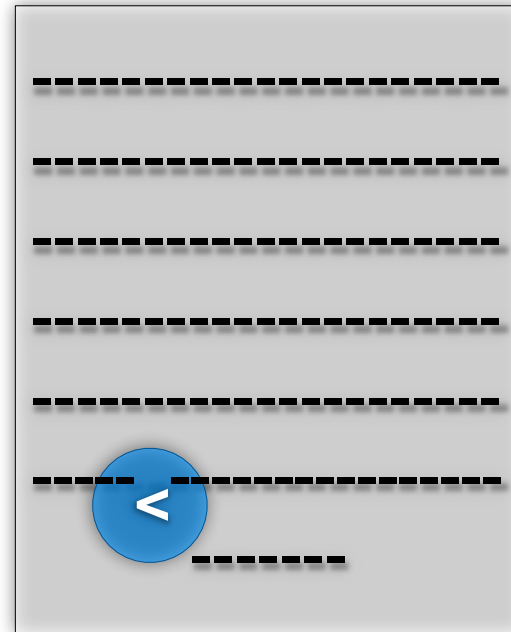
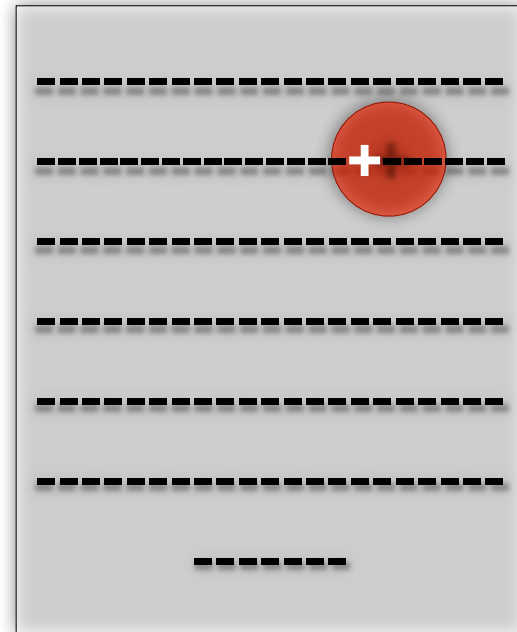
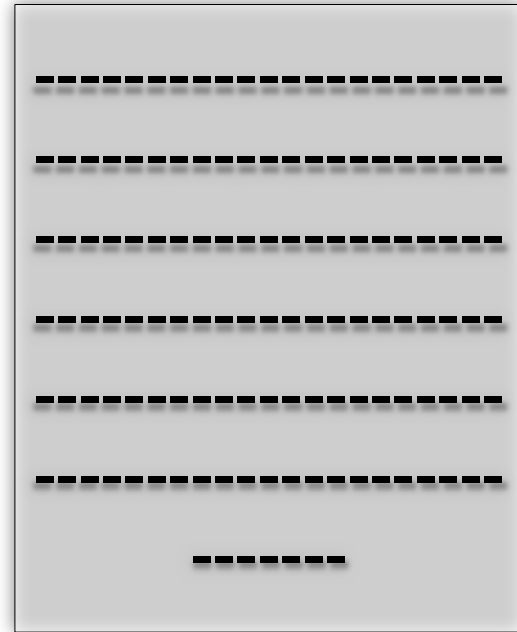


Test 1

Test 2

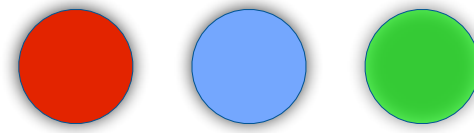
Test 3

# Original Program

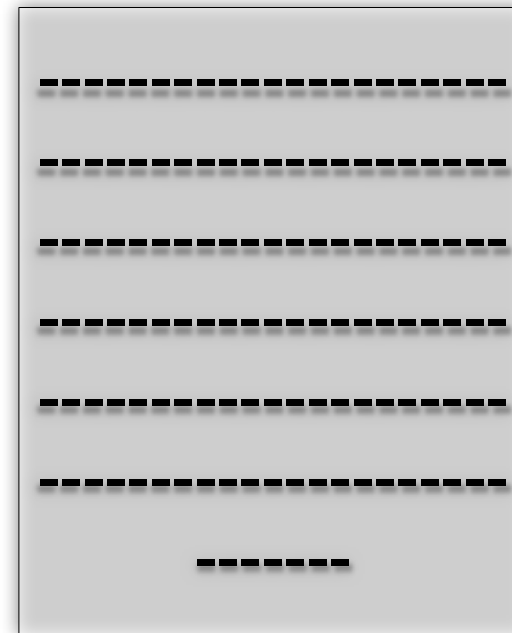




Operators



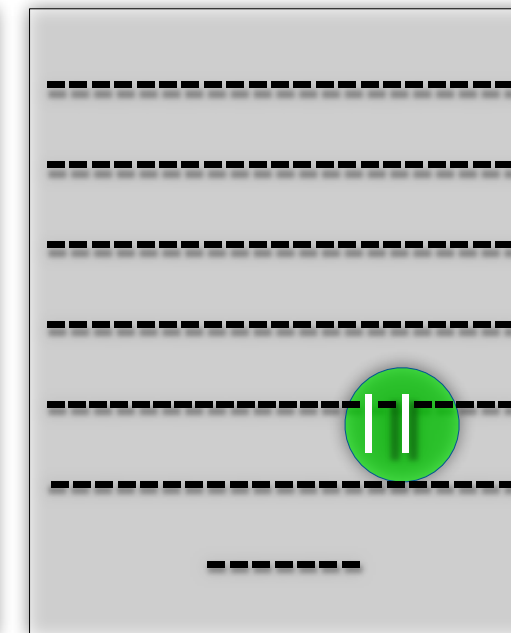
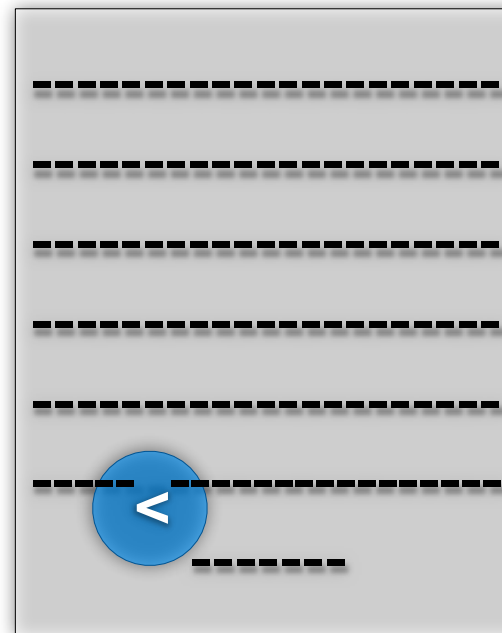
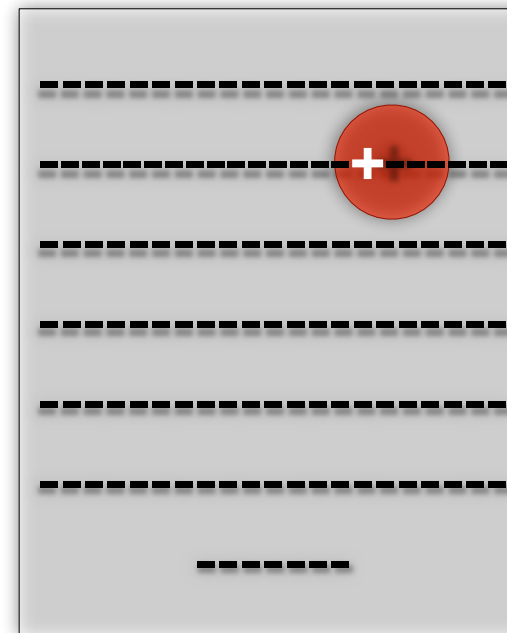
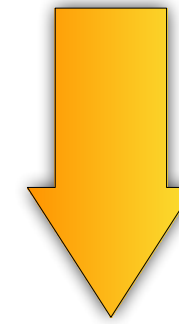
Original Program



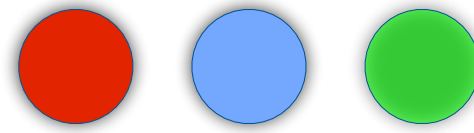
Test 1

Test 2

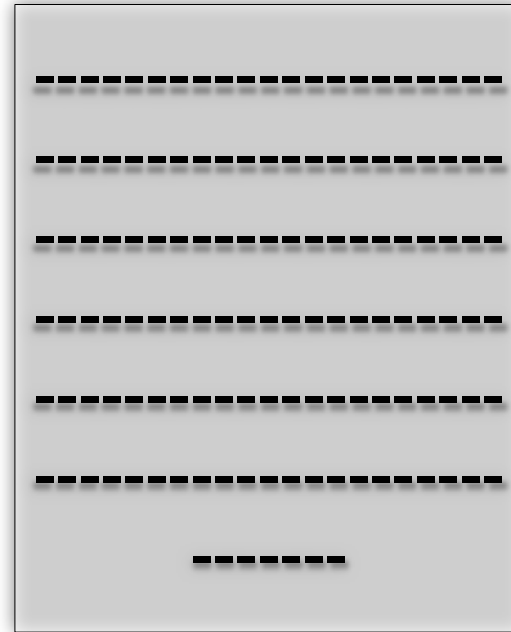
Test 3



Operators



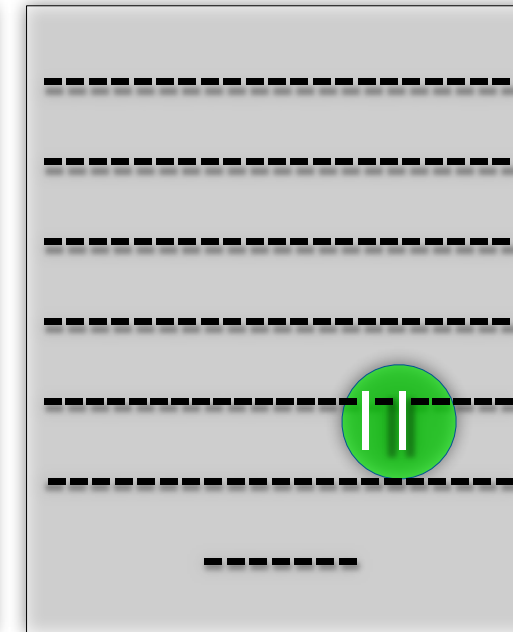
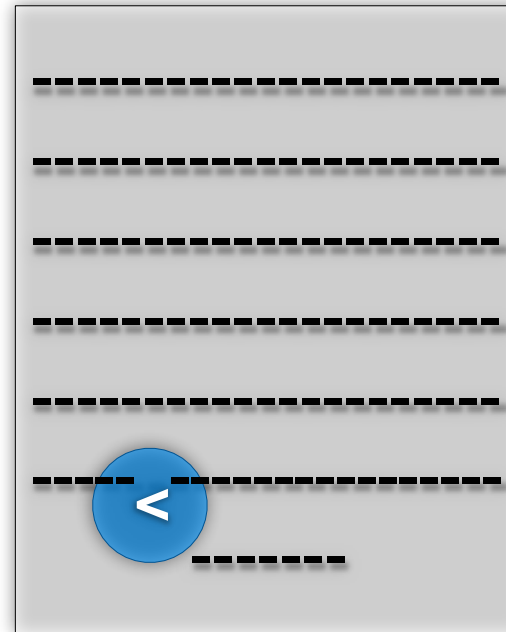
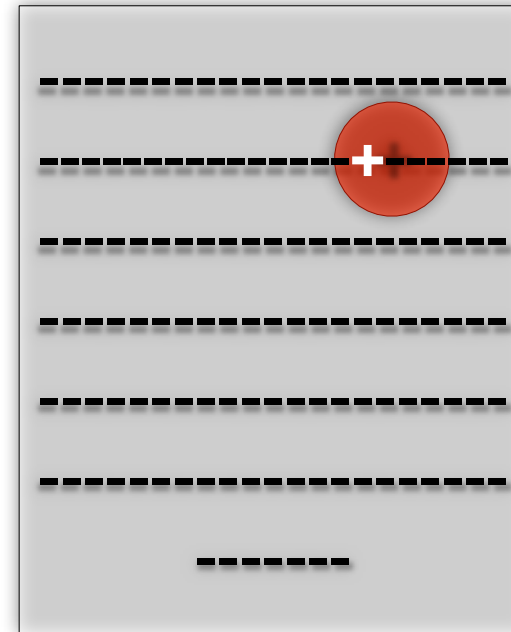
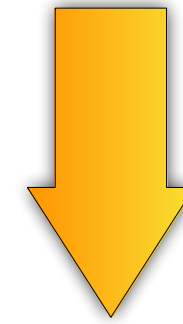
Original Program



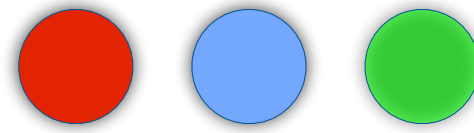
Test 1

Test 2

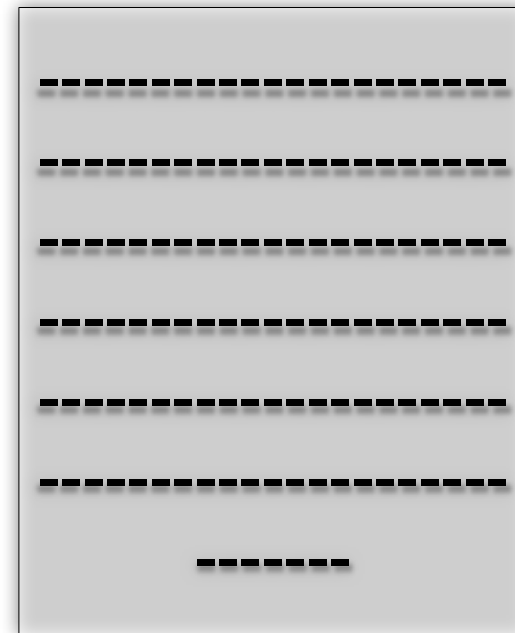
Test 3



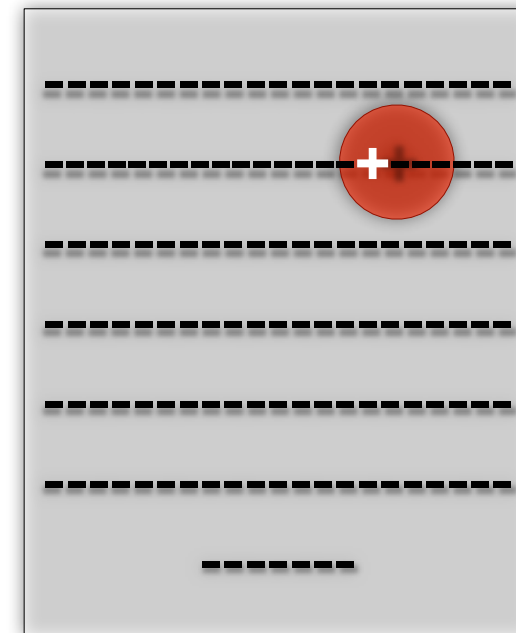
Operators



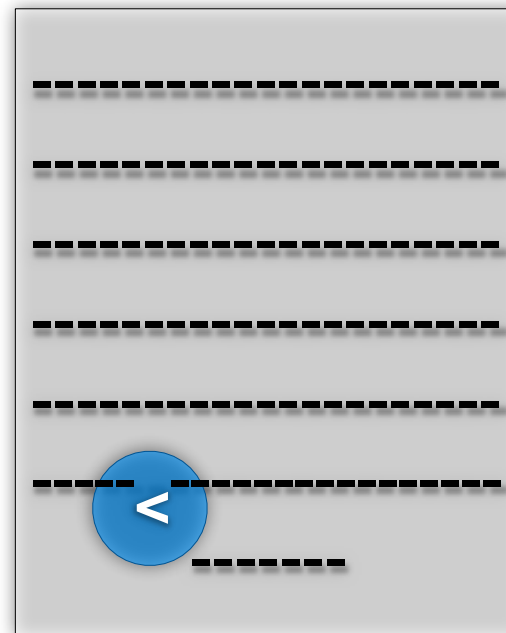
Original Program



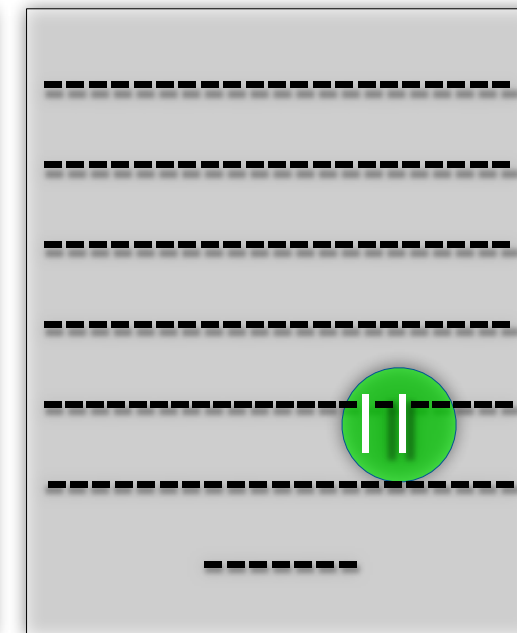
Test 1



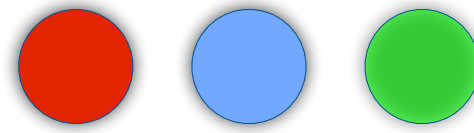
Test 2



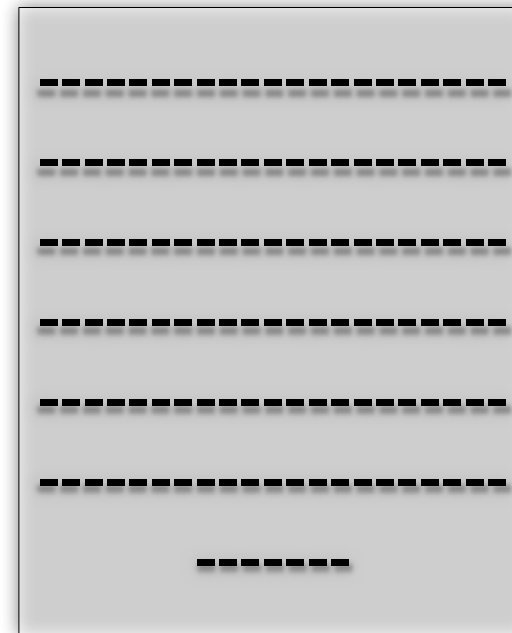
Test 3



Operators



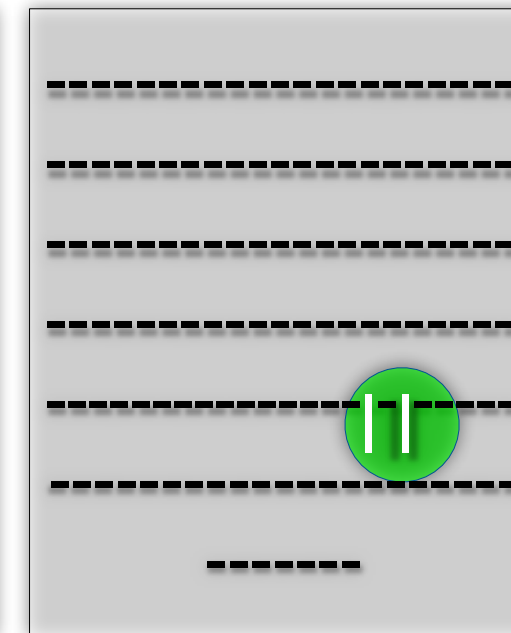
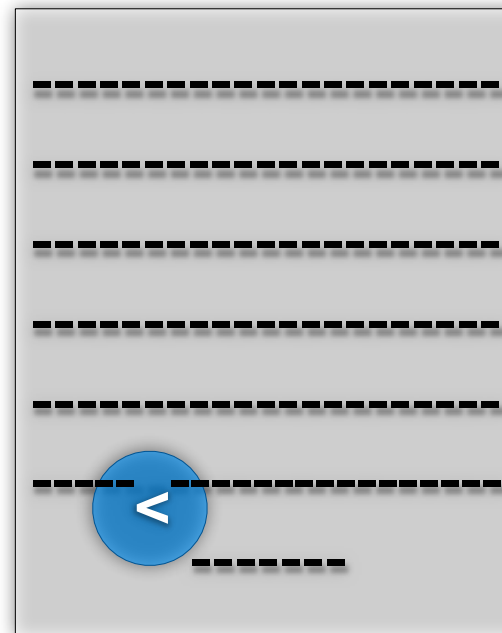
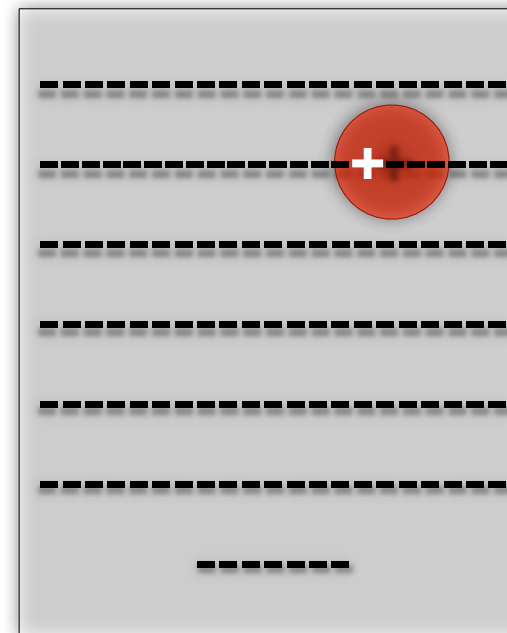
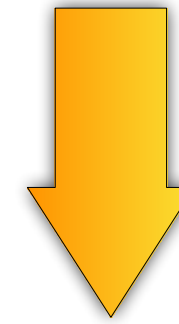
Original Program



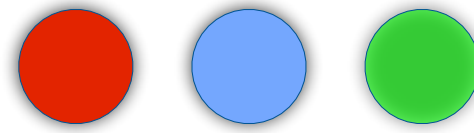
Test 1

Test 2

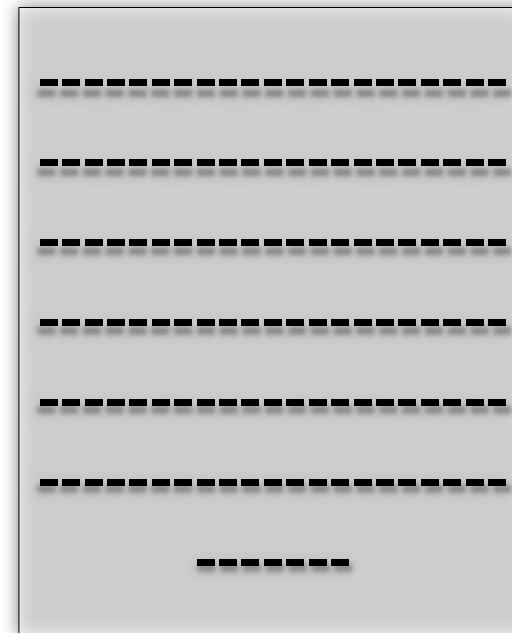
Test 3



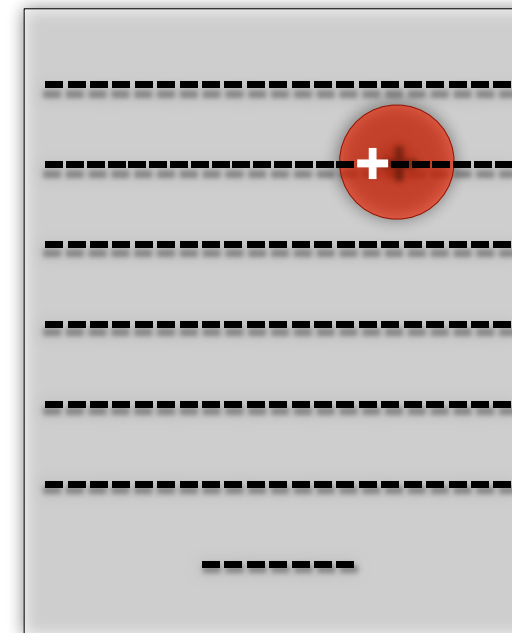
Operators



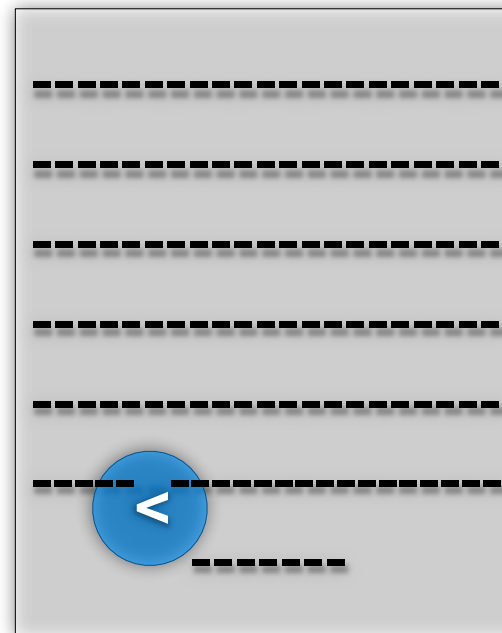
Original Program



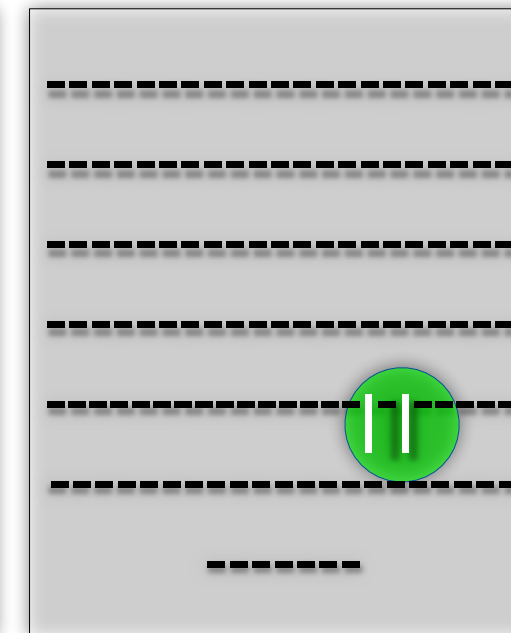
Test 1

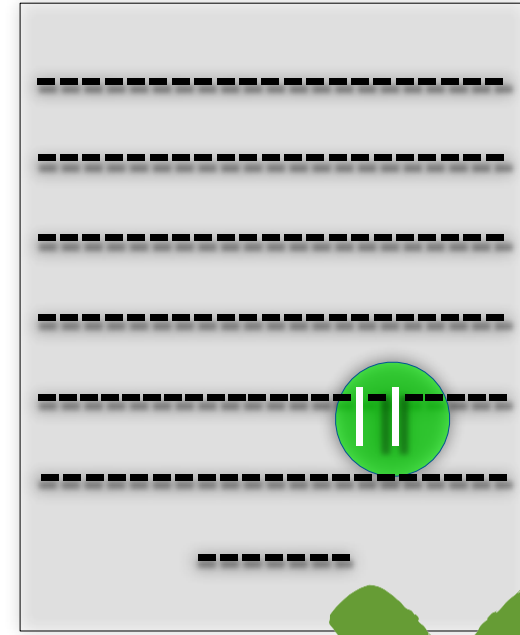


Test 2

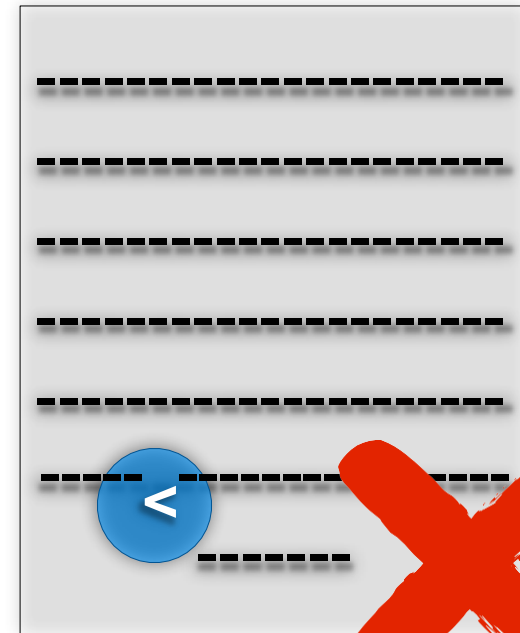


Test 3

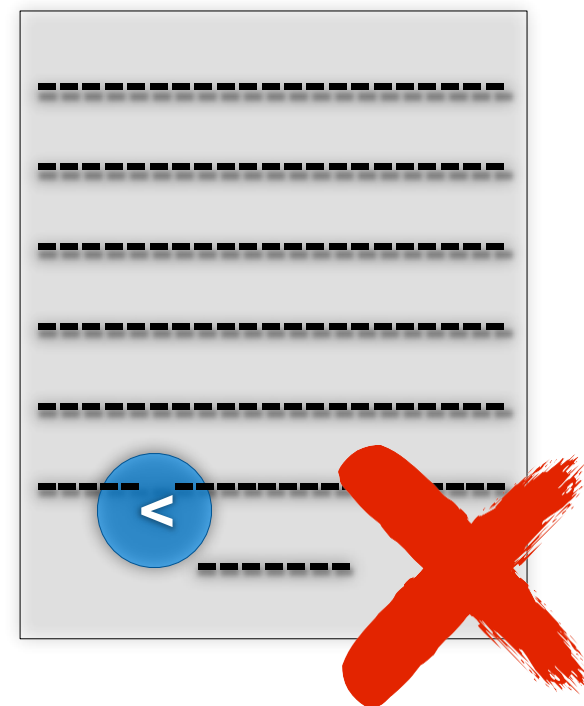
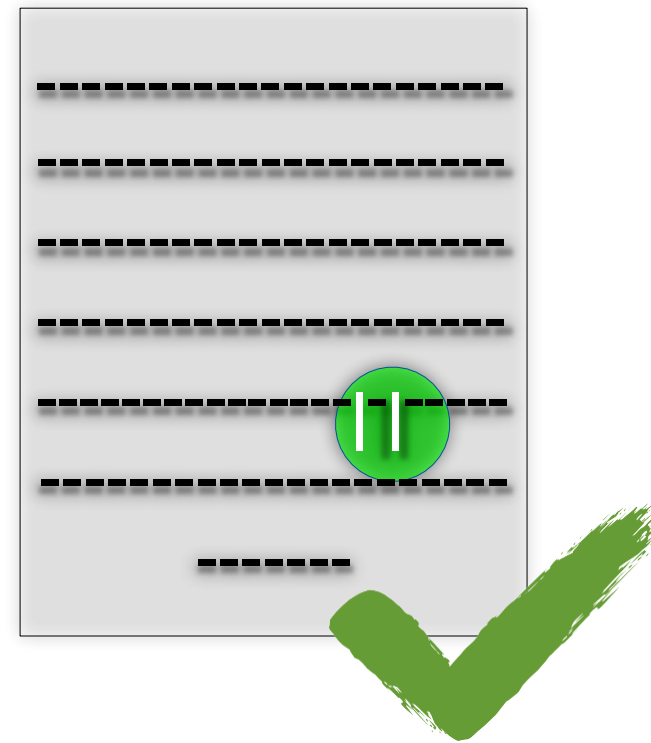




**Canlı mutant** – daha fazla teste  
ihtiyaç var



**Ölü mutant** – daha fazla teste  
gerek yok



**Mutation Score**

Killed Mutants

---

Total Mutants

```
int do_something(int x, int y)
{
    if(x < y)
        return x+y;
    else
        return x*y;
}
```

**Program**



```
int do_something(int x, int y)
{
    if(x < y)
        return x+y;
    else
        return x*y;
}
```

**Program**

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

**Test**

```
int do_something(int x, int y)
{
    if(x < y)
        return x+y;
    else
        return x*y;
}
```

**Program**

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

**Test**



```
int do_something(int x, int y)
{
    if(x < y)
        return x+y;
    else
        return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test



```
int do_something(int x, int y)
{
    if(x < y)
        return x-y;
    else
        return x*y;
}
```

Mutant

```
int do_something(int x, int y)
{
    if(x < y)
        return x+y;
    else
        return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test



```
int do_something(int x, int y)
{
    if(x < y)
        return x-y;
    else
        return x*y;
}
```

Mutant

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
    if(x < y)
        return x+y;
    else
        return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test



```
int do_something(int x, int y)
{
    if(x < y)
        return x-y;
    else
        return x*y;
}
```

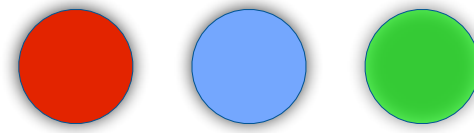
Mutant

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

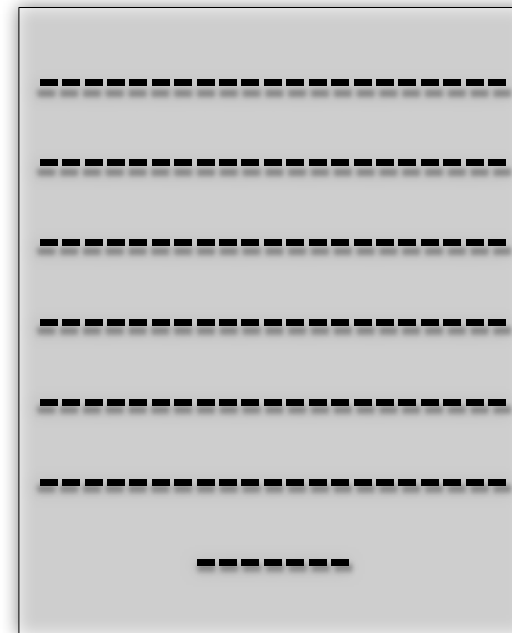
Test



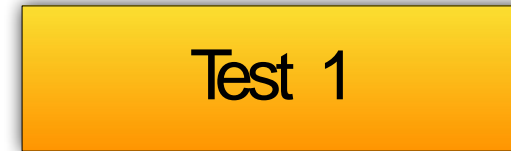
Operators



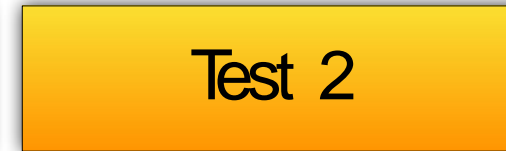
Original Program



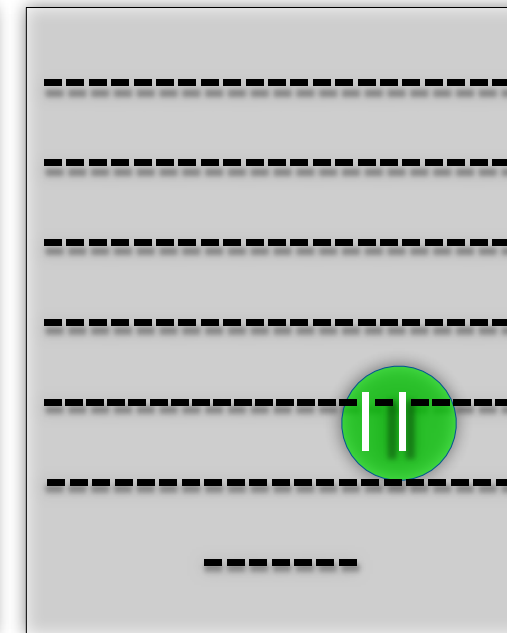
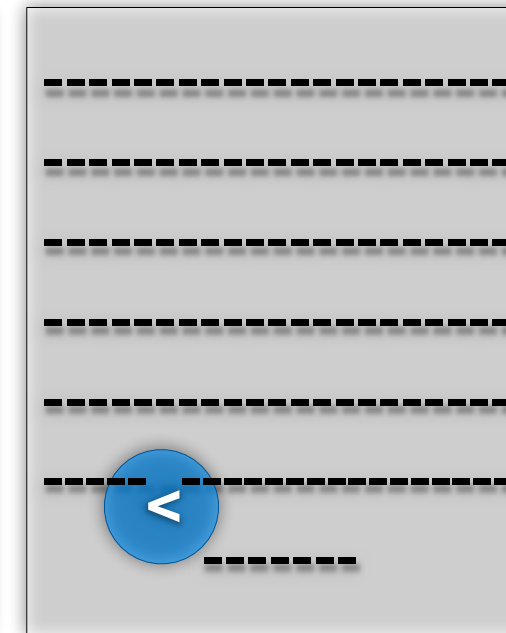
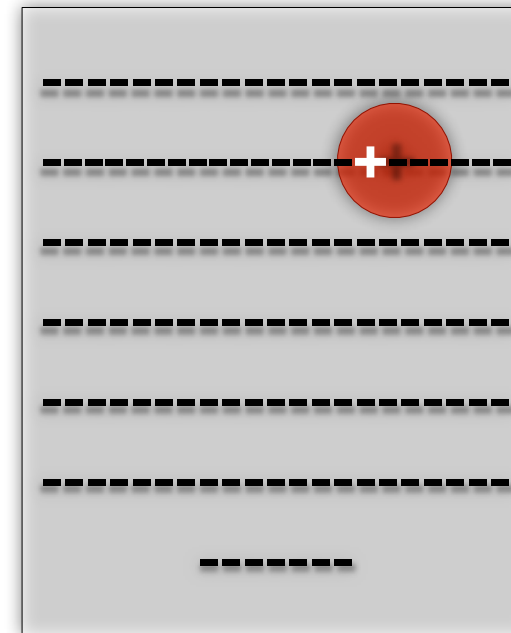
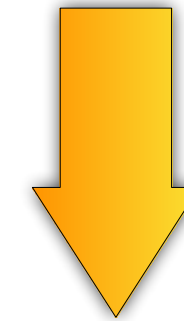
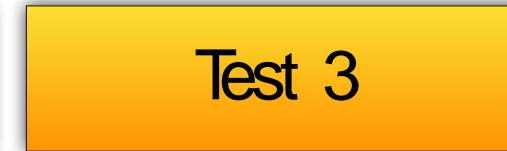
Test 1

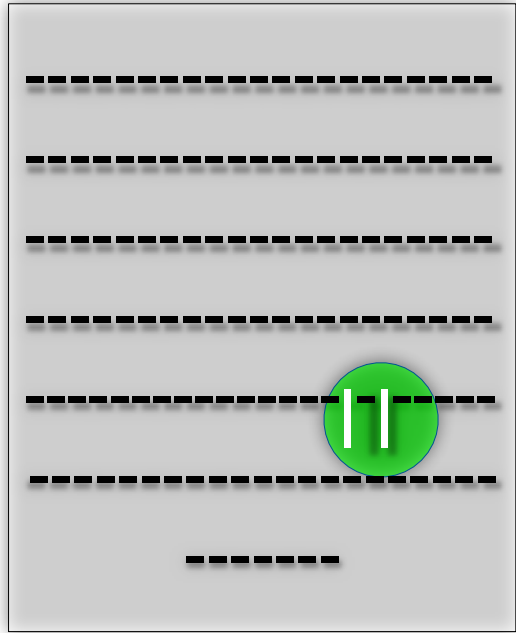
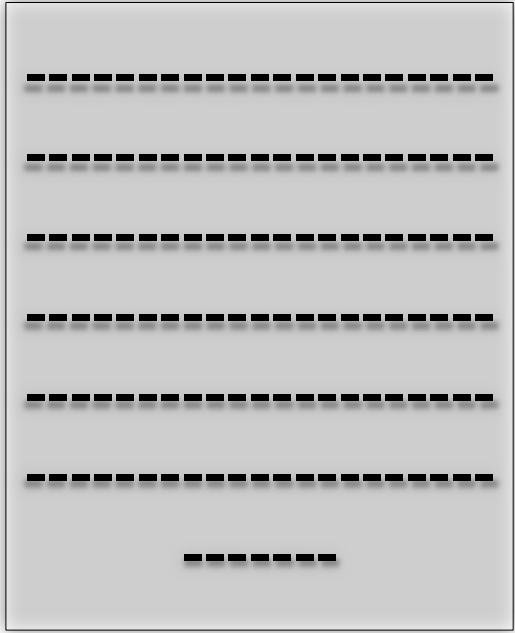


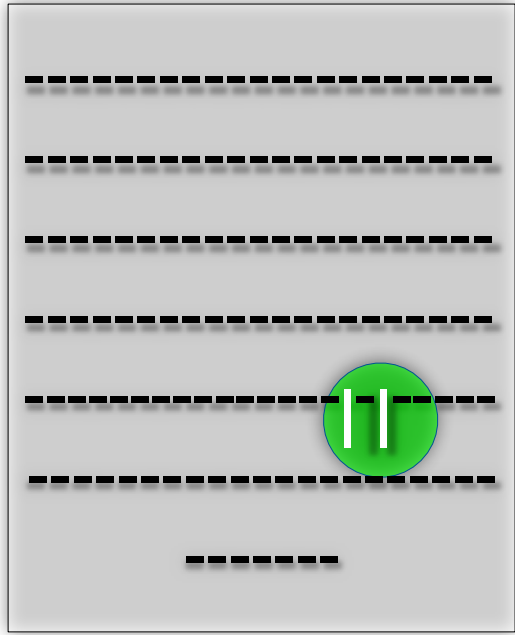
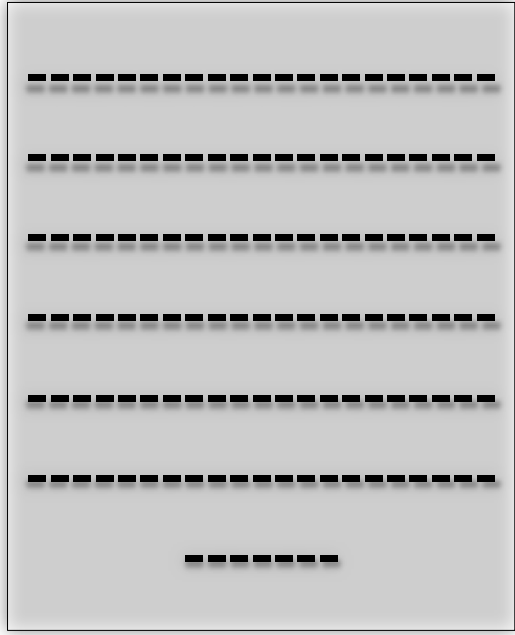
Test 2



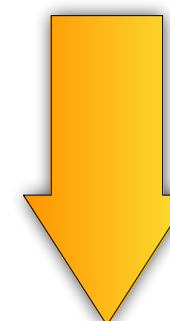
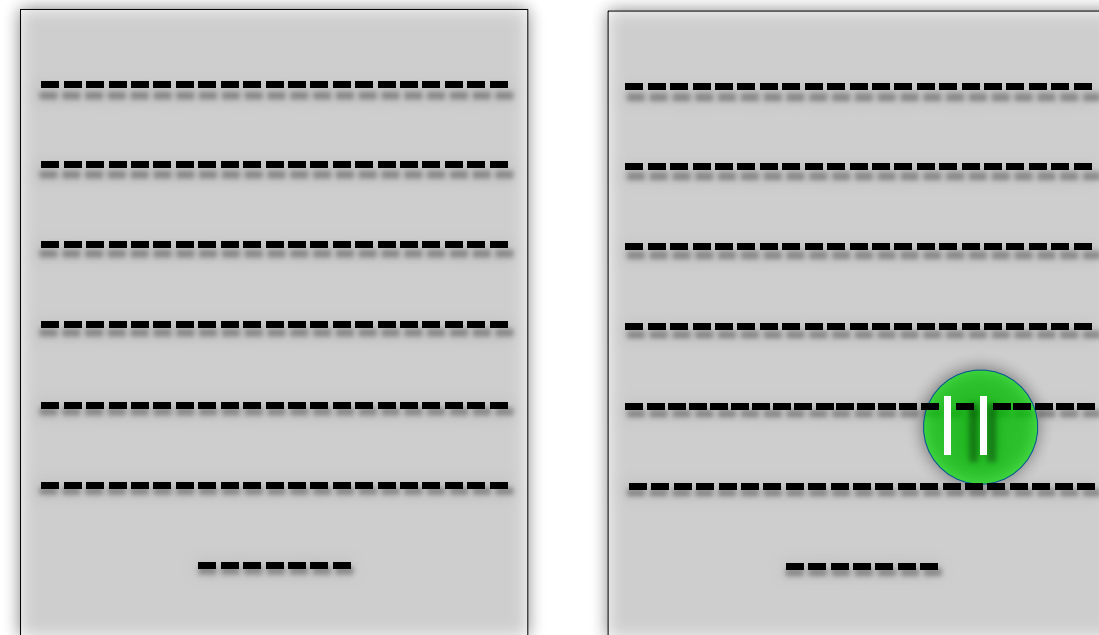
Test 3





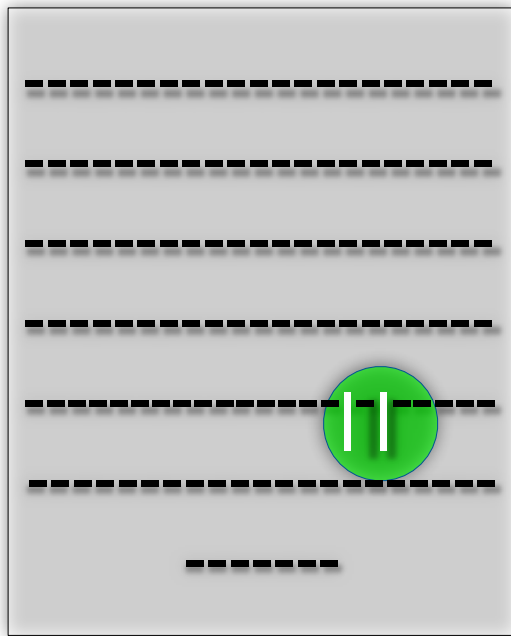
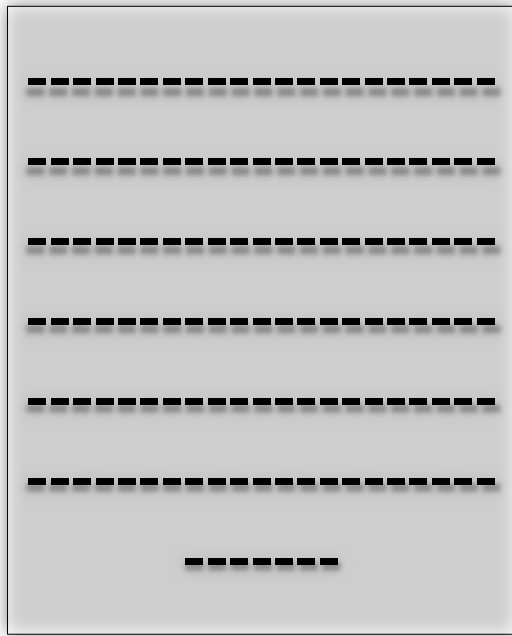
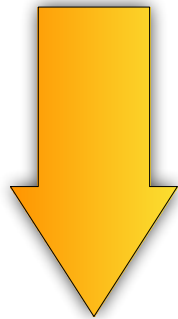




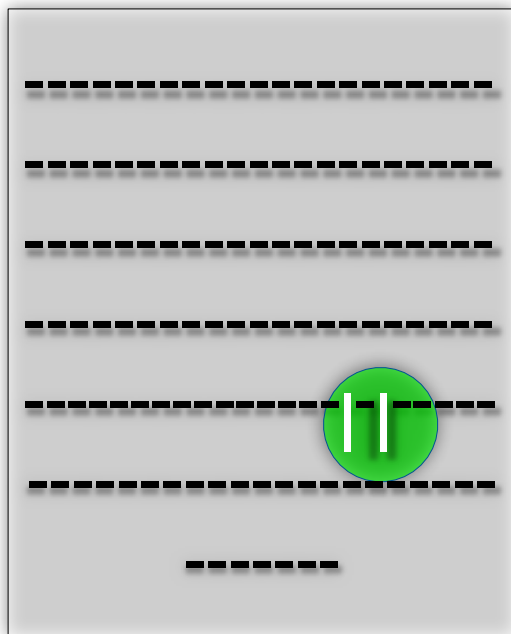
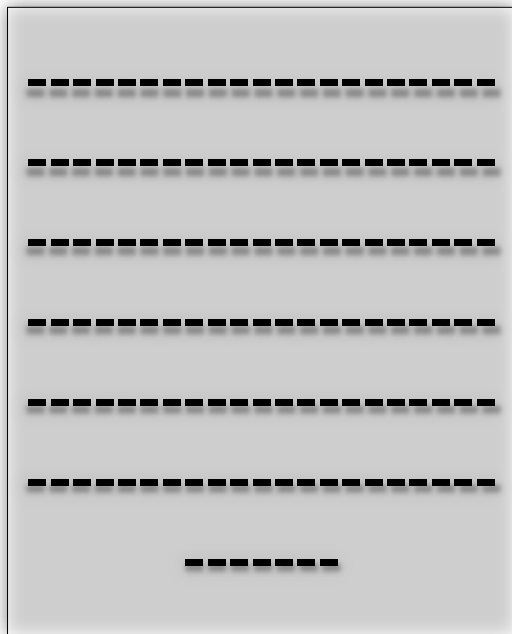
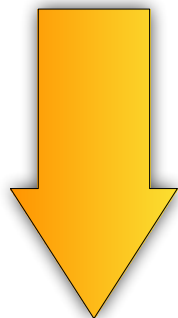


Test 4

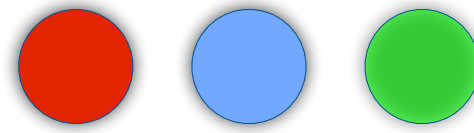
Test 4



Test 4



Operators

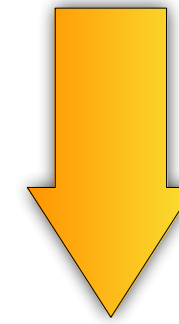


Test 1

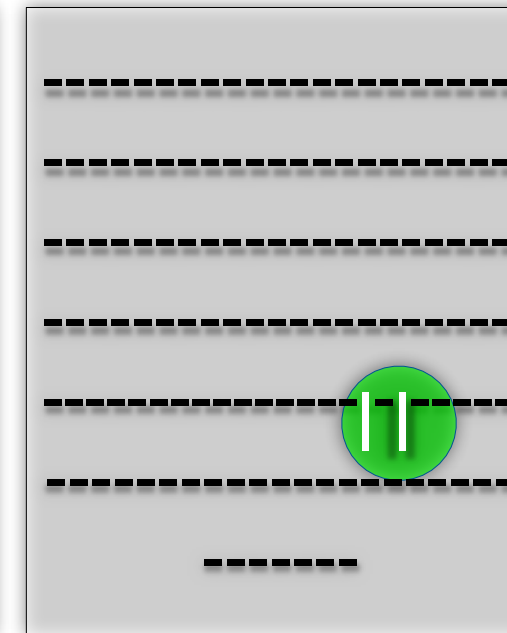
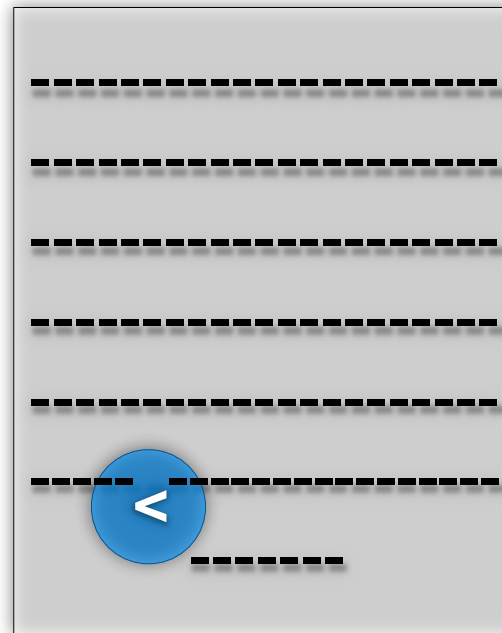
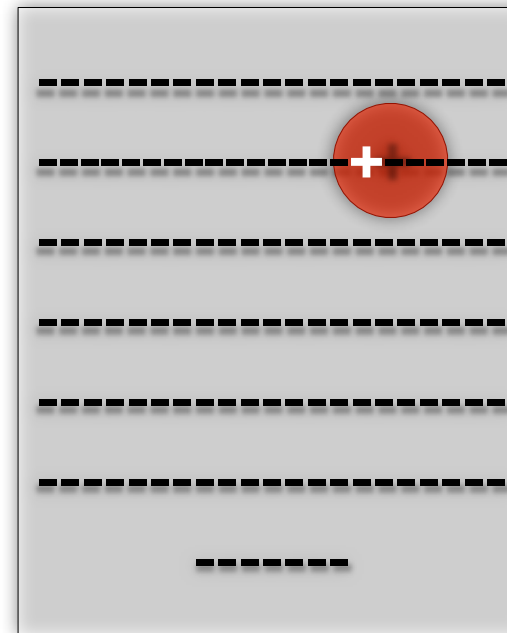
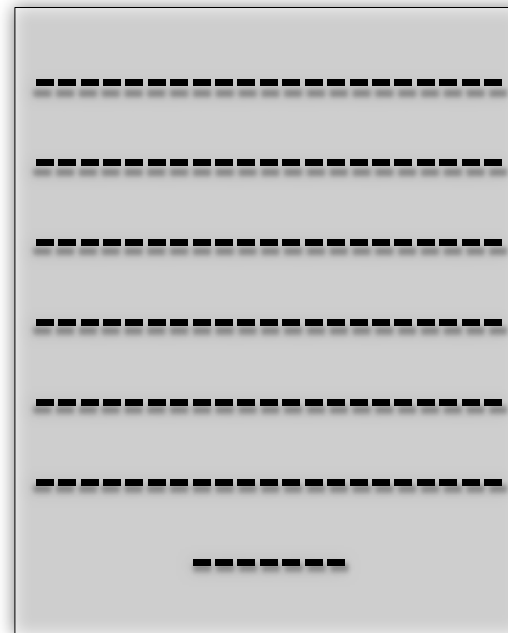
Test 2

Test 3

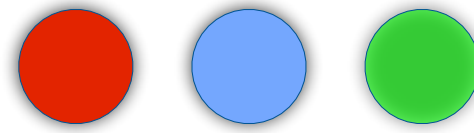
Test 4



Original Program



Operators

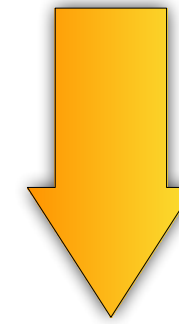


Test 1

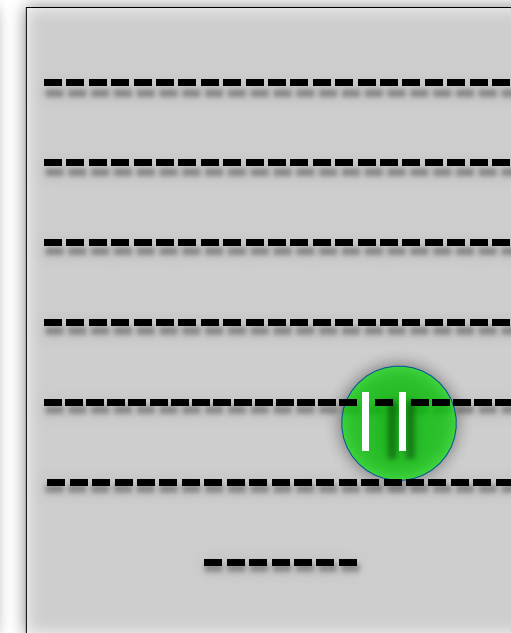
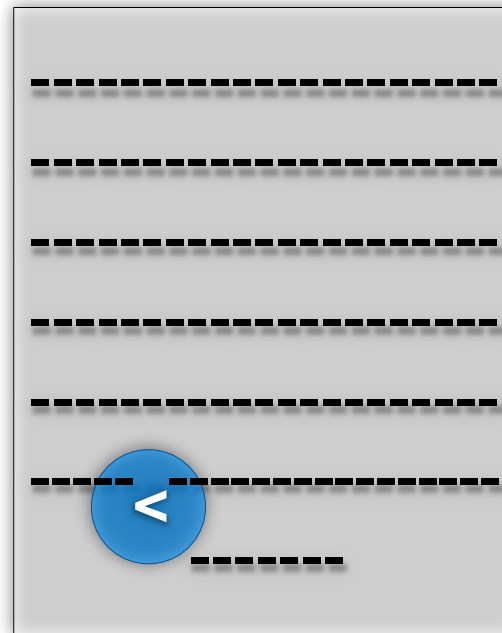
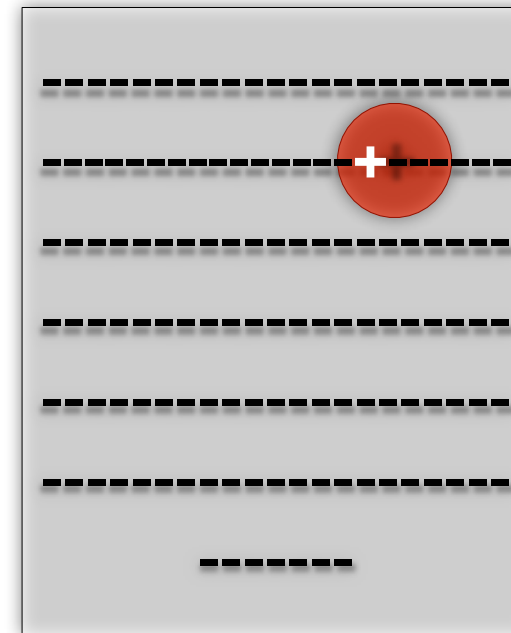
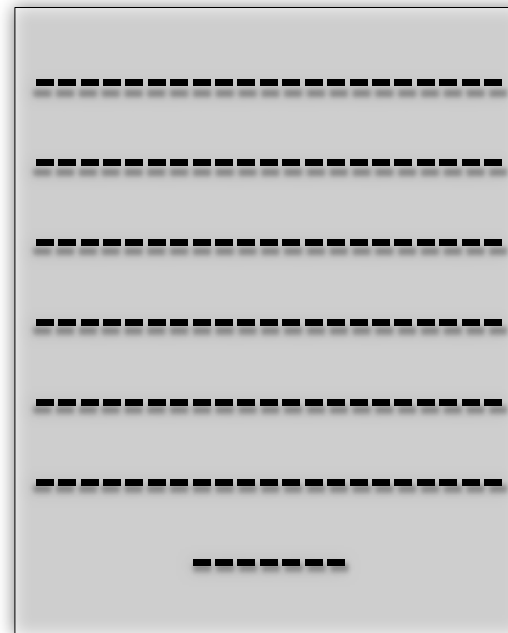
Test 2

Test 3

Test 4



Original Program



# Mutantlar

Orijinal programın biraz değiştirilmiş versiyonudur.  
Sözdizimi değişikliğidir yani geçerli ve derlenebilir bir kod elde edilir.

Basit, programlama hatasına benzeyen, hata hiyerarşisindeki hatalara dayalı değişikliklerdir.

# Mutantlar Üretmek

## Mutasyon **operatörleri**

Bir programdan mutant türetme kurallarıdır

## Gerçek **hatalara dayalı** mutasyonlar

Mutasyon operatörleri tipik hataları temsil eder

Çoğu dil için özel mutasyon operatörleri tanımlanmıştır. Örneğin, C programlama dili için 100'den fazla operatör bulunur.

Bazı örnek operatörler...

# ABS - Absolute Value Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```



# ABS - Absolute Value Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y != 0) {  
        tmp = x % y;  
        x = abs(y);  
        y = tmp;  
    }  
  
    return x;  
}
```

# ABS - Absolute Value Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x % y;  
        x = abs(y);  
        y = tmp;  
    }  
  
    return x;  
}
```

# ABS - Absolute Value Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y != 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return -abs(x) ;  
}
```

# ABS - Absolute Value Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return -abs(x);  
}
```

# AOR - Arithmetic Operator

```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y != 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# AOR - Arithmetic Operator

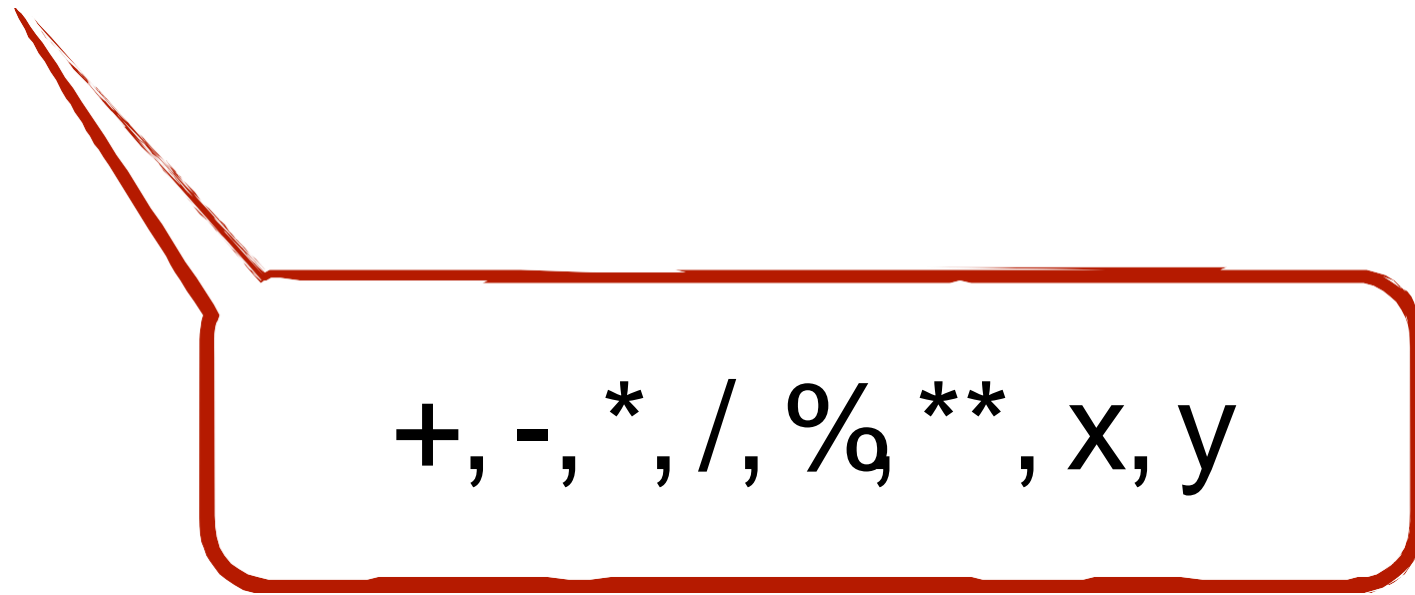
```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y != 0) {  
        tmp = x * y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# AOR - Arithmetic Operator

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x * y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# AOR - Arithmetic Operator

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x * y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```



+, -, \*, /, %, \*\*, x, y



# ROR - Relational Operator

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# ROR - Relational Operator

```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y > 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# ROR - Relational Operator

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y > 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# ROR - Relational Operator

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y > 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

<, >, <=, >=, =,  
!=, false, true

# COR - Conditional Operator

```
if(a && b)
```

# COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

# COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

```
if(a & b)
```

# COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

```
if(a & b)
```

```
if(a | b)
```



# COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

```
if(a & b)
```

```
if(a | b)
```

```
if(a ^ b)
```

# COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

```
if(a & b)
```

```
if(a | b)
```

```
if(a ^ b)
```

```
if(false)
```

# COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

```
if(a & b)
```

```
if(a | b)
```

```
if(a ^ b)
```

```
if(false)
```

```
if(true)
```

# COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

```
if(a & b)
```

```
if(a | b)
```

```
if(a ^ b)
```

```
if(false)
```

```
if(true)
```

```
if(a)
```

# COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

```
if(a & b)
```

```
if(a | b)
```

```
if(a ^ b)
```

```
if(false)
```

```
if(true)
```

```
if(a)
```

```
if(b)
```

# UOI - Unary Operator Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y != 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# UOI - Unary Operator Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y != 0) {  
        tmp = x % +y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# UOI - Unary Operator Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x % +y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```



# UOI - Unary Operator Insertion

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x % +y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```



+, -, !, ~, ++, --

# SVR - Scalar Variable

```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y != 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# SVR - Scalar Variable

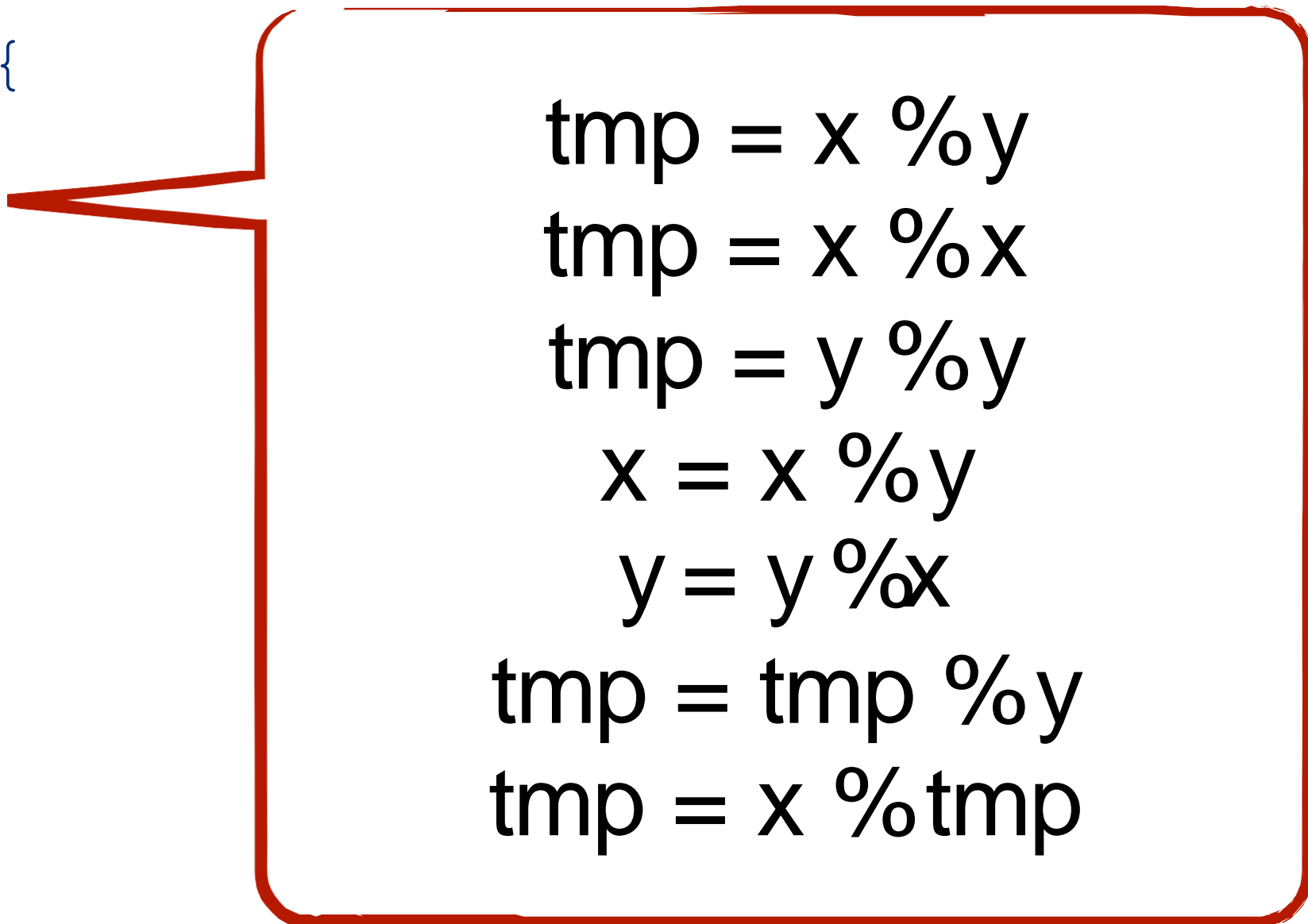
```
int gcd(int x, int y) {  
    int tmp;  
  
    while(y != 0) {  
        tmp = y % x;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# SVR - Scalar Variable

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = x % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```

# SVR - Scalar Variable

```
int gcd(int x, int y) {  
    int tmp;  
  
    while (y != 0) {  
        tmp = y % y;  
        x = y;  
        y = tmp;  
    }  
  
    return x;  
}
```



tmp = x % y  
tmp = x % x  
tmp = y % y  
x = x % y  
y = y % x  
tmp = tmp % y  
tmp = x % tmp

# OO Mutation

Şu ana kadar operatörler yalnızca metot gövdelerine göreymişti.

Sınıf düzeyindeki öğeler de mutasyona uğratılabilir:

# OO Mutation

Şu ana kadar operatörler yalnızca metot gövdelerine göreymi.

Sınıf düzeyindeki öğeler de mutasyona uğratılabilir:

```
public class test {  
    // ..  
    protected void do() {  
        // ...  
    }  
}
```

# OO Mutation

Şu ana kadar operatörler yalnızca metot gövdelerine göreydi.

Sınıf düzeyindeki öğeler de mutasyona uğratılabilir:

```
public class test {  
    // ..  
    protected void do() {  
        // ...  
    }  
}
```

```
public class test {  
    // ..  
    private void do() {  
        // ...  
    }  
}
```



# OO Mutation

AMC - Access Modifier Change

HVD - Hiding Variable Deletion

HVI - Hiding Variable Insertion

OMD - Overriding Method Deletion

OMM - Overridden Method Moving

OMR - Overridden Method Rename

SKR - Super Keyword Deletion

PCD - Parent Constructor Deletion

ATC - Actual Type Change

# Temel Hipotezler

## Yetenekli Programcı Hipotezi

Programcılar, doğru program kümesinin genel komşuluğunda olan, yani çoğunlukla doğru olan programları yazma eğilimindedirler.

## Bağlama Etkisi

Tüm programları sadece basit hatalardan farklılaştıran test verisi, o kadar hassastır ki, daha karmaşık hataları da dolaylı olarak ayırt eder.

Bu nedenle, mutasyon testi **birinci derece mutasyonlara** odaklanır.

Bir seferde tek mutasyon; yani Higher Order Mutant (HOM) karşıtı, ör., mutantın mutantı.

$$a + b > c$$

$$a + b > c$$

$$a - b > c$$

$$a * b > c$$

$$a / b > c$$

$$a \% b > c$$

$$a > c$$

$$b > c$$

$$\text{abs}(a) - b > c$$

$$a - \text{abs}(b) > c$$

$$a - b > \text{abs}(c)$$

$$\text{abs}(a - b) > c$$

$$-\text{abs}(a) - b > c$$

$$a - -\text{abs}(b) > c$$

$$a - b > -\text{abs}(c)$$

$$-\text{abs}(a - b) > c$$

$$a - b >= c$$

$$a - b < c$$

$$a - b <= c$$

$$a - b = c$$

$$a - b \neq c$$

$$b - b > c$$

$$a - a > c$$

$$c - b > c$$

$$a - c > c$$

$$a - b > a$$

$$a - b > b$$

$$a - b > c$$

$$0 - b > c$$

$$a - 0 > c$$

$$a - b > 0$$

$$++a - b > c$$

$$a - ++b > c$$

$$a - b > ++c$$

$$--a - b > c$$

$$a - --b > c$$

$$a - b > --c$$

$$++(a - b) > c$$

$$--(a - b) > c$$

$$-a - b > c$$

$$a - -b > c$$

$$a - b > -c$$

$$-(a - b) > c$$

$$0 > c$$

# Performans Problemleri



# Performans Problemleri

**Pek çok** mutasyon operatörü bulunmaktadır.

Proteum – C için 103 mutasyon operatörü

MuJava – 24 sınıf seviyesinde mutasyon operatörü

Her mutasyon operatörü **birçok mutant** oluşturur.

Test edilen programa bağlı olarak

Her mutantın **derlenmesi gerekir**

Her test senaryosunun **her mutanta karşı** yürütülmesi gerekir





# Eşdeğer Mutantlar

Mutasyon = **Sözdizimsel** değişim

Mutantlar, **semantiği**  
**değiştirmeden** kalabilir.

Eşdeğer mutasyonları tespit etmek zordur. (**kararsızlık problemi**)

Mutasyona **erişilmiş** ancak **enfekte** olmamış olabilir

Enfekte **olmuş** ancak **yayılmamış** olabilir.



# Örnek

```
int max(int[] values) {  
    int r, i;  
  
    r = 0;  
    for(i = 1; i<values.length; i++) {  
        if (values[i] > values[r])  
            r = i;  
    }  
  
    return values[r];  
}
```



# Örnek

```
int max(int[] values) {  
    int r, i;  
  
    r = 0;  
    for(i = 0; i<values.length; i++) {  
        if (values[i] > values[r])  
            r = i;  
    }  
  
    return values[r];  
}
```

# Örnek

```
int max(int[] values) {  
    int r, i;  
  
    r = 0;  
    for(i = 0; i<values.length; i++) {  
        if (values[i] > values[r])  
            r = i;  
    }  
  
    return values[r];  
}
```

# Örnek

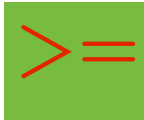
```
int max(int[] values) {  
    int r, i;  
  
    r = 0;  
    for(i = 0; i<values.length; i++) {  
        if (values[i] > values[r])  
            r = i;  
    }  
  
    return values[r];  
}
```

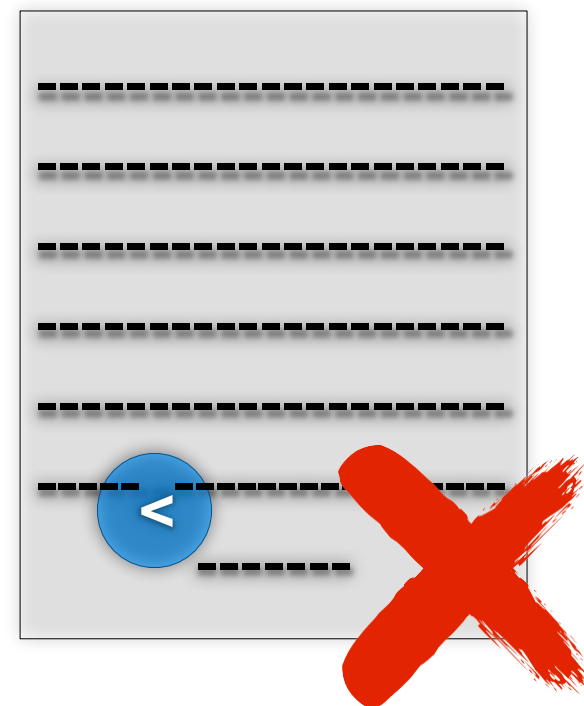
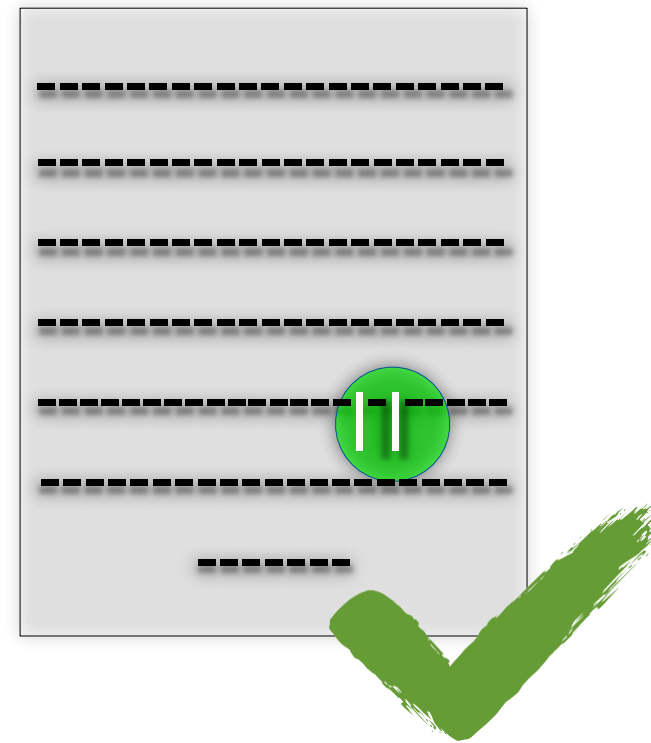
Bu mutant eşdeğerdir çünkü sadece ilk öğenin kendisine ek bir karşılaştırma getirir - **bu işlevsel davranışı değiştiremez.**

# Örnek

```
int max(int[] values) {  
    int r, i;  
  
    r = 0;  
    for(i = 1; i<values.length; i++) {  
        if (values[i] >= values[r])  
            r = i;  
    }  
  
    return values[r];  
}
```

# Örnek

```
int max(int[] values) {  
    int r, i;  
  
    r = 0;  
    for(i = 1; i<values.length; i++) {  
        if (values[i]  values[r])  
            r = i;  
    }  
  
    return values[r];  
}
```

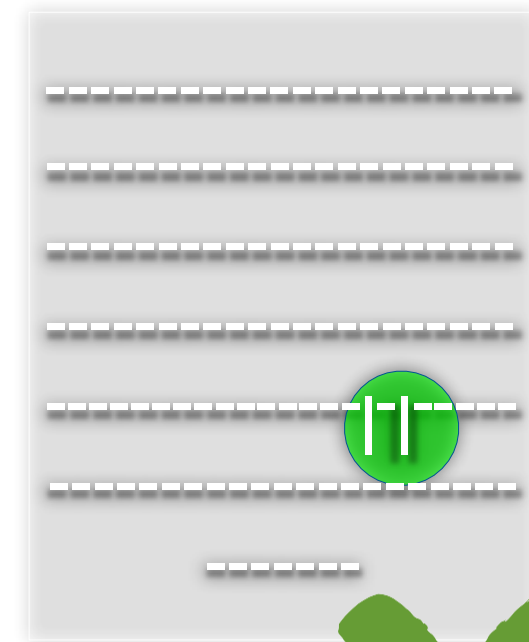


**Mutation Score**

Killed Mutants

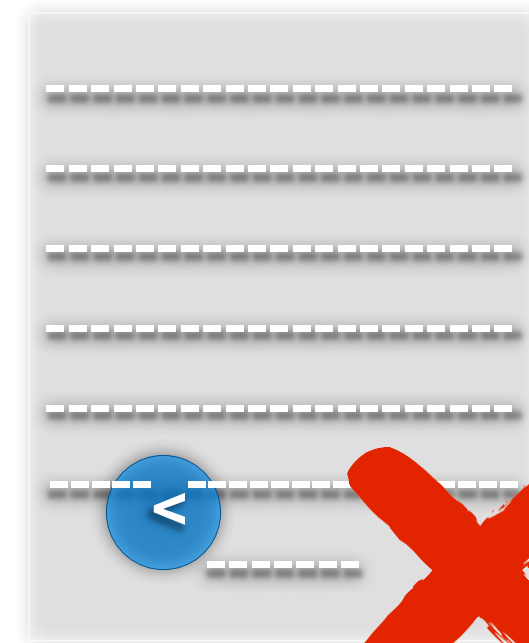
---

Total Mutants



**Mutation Score**

Killed Mutants



---

Total Mutants – Eşdeğer mutantlar

# Özet

**Mutasyon Testi**, bir test paketinin **etkinliğini tahmin etmek için** mutantları kullanır.

Diğer kapsam kriterlerini **simüle etmeye** izin verir

Bazı **iyi bilinen sınırlamaları** vardır

Ölçeklenebilirlik sorunları, örneğin, çok sayıda mutant

**Eşdeğer mutantlar**

Ancak endüstride giderek daha popüler hale geliyor