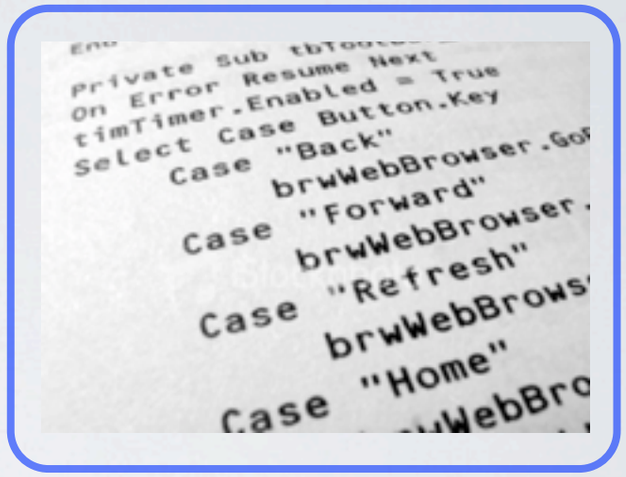
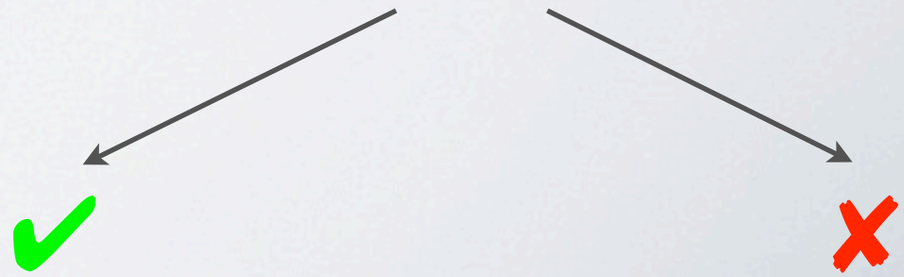
WERYFIKACJA WSPOMAGANA KOMPUTEROWO

Wykład 10

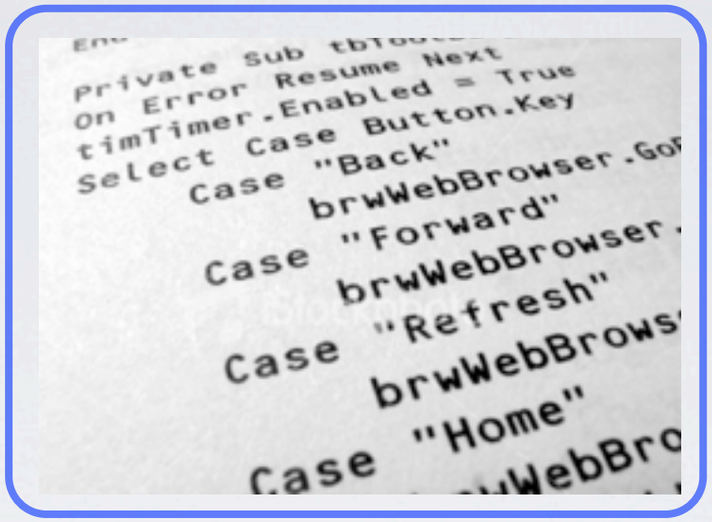




Dowodzenie poprawności programów. Java Modeling Language

I. Dowodzenie poprawności

DOWODZENIE POPRAWNOŚCI



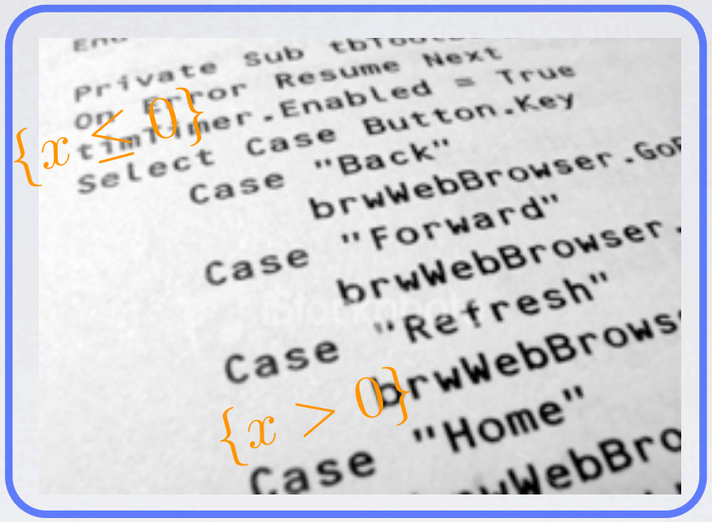
obligacje dowodowe

system wspomagający dowodzenie

dowód

?

DOWODZENIE POPRAWNOŚCI



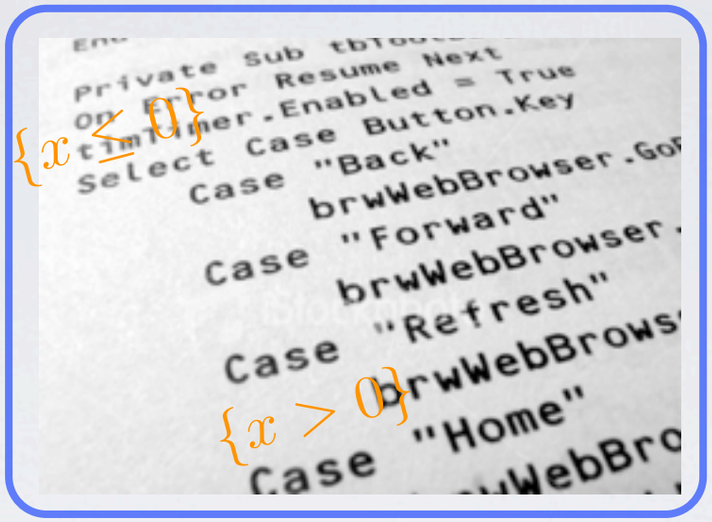
obligacje dowodowe

system wspomagający dowodzenie

dowód

?

DOWODZENIE POPRAWNOŚCI



obligacje dowodowe

automatycznie
lub
interakcyjnie

system wspomagający dowodzenie

dowód

?

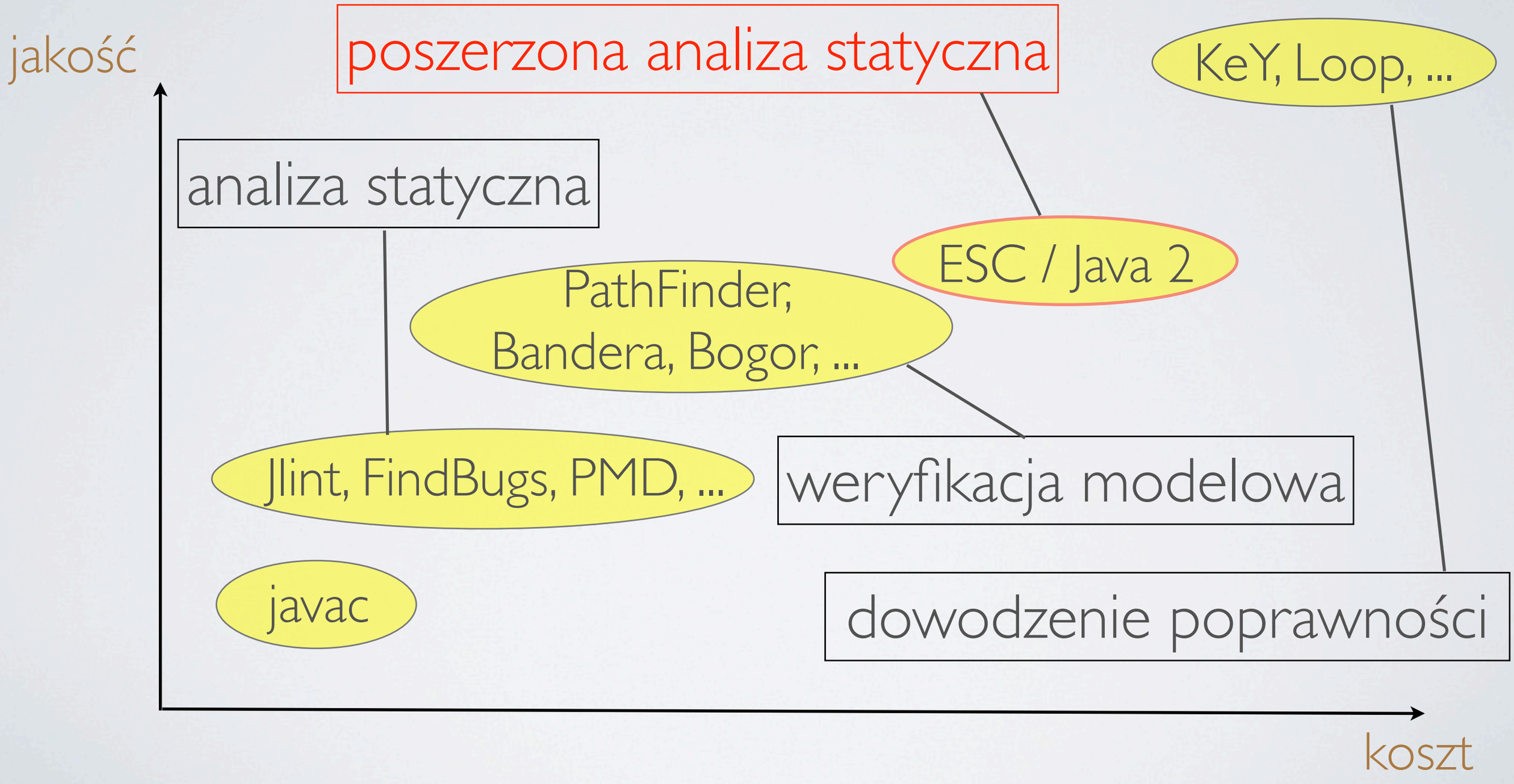
DOWODZENIE POPRAWNOŚCI - CECHY CHARAKTERYSTYCZNE

- analizujemy udekorowany program źródłowy
- na ogół tylko częściowo automatycznie
- na ogół konieczny duży nakład pracy specjalisty
- stosowalne do niewielkich programów
- parametryzacja/generalizacja

ESC / JAVA 2

- Automatyczny system dowodzący Simplify
  - semantyka Javy i JMLa w logice 1go rzędu (ok. 100 aksjomatów)
- Dowodzenie poprawności podczas kompilacji (analiza statyczna)
- Brak pełności i poprawności
  - nie znajduje wszystkich błędów
  - fałszywe alarmy

POSZERZONA ANALIZA STATYCZNA



ESC / JAVA 2

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;
do q1 > q2 → q1, q2 := q2, q1
 [] q2 > q3 → q2, q3 := q3, q2
 [] q3 > q4 → q3, q4 := q4, q3
od    .
```

warunki „bazowe”

(rozwinięcie pętli)

Simplify

obligacje dowodowe (VC)

dowód

kontrprzykład → ostrzeżenie

ESC / JAVA 2

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;
do q1 > q2 → q1, q2 := q2, q1
 [] q2 > q3 → q2, q3 := q3, q2
 [] q3 > q4 → q3, q4 := q4, q3
od    .
```

warunki „bazowe”

(rozwinięcie pętli)

Simplify

obligacje dowodowe (VC)

najsłabsze warunki wstępne

dowód

kontrprzykład → ostrzeżenie

II. JML

ŹRÓDŁA

- G. T. Leavens, Y. Cheon, Design by Contract with JML
- P. Chalin, J. R. Kiniry, G. T. Leavens, E. Poll, Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2
- J. R. Kiniry, G. T. Leavens, E. Poll, A JML Tutorial: Modular Specification and Verification of Functional Behavior for Java, CAV 2007 Tutorial
- K. Rustan, M. Leino, G. Nelson, J.B. Saxe, ESC/Java User's Manual

ZAŁOŻENIA

- formalny język specyfikacji (i dokumentacji)
- inspiracja:
    - logika Hoare'a
        - najsłabszy warunek wstępny Dijkstry
- łatwy w użyciu dla programisty (asercje)
- „lekkie" częściowe specyfikacje
- DBC: design by contract

KONTRAKT

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

KONTRAKT

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

- warunki wstępne (ang. pre-conditions)

KONTRAKT

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

- warunki wstępne (ang. pre-conditions)
- warunki końcowe (ang. post-conditions)

kontrakt

KONTRAKT

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

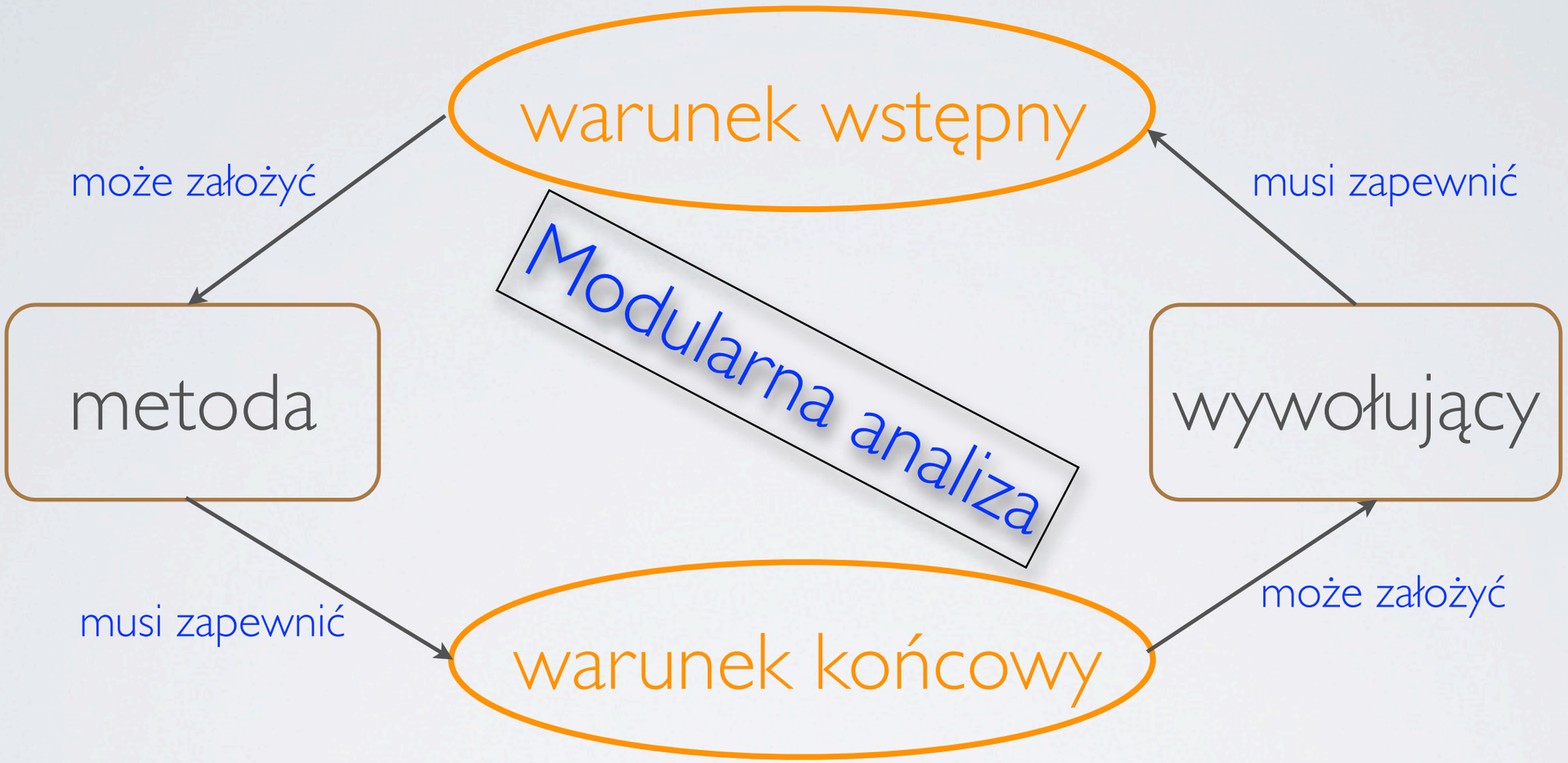
- warunki wstępne (ang. pre-conditions)
- warunki końcowe (ang. post-conditions)
- semantyka: poprawność częściowa

kontrakt

KONTRAKT

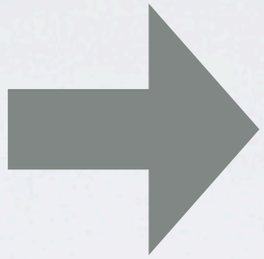

warunek wstępny
może założyć
musi zapewnić
metoda
wywołujący
musi zapewnić
może założyć
warunek końcowy

KONTRAKT


warunek wstępny
może założyć
musi zapewnić
metoda
wywołujący
Modularna analiza
musi zapewnić
może założyć
warunek końcowy

KONTRAKT

```
//@ requires P;
//@ ensures Q;
public void m() {
   S
}
```



```
//@ assert P;
o.m();
//@ assume Q;
```

```
public void m() {
   //@ assume P;
   S
   //@ assert Q;
}
```

MODULARNOŚĆ

```
import org.jmlspecs.models.JMLDouble;

public class SqrtExample {

    public final static double eps = 0.0001;

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, eps);
    public static double sqrt(double x) {
        return Math.sqrt(x);
    }
}
```

```
//@ assert 9.0 >= 0.0;
double res = SqrtExample.sqrt(9.0);

/*@ assert JMLDouble
  @         .approximatelyEqualTo
  @          (9.0, res * res,
  @           SqrtExample.eps);
  @*/
```

LOGIKA

```
(\forall Student s;
  juniors.contains(s);
    s.getAdvisor() != null)
```

- && ==> ! ==

- kwantyfikacja:

```
/*@ also
  @ requires kgs >= 0;
  @ requires weight + kgs >= 0;
  @ ensures weight == \old(weight + kgs);
  @*/
public void addKgs(int kgs);
```

  - \forall \exists \sum \product \min \max \num_of

- \result \old( )

- język naturalny

```
/*@ also
  @ ensures \result != null
  @   && (* \result is a displayable
  @       form of this person *);
  @*/
public String toString() {
  return "Person(\"" + name + "\","
    + weight + ")";
}
```

WYKONYWALNOŚĆ

```
/*@ requires a != null
  @        && (\forall int i;
  @                0 < i && i < a.length;
  @                a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
  // ...
}
```

```
(\forall Student s;
  juniors.contains(s);
    s.getAdvisor() != null)
```

WYKONYWALNOŚĆ

```
/*@ requires a != null
  @       && (\forall int i;
  @                0 < i && i < a.length;
  @                a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
  // ...
}
```

```
(\forall Student s;
  juniors.contains(s);
    s.getAdvisor() != null)
```

- możliwa analiza dynamiczna (w czasie wykonania)

WYKONYWALNOŚĆ

```
/*@ requires a != null
  @       && (\forall int i;
  @                0 < i && i < a.length;
  @                a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
  // ...
}
```

```
(\forall Student s;
  juniors.contains(s);
    s.getAdvisor() != null)
```

- możliwa analiza dynamiczna (w czasie wykonania)
- jmlc

WARUNEK KOŃCOWY

- normalny: ensures
- wyjątkowy: signals
  - signals_only

```
class SettableClock extends TickTockClock {

    // ...

    /*@ public normal_behavior
      @   requires   0 <= hour && hour <= 23 &&
      @              0 <= minute && minute <= 59;
      @   assignable _time_state;
      @   ensures    getHour() == hour &&
      @              getMinute() == minute && getSecond() == 0;
      @ also
      @  public exceptional_behavior
      @   requires     !(0 <= hour && hour <= 23 &&
      @                  0 <= minute && minute <= 59);
      @   assignable   \nothing;
      @   signals      (IllegalArgumentException e) true;
      @   signals_only IllegalArgumentException;
      @*/
    public void setTime(int hour, int minute) {
      if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
        throw new IllegalArgumentException();
      }
      this.hour = hour;
      this.minute = minute;
      this.second = 0;
    }
}
```

WYJĄTKI

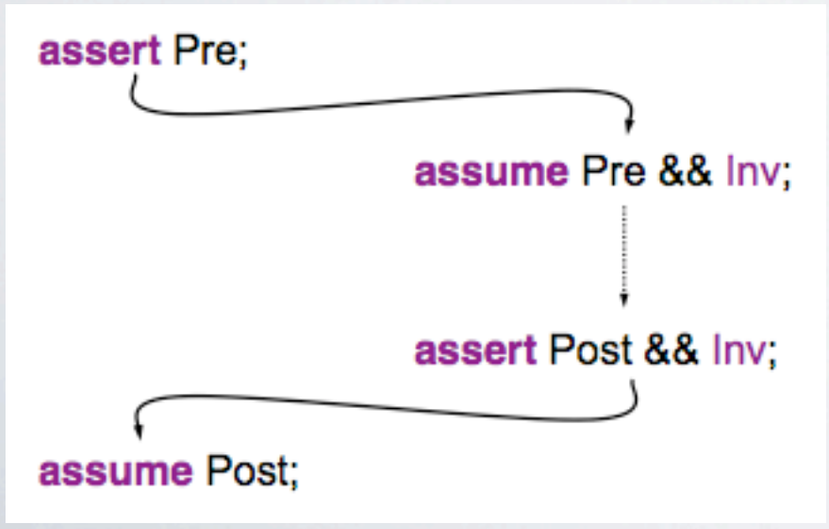
signals(Exception) false

ensures false

```
class SettableClock extends TickTockClock {

    // ...

    /*@ public normal_behavior
      @   requires   0 <= hour && hour <= 23 &&
      @              0 <= minute && minute <= 59;
      @   assignable _time_state;
      @   ensures    getHour() == hour &&
      @              getMinute() == minute && getSecond() == 0;
      @ also
      @  public exceptional_behavior
      @   requires     !(0 <= hour && hour <= 23 &&
      @                  0 <= minute && minute <= 59);
      @   assignable   \nothing;
      @   signals      (IllegalArgumentException e) true;
      @   signals_only IllegalArgumentException;
      @*/
    public void setTime(int hour, int minute) {
      if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
        throw new IllegalArgumentException();
      }
      this.hour = hour;
      this.minute = minute;
      this.second = 0;
    }
}
```

NIEZMIENNIKI (OBIEKTOWE)

- implicite w
  - warunkach wstępnych i końcowych (również wyjątkowych) wszystkich metod
  - warunkach końcowych konstruktorów
- chwilowo mogą nie zachodzić
- ułatwiają zrozumienie kodu



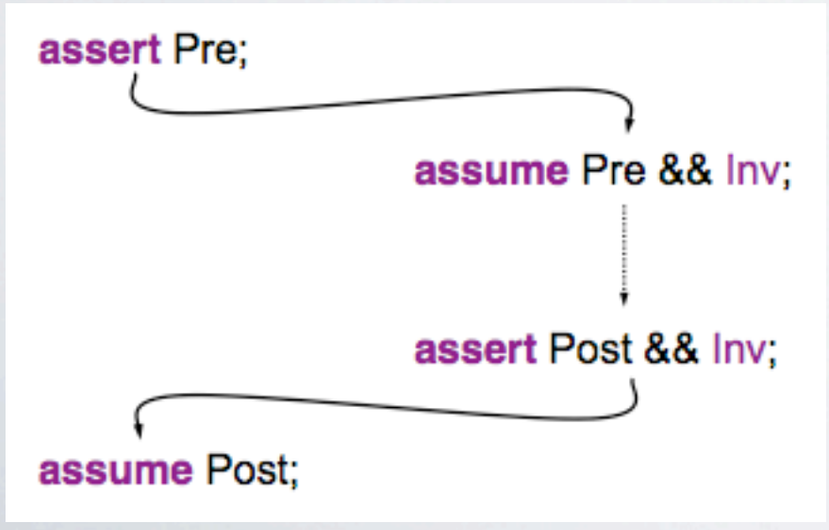

```
public class Person {
  private /*@ spec_public non_null @*/
    String name;
  private /*@ spec_public @*/
    int weight;

  /*@ public invariant !name.equals("")
    @          && weight >= 0; @*/
```

NIEZMIENNIKI (OBIEKTOWE)

- implicite w
  - warunkach wstępnych i końcowych (również wyjątkowych) wszystkich metod
  - warunkach końcowych konstruktorów

skrót ?

- chwilowo mogą nie zachodzić
- ułatwiają zrozumienie kodu




```
public class Person {
  private /*@ spec_public non_null @*/
    String name;
  private /*@ spec_public @*/
    int weight;

  /*@ public invariant !name.equals("")
    @          && weight >= 0; @*/
```

NIEZMIENNIKI

- niezmienniki to dużo więcej niż użyteczny skrót
  - niezmienniki są dziedziczone
  - mogą zależeć od innych obiektów!
  - muszą być prawdziwe we wszytkich stanach obserwowalnych

```
public void tick() {
  second++;
  // object invariant might no longer hold
  canvas.paint();
  /* ... */
}
```

NIEZMIENNIKI

- niezmienniki to dużo więcej niż użyteczny skrót
  - niezmienniki są dziedziczone
  - mogą zależeć od innych obiektów!
  - muszą być prawdziwe we wszytkich stanach obserwowalnych

```
public void tick() {
  second++;
  // object invariant might no longer hold
  canvas.paint();
  /* ... */
}
```

wejście do/wyjście z dowolnej metody

NON_NULL

- skrótowy zapis
  - niezmiennika

```
private /*@ non_null @*/ String fullName;
```

  - warunku wstępnego

```
public /*@ pure @*/ AlarmClock(/*@ non_null @*/ AlarmInterface alarm) {
  this.alarm = alarm;
}
```

  - końcowego

```
//@ also
//@ ensures \result != null;
public String toString();
```

ASSIGNABLE, PURE

- assignable specyfikuje dozwolone efekty uboczne
- pure = assignable \nothing

```
/*@ public normal_behavior
  @   requires   0 <= hour && hour <= 23 &&
  @              0 <= minute && minute <= 59;
  @   assignable _time_state;
  @   ensures    getHour() == hour &&
  @              getMinute() == minute && getSecond() == 0;
  @ also
  @  public exceptional_behavior
  @   requires     !(0 <= hour && hour <= 23 &&
  @                  0 <= minute && minute <= 59);
  @   assignable   \nothing;
  @   signals      (IllegalArgumentException e) true;
  @   signals_only IllegalArgumentException;
  @*/
```

```
//@ ensures _time == 12*60*60;
public /*@ pure @*/ Clock() { hour = 12; minute = 0; second = 0; }

//@ ensures 0 <= \result && \result <= 23;
public /*@ pure @*/ int getHour() { return hour; }
```

„LEKKIE” I „CIĘŻKIE” SPECYFIKACJE

analiza statyczna

analiza dynamiczna

```
class SettableClock extends TickTockClock {

    // ...

    /*@ public normal_behavior
      @   requires   0 <= hour && hour <= 23 &&
      @              0 <= minute && minute <= 59;
      @   assignable _time_state;
      @   ensures    getHour() == hour &&
      @              getMinute() == minute && getSecond() == 0;
      @ also
      @  public exceptional_behavior
      @   requires     !(0 <= hour && hour <= 23 &&
      @                  0 <= minute && minute <= 59);
      @   assignable   \nothing;
      @   signals      (IllegalArgumentException e) true;
      @   signals_only IllegalArgumentException;
      @*/
    public void setTime(int hour, int minute) {
      if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
        throw new IllegalArgumentException();
      }
      this.hour = hour;
      this.minute = minute;
      this.second = 0;
    }
}
```

DZIEDZICZENIE SPECYFIKACJI

- podklasa dziedziczy specyfikację
- niezmiennik podklasy jest silniejszy
- warunek wstępny podklasy jest słabszy
- warunek końcowy podklasy jest silniejszy

```
//@ assignable _time;
//@ ensures _time == \old(_time + 1) % 24*60*60;
public void tick() {
  second++;
  if (second == 60) { second = 0; minute++; }
  if (minute == 60) { minute = 0; hour++; }
  if (hour == 24)   { hour = 0; }
}
```

```
// spec inherited from superclass Clock
public void tick() {
  super.tick();
  if (getHour() == alarmHour & getMinute() == al
    alarm.on();
    //@ set _alarmRinging = true;
  }
  if ((getHour() == alarmHour & getMinute() ==
      (getHour() == alarmHour+1 & alarmMinute ==
    alarm.off();
    //@ set _alarmRinging = false;
  }
}
```

WIDOCZNOŚĆ - ABSTRAKCJA

```
//@ private invariant 0 <= alarmSecondsRemaining &&
//@                    alarmSecondsRemaining <= 60;

/*@ private invariant _alarmRinging
  @                    <==> alarmSecondsRemaining > 0; @*/
private int alarmSecondsRemaining = 0; //@ in _time;
```

```
private /*@ spec_public non_null @*/
  String name;
```

ZMIENNE MODELOWE

- abstrakcyjnie reprezentują zmienne konkretne
- niemodyfikowalne
- zmieniają wartość wraz ze zmiennymi konkretnymi

```
public class Clock {
  //@ public model long _time;
  //@ private represents _time = second + minute*60 + hour*60*60;

  //@ public invariant _time == getSecond() + getMinute()*60 + getHour()*60*60;
  //@ public invariant 0 <= _time && _time < 24*60*60;

  //@ private invariant 0 <= hour && hour <= 23;
  private int hour; //@ in _time;
  //@ private invariant 0 <= minute && minute <= 59;
  private int minute; //@ in _time;
  //@ private invariant 0 <= second && second <= 59;
  private int second; //@ in _time;

  //@ ensures _time == 12*60*60;
  public /*@ pure @*/ Clock() { hour = 12; minute = 0; second = 0; }

  //@ ensures 0 <= \result && \result <= 23;
  public /*@ pure @*/ int getHour() { return hour; }

  //@ ensures 0 <= \result && \result <= 59;
  public /*@ pure @*/ int getMinute() { return minute; }

  //@ ensures 0 <= \result && \result <= 59;
  public /*@ pure @*/ int getSecond() { return second; }

  /*@ requires   0 <= hour && hour <= 23;
    @ requires   0 <= minute && minute <= 59;
    @ assignable _time;
    @ ensures    _time == hour*60*60 + minute*60;
    @*/
  public void setTime(int hour, int minute) {
    this.hour = hour; this.minute = minute; this.second = 0;
  }
```

ZMIENNE-DUCHY

- istnieją tylko w specyfikacji
- modyfikowalne (set)

```
/** The number of seconds remaining to keep ringing the alarm.
  * If zero, the alarm is silent (off). */
//@ private invariant 0 <= alarmSecondsRemaining &&
//@                     alarmSecondsRemaining <= 60;

/*@ private invariant _alarmRinging
  @                   <==> alarmSecondsRemaining > 0; @*/
private int alarmSecondsRemaining = 0; //@ in _time;
...


public boolean tick() {
  super.tick();
  if (alarmSecondsRemaining > 0) {
    alarmSecondsRemaining--;
    if (alarmSecondsRemaining == 0) {
      alarm.off();
      //@ set _alarmRinging = false;
    }
  } else if (getHour() == alarmHour &
             getMinute() == alarmMinute) {
           alarm.on();
           alarmSecondsRemaining = 60 - getSecond();
           //@ set _alarmRinging = true;
  }
}
```

I INNE...

- niezmienniki pętli
- aliasy
- grupy danych
- „omijanie” niezmienników: metody pomocnicze
- poprawność synchronizacji
- ...

III. ESC / Java 2

WŁASNOŚCI

- wyjątki:
    - dereferencja NULLa
    - indeks tablicy poza zakresem, ujemny rozmiar tablicy
    - brak rzutowania na podklasę
    - dzielenie przez 0
- nie są sprawdzane błędy wykonania, np. OutOfMemoryError

WŁASNOŚCI

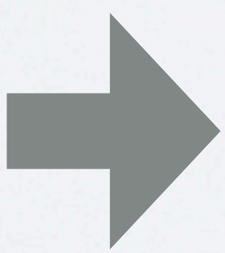
- poprawność metod
    - częściowa popawność względem warunków wstępnych i końcowych
    - naruszenie assignable (pure)
    - niezadeklarowane wyjątki
    - naruszenie non_null

WŁASNOŚCI

- poprawność pętli
  - loop_invariant
  - decreases

Uwaga: pętle są rozwijane zadaną liczbę razy!

```
//@ loop_invariant E;
while (B) {
  S
}
```



```
//@ assert E;
if (!(B)) break;
S
```

ESC/Java User's Manual

WŁASNOŚCI

- poprawność przepływu sterowania:
  - assert
  - reachable

WŁASNOŚCI

- poprawność klas - niezmienniki obiektowe:
  - initially
  - invariant (tylko w stanach obserwowalnych)

WŁASNOŚCI

- wielowątkowość
- monitored
- zakleszczenie (np. spowodowane przez synchronized)
- nie jest sprawdzany brak synchronizacji

INFORMACJA DIAGNOSTYCZNA

- czasem nieczytelna
- punkt wyjścia: formuła uznana przez Simplify za nieprawdziwą (spełnialną)

NIEPOPRAWNOŚĆ - BRAK BŁĘDU

ESC / Java 2 nie sprawdza pewnych własności,
można wyłączyć sprawdzanie pewnych własności

NIEPOPRAWNOŚĆ - PRZYCZYNY

- assume, nowarn
- pętle - skończone rozwinięcia
- niezmienniki obiektów tworzonych dynamicznie ignorowane
- tylko niektóre niezmienniki są brane pod uwagę (heurystyka)
- nie jest sprawdzany brak synchronizacji
- przepełnienie arytmetyczne jest ignorowane
- Simplify ma ograniczony czas działania
- Simplify: błąd w heurystycznej procedurze dla arytmetyki

NIEPEŁNOŚĆ - FAŁSZYWE ALARMY

ESC / Java 2 nie jest w stanie stwierdzić,
że dany błąd nie może zajść

NIEPEŁNOŚĆ - PRZYCZYNY

- Simplify jest niepełny (zwraca kontrprzykład „za wcześnie”)
- niepełna semantyka Javy:
  - arytmetyka zmiennopozycyjna, operacje bitowe, napisy
- przepełnienie arytmetyczne jest ignorowane
- przechwytywany wyjątek też jest traktowany jako błąd
- modularność: statyczny typ obiektu, którego metodę wywołujemy, może mieć słabszą specyfikację niż typ dynamiczny

ESC / JAVA 2 - SILNE STRONY

- szybko znajduje niemało błędów
- narzędzie w pełni automatyczne
- można uruchomić nawet bez specyfikacji
- modularność
- informacja diagnostyczna
- integracja ze środowiskami programistycznymi (Eclipse)

ESC / JAVA 2 - SŁABE STRONY

- brak pełności i poprawności
- czasochłonna specyfikacja jest konieczna
- informacja diagnostyczna nieczytelna
- tylko dla Javy 1.4
- niedosyt dokumentacji

NARZĘDZIA DLA JMLA

poszerzona analiza statyczna

weryfikacja modelowa

Esc/Java2

Bogor

*.html

jmldoc

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @          .approximatelyEqualTo
  @           (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

jmlc

*.class

jmlunit

KeY, LOOP, Krakatoa, JACK

testy jednostkowe

dowód poprawności

NARZĘDZIA DLA JMLA

poszerzona analiza statyczna

weryfikacja modelowa

Esc/Java2

Bogor

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @          .approximatelyEqualTo
  @           (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

jmldoc

*.html

jmlc

*.class

jmlunit

testy jednostkowe

Daikon

*.java

KeY, LOOP, Krakatoa, JACK

dowód poprawności