Techniki zabezpieczania kodu - dziedziczenie

Temat X

Specyfikacje behawioralne

1. Typ klasy: konstruktory, metody (z sygnaturami)
2. To nie opisuje zachowania
3. W JML-u typ to: jak w Javie + opis zachowania

Uszczegółowienie

- Uszczegółowienie C specyfikacji A to taki opis, że każda implementacja C jest także implementacją A.
- Tłumacząc na prewarunki i postwarunki:
  Oznaczmy przez R_A, R_C prewarunki, zaś przez E_A, E_C odpowiednie postwarunki, powinno zachodzić:
  - uszczegółowienie powinno mieć tę samą składnię (nazwy metod, liczby argumentów itp.)
  - prewarunki są związane: $R_C \Rightarrow R_A$
  - postwarunki są związane: $(R_C \Rightarrow E_C) \Rightarrow (R_A \Rightarrow E_A)$

Behawioralny podtyp

- Podklasa – uszczegółowienie typu klasy
- Podsumowanie
  - relacja bycia podklasą – gwarantuje poprawność strukturalną (obecność pól, metod itp.)
  - relacja bycia podtypem – gwarantuje brak błędów typowych, gdy podtypy są używane w miejscu typów
  - relacja bycia podtypem behawioralnym – gwarantuje brak problemów z dziwnym zachowaniem, gdy podtypy są używane w miejscu typów

Behawioralny podtyp w JML-u

- Nadpisywana metoda dziedziczy specyfikacje z nadklasy
- Można dodać bardziej szczegółowy opis za pomoca słowa kluczowego `also`

Behawioralny podtyp w JML-u – niezmienniki

- Niezmienniki są dziedziczone do podklas:

```
class Parent {
   ...
   //@ invariant invParent;
   ...
}

class Child extends Parent {
   ...
   //@ invariant invChild;
   ...
}
```

niezmiennik w klasie `Child` to `invChild && invParent`

Behawioralny podtyp w JML-u – metody

- W wypadku specyfikacji metod wygląda to tak:

```
class Parent {
    //@ requires i >= 0;
    //@ ensures \result >= i;
    int m(int i){ ...
    }
}
class Child extends Parent {
    //@ also
    //@ requires i <= 0
    //@ ensures \result <= i;
     int m(int i){ ...
     }
}
```

  Słowo kluczowe `also` wskazuje, że specyfikacje są dziedziczone.

Behawioralny podtyp w JML-u – metody c.d.

- Metoda `m` w !`Child` musi przestrzegać obu specyfikacji, więc pełna specyfikacja dla niej to:

```
class Child extends Parent {
   /*@ requires i >= 0;
     @ ensures \result >= i;
     @ also
     @ requires i <= 0;
     @ ensures \result <= i;
     @*/
     int m(int i){ ... }
}
```

  Jakich wyników możemy się spodziewać?

Behawioralny podtyp w JML-u – metody c.d.

- Jeszcze inny sposób przedstawienia tej specyfikacji:

```
class Child extends Parent {
   /*@ requires i >= 0 || i <= 0;
     @ ensures \old(i)>=0 ==> \result >= i;
     @ ensures \old(i)<=0 ==> \result <= i;
     @*/
     int m(int i){ ... }
}
```

Behawioralny podtyp – różne

- Można „osłabić” behawioralność
- Należy zrelatywizować typ:
  `\typeof(this) == \type(Nadklasa)`
- Dziedziczone też jest: `normal_behavior`,
  `exceptional_behavior`,
  `signals (E e) false` **i** **in**.

Techniki zabezpieczania kodu - kanały informacyjne

Temat XI

Zaznaczanie obiektów

- Można w klasie `Object` dodać specjalne pole:

  ```
  //@ ghost public boolean isConfidential = false;
  ```

  za pomocą którego można zaznaczać, czy dany obiekt jest tajny, czy nie.

Specyfikacja interfejsów

- Można dodać do interfejsu specyfikacje zabraniające złych wywołań:

```
//@ public normal_behavior
//@   requires !s.isConfidential;
//@   modifies outputText, endsInNewLine;
//@   ensures endsInNewLine;
//@   ensures s != null ==>
//@        outputText.equals(\old(outputText) +
//@                          s + eol);
public void println(String s);
```

  To oznacza, że można wywołać `println` tylko, gdy `s` jest tajne

Specyfikacja interfejsów c.d.

- Trzeba też zapewnić przenoszenie tajności przy manipulacjach na danych:

```
  ...
  @  public normal_behavior
  @   ensures \result.isConfidential <==>
  @             this.isConfidential;
  @*/
public /*@ pure @*/ /*@ non_null @*/
            String substring(int beginIndex)
  throws StringIndexOutOfBoundsException;
```

Tworzenie obiektów

- Najlepiej umieścić informację o tajności wewnątrz konstruktora

```
public Computer(int compId,
                /*@ non_null @*/ String name) {
   this.passwords = new HashSet();
   //@ set passwords.isConfidential = true;
   //@ set this.isConfidential = true;
}
```

Niezmienniki

- Dobrze jest utrzymywać niezmiennik o tajności:

```
//@ invariant passwords.isConfidential == true;
```

- Uwaga: można to w metodzie naruszyć, co może być przydatne (np. szyfrowanie oznacza przepływ od tajnych danych do jawnych)

Problemy

- Pole isConfidential jest public
- Duży wysiłek przy anotowaniu standardowej biblioteki
- Problem ze zmiennymi prymitywnych typów