Specifying and Verifying an example:

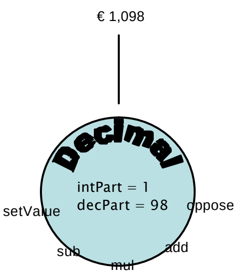
a decimal representation in Java
for smart cards

Cees-Bart Breunesse

Gemplus electronic purse

- JavaCard applet that runs on JavaCard smart cards.
- Debit and credit operations on a global balance.
- Secured communication and authentication.
- Loyalty support.

Representing the balance by a Decimal object



- No floating point numbers in JavaCard.
- `intPart` and `decPart` are shorts.
- `decPart` has a precision of 3 digits.
- Precision is expressed by the final short `PRECISION` which has value 1000.

Issues of the Decimal class

- Negative numbers?
- Object invariant:

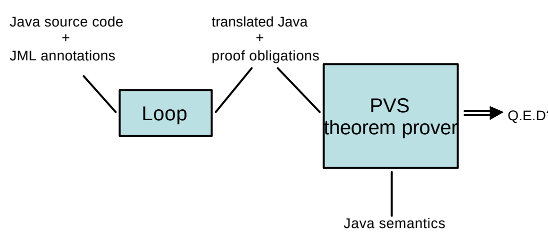
```
    -PRECISION < decPart
 && decPart < PRECISION
```

- Specified and verified all but two methods of the Decimal class (4/26).
- Also specified/verified: DecimalException (2/1).
- Redundant code?
- Gemplus code is *not* changed, but verified as is.

Specification + Verification

```
//@ invariant    -PRECISION < decPart
               && decPart < PRECISION


/*@normal_behavior
  @    requires true;
  @  modifiable intPart, decPart;
  @     ensures    intPart == -\old(intPart)
  @             && decPart == -\old(decPart);
  @*/
Decimal oppose()
{
  intPart = -intPart;
  decPart = -decPart;
  return this;
}
```


Specification + Verification



Verification is done by applying Hoare rules.
Advantage: stepwise refinement of proofs.

A member of Decimal: public add

```
public void add(Decimal d)
{
  ..
  add(d.intPart, d.decPart);
  ..
}
```

- Public method add is defined by a private helper function on the integer and decimal parts of argument d.

A member of Decimal: private add

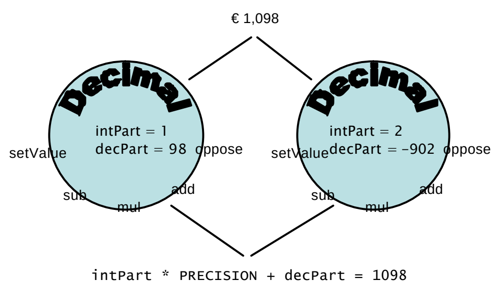
```
private void add(short e, short f) {
        intPart += e;

        decPart += f;

        .. make decPart obey its boundaries ..

}
```

- Private add method is simplified.
- Extra checks make execution of the code faster/more efficient.
- Consequence: less readable code.
- Reason for choosing method add in this talk: its code is complex, its specification is clear.

Private method add

```
private void add(short e, short f) {
        intPart += e;

        decPart += f;

        short retenue = 0;
        short signe = 1;
        if ( decPart < 0 ) {
            signe = -1;
            decPart = -decPart;
        }

        retenue = decPart / PRECISION;
        decPart = decPart % PRECISION;
        retenue *= signe;
        decPart *= signe;
        intPart += retenue;
}
```


How to specify add?



Specification of add

```
/*@ normal_behavior
  @   requires  f < PRECISION && f > -PRECISION;
  @ modifiable intPart, decPart;
  @    ensures (\old(intPart) + e)*PRECISION + \old(decPart) + f
  @                   == intPart*PRECISION + decPart;
  @*/
  private add(short e, short f)
  { .. }
```

Complex and obscure method source code

Short and clear method specification

Verification of add

Assertions	Code
	`void add(short e, short f) {` `intPart += e;`
`decPart == (\old(intPart) + e) * PRECISION + \old(decPart) + f - intPart * PRECISION`	`decPart += f;`
	`.. make decPart obey its boundaries ..` `}`

Verification of add

Assertions

```
decPart
== (\old(intPart) + e)
   * PRECISION
   + \old(decPart) + f
   - intPart * PRECISION

    -PRECISION < decPart
 && decPart < PRECISION
```

Code

```
void add(short e, short f) {
  intPart += e;

  decPart += f;

  .. make decPart obey its
     boundaries ..

}
```


Verification of add

Assertions

```
decPart
== signe *
   ((\old(intPart) + e)
    * PRECISION
    + \old(decPart) + f
    - intPart * PRECISION)
```

Code

```
short retenue = 0;
short signe = 1;

if ( decPart < 0 ) {
  signe = -1;
  decPart = -decPart;
}

retenue = decPart / PRECISION;
decPart = decPart % PRECISION;
retenue *= signe;
decPart *= signe;
intPart += retenue;
}
```


Verification of add

Assertions

```
retenue * PRECISION
+ decPart % PRECISION
== signe *
   ((\old(intPart) + e)
    * PRECISION
    + \old(decPart) + f
    - intPart * PRECISION)
```

Code

```
short retenue = 0;
short signe = 1;

if ( decPart < 0 ) {
  signe = -1;
  decPart = -decPart;
}

retenue = decPart / PRECISION;
decPart = decPart % PRECISION;
retenue *= signe;
decPart *= signe;
intPart += retenue;
}
```


Verification of add

Assertions

```
retenue * PRECISION
+ decPart
== signe *
   ((\old(intPart) + e)
    * PRECISION
    + \old(decPart) + f
    - intPart * PRECISION)
```

Code

```
short retenue = 0;
short signe = 1;

if ( decPart < 0 ) {
  signe = -1;
  decPart = -decPart;
}

retenue = decPart / PRECISION;
decPart = decPart % PRECISION;
retenue *= signe;
decPart *= signe;
intPart += retenue;
}
```


Verification of add

Assertions

```
    -PRECISION < decPart
 && decPart < PRECISION

retenue * PRECISION
+ decPart
== signe *
   ((\old(intPart) + e)
    * PRECISION
    + \old(decPart) + f
    - intPart * PRECISION)
```

Code

```
short retenue = 0;
short signe = 1;

if ( decPart < 0 ) {
  signe = -1;
  decPart = -decPart;
}

retenue = decPart / PRECISION;
decPart = decPart % PRECISION;
retenue *= signe;
decPart *= signe;
intPart += retenue;
}
```


Verification of add

Assertions

```
retenue * PRECISION
+ decPart * signe
== (\old(intPart) + e)
   * PRECISION
   + \old(decPart) + f
   - intPart * PRECISION
```

Code

```
short retenue = 0;
short signe = 1;

if ( decPart < 0 ) {
  signe = -1;
  decPart = -decPart;
}

retenue = decPart / PRECISION;
decPart = decPart % PRECISION;
retenue *= signe;
decPart *= signe;
intPart += retenue;
}
```

Verification of add

Assertions

```
retenue * PRECISION
+ decPart
== (\old(intPart) + e)
    * PRECISION
    + \old(decPart) + f
    - intPart * PRECISION
```

Code

```
short retenue = 0;
short signe = 1;

if ( decPart < 0 ) {
  signe = -1;
  decPart = -decPart;
}

retenue = decPart / PRECISION;
decPart = decPart % PRECISION;
retenue *= signe;
decPart *= signe;
intPart += retenue;
}
```


Verification of add

Assertions

```
decPart
== (\old(intPart) + e)
    * PRECISION
    + \old(decPart) + f
    - intPart * PRECISION
```

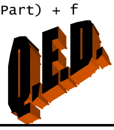
Code

```
short retenue = 0;
short signe = 1;

if ( decPart < 0 ) {
  signe = -1;
  decPart = -decPart;
}

retenue = decPart / PRECISION;
decPart = decPart % PRECISION;
retenue *= signe;
decPart *= signe;
intPart += retenue;
}
```


Verification of add

Assertions

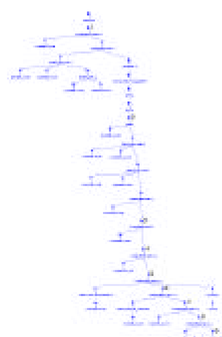
```
decPart - intPart * PRECISION
== (\old(intPart) + e)
    * PRECISION
    + \old(decPart) + f
```



Code

```
short retenue = 0;
short signe = 1;

if ( decPart < 0 ) {
  signe = -1;
  decPart = -decPart;
}

retenue = decPart / PRECISION;
decPart = decPart % PRECISION;
retenue *= signe;
decPart *= signe;
intPart += retenue;
}
```


Verification of add



```
private void add(short e, short f){
    intPart += e;

    if ( intPart > 0 && decPart < 0 ) {
        intPart--;
        decPart = (short) (decPart + PRECISION);
    }
    else if ( intPart < 0 && decPart > 0 ){
        intPart++;
        decPart =(short) (decPart - PRECISION);
    }

    decPart += f;

    if ( intPart > 0 && decPart < 0 ) {
        intPart--;
        decPart = (short) (decPart + PRECISION);
    }
    else if ( intPart < 0 && decPart > 0 ){
        intPart++;
        decPart = (short) (decPart - PRECISION);
    }
    else {

        short retenue = (short) 0;
        short signe = 1;
        if ( decPart < 0 ) {
            signe = (short) -1;
            decPart = (short) -decPart;
        }
        retenue = (short) (decPart / PRECISION);
        decPart = (short) (decPart % PRECISION);
        retenue *= signe;
        decPart *= signe;
        intPart += retenue;
    }
}
```


Verification of an erroneous method

- Consider the round method in Decimal.
- The semantics for round is clear: return the nearest integer.

6,009 ➔ 6,0
6,060 ➔ 6,0
6,501 ➔ 7,0

Verification of an erroneous method

```
public Decimal round(){
        short aux = decPart;
        if ( aux < 0 ) aux = -aux;
        while ( aux > 10 ) aux /= 10;
        if ( aux > 5 ) {
              if ( decPart > 0 ) intPart++;
              else intPart--;
        }
        decPart = 0;
        return this;
    }
```

Unfortunately, Gemplus' implementation is incorrect for inputs where the absolute value of decPart is within 6-10, 60-100, 501-599.

6,009 ➔ 7,0
6,060 ➔ 7,0
6,501 ➔ 6,0

Conclusions

- Specified and verified all but two members of the Decimal class.
- Formal verification is justified in critical applications.
- The LOOP verification technology is ready for such non-trivial examples.
- Software developers like Gemplus should annotate their code with assertions.
- Related work: ESC/Java.

ESC/Java vs. LOOP

ESC/Java:

- Errors caused by null references, out-of-bounds array access, type casts.
- Automatic checking.

LOOP:

- More aimed at functional program verification.
- It needs user interaction.

```
ensures intPart == \old(decPart) >= PRECISION/2 ?
                              (\old(intPart) + 1) :
                               \old(intPart)
       &&  decPart == 0;
```