Specifying and Verifying an example:
a decimal representation in Java for smart cards

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:
  \[-\text{PRECISION} < \text{decPart} \land \text{decPart} < \text{PRECISION}\]
- 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
- Java semantics + JML annotations
- translated Java + proof obligations
- Verification is done by applying Hoare rules.
  Advantage: stepwise refinement of proofs.
A member of Decimal: public add

```java
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

```java
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

```java
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;
}
```

- Complex and obscure method source code
- Short and clear method specification

How to specify add?

```
<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>DECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>intPart = 1</td>
<td>intPart = 2</td>
</tr>
<tr>
<td>decPart = 98</td>
<td>decPart = -902</td>
</tr>
</tbody>
</table>
```

```java
$1,098$
```

Verification of add

```java
void add(short e, short f) {  
    intPart += e;
    decPart += f;
    .. make decPart obey its boundaries ..
}
```

- Complex and obscure method source code
- Short and clear method specification
## Verification of add

### Code
```c
void add(short e, short f) {
    intPart += e;
    decPart += f;
    // make decPart obey its boundaries ..
}
```

### Assertions
- \( \text{decPart} = (\text{old}(\text{intPart}) + e) \times \text{PRECISION} + \text{old}(\text{decPart}) + f - \text{intPart} \times \text{PRECISION} \)
- \( \text{decPart} \geq \text{PRECISION} \) or \( \text{decPart} < \text{PRECISION} \)

## Verification of add

### Code
```c
short retenue = 0;
short signe = 1;
if (decPart < 0) {
    signe = -1;
    decPart = -decPart;
}
retenue = decPart / \text{PRECISION};
retenue += \text{signe} * (\text{old}(\text{intPart}) + e) \times \text{PRECISION} + \text{old}(\text{decPart}) + f - \text{intPart} \times \text{PRECISION};
inPart += retenue;
```

### Assertions
- \( \text{retenue} \times \text{PRECISION} + \text{decPart} \times \text{signe} = (\text{old}(\text{intPart}) + e) \times \text{PRECISION} + \text{old}(\text{decPart}) + f - \text{intPart} \times \text{PRECISION} \)
Verification of add

Assertions

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

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

Verification of add

Assertions

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

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

Verification of an erroneous method

Consider the round method in Decimal.
The semantics for round is clear: return the nearest integer.
6.009 \rightarrow 6.0
6.060 \rightarrow 6.0
6.501 \rightarrow 7.0

Verification of an erroneous method

Unfortunately, Gemplus' implementation is incorrect for inputs where the absolute value of decPart is within 6-10, 60-100, 501-599.
6.009 \rightarrow 7.0
6.060 \rightarrow 7.0
6.501 \rightarrow 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.

\[
\text{ensures } \text{intPart} = \text{old(intPart)} \geq \text{PRECISION/2} \Rightarrow \\
\text{old(intPart) + 1) :} \\
\text{old(intPart) } \\
\text{decPart == 0;}
\]