Eclipse UOMo
Click to edit Master subtitle style
Embedded Measurement

evergn
Success is a planned event
Our Goal

Avoiding Interface and Arithmetic Errors
Most of today’s technologies including the current Java Language Releases lack support for common non-trivial Arithmetic problems like Unit Conversions.
Summary

- Present Situation
  - Historic IT Errors and Bugs
  - Cause of Conversion Errors
- Proposed Changes
  - Unit and Measure Support
  - Type Safety
- Case Studies
- Demo
- Q&A
What do these disasters have in common?

- **Patriot Missile**
  The cause was an inaccurate calculation of the time since boot due to a computer arithmetic error.

- **Ariane 5 Explosion**
  The floating point number which a value was converted from had a value greater than what would be represented by a 16 bit signed integer.
Mars Orbiter

Preliminary findings indicate that one team used English units (e.g. inches, feet and pounds) while the other used metric units for a key spacecraft operation.

- NASA lost a $125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency's team used the more conventional metric system for a key spacecraft operation.
- This also underlines the added risk when 3rd party contractors are involved or projects are developed Offshore.
23rd March 1983. Ronald Reagan announces SDI (or “Star Wars”): ground-based and space-based systems to protect the US from attack by strategic nuclear ballistic missiles.
1985

SDI Experiment: The Plan

Mirror on underside of shuttle

Big mountain in Hawaii
1985

SDI Experiment: What really happened
1985: What happened?

Attention All Units, Especially Miles and Feet!

Much to the surprise of Mission Control, the space shuttle Discovery flew upside-down over Maui on 19 June 1985 during an attempted test of a Star-Wars-type laser-beam missile defense experiment. The astronauts reported seeing the bright-blue low-power laser beam emanating from the top of Mona Kea, but the experiment failed because the shuttle’s reflecting mirror was oriented upward! A statement issued by NASA said that the shuttle was to be repositioned so that the mirror was pointing (downward) at a spot 10,023 feet above sea level on Mona Kea; that number was supplied to the crew in units of feet, and was correctly fed into the onboard guidance system -- which unfortunately was expecting units in nautical miles, not feet. Thus the mirror wound up being pointed (upward) to a spot 10,023 nautical miles above sea level. The San Francisco Chronicle article noted that “the laser experiment was designed to see if a low-energy laser could be used to track a high-speed target about 200 miles above the earth. By its failure yesterday, NASA unwittingly proved what the Air Force already knew -- that the laser would work only on a ‘cooperative target’ -- and is not likely to be useful as a tracking device for enemy missiles.” [This statement appeared in the S.F. Chronicle on 20 June, excerpted from the L.A. Times; the NY Times article on that date provided some controversy on the interpretation of the significance of the problem.] The experiment was then repeated successfully on 21 June (using nautical miles). The important point is not whether this experiment proves or disproves the viability of Star Wars, but rather that here is just one more example of an unanticipated problem in a human-computer interface that had not been detected prior to its first attempted actual use.
Metric mishap caused loss of NASA orbiter

September 30, 1999

Web posted at: 4:21 p.m. EDT (20:21 GMT)

In this story:
Metric system used by NASA for many years
Error points to nation's conversion lag

Related stories, sites

(CNN) -- NASA lost a $125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency's team used the more conventional metric system for a key spacecraft operation, according to a review finding released Thursday.

The units mismatch prevented navigation information from transferring between the Mars Climate Orbiter spacecraft team at Lockheed Martin in Denver and the flight team at NASA's Jet Propulsion Laboratory in Pasadena, California.

Lockheed Martin helped build, develop and operate the spacecraft for NASA. Its engineers provided navigation commands for Climate Orbiter's thrusters in English units although NASA has been using the metric system predominately since at least 1990.

No one is pointing fingers at Lockheed Martin, said Tom Gavin, the JPL administrator to whom all project managers report.
All previous example illustrate three categories of errors difficult to find through Unit Testing:

- Interface Errors (e.g. millisecond/second, radian/degree, meters/feet).
- Arithmetic Errors (e.g. overflow).
- Conversion Errors.
Causes of Conversion Errors

- **Ambiguity on the unit**
  - Gallon Dry / Gallon Liquid
  - Gallon US / Gallon UK
  - Day Sidereal / Day Calendar
  - ...

- **Wrong conversion factors:**
  ```java
  static final double PIXEL_TO_INCH = 1 / 72;
  double pixels = inches * PIXEL_TO_INCH
  ```
Java does not have strongly typed primitive types (like e.g. Ada language).

For performance reasons most developer prefer primitive types over objects in their interface.

Primitives type arguments often lead to name clashes (methods with the same signature)
Unified Code for Units of Measure

- The Unified Code for Units of Measure is a code system intended to include all units of measures being contemporarily used in international science, engineering, and business. The purpose is to facilitate unambiguous electronic communication of quantities together with their units. The focus is on electronic communication, as opposed to communication between humans. A typical application of The Unified Code for Units of Measure are electronic data interchange (EDI) protocols, but there is nothing that prevents it from being used in other types of machine.
The Unified Code for Units of Measure is inspired by and heavily based on:

- ISO 2955-1983
- ANSI X3.50-1986
- HL7's extensions called ISO+
Base Classes and Packages

- **Namespace:** `javax.measure.*`
- **Only one interface and one abstract class**
  - `Measurable<Q extends Quantity>` (interface)
  - `Measure<V, Q extends Quantity>` (abstract class)
- **Three sub-packages**
  - `Unit` (holds the SI and NonSI units)
  - `Quantity` (holds dimensions mass, length)
  - `Converter` (holds unit converters)
Units and System of Units

- Units are immutable.
- Units are mutually independent, which means they cannot be derived from any other base unit (m, s, kg).
- Product units are formed by the product of rational powers of existing units (m/s², N·m).
- Alternate units are used in expressions to distinguish between quantities of a different nature but of the same dimensions (e.g., V, Pa, Rad).
- Transformed units are derived from other units through a converter (ft, °C, dB).

SystemOfUnits groups units together for historical or cultural reasons.

- SI
- NonSI

DerivedUnit<Q extends Quantity> 

BaseUnit<Q extends Quantity> 

CompoundUnit<Q extends Quantity> 

ProductUnit<Q extends Quantity> 

TransformedUnit<Q extends Quantity> 

AlternateUnit<Q extends Quantity> 

Unit<Q extends Quantity>: ONE

- getStandardUnits()
- getDimension()
- getConverterTo(Unit)
## Results with

<table>
<thead>
<tr>
<th>Same Dimension</th>
<th>Different Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary Operations</strong></td>
<td><strong>Binary Operations</strong></td>
</tr>
<tr>
<td>plus (double) or (long)</td>
<td>root(int)</td>
</tr>
<tr>
<td>times(double) or (long)</td>
<td>power(int)</td>
</tr>
<tr>
<td>divide(double) or (long)</td>
<td>times(Unit)</td>
</tr>
<tr>
<td>compound(Unit)divide(Unit)</td>
<td></td>
</tr>
</tbody>
</table>

**Unary Operations**

inverse()
Units of Measure API

- **Namespace**: org.unitsofmeasure.*
- **Only interfaces and one abstract class**
  - public interface Quantity<Q extends Quantity<Q>>
  - public interface Unit<Q extends Quantity<Q>>
- **Two sub-packages**
  - quantity (holds dimensions mass, length)
  - util (misc items and helpers, **optional**)
Mobile Sensor API

- **Namespace:** `javax.microediton.sensor`
- **Focusing on Sensors**, but it got a **minimalistic Unit API “in the closet”**
  - **Unit**: Essentially an SI singleton holding relevant unit constants, too.
  - **ChannelInfo**: Holding name, accuracy, data type, measurement ranges, scale and unit
  - **MeasurementRange**: Range of possible values from minimum to maximum
Sensor States

- **OPENED**
  - Connector open
  - close
  - setDataListener
  - removeDataListener

- **LISTENING**
  - close

- **CLOSED**
Sensor Groups

- Context types categorize sensors into three groups:
  1. ambient, sensors measuring some ambient property of the environment
  2. device, sensors measuring properties related to the device
  3. user, sensors measuring some function of the user

By using context types, it is possible to find, for example, user-related sensors for fitness, or ambient sensors for smart home controlling. Although the granularity of this level does not qualify the sensors very precisely, still in some cases it provides the way to make the distinction, for example, cf. air thermometer (ambient) / clinical thermometer (user).
The quantity provides a more precise qualifier. The unit and the quantity has a close relation. Some quantities are listed in tables of Unit class. When the quantity and context type is known, it is often easy to guess the full purpose of the sensor. Some examples are given here:

Quantity: electric_current + context type: ambient = sensor measuring electric current, amperemeter

Quantity: catalytic_activity + context_type: ambient = sensor measuring catalytic activity
Measurement Package

- **Namespace**: org.osgi.util.measurement
- **SI only Unit API “in the closet”**
  - **Unit**
    - Essentially an SI singleton holding relevant unit constants, too.
  - **Measurement**
    - Represents a value with an error, a unit and a time-stamp.
  - **State**
    - Groups a state name, value and timestamp.
One Small Step…

“That’s one small step for (a) man; one giant leap for mankind.”

- Neil Armstrong - July 20, 1969
One Unit Framework to Measure them All

- **Namespace:** org.eclipse.uomo.*
- **Two main areas**
  - Static Type Safe Units of Measure Support
    - Based on Units of Measure API
    - On top of ICU4J, the Globalization standard at Eclipse and others (Android, GWT, Google Financial, etc.)
  - UCUM Reference Implementation
    - Successor to Eclipse OHF UCUM Bundle
Monetary systems are not currently in scope for UOMo, but this illustrates how easily the framework can be extended to non-physical or scientific quantities. Such extension can be valuable by leveraging the framework’s capabilities (formatting, conversion,...) and applying its usefulness beyond what e.g. java.util.Currency now has to offer.
Monetary Types

The Quantity type for parameterization is Money.

Currency is a particular type of unit, with a dimension of [$].

CurrencyConverter is unusual in that its values change with time, and the conversion from A to B is not the inverse of the conversion from B to A (because a currency's buying price can differ from its selling price).

The monetary system is based on ISO 4217, which is the international standard describing three-letter codes (also known as the currency code) to define the names of currencies.
### Currency Conversion

**Currency rates from 03/08/2007 12:00pm EST**

<table>
<thead>
<tr>
<th>Currency Name</th>
<th>Currency Code</th>
<th>Exchange Rate to US $</th>
<th>Exchange Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australian Dollar</td>
<td>AUD</td>
<td>1.287830006</td>
<td>128.78</td>
</tr>
<tr>
<td>Baht</td>
<td>THB</td>
<td>32.7</td>
<td>3270.00</td>
</tr>
<tr>
<td>Bolivar</td>
<td>VEB</td>
<td>2144.6</td>
<td>214460.00</td>
</tr>
<tr>
<td>Brazilian Real</td>
<td>BRL</td>
<td>2.114</td>
<td>211.40</td>
</tr>
<tr>
<td>Canadian Dollar</td>
<td>CAD</td>
<td>1.1785</td>
<td>117.85</td>
</tr>
<tr>
<td>Danish Krone</td>
<td>DKK</td>
<td>5.6645</td>
<td>566.45</td>
</tr>
<tr>
<td>Euro</td>
<td>EUR</td>
<td>0.76057195</td>
<td>76.06</td>
</tr>
<tr>
<td>Hong Kong Dollar</td>
<td>HKD</td>
<td>7.8177</td>
<td>781.77</td>
</tr>
<tr>
<td>Indian Rupee</td>
<td>INR</td>
<td>44.18</td>
<td>4418.00</td>
</tr>
<tr>
<td>Malaysian Ringgit</td>
<td>MYR</td>
<td>3.509</td>
<td>350.90</td>
</tr>
<tr>
<td>Mexican Nuevo Peso</td>
<td>MXN</td>
<td>11.155</td>
<td>1115.50</td>
</tr>
<tr>
<td>New Taiwan Dollar</td>
<td>TWD</td>
<td>32.92</td>
<td>3292.00</td>
</tr>
<tr>
<td>New Zealand Dollar</td>
<td>NZD</td>
<td>1.462202076</td>
<td>146.22</td>
</tr>
<tr>
<td>Norwegian Krone</td>
<td>NOK</td>
<td>6.1983</td>
<td>619.83</td>
</tr>
<tr>
<td>Pound Sterling</td>
<td>GBP</td>
<td>0.517732332</td>
<td>51.77</td>
</tr>
<tr>
<td>Rand</td>
<td>ZAR</td>
<td>7.4247</td>
<td>742.47</td>
</tr>
<tr>
<td>Singapore Dollar</td>
<td>SGD</td>
<td>1.526</td>
<td>152.60</td>
</tr>
<tr>
<td>Sri Lanka Rupee</td>
<td>LKR</td>
<td>108.85</td>
<td>10885.00</td>
</tr>
<tr>
<td>Swedish Krona</td>
<td>SEK</td>
<td>7.0432</td>
<td>704.32</td>
</tr>
</tbody>
</table>
Trading Example

What happens, if we use built in `java.util.Currency` and Standard JSP formats

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Company</th>
<th>Price</th>
<th>Change</th>
<th>% Change</th>
<th>Shares</th>
<th>Open</th>
<th>Volume</th>
<th>Current Value *</th>
<th>Gain/Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>&quot;IBM&quot;</td>
<td>€115.43</td>
<td>€0.37</td>
<td>-32%</td>
<td>50</td>
<td>€115.80</td>
<td>2,655,471</td>
<td>3699.68 €</td>
<td>€-15.98</td>
</tr>
<tr>
<td>JAVA</td>
<td>&quot;JAVA&quot;</td>
<td>€16.56</td>
<td>€0.44</td>
<td>273%</td>
<td>200</td>
<td>€16.12</td>
<td>5,750,460</td>
<td>2123.08 €</td>
<td>€545.90</td>
</tr>
<tr>
<td>DELL</td>
<td>&quot;DELL&quot;</td>
<td>€19.52</td>
<td>€0.08</td>
<td>41%</td>
<td>200</td>
<td>€19.44</td>
<td>14,293,015</td>
<td>2502.56 €</td>
<td>€82.30</td>
</tr>
<tr>
<td>GOOG</td>
<td>&quot;GOOG&quot;</td>
<td>€426.88</td>
<td>€1.62</td>
<td>38%</td>
<td>100</td>
<td>€425.26</td>
<td>5,523,309</td>
<td>27363.97 €</td>
<td>€38.05</td>
</tr>
<tr>
<td>MSFT</td>
<td>&quot;MSFT&quot;</td>
<td>€28.58</td>
<td>€0.20</td>
<td>71%</td>
<td>100</td>
<td>€28.38</td>
<td>47,317,464</td>
<td>1832.15 €</td>
<td>€71.00</td>
</tr>
</tbody>
</table>

* in local Currency

Make a trade
Log out
We’ll extend MoneyDemo to show fuel costs in Indian Rupees. First by adding a new currency to MonetarySystem.

// Use currency not defined as constant (Indian Rupee).
public static final DerivedUnit<Money> INR = monetary(
    new Currency("INR"));

Then add this line to MoneyDemo. (also change static import to MonetarySystem.*; )

UnitFormat.getInstance().label(INR, "Rp");
Next set the Exchange Rate for Rp.

```java
((Currency) INR).setExchangeRate(0.022);  // 1.0Rp = ~0.022 $
```

Note, the explicit cast is required here, because `getUnits()` in `SystemOfUnits` currently requires a neutral `<?>` generic collection type.
Then we add the following line to the “Display cost.” section of **MoneyDemo**

```
System.out.println("Trip cost = " + tripCost + " (" +
tripCost.to(INR) + ")");
```

Resulting in the additional output:

Trip cost = 87.50 $ (3977.26 Rp)
Eclipse – Project UOMo
http://www.eclipse.org/proposals/uomo/

UCUM
http://www.unitsofmeasure.org

Units of Measure API
http://www.javaforge.com/project/uom
Questions

werner@emergn.com

or

info@catmedia.us

Twitter: @wernerkeil