Computing with numeric objects

- Background: Data and objects
- Design patterns for numeric objects with C++ implementations
- What's wrong with Java?

For APCU
April 24, 2010
Conrad Weisert

Numeric data items in non-O.O. languages

- Representation-based declaration
  - The programmer declares how an item is to be represented internally, i.e. what it looks like, not what it is:

```
double float startTemperature;  (C)
5 UNIT-COST PICTURE S9(5)V.99 (COBOL)
```

What's wrong with that?

Numeric data items in non-O.O. languages

- There's no way to specify a data item's:
  - meaning
  - type
  - behavior
  - unit of measure
  - range
  - other attributes
- Sensible processing of a data item relies upon:
  - its mnemonic name
  - Standards for the type of data item
  - programmers' whims

Program bugs stemming from data misunderstandings

- Confusion about meaning
  - e.g. `proRatedPremium`
- Confusion about attributes / representation
  - e.g. Mars lander crash

How can objects help?
Data representation standards

- Should each program designer decide how to represent each data item?
- Does a company need more than one way of representing:
  - dates?
  - amounts of money?
  - time?
  - weight?
  - distance?

The old enlightened way:
- Organizations ("Data Administration") established standards for representing types of data.
- Application designers followed those standards for instances of those types

But what happens when we need to change the internal representation or a data item or of a whole family of data items?
- Y2K crisis
- DDA inflation

Original OOP Concept

- Objects represent data items
- An object's behavior is encapsulated with the definition of the class of which it is an instance
- An object's internal representation is hidden from all modules outside the class definition

Does that address some of what's wrong with non O.O. numeric data processing?

Why are we talking about this?

- Most textbooks, most courses, and many OOP practitioners either:
  - Don't use objects at all for numeric computation, or
  - Use objects with serious flaws and errors. (www.idinews.com/opsOverload.html)

Yet, computational software development projects can reap significant benefits from OOP.
Natural Data Type Taxonomy

■ An elementary data item cannot be decomposed into component data items(126,200),(410,257)
■ A composite data item is a fixed aggregate of other (elementary or composite) data items
■ A container data item is a structure into which we can store other data items (Java calls them collections.)

Are there any others?
What about GUI objects? Sound? Video?

Further Data Taxonomy

■ Every elementary data item belongs to one of these four categories:
  - discrete (coded) data
  - numeric data
  - text data
  - logical (Boolean) data

Today's subject
Definition

- A **numeric data item** is an elementary item on which some arithmetic operation yields a meaningful result.

Properties of a numeric data item

- Unit of measure
- Range
- Precision & scale
- Interactions with other numeric types

What else?

False Numerics

Why?

- COBOL and some representation-based data dictionary systems consider these to be numeric data items:
  - ZIP code
  - account number
  - employee number

If they're not numeric then what are they?

Natural Data Type Taxonomy so far

Designing a numeric data class--Two things to do:

A. Choose the **internal representation** (i.e. the hidden internal private data) to support the unit of measure, range, precision, etc.

B. Specify an object's **behavior** (i.e. its public interface)

- what an object can do, or
- the operations you can perform on:
  - an object of the class
  - a pair of objects of the class
  - an object of the class and some other object

Which should we do first? Why?
What behavior?

- C++, Smalltalk, and other reasonable O.O. languages allow us to extend the meaning of a built-in binary operator. These come from C:
  
  | +   | -   | *   | /   | %   | &   | |   | <<= | >>= |
  | +=  | -=  | *=  | /=  | %=  | &=  | |=  | <<= | >>= |
  
  for operands that are either:
  - 2 objects of one class, or
  - a pair of objects of different classes

- For numeric objects, such overloaded operator definitions will constitute much, often most, of the defined behavior.

Examples

We're going to look at four design patterns

1. pure numeric data
   example: Rational numbers
2. the additive pattern
   example: Money
3. the point-extent pattern
   example: Date and Days
4. interacting physical quantities
   example: Electrical circuit elements

1. Defining a pure numeric class

- Examples come from mathematics. (There's no unit of measure)
  - Complex numbers
  - Rational (fractions) numbers
  - Huge integers
  - Exact decimal arithmetic
  - . etc.

- No decisions to make about which arithmetic operators to overload, because these classes are closed under all four operations + − * / (except division by 0).

A Rational Number Class

- What use is it?

  - Proposed internal representation:
    - Pair of long integers (numerator & denominator)
    - No common divisor*
    - Denominator always positive*

  - Behavior:
    - Closed under all 4 arithmetic operators
    - All six relational (Boolean) operators

  What else?
2. The **additive pattern**

**Example: Money**

- Amounts of money are among the most common data items in business applications.
- C, C++, Java provide no built-in support for money data. We must define our own class.
- Robert Glass and others often cite that shortcoming as a reason to avoid C++ and Java for business applications!

![Money icon]

### Money class: minimum requirements

- **Unit:** U.S. dollar (or any other national currency)
- **Range:** 
  - $-100,000,000,000 to $100,000,000,000
- **Precision:** $.01

---

**Money: designing the operators**

<table>
<thead>
<tr>
<th>operator</th>
<th>left operand</th>
<th>right operand</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Money</td>
<td>Money</td>
<td>Money</td>
</tr>
<tr>
<td>-</td>
<td>Money</td>
<td>Money</td>
<td>Money</td>
</tr>
<tr>
<td>*</td>
<td>Money</td>
<td>Money</td>
<td>?</td>
</tr>
</tbody>
</table>

```c++
// Data declarations
// -----------------
Money unitPrice;
Money totalPrice;
float commissionRate = 0.12;
int quantityOrdered;
Money commission;
Money totalCommission;
```

```c++
// Procedural program logic
// ------------------------
.
.
totalPrice = unitPrice * quantityOrdered;
totalCommission = totalPrice * commission;
```

What should these multiplications yield?
Money: designing the operators

<table>
<thead>
<tr>
<th>operator</th>
<th>left operand</th>
<th>right operand</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ -</td>
<td>Money</td>
<td>Money</td>
<td>Money</td>
</tr>
<tr>
<td>*</td>
<td>Money</td>
<td>Money</td>
<td>illegal!</td>
</tr>
<tr>
<td>*</td>
<td>Money</td>
<td>pure number</td>
<td>Money</td>
</tr>
<tr>
<td>/</td>
<td>Money</td>
<td>Money</td>
<td>Money</td>
</tr>
<tr>
<td>/</td>
<td>Money</td>
<td>pure number</td>
<td>Money</td>
</tr>
<tr>
<td>/</td>
<td>pure number</td>
<td>Money</td>
<td>illegal!</td>
</tr>
</tbody>
</table>

This is a pattern that will also apply to some other common numeric types.
Should we provide ++ and -- ? What should they mean?
What about % (modulo, remainder)?

What about the internal representation of Money data?

- Need to address Glass's complaint
  - Do accountants accept that
    $10,384,562.499999999999$
    is $10,384,562.50$ ?
- Many ways of solving that problem in C family

A regrettable phenomenon

- Many financial applications in C++ and even more in Java don't use Money objects at all.
- They represent amounts of money as floating-point primitives, as if they were coding in Fortran.

Why is that?
What's wrong with that?

Generalizing the additive pattern

- Many physical quantities follow exactly the same pattern of operations as Money:
  - Weight
  - Duration (ElapsedTime)
  - Distance
  - Plane Angle
  - ...  
  - What's different about this one?
- We don't have to think about which ones to define; just reuse the pattern.
3. The point-extent pattern
   example: Date

- Next to amounts of money dates are among the most common numeric data in business applications, and are also needed in some scientific/engineering applications.
- C and C++ provide even worse support for dates than for amounts of money.
- Java's support for dates is an abomination. See [www.idinews.com/javaDate1.html](http://www.idinews.com/javaDate1.html)

Date class
   requirements

- Units:
  Gregorian calendar (when it applies)
- Range:
  at least 1000 years
- Precision
  1 day

Date: designing the operators

<table>
<thead>
<tr>
<th>operator</th>
<th>left operand</th>
<th>right operand</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Date</td>
<td>Date</td>
<td>?</td>
</tr>
<tr>
<td>-</td>
<td>Date</td>
<td>Date</td>
<td>?</td>
</tr>
</tbody>
</table>

- What is July 4, 1776 plus December 7, 1941?
- What is January 1, 2008 minus December 25, 2007?

Adding & subtracting dates

- Addition of two dates is nonsense. It must be illegal. How do we tell the compiler?
- However subtraction of one date from another is meaningful; it yields a number of days.
  - Thus, we need a companion Days class
  - Days by itself observes the additive pattern
- It follows that adding or subtracting a Days object and a Date object must yield a Date object.
Generalizing the point-extent pattern

- A `Date` object represents a point in time;
- A `Days` object represents an extent of time.

The point-extent pattern occurs elsewhere:
- point in plane
- latitude-longitude
- time of day
- pointer to memory
- temperature
- 2-dimensional distance vector
- great circle distance
- elapsed time
- memory offset
- ???

We can apply our general pattern to any of those pairs.

The complete pattern

<table>
<thead>
<tr>
<th>operator</th>
<th>left operand</th>
<th>right operand</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Point</td>
<td>Point</td>
<td>illegal!</td>
</tr>
<tr>
<td>-</td>
<td>Point</td>
<td>Point</td>
<td>Extent</td>
</tr>
<tr>
<td>+ -</td>
<td>Extent</td>
<td>Point</td>
<td>Point</td>
</tr>
<tr>
<td>*</td>
<td>Extent</td>
<td>Extent</td>
<td>illegal!</td>
</tr>
<tr>
<td>*</td>
<td>Point</td>
<td>anything</td>
<td>illegal</td>
</tr>
<tr>
<td>*</td>
<td>Extent</td>
<td>pure number</td>
<td>Extent</td>
</tr>
<tr>
<td>/</td>
<td>Point</td>
<td>anything</td>
<td>illegal!</td>
</tr>
<tr>
<td>/</td>
<td>Extent</td>
<td>pure number</td>
<td>Extent</td>
</tr>
</tbody>
</table>

Note that extents are Additive.

Packaging a pattern for reuse

- Awkward in C++
- Depends on macro preprocessor or some kind of code template
- Impossible in Java

4. Beyond point extent: Interacting physical quantities

- Objectives:
  - Preserve units integrity
  - Preclude illegal operations
  - Simplify programming
  - Model the application domain

- Example:
  - DC electrical circuit elements
**Direct Current Electrical Quantities**

- Some types (classes) required:
  
<table>
<thead>
<tr>
<th>Data type</th>
<th>Unit of measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potential</td>
<td>Volts</td>
</tr>
<tr>
<td>Current</td>
<td>Amperes</td>
</tr>
<tr>
<td>Power</td>
<td>Watts</td>
</tr>
<tr>
<td>Resistance</td>
<td>Ohms</td>
</tr>
</tbody>
</table>

- Some operations:
  - power  = voltage * current
  - voltage = current * resistance

**Defining the required operators**

- Ohm's law,  \( V = I \times R \) demands 4 operator definitions for
  - \( V = I \times R \)
  - \( V = R \times I \)
  - \( I = V / R \)
  - \( R = V / I \)

- So does \( P = I \times V \)

- That's awfully tedious to prepare
  
  *Why not just define the ones our own program needs?*
  
  *Is it worth the bother?*  
  
  ("YAGNI")

---

**A troublesome operator issue**

- Electrical engineers often want to combine the two earlier formulas
  \( V = I \times R \) and \( P = I \times V \)
  into \( P = I \times I \times R \)

- What would we need to define to support that? What problems arise?

  *(An exercise for the student)*

**A special operator opportunity**

- In modeling a circuit we'd like to express combining resistances both in series and in parallel.
  - Series is just addition \( R = R1 + R2 \)
  - Parallel is \( R = (R1 \times R2) / (R1 + R2) \)

- C's *bitwise* boolean operators suggest a simpler notation
  - Series combination: \( R1 \& R2 \)
  - Parallel combination: \( R1 \mid R2 \)

  *We can do this. Should we?*
A raging controversy among OOP insiders

- Never define any operator to mean anything other than what it originally meant in C.
- If you do, it will confuse the reader of a program and impair readability.
- Nonsense! Operator notation is simple and natural.
- There's precedent in standard C++.
- There's no chance of anyone's misinterpreting | or & between resistances.

Which point of view is more sensible?

Code fragment and output #1

```
cout << endl << "1: Simple voltage, current, and power" << endl;
Potential v1 = 115.0; display(v1);
Current c1 = 15.0; display(c1);
Power w = v1 * c1; display(w);
display(w / c1);
display(w / v1);
```

1: Simple voltage, current, and power
v1 = 115 volts
c1 = 15 amperes
w = 1725 watts
w / c1 = 115 volts
w / v1 = 15 amperes

Code fragment and output #2

```
cout << endl << "2: Resistances in series and in parallel" << endl;
Resistance r1 = 6.0; display(r1);
Resistance r2 = 4.0; display(r2);
Resistance r3 = r1; display(r3);
display(r1 & r2);
display(r1 | r2);
display(r1 & (r2 | r3));
```

2: Resistances in series and in parallel
r1 = 6 ohms
r2 = 4 ohms
r3 = 6 ohms
r1 & r2 = 10 ohms
r1 | r2 = 2.4 ohms
r1 & (r2 | r3) = 8.4 ohms

Code fragment and output #3

```
cout << endl << "3: More complicated computations" << endl;
cout << v1 << " across " << r2 << " gives " << v1 / r2 << endl;
cout << c1 << " through " << r2 << " requires " << c1 * r2 << " and uses " << c1 * (c1 * r2) << endl;
cout << "4 60-watt bulbs use " << 4 * Power(60) << " and draw " << Power(60) / v1 * 4 << " at ";
cout << "A " << v1 << ", " << c1 << " circuit can support " << int(v1 * c1 / Power(60)) << " 60-watt light bulbs.";
```

3: More complicated computations
115 volts across 4 ohms gives 28.75 amperes
15 amperes through 4 ohms requires 60 volts and uses 900 watts
4 60-watt bulbs use 240 watts and draw 2.08696 amperes at 115 volts
A 115 volts, 15 amperes circuit can support 28 60-watt light bulbs.
Other physical systems / laws that OOP can model

- Newtonian motion: $f = m \cdot a$
  $a = \frac{d^2s}{dt^2}$
- Boyle’s law for gases
  $P \cdot V / T = R$
- High-energy particle physics
- Etc.

What’s wrong with Java for numeric computing?

- Almost everything
  - can’t define operators
  - object reference semantics and overhead
- Java is really two languages:
  - one for primitive data (built-in types)
  - one for everything else (references)

Example

- This is legal in both C++ and Java:

```cpp
double creditLimit;
double unitPrice = 49.95;
double totalPrice = 0;
int quantityOrdered;
.
.
totalPrice += quantityOrdered * unitPrice;
if (totalPrice > creditLimit) . . .
.
```

- Changed to use Money class

```cpp
Money creditLimit;
Money unitPrice = Money(49.95);
Money totalPrice = Money(0);
int quantityOrdered;
.
.
totalPrice += quantityOrdered * unitPrice;
if (totalPrice > creditLimit) . . .
.
```

Which lines would have to be changed to use a Money class in C++? in Java?
**Changed to use Money class**

- **Java version:**
  ```java
  Money creditLimit;
  Money unitPrice = new Money(49.95);
  Money totalPrice = new Money(0);
  int quantityOrdered;

  totalPrice.addSet(unitPrice.mpy(quantityOrdered));
  if (totalPrice.greaterThan(creditLimit)) .

  total += unitPrice * quantityOrdered
  ```

**Why doesn't Java support operator definition?**

- "...the language designers decided (after much debate) that overloaded operators were a neat idea, but that code that relied on them became hard to read and understand." - David Flanagan: *Java in a Nutshell*

  ```java
  total += unitPrice * quantityOrdered
  must be harder to read and understand than
  total.addSet(unitPrice.mpy(quantityOrdered))
  or
  total.setValue(total.getValue() +
          unitPrice.getValue() * quantityOrdered)
  ```

**What do Java programmers do?**

- Many Java programmers don't bother with numeric objects. They either:
  - Ignore OOP altogether for numeric computing, as if they were coding in Fortran, or
  - Use awkward wrapper classes, with accessor functions that violate OOP

**Conclusion:**

- Java is a horrible choice for computation.
- But if you have to use Java, use numeric objects anyway.
  - Everything we've shown is possible (although ugly) in Java.

**Recap**

- **Advantages of using numeric objects:**
  - Units integrity
  - Range integrity
  - Easier component reuse
  - Higher abstraction level
    - Code is closer to problem
    - Code readability
    - Ease of making changes
    - Productive coding & unit testing

- **Disadvantages:**
  - None
Further information

- www.idinews.com

Further misinformation

- Many books, articles, courses on OOP
  *Such as?*