Point-Extent Pattern for Dimensioned Numeric Classes

Conrad Weisert, Information Disciplines, Inc., Chicago

(Originally published in ACM *SIGPLAN Notices*, November, 1997; revised and expanded, January, 1998)

When we design an elementary numeric class, we usually overload some of the arithmetic operators, +, —, *, /. Deciding which operators to overload and how to define them is a matter of common-sense and high-school algebra. Our class must support *all* the operators that make mathematical sense and *none* of the ones that don't.

That seems obvious, but a lot of courses and textbooks that tell us all about *how* to overload an operator give us little guidance on *when* to do so.

Closed domains – a rare exception

Suppose ž is any binary operator and Â is a class. Then Â is said to be *closed under* ž if xžy is an instance of Â whenever x and y are instances of Â.

Rational numbers, real numbers, and complex numbers are closed under all four arithmetic operations (except for division by zero). Therefore, any classes we define that deal with such *pure numbers* should overload all four arithmetic operators accordingly.

On the other hand, most of the numeric types we encounter in programming have some dimension or *unit of measure*. Unlike pure numbers they are rarely closed under all operations. For example:

- **`Money`** is closed under addition and subtraction, but not under multiplication or division.
- **`Date`** and Cartesian **`Point`** are not closed under any of the four operations.

Beginners may be tempted to define *every* operator as is if a class were closed under it, and even textbook authors slip up occasionally. Defining too *few* overloaded operators leads to client inconvenience and annoyance, but defining too *many* threatens type safety and program integrity.

Results: outside the class or nonsense

Some operations make no sense at all, e.g. multiplying two dates or two amounts of money. Others make good sense, but yield a result of another type, such as these:

- Dividing **`Money`** by **`Money`** yields a *pure number*. (How many 99¢ items do you get for $5.94?)
- Subtracting a **`Date`** from another **`Date`** yields neither a date nor a pure number but a *duration*. If we're building a **`Date`** class, then we must also build a companion duration (or number of **`Days`**) class.

In C++ those two examples could be:

```
double operator/ (MONEY[1] ls, MONEY rs);
Days   operator- (DATE  ls, DATE  rs);
```

Operators with mixed operands

From these two C++ examples, we derive the following just by rearranging terms:

```
Money operator/(MONEY ls, DOUBLE, rs);
Date  operator+(DATE  ls, DAYS    rs);
```

We have no choice here; we *must* do this if the class is to behave as client programs expect. Furthermore, since addition is commutative, we must also provide:

```
Date  operator+ (DAYS  ls, DATE  rs);
```

but not, of course **`operator/(DOUBLE, MONEY)`**, since that result ("inverse dollars") is meaningless.

Ad hoc design versus a pattern

Nothing so far is controversial. Almost everyone will agree, but will we always think of all the special cases? Fortunately, we can generalize the rules for many numeric classes, certainly as a design pattern and, we hope, also as actual reusable code.

Let's begin with a familiar example, specifying all combinations of operands for all four binary arithmetic operators. Then, we'll generalize that example as a pattern to apply to other elementary numeric classes.

Example: `Date`

Once they understand the principle, everyone agrees on these **`Date`** and **`Days`** (duration) operations:

`Date`	**`+`**	**`Date`**	→ illegal
`Date`	**`+`**	**`Days`**	→ **`Date`**[2]
`Days`	**`+`**	**`Days`**	→ **`Days`**
`Date`	**`–`**	**`Date`**	→ **`Days`**
`Date`	**`–`**	**`Days`**	→ **`Date`**
`Days`	**`–`**	**`Date`**	→ illegal
`Days`	**`–`**	**`Days`**	→ **`Days`**
`Date`	**`*`**	anything	→ illegal[2]
`Days`	**`*`**	**`Days`**	→ illegal
`Days`	**`*`**	pure number	→ **`Days`**[2,3]
`Date`	**`/`**	anything	→ illegal
anything	**`/`**	**`Date`**	→ illegal
`Days`	**`/`**	**`Days`**	→ pure number
`Days`	**`/`**	pure number	→ **`Days`**
Pure number	**`/`**	**`Days`**	→ illegal

---

[1] To save space I follow the convention that an all upper-case class name includes **`const`**.

[2] and conversely for the commutative operator.

[3] This one is an inevitable consequence of Days + Days

The pattern emerges

Suppose we're designing classes later to deal with two-dimensional geometric figures. One such class is a *point* in the Cartesian plane, having X and Y coordinates. We note that it makes no sense to add two such points, but that subtracting two points yields a *plane vector* or directed *distance*. As we continue we see that points and directed distance have the same relationship to each other as dates and days. The operations we need for these new classes are exactly the same as for **`Days`** and **`Date`** (but not the same as for **`Money`**).

This suggests that we can generalize the pattern to pairs of related dimensioned classes: a *point* (in time, in space, or in something else) class and a corresponding *extent* (of time, of space, etc.) class.[4]

If we go back and substitute the generic *extent* for **`Days`** and *point* for **`Date`** in our table of operations, we get the right results, except that we may sometimes choose to let *extent* * *extent* yield an object of a corresponding *square extent* class, e.g. **`Area`**.

For clarity it's helpful to rearrange our table like this:

1. Operations with extent operands:

extent	**+**	extent	→ extent
extent	**-**	extent	→ extent
extent	*****	extent	→ illegal (or square extent)
extent	*****	pure number	→ extent
pure number	*****	extent	→ extent
extent	**/**	extent	→ pure number
extent	**/**	pure number	→ extent
pure number	**/**	extent	→ illegal

2. Operations with point operands:

point	**+**	point	→ illegal
point	**-**	point	→ extent
point	*****	anything	→ illegal
Anything	*****	point	→ illegal
point	**/**	anything	→ illegal
Anything	**/**	point	→ illegal

3. Operations with both point and extent operands:

point	**+**	extent	→ point
extent	**+**	point	→ point
point	**-**	extent	→ point
extent	**-**	point	→ illegal

[4] Alternative terms: *position-amount*, *absolute-relative*, and *location-displacement*.

Applicability of the pattern

Beyond **`Date`** and **`Days`**, what other pairs of classes fit this pattern? Some don't (e.g. velocity and acceleration), but here are five that do:

- **`Time of day`**[5], by obvious analogy with **`Date`**.
- Geometric **`points`** and **`vectors`** in 3-dimensional (or higher) space.
- **latitude-longitude** and **great-circles** on the surface of the earth (or any sphere).
- Less obvious is **`Temperature`**, since we lack a common name for *temperature change*.[6]
- In addressable memory, **`address`** and **`offset`**[7] fit the pattern. PL/I pointers support this duality explicitly, while C reverts to integer type for an offset. This is not the common use of offset as a synonym for *displacement* from a segment origin.

In addition, we can apply just the *extent* portion of the pattern to other numeric types that have a unit of measure, but lack the point-extent duality. For example:

`Money`	**+**	**`Money`**	→ **`Money`**
`Money`	**-**	**`Money`**	→ **`Money`**
`Money`	*****	**`Money`**	→ illegal
`Money`	*****	pure number	→ **`Money`**
Pure number	*****	**`Money`**	→ **`Money`**
`Money`	**/**	**`Money`**	→ pure number
`Money`	**/**	pure number	→ **`Money`**
Pure number	**/**	**`Money`**	→ illegal

No common derivation

Often, but not always, a point class and its corresponding extent class share an internal data representation and a few methods. It's tempting, therefore, to try to derive one from the other or to derive both from an abstract base class. Such attempts, however, can complicate the pattern by legalizing all sorts of things we don't want, which we must then override to get rid of. It's usually simpler to design the two as independent classes, friends if necessary. And we shouldn't allow implicit conversion between them.

[5] Some designers prefer to combine date and time-of-day into a combined **`Time`** class; we'd then need a single **`Duration`** class.

[6] An amusing misunderstanding arose when a newspaper copy editor, told to convert metric units to English, replaced 1.5°C with 35°F in an article explaining the amount of global warming expected in the next century.

[7] Note that this use of "offset" is not the same as in the Intel segmented (16-bit) memory structure.

Object orientation?

Although we can easily implement point-extent in today's object-oriented languages, there's nothing really object-oriented about it. We can do the same in any language that supports type definition and operator overloading, such as Ada and (1961!) MAD.

Packaging for reuse

Just having the point-extent pattern at hand will save us a lot of work and, more important, a lot of errors. But we shouldn't stop there. Whenever we find ourselves writing the same pattern over and over, we should, of course, try to isolate the actual code for easy reuse.

Java's approach – forget it!

Java, in most respects an object-oriented language, doesn't support overloaded operators at all. That choice, we're told, was made not in order to keep the language small, but in order to protect programmers from *misusing* the facility!

". . 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." [10]

So instead of this (C++):

```
total = unitPrice * quantity + shipChg;
```

we get this something like this (Java):

```
total.setvalue(
  shipChg.add(unitPrice.multiply(quantity)));
```

You may wonder about people who think the second is easier to read and understand than the first, but it's unlikely anyone really does. More likely, Java just wasn't designed for non-trivial type-safe computation. If you're going to do much with elementary numeric classes, Java isn't a suitable language.[11]

C++ experts may want to explore either (a) *multiple inheritance* using mix-in generalized point and extent classes or (b) *class templates* for generalized points and extents. However I haven't found a clean way to exploit either of those facilities to implement this pattern without either running afoul of restrictions or interfering with some other use of classes or class templates.

Much simpler is a preprocessor package, provided we're willing to follow a few conventions. I shall describe such a scheme in a companion article, "Implementing the Point-Extent Pattern in C++.

C++ epilog – the derived operator pattern

If we're working in operator-rich C++, we're not done yet. We still have to define:

- compound assignment operators corresponding to each binary operator, e.g. **`*=`**
- the increment and decrement prefix and postfix operations, **`++`** and **`--`**
- unary minus
- the modulo operator **`%`**

These are automatic, requiring no thought, independent of the *point-extent pattern*. We may consider the following rules a mini-pattern: For *any* numeric class intended to be consistent with C's operator repertoire and thus support client programs' reasonable expectations:

- Whenever **`x=x`**W**`y`** is meaningful for **`x`** an object of a class then we must define **`x`**W**`=y`** in the obvious way.
- If **`x+=1`** is meaningful, then we must define the increment and decrement operators in the obvious way.
- If **`0-x`** is meaningful then **`-x`** must be supported.
- Whenever **`x/y`** is meaningful, then **`x%y`** should be defined[8], yielding result of the class of **`x`**.

We can usually define the above in terms of the four basic numeric operations.[9]

Also note that automatic conversions force us to compromise in a few places if we're going to allow *literal* constant extents, such as:

```
Days vacation = 14;
```

If we allow that, then we can't prevent addition of a pure number to a duration. The new **`explicit`** constructors will help us control some but not all such conversions.

---

8 C itself violates this rule for floating point, an irritating complication when we write function templates.

9 For efficiency Scott Myers (*More Effective C++*, 1996, Addison Wesley, ISBN 0-201-63371-X, Item #22) recommends defining the compound assignments *first*, and then defining the binary operations in terms of them. In some cases, that avoids the need to construct a temporary.

10 David Flanagan, *Java in a Nutshell*, O'Reilly Associates, 1996, p. 35.

11 Actually, if you're determined, you can indeed apply the pattern in Java. See www.idinews.com/arithOps.pdf.