ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

Design assessment

- the OO solution represents several improvements over the initial implementation
  - the problem has been decomposed into several parts that can evolve independently
  - there is a natural correspondence with the mathematics of Monte Carlo integration
  - our driver assembles the pieces together in a natural fashion
- our abstract base classes
  - do not play a string rôle in dynamically typed languages, such as python
  - in strongly typed languages, they become constraints on their subclasses
- change involves modifying the script and running again
  - how would you build the performance table, or compare multiple random number generators?

Reassembling the pieces

if we are interested in turning our simple script into a package, we should build a class to take responsibility of managing all the necessary parts; consider the class `Integrator`

```
class Integrator:
    """
    The abstract base class for integrators
    """

    # interface
    def integrate(self):
        """
        Integrate my {integrand} over my {region}
        """
        raise NotImplementedError(
            "class {.__name__!r} should implement 'integrate'".format(type(self)))
```

and let descendants specify the details of the quadrature

The Monte Carlo integrator

perhaps something like

```python
from Integrator import Integrator

class MonteCarlo(Integrator):
    """
    A Monte Carlo integrator
    """

    # public state
    samples = 10**5 # default value?
    box = ????
    mesh = ????
    region = ????
    integrand = ????

    # interface
    def integrate(self):
        """
        Integrate my {integrand} over the {region} by sampling it using random
        numbers from {mesh}
        """
        # compute the normalization
        normalization = self.box.measure() / self.samples
        # build the sample set
        points = self.mesh.points(n=self.samples, box=self.box)
        # narrow the set down to the interior points
        interior = self.region.interior(points=points)
        # build the value of the integral
        integral = normalization * sum(self.integrand.eval(interior))
        # and return it
        return integral
```

where the parts are specified at *runtime*?

Inversion of control

let's be a bit more ambitious:

- can we postpone until *runtime* the selection, instantiation and initialization of some subset of the classes in our applications?
- and hence give the end user total control over *what* and *how*?

the correct solution would to the application like a user interface

Small steps: properties

let's step back and contemplate a simpler problem

```
class Disk:

   # public state
   radius = 1  # default value
   center = (0,0)  # default value

   # interface
   def interior(self, points):
      ...
```

what do we have to do to tie instances of `Disk` with information in some configuration file

```
   [ disk1 ]
   center = (-1,1) ; leave {radius} alone

   [ disk2 ]
   radius = .5
   center = (1,1)
```

or, equivalently, from the command line

```
gauss.py --disk1.center=(1,1) --disk2.radius=.5 --disk2.center=(-1,1)
```

Components

- informally, *classes* are software specifications that establish a relationship between *state* and *behavior*
  - we have syntax that let's us specify these very close to each other
- *instances* are containers of state; there are special rules
  - that grant access to this state
  - allow you to call functions that get easy access to this state
- *components* are classes that specifically grant access to some of their state to the end user
  - the public data are the *properties* of the component
- rule 1: components have properties

A trivial component

`pyre` is a package that provides support for writing components

```python
import pyre

class Disk(pyre.component):

    # public state
    radius = pyre.properties.float()
    radius.default = 1
    radius.doc = 'the radius of the disk'

    center = pyre.properties.array()
    center.default = (0,0)
    center.doc = 'the location of the center of the circle'

    # interface
    ...
```

why bother specifying the type of component properties?

- command line, configuration files, dialog boxes, web pages: they all gather information from the user as strings
- we need *metadata* so we can convert from strings to the intended object

The names of things

in order to connect components to configurations, we need explicit associations

- component instances must be given unique names
- component classes must be given unique family names
- components belong to packages

```python
import pyre

class Disk(pyre.component, family="gauss.shapes.disk"):

    # public state
    radius = pyre.properties.float()
    radius.default = 1
    radius.doc = 'the radius of the disk'

    center = pyre.properties.array()
    center.default = (0,0)
    center.doc = 'the location of the center of the circle'
    ...
```

and here are a couple of component instances

```python
left = Disk(name='disk1')
right = Disk(name='disk2')
```

Configuration

- the package name is deduced from the component family name
  - it is the part up to the first delimiter
- `pyre` automatically loads configuration files whose name matches the name of a package
- there's even a way to override the default values that the developer hardwired into the class declaration

```
[ gauss.shapes.disk ] ; the family name
radius = 2
center = (-1,-1)

[ disk1 ] ; the name of an instance
center = (-1,1) ; leave {radius} alone

[ disk2 ] ; the name of another instance
radius = .5
center = (1,1)
```