The `StagedTaskCollection`

GC3: Grid Computing Competence Center,
University of Zurich

Oct. 2, 2012

Running jobs in sequence

`StagedTaskCollection` provides a simplified interface for constructing sequences of jobs, but it only applies when the number and content of steps is *known and fixed* at programming time.

(By contrast, the most general `SequentialTaskCollection` can alter the sequence on the fly, insert new stages while running and loop back. But the code is also harder to write.)

```
class Pipeline (StagedTaskCollection):
  def __init__(self, image):
    self.source = image

  def stage0(self):
    # run 1st step
    return Application(...)

  def stage1(self):
    if self.tasks[0].execution.exitcode != 0:
      self.execution.exitcode = 1
      return Run.State.TERMINATED
    else:
      # run 2nd step
      return Application(...)

  # ...
  def stageN(self):
    # ...
```

Example of a `StagedTaskCollection` subclass.

```
class Pipeline(StagedTaskCollection):
  def __init__(self, image):
    self.source = image

  def stage0(self):
    # ...

  def stage1(self):
    # ...

  # ...
  def stageN(self):
    # ...
```

Stages are numbered starting from 0.

You can have as many stages as you want.

```
class Pipeline(StagedTaskCollection):
  # ...

  def stage0(self):
    # run 1st step
    return Application(
      ['convert', self.source,
       '-colorspace', 'gray',
       'grayscale_' + self.source],
      inputs = [self.source],
      ...)

  # ...
```

Each `stage`N method can return a `Task` instance, that will run as step N in the sequence.

```
class Pipeline(StagedTaskCollection):
  # ...

  def stage1(self):
    if self.tasks[0].execution.exitcode != 0:
      self.execution.exitcode = 1
      return Run.State.TERMINATED
    else:
      # run 2nd step
      return Application(...)

  # ...
  def stageN(self):
    # ...
```

In later stages you can check the exit code of earlier ones, and decide whether to continue the sequence or abort.

```
class Pipeline(StagedTaskCollection):
  # ...

  def stage1(self):
    if self.tasks[0].execution.exitcode != 0:
      self.execution.exitcode = 1
      return Run.State.TERMINATED
    else:
      # run 2nd step
      return Application(...)

  # ...
  def stageN(self):
    # ...
```

To abort the sequence, return `Run.State.TERMINATED`, instead of a `Task` instance.

```
class Pipeline(StagedTaskCollection):
  # ...

  def stage1(self):
    if self.tasks[0].execution.exitcode != 0:
      self.execution.exitcode = 1
      return Run.State.TERMINATED
    else:
      # run 2nd step
      return Application(...)

  # ...
  def stageN(self):
    # ...
```

Don't forget to set the `StagedTaskCollection`'s own exit code if you do this.

Detour: grayscaling an image

The ImageMagick command to reduce an image to grayscale is:

```
$ convert image1 -colorspace gray image2
```

It reads the image in file *image1*, converts it to a black&white picture, and saves the result into file *image2*.

Detour: inverting colors

The ImageMagick command to invert colors is:

```
$ convert image1 +negate image2
```

It reads the image in file *image1*, inverts colors (black → white and reverse), and saves the result into file *image2*.

Detour: mounting images side-by-side

The ImageMagick command to invert colors is:

```
$ montage image1 image2 -tile 2x1 image3
```

It reads files *image1* and *image2*, creates a combined picture by putting the two side-by-side[1] and saves the result into file *image3*.

[1] a tile with 2 columns by 1 row

Exercise A: Write a *SideBySide* sequence.

The sequence is initialized with the file name of a picture:

```
sbs = SideBySide('fig/lena.jpg')
```

The sequence runs the following steps on the input image:

1. Convert it to grayscale.
2. Invert colors in the grayscale picture.
3. Mount the two images side by side and write them into a final output image.

Plug this class into your standard session based script and verify that it works.