Introduction to Design Patterns & Singletons

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction

Last time we talked about the development of a body of knowledge around Object Oriented Design that included SOLID Principles.

Another major part of that project is the idea of *Design Patterns*. The principal reference on that topic is the 1995 book *Design Patterns: elements of reusable object-oriented software* by Gamma, et al. (Typically known of Gang of Four or GoF). The book describes 23 patterns.

What is a Design Pattern?

A design pattern is not code. It's a description of how code (typically a class) should function. It describes a broadly useful approach to solving a common problem.

Patterns are generally language-independent, although their implementations are not. They are typically language paradigm-dependent, so that OO patterns might not apply to other language types.

So what?

The primary test of the utility of a particular design pattern is that it **works**. It solves the problem it's intended to solve. When you're confronted by a problem that is probably pretty common, look to see if there is a well known pattern that solves it.

Conversely, it's good to know about common patterns so that you recognise common problems when you see them. Using a well-known pattern also helps other programmers understood what you are doing with your code.

Criticisms

Some criticsims of design patterns have been raised.

- Design patterns merely point out shortcomings in particular languages.
- Patterns lead to needless complexity in software design.

The Singleton Pattern

One well known pattern is the *Singleton*.

"Ensure a class only has one instance and provide a global point of access to it." (Gof)

Applications for a singleton include a central logging object, a configuration object, or a cache.

In Python, `None` is a singleton.

Singleton: Classic Implementation

```
class SoloGoF:
  _instance = None  # This is a class variable

  def __init__(self):
      raise RuntimeError('Use instance() instead.')

  @classmethod
  def instance(cls):
      if not cls._instance:
          cls._instance = cls.__new__(cls)
          # extra initialisation can happen here
      return cls._instance

  solo = SoloGoF.instance()
```

Singleton: A More Pythonic Implementation

```
class SoloPy:
  _instance = None

  def __new__(cls):
      if not cls._instance:
          cls._instance = super(SoloPy, cls).__new__(cls)
          # extra initialisation can happen here
      return cls._instance

  solo3 = SoloPy()
```

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `10-practical` in your practicals repo.
3. Add a subdirectory, `10-practical` and copy `10-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first questions.
5. We will discuss results in 20ish minutes.

Criticisms of the Singleton

If you Google for "singelton antipattern", you'll get a lot of results.

- It's too complicated. If you only want one instance, just create one instance.
- Singletons are thinly disguised globals.
- Singletons are inflexible. What if you really need two?
- It's difficult to unit test singletons, since you can't control their instantiation.
- It's difficult to test objects that depend on singletons, especially if they contain state.

Singletons Using Modules

In Python, we have *modules* that we can *import*. Recall that a module's code is only executed one time and the results are cached for subsequent imports. In effect, modules are singletons.

Singleton: Module Implementation

```
file: solomod.py
--------
class _SoloMod:
  def __init__(self):
      # do some init stuff, idk
      pass

solomod = _SoloMod()

# alternatively:
class SoloMod:
  def __new__(cls):
      return solomod

# in a client module:
import solomod

solo = solomod.solomod
```