OOP via Python: Session 09 Design Patterns

Stephen Leach, Feb 2022

Design Patterns Book

- Design Patterns: Elements of Reusable Object_Oriented Software by Gamma, Helm, Johnson, Vlissides
- 23 solution templates for common programming situations
- Enormously influential
- But seemingly whimsical with no systematic approach (why 23? why not 123?)
- Vary wildly in their nature

Wild West

- Iterator - a dedicated class for implementing an iterator
- Visitor - a workaround for a lack of multiple dispatch (c.f. open-closed "principle")
- Singleton - workaround for inability to limit access to functions
- Builder - class for constructing complex types
- Facade - class for using some other classes

Deceptively Ambitious

The ambition is to identify enough design patterns that design becomes a matter of "stringing together patterns, in a rather loose way"

Question: Could this be the basis of a design language?

Two Important Design Patterns

- Factory Pattern - a class that creates new instances of something else
- State Pattern - a way of handing phase guards

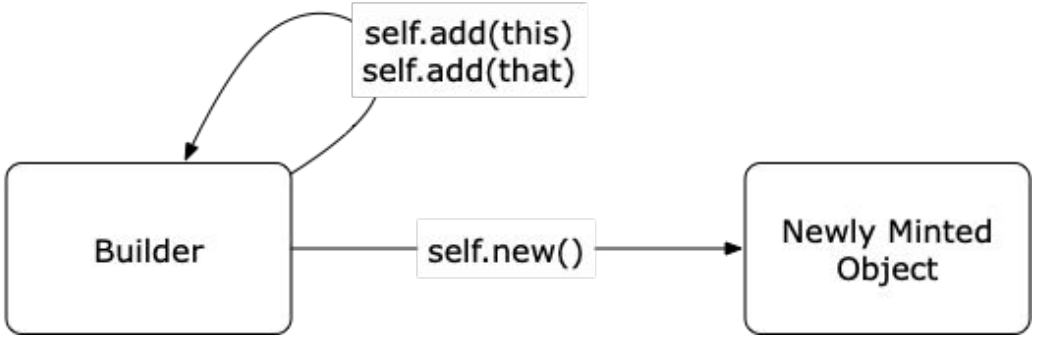
Factory Pattern

Factory Pattern

The GoF book has three easily confused patterns for creating instances:

- **Factory Method**, which is not a method
- Abstract Factory, which is not abstract and
- **Builder Pattern**, which, slightly surprisingly, is for building.

Factory Method Pattern -vs- Builder Pattern



Factory-Builder Pattern: Example

```
class TreeMaker:
    def __init__( self ): ...
    def add( self, **kwargs ): ...
    def start( self, key: str ): ...
    def end( self ): ...
    def make( self ) -> Dict[str, Any]: ...
```

Example cont'd

```
>>> m = TreeMaker()
>>> m.add( name = 'Stephen' )
>>> m.add( age = 62 )
>>> m.start( 'activities' )
>>> m.start( 'coding' )
>>> m.add( duration = 45 )
>>> m.end()
>>> m.start( 'birdwatching' )
>>> m.add( duration = 40 )
>>> m.end()
>>> m.end()
>>> tree = m.make()
>>> print( tree )
{'name': 'Stephen', 'age': 62, 'activities': {'coding': {'duration': 45}, 'birdwatching': {'duration': 40}}}
```

Why is the Factory/Builder Pattern important?

- The crucial property of methods is *dispatch* = they choose the function to execute based on the run-time implementation
- They give us a way of writing code that corresponds to what we want to do, independent of how we do it.
- But constructors do not dispatch! They are hard-coded to return a particular implementation.

Consider logging

- Suppose we wanted the Treemaker to log its activity …
- We could allocate ourselves a logging service like this …

```
class TreeMaker:
   def __init__( self ):
       self._logger = FancyLogger()
       self._focus = {}
       self._dump = []
```

- … which would be incredibly annoying if you were trying to add it into an application that used the brand new **`EvenFancierLogger`**.

Dispatch for Constructors

- From our perspective, the important aspect of the Factory/Builder type is dispatch
- And that's why it is ubiquitous in OOP

State Pattern

State Pattern

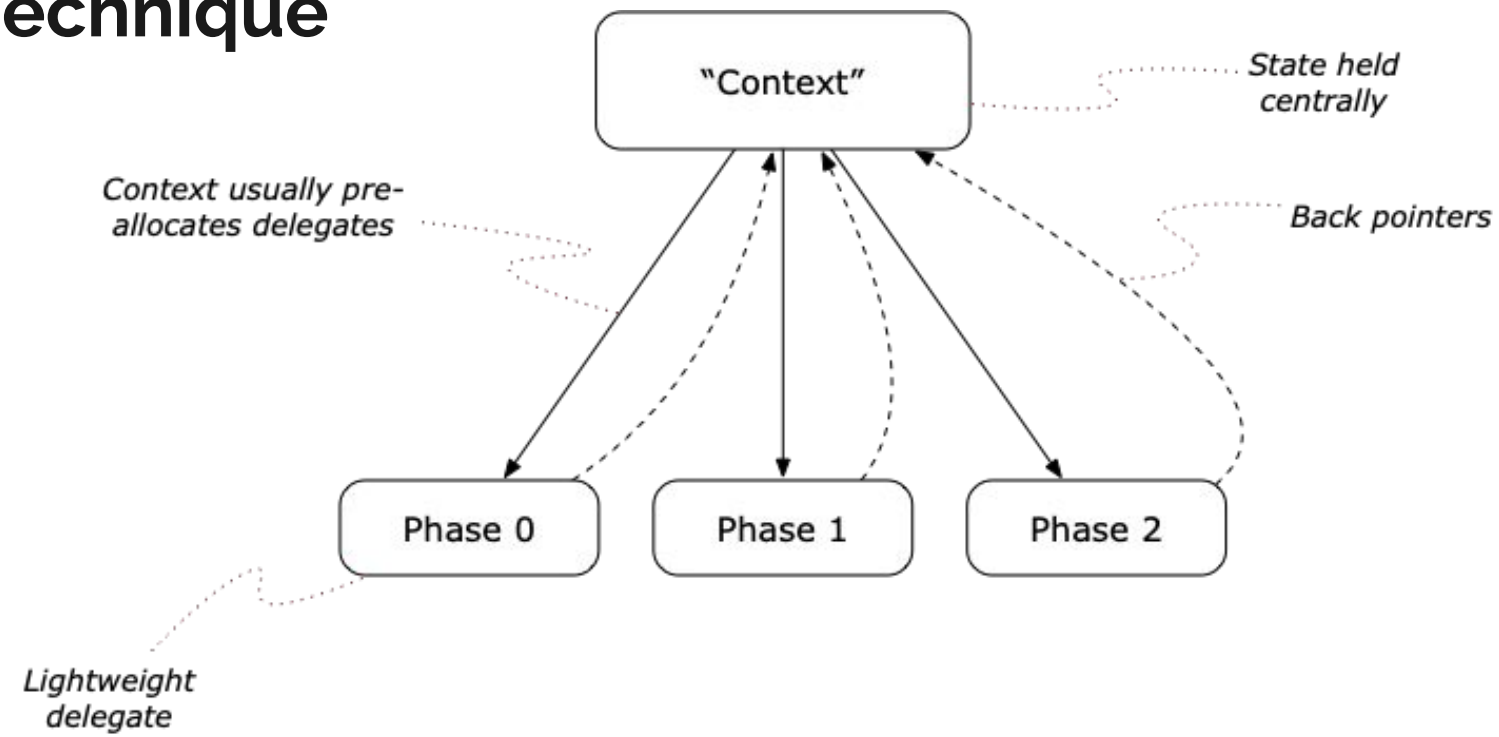
"The state pattern is intended to solve two main problems:

- [When] an object should change its behavior [in reaction to] its internal state changes.
- [When] state-specific behavior should be defined independently. That is, adding new states should not affect the ~~behavior~~ implementations of existing ~~states~~ behaviours."

Phase

- A phase is a pattern of availability (total, partial, unavailable) for a set of methods
- As an object changes phase methods change their availability
- Hence the State Pattern is the GoF Design Pattern for dealing with this
- The key technique is that we allocate sub-objects, one per phase, but retain the central state in the main object.

Key Technique



Example: Freezable List

```
class Freezable(MutableSequence):
   def __init__(self, lst, frozen=False): ...
   def freeze(self): ...
   def thaw(self): ...

   def __getitem__(self, n): ...
   def __len__(self): ...
   def __setitem__(self, n, x): ...
   def __delitem__(self, n): ...
   def insert(self, index, value): ...
```

Plain implementation: freezable.py

State Pattern Implementation: state_pattern.py

Summary of Sessions /0[1-9]/

- Session 01 Methods implementing Behaviour
- Session 02 Overlapping Behaviour
- Session 03 Object Life Cycles
- Session 04 Organisation of Code - Core vs non-core - Public vs private methods (behaviour vs implementation)
- Session 05 X-Ray Vision - How to spot opportunities to identify & extract classes from code.
- Session 06 Enhanced X-Ray Vision - How to transform function calls into objects and vice versa.
- Session 07 Data Encapsulation, Ownership and Part-Of relationships
- Session 08 SOLID
- Session 09 Design Patterns - Builder Pattern - State Pattern