"Template Method" and "Factory" Design Patterns

Alex Martelli
This talk's audience...:

- "fair" to "excellent" grasp of Python and OO development
- "none" to "good" grasp of Design Patterns in general
- wants to learn more about: DP, Template Method, Factories, DPs for Python, DP/language issues
Design Patterns

- rich, thriving subculture of the OO development culture
- Gamma, Helms, Johnson, Vlissides: "Design Patterns", Addison-Wesley 1995 (Gof4)
- PLoP conferences & books
- ...
DPS and language choice [1]

- are **not** independent
- design and implementation **must** interact
- in machine-code: "if", "while", "procedure" ... are patterns!
- HLLs embody these, so they are not patterns in HLLs
many DPs for Java/C++ are "workarounds for static typing"

cfr Alpert, Brown, Woolf, "The DPs Smalltalk Companion" (AW)

Pythonic patterns = classic ones, minus the WfST, plus optional exploits of Python's strengths
Two highly Pythonic DPs

- **Template Method**: organize elementary actions into a predefined structure/sequence
- **Factory**: control and coordinate the construction of instances
The "Template Method" DP

- great pattern, lousy name
- "template" is **very** ambiguous:
  - in C++, keyword used in "generic programming" mechanisms
  - "templating" is yet another thing (empy, preppy, YAPTU, Cheetah)
Classic Template Method DP

- abstract base class "organizing method" calls "hook methods"
- concrete subclasses implement the elementary "hook methods"
- client code calls the "organizing method" on concrete instances
class AbsBase(object):
    def orgMethod(self):
        self.dothis()
        self.dothat()

class Concrete(AbsBase):
    def dothis(self): ...
Ex: pager abstract class [1]

class AbsPager(object):
    def __init__(self, mx=60):
        self.cur = self.pg = 0
        self.mx = mx
    def writeln(self, line):
        """organizing method"""
        ...

def writeline(self,line):
    if self.cur == 0:
        self.dohead(self.pg)
    self.dowrite(line)
    self.cur += 1
    if self.cur>=self.mx:
        self.dofoot(self.pg)
        self.cur = 0
        self.pg += 1
Ex: concrete pager to stdout

class Pagerout(AbsPager):
    def dowrite(self, line):
        print line
    def dohead(self, pg):
        print 'Page %d:
' % pg + 1
    def dofoot(self, pg):
        print '\f',

Ex: concrete pager w/curses

```python
class Cursepager(AbsPager):
    def dowrite(self, line):
        w.addstr(self.cur, 0, line)
    def dohead(self, pg):
        w.move(0, 0); w.clrtobot()
    def dofoot(self, pg):
        w.getch()  # wait for key
```
Classic TM rationale

- "organizing method" provides structural logic (sequencing &c)
- "hook methods" perform actual "elementary" actions
- often-appropriate factorization of commonality and variation
The Hollywood Principle in TM

- base class \textit{calls} hook methods on self, subclasses \textit{supply} them
- it's "The Hollywood Principle":
  \begin{itemize}
  \item "don't call us, we'll call you!"
  \end{itemize}
- focus on objects' responsibilities and collaborations
A useful naming convention

- identify "hook methods" by starting their names with 'do'
- avoid names starting with 'do' for other identifiers
- usual choices remain: dothis vs doThis vs do_this
A choice for hook methods [0]

class AbsBase(object):
    def dothis(self):
        # [1] provide a default
        pass  # often a no-operation
    def dothat(self):
        # [2] force subclass to supply
        raise NotImplementedError
A choice for hook methods [1]

- can force concrete classes to provide hook methods ("purer"):
  - classically: "pure virtual"/abstract
  - Python: do not provide in base class (raises AttributeError) or raise NotImplementedError
A choice for hook methods [2]

- can provide handy defaults in abstract base (often handier):
  - may avoid some code duplication
  - often most useful is "no-op"
  - subclasses may still override (& maybe "extend") base version

- can do some of both, too
Pydiom: "data overriding"

class AbsPager(object):
    mx = 60
    def __init__(self):
        self.cur = self.pg = 0

class Cursepager(AbsPager):
    mx = 24
    #just access as self.mx...!
"d.o." obviates accessors

class AbsPager(object):
    def getMx(self): return 60

... 

class Cursepager(AbsPager):
    def getMx(self): return 24

# needs self.getMx() call
"d.o." is easy to individualize

# i.e. easy to make per-instance
class AbsPager(object):
    mx = 60
    def __init__(self, mx=0):
        self.cur = self.pg = 0
        self.mx = mx or self.mx
DP write-up components:

- name, context, problem
- forces, solution, (examples)
- results, (rationale), related DPs
- **known uses**: DPs are **discovered**, not **invented**!
The Template Method DP...

- emerges naturally in refactoring
  - much refactoring is "removal of duplication"
  - the TM DP allows removing structural duplication

- guideline: don't write a TM unless you're removing dups
def cmdloop(self):
    self.preloop()
    while True:
        s = self.doinput()
        s = self.precmd(s)
        f = self.docmd(s)
        f = self.postcmd(f, s)
        if f: break
    self.postloop()
# several template-methods e.g:

def handle_write_event(self):
    if not self.connected:
        self.handle_connect()
        self.connected = 1
    self.handle_write()
Variant: factor-out the hooks

- "organizing method" in a class
- "hook methods" in another
- KU: HTML formatter vs writer
- KU: SAX parser vs handler
- advantage: add one axis of variability (thus, flexibility)
Factored-out variant of TM

- shades towards the Strategy DP
- Strategy:
  - 1 abstract class per decision point
  - independent concrete classes
- Factored-out Template Method:
  - abstract/concrete classes grouped
Factored-out TM in Python [1]

class AbsParser(object):
    def setHandler(self, h):
        self.handler = h
    def orgMethod(self):
        self.handler.dothis()
        self.handler.dothat()
# ...optional...:
class AbsHandler(object):
    def dothis(self):
        pass  # or: raise NIE
    def dothat(self):
        pass  # or: raise NIE
Factored-out TM Python notes

- inheritance becomes optional
- so does existence of AbsHandler
- "organizing" flow doesn't have to be inside a method...
- merges into Python's intrinsic "signature-based polymorphism"
Pydiom: TM+introspection

- abstract base class can snoop into descendants at runtime
- find out what hook methods they have (naming conventions)
- dispatch appropriately (including "catch-all" / "error-handling")
def docmd(self, cmd, a):
    ...
    try:
        fn = getattr(self, 'do_' + cmd)
    except AttributeError:
        return self.default(cmd, a)
    return fn(a)
def finish_starttag(self, tag, ats):
    try:
        meth = getattr(self, 'start_' + tag)
    except AttributeError:
        [[ snip  snip ]]
        return 0
    else:
        self.tagstack.append(tag)
        self.handle_starttag(tag, meth, ats)
        return 1
Multiple TM variants weaved

- plain + factored + introspective
- multiple axes to carefully separate multiple variabilities
- Template Method DP equivalent of JS Bach's Kunst der Fuge's *Fuga a tre soggetti* ... ;-)}
class TestCase:
    def __call__(self, result=None):
        method = getattr(self, self.[...])
        try: self.setUp()
        except: result.addError([...])
        try: method()
        except self.failException, e:...
        try: self.tearDown()
        except: result.addError([...])
        ... result.addSuccess([...]) ...
Classic Factory DPs

- **Factory Method**: method that builds and return a new object
- **Abstract Factory**: abstract base class that supplies many related FM
- Each FM might also choose not to build, but rather return an already-existing, suitable object
Factory DPs advantages

- principle "program to an interface, not to an implementation" requires decoupling client from concrete class
- Abstract Factory ensures cohesion between multiple choices (shades of Template Method vs Strategy)
Factory Method as "just a hook"

- a FM can be seen as a hook-method (part of a TM DP)
- in this case the "creational" role is not emphasized
- generalizes to "object accessor" (need not build, just return a suitable object)
Factory DPs in Python

- types are Factory "methods"
- modules may be "abstract" factories (w/o inheritance):
  - os (concrete: posix, nt, ...)
  - DB API compliant modules
- strong connections between TM and Factory DPs
KU: type.__call__

```python
def __call__(cls,*a,**k):
    nu=cls.__new__(cls,*a,**k)
    if isinstance(nu, cls):
        cls.__init__(nu,*a,**k)
    return nu
```

(An example of "2-phase construction")
btw: the object.__new__ hook

In Python 2.2, quietly ignores args/kws:
```python
def __new__(cls,*a,**k):
    return ...
```

In Python 2.3, doesn't tolerate args/kws:
```python
def __new__(cls,*a,**k):
    if a or k: raise TypeError
    return ...
```
A Factory function example

```python
def load(pkg, obj):
    m=__import__(pkg, globals(), locals(), [obj])
    return getattr(m, obj)

# a typical use-case being:
cls=load('p1.p2.mod', 'c3')
```
Factory variant: factory-chain

module -> Connection
Connection -> Cursor
Cursor -> ResultSet
ResultSet -> ResultItem
ResultItem -> ...

KU: the DB API