The "Template Method" Design Pattern in Python

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, DP suitability for Python, "DP vs 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 [0]

- ...but but but...???
- why is there any connection?
- isn't programming-language choice *just* about implementation?
- and shouldn't all design precede all implementation...?!

and the answer is...: *** NO!!! ***

DPs and language choice [1]

- can **never** be independent
- design and implementation **must** interact (no waterfalls!)
- e.g.: in machine-code: "if", "while", "procedure" ... <u>are design-patterns</u>!
- HLLs embody these, so they are not design-patterns in HLLs

DPs and language choice [2]

- many DPs for Java/C++ are/have "workarounds for static typing"
- cfr Alpert, Brown, Woolf, "The DPs Smalltalk Companion" (AW)
- Pythonic patterns = classic ones, <u>minus</u> the WfST, <u>plus</u> optional exploits of Python's strengths

The "Template Method" DP

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

Classic Template Method DP

- abstract base class's "**organizing** method" calls "hook methods"
- concrete subclasses implement "hook methods"
- client code calls "organizing method" on concrete instances

Classic TM in Python

```
class AbsBase(object):
   def orgMethod(self):
      self.dothis()
      self.dothat()
class Concrete(AbsBase):
   def dothis(self): ...
```

Example: "pagination" of text

To "Paginate text", you must...:

- remember max number lines/page
- *output each line*, while tracking where you are on the page
- just before the first line in each page, *emit a page-header*
- just after the last line in each page, *emit a page-footer*

Ex: pager abstract class [0]

The abstract-pager will thus have:

- init: record max lines/page
- organizing method: "write a line"
- hook (abstract) methods:
  - emit header
  - emit line
  - emit footer

Ex: pager abstract class [1]

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

Ex: pager abstract class [2]

```
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:\n' % pg+1
  def dofoot(self,pg):
    print '\f', # form feed
```

Ex: concrete pager w/curses

```
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 <u>calls</u> hook methods on self, subclasses <u>supply</u> them
- it's "The Hollywood Principle":
  - "don't call us, we'll call you!"
- 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):
  # provide a default
  pass  # often a no-operation
 def dothat(self):
  # 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
```

When you write up a DP…:

…you provide several *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 lets you remove *structural* duplication
- guideline: don't write a TM unless you're removing duplication

KU: cmd.Cmd.cmdloop (simpl.)

```
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()
```

KU: asyncore.dispatcher

```
# 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
- (Pure) Strategy DP:
  - 1 abstract class per decision point
  - usually independent concrete classes
- (Factored) Template Method DP:
  - 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()
```

Factored-out TM in Python [2]

```
# ...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 <u>have</u> 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")

KU: cmd.Cmd.onecmd (simpl.)

```
def docmd(self,cmd,a):
 ...
  try:
    fn=getattr(self,'do_'+cmd)
  except AttributeError:
    return self.default(cmd,a)
  return fn(a)
```

KU: sgmllib ... (sample)

```
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* ... ;-)

but then, *all art aspires to the condition of music*

KU: unittest ... (simpl.)

```
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([...]) ...
```