Powerful Pythonic Patterns

http://www.aleax.it/oscon010_ppp.pdf

Google

Audience levels for this talk

Shu

("Retain")

Ha

("Detach")

Ri

("Transcend")

5' Q & A at the end

(+: let's chat later!)

What's a Pattern?

- identify a closely related <u>class of problems</u>
  - if there is no problem, why solve it?-)
- identify a <u>class of solutions</u> to the problems
  - closely related, just like the problems are
- may exist in any one of many different possible <u>scales</u> (or "phases of work")
  - just like the problems do

A Pattern's "problem(s)"

- each Pattern addresses a problem
  - rather, a closely related <u>class of problems</u>
- a problem is defined by:
  - "forces"
    - constraints, desiderata, side effects, ...
  - "context" (including: what technologies can be deployed to solve the problem)

A Pattern's "solution(s)"

- to describe a pattern, you must identify a <u>class of solutions</u> to the problems
  - meaningful <u>name</u> and summary
  - a "middling-abstraction" <u>description</u>
  - real-world <u>examples</u> (if any!-), "stars"
    - one-star == "0/1 existing examples"
  - <u>rationale</u>, "quality without a name"
  - how it balances forces / +'s & issues
  - <u>pointers</u> to related/alternative patterns

Is any field Pattern-less?

- if a field of endeavor is bereft of patterns,
  - either they haven't been looked for yet
    - i.e.: they exist, but aren't published
  - or else, that alleged "field" is merely a bunch of perfectly chaotic, indeed ergodic processes
    - in fact, not "a field" at all!-)

Why bother w/Patterns?

- identifying patterns helps practitioners of a field "up their game"...
- ...towards the practices of the very best ones in the field
  - precious in teaching, training, self-study
  - precious in concise communication, esp. in multi-disciplinary cooperating groups
  - also useful in enhancing productivity
    - to <u>recognize</u> is faster than to <u>invent</u>
    - structured description helps recognition

What's a DESIGN Pattern?

- we work in order to <u>deliver value to people</u>
- our work is a <u>connected mesh</u> of activities that fall in distinguishable, different areas
- <u>design</u> is one of the many areas of activity ("phases of work") into which we can classify our work
  - taxonomies are never perfect
  - but sometimes they can help a little;-)

Why do we work, at all?

- we work in order to <u>deliver value to people</u>
  - "make them feel more alive"
  - AKA "the Quality without a name"

Insert any Dilbert cartoon;-)

How do we work effectively?

- our work is a <u>connected mesh</u> of activities
  - <u>find</u> problems, opportunities, connections
  - <u>identify</u> system structure, details, forces
  - <u>invent</u> (or discover!) possible solutions
  - <u>experiment</u> (prototype) to evaluate them
  - <u>develop</u> (apply) solid implementations
  - <u>test</u>, <u>deploy</u> (deliver, distribute), <u>document</u>
- as for every taxonomy, lines are blurred
  - and even somewhat arbitrary...
  - ...but it can still help organize ourselves

"Design" is a vague term...

- most generically, it means "purpose"
- or specifically, "a plan towards a purpose"
- a geometrical or graphical arrangement
- an "arrangement" in a more abstract sense
  - ...
- in "Design Patterns", we mean "design" in the sense common to buildings-architecture and SW development:
  - work phase "between" study/analysis and "actual building" (not <u>temporally</u>;-)
- (SWers use "architecture" differently;-)

Other kinds of Patterns

- <u>Analysis</u>: find/identify value-opportunities
- <u>Architecture</u>: large-scale overall-system approaches to let subsystems cooperate
- <u>Human Experience</u>: focus on how a system presents itself and interacts with people
- <u>Testing</u>: how best to verify system quality
- <u>Cooperation</u>: how to help people work together productively to deliver value
- <u>Delivery/Deployment</u>: how to put the system in place (& adjust it iteratively)
- ...

What's a "Pythonic" Pattern?

- a Design Pattern arising in contexts where (part of) the technology in use is Python
- well-adapted to Python's strengths, if and when those strengths are useful
- dealing with Python-specific issues, if any
- e.g: http://www.aleax.it/oscon010_pydp.pdf

Pythonic Template Method

- "template" here means "self-delegation"
  - classically, via inheritance: base class has organizing-method, subclasses do hooks
- specifically-Pythonic aspects/variants:
  - overriding data (Queue, ...)
  - ABCs (or mix-ins) w/organizing-methods
  - "factored-out" hooks (via delegation)
  - organizing class can use runtime introspection to find hook-methods
  - all of the above (unittest.TestCase)

Dependency Injection as TM

- a form of "factored-out" TM (<u>and</u> a form of "Hollywood Principle" aka "Callback" DP)
  - → DPs are <u>not</u> a taxonomy!-)
- "inject" hooks (callables) as arguments (or settable attributes of organizing class)
  - works well with Factory, when the hooks' job is to build/return usable objects
- works best with first-class callables
  - in Python: functions, classes, bound methods, closures, callable instances, ...
  - ...wide variety → high applicability!

BTW: what's an "Idiom"?

- small-scale, technology-specific, common choice of name, arrangement, or procedure
- e.g.: "brick-overlap wall" (brick-specific)
  - pre-stressed concrete, wood, &c have somewhat-related but different idioms
- `if __name__ == '__main__': ...`
  - only makes sense in Python
- `while(*dest++ = *source++) {}`
  - only makes sense in C (or C++)
- `for(x=y.begin(); x!=y.end(); ++x)...`
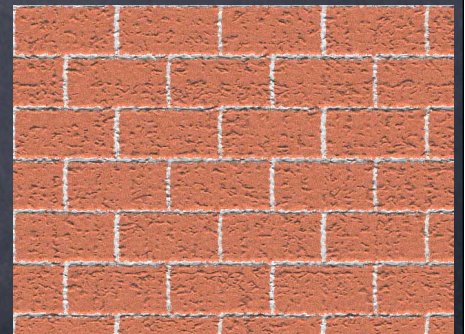
ANTI-Patterns (& Idioms)

- commonly-occurring, but counterproductive
- Waterfall, Analysis Paralysis, Moral Hazard, Groupthink, Abstraction Inversion, Fat Base, Copy&Paste, Backup Generator, Polling, ...
- Python-specific examples...:

```
def __init__(self, this, that):   # useless override
    super(Cls, self).__init__(this, that)
```

```
for string_piece in many_pieces: # += loop on str
    big_string += string_piece
```

```
sum(list_of_lists, [])                    # same (!) on list
```

+, most uses of `lambda`, **and** <u>any</u> **use of** `reduce`**!-)**

Pattern *Languages*

- think of each pattern as a <u>word</u>
- how are they combined in "discourse"?
  - "grammar", semantics, pragmatics
- hierarchical relationship among patterns of different scales / levels of abstractions
- "peer" relationship among "sibling" patterns

Hierarchical relationships

- different scales compose/decompose "into each other" (smaller-scale patterns often emerge in the context of larger-scale ones)
    - Plug In architecture pattern is helped by design patterns Template, Factory, DI, ...
    - simple Factory or Facade cases can use import/as idiom:

```
if ...: import posix as os
else: import nt as os
# then use os.this, os.that freely
```

"Peer" pattern cooperation

- patterns at the same scale work together
  - methodology-patterns CodeReviews, FanaticalTests, ContinuousBuild cooperate
  - Dependency Injection uses Callback to implement a variant of Template Method
    - and often uses Factory patterns too
  - Strategy and Memento used together let a Skeleton class delegate _both_ behavior _and_ state issues (!)

Q & A

http://www.aleax.it/oscon010_ppp.pdf

? !

Google