

Design

Week 5, day 2

Iterative design

- Aka or like: evolutionary design / spiral design / prototyping / agile programming
- Do work in phases, test, correct, improve / expand / move on
- Start anywhere but just start
- Start small and expand
- Release early, release often
- Test early, test often

Example: comparative phylogenetic analysis

- Final program must:
 - Read in data (phylogenies, taxa traits / data, species richness)
 - Allow selection of taxa & traits for comparative analysis
 - Show & save results of analysis
- What functions / modules are obvious?

Object oriented design

- Stay close to your problem domain
- Discover objects, don't invent them
- Objects are nouns, methods are verbs
- Good objects mean good modularity
- isA versus hasA
- Liskov substitution principle: a subclass must honour all the promises of its superclass

Patterns

- “Design Patterns” by the Gang of Four (Gamma, Helm, Johnson and Vlissides)
- A problem that occurs frequently in programming and a solution
- Usually not language specific
- Not algorithms or syntax but architecture / design

Example pattern: Borg

```
class Borg:
    """
    All instances need to access same data.
    """
    # shared by all
    shared_data = {}
    # prep in constructor
    def __init__(self):
        self.common_data = self.shared_data
        self.mine1 = "allmine"
        self.mine2 = "thistoo"
```

Example pattern: Observer

```
class Observable:
    "Something that can be observed."
    def addObserver (self, obs):
        self._clients.append (obs)

    def update (self):
        for item in self._clients:
            item.notify()

class Observer:
    "Needs to be informed of changes."
    def notify (self):
        [...]
```

Other examples

- Iterators
- Masquerade: “pretend to be” (e.g. file-like, sequence-like, User* subclasses)
- Adaptors: wrap / hide an object that may change or is incompatible (e.g. PIL's ImageDraw object)
- Facade: provides a (richer) interface to a variety of services (e.g. my XML Document wrapper)

Anti-patterns

- Sequential coupling: requiring calls to be in a specific order
- God objects: classes that do everything
- Coding by exception: developing by a series of special cases
- Action at a distance: objects need to know about things that aren't "close"

Applications vs. frameworks

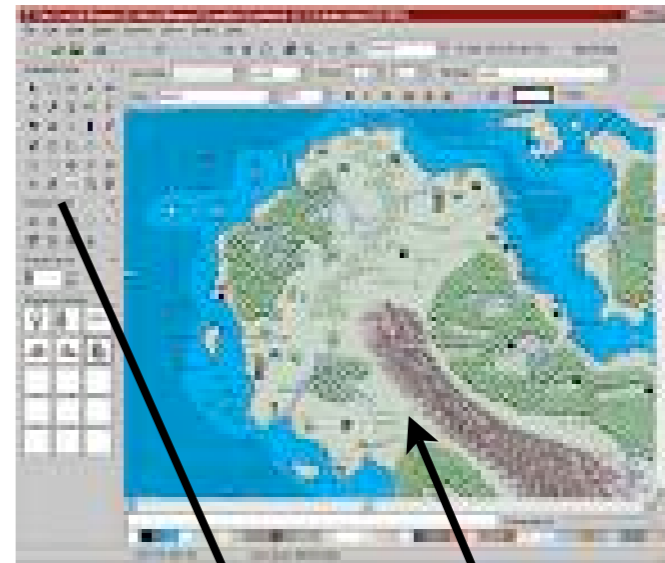
- What can be reused?
- Forms a logical module
- How can function be generalised?
- What is specific to the application?

Example: comparative phylogenetic analysis

- Final program must:
 - Read in data (phylogenies, taxa traits / data, species richness)
 - Allow selection of taxa & traits for comparative analysis
 - Show & save results of analysis
- What functions / modules are obvious?

Model-View-Controller

- MVC
- Model: data & functions that access & mutate
- View: presentation of data
- Controller: input



```
class Observable:
    "Something that can be
    observed."
    def addObserver (self,
    obs):
        self._clients.append
        (obs)

        def update (self):
            for item in
            self._clients:
                item.notify()

class Observer:
    "Needs to be informed
    of changes."
    def notify (self):
        [...]
```

Efficiency

- What's the most fastest way of doing this?
- Fast in, slow out. Slow in, fast out.
- Dictionaries are cheap
- Computational complexity
- Every function call has an overhead

Aphorisms

- Don't reinvent the wheel
- Test from the very beginning
- When in doubt, do anything
- Someone else has probably solved your problem for you already

Aphorisms 2

- Everything should always be in a valid state
- Distrust user input (or anything from the outside)
- Try to catch errors as soon as they occur
- Psuedocode

Resources

- Patterns in Python at <<http://www.suttoncourtenay.org.uk/duncan/accu/pythonpatterns.html>>
- Data Structures and Algorithms with Object-Oriented Design Patterns in Python at <<http://www.brpreiss.com/books/opus7/html/book.html>>
- Thinking in Python <<http://www.freetechbooks.com/about139.html>>

Resources 2

- Anti-Patterns at <<http://en.wikipedia.org/wiki/Anti-patterns>>
- Design patterns in Python at <<http://www.python.org/workshops/1997-10/proceedings/savikko.html>>