Programming Paradigms

Kilian Lieret[1,2]

Mentors:
Sebastien Ponce[3], Enric Tejedor[3]

[1]Ludwig-Maximilian University
[2]Excellence Cluster Origins
[3]CERN

28 September 2020

Overview

- Lecture 1: Programming Paradigms (PPs): Monday 14:15 – 15:25
- Lecture 2: Design Patterns (DPs): Tuesday 14:00 – 15:10
- Exercise consultation time: Thursday 17:00 – 17:30

All material at: github.com/klieret/icsc-paradigms-and-patterns

The goal of this course

- This course does not try to make you a better programmer
- But it **does** convey basic concepts and vocabulary to make your design decisions more **consciously**
- Thinking while coding + reflecting your decisions after coding ⟶ Experience ⟶ Great code!

Programming Paradigms

What **is** a programming paradigm?

- A classification of programming languages based on their features (but most popular languages support multiple paradigms)
- A programming style or way programming/thinking
- Example: Object Oriented Programming (thinking in terms of objects which contain data and code)
- Many common languages support (to some extent) multiple paradigms (C++, python, ...)

Why should I care?

- Discover new ways of thinking ⟶ challenge your current believes about how to code
- Choose the right paradigm for the right problem or pick the best of many worlds

Programming Paradigms

Some problems

- Too formal definitions can be hard to grasp and sometimes impractical, too loose definitions can be meaningless
- Comparing different paradigms requires experience and knowledge in both (if all you [know] is a hammer, everything looks like a nail)
- A perfect programmer might write great software using any PP

My personal approach

- Rather than asking "How to define paradigm X?", ask "How would I approach my problems in X?".
- Try out "academic languages" that enforce a certain paradigm
  ⟶ How does it *feel* to program in X
- Get back to your daily programming and rethink your design decisions

Good code: Objectives

Key objectives

- Testability: Make it easy to ensure the software is working correctly
- Maintainability: Make it easy to keep the software working (debugging, readability, ...)
- Extendibility: Make it easy to add new functionality
- Flexibility: Make it easy to adapt to new requirements
- Reusability: Make it easy to reuse code in other projects

⟶ How do I achieve all this?

Modularity

Perhaps the most important principle of good software

Split up code into parts, e.g. functions, classes, modules, packages, ...

You have done **well** if the parts are

- independent of each other
- have clear responsibilities

You have done **badly** if the parts

- are very dependent on each other (changes in one part require changes in many others)

This has benefits for almost all of your goals:

- Easier and more complete testability by using unit tests, better debugging
- Confidence from unit tests allows for better maintainability and flexibility
- Allowing to split responsibilities for different “modules” enhances collaboration and thereby maintainability
- Code reusability (obvious)

Modularity

Perhaps the most important principle of good software

A related principle: Isolate what changes!

- Which parts of your code will likely have to change in the future?
  - ⟶ These parts should be isolated (you should be able to change them in one place, without having to change anything else)
- This also leads to the concept of a separation of
  - interface (used by other "modules", stays untouched) and
  - implementation (only used by the module itself, can change easily)

Complex vs Complicated

From the Zen of python:

Simple is better than complex.
Complex is better than complicated.

- The more *complicated* something is, the harder it is to understand
- The more *complex* something is, the more parts it has

- Complicated problems might not have simple solutions
- But it is often still possible to modularize to have several simple components

- For example, using classes and objects will make your code more *complex*, but still easier to understand

OOP: Idea

- Before OOP: Two **separate** entities: *data* and *functions* (logic)
- Inspiration: In the real world, objects have a “state” (data) and “behaviors” (functions)

OOP

- **Think** in terms of *objects* that contain data and offer *methods* (functions that operate on objects) ⟶ Data and functions form a unit
- **Focus** on object structure rather than manipulation logic
- **Organize** your code in *classes* (blueprints for objects): Every object is *instance* of its class

A basic class in python

```python
class Rectangle:
    def __init__(self, width, height):  # <-- constructor
        # 'self' represents the instance of the class
        self.width = width              # <-- attribute = internal variable
        self.height = height

    def calculate_area(self):           # <-- method (function of class)
        return self.width * self.height


r1 = Rectangle(1, 2)         # <-- object (instance of the class)
print(r1.calculate_area())   # <-- call method of object
print(r1.width)              # <-- get attribute of object
r1.width = 5                 # <-- set attribute of object
```

Encapsulation and data hiding

- Do not expose object internals that may change in the future ⟶ Make certain attributes and methods private (*data hiding*)
- Rephrased: Separate interface (won't be touched because it's used by others) from implementation (might change)
- In some languages this is "enforced" (e.g. using the `private` keyword), in others it is denoted by naming conventions (e.g. leading underscore)

Subclasses and Inheritance

Subclasses are specializations of a class

- inherit attributes/methods of their superclass
- can introduce new attributes/methods
- can override methods of superclass

```python
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, I'm {self.name}")


class Child(Person):
    def __init__(self, name, school):
        super().__init__(name)
        self.school = school

    def learn(self):
        print(f"I'm learning a lot at {self.school}")


c1 = Child("john", "iCSC20")
c1.greet()
c1.learn()
```

Abstract methods

- An *abstract method* is a method that has to be implemented by a subclass
- An *abstract class* (*abstract type*) is a class that cannot be instantiated directly but it might have *concrete subclasses* that can
- Use abstract classes to enforce interfaces for the concrete classes

```python
from abc import ABC, abstractmethod


class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

    @abstractmethod
    def draw(self):
        pass


class Rectangle(Shape):
    def __init__(self, ...):
        ...

    def calculate_area(self):
        # concrete implementation here
```

Strenghts and Weaknesses

Strengths

- Easy to read and understand if done well (very natural way of thinking if classes model real world objects)
- Natural way to structure large projects (e.g. taking classes as components)
- Very wide spread way of thinking
- Especially applicable to problems that center around data and bookkeeping with logic that is strongly tied to the data

Weaknesses

- Wrong abstractions can lead to less code reusability
- Lasagna code: Too many layers of classes can be hard to understand
- Can be hard to parallelize if many entangled and interdependent classes with shared mutable states are involved ($\longrightarrow$ if required, should be design requirement from the start; *parallel patterns* address some difficulties)

Functional programming

Functional programming

- expresses its computations in the style of mathematical functions
- emphasizes
    - expressions ("is" something: a series of identifiers, literals and operators that reduces to a value)

  over
    - statements ("does" something, e.g. stores value, etc.)

  ⟶ declarative nature
- Data is immutable (instead of changing properties, I need to create copies with the changed property)
- Avoids side effects (expressions should not change or depend on any external state)

Examples

Languages made for FP (picture book examples):

- `Lisp` and derivatives: `Common Lisp`, `Clojure`, ...
- `Haskell`
- `OCaml`
- `F#`
- `Wolfram Language` (Mathematica etc.)
- ...

With emphasis on FP:

- `JavaScript`
- `R`
- ...

Not designed for, but offering strong support for FP:

- `C++` (from `C++11` on)
- `Perl`
- `Python` (?) (⟶ You might also want to check out the `coconut` language)
- ...

Pure functions

A function is called pure if

1. Same arguments ⟹ same return value $(x = y \implies f(x) = f(y))$
2. The evaluation has no side effects (no change in non-local variables, ...)

Which of the following functions are pure?

```python
def f1(x):
    return x**2


def f2(x):
    print(x)
    return x**2


global y = 0


def f3(x):
    y += 1
    return x + y
```

```python
def f4():
    return int(input()) + 1


def f5(lst: List):
    lst[0] = 3
    return lst
```

Answer: `f1` is pure; `f2`, `f3`, `f5` violate rule 2; `f4`, `f3` violate rule 1.

Non strict evaluation

- Some functional programming languages use *non-strict evaluation*: The arguments of a function are *only* evaluated once the function is called.

  Example: `print(sqrt(sin(a**2)))`

  In a strict language (e.g. `Python`, `C++`), we evaluate inside out:

  $$a \longmapsto a^2 \longmapsto \sin a^2 \longmapsto \sqrt{\sin a^2}$$

  In a non-strict language, the evaluation of the inner part is **deferred**, until it is actually needed.

- But `Python` actually has something similar in the concept of `generators`:

```
%time a = range(int(1e8))
>>> Wall time: 7.63 µs

%time b = list(a)
>>> Wall time: 2.33 s
```

- This allows for infinite data structures (which can be more practical than it sounds)

Memoization

- Non strict evaluation together with *sharing* (avoid repeated evaluation of the same expression) is called *lazy evaluation*
- Generally, functional programming can get cheap performance boosts by very simple *memoization*: Storing the results of expensive **pure** function calls in a cache

```
import time
from functools import lru_cache


@lru_cache()
def expensive(x):
    time.sleep(1)
    return x+42


%time expensive(2)
>>> Wall time: 1 s

% time expensive(2)
>>> Wall time: 6.2 µs
```

Higher order functions

A *higher order function* does one of the following:

- returns a function
- takes a function as an argument

Opposite: *first-order function*.

Mathematical examples (usually called *operators* or *functionals*): differential operator, integration, ...

Higher level functions are the FP answer to template methods in OOP ("configuring" object behavior by overriding methods in subclasses).

Classic example of a higher order function: `map` (applies function to all elements in list):

```
def map(function, iterator):
    """ Our own version of map (returns a list rather than a generator) """
    return [function(item) for item in iterator]


map(lambda x: x**2, [1, 2, 3])
>>> [1, 4, 9]
```

Higher order functions II

A function that also returns a function:

```python
def get_map_function(function):
    """ Takes a function f and returns the function map(f, *) """
    def _map_function(iterator):
        return map(function, iterator)

    return _map_function


mf1 = get_map_function(lambda x: x**2)
mf2 = get_map_function(lambda x: x+1)

mf2(mf1([1, 2, 3]))
>>> [2, 5, 10]
```

Type systems

Types:

- In OOP, type and class are often used interchangeably (e.g. `"abc"` is of type *string* = is an instance of the `str` class)
- In FP we talk about *type*s
- Complex types can be built from built in types (e.g. `List[Tuple[str, int]]`, we can also use `structs`)
- In many languages, types of variables, arguments, etc. have to be declared (e.g. `def len(List[float]) -> int`)
- Real FP languages usually have very powerful type systems

Polymorphism

In FP, the type system allows to bring back some OOP thinking but is more flexible. Usually you can do some of the following:

Single/multiple dispatch/ad hoc polymorphism:

- Can overload function definitions (e.g. define `def print(i: int)` differently from `def print(string: str)`)
- The right function is resolved based on the type at compile- or runtime

Parametric polymorphism:

- Parameterize types in function signatures (e.g. `def first(List[a]) -> a`; `a` represents an arbitrary type)

Type classes:

- Define a "type" by what functions it has to support (e.g. define `Duck` as anything that allows me to call the `quack` function on it)
- Similar to a class with only abstract methods (=*interface*) and no encapsulated data

Looping in functional programming

Let's consider a function that calculates $\sum_{i=0}^{N} i^2$:

```
def sum_squares_to(n):
    result = 0
    for i in range(n+1):
        result += i^2
    return result
```

This is a function, but does not follow the FP paradigm:

- More statements (assignments, loops, ...) than expressions
- The for loop segment is not free of side effects (value of `result` changes)
- Repeated reassignments of `result` are frowned upon (or impossible)

How to change this? ⟶ Use recursion

```
def sum_squares_to(n):
    return 0 if n == 0 else n^2 + sum_squares_to(n-1)
```

Looping in functional programming

The previous example is called a head recursion (recursion before computation); using a tail recursion (recursion after computation) is preferable due to better compiler optimization:

```
def sum_squares_to(n, partial_sum=0):
    return partial_sum if n == 0 else sum_squares_to(n-1, partial_sum + n^2)
```

Another FP way is to use the higher level functions `map` and `reduce` together with anonymous functions (lambda):

```
from functools import map, reduce


def sum_squares_to(n):
    return reduce(
        lambda x, y: x+y,
        map(lambda x: x**2, range(n+1))
    )
```

This also opens the door for concurrency (⟶ parallel versions of `map` and `reduce`)

Strengths and Weaknesses

Strengths

- Proving things mathematically (referential transparency, ...)
- Testability (no object initializations and complex dependencies, pure functions)
- Easy debugging (no hidden states)
- Can be very short and concise ⟶ easy to verify
- Sophisticated logical abstractions (using high level functions) ⟶ modularity, code reuse
- Easy parallelization (no (shared) mutable states)

Strengths and Weaknesses

Weaknesses

- Structuring code in terms of objects can feel more intuitive if logic (methods) are strongly tied to data
- Imperative algorithms might be easier to read and feel more natural than declarative notation
- FP might have a steeper learning curve (e.g. recursions instead of loops, ...)
- Performance issues: Immutable data types and recursion can lead to performance problems (speed and RAM), whereas many mutable data structures are very performant on modern hardware
- Pure FP has still only a small user base outside of academia, but FP support more and more wide spread in common languages

Object oriented vs functional programming

Some key aspects to keep in mind:

- FP $\neq$ OOP − classes
- FP is **not** the opposite of OOP: Both paradigms take opposite stances in several aspects: declarative vs imperative, mutable vs immutable, ... $\Longrightarrow$ Not everything can be classified into one of these categories
- Rather: Two different ways to think and to approach problems ⟶ see caveats at the beginning

In a multi-paradigm language, you can use the best of both worlds!

- OOP has its classical use cases where there is strong coupling between data and methods and the bookkeeping is in the focus (especially of ”real-world” objects)
- FP instead focuses on algorithms and *doing* things
- Some people advocate “OOP in the large, FP in the small” (using OOP as the high level interface, using FP techniques for implementing the logic)

For example:

- Many complicated class structures implementing manipulations can be made more flexible with a system of high level functions, anonymous functions etc. (`pandas.DataFrame.apply`)

Declarative vs imperative programming

Declarative programming:

- Program describes logic rather than control flow
- Program describes “what” rather than “how”
- Aims for correspondence with mathematical logic
- FP is usually considered a subcategory

Opposite: imperative programming:

- Algorithms as a sequence of steps
- Often used synonymously: procedural programming (emphasizing the concept of using procedure calls (functions) to structure the program in a modular fashion)
- OOP is usually considered a subcategory

Examples

"Pure" declarative languages:

- `SQL` (Structured Query Language – language to interact with databases):

```
SELECT * FROM Customers WHERE Country='Mexico';
```

- Markup languages, like `HTML`, `CSS` (Cascading Style Sheets – language to describe styling of e.g. HTML pages), ...

```
<h1 style="color:blue;">This is a Blue Heading</h1>
```

- Functional programming languages like `Haskell` (even though they allow some "encapsulated" imperative parts)
- ...

Powerful backends I

Idea:

- Split up your code into application/analysis specific code (describing the problem) and a backend/library (implementing solution strategies)
- The application specific code starts to *feel* very declarative
- The backend can use different strategies depending on the nature/scale of the problem

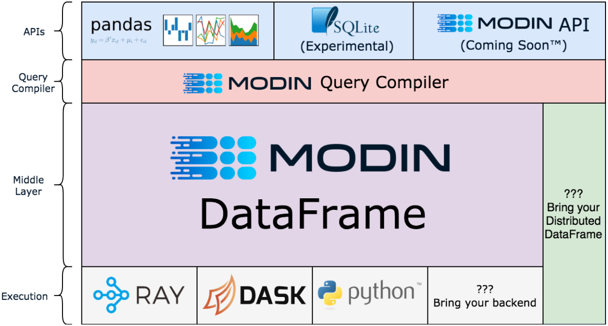
Powerful backends II

Example:

```python
# "Chi2 distance" using plain python
def chi2(data, theory, error):
    err_sum = 0
    for i in range(len(data)):
        if data[i] == theory[i] and error[i] == 0:
            continue
        err_sum += (data[i] - theory[i])**2 / (error[i]**2)
    return err_sum


# Using DataFrames: Table contains columns experiment, theory, error

# ROOT RDataFrame example:
chi2 = ROOT.ROOT.RDataFrame(...)  # initialize
        .Filter("!(data==theory & error==0.)")  # filter rows
        .Define("sqd", "pow(data-theory, 2) / pow(error, 2)")  # new col
        .Sum("sqd").GetValue()  # sum it up

# Pandas example:
chi2 = pd.DataFrame(...)  # initialize
_df = df.query("~(data==theory & error==0)")  # filter
chi2 = (_df["data"] - _df["theory"]).pow(2) / _df["error"].pow(2)).sum()
```

Powerful backends III

- We might want even more of our backend, e.g. delayed or distributed execution
- pandas can also be viewed as a "declarative language" describing the problem ⟶ have a more sophisticated backend handle all operations ⟶ `modin pandas`



Powerful backends IV

Belle II steering file:

```python
path = create_path()

# Load data
inputMdstList("default", "/path/to/input/file", path=path)

# Get final state particles

# Fill 'pi+:loose' particle list with all particles that have pion ID > 0.01:
fillParticleList("pi+:loose", "piid > 0.01", path=path)
# Fill 'mu+:loose' particle list with all particles that have muon ID > 0.01:
fillParticleList("mu+:loose", "piid > 0.01", path=path)

# Reconstruct decay
# Fill 'K_S0:pipi' particle list with combinations of our pions and muons
reconstructDecay(
    "K_S0:pipi -> pi+:loose pi-:loose", "0.4 < M < 0.6", path=path
)
```

Powerful backends V

Many more high level tools available:

- LINQtoROOT: Uses `C#` with `LINQ` (`SQL` like) queries to describe problem

```
events
    .Select(e => e.Data.eventWeight)
    .FuturePlot("event_weights", "Sample EventWeights",100, 0.0, 1000.0)
    .Save(hdir);
```

- The FAST HEP toolkit: Uses yaml config files to describe problem; using pandas, numpy, etc. in the backend

```
stages:
  - BasicVars: Define
  - DiMuons: cms_hep_tutorial.DiObjectMass
  - NumberMuons: fast_carpenter.BinnedDataframe
  - EventSelection: CutFlow
  - DiMuonMass: BinnedDataframe
```

- Many more...

Other paradigms

- Logic programming (LP) (subset of declarative programming): Automatic reasoning by applying inference rules
  - LP languages: `Prolog`, `Datalog`
  - LP can be made available with libraries, e.g. for `Python`: `Pyke` (inspired by `prolog`), `pyDatalog` (inspired by `Datalog`)
  - Example:

```
% X, Y are siblings if they share a parent
sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

% Father, mother implies parent
parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).

% Introduce some people
father_child(tom, sally).
father_child(tom, erica).

% Ask:
?- sibling(sally, erica).
Yes
```

- Symbolic programming
- Differentiable programming

Outlook

Next lecture: Software design patterns

- Focus on OOP
- Introduce some “golden rules” of OOP
- Patterns: Reusable solutions to common problems

Discussion on mattermost:
mattermost.web.cern.ch/csc/channels/programming-paradigms

Get the exercises at github.com/klieret/icsc-paradigms-and-patterns