Software Design Patterns

Kilian Lieret$^{1,2}$

Mentors:
Sebastien Ponce$^3$, Enric Tejedor$^3$

$^1$Ludwig-Maximilian University
$^2$Excellence Cluster Origins
$^3$CERN

27 March 2019
1. More on OOP
   - Class Diagrams
   - The SOLID rules of OOP

2. Patterns
   - Patterns
   - Creational Patterns
   - Structural Patterns
   - Behavioral Patterns
   - Concurrency Patterns
   - Antipatterns

3. Discussion

Slides + material available at
github.com/klieret/icsc-paradigms-and-patterns
Repetition: Object Oriented Programming

- **Inheritance:** Subclasses inherit all (public and protected) attributes and methods of the base class

- Methods and attributes can be public (anyone has access), private (only the class itself has access) or protected (only the class and its subclasses have access)

- **Abstract methods** of an abstract class are methods that have to be implemented by a subclass (concrete class)

**New:** class methods: Methods of the class, rather than its instances

```python
class Class:
    def method(self):
        # needs to be called from INSTANCE and can access instance attributes
        @classmethod
        def classmethod(cls):
            # no access to instance attributes

    # This won't work:
    Class.method()  # <-- needs an instance, e.g. Class(...).method()

    # But this does:
    Class.classmethod()
```
Class diagrams I

UML (Unified Markup Language) class diagrams visualize classes and the relationships between them. We will use the following subset of notations:

<table>
<thead>
<tr>
<th>Class A</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ public variable : type</td>
</tr>
<tr>
<td>− private variable : type</td>
</tr>
<tr>
<td># protected variable : type</td>
</tr>
<tr>
<td>+ public method(arg : type) : return type</td>
</tr>
</tbody>
</table>

Subclasses:

- class B: subclass of A
- class C: subclass of B
- class D: A contains instances of D
- class E: A is otherwise associated with E
1 More on OOP
   - Class Diagrams
   - The SOLID rules of OOP

2 Patterns
   - Patterns
   - Creational Patterns
   - Structural Patterns
   - Behavioral Patterns
   - Concurrency Patterns
   - Antipatterns

3 Discussion
The SOLID rules of OOP: Single responsibility principle

**SOLID**

Commonly (mis-)quoted as:

*A class should only have one responsibility.*

More accurate:

*A class should only have one reason to change.*

Better still:

*Gather together the things that change for the same reasons.*
*Separate those things that change for different reasons.*

So this actually proposes a **balance**!

- Avoid classes that do too much ("god class")
- But also avoid having changes always affect many classes ("shotgun surgery")
The SOLID rules of OOP: Open Closed Principle

You should be able to **extend the behavior of a system** without having to **modify that system**.

- Writing a library, **modifying** functionality means that all users have to be informed (**not backwards compatible**) $\rightarrow$ Avoid!
- In your own code: Modifying one functionality (**also by overriding methods of the super class, etc.**) **poses the danger of breaking other parts** (**though tests can help with that**)
- Extending code by providing additional methods, attributes, etc. does not have this danger $\rightarrow$ preferred!
- Requires thinking ahead: What parts have to be flexible, what remains constant?
- Again a **balance** is required:
  - Be **too generic** (**avoid modifications**) and your code won't do anything
  - Be **too concrete** and you will need to modify (**and potentially break things**) often
The SOLID rules of OOP: Liskov Substitution Principle

**SOLID**

If S is a **subtype (subclass)** of T, then objects of type T can be **replaced** with objects of type S without breaking anything

This can be expanded to a series of properties that should be fulfilled:

- **Signature** of methods of the subclass:
  - Required type of arguments should be supertype (**contravariance**)
    Violation: Supermethod accepts any Rectangle, submethod only Square
  - Return type of method should be a subtype (**covariance**)
    Violation: Supermethod returns Square, submethod returns Rectangle

- **Behavior**: 
  - **Preconditions** (requirements to be fulfilled before calling method) cannot be strengthened in the subtype
    Violation: Only in subclass prepare() must be called before method()
  - **Postconditions** (conditions fulfilled after calling a method) cannot be weakened by the subtype
  - **Invariants** (properties that stay the same) of supertype must be preserved in the subtype
  - **History constraint**: Subtypes cannot modify properties that are not modifyable in supertype
    Violation: VariableRadiusCircle as subtype of FixedRadiusCircle
The SOLID rules of OOP: Interface segregation principle (ISP)

Clients should not be forced to depend on methods they do not use

- “Thin” interfaces offering a reasonably small number of methods with high cohesion (serve similar purposes; belong logically together) are preferred over “fat” interfaces offering a large number of methods with low cohesion.
- Sometimes we should therefore split up (segregate) fat interfaces into thinner role interfaces.
- This leads to a more decoupled system that is easier to maintain.
- Example: Even if all data is contained in one (e.g. SQL) database, the ISP asks to write different interfaces to do different things, e.g. have a CustomerDb, OrderDb, StoreDb, ...
The SOLID rules of OOP: Dependency Inversion Principle

This is about **decoupling** different classes and modules:

1. *High-level modules should not depend on low-level modules.*
   Both should depend on abstractions (*interfaces*).

Let’s consider a very simple example: A **button** controlling a lamp. One way to implement this:

This violates the DIP, because **Button** (high-level) depends on **Lamp** (detail).

What if we have multiple consumers (**Motor**, **Lamp**, ...) and multiple types of buttons (**swipe button**, **switch**, **push button**, ...)? How can we force them to behave consistent? What methods does a consumer have to implement to work together with the button?

→ Enter abstractions (*interfaces*)
The SOLID rules of OOP: Dependency Inversion Principle

SOLID

AbstractButton
- client
  + turn_on()
  + turn_off()
  + flip()

ConcreteButton

AbstractClient

ConcreteClient

Now it’s clear which methods the concrete client has to implement. Both high level and low level modules only depend on abstractions.

This also fulfills the second part of the DIP:

2. **Abstractions should not depend on details. Details (i.e. concrete implementations) should depend on abstractions.**
Performance considerations

Some patterns will advocate:

- Classes that only act as interfaces and pass on calls to other (worker) classes
- Using separate classes to facilitate communication between classes
- Accessing attributes (only) through methods
- Prefer composition over inheritance

However, when writing performance critical (C++, ...) code, you should avoid unnecessary “detours”:

- Avoid unnecessary interfaces
- Consider inlining simple, often-called functions (e.g. getters and setters)
- Inheritance > composition > if statements

Modern compilers will try to apply some optimization techniques automatically (automatic inlining, return value optimization, ...)

General rule: Profile before Optimizing
More on OOP
- Class Diagrams
- The SOLID rules of OOP

Patterns
- Patterns
- Creational Patterns
- Structural Patterns
- Behavioral Patterns
- Concurrency Patterns
- Antipatterns

Discussion
Patterns

Software design patterns try to offer general and reusable solutions for commonly occurring problems in a given context.

Commonly categorized as:

- **Creational patterns**: How are instances of classes instantiated? (What if I have a class that can create instances in different ways?)
- **Structural patterns**: Concerned with relationships between classes. (How can classes form flexible larger structures?)
- **Behavioral patterns**: Concerned with algorithms and communication between classes. (How are responsibilities assigned between classes?)
- **Parallel patterns**: Parallel processing and OOP → only mentioned briefly
1 More on OOP
   - Class Diagrams
   - The SOLID rules of OOP

2 Patterns
   - Patterns
   - Creational Patterns
   - Structural Patterns
   - Behavioral Patterns
   - Concurrency Patterns
   - Antipatterns

3 Discussion
Factory method

If there are multiple ways to instantiate objects of your class, use factory methods rather than adding too much logic to the default constructor.

Bad:

```python
class Uncertainty:
    def __init__(self, absolute_errors=None, relative_error=None, data=None, config=None, ...):
        if config is not None:
            # load from config
        elif absolute_errors is not None:
            # add absolute errors
        elif relative_errors is not None and data is not None:
            # add relative errors
        ...

instance = Uncertainty(config="path/to/my/config")
```
Factory method

*If there are multiple ways to instantiate objects of your class, use *factory methods* rather than adding too much logic to the default constructor.*

**Good:**

```python
class Uncertainty:
    def __init__(self, absolute_errors):
        # construct from absolute errors

    @classmethod
    # <-- doesn't need instance to be called (cf. first slide)
    def from_config(cls, config):
        # get absolute errors from config file
        return cls(absolute_errors)

    @classmethod
    from relative_errors(cls, data, relative_errors):
        return cls(data * relative_errors)

instance = Uncertainty.from_config("path/to/my/config")
```

Alternatively, you can also have subclasses that provide (implementations to) factory methods.
Builder Pattern

If you build a very complex class, try to instantiate (build) it in several steps.

Bad:

```python
class Data:
    def __init__(
        data: array,
        data_error: array
        mc_components: List[array],
        mc_errors: List[array],
        mc_float_normalization: List[bool],
        mc_color: List[string],
        ...
    )

    def fit(...)

    def plot(...)  
```

You will probably consider different fits and plots; violates Single Responsibility Principle  →  Rather have Fit and Plot classes
Builder Pattern

*If you build a very complex class, try to instantiate (build) it in several steps.*

**Bad:**

```python
class Data:
    def __init__(
        data: array,
        data_error: array
        mc_components: List[array],
        mc_errors: List[array],
        mc_float_normalization: List[bool],
        mc_color: List[string],
        ...
    )
```

Kilian Lieret
Software Design Patterns
Builder Pattern

*If you build a very complex class, try to instantiate (build) it in several steps.*

**Better:**

```python
class Data:
    def __init__(data: array, data_error: array)
        pass

    def add_mc_component(data, errors, floating=False, color="black", ...):
        pass

data = Data(...)
data.add_mc_component(...)  ...
data.add_mc_component(....)
```

- What if we have multiple ways to build of the object?
- Do I want to have the add_mc_component method after I start using the data?
  ➞ Have a separate Data and Builder hierarchy.
**Builder Pattern**

*If you build a very complex class, try to instantiate (build) it in several steps.*

**Best:**

```python
class Data:
    pass

class Builder:
    def __init__(...) 
    def add_mc_component(...) 
    def create(...) -> Data 

builder = Builder(...) 
builder.add_mc_component(...) 
... 
builder.add_mc_component(...) 
data = builder.create()
```

And of course I could now create AbstractData and AbstractBuilder etc.
1 More on OOP
   - Class Diagrams
   - The SOLID rules of OOP

2 Patterns
   - Patterns
   - Creational Patterns
   - Structural Patterns
   - Behavioral Patterns
   - Concurrency Patterns
   - Antipatterns

3 Discussion
Proxy, Adapter, Facade

Three patterns that deal with interfaces:

- **Proxy**: Given a *servant* class (doing the actual work), create a new *proxy* class with the same interface in order to *inject code*. The client can then use the Proxy instead of using the Service class directly.

Usage examples:

- **Protection proxy**: Enforce access rights (always check authorization before method call/attribute access; e.g., in web applications)
- **Remote proxy**: If the Service is located remotely, the proxy deals with transferring requests and results
- Extend the Service class with caching or logging
- ...
Adapter

- **Facade**: A class providing a simple interface for complicated operations that involve multiple servant classes

- **Adapter**: We have a 3rd party class `ForeignClass` whose interface is incompatible to interface `OurInterface` → Create an *adapter* class as a wrapper

Usage case example: We want to switch between different machine learning models (strategy pattern → later). Our models have a `train()` method, models from a foreign library have a `training()` method → create adapter(s) for library
Adapter

```python
class OurMLModel(ABC):
    """ Our interface """
    @abstractmethod
    def train(...):
        pass

class TheirMLModel(ABC):
    """ Their interface """
    @abstractmethod
    def training(...):  # <-- this method should be called train
        pass

class ModelAdapter(OurMLModel):  # <-- implements our interface
    def __init__(self, model: TheirMLModel):
        self._model = model  # <-- our adapter holds the foreign model

    def train(...):  # <-- and defines a different interface for it
        self._model.training(...)

# Their model with our interface:
model = ModelAdapter(TheirMLModel(...))
```
1 More on OOP
   - Class Diagrams
   - The SOLID rules of OOP

2 Patterns
   - Patterns
   - Creational Patterns
   - Structural Patterns
   - Behavioral Patterns
   - Concurrency Patterns
   - Antipatterns

3 Discussion
Template Method

- **Use case:** Several different classes that only contain minor differences in few places
- **Suggestion:** Put shared code in superclass, have subclasses implement or override specific methods

```
MLModel
+ preprocess()
+ train()
+ validate()
```

```
BDT
+ train()
```

```
RandomForest
+ train()
```

- **Advantages:** Simple and clean with little overhead
- **Warnings:**
  - If there are many differences between original classes, we need (to override) many methods → increasingly hard to read and maintain
  - If there are multiple “options” for every method and we want to realize them, the number of subclasses grows exponentially → **Strategy pattern**
  - If overriding default methods, the Liskov Substitution Principle can be easily violated
Template Method

**Questionable:**

```python
class MLModel:
    def load_data(...)
    def prepare_features(...)

    def train(...):
        if self.model == "BDT":
            # train BDT
        elif self.model == "RandomForest":
            # train random forest
        elif ...

    def validate(...)
```

- What if multiple methods depend on the model?  →  Need to keep track of more ifs everywhere
- What if we want to add or remove a model?  →  Need to make changes in many places →  Open/Closed Principle, “divergent change”
- Depend on all implementations →  Dependency Inversion Principle
Template Method

Better:

class MLModel(ABC):  # <-- abstract class
    def load_data(...)
    def prepare_features(...)

        @abstractmethod
        def train(...):
            pass

    def validate(...)

class BDT(MLModel):  # <-- concrete class
    def train(...):
        # Implementation

class RandomForest(MLModel):
    def train(...):
        # Implementation
Strategy

- **Usage:** You have a problem that can be solved with different algorithms (strategies)
- **Suggestion:** Create abstract class for algorithm and concrete subclasses with specific implementation; original class holds instance of algorithm class

```
+ ml_model: MLModel
+ fitter: Fitter
+ train()
+ fit()
```

```python
class MyAnalysis:
    def __init__(self, ml_model, fitter):
        self.ml_model = ml_model
        self.fitter = fitter

    def train(self):
        # Train the model
        self.ml_model.train()
        self.fitter.fit()

    def fit(self):
        # Fit the model
        self.ml_model.fit()
```

```
class MLModel:
    def train(self):
        # Train the model

    def fit(self):
        # Fit the model

class Fitter:
    def fit(self):
        # Fit the model

    def train(self):
        # Train the model
```

```
class KernelDensityFit(Fitter):
    def fit(self):
        # Fit the kernel density

class CrystalBallFit(Fitter):
    def fit(self):
        # Fit the crystal ball

class BDT(MLModel):
    def train(self):
        # Train the BDT

    def fit(self):
        # Fit the BDT

class RandomForest(MLModel):
    def train(self):
        # Train the random forest

    def fit(self):
        # Fit the random forest
```
class MyAnalysis():
    def __init__(ml_model: MLModel, fitter: Fitter)
        self.ml_model = ml_model
        self.fitter = fitter

    def fit(...):
        self.fitter.fit(...)

    def train(...):
        self.ml_model.train(...)

class MLModel(ABC):
    @abstractmethod
    def train(...)

class RandomForest(MLModel):
    def train(...):
        # Implementation

my_analysis = MyAnalysis(RandomForest(...), KernelDensityEstimator(...))
my_analysis.train(...)
my_analysis.fit(...)
Note: The main class holds instances of algorithm classes; the algorithm classes use the template method pattern.

Advantages:
- Open/Closed principle: Easily add new strategies
- Dependencies inverted (MyAnalysis does not depend on the individual implementations)
- Small number of subclasses
- Separated implementation of algorithms from higher level code
- For compiled languages: Change algorithms at runtime

Warnings:
- Might be overkill for very simple problems
- For maximum performance, avoid virtual calls

Alternatives:
- If your language supports it: Use functions instead of objects (e.g. provide several fit() functions and pass them to the class)
- Use template pattern if there is only one strategy that can be replaced
Command

*The command pattern turns a method call into a standalone object.*

Rather than directly calling a method, the interface creates a Command object (describing what we want to execute) and passing it on to a Receiver that executes it.

Use cases:

- **Decouple** user interfaces from the backend (by using Command objects as means of communication)
- **Build up** a command history with undo functionality
- **Remote execution** of commands
- **Queue** or **schedule** operations

HEP specific use case example (Belle II software framework):

- Build up analysis by adding modules (Command objects) to a path (list of modules), each implementing a `event()` method to process one event
- After all modules are added, process the path: Loop over all events, calling the `event()` method of all modules in order
Slightly simplified Belle II steering file:

```python
# Create path to add modules (=Command objects) to
path = create_path()

# Load data (convenience function that adds a "DataLoader" module to the path)
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
)

# Process path = call execute() on all Command objects
process(my_path)
```
Command

Upon processing path:

- Strictly **declarative** approach (no for loops or implementation details)
- Modules can be implemented in **python** or **C++**
- “**Building block**” approach makes steering files extremely easy to write and understand (even browser based graphical interface for highschoolers: try it at masterclass.ijs.si)
Visitor

- **Concrete example**: Serialize a collection of instances of different classes (e.g. provide JSON export for a list of different data objects)
- **Possible implementation**: Provide a `to_json()` method to all classes
- **Potential issue**: Might soon want to add export for export possibilities: (XML, CSV, YAML, etc.)
  - more and more unrelated methods need to be added to the data class
  - “polluted” interface (methods are irrelevant for core functionality); people might be wary of frequent changes to a well working class
- **Solution**: Separate algorithms from the objects they operate on
Visitor

Use case (more formally):
- Given a heterogeneous family of Element classes
- Do not expect significant changes to Element classes
- Various unrelated operations need to be performed on a collection of Element objects
- We expect frequent additions and changes for the operations
- Do not want to frequently change Element classes because of that

Advantages of visitor pattern:
- Single responsibility principle: All the operation functionality is in one place
- Open/Closed principle: Easy to add new operations

Disadvantages of the visitor pattern:
- No access to private information of Element classes
- Changes to Element classes can require changes to all visitors
1 More on OOP
- Class Diagrams
- The SOLID rules of OOP

2 Patterns
- Patterns
- Creational Patterns
- Structural Patterns
- Behavioral Patterns
- Concurrency Patterns
- Antipatterns

3 Discussion
Concurrency Patterns

Concurrency patterns need their own lecture, so this will only quickly mention basic concepts.

Use cases:

- Manage/synchronize access to shared resources (e.g. to avoid race conditions when several threads perform read and write operations)
- Scheduling tasks in parallel
- (A)synchronous event handling
Concurrency Patterns II

Advanced example: Active object pattern

- Want to decouple method calling from method execution
- Can request a calculation early and check later whether the result is available

The pattern consists of multiple components:

- The client calls a method of a proxy, which (immediately) returns a future object (can be used to check if results are available and get them)
- At the same time the proxy turns the method call into a request object and adds it to a request queue
- A scheduler takes requests from the request queue and executes it (on some thread)
- Once the request is executed, the result is added to the future object
- Only when the client accesses the future (wants to get the result value), the client thread waits (if the result is already available by that time, no waiting/blocking occurs)
More on OOP

Class Diagrams
The SOLID rules of OOP

Patterns

Patterns
Creational Patterns
Structural Patterns
Behavioral Patterns
Concurrency Patterns
Antipatterns

Discussion
Anti-patterns

- **God object, The blob**: One massive class containing all functionality
- **Object orgy**: Not using data encapsulation (not distinguishing between public and private members); some objects modify the internals of others more than their own (→ it is not clear “who is doing what to whom”)
- **Not using polymorphism**: Having many parallel sections of identical if statements rather than using classes and subclasses
- **Misusing (multiple) inheritance**: Some inherited methods do not make sense for subclass; violations of the Liskov substitution principle
- **Overgeneralization/inner platform effect**: A system so general and customizable that it reproduces your development platform
1 More on OOP
   - Class Diagrams
   - The SOLID rules of OOP

2 Patterns
   - Patterns
   - Creational Patterns
   - Structural Patterns
   - Behavioral Patterns
   - Concurrency Patterns
   - Antipatterns

3 Discussion
Common criticism

“Repetitive use of the same patterns and lots of boilerplate indicates lack or abstraction or lacking features of your programming language.”

- **Example**: If functions are first-level objects (can be passed around like normal datatypes), I do not need to define a strategy class hierarchy. However, this could still be considered the same “pattern” (only with a simpler implementation)

- The “Patterns” give you **vocabulary** to describe your problem in an abstract way, even if the implementation details vary a lot between languages

- Lots of pattern boilerplate should make you think about your design and language choices

- Be aware that the implementation of (or even the need for) certain patterns can be very dependent on your language features
"Design patterns are used excessively and introduce unneeded complexity."

- Remember the zen of python: simple is better than complex; but complex is better than complicated.
- Do not introduce complexity (use the design pattern) if you do not fully understand why you need it.
- Some people highlight the KISS (keep it simple, stupid) and YAGNI (you aren’t gonna need it) principle.
Common criticism

“The common design patterns are often the direct results of thinking about good software design; focusing on patterns replaces actual thought with cut-and-paste programming.”

- Take discussion of patterns as a mental practice of thinking about good design; avoid simple cut-and-paste
Exercises tomorrow 09:00 – 11:00!

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