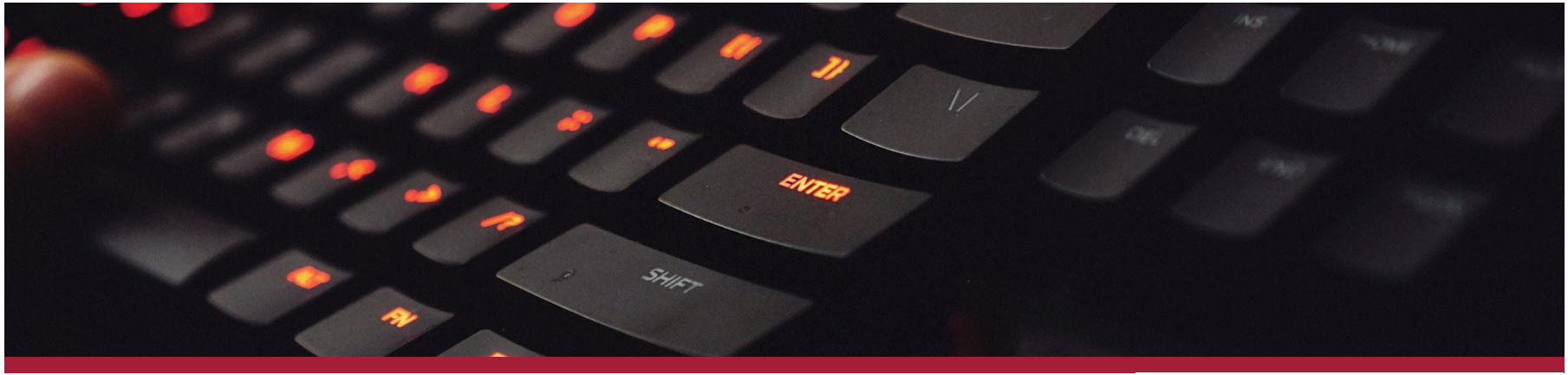
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Quantitative Biology Center
(QBiC)

QBiC



https://www.pexels.com/photo/black-lighted-gaming-keyboard-841228

SOLID coding with “Uncle Bob”

Sven Fillinger

Quantitative Biology Center
University of Tübingen

Who am I?

agricultural biology

molecular biologist

&

genetic engineer

---

bioinformatics

data scientist

&

software engineer

Where do I work?

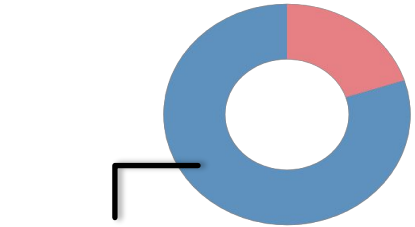
Quantitative Biology Center

Core facility

University of Tübingen

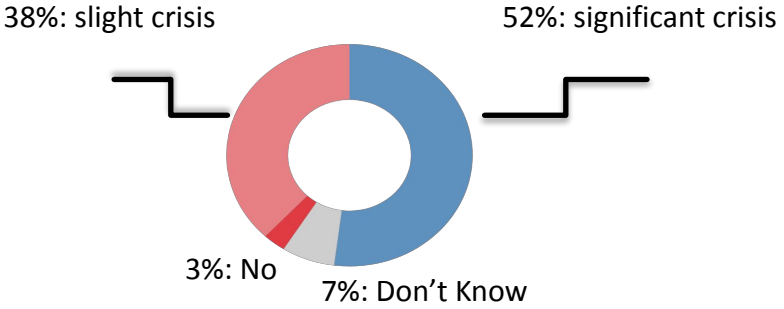
The problem we aim to solve

Data sustainability



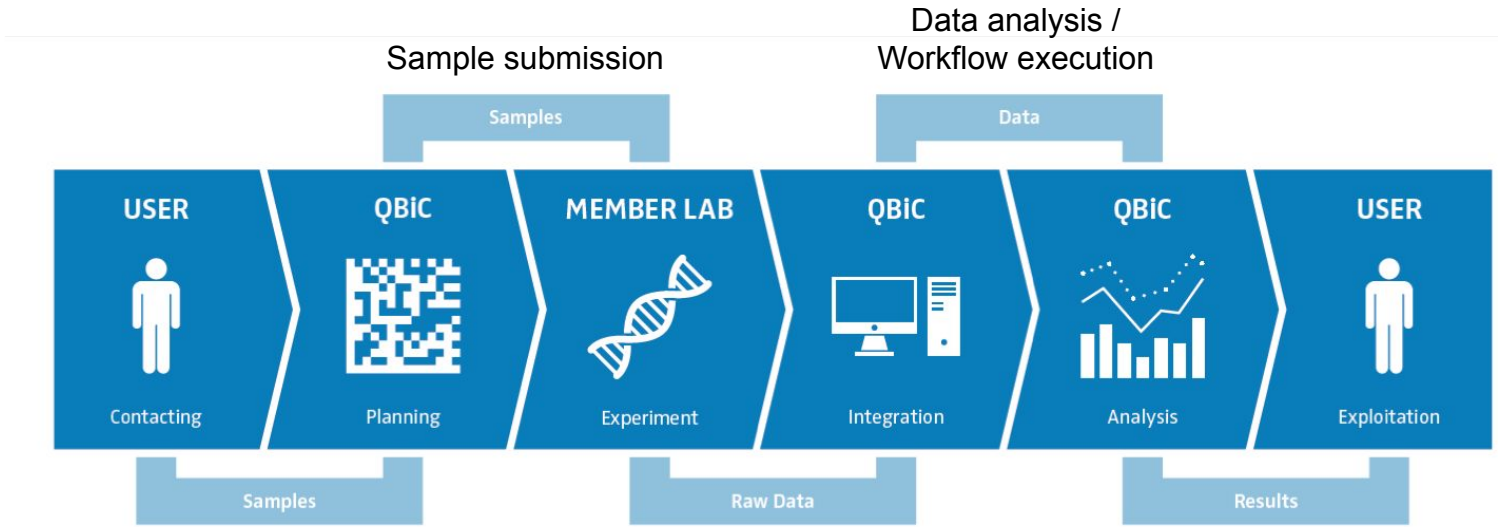
Data availability loss 17% p. a.; **80%** for data older than 20 years (*Vines et al., Curr Biol, 2014)*

Best practices in omics data analysis: Focus on FAIR and reproducibility



Large scale survey: 90% of scientists **fear reproducibility issues** (*Baker, Nature*, 2016)

QBiC process



Consulting / Planning
(support@qbic.zendesk.com)

Automated data
transfer

Visualization /
Reporting of results

nf-core
nextflow

scalable and reproducible scientific workflows

community curated best-practice pipelines

Correspondence | Published: 13 February 2020

The nf-core framework for community-curated bioinformatics pipelines

Philip A. Ewels, Alexander Peltzer, Sven Fillinger, Harshil Patel, Johannes Alneberg, Andreas Wilm, Maxime Ulysse Garcia, Paolo Di Tommaso & Sven Nahnsen

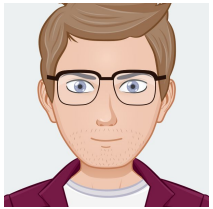
Nature Biotechnology **38**, 276–278(2020) | Cite this article

5137 Accesses | **22** Citations | **175** Altmetric | Metrics

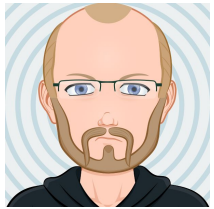
Funding (2020)

Chan Zuckerberg Initiative

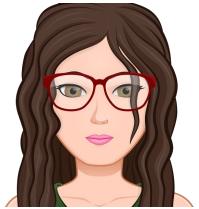
The development team



Andreas

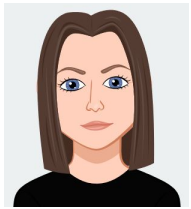


Matthias



Jennifer
Deputy Lead

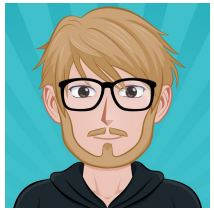
Sven
Lead



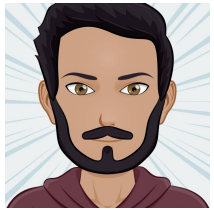
Aline

Luis

Tobias



Steffen



John

What is software architecture

Text cloud

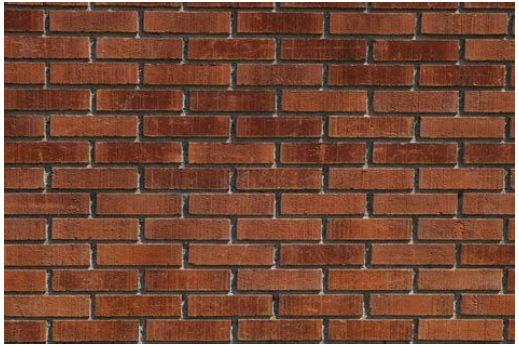
https://livecloud.online/join/44VT

What is software architecture

software

flexibel extendable maintainable

What is the topic all about?

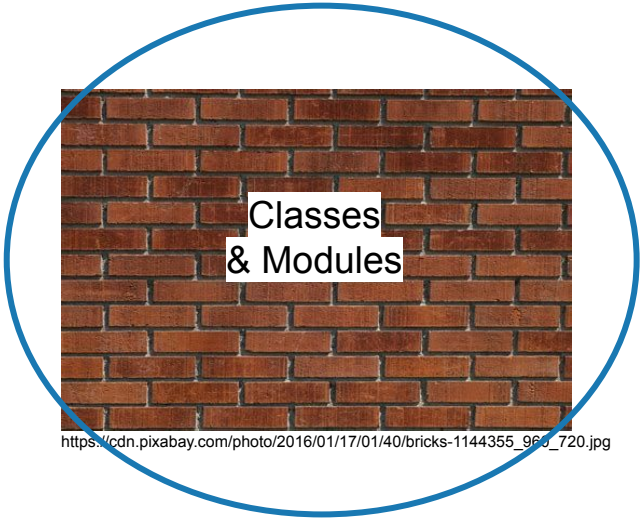


https://cdn.pixabay.com/photo/2016/01/17/01/40/bricks-1144355_960_720.jpg

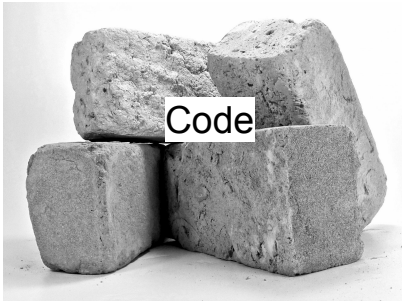
https://www.publicdomainpictures.net/en/view-image.php?image=103587&picture=bricks

https://www.pexels.com/photo/brick-house-3631733/

What is the topic all about?



https://cdn.pixabay.com/photo/2016/01/17/01/40/bricks-1144355_960_720.jpg



https://www.publicdomainpictures.net/en/view-image.php?image=103587&picture=bricks

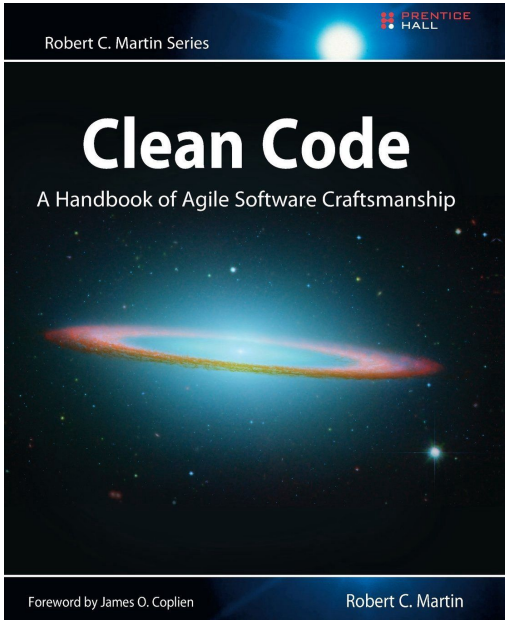
https://www.pexels.com/photo/brick-house-3631733/

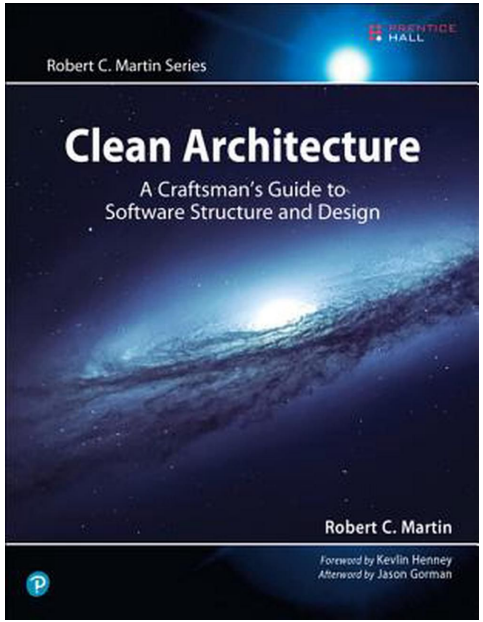
Robert C. Martin aka “Uncle Bob”

- Software engineer, instructor, author
- Co-founder of the Agile Manifesto
- Creator of many software design principles

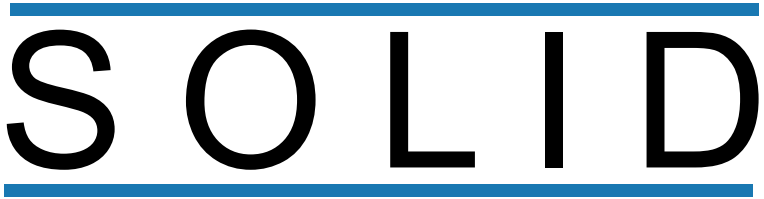
https://en.wikipedia.org/wiki/Robert_C._Martin

Robert C. Martin aka “Uncle Bob”





The SOLID principles



Single Responsibility

Open-closed

Liskov substitution

Interface segregation

Dependency inversion

SOLID: Single Responsibility Principle

What it is NOT:

“Every module (class) should do only one thing.” ✖

Function definition!

SOLID: Single Responsibility Principle

Historical description:

“A module should have one, and only one, reason to change.”

SOLID: Single Responsibility Principle

Historical description:

“A module should have one, and only one, reason to change.”

SOLID: Single Responsibility Principle

Historical description:

“A module should be responsible to one, and only one, user or stakeholder.”

SOLID: Single Responsibility Principle

Historical description:

“A module should be responsible to one, and only one, user or stakeholder.”

groups of individuals or components (actor)

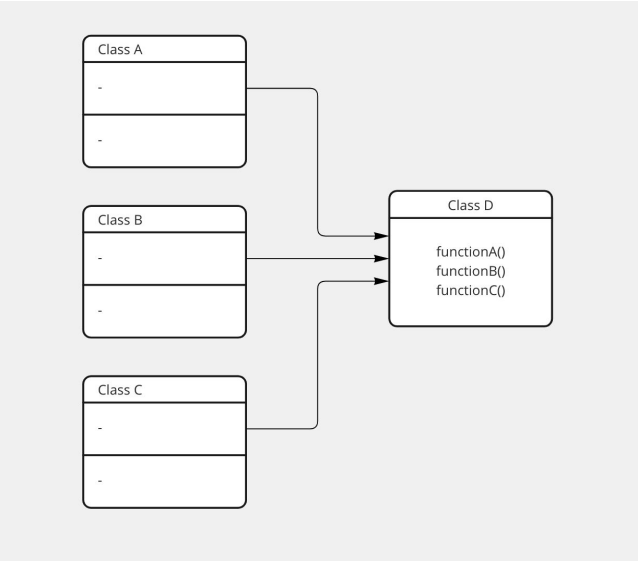
SOLID: Single Responsibility Principle

Historical description:

“A module should be responsible to one, and only one, actor.”

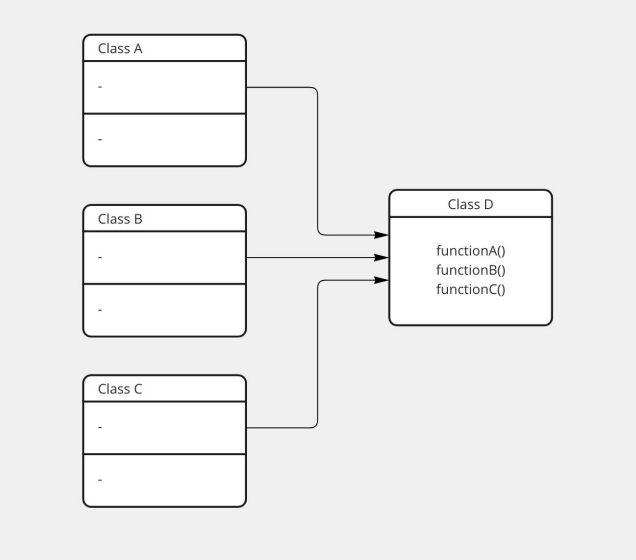
SOLID: Single Responsibility Principle

“A module should be responsible to one, and only one, actor.”



SOLID: Single Responsibility Principle

“A module should be responsible to one, and only one, actor.”

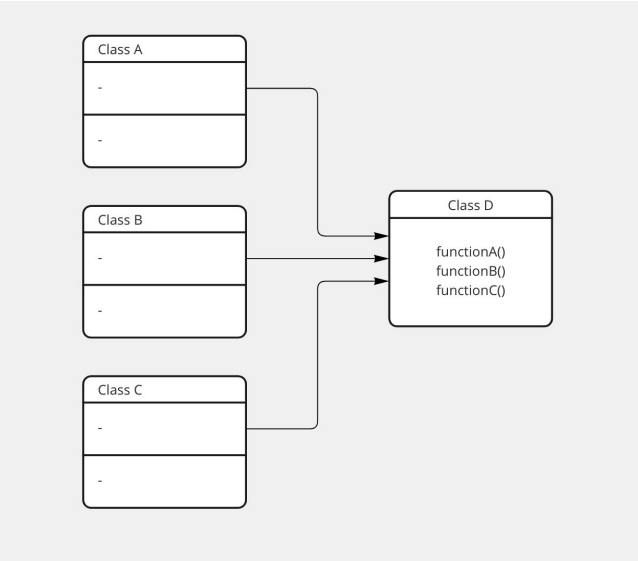


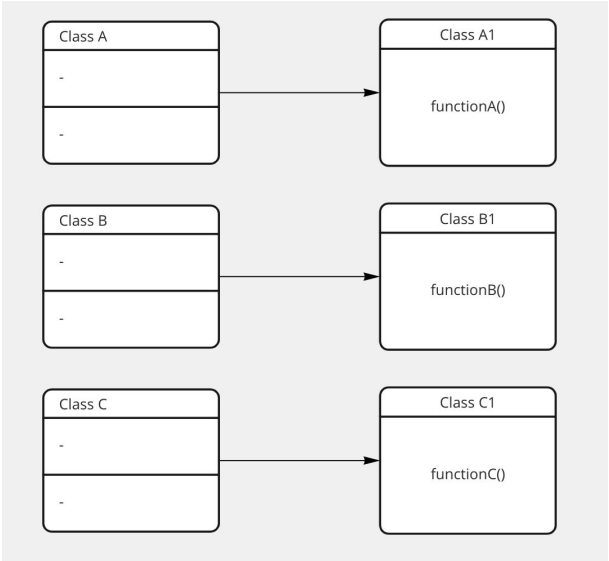

```java
public class ClassD {

  public int functionA() {
    return helperFunction() - 20;
  }

  public int functionB() {
    return helperFunction() + 10;
  }

  2 usages
  private int helperFunction() {
    var x = 0;
    // do some fancy calculations
    return x;
  }

  public void functionC() {
  }

}
```

```java
public class ClassD {

  public int functionA() {
    return helperFunction() - 20;
  }

  public int functionB() {
    return helperFunction() + 10;
  }

  2 usages
  private int helperFunction() {
    var x = 0;
    // do some fancy calculations
    return x + 100;
  }

  public void functionC() {
  }

}
```

SOLID: Single Responsibility Principle

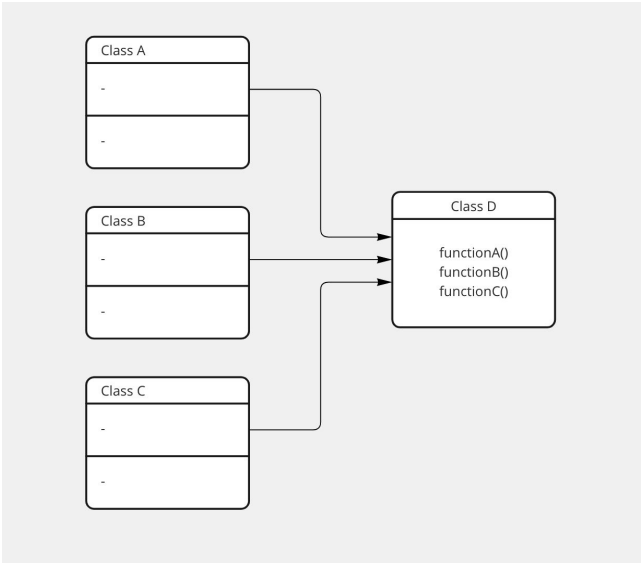
“A module should be responsible to one, and only one, actor.”

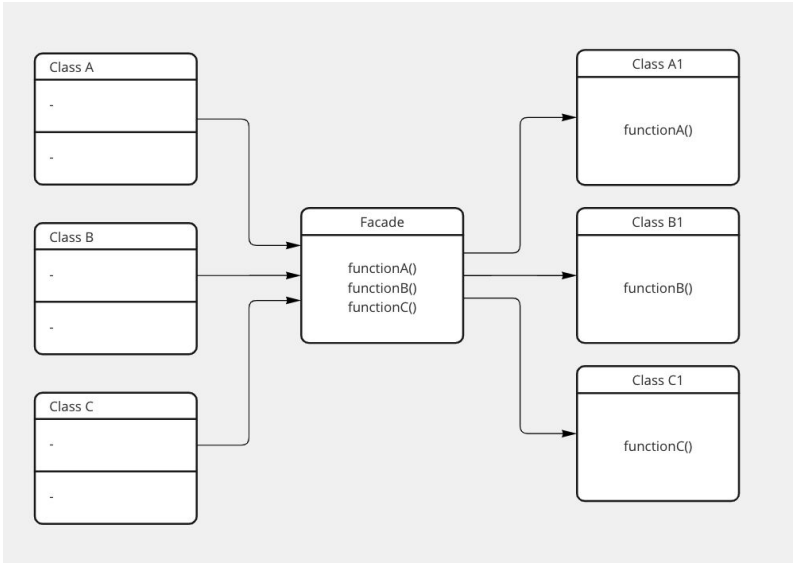




SOLID: Single Responsibility Principle

“A module should be responsible to one, and only one, actor.”





SOLID: Open-Closed Principle

"*A software artifact should be open for extension but closed for modification.*"

```
List t = {1, 2, 3, 4, 5, 6, 7}
```

Addition operation with
`operand o`

SOLID: Open-Closed Principle

"A software artifact should be open for extension but closed for modification."

```
List t = {1, 2, 3, 4, 5, 6, 7}
```

Addition operation with
`operand o`

```java
public class HyperList {

  1 usage
  private final List<Integer> hyperList = new ArrayList<>();

  public HyperList() {}

  public void add(int operand) {
    // Iterate through list and add the operand
    // to every element
  }
```

Open for extension: 

Closed for modification: 

SOLID: Open-Closed Principle

"A software artifact should be open for extension but closed for modification."

```
List t = {1, 2, 3, 4, 5, 6, 7}
```

```
public interface Operation {

  1 usage
  int apply(int o1, int o2);
}
```

```
public class Addition implements Operation {

  1 usage
  @Override
  public int apply(int o1, int o2) {
    return o1 + o2;
  }
}
```

```
public class Subtraction implements Operation {

  1 usage
  @Override
  public int apply(int o1, int o2) {
    return o1 - o2;
  }
}
```

SOLID: Open-Closed Principle

"A software artifact should be open for extension but closed for modification."

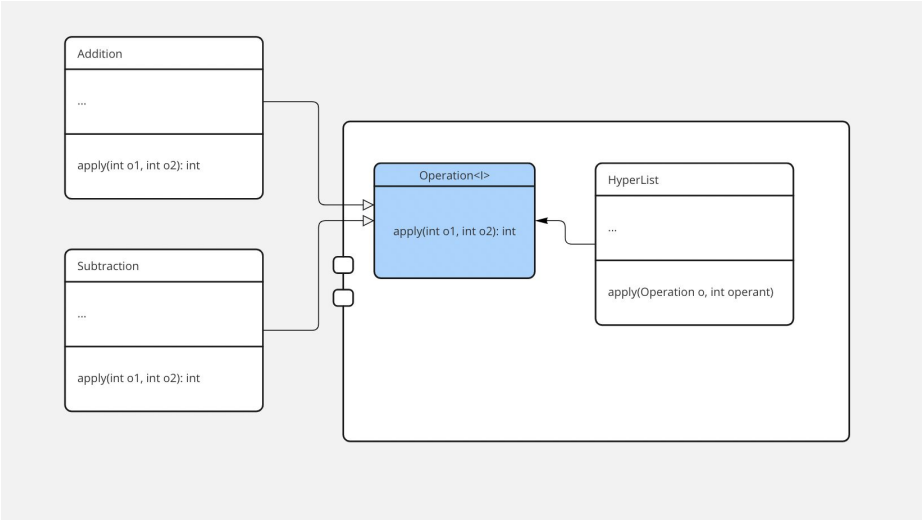
```
List t = {1, 2, 3, 4, 5, 6, 7}
```

```java
public interface Operation {

    1 usage
    int apply(int o1, int o2);
}
```

```java
public class HyperList {

    1 usage
    private final List<Integer> hyperList = new ArrayList<>();

    public HyperList() {}

    public void apply(Operation operation, int operand) {
        // for every element apply the operation
        hyperList.replaceAll(element -> operation.apply(element, operand));
    }
}
```

SOLID: Open-Closed Principle

"*A software artifact should be open for extension but closed for modification.*"



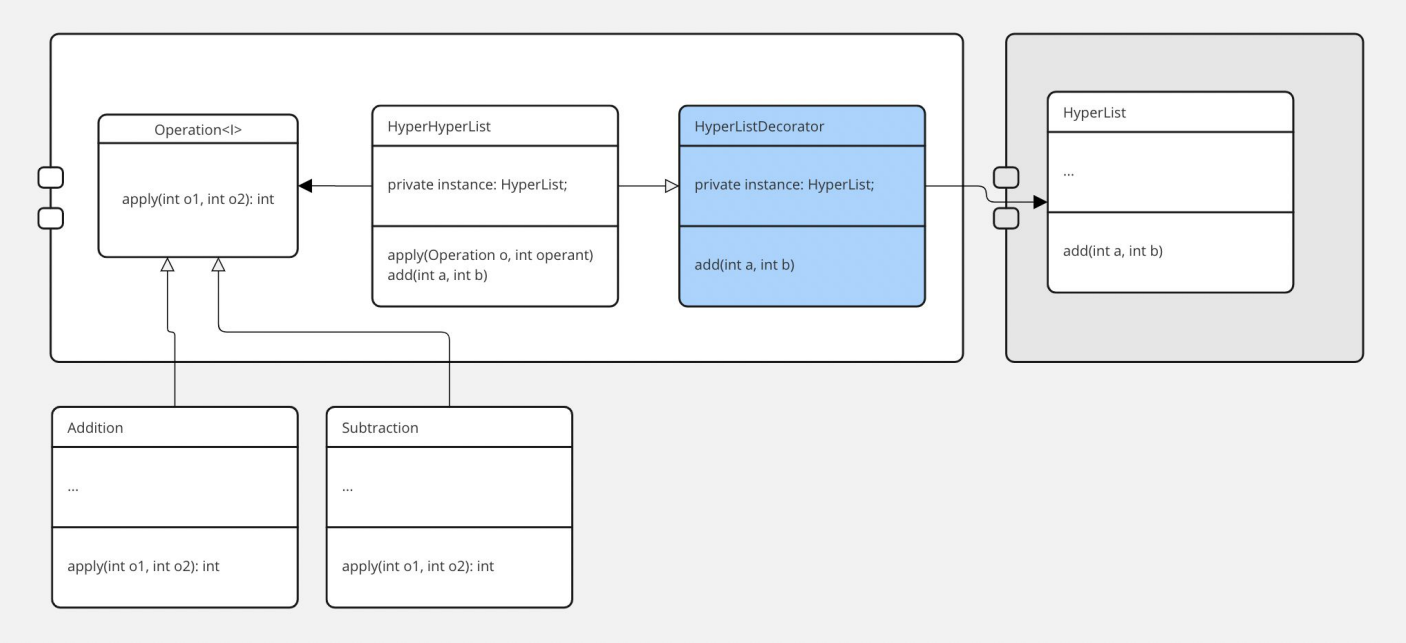
open:

closed:

SOLID: Open-Closed Principle

"*What if don't control the source code?*"

a) Decorate Pattern
b) Inheritance



SOLID: Liskov Substitution Principle

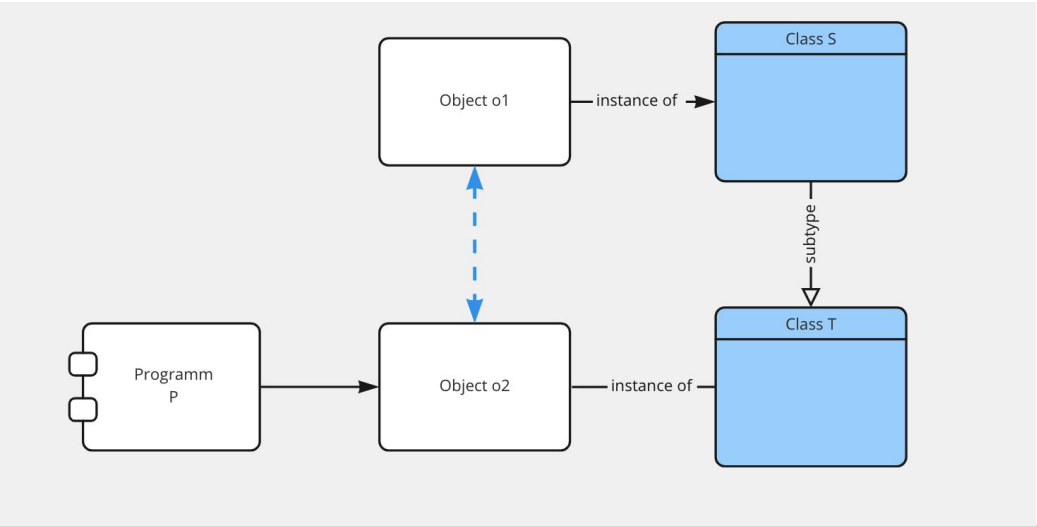
[...] If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

(Barbara Liskov, 1988)

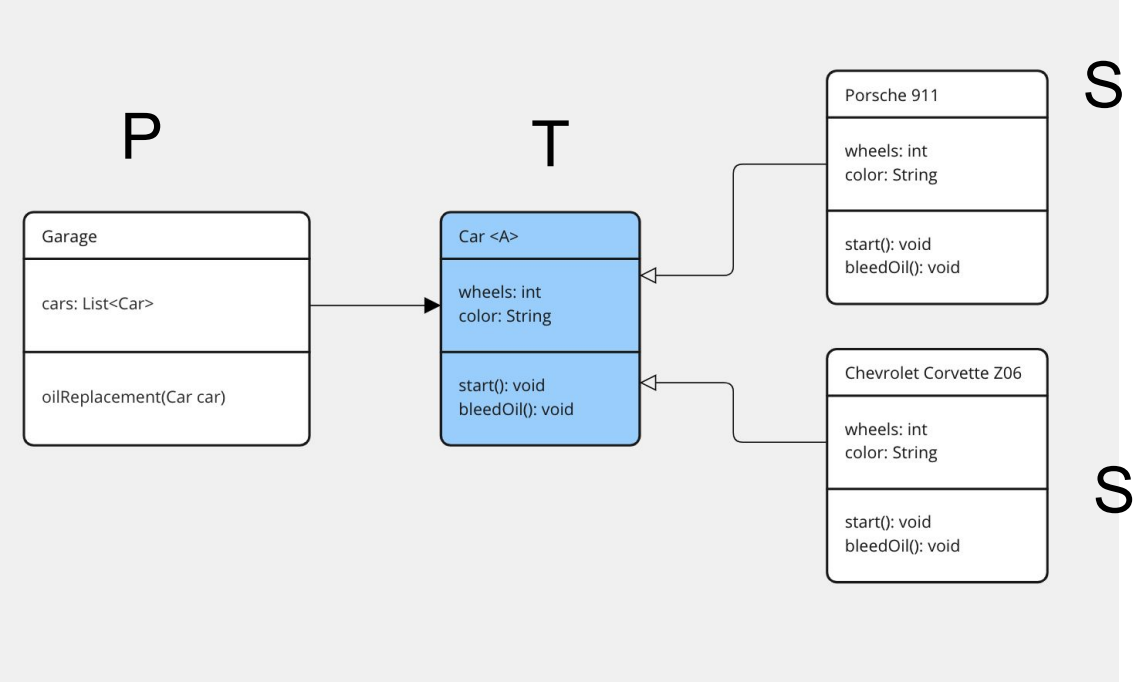
SOLID: Liskov Substitution Principle

[...] If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

(Barbara Liskov, 1988)



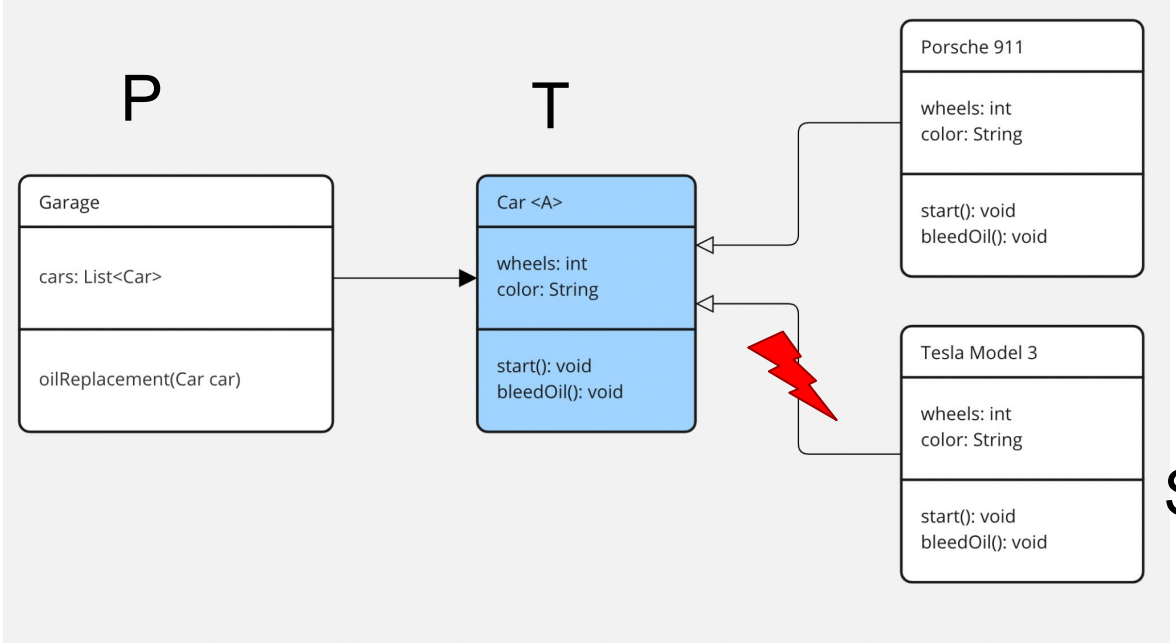
SOLID: Liskov Substitution Principle



S is a subtype of T

behaviour of P unchanged

SOLID: Liskov Substitution Principle

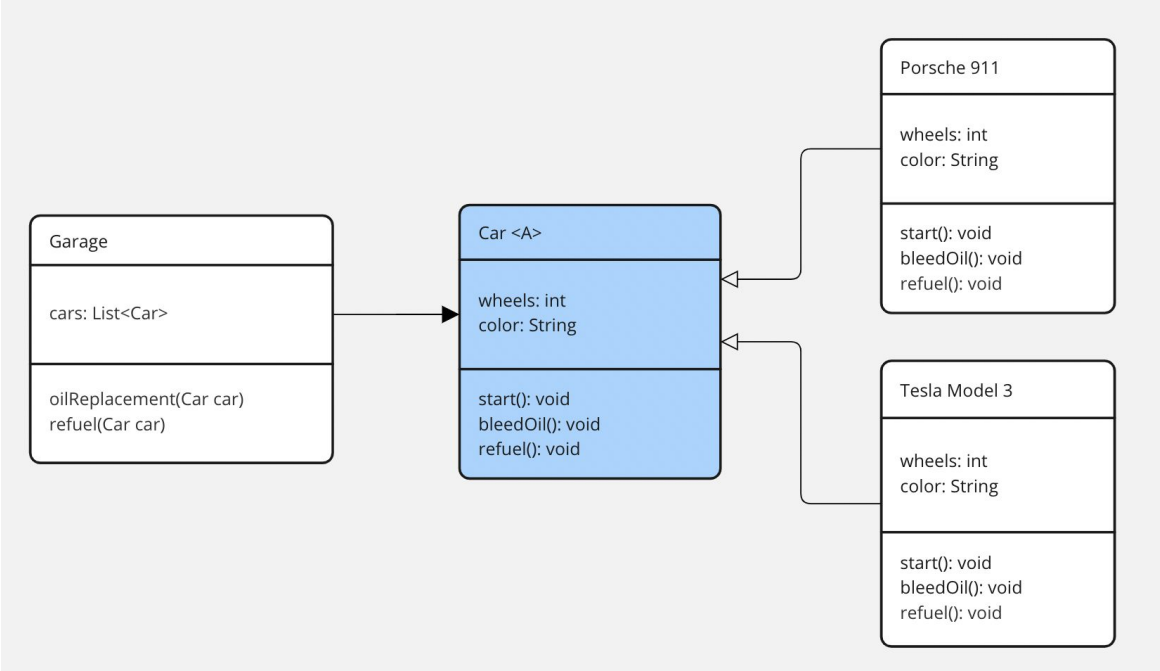


S is not a subtype of T

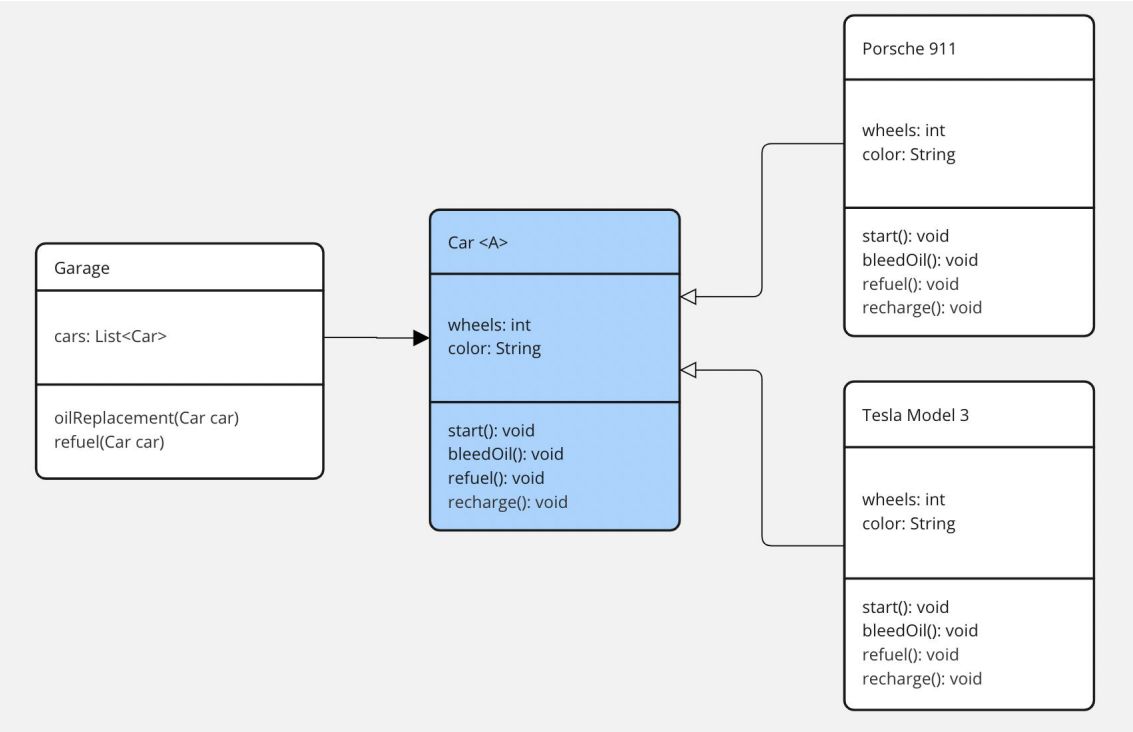
behaviour of P changed

SOLID: Interface Segregation Principle

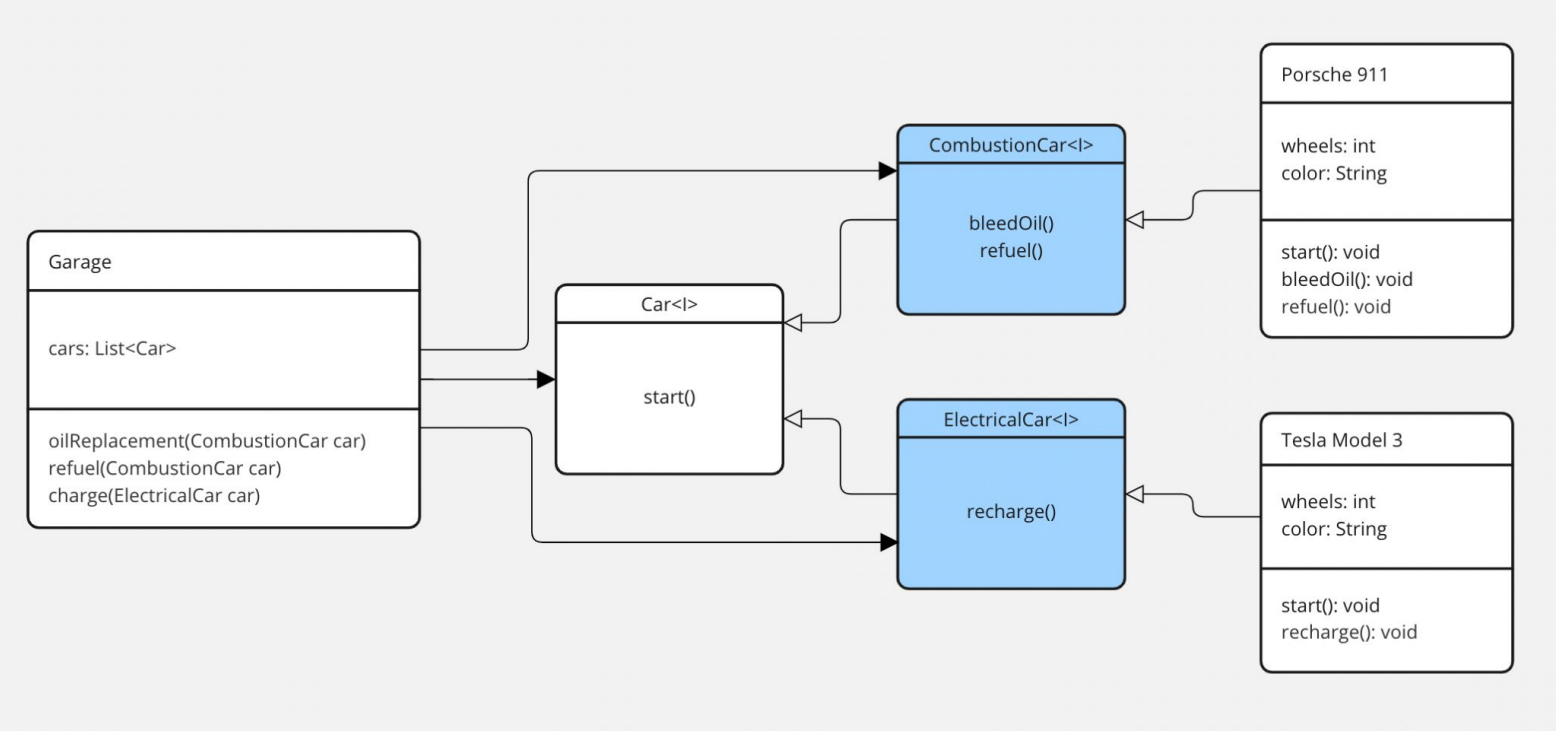
“No code should be forced to depend on methods it does not use.”



SOLID: Interface Segregation Principle



SOLID: Interface Segregation Principle



SOLID: Interface Segregation Principle

Why should I care in Python?

Duck-typing in Python

SOLID: Interface Segregation Principle

```python
class Shape():

    def draw_circle(self):
        pass

    def draw_square(self):
        print("drawing a square")

class Circle(Shape):

    def draw_circle(self):
        print("drawing an awesome circle")


def main():
    circle = Circle()
    circle.draw_circle()
    circle.draw_square()

if __name__ == "__main__":
    main()
```


?

“drawing an awesome circle”
“drawing a square”

But Circle is a real subtype of Shape (LSP)

SOLID: Interface Segregation Principle

```python
class RectangularShape():

    def draw_rectangle(self):
        print("drawing a square")

    def stack(self):
        print("stack it")

class CircularShape():

    def draw_circle(self):
        print("drawing an awesome circle")

    def spin(self):
        print("spin in in ing")

class RedCircle(CircularShape):
    pass

class BigRectangle(RectangularShape):
    pass
```

```python
def main():
    circle = RedCircle()
    circle.draw_circle()
    circle.spin()
    rectangle = BigRectangle()
    rectangle.draw_rectangle()
    rectangle.stack()
```

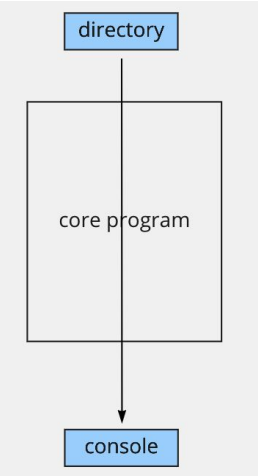
“drawing an awesome circle”
“spin in in ing”
“drawing a square”
“stack it”

SOLID: Dependency Inversion Principle

"Depend in the direction of abstraction. High level modules should not depend upon low level details."

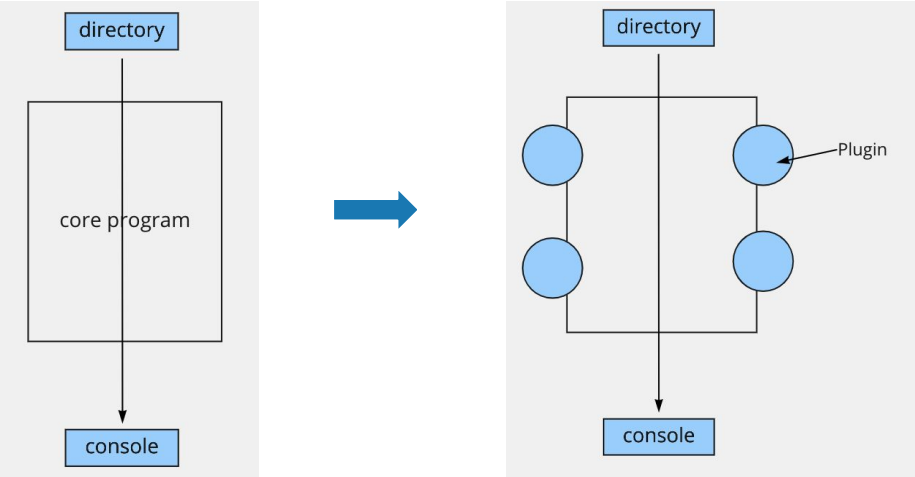
SOLID: Dependency Inversion Principle

Example: An extensible app to perform code linting

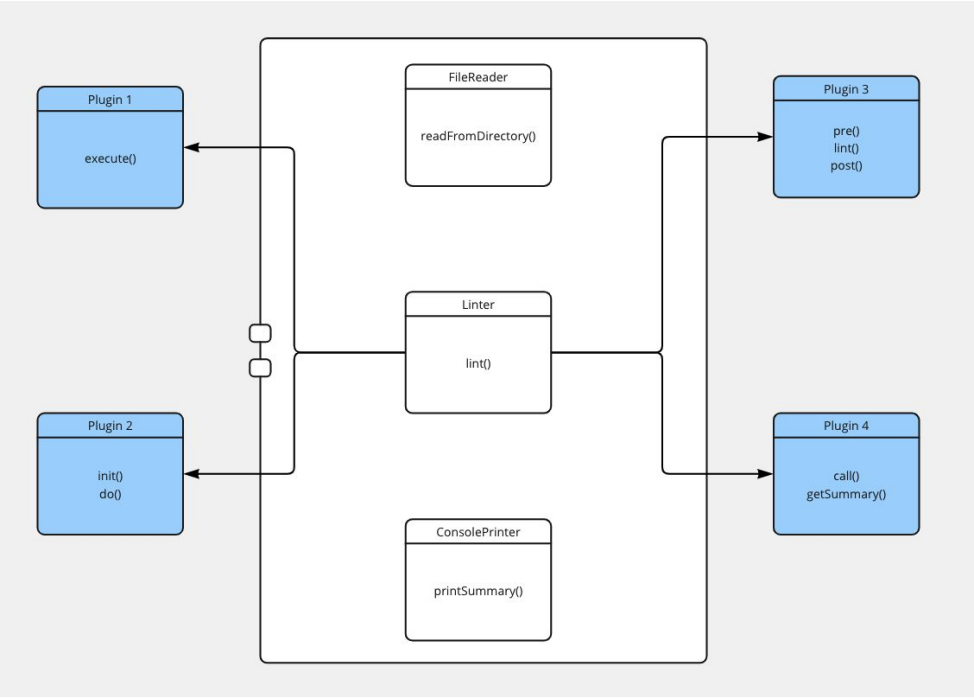


SOLID: Dependency Inversion Principle

Example: An extensible app to perform code linting



SOLID: Dependency Inversion Principle

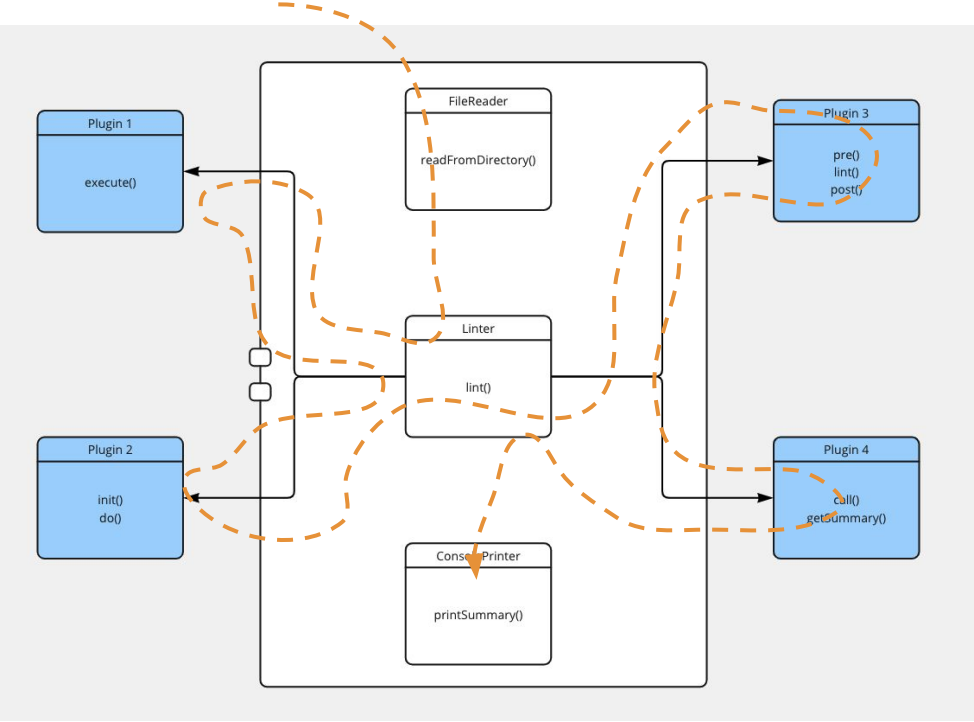


Flexibility

Extendability

Stability

SOLID: Dependency Inversion Principle



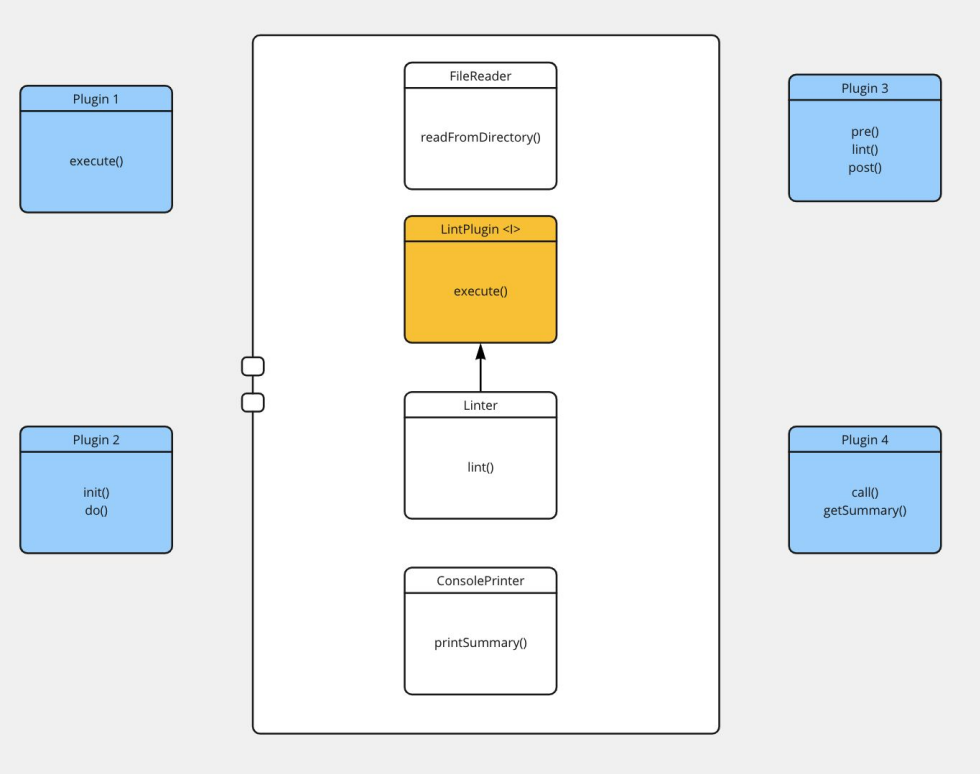
Flexibility

Extendability

Stability

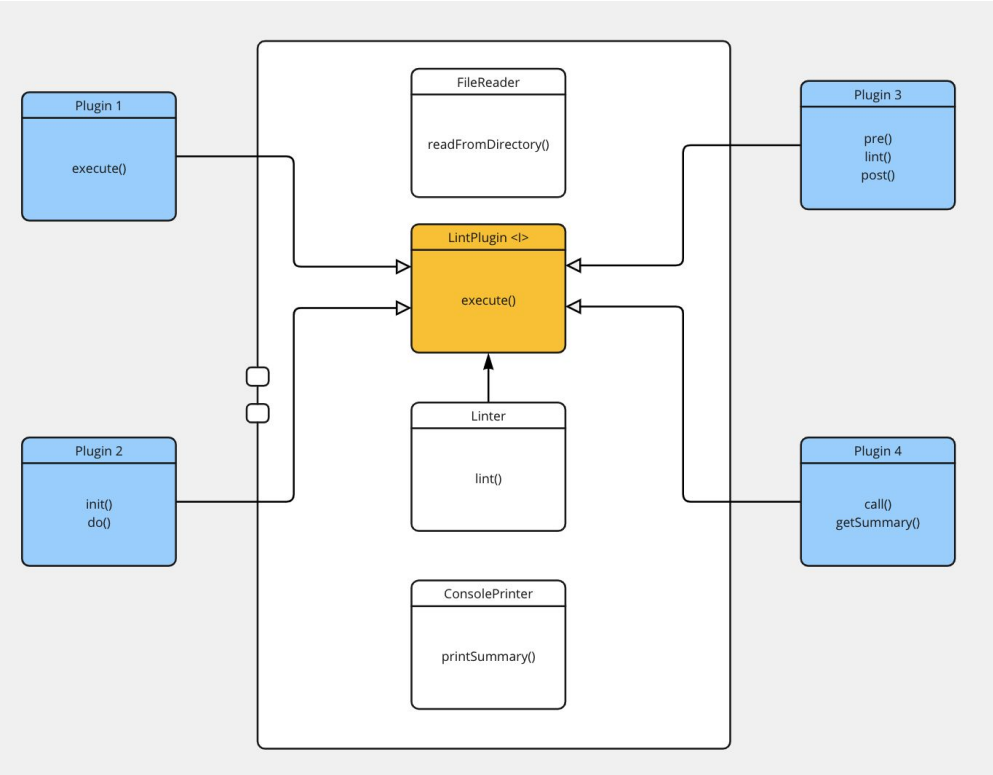
Control flow

SOLID: Dependency Inversion Principle



Introduce interface
(or abstract class)

SOLID: Dependency Inversion Principle

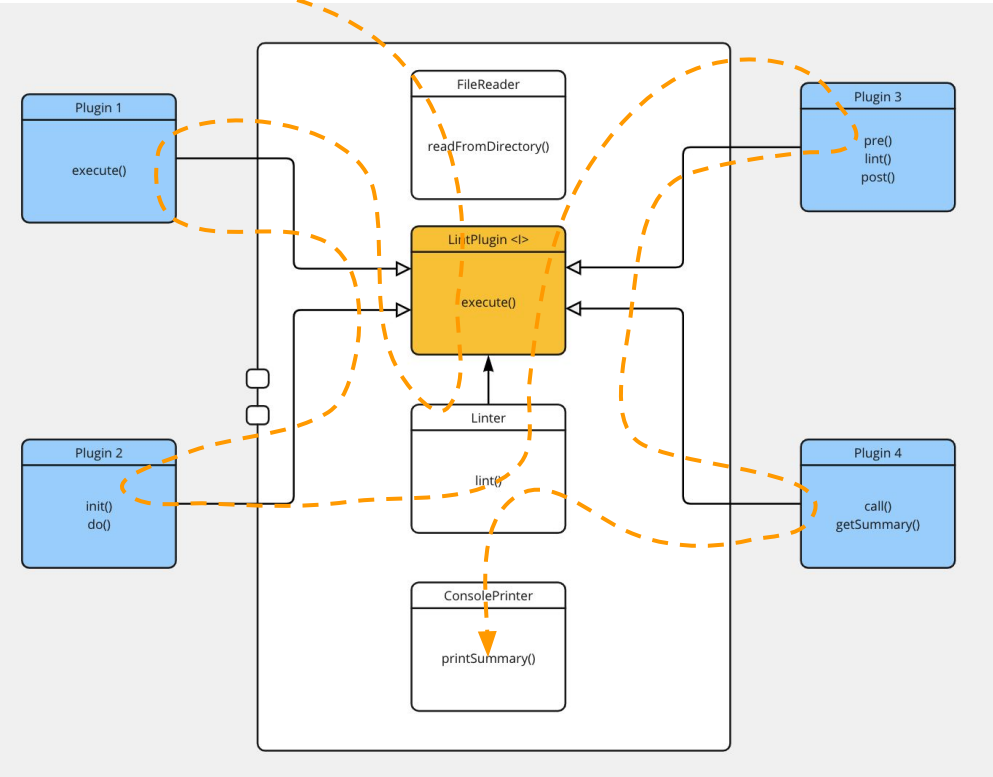


Flexibility

Extendability

Stability

SOLID: Dependency Inversion Principle



Flexibility

Extendability

Stability

Control flow

Micro-kernel
or
Plugin
Architecture