Design Principles

Previous Code Convention Discussions

- Style
  - What does the code look like?
  - Is it consistent, scalable, and maintainable?
  - Appearance, format, readability

- Naming
  - How are we describing the identifiers in our code?
  - Are they understandable?
  - Do the names reflect purpose/responsibility?
  - Program comprehension

SOLID Principles for Object-Oriented Design

- Five basic principles (guidelines) for Object-Oriented Design (OOD)

- Results in systems that are:
  - Easy to maintain
  - Easy to extend

- SOLID is a guide for:
  - Creating designs from scratch
  - Improving existing designs

SOILD Principles

- **S**ingle Responsibility Principle (SRP)
- **O**pen/closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

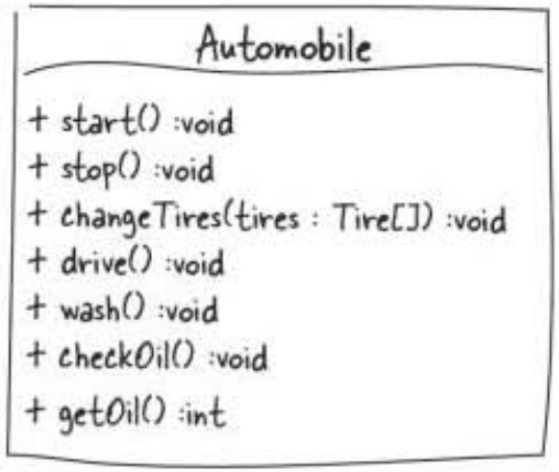
Single Responsibility Principle

- Every class should have **A SINGLE RESPONSIBILITY**
- The responsibility of a class drives its need to change
- Responsibility should be entirely encapsulated by the class
- All functionality of the class should focus on that single responsibility
- Why?
  - More cohesive
  - Easier to understand
  - Easier to maintain

SRP Analysis

- A basic method for determining if a method belongs with a given class.
- A rough approximation
- Need to apply context about the domain and the abstraction

The _[class name]_ _[method name]_ itself.

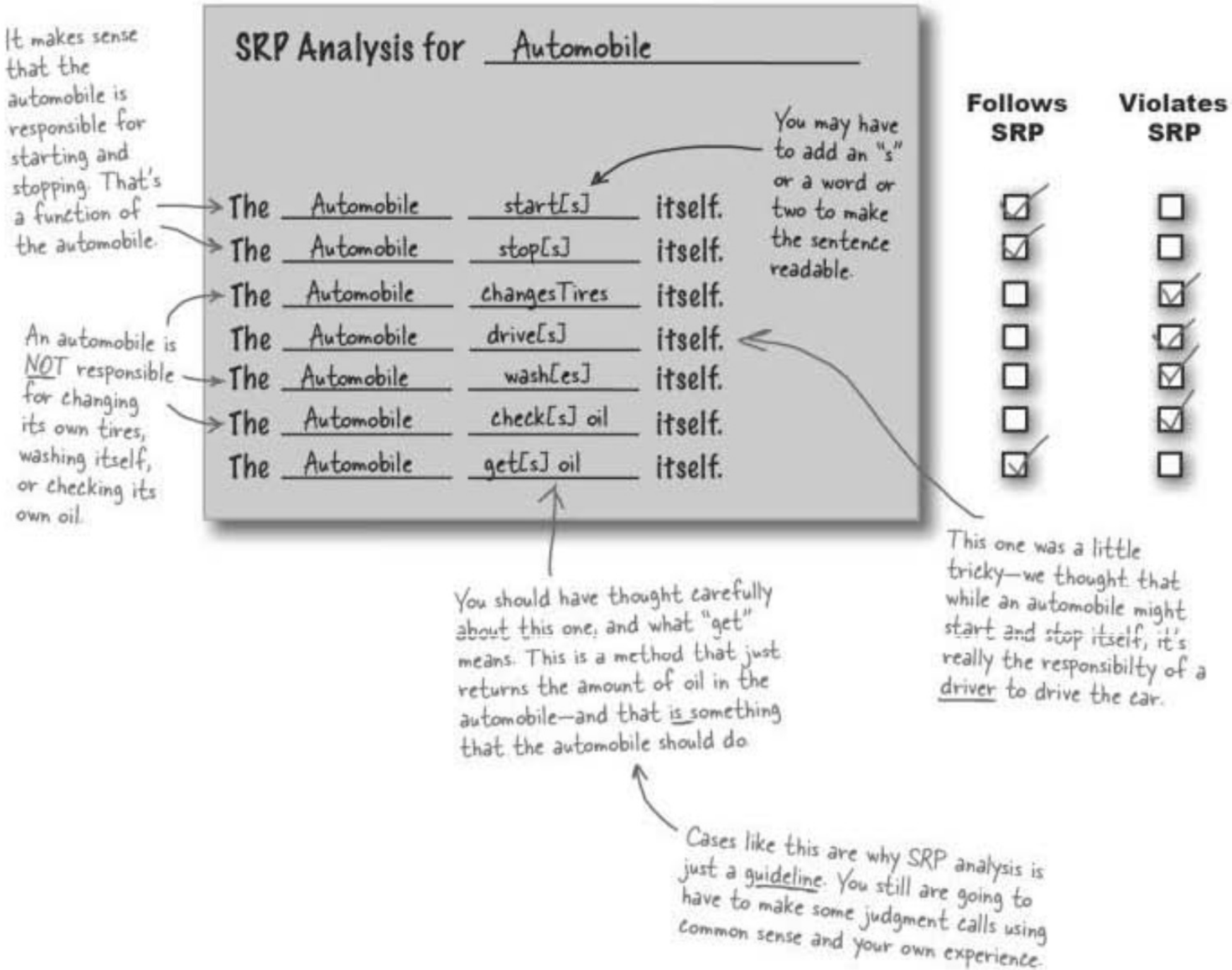


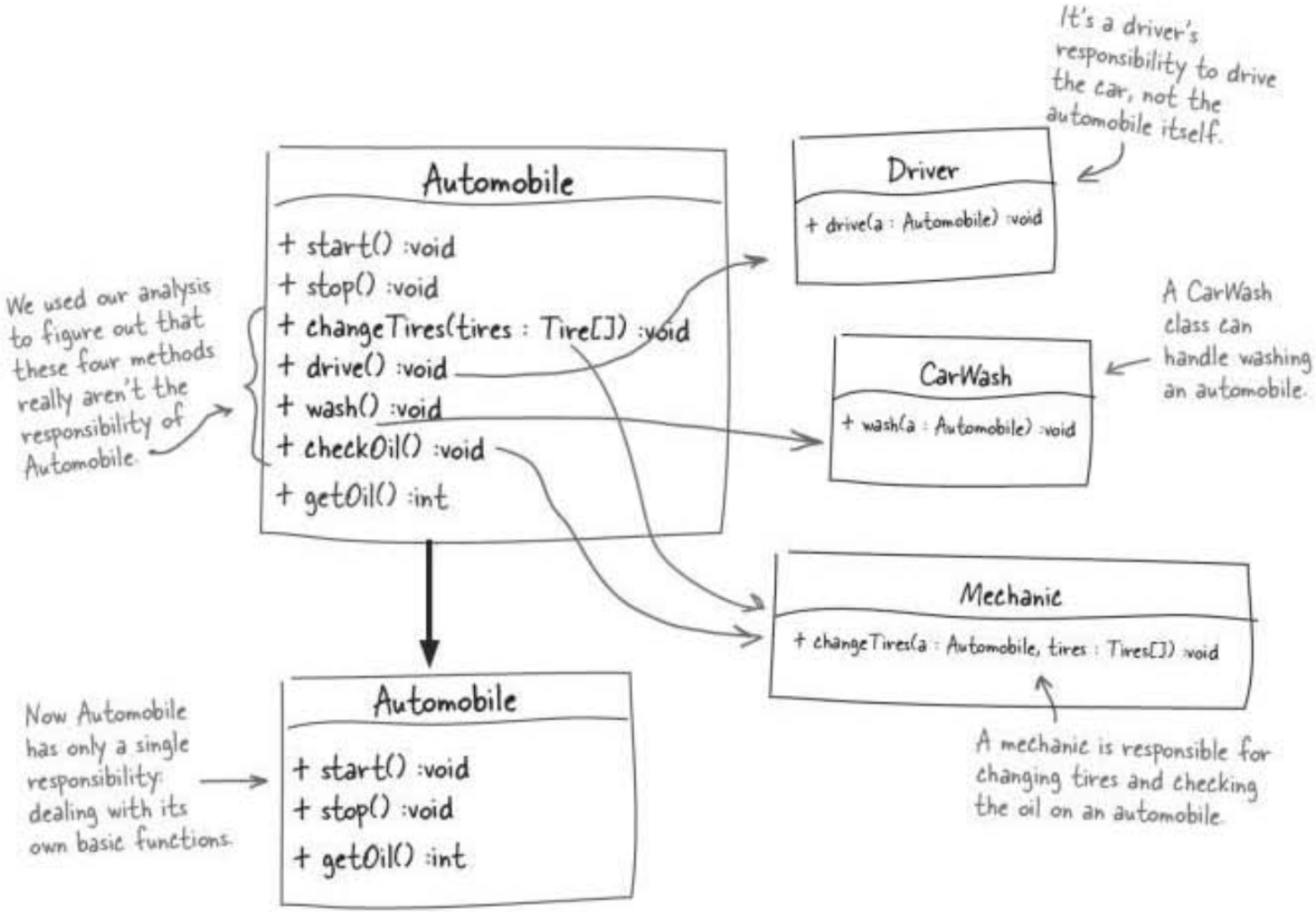
SRP Analysis for Automobile

The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.

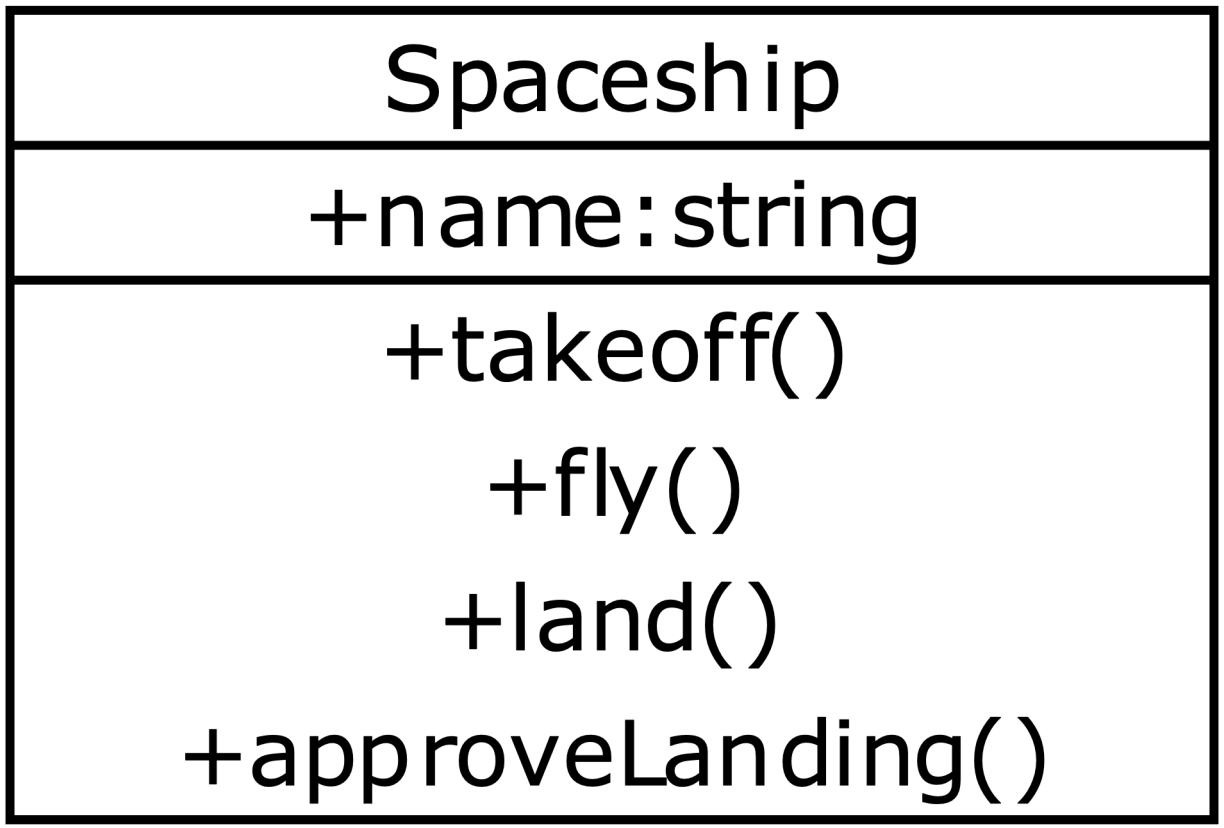
Follows SRP	Violates SRP
☐	☐
☐	☐
☐	☐
☐	☐
☐	☐
☐	☐
☐	☐

If what you read doesn't make sense, then the method on that line is probably violating the SRP.

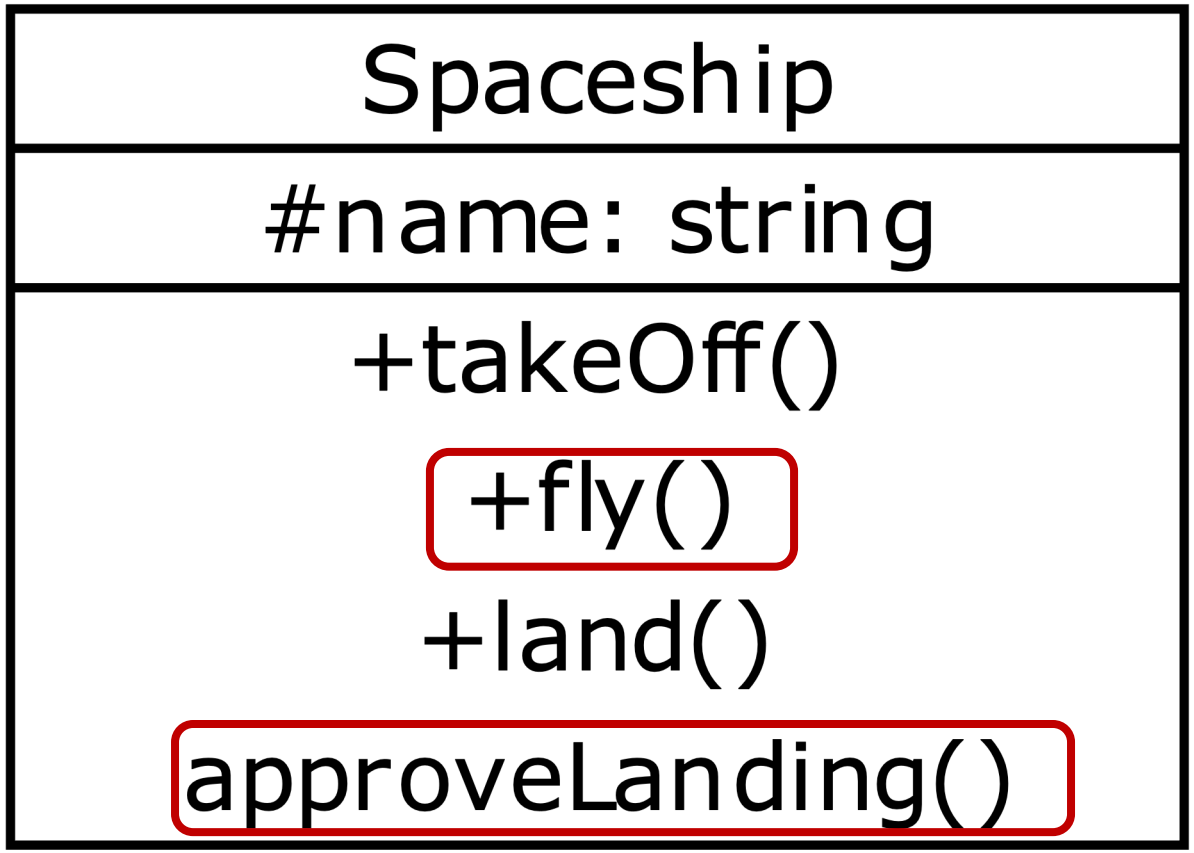

It makes sense that the automobile is responsible for starting and stopping. That's a function of the automobile.
SRP Analysis for Automobile
You may have to add an "s" or a word or two to make the sentence readable.
Follows SRP
Violates SRP
The Automobile start[s] itself.
The Automobile stop[s] itself.
The Automobile changesTires itself.
The Automobile drive[s] itself.
The Automobile wash[es] itself.
The Automobile check[s] oil itself.
The Automobile get[s] oil itself.
An automobile is NOT responsible for changing its own tires, washing itself, or checking its own oil.
This one was a little tricky—we thought that while an automobile might start and stop itself, it's really the responsibilty of a driver to drive the car.
You should have thought carefully about this one, and what "get" means. This is a method that just returns the amount of oil in the automobile—and that is something that the automobile should do.
Cases like this are why SRP analysis is just a guideline. You still are going to have to make some judgment calls using common sense and your own experience.


It's a driver's responsibility to drive the car, not the automobile itself.
Automobile
+ start() :void
+ stop() :void
+ changeTires(tires : Tire[]) :void
+ drive() :void
+ wash() :void
+ checkOil() :void
+ getOil() :int
Driver
+ drive(a : Automobile) :void
A CarWash class can handle washing an automobile.
CarWash
+ wash(a : Automobile) :void
We used our analysis to figure out that these four methods really aren't the responsibility of Automobile.
Mechanic
+ changeTires(a : Automobile, tires : Tires[]) :void
A mechanic is responsible for changing tires and checking the oil on an automobile.
Now Automobile has only a single responsibility: dealing with its own basic functions.
Automobile
+ start() :void
+ stop() :void
+ getOil() :int

SRP Example

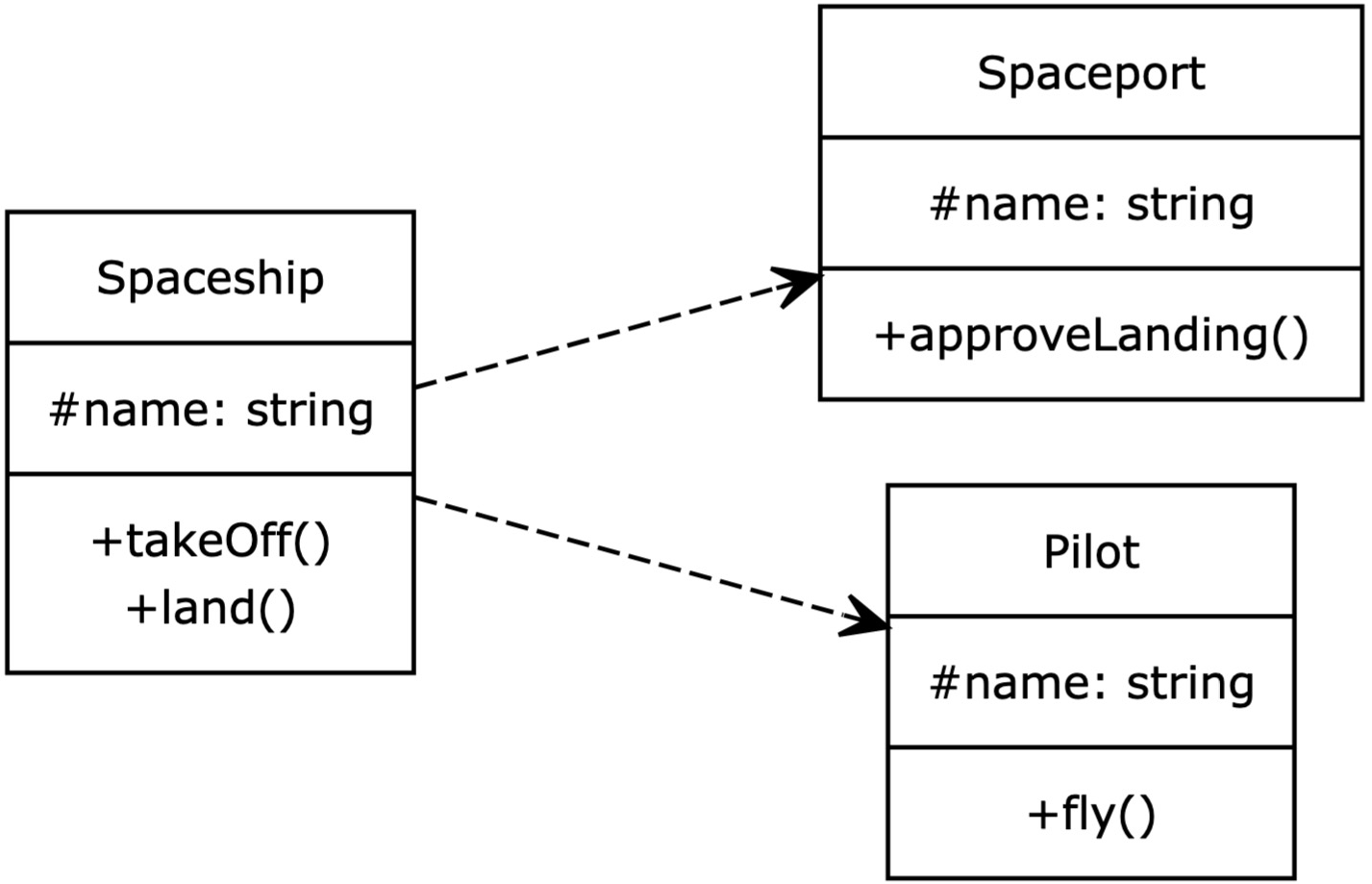


SRP Example - Violation



CREATED WITH YUML

SRP Example - Compliant

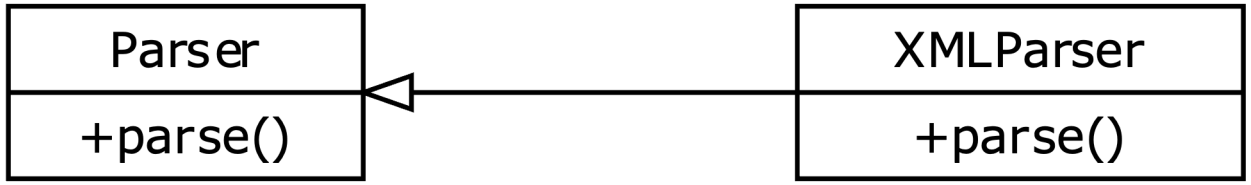


Open/Closed Principle

- Software entities (classes, function, etc.) should be open for extension but closed for modification
- Closed - as can be compiled, stored in a library, and used by client classes
- Open - as any new class can inherit and add new features
- Why?
  - Client code dependent on base (closed) class unaffected
  - Less testing
  - Less code to review

Meyer's Open/Closed Principle

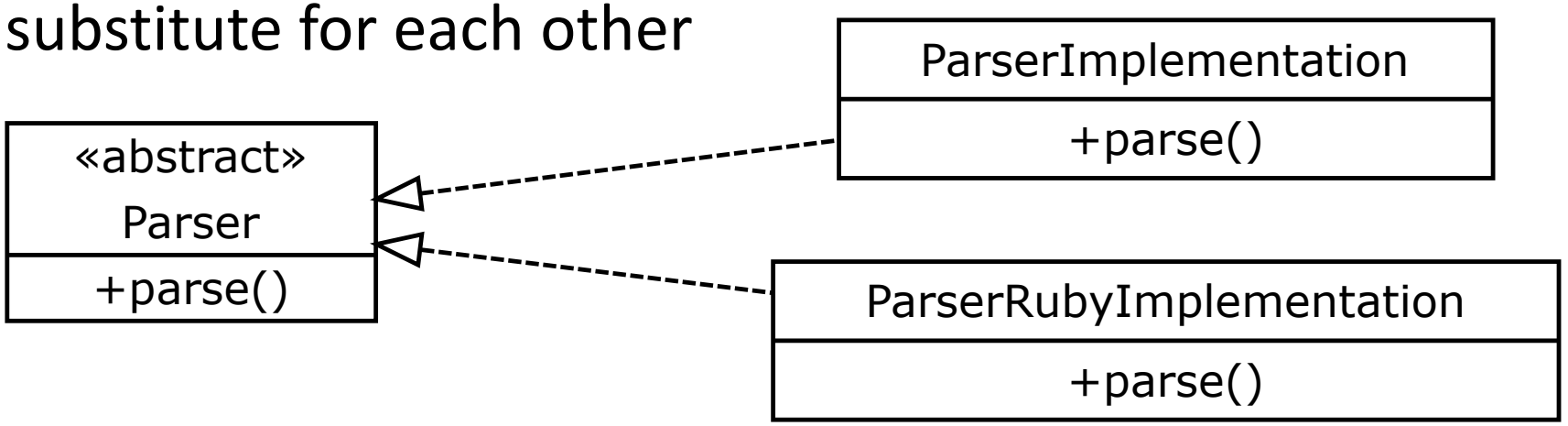
- Implementation is extended through inheritance



- "Open" means available for extension (generalization/inheritance)
- "Closed" to avoid changes to the original class
- New functionality by adding a new class, not changing current ones
- Results in tight coupling between base and derived classes

Polymorphic Open/Closed Principle

- Abstract base class and multiple implementations that we can substitute for each other

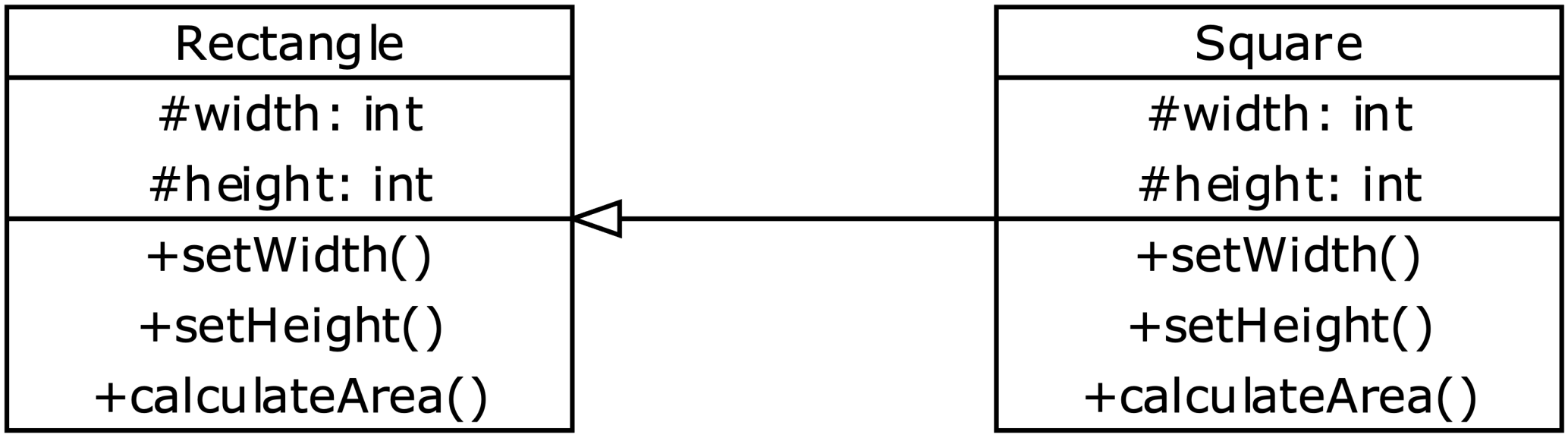


- Base design on abstract base classes
- Focus on sharing the interface, not the implementation
  - “Code to an interface, not an implementation”
- Reuse implementation via delegation

Liskov Substitution Principle

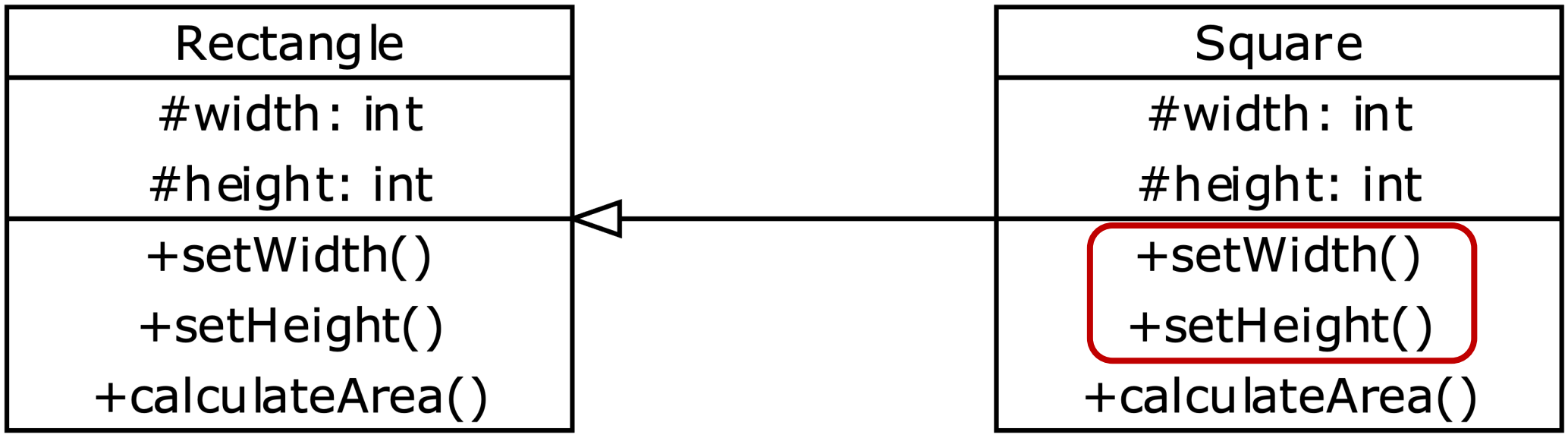
- An Object in a program should be replaceable with an instance of subtypes without affecting program correctness
  - “Objects of subtypes should behave like those of supertypes if used via supertype methods.”
- Preconditions cannot be strengthened in a subtype
- Postconditions cannot be weakened in a subtype
- Invariants of supertype must be preserved in subtype
- History constraint - new methods in subtype cannot introduce state changes in a way that is not permissible in the supertype
- Why?
  - Knowledge/assumptions about base class apply to the subclass
  - Easier to understand
  - Easier to maintain.

LSP Example



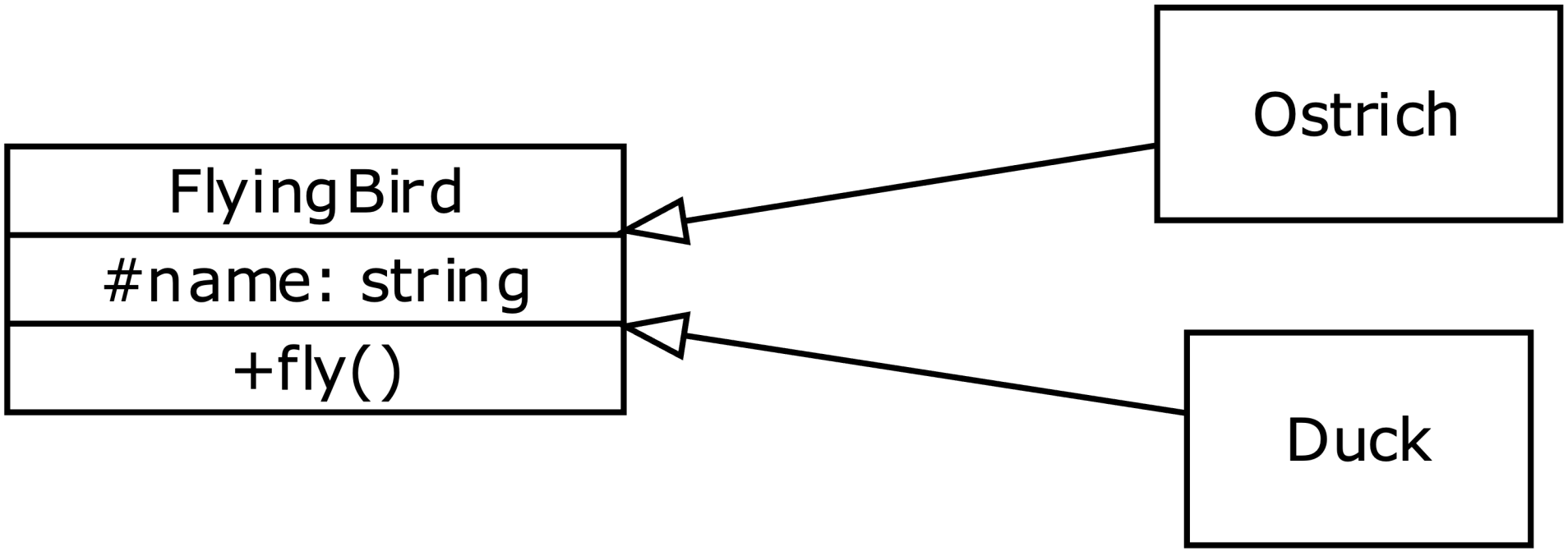
CREATED WITH YUML

LSP Example - Violation



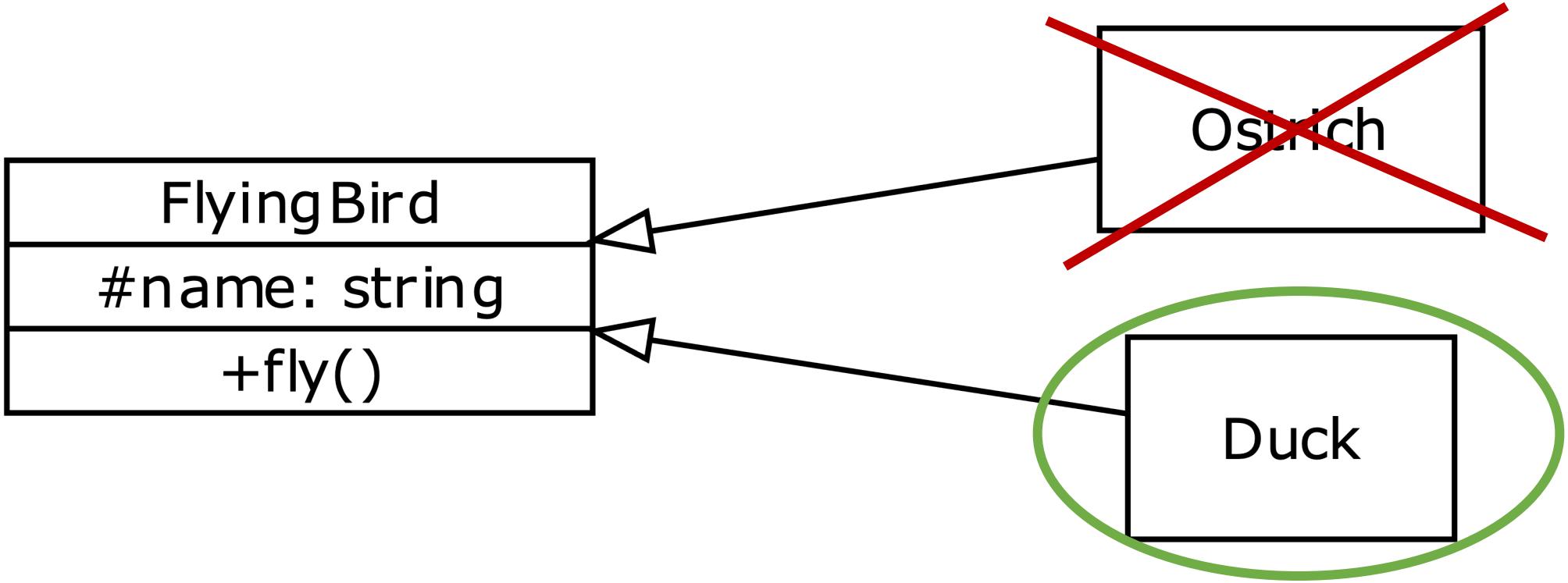
CREATED WITH YUML

LSP Example



CREATED WITH YUML

LSP Example



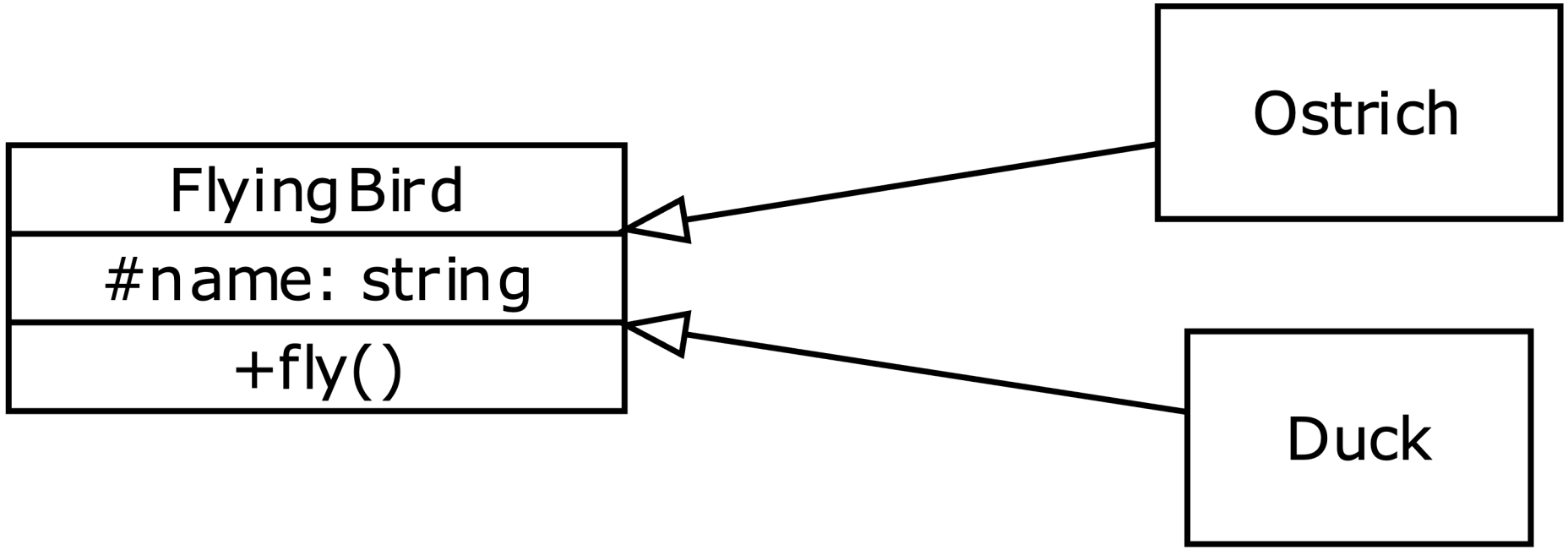
CREATED WITH YUML


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Interface Segregation Principle

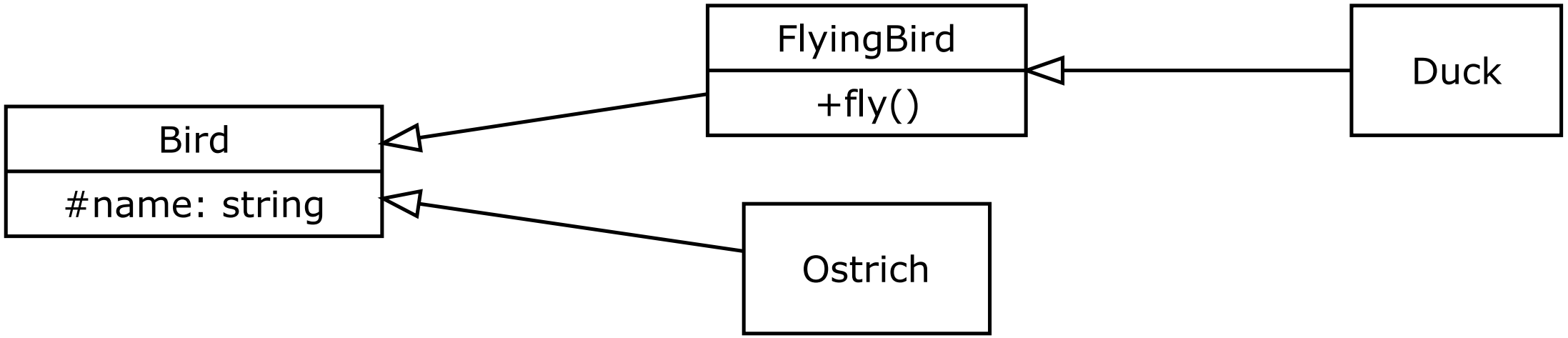
- A client should **NOT** be forced to depend on methods it does not use
- Having many client-specific interfaces is better than one general-purpose interface
- Why?
  - More cohesive
  - Lower coupling
  - Easier to understand
  - Easier to maintain

ISP Example – How can we make this better?



CREATED WITH YUML

ISP Example – How can we make this better?

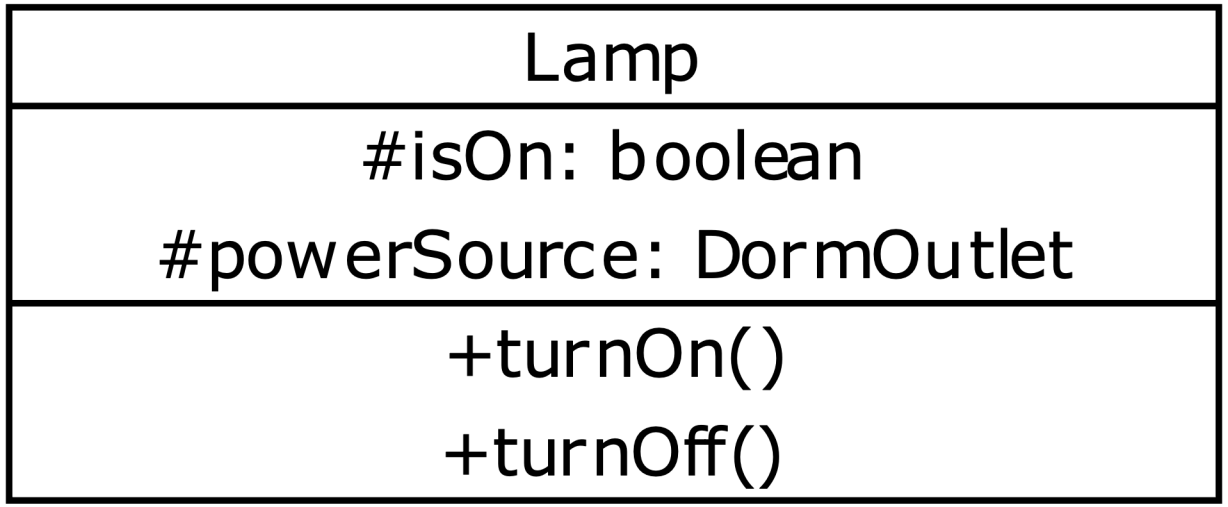


CREATED WITH YUML

Dependency Inversion Principle

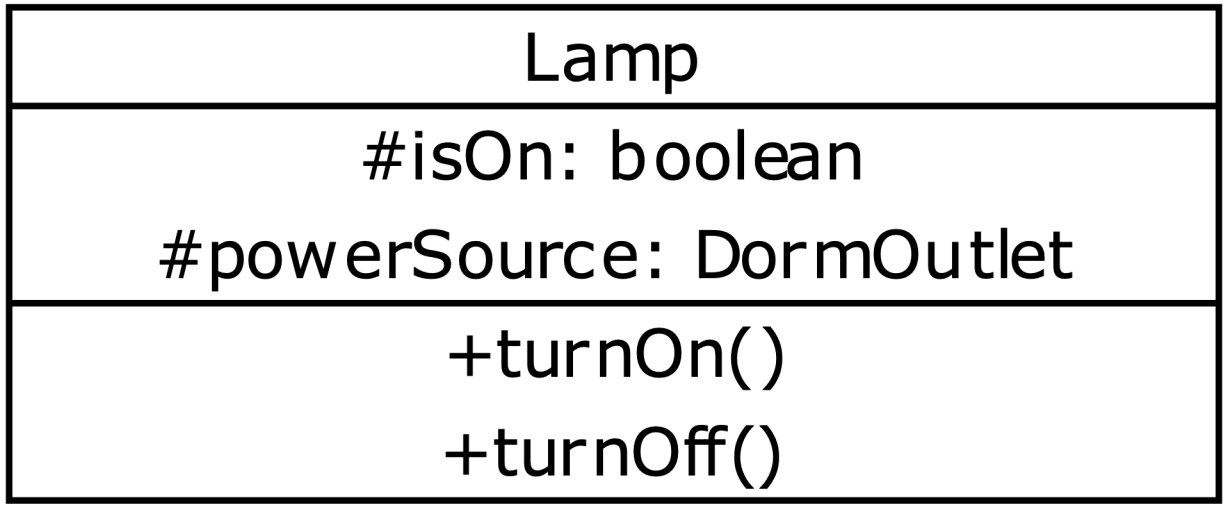
- Depend upon abstractions, not concretions (specific implementations of an abstraction)
- Abstractions should not depend on details, but details on abstractions
- High-level modules are independent and should not depend on low-level modules
- Why?
  - Lower coupling
  - Reuse
  - Easier to test
  - Easer to understand
  - Easier to maintain

DIP Example



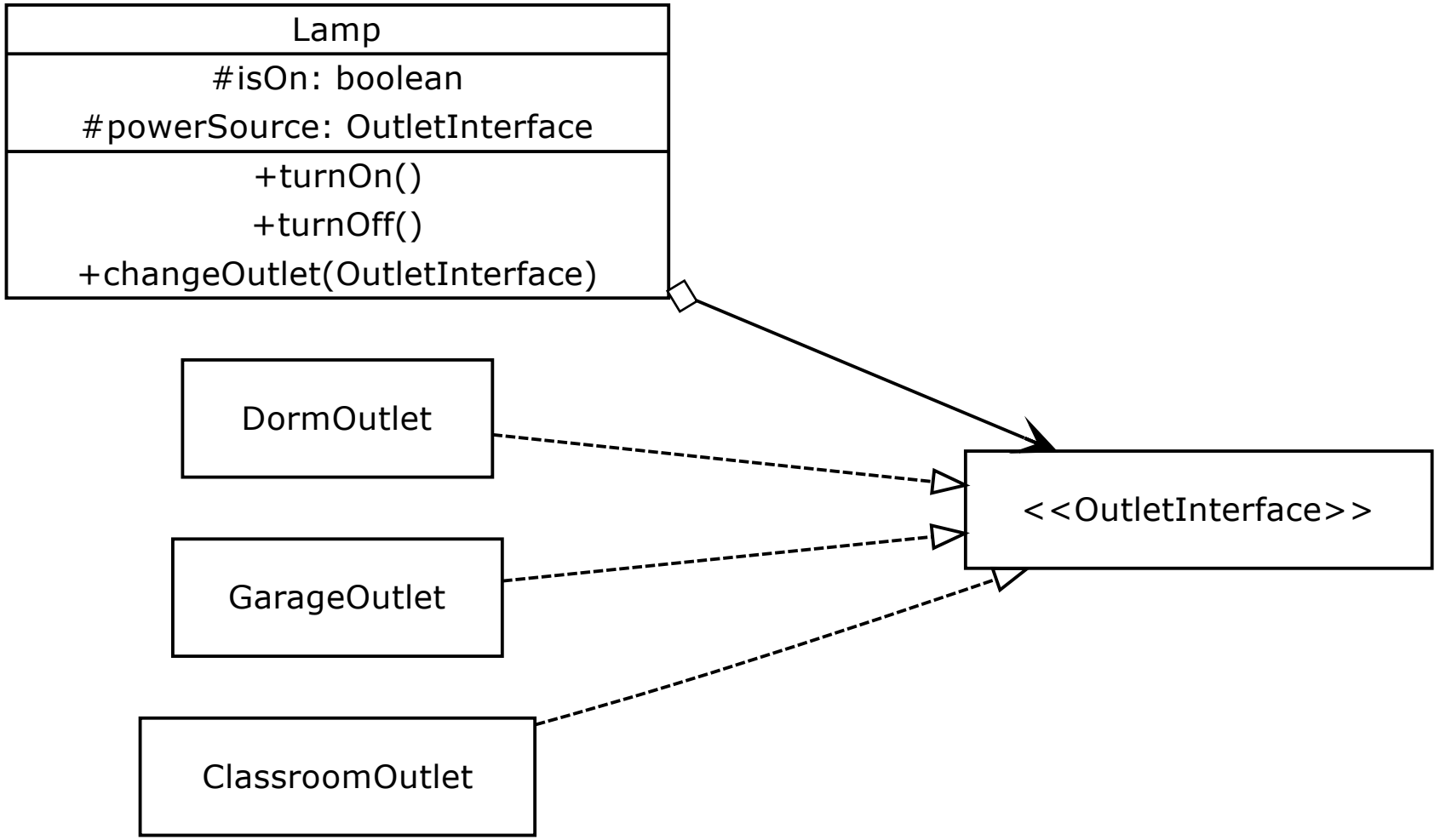
CREATED WITH YUML

DIP Example – Can We Do Better?



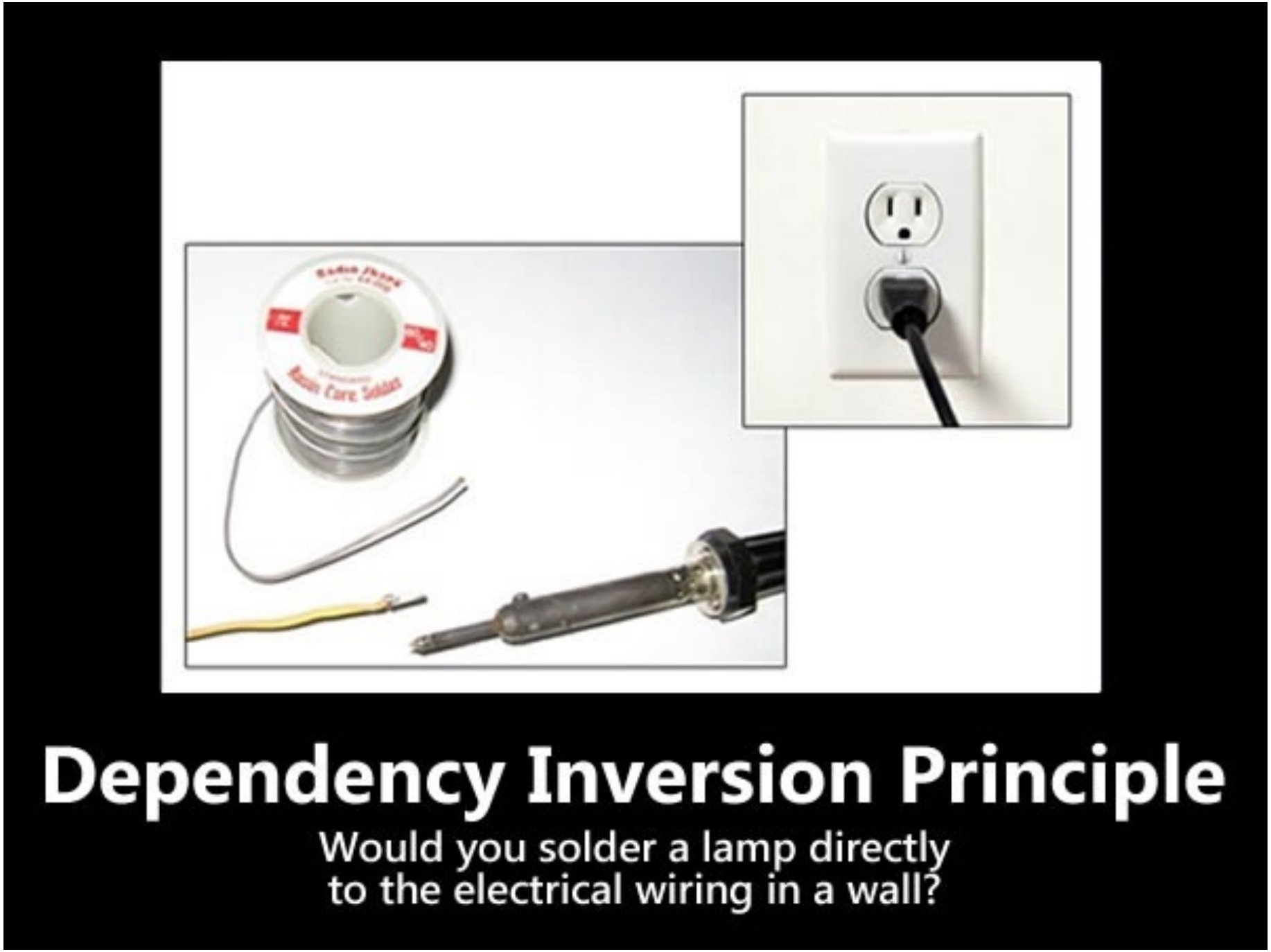
CREATED WITH YUML

DIP Example – Can We Do Better?



CREATED WITH YUML

https://medium.com/@kedren.villena/simplifying-dependency-inversion-principle-dip-59228122649a


Dependency Inversion Principle
Would you solder a lamp directly
to the electrical wiring in a wall?

Other Helpful Principles

- DRY- **D**on't **R**epeat **Y**ourself
  - Use functional decomposition or abstractions to reduce redundancies
- YAGNI – **Y**ou **A**ren't **G**onna **N**eed **I**t
  - Don't try to build out features now that you think your software **MIGHT** need later
  - Software development is too volatile for that, focus on what is needed now and the maintainability of your design
- Occam's Razor/KISS – Keep it simple
  - Don't introduce unnecessary complexity or overblown designs
- GRASP – **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
  - Design patterns that can help with your software design/implementation
  - More on design patterns later…

Conclusion

- Meant to be applied together
- Make it more likely that the system is easy to maintain and extend over time
- SOLID principles are guidelines
  - Do not guarantee success
  - Can be misused
- Use in conjunction with other principles
- Don’t chase perfection
  - Design based on your needs
  - Good enough design gets software delivered