# SOLID DESIGN PRINCIPLES

CSC207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

## Course Overview

**Tools (Weeks 1-4)**

- Java
- Version Control
- Software Tools

**Design (Weeks 5-8)**

- Clean Architecture
- SOLID
- Design Patterns

**Professional Topics (Weeks 9-12)**

- Ethics
- Internships
- GenAI

- Last week, we concluded our first 4-week block of the course on fundamental developer skills!
- This week, we begin our first of four weeks all about design! We'll be discussing:
  - the SOLID design principles
  - Exceptions

## Questions to be answered this week...

- What is a design principle?
- What are the SOLID principles?
- What is an Exception in Java?
- What is the difference between a "checked" and "unchecked" Exception? When should each be used?

## LEARNING OUTCOMES

- Know what the five SOLID design principles are.

## FUNDAMENTAL OOD PRINCIPLES

SOLID: five basic principles of object-oriented design

(Developed by Robert C. Martin, affectionately known as “Uncle Bob”.)

**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)

## SOLID (PRINCIPLES OF CLASS DESIGN)

| | | |
|---|---|---|
| SRP | The Single Responsibility Principle | A class should have one, and only one, reason to change. |
| OCP | The Open Closed Principle | You should be able to extend the behaviour of a class without modifying the class. |
| LSP | The Liskov Substitution Principle | Derived classes must be substitutable for their base classes. |
| ISP | The Interface Segregation Principle | Make fine grained interfaces that are client specific. |
| DIP | The Dependency Inversion Principle | Depend on abstractions, not on concretions. |

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

## THE SINGLE RESPONSIBILITY PRINCIPLE

## SINGLE RESPONSIBILITY PRINCIPLE

SRP

- Every class should have a single responsibility.
- Another way to view this is that **a class should only have one reason to change**.
- But who causes the change? An actor!

  Actor: a user of the program or a stakeholder, or a group of such people, or an automated process.

## SINGLE RESPONSIBILITY PRINCIPLE

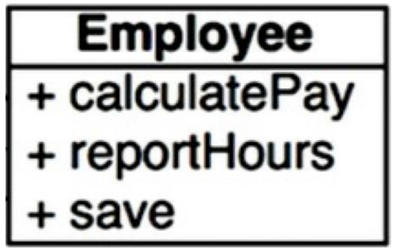
SRP

“This principle is about people. … When you write a software module, you want to make sure that when changes are requested, **those changes can only originate from a single person, or rather, a single tightly coupled group of people representing a single narrowly defined business function**. You want to **isolate your modules from the complexities of the organization as a whole**, and design your systems such that **each module is responsible (responds to) the needs of just that one business function**.”

[Uncle Bob, The Single Responsibility Principle]

## A STORY OF THREE ACTORS

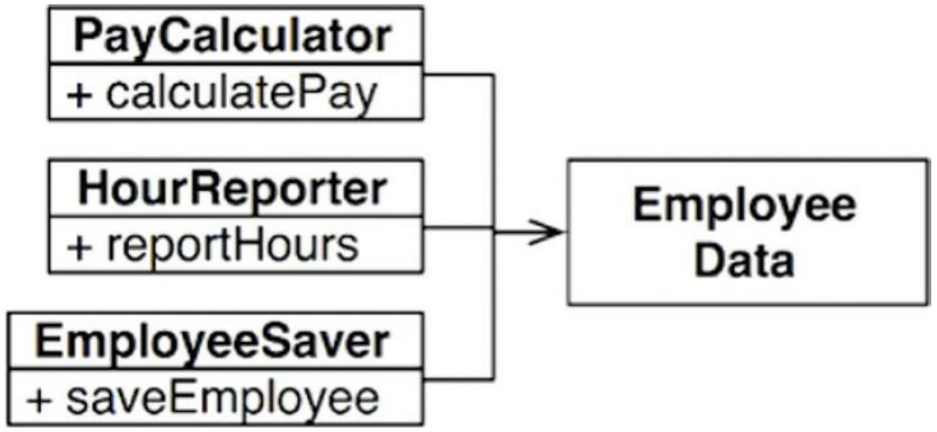
- Domain: an `Employee` class from a payroll application.
    - `calculatePay`: accounting department (CFO)
    - `reportHours`: human resources department (COO)
    - `save`: database administrators (CTO)



- Suppose methods `calculatePay` and `reportHours` share a helper method to calculate `regularHours` (and avoid duplicate code).
- CFO decides to change how non-overtime hours are calculated and a developer makes the change.
- The COO doesn't know about this. What happens?

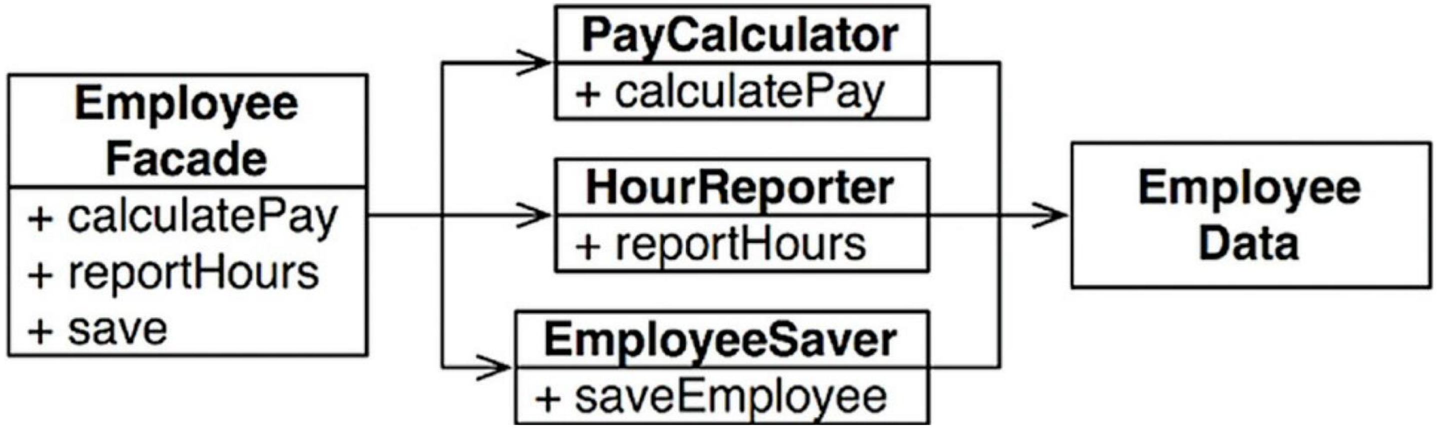
## CAUSE OF PROBLEM AND SOLUTION

- Cause of Problem: code is “owned” by more than one actor
- Solution: adhere to the Single Responsibility Principle
  - Factor out the data storage into an `EmployeeData` class.
  - Create three separate classes, one for each actor.



## FAÇADE DESIGN PATTERN

- Downside of solution: need to keep track of three objects, not one.
- Solution: create a façade ("the front of a building").
  - Very little code in the façade. Delegates to the three classes.



- We'll talk about the Façade design pattern and many more design patterns throughout the term.

## THE OPEN / CLOSED PRINCIPLE

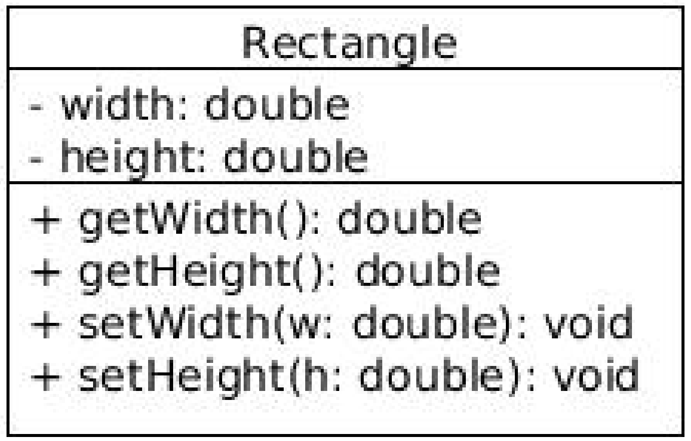
## OPEN/CLOSED PRINCIPLE

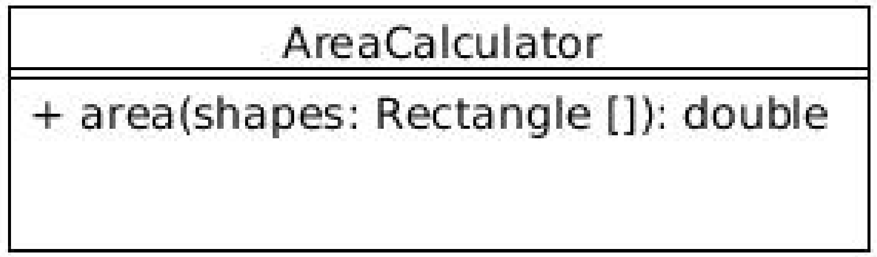
OCP

- Software entities (classes, modules, functions, etc.) should be **open for extension, but closed for modification**.
- Add new features not by modifying the original class, but rather by extending it and adding new behaviours, or by adding plugin capabilities.
- “I’ve heard it said that the OCP is wrong, unworkable, impractical, and not for real programmers with real work to do. The rise of plugin architectures makes it plain that these views are utter nonsense. On the contrary, a strong plugin architecture is likely to be the most important aspect of future software systems.” [Uncle Bob, The Open Closed Principle]

## OPEN/CLOSED PRINCIPLE

- An example using inheritance
- The `area` method calculates the area of all Rectangles in the given array.
- What if we need to add more shapes?

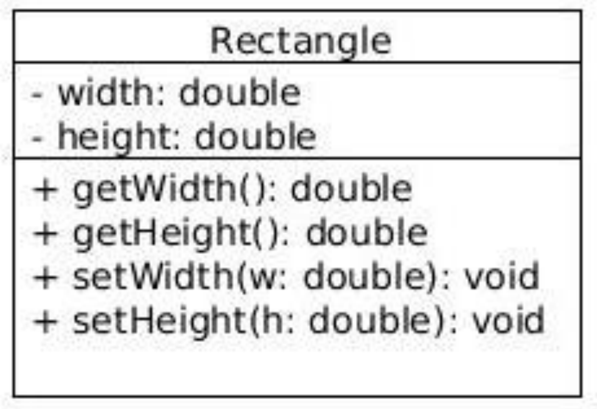


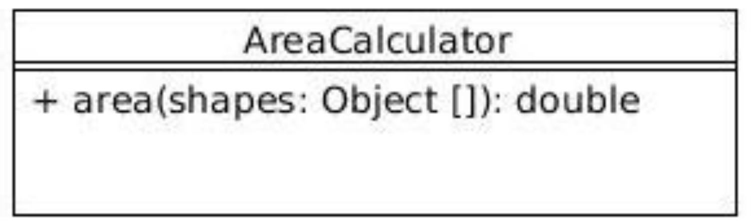


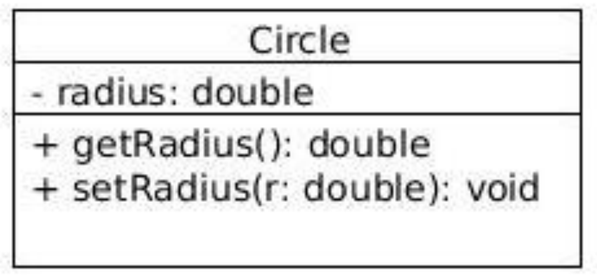
OCP

## OPEN/CLOSED PRINCIPLE

- We might make it work for circles too.
- We could implement a `Circle` class and **rewrite** the `area` method to take in an **array of Objects** (using `isinstance` to determine if each `Object` is a `Rectangle` or a `Circle` so it can be cast appropriately).







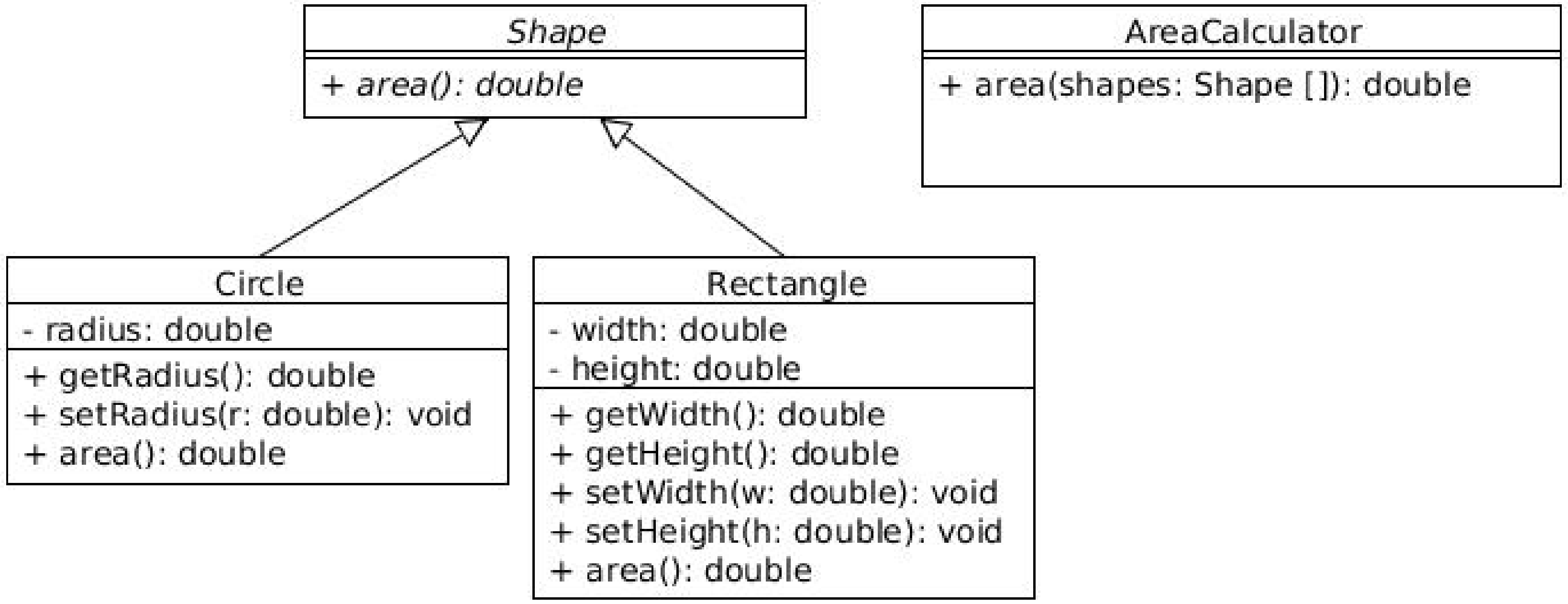
- But what if we need to add even more shapes?

Computer Science
UNIVERSITY OF TORONTO

## OPEN/CLOSED PRINCIPLE

OCP

- With this design, we can add any number of shapes (open for extension) and we don't need to re-write the `AreaCalculator` class (closed for modification).



## THE LISKOV SUBSTITUTION PRINCIPLE

## LISKOV SUBSTITUTION PRINCIPLE

- If S is a subtype of T, then objects of type S may be substituted for objects of type T, without altering any of the desired properties of the program.
- "S is a subtype of T"?

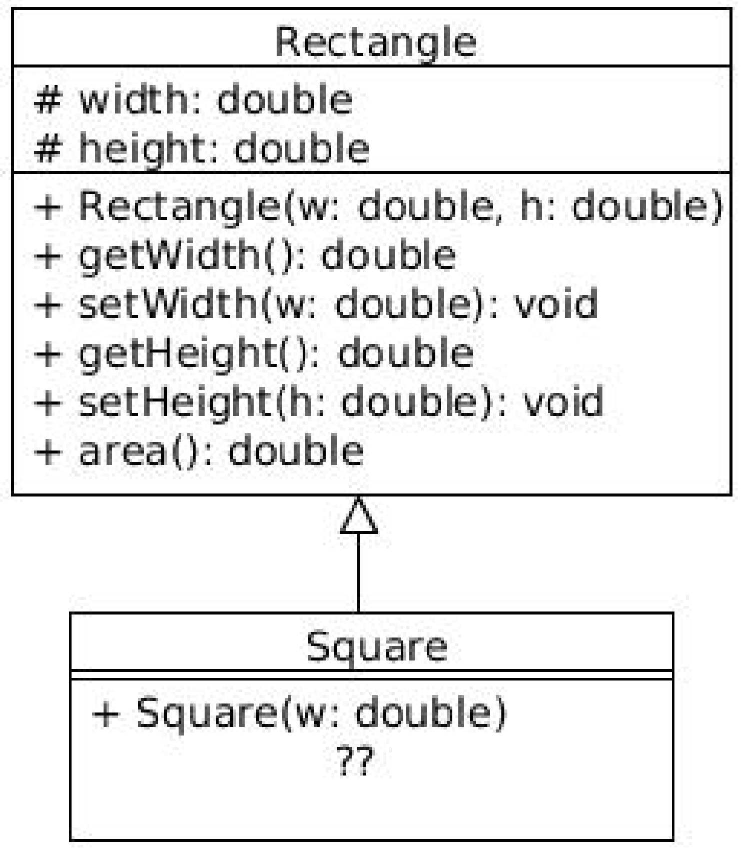
  In Java, this means that S is a child class of T, or S *implements* interface T.

- "A program that uses an interface must not be confused by an implementation of that interface." [Uncle Bob]

## LISKOV SUBSTITUTION PRINCIPLE

Example

- Mathematically, a square “is a” rectangle.
- In object-oriented design, it is not the case that a `Square` “is a” `Rectangle`!
- This is because a `Rectangle` has *more* behaviours than a `Square`, not less.
- The LSP is related to the Open/Closed principle: the subclasses should only extend (add behaviours), not modify or remove them.



## THE INTERFACE SEGREGATION PRINCIPLE

ISP

## INTERFACE SEGREGATION PRINCIPLE

- Here, interface means the public methods of a class. (In Java, these are often specified by defining an interface, which other classes then implement.)
- Context: a class that provides a service for other “client” programmers usually requires that the clients write code that has a particular set of features. The service provider says “your code needs to have this interface”.
- **No client should be forced to implement irrelevant methods of an interface. Better to have lots of small, specific interfaces than fewer larger ones: easier to extend and modify the design.**
- (Uh oh: “The interface keyword is harmful.” [Uncle Bob, 'Interface' Considered Harmful] — we encourage you to read this and discuss with others. Does the fact that Java supports “default methods” for interfaces change anything?)

## INTERFACE SEGREGATION PRINCIPLE

ISP

**“Keep interfaces small so that users don’t end up depending on things they don’t need.**

We still work with compiled languages. We still depend upon modification dates to determine which modules should be recompiled and redeployed. So long as this is true we will have to face the problem that when module A depends on module B at compile time, but not at run time, then changes to module B will force recompilation and redeployment of module A.” [ Uncle Bob, SOLID Relevance ]

## THE DEPENDENCY INVERSION PRINCIPLE

## DEPENDENCY INVERSION PRINCIPLE

- When building a complex system, programmers are often tempted to define “low-level” classes first and then build “higher-level” classes that use the low-level classes directly.
- But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced — an indication of high coupling.
- To avoid such problems, we introduce an **abstraction layer** between low-level classes and high-level classes.

# DEPENDENCY INVERSION PRINCIPLE

DIP

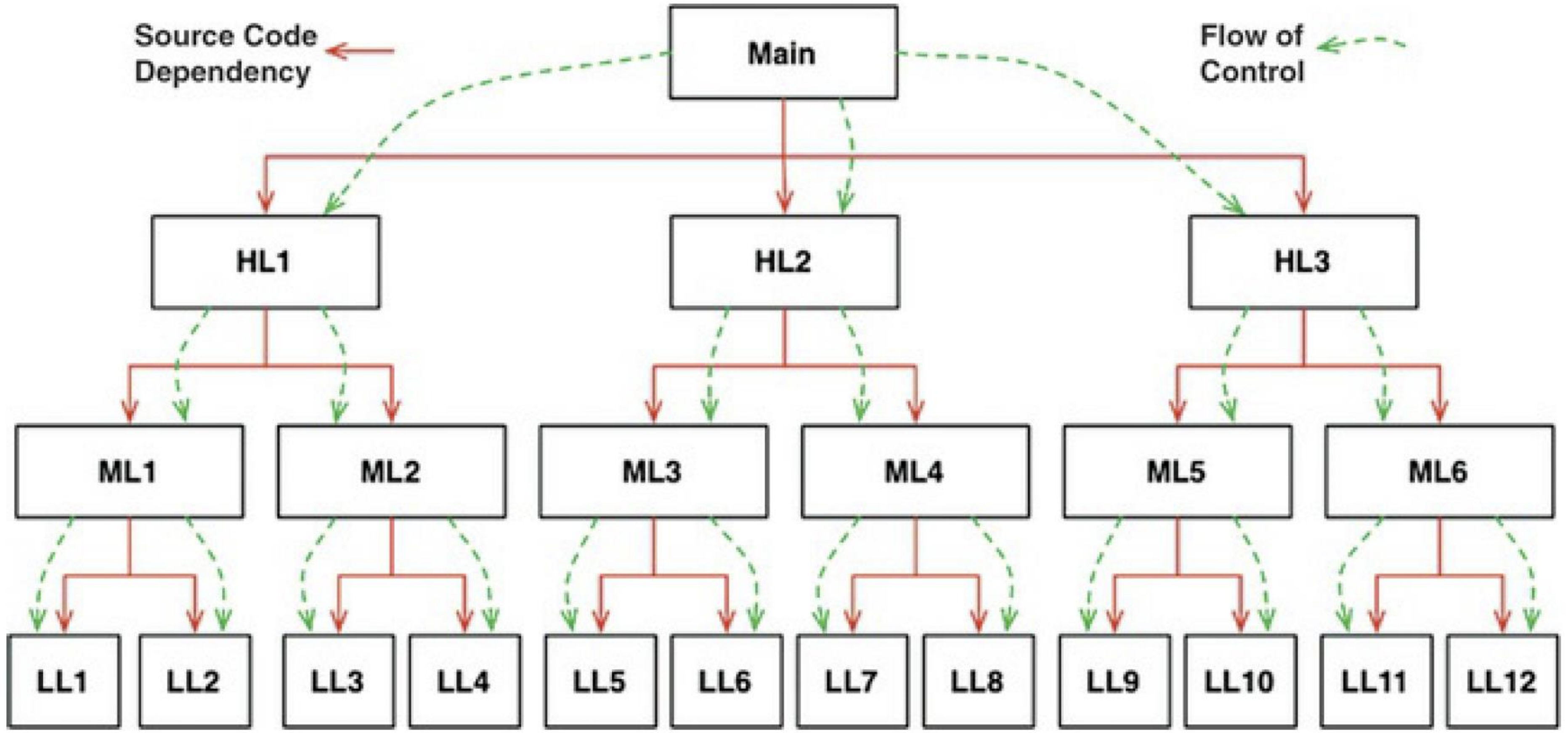


**Figure 5.1** Source code dependencies versus flow of control

Clean Architecture, Robert C. Martin

## DEPENDENCY INVERSION PRINCIPLE

DIP

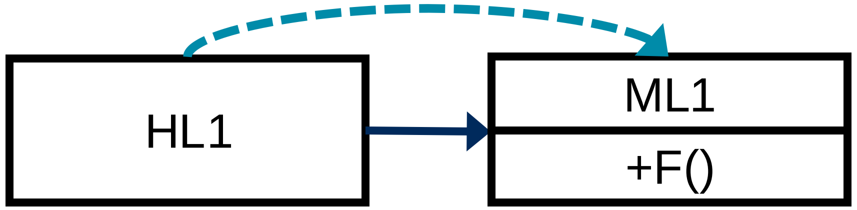
Goal:

- You want to decouple your system so that you can change individual pieces without having to change anything more than the individual piece.
- Two aspects to the dependency inversion principle:
    - High-level modules should not depend on low-level modules. Both should depend on abstractions.
    - Abstractions should not depend upon details. Details should depend upon abstractions.

# DEPENDENCY INVERSION PRINCIPLE

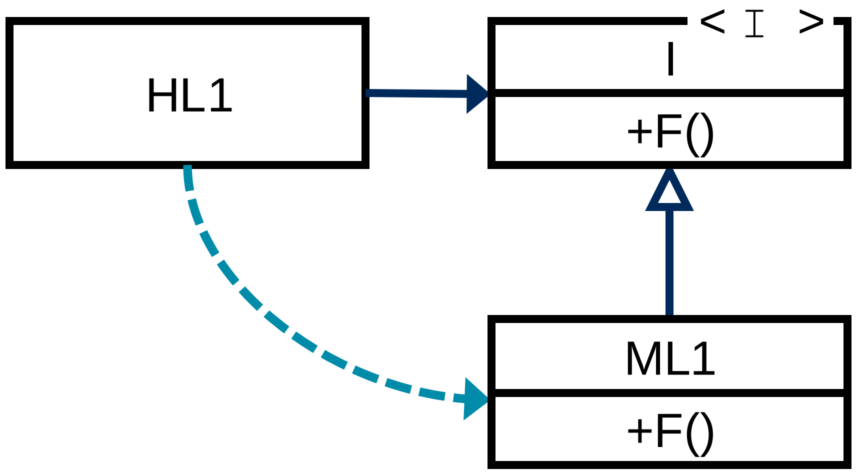
DIP

How do we invert the source code dependency?



We introduce an interface!

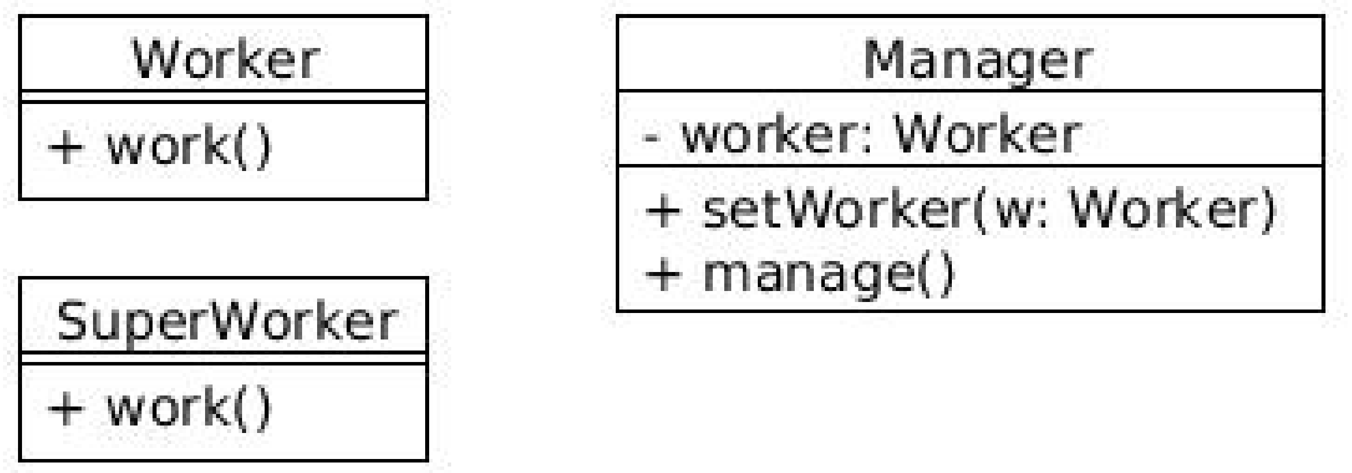
- The flow of control remains the same.
- HL1 depends on the interface and ML1 implements that interface.
- There is no longer a source code dependency between HL1 and ML1!



## EXAMPLE FROM DEPENDENCY INVERSION PRINCIPLE ON OODESIGN

DIP

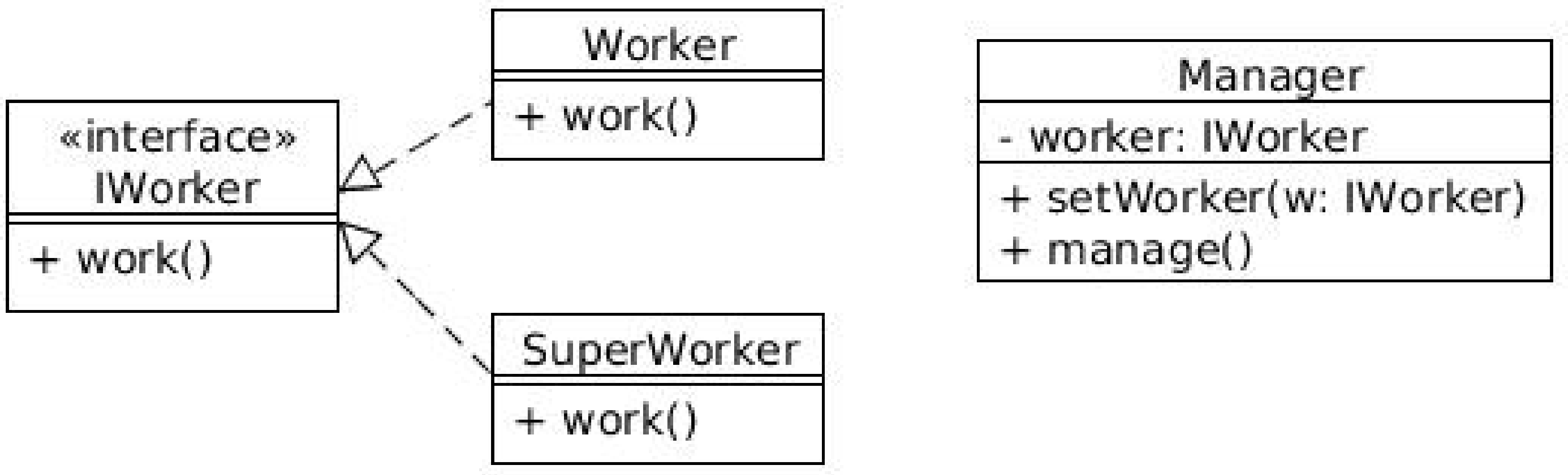
- A company is structured with managers and workers. The code representing the company's structure has Managers that manage Workers. Let's say that the company is restructuring and introducing new kinds of workers. They want the code updated to reflect this change.
- Your code currently has a Manager class and a Worker class. The Manager class has one or more methods that take Worker instances as parameters.
- Now there's a new kind of worker called SuperWorker whose behaviour and features are separate from regular Workers, but they both have some notion of "doing work".



Computer Science
UNIVERSITY OF TORONTO

# EXAMPLE FROM DEPENDENCY INVERSION PRINCIPLE ON OODESIGN

- To make Manager work with SuperWorker, we would need to rewrite the code in Manager (e.g. add another attribute to store a SuperWorker instance, add another setter, and update the body of manage())
- Solution: create an IWorker interface and have Manager depend on it instead of directly depending on the Worker and SuperWorker classes.
- In this design, Manager does not know anything about Worker, nor about SuperWorker. The code will work with any class implementing the IWorker interface and the code in Manager does not need to be rewritten.


«interface»
IWorker
+ work()
Worker
+ work()
SuperWorker
+ work()
Manager
- worker: IWorker
+ setWorker(w: IWorker)
+ manage()