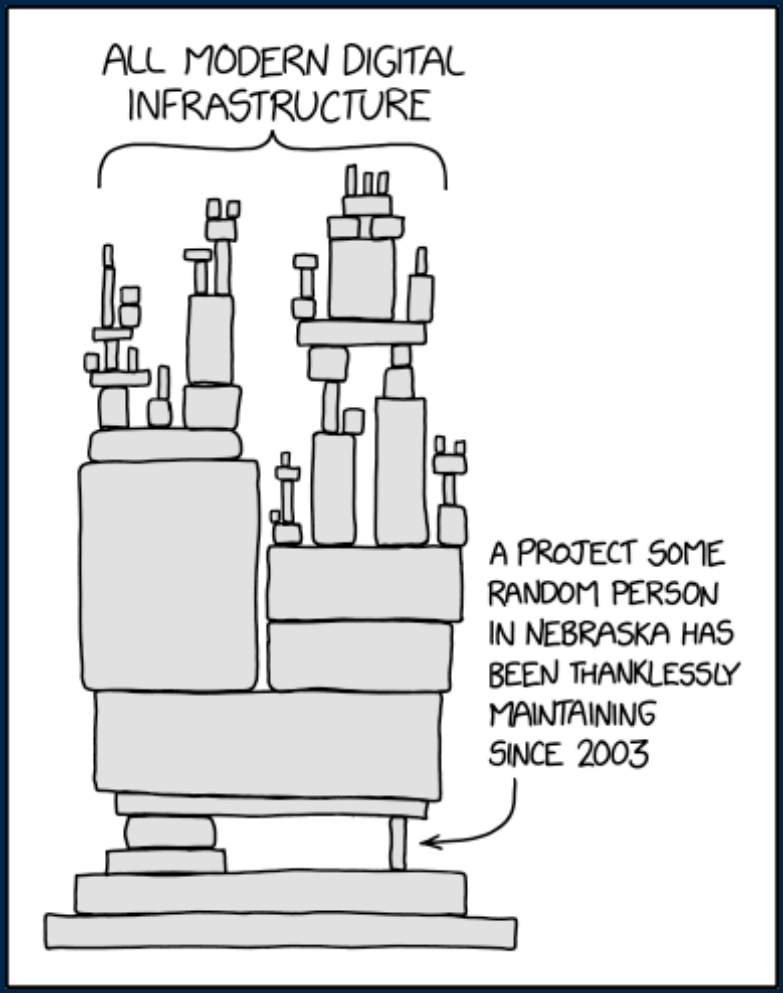
SOLID

"The critical design tool for software development is a mind well educated in design principles"



Ref: https://xkcd.com/2347/

THE SOLID ACRONYM

- The SOLID acronym is composed of the first letters of 5 design principles:
  - **S** Single Responsibility Principle (SRP)
  - **O** Open Closed Principle (OCP)
  - **L** Liskov's Substitution Principle (LSP)
  - **I** Interface Segregation Principle (ISP)
  - **D** Dependency Inversion Principle (DIP)

INTERFACE SEGREGATION PRINCIPLE (ISP)

I want...

I got

INTERFACE SEGREGATION PRINCIPLE

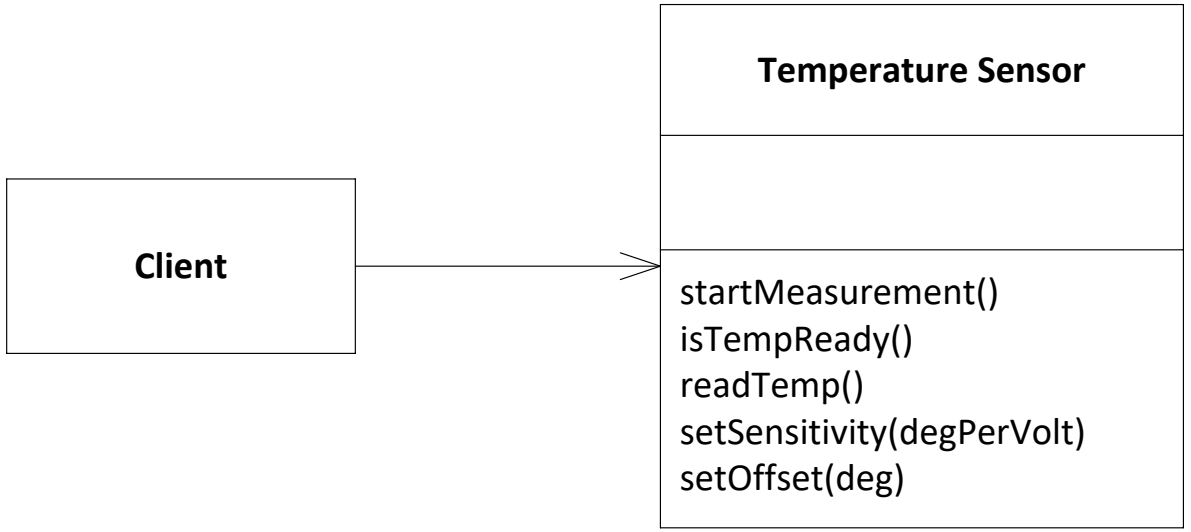
> "Clients should not be forced to depend on methods they do not use"

- ISP requests that interfaces are kept small and cohesive. This yields at least two advantages:
  - Implementors of an interface do not need to implement "dummy" methods for the parts of an interface which they do not implement.
  - Consumers of an interface do not have to consider methods that they do not need.

Note how the word "client" of an interface is used for both "consumer" and "implementor" in the literature, causing confusion

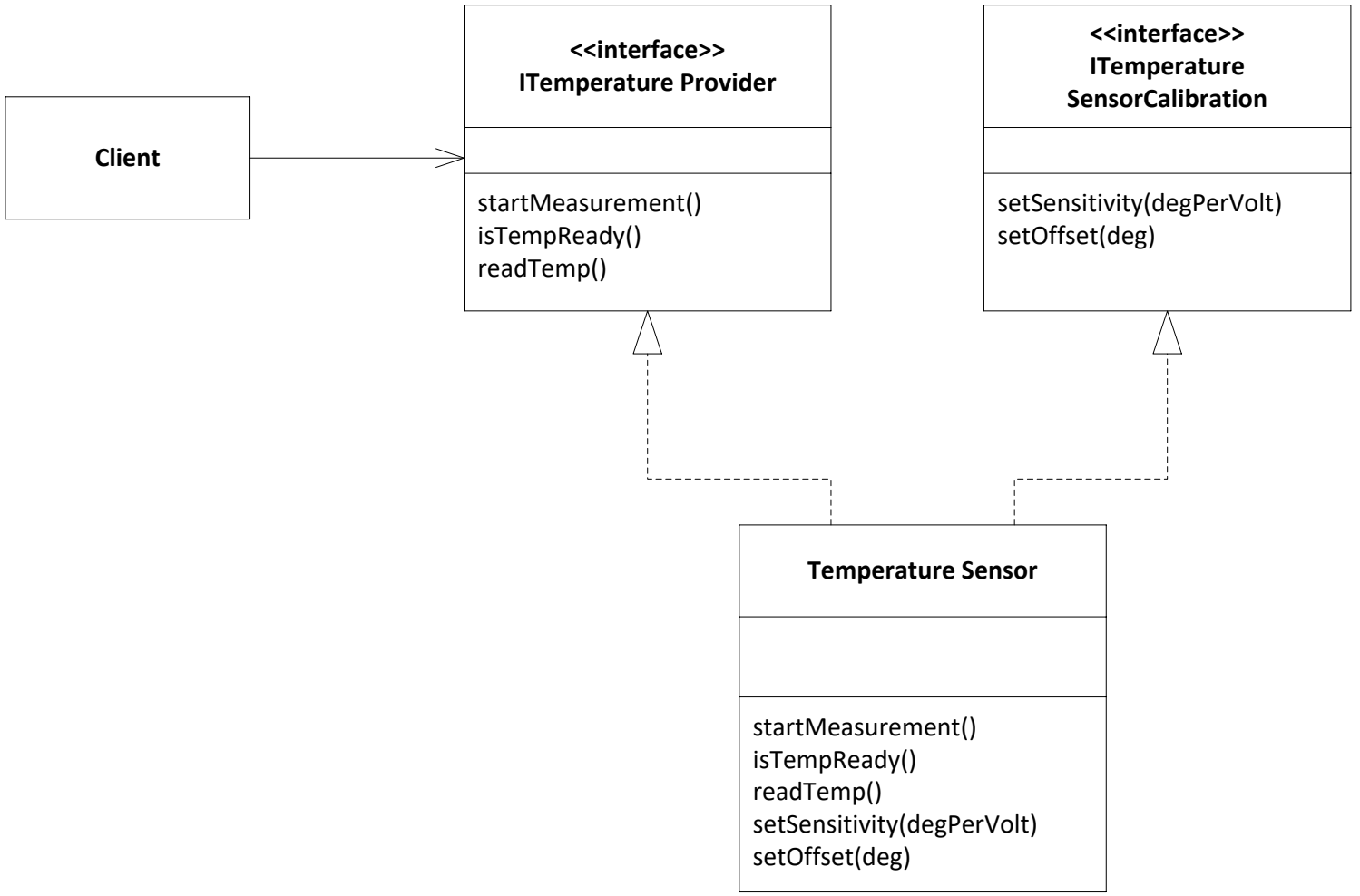
ISP VIOLATION (CONSUMER?)

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

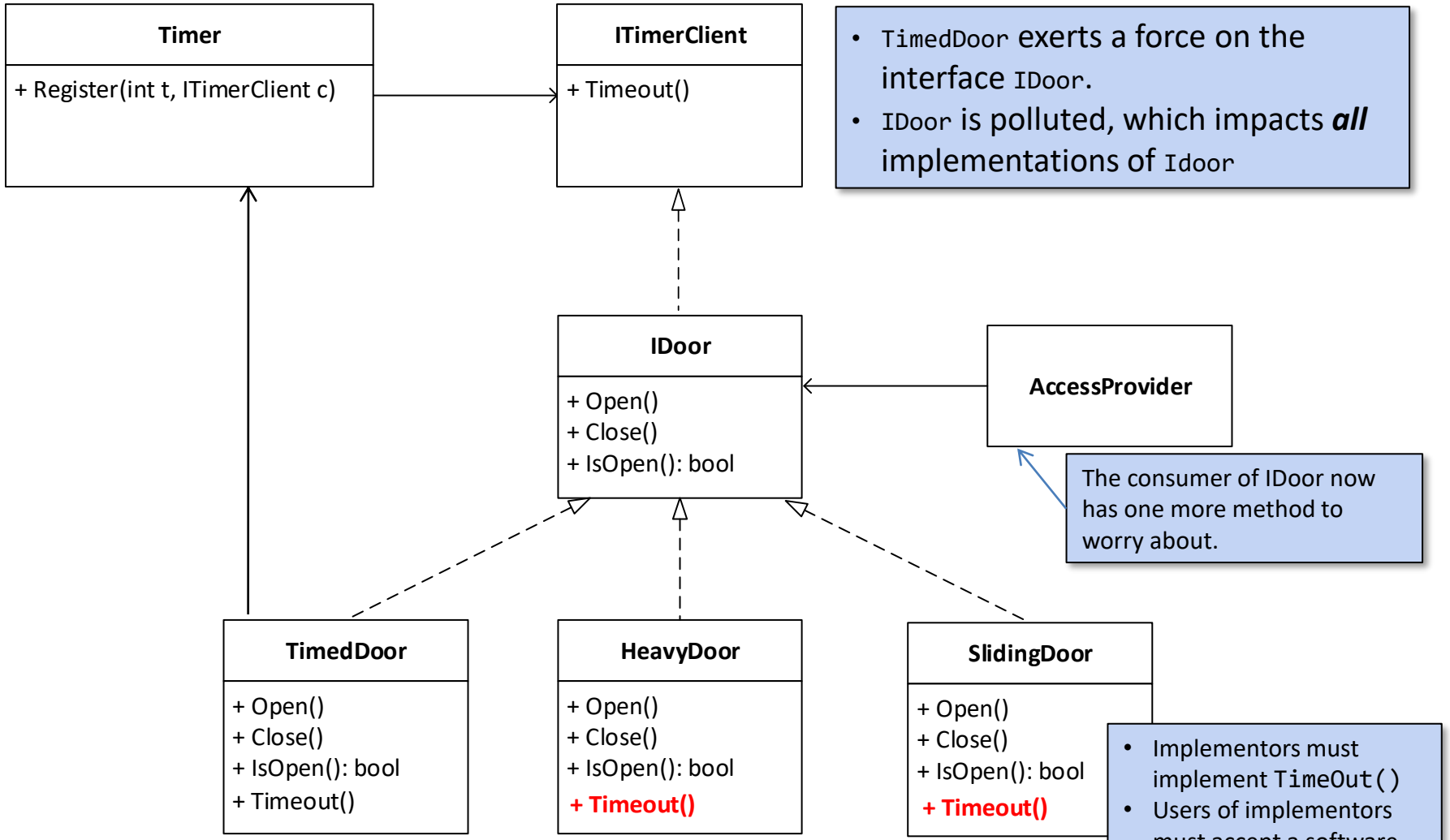


What is the problem?

ISP (CONSUMERS)



ISP VIOLATION (IMPLEMENTORS)

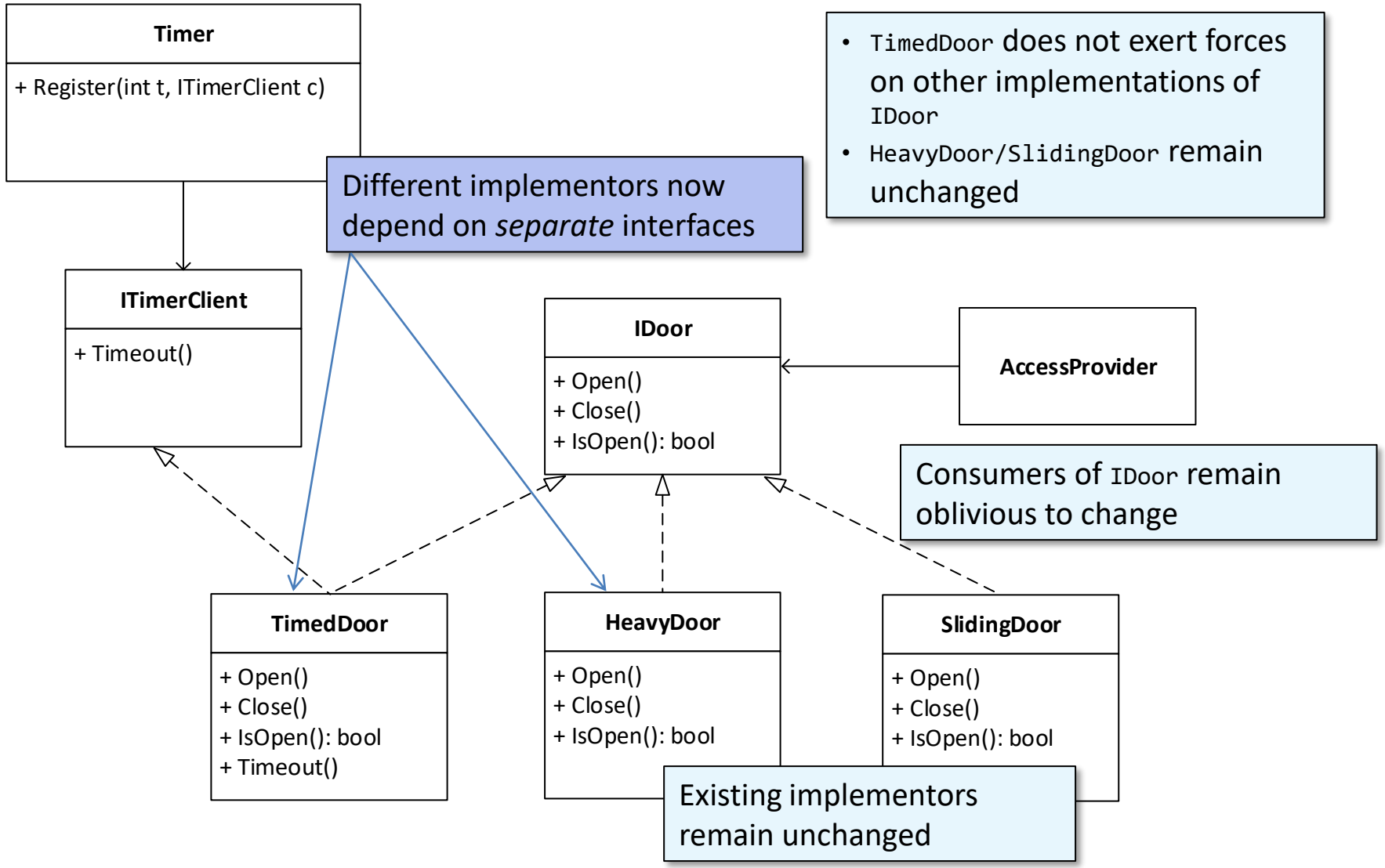


- `TimedDoor` exerts a force on the interface `IDoor`.
- `IDoor` is polluted, which impacts ***all*** implementations of `Idoor`

The consumer of IDoor now has one more method to worry about.

- Implementors must implement `TimeOut()`
- Users of implementors must accept a software update for *zero* benefit

ISP (IMPLEMENTORS)



ISP IN THE REAL WORLD

- C#'s List<T>
  - List<T> API
- Implements a number of interfaces
  - ICollection<T>
  - IEnumerable<T>
  - IList<T>
  - IReadOnlyCollection<T>
  - IReadOnlyList<T>
  - ICollection
  - IEnumerable
  - IList

DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

DEPENDENCY INVERSION PRINCIBLE (DIP)

Dependency Inversion Principle (DIP):

- *"A: High-level modules should not depend on low-level modules. Both should depend on abstractions."*
- *"B: Abstractions should not depend on details. Details should depend on abstractions"*

DIP

- *High-level modules* should not depend on *low-level modules*. Both should depend on *abstractions*.
  - *High-level modules* are abstract from details (of e.g. communication, hardware, etc) and contain policies, business models, etc.
  - *Low-level modules*, e.g. drivers, know about the details of HW, communication etc., but know nothing about high-level concepts e.g. business or policies.

An example...

DIP EXAMPLES - ECS

- An *Environmental Control System (ECS)* is installed in a greenhouse

- The ECS monitors the temperature in the greenhouse
  - $T > T_{max}$ → open windows, stop heater
  - $T_{min} < T < T_{max}$ → close windows, stop heater
  - $T < T_{min}$ → close windows, start heater

DIP EXAMPLE - ECS

```
class ECS
{
 ...
 void Main(string args[])
 {
  while(true)
  {
    curTemp = in(TEMP_SENSOR_DATA_ADDR);
    switch(curTemp):
    {
      case curTemp > MAX_TEMP:
        out(WINACT_CMD_ADDR, 1);      // opens window
        out(HEATER_CMD_ADDR, 0x00FF); // stops heater
        break;
      case curTemp < MIN_TEMP:
        out(WINACT_CMD_ADDR, 0);      // closes window
        out(HEATER_CMD_ADDR, 0xFF00); // starts heater
        break;
      default:
        out(WINACT_CMD_ADDR, 0);      // closes window
        out(HEATER_CMD_ADDR, 0x00FF); // stops heater
        break;
    }
    Thread.Sleep(10000);// Sleep 10s before next regulation
  }
 }
}
```

What are the dependencies here?

Concrete temperature sensor HW

Concrete window actuator / heater HW

OS' I/O system

OS' scheduling system

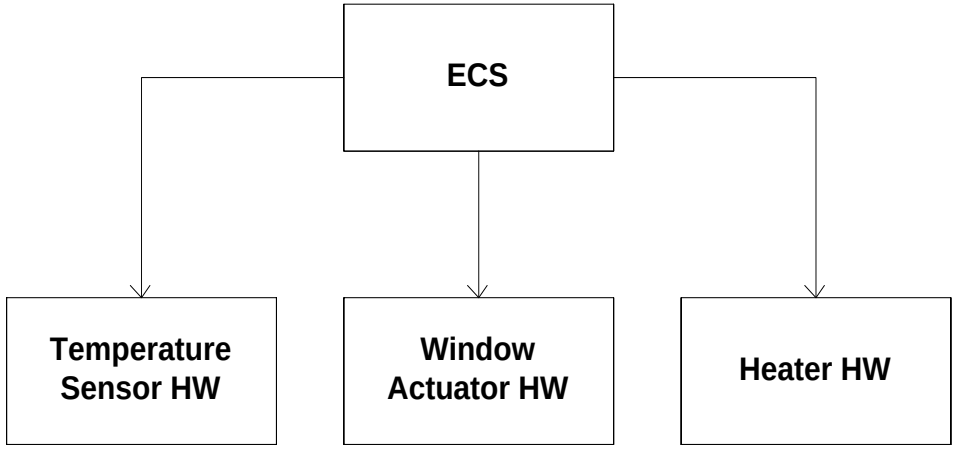
DIP EXAMPLE - ECS

- What are high-level and low-level modules?
  - High-level: ECS, Temperature controller
  - Low-level: Heater, window, sensor/actuator, OS
- The *high-level module* (ECS) implement the *policy* of how to regulate temperature
- …but it depends on *low-level implementations* to do it

```
class ECS
{
 ...
 void RegulateTemp()
 {
  while(true)
  {
    curTemp = in(TEMP_SENSOR_DATA_ADDR);
    switch(curTemp):
    {
      case curTemp > MAX_TEMP:
        out(WINACT_CMD_ADDR, 1);      // opens window
        out(HEATER_CMD_ADDR, 0x00FF); // stops heater
        break;
      case curTemp < MIN_TEMP:
        out(WINACT_CMD_ADDR, 0);      // closes window
        out(HEATER_CMD_ADDR, 0xFF00); // starts heater
        break;
      default:
        out(WINACT_CMD_ADDR, 0);      // closes window
        out(HEATER_CMD_ADDR, 0x00FF); // stops heater
        break;
    }
    Thread.Sleep(10000);// Sleep 10s before next regulation
  }
 }
}
```

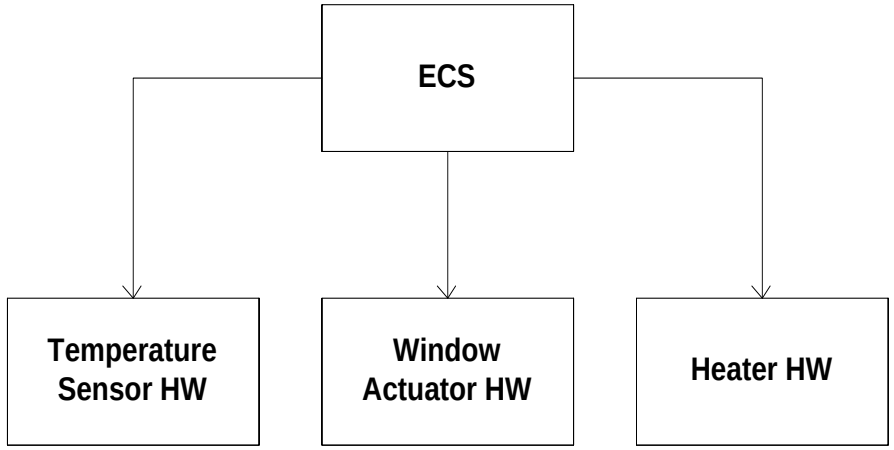
DIP EXAMPLE - ECS

The problem is that the dependencies are *inverted* – high-level modules depend on low-level ones. DIP tells us not to do that.



So…how do we avoid this situation? We apply DIP!

DIP EXAMPLE - ECS



Dependency Inversion Principle (DIP):

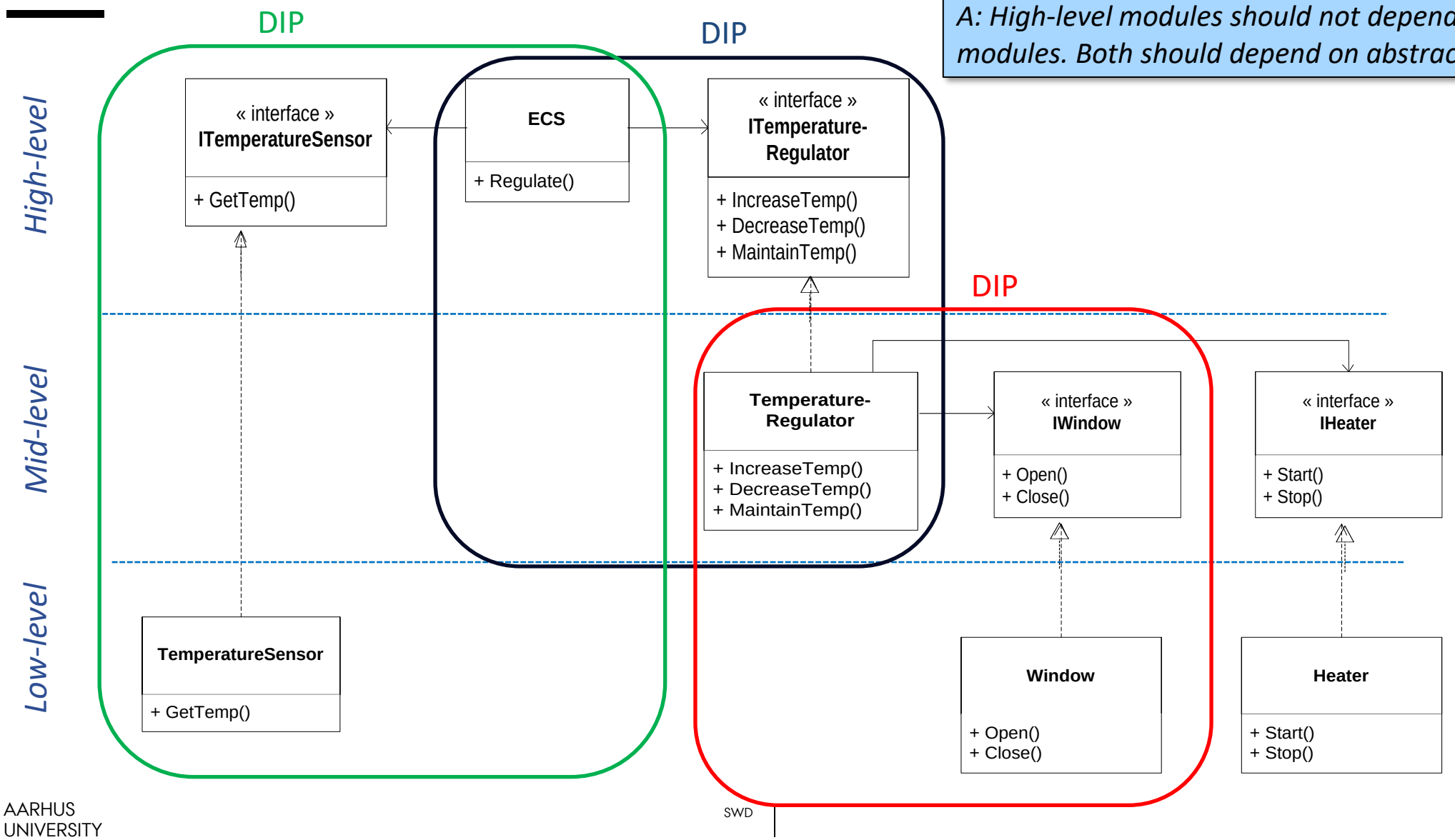
- *"A: High-level modules should not depend on low-level modules. Both should depend on abstractions."*
- *"B: Abstractions should not depend on details. Details should depend on abstractions"*

- We want the high-level *and* the low-level module to both depend on abstractions.

- In this way, the high-level modules can care about *what* to do, not *how* to do it.

DIP EXAMPLE - ECS

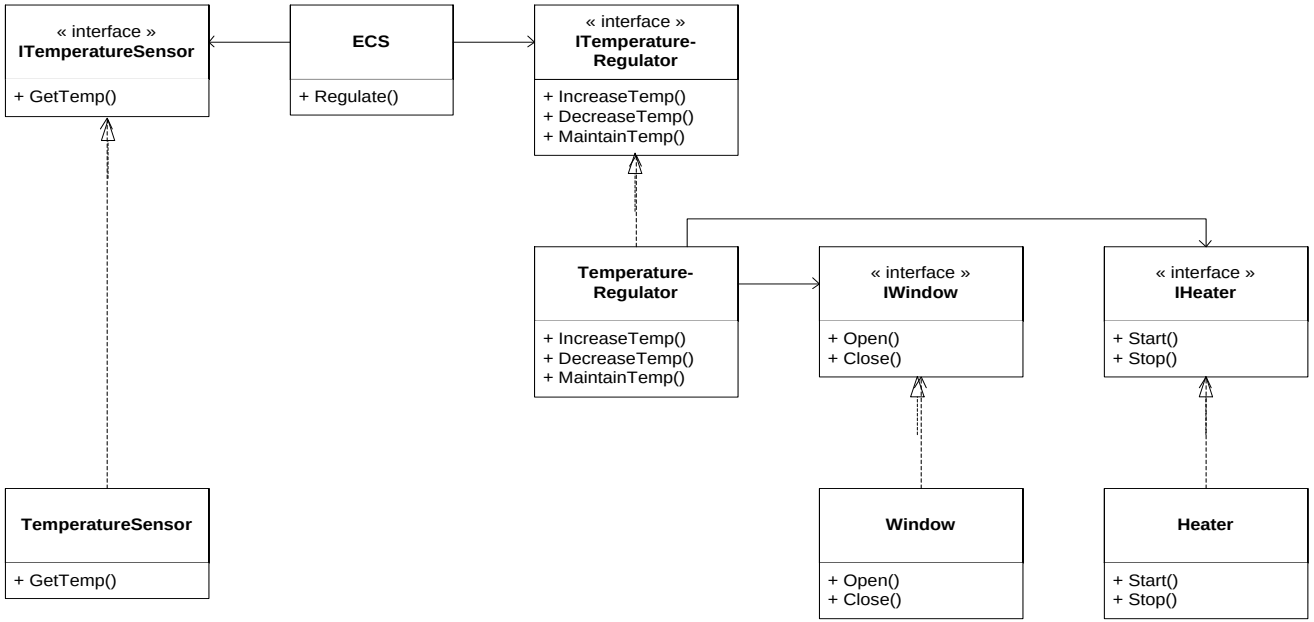
A: High-level modules should not depend on low-level modules. Both should depend on abstractions.



AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DIP EXAMPLE - ECS

VS?




```
class ECS
{
 ...
 void RegulateTemp()
 {
  while(true)
  {
    curTemp = in(TEMP_SENSOR_DATA_ADDR);
    switch(curTemp):
    {
      case curTemp > MAX_TEMP:
        out(WINACT_CMD_ADDR, 1);      // opens window
        out(HEATER_CMD_ADDR, 0x00FF); // stops heater
        break;
      case curTemp < MIN_TEMP:
        out(WINACT_CMD_ADDR, 0);      // closes window
        out(HEATER_CMD_ADDR, 0xFF00); // starts heater
        break;
      default:
        out(WINACT_CMD_ADDR, 0);      // closes window
        out(HEATER_CMD_ADDR, 0x00FF); // stops heater
        break;
    }
    Thread.Sleep(10000);// Sleep 10s before next regulation
  }
 }
}
```

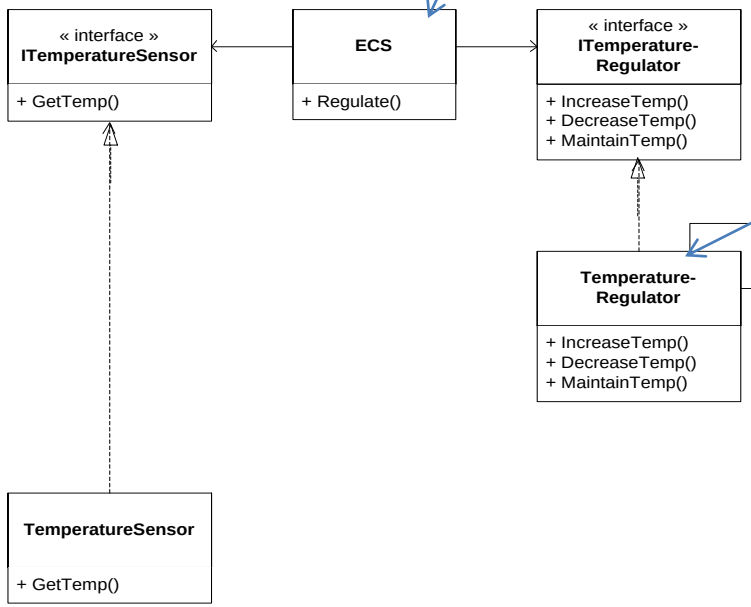
DIP EXAMPLE - ECS

```
void Regulate()
 {
  while(true)
  {
    curTemp = tempSensor.GetTemp();
    switch(curTemp):
    {
      case curTemp > MAX_TEMP: tempRegulator.DecreaseTemp();
                               break;
      case curTemp < MIN_TEMP: tempRegulator.IncreaseTemp();
                               break;
      default:                   tempRegulator.MaintainTemp();
                               break;
    }
    Thread.Sleep(10000);
  }
 }
```

```
void IncreaseTemp()
{
  Window.Close();
  Heater.Start();
}

void DecreaseTemp()
{
  Window.Open();
  Heater.Stop();
}

void MaintainTemp()
{
  Window.Close();
  Heater.Stop();
}
```



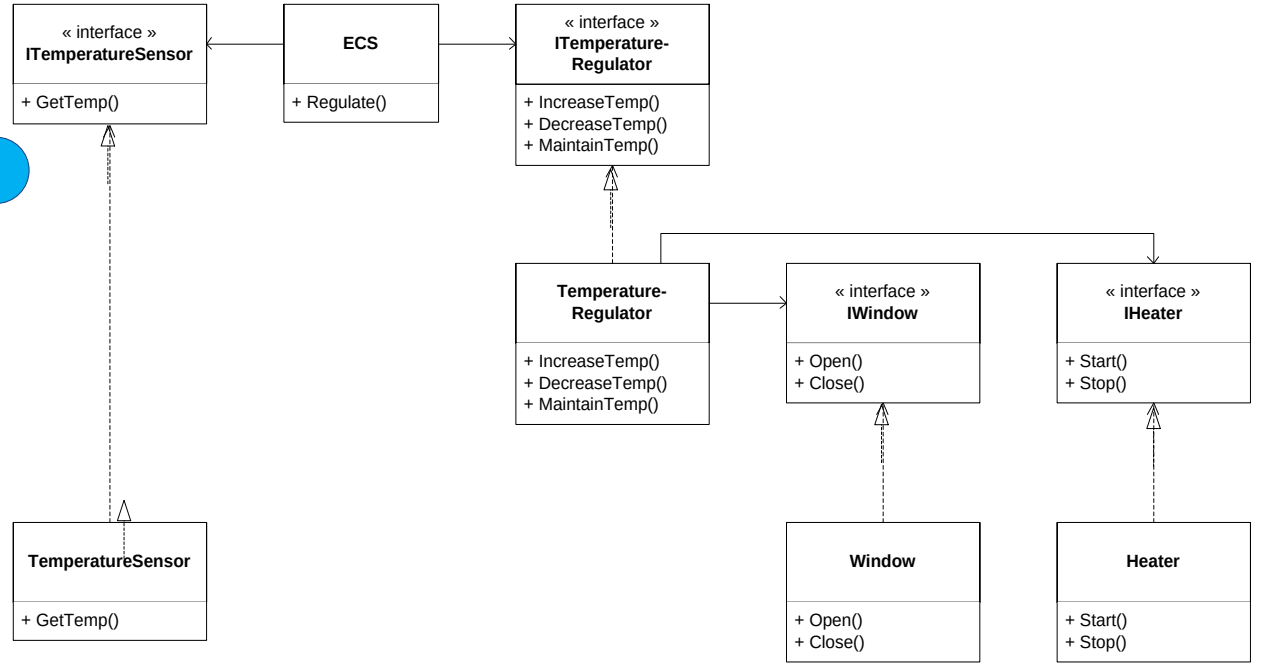

```
void Open() { out(WINACT_CMD_ADDR, 1); }
void Close(){ out(WINACT_CMD_ADDR, 0); }
```

```
void Start()
{
  out(HEATER_CMD_ADDR, 0x00FF);
}

void Stop()
{
  out(HEATER_CMD_ADDR, 0xFF00);
}
```

DIP EXAMPLE - ECS

- Discuss:
  1. What would be the impact of changing the temperature sensor, window actuator or heater hardware
  2. How is the wording in DIP manifest in this design?



Dependency Inversion Principle (DIP):

- *"A: High-level modules should not depend on low-level modules. Both should depend on abstractions."*
- *"B: Abstractions should not depend on details. Details should depend on abstractions"*

QUESTIONS

AARHUS
UNIVERSITY

REFERENCE

Frontpage: https://xkcd.com/2347/

Dilbert: https://dilbert.com/strip/1996-05-03

DIP: https://www.abhishekshukla.com/net-2/dependency-inversion-principle-dip/