Design Principles

SOLID Design Principles, Part 2

SOLID: Interface Segregation Principle

Design Principle:
Interface Segregation Principle

Many client-specific interfaces are better than one general purpose interface. Clients should not be forced to depend upon interfaces that they do not use.

SOLID: Interface Segregation Principle

- A “fat interface” is supplied by a class whose interface is not cohesive
  - It has many responsibilities and is unfocused and hard to understand/modify
- The Interface Segregation Principle seeks to avoid fat interfaces
  - Some objects may require non-cohesive interfaces
  - Clients should not know about them as a single class
  - Clients should know about abstract base classes with cohesive interfaces

SOLID: Interface Segregation Principle

- Suppose we are implementing a security system
- We start with an abstract class Door:

```
class Door {
   public:
      virtual void lock()   = 0;
      virtual void unlock() = 0;
      virtual bool isDoorOpen() = 0;
};
```

pure virtual methods.

Any subclass of door have to implement these methods.

SOLID: Interface Segregation Principle

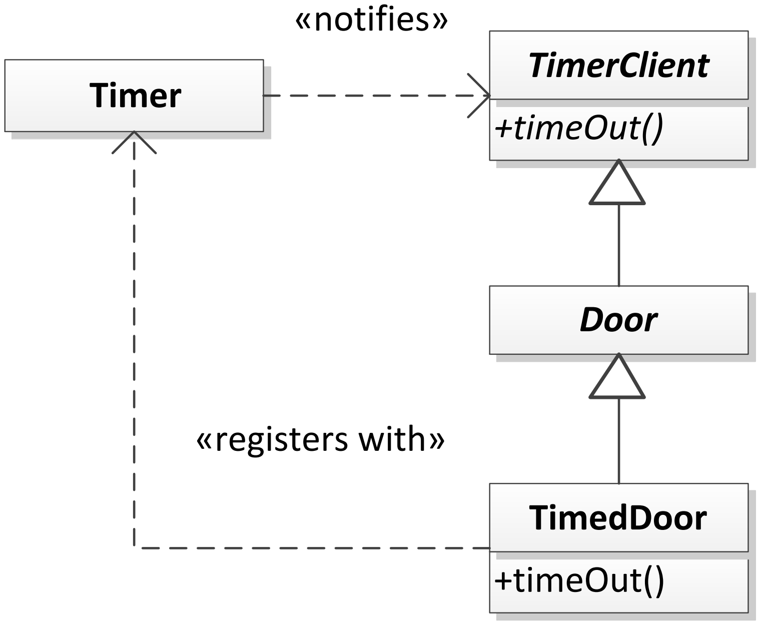
- We wish to have a class `TimedDoor` that will sound an alarm if left open for too long
- First, we will create a class `Timer` which `TimerClients` can register with to receive notifications about timeouts

```
class Timer {
   public:
      void subscribe(int timeout, TimerClient* client);
};

class TimerClient {
   public:
      virtual void timeOut() = 0;
};
```

SOLID: Interface Segregation Principle

- We want `TimedDoor` to be able to register itself with `Timer` so that it can receive notifications when the door has been open for too long
- We choose to have `Door` extend `TimerClient` , so that a new derived class `TimedDoor` will be able to register itself with `Timer`

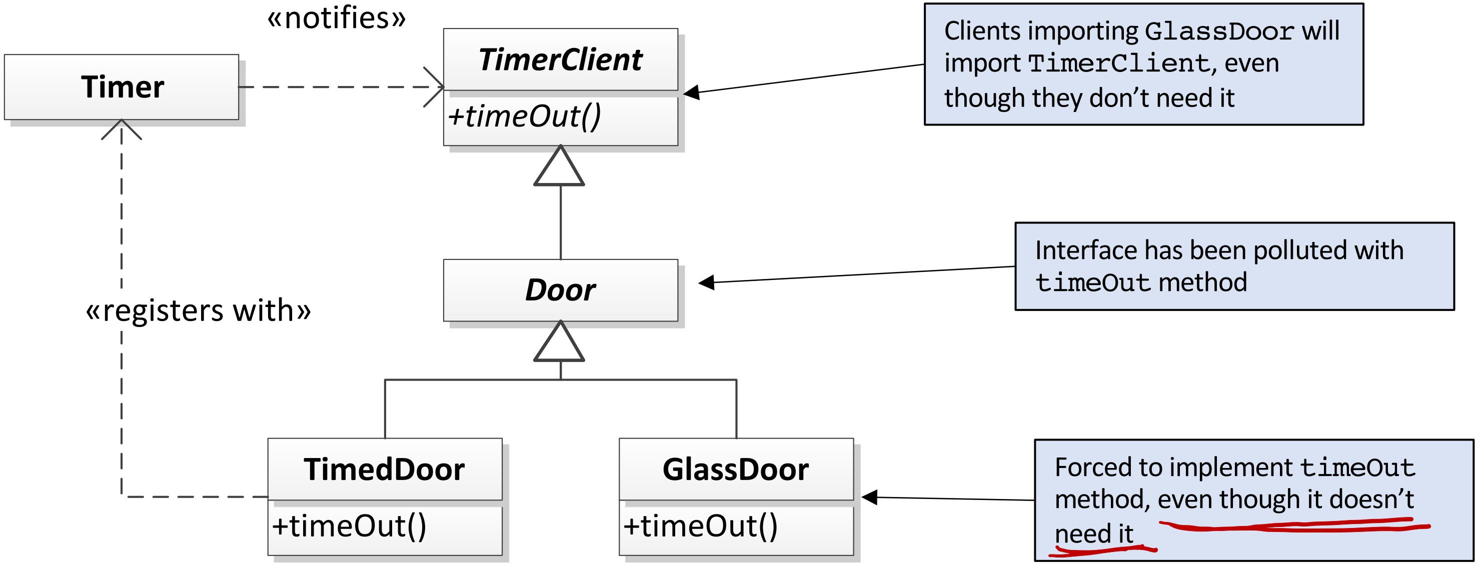
SOLID: Interface Segregation Principle



SOLID: Interface Segregation Principle

- Problems:
  - The interface of `Door` has been polluted with an interface it does not require
  - Door is now dependent on `TimerClient`, but not all doors need timing
  - Those that don't need timing will have to override the `timeOut` method to do nothing
  - When clients `#include` those timing-free doors, they will include the definition of the `TimerClient` class even though it won't be used

SOLID: Interface Segregation Principle



Clients importing `GlassDoor` will import `TimerClient`, even though they don't need it

Interface has been polluted with `timeOut` method

Forced to implement `timeOut` method, even though it doesn't need it

But you still have to implement the pure virtual timeout

SOLID: Interface Segregation Principle

- If we continue this practice, then each time we need a new interface, we will have to add it to the base class, further polluting its interface
- We will have to go back and implement the new interface methods in every subclass, violating the Open/Closed Principle

SOLID: Interface Segregation Principle

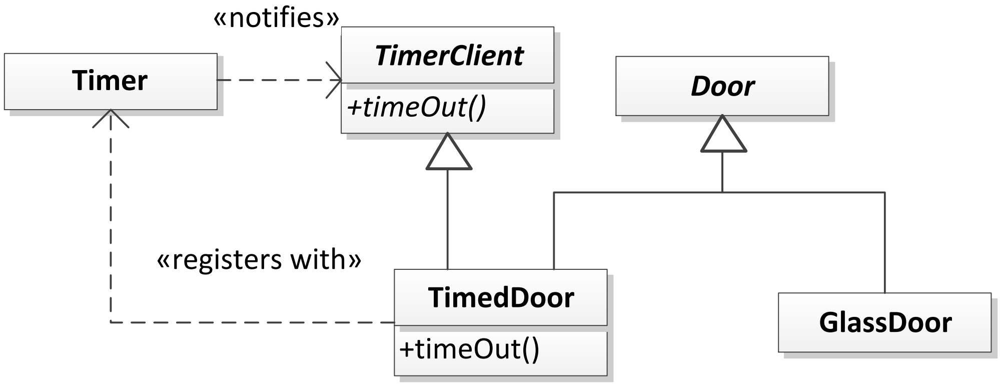
- `Door` and `TimerClient` provide interfaces used by completely different clients:
    - `Timer` uses `TimerClient`
    - Classes that manipulate doors use `Door`
    - If the clients are separate, then so, too, should the interfaces be separate

SOLID: Interface Segregation Principle

- Bottom line:
  - Don’t add new methods appropriate to only one or a few implementation classes
  - Instead, divide the bloated interface into multiple smaller, more cohesive interfaces
  - New classes can then implement only the ones they need

SOLID: Interface Segregation Principle

- Solution using multiple inheritance:



- The Adapter design pattern can also be used to solve this sort of problem – more on this pattern later

SOLID: Dependency Inversion Principle

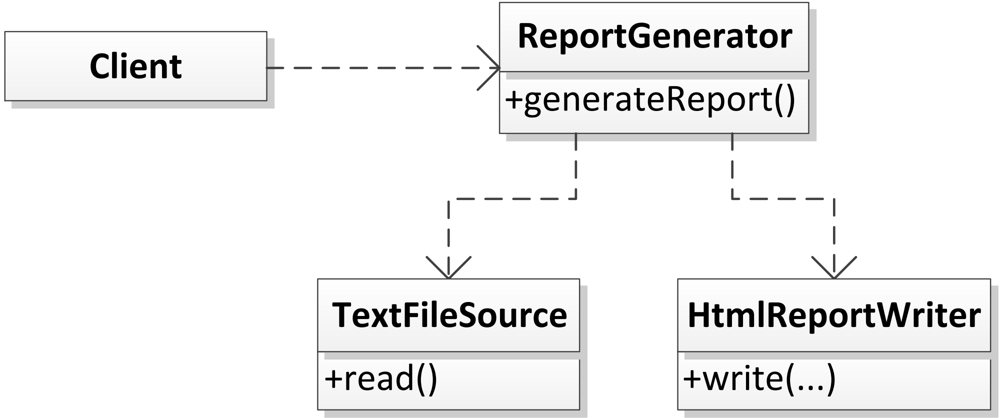
Design Principle:
Dependency Inversion Principle

High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.

SOLID: Dependency Inversion Principle

- Suppose we want to take data stored in text files and generate reports in HTML format …



SOLID: Dependency Inversion Principle

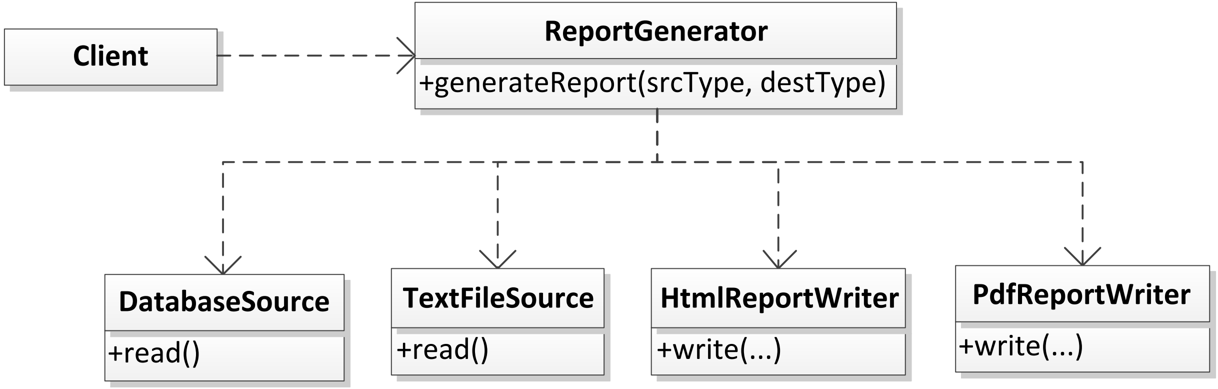
```
class ReportGenerator {
   public:

   ...

   void generateReport() {
      TextFileSource* src = new TextFileSource(this->_inFile);
      HtmlReportWriter* dest = new HtmlReportWriter(this->_outFile);

      string line;
      while (line = src->read()) {
         // Compile report
      }

      // Write report in HTML format
      dest->write(...);
   }
};
```

SOLID: Dependency Inversion Principle

- `TextFileSource` and `HtmlReportWriter` are certainly reusable
- But, we cannot reuse `ReportGenerator` unless we want to read from text files and write to HTML files
- Suppose we write a new program that needs to read from a database and write to PDF files – it would be nice to reuse `ReportGenerator`
- `ReportGenerator` is dependent on `TextFileSource` and `HtmlReportWriter`, so this is not possible

SOLID: Dependency Inversion Principle

- We could modify `generateReport` to accept the type of source and destination to use …



SOLID: Dependency Inversion Principle

```
class ReportGenerator {
   public:

   ...

      void generateReport(string srcType, string destType) {
         if ((srcType == "text") && (destType == "html")
            generateHtmlReportFromText();
         else if ((srcType == "text") && (destType == "pdf"))
            generatePdfReportFromText();
         else if ((srcType == "db") && (destType == "html"))
            generateHtmlReportFromDb();
         else if ((srcType == "db") && (destType == "pdf"))
            generatePdfReportFromDb();
         else
                   // throw exception
      }
};
```

SOLID: Dependency Inversion Principle

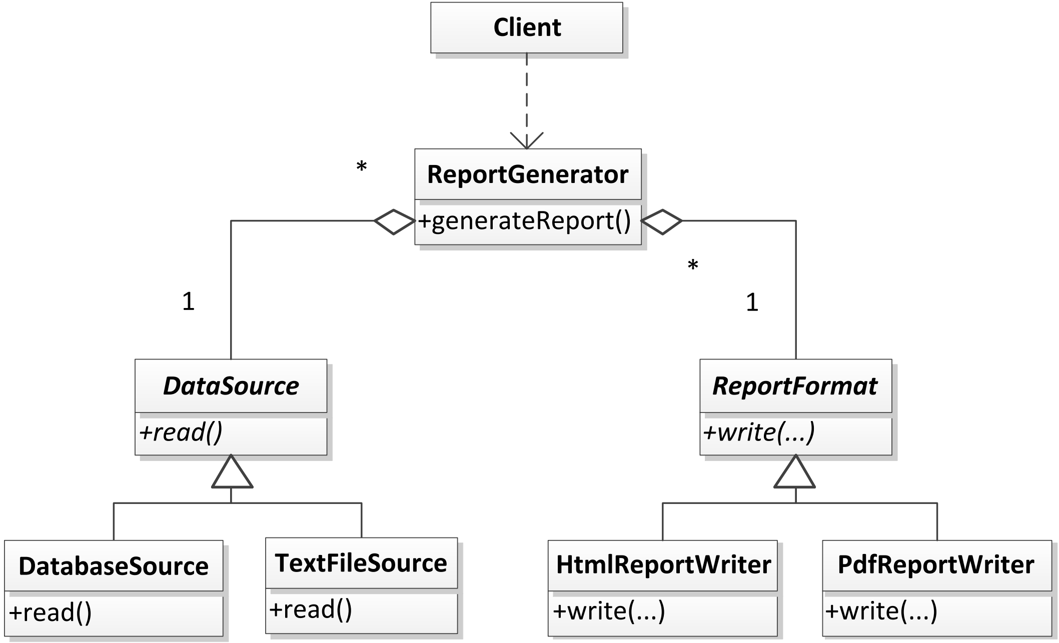
- This drastically increases coupling in the system
  - Over time, more source and destination types will be added to `generateReport`
  - The `ReportGenerator` class will be littered with `if-else` statements and dependent upon many lower-level modules
- This also results in a rigid and fragile system
  - **Rigid**: the system will become hard to change since every change will affect too many parts of the system
  - **Fragility**: when changes are made to the system, unexpected parts will break due to the changes

SOLI**D**: Dependency Inversion Principle

- Better solution:
  - Make `ReportGenerator` (the higher-level class) independent of the lower-level classes it controls
  - We can then reuse it freely
  - This is called *dependency inversion*

depend on new abstraction.

SOLID: Dependency Inversion Principle



SOLID: Dependency Inversion Principle

```
class ReportGenerator {
   public:

   ...

   void generateReport() {
      string line;
      while (line = this->_src->read()) {
         // Compile report
      }
      // Write report
      this->_dest->write(...);
   }

   private:
      DataSource* _src;
      ReportFormat* _dest;
};
```

Summing up SOLID, courtesy of globalnerdy.com

SOLID

Single Responsibility Principle

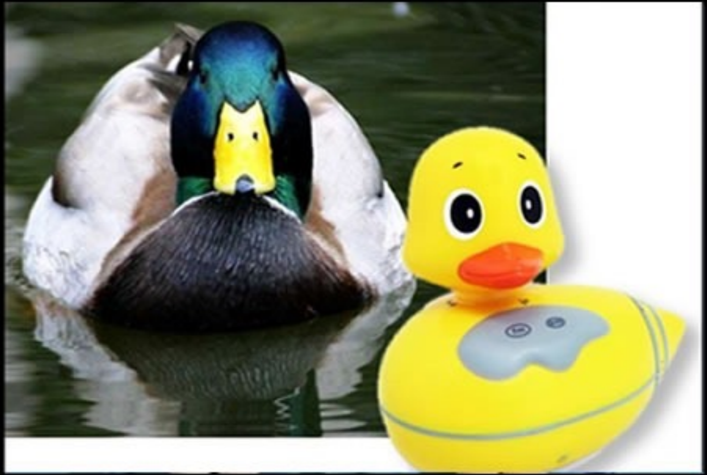
Just because you *can* doesn't mean you *should*.

SOLID

Open-Closed Principle

Open-chest surgery isn’t needed when putting on a coat.

SOLID



Liskov Substitution Principle

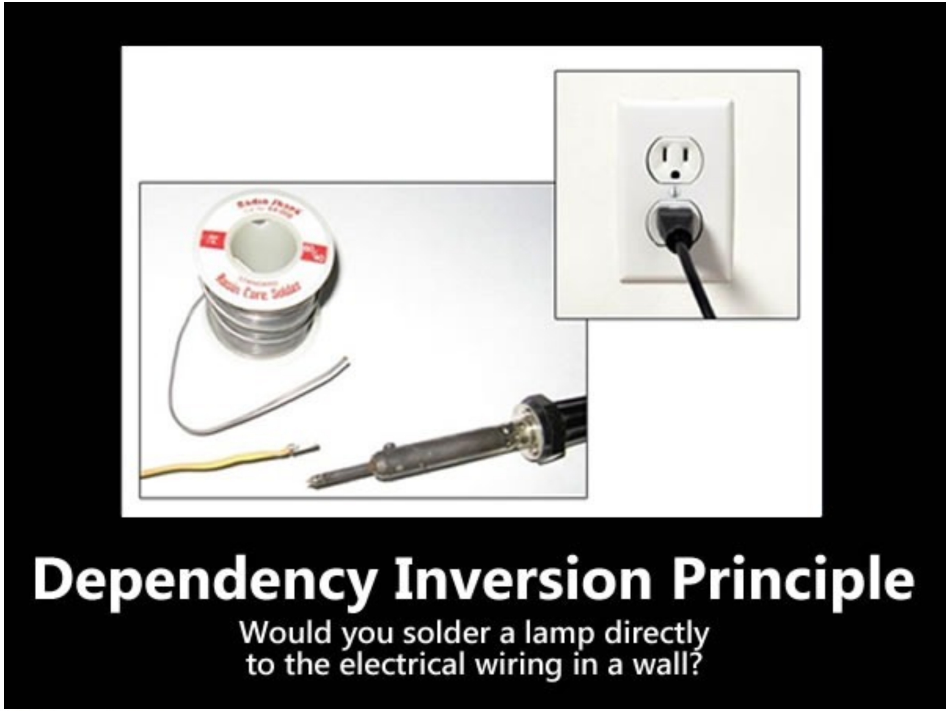
If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

SOLID

Interface Segregation Principle

You want me to plug this in *where?*

SOLID


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