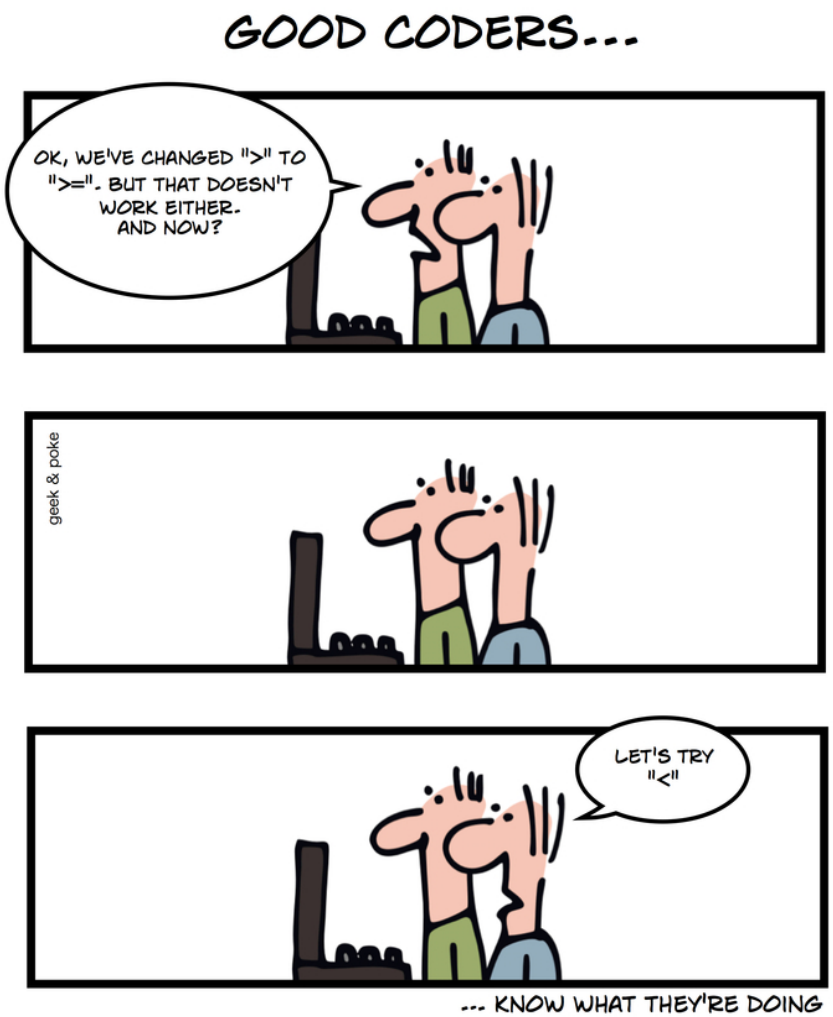
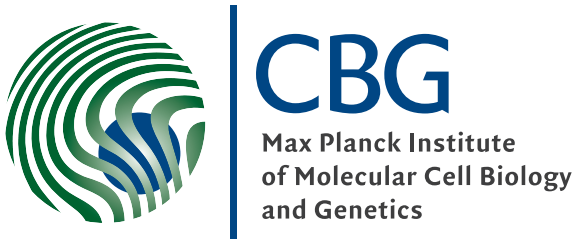

GOOD CODERS...
OK, WE'VE CHANGED ">" TO ">=". BUT THAT DOESN'T WORK EITHER. AND NOW?
geek & poke
LET'S TRY "<"
... KNOW WHAT THEY'RE DOING

Class Design Principles in Object-Oriented Programming

Peter Steinbach

Scionics Computer Innovations GmbH

October 9th, 2012



Outline

Motivation

Motivation

. . .

Motivation

. . .

Taken from http://sourcemaking.com

Class Design Principles

0. Orthogonality/Cohesion

Definition

A **Responsibility** of a class is defined as *a reason for the class to change*.

0. Orthogonality/Cohesion

Definition

A **Responsibility** of a class is defined as *a reason for the class to change*.

Exercise

How many responsibilities do classes a) and b) have?

- the modem class

```
class Modem
{
  public:
    void dial(String phoneNumber);
    void hangup();
    void send(char aCharacter);
    char receive();
}
```

0. Orthogonality/Cohesion

Definition

A **Responsibility** of a class is defined as *a reason for the class to change*.

Exercise

How many responsibilities do classes a) and b) have?

- the modem class $\rightarrow$ 2 responsibilities (dialing and transmitting data)!

```
class Modem
{
  public:
    void dial(String phoneNumber);
    void hangup();
    void send(char aCharacter);
    char receive();
}
```

0. Orthogonality/Cohesion

Definition

A **Responsibility** of a class is defined as *a reason for the class to change*.

Exercise

How many responsibilities do classes a) and b) have?

- the modem class $\rightarrow$ 2 responsibilities (dialing and transmitting data)!

```
class Modem
{
  public:
    void dial(String phoneNumber);
    void hangup();
    void send(char aCharacter);
    char receive();
}
```

- an example of a god class: TROOT.html

0. Orthogonality/Cohesion

Definition

A **Responsibility** of a class is defined as *a reason for the class to change*.

Exercise

How many responsibilities do classes a) and b) have?

- the modem class → 2 responsibilities (dialing and transmitting data)!

```
class Modem
{
  public:
    void dial(String phoneNumber);
    void hangup();
    void send(char aCharacter);
    char receive();
}
```

- an example of a god class: TROOT.html → infinite responsibilities (getters/setters)

0. Orthogonality/Cohesion

Definition

A **Responsibility** of a class is defined as *a reason for the class to change*.

Exercise

How many responsibilities do classes a) and b) have?

- the modem class → 2 responsibilities (dialing and transmitting data)!

```
class Modem
{
  public:
    void dial(String phoneNumber);
    void hangup();
    void send(char aCharacter);
    char receive();
}
```

- an example of a god class: TROOT.html → infinite responsibilities (getters/setters)
- typically derivatives of TSelector have too many responsibilities

0. Orthogonality/Cohesion

Definition

A **responsibility** of a class is defined as *a reason for the class to change*.

0. Orthogonality/Cohesion

Definition

A **responsibility** of a class is defined as *a reason for the class to change*.

. . . from this it follows . . .

0. Orthogonality/Cohesion

Definition

A **responsibility** of a class is defined as *a reason for the class to change*.

. . . from this it follows . . .

Definition

Orthogonality([2]) of a system of classes can be defined as the degree of how many classes have independent or non-overlapping *responsibilities*.

1. Single-Responsibility Principle

Theorem (from [4])

A class should only have **one** *reason to change, i.e. try to create systems with high orthogonality.*

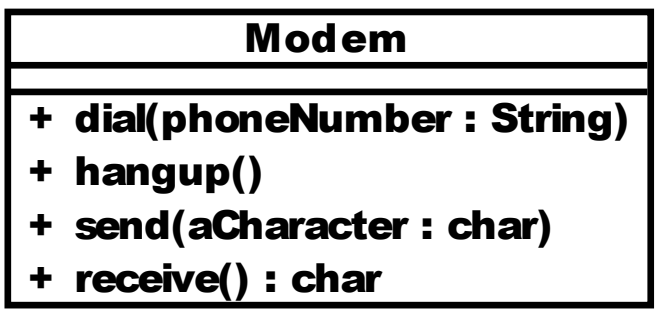
1. Single-Responsibility Principle

Theorem (from [4])

A class should only have **one** *reason to change, i.e. try to create systems with high orthogonality.*

Looking back at the modem exercise

before



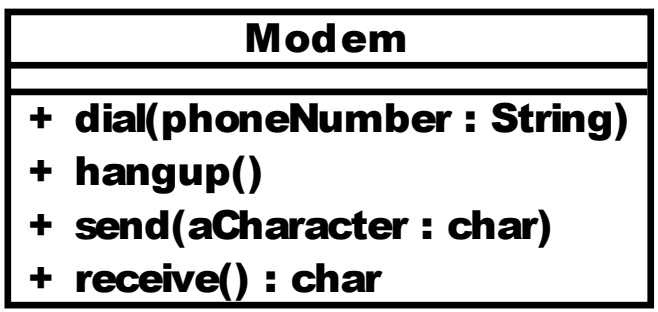
1. Single-Responsibility Principle

Theorem (from [4])

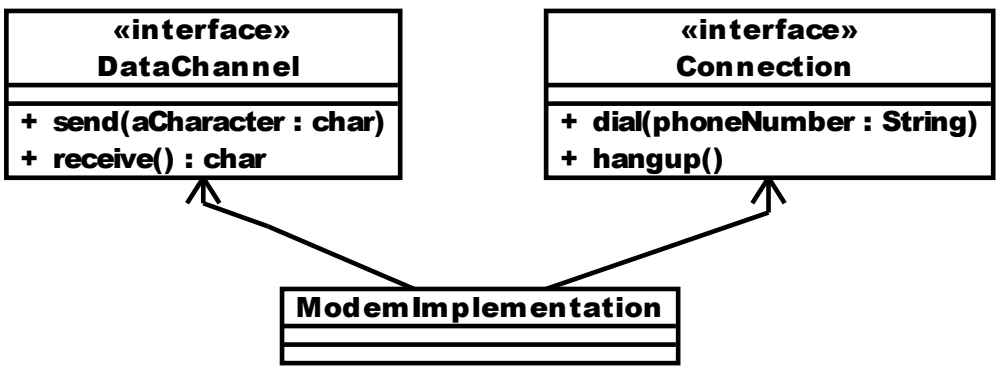
A class should only have **one** *reason to change, i.e. try to create systems with high orthogonality.*

Looking back at the modem exercise

before



after



2. Open-Closed Principle

Theorem (from [4])

Software Entities (classes, modules, functions, etc) should be open for extension, but closed for modification.

2. Open-Closed Principle

Theorem (from [4])

Software Entities (classes, modules, functions, etc) should be open for extension, but closed for modification.

Open

Closed

2. Open-Closed Principle

Theorem (from [4])

Software Entities (classes, modules, functions, etc) should be open for extension, but closed for modification.

Open

- the **behavior** of an entity can be extended
- as requirements of a system change (that's a fact!), the entities behavior can be **extended or modified** to satisfy these changes

Closed

2. Open-Closed Principle

Theorem (from [4])

Software Entities (classes, modules, functions, etc) should be open for extension, but closed for modification.

Open

- the **behavior** of an entity can be extended
- as requirements of a system change (that's a fact!), the entities behavior can be **extended or modified** to satisfy these changes

Closed

- extension of behavior does **NOT** result in changing the source code
- the binary executable version of a given entity remains **untouched**

2. Open-Closed Principle

Theorem (from [4])

Software Entities (classes, modules, functions, etc) should be open for extension, but closed for modification.

Open

- the **behavior** of an entity can be extended
- as requirements of a system change (that's a fact!), the entities behavior can be **extended or modified** to satisfy these changes

Closed

- extension of behavior does **NOT** result in changing the source code
- the binary executable version of a given entity remains **untouched**

Exercise

The above is way too complicated for one slide! Let's have a look at **Problem 1** on the Exercise sheet!

2. Open-Closed Principle, Reviewed

The Square/Circle Problem

- *rigid*: adding triangle requires `Shape`, `Square`, `Circle`, `DrawAllShapes` to be recompiled and redeployed
- *fragile*: `switch/case` will be required by all client classes that use `Shapes`
- *immobile*: reusing `DrawAllShapes` is impossible without including `Shape`, `Square`, `Circle` as well

2. Open-Closed Principle, Reviewed

The Square/Circle Problem

- *rigid*: adding triangle requires `Shape`, `Square`, `Circle`, `DrawAllShapes` to be recompiled and redeployed
- *fragile*: `switch/case` will be required by all client classes that use `Shapes`
- *immobile*: reusing `DrawAllShapes` is impossible without including `Shape`, `Square`, `Circle` as well

Solution: Using Abstraction

```
struct Shape {
  virtual void Draw() const = 0;
}

struct Square : public Shape {
  virtual void Draw() const;
}
```

```
void DrawAllShapes(
  const std::vector<Shape*>& list) {

  std::vector<Shape*>::const_iterator itr;

  for(itr=list.begin();itr!=list.end(); ++itr)
  {
    itr->Draw();
  }

}
```

2. Open-Closed Principle, Summary

But hold on ...

- did the abstraction from above close `DrawAllShapes` against all changes?
  - **No**, there is no model of abstraction that is natural to all contexts!
  - closure can never be complete, only strategic

2. Open-Closed Principle, Summary

But hold on ...

- did the abstraction from above close `DrawAllShapes` against all changes?
  - **No**, there is no model of abstraction that is natural to all contexts!
  - closure can never be complete, only strategic
- how to deal with possible changes?
  1. derive possible changes from software requirements
  2. implement necessary abstractions
  3. wait!

2. Open-Closed Principle, Summary

But hold on ...

- did the abstraction from above close `DrawAllShapes` against all changes?
  - **No**, there is no model of abstraction that is natural to all contexts!
  - closure can never be complete, only strategic
- how to deal with possible changes?
  1. derive possible changes from software requirements
  2. implement necessary abstractions
  3. wait!

To Summarize

- conforming to the open-closed principle yields greatest benefits of OOP (flexibility, reusability, maintainability)
- apply abstraction to parts of software that exhibit frequent change
- Resisting premature abstraction is as important as abstraction itself.

3. Liskov Substitution Principle

Theorem (paraphrased from [3])

Subtypes must be substitutable for their base types.

3. Liskov Substitution Principle

Theorem (paraphrased from [3])

Subtypes must be substitutable for their base types.

Exercise

Try to answer Problem 2 a) and b) on your Exercise Sheet!

Review & Summary: The 3. Liskov Substitution Principle

Observations from the `square/rectangle` problem

Review & Summary: The 3. Liskov Substitution Principle

Observations from the `square/rectangle` problem

- Violations of Liskov Substitution Principle result in Run-Time Type Information to be used
  - violates the Open-Closed Principle

Review & Summary: The 3. Liskov Substitution Principle

Observations from the `square/rectangle` problem

- Violations of Liskov Substitution Principle result in Run-Time Type Information to be used
  - violates the Open-Closed Principle
- an (inheritance) model can never be validated in isolation
  - but rather with its use (users) in mind
  - `Is-A` relationship within inheritance refers to **behavior** that can be **assumed** or that **clients depend upon**.

Review & Summary: The 3. Liskov Substitution Principle

Observations from the `square/rectangle` problem

- Violations of Liskov Substitution Principle result in Run-Time Type Information to be used
  - violates the Open-Closed Principle
- an (inheritance) model can never be validated in isolation
  - but rather with its use (users) in mind
  - `Is-A` relationship within inheritance refers to **behavior** that can be **assumed** or that **clients depend upon**.
- how to ensure/enforce Liskov Substitution Principle?
  - Design-by-Contract
  - in C++: only by assertions or Unit Tests

Review & Summary: The 3. Liskov Substitution Principle

Observations from the `square/rectangle` problem

- Violations of Liskov Substitution Principle result in Run-Time Type Information to be used
  - violates the Open-Closed Principle
- an (inheritance) model can never be validated in isolation
  - but rather with its use (users) in mind
  - `Is-A` relationship within inheritance refers to **behavior** that can be **assumed** or that **clients depend upon**.
- how to ensure/enforce Liskov Substitution Principle?
  - Design-by-Contract
  - in C++: only by assertions or Unit Tests

Summary

- this principle ensures: maintainability, re-usability, robustness
- Liskov Substitution Principle enables the Open-Closed Principle
- the contract of a base type has to be well understood, if not even enforced by the code

4. Dependency-Inversion Principle

Theorem (from [4])

1. *High level modules* **should not depend** *upon low level modules. Both should depend upon abstractions!*
2. *Abstractions* **should not depend** *upon details. Details should depend upon abstractions!*

4. Dependency-Inversion Principle

Theorem (from [4])

1. *High level modules* **should not depend** *upon low level modules. Both should depend upon abstractions!*
2. *Abstractions* **should not depend** *upon details. Details should depend upon abstractions!*

Exercise

Please complete problem 3 - the `Lamp` class!

4. Dependency-Inversion Principle, Observations

Exercise continued

1. The vendor of `Lamp` changes it's definition. All methods containing `Turn` are renamed to `Ramp`! Face your design with that!

4. Dependency-Inversion Principle, Observations

Exercise continued

1. The vendor of `Lamp` changes it's definition. All methods containing `Turn` are renamed to `Ramp`! Face your design with that!
2. Look at `Button`: Can it be reused for classes of type `Signal`?

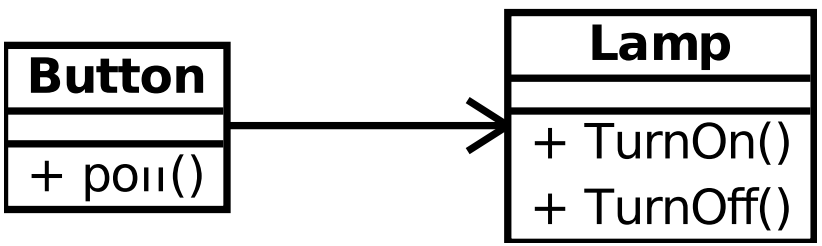
4. Dependency-Inversion Principle, Observations

Exercise continued

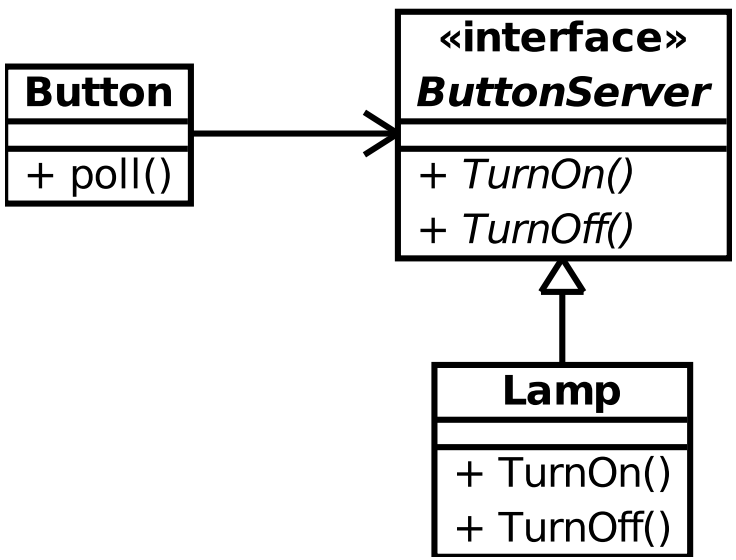
1. The vendor of `Lamp` changes it's definition. All methods containing `Turn` are renamed to `Ramp`! Face your design with that!
2. Look at `Button`: Can it be reused for classes of type `Signal`?

Exercise: A Solution

Naive Ansatz



Inverted Dependency

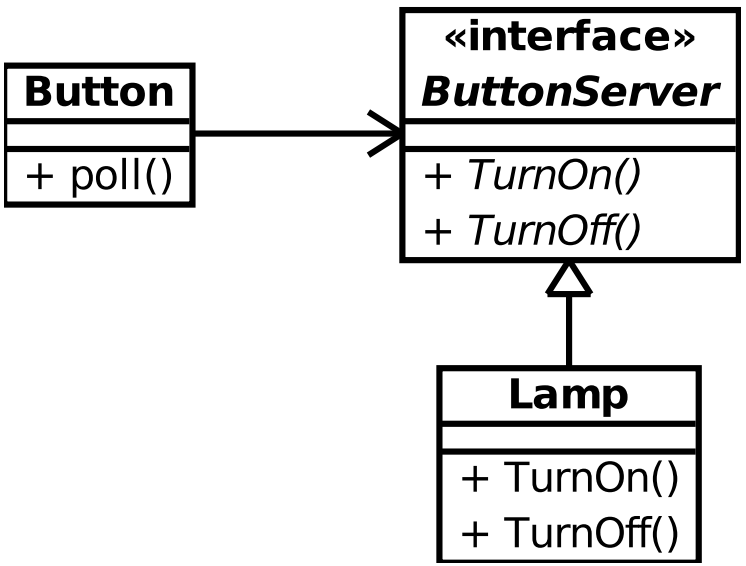


4. Dependency-Inversion Principle, Review

Dynamic and Static Polymorphism

in C++, both can help to invert dependencies

Dynamic Polymorphism through Abstract Interfaces

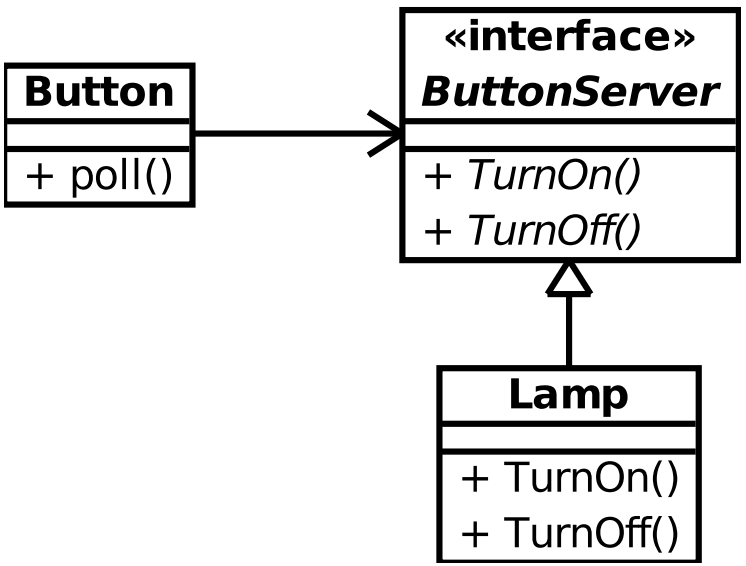


4. Dependency-Inversion Principle, Review

Dynamic and Static Polymorphism

in C++, both can help to invert dependencies

Dynamic Polymorphism through Abstract Interfaces



Static Polymorphism through template classes

```
template <class TurnableObject>
class Button {

TurnableObject* itsTurnable;

public:
  Button(TurnableObject* _object = 0 ):
    itsTurnable(_object)
    {};

  void poll() {
    if(/*some condition*/)
      itsTurnable.TurnOn();
    }
};
```

- compile-time polymorphism
- design-by-policy, see [1]

4. Dependency-Inversion Principle, Summary

Summary

- dependency of policies on details is natural to procedural design
- inversion of dependencies is hallmark of (good) object-oriented design
- Dependency-Inversion Principle is at the heart of reusable frameworks (no matter what size)
- enables the Open-Closed Principle

5. Interface-Segregation Principle

Exercise

Let's complete Problem 4 a) and b)!

5. Interface-Segregation Principle

Exercise

Let's complete Problem 4 a) and b)!

Solutions 4.a) & 4.b)

5. Interface-Segregation Principle

Exercise

Let's complete Problem 4 a) and b)!

Solutions 4.a) & 4.b)

```
class Timer
{
  public:
  void Register(const int& timeout,
                const int& timerID,
                TimerClient* client);

};

class TimerClient
{
  public:
  virtual void TimeOut(const int& timerID) = 0;

};
```

- `TimelessDoor` does hold methods that it does not need
- client behavior of `DoubleTimedDoor` forces Timer and TimerClient to change (and maybe all clients of it)
- Clients can exert a force on their interfaces to change

5. Interface-Segregation Principle

Theorem (from [4])

Clients should not be forced to depend on methods that they do not use.[a]

[a] Don't write fat interfaces.

5. Interface-Segregation Principle

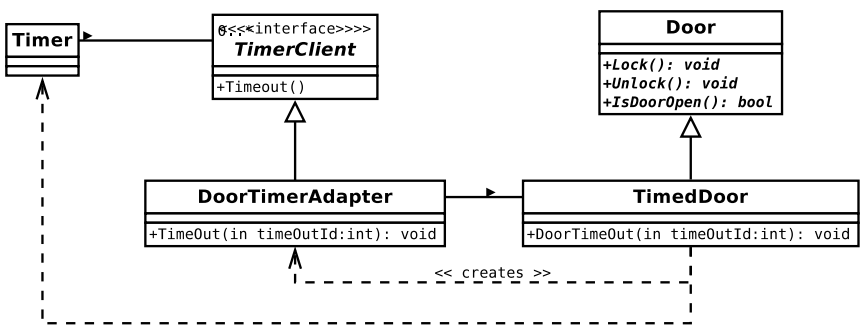
Theorem (from [4])

Clients should not be forced to depend on methods that they do not use.[a]

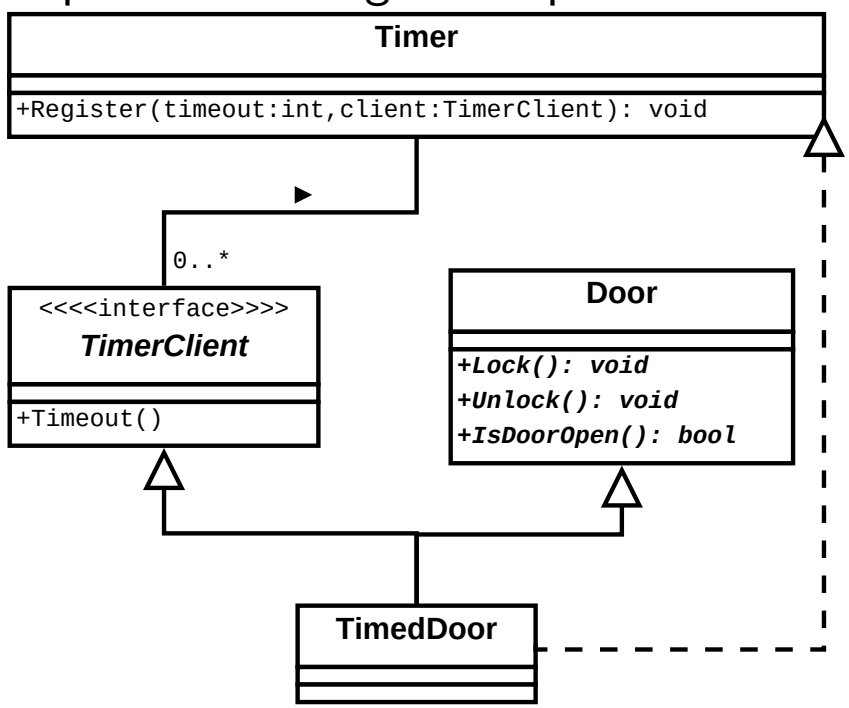
[a] Don't write fat interfaces.

Two ways out

Separation through Delegation



Separation through Multiple Inheritance



Summary

- although having a slow learning curve, OOP can help solve highly-sophisticated problems with software

Summary

- although having a slow learning curve, OOP can help solve highly-sophisticated problems with software
- learning OO Class Design can prevent sleepless nights of debugging or copy-and-past'ing

Summary

- although having a slow learning curve, OOP can help solve highly-sophisticated problems with software
- learning OO Class Design can prevent sleepless nights of debugging or copy-and-past'ing
- Coding may not be our profession, but we do it everyday anyhow, so we better know our craft!

Thank you for your attention!

References

[1] Andrei Alexandrescu.
Modern C++ Design: Generic Programming and Design Patterns Applied.
Addison-Wesley Professional, 2001.

[2] Andrew Hunt and David Thomas.
The Pragmatic Programmer.
Addison Wesley, 2000.
pragmaticprogrammer.com.

[3] Barbara Liskov.
Keynote address - data abstraction and hierarchy.
SIGPLAN Not., 23:17–34, January 1987.

[4] Robert C. Martin, James W. Newkirk, and Robert S. Koss.
Agile Software Development.
Prentice Hall, 2003.
Class Design Principles at Author's Homepage.