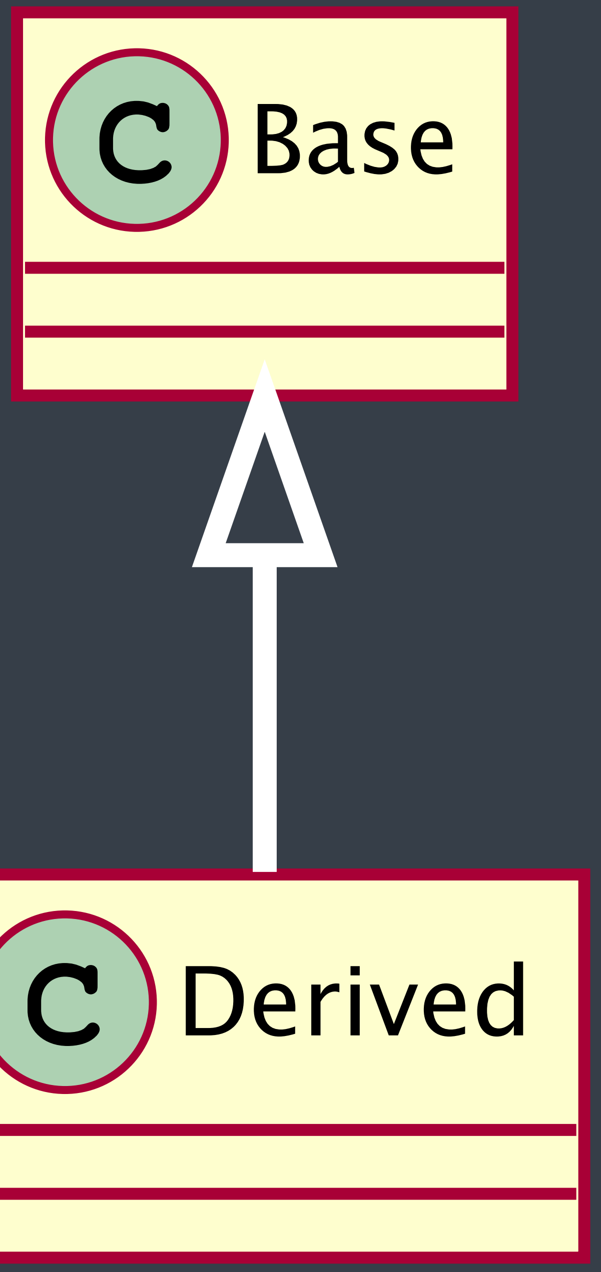
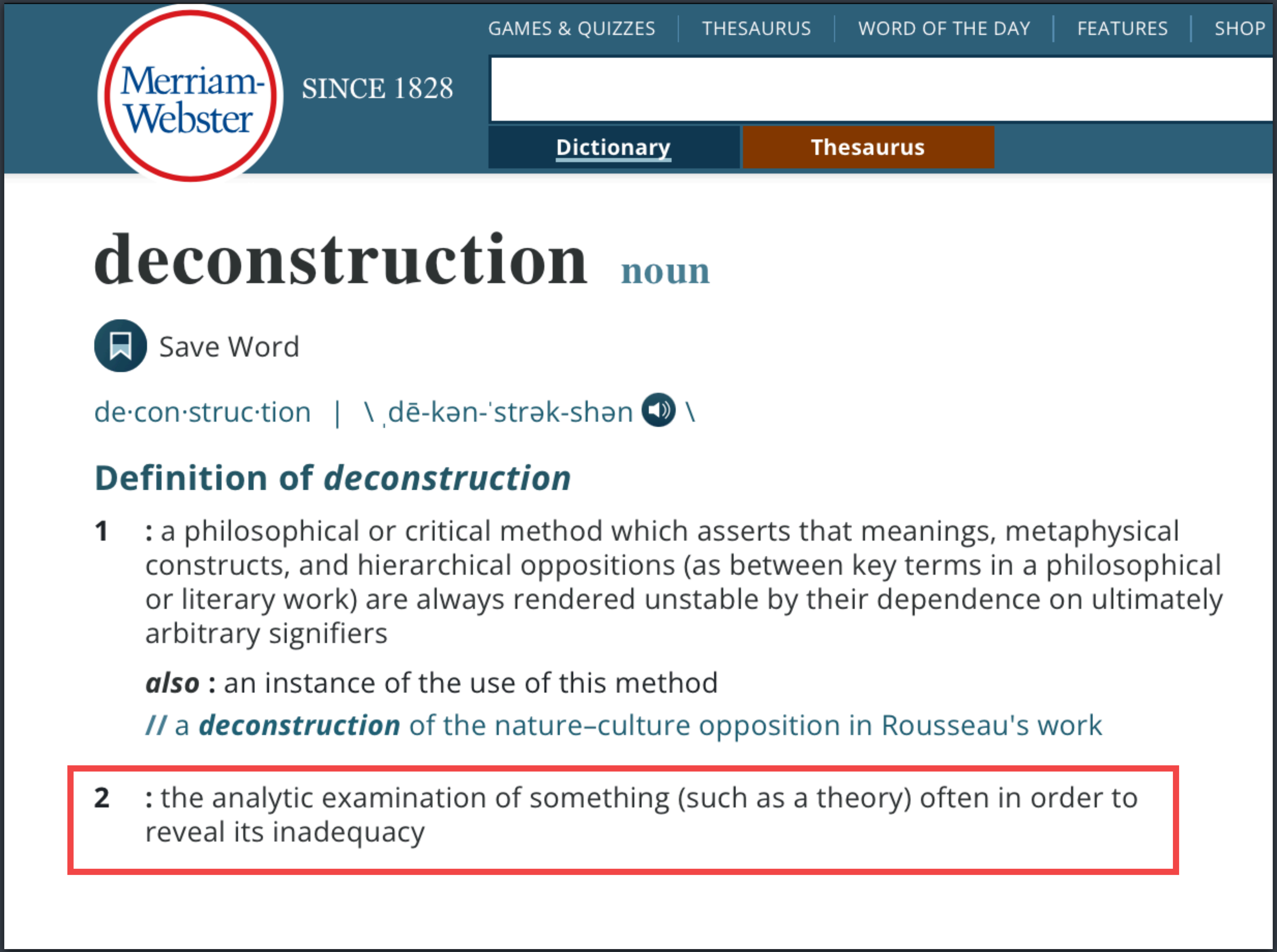
Deconstructing Inheritance

lucteo.ro/pres/2021-meetingcpp/

LUCIAN RADU TEODORESCU
GARMIN


C
Base
C
Derived



deconstruction *noun*

de·con·struc·tion | \ ˌdē-kən-ˈstrək-shən \

Definition of *deconstruction*

1 : a philosophical or critical method which asserts that meanings, metaphysical constructs, and hierarchical oppositions (as between key terms in a philosophical or literary work) are always rendered unstable by their dependence on ultimately arbitrary signifiers

also : an instance of the use of this method

// a ***deconstruction*** of the nature–culture opposition in Rousseau's work

2 : the analytic examination of something (such as a theory) often in order to reveal its inadequacy

@LucT3o

inheritance in C++

heavily used

why use inheritance?

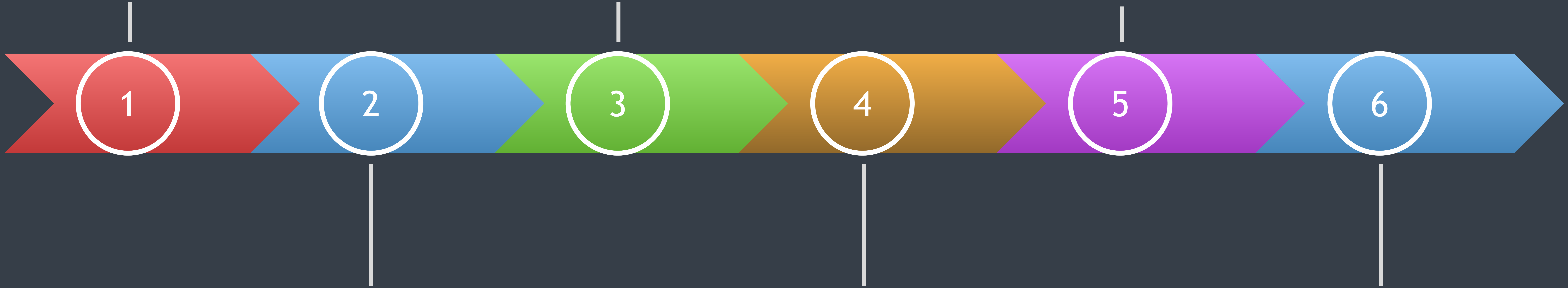
model real-world concepts

reuse members

subtyping

implement an interface

Agenda



Simple problems are hard

The grand IS-A confusion

Liskov Substitution Principle

Inheritance and Friendship

Inheritance vs Composition

Putting things together

1

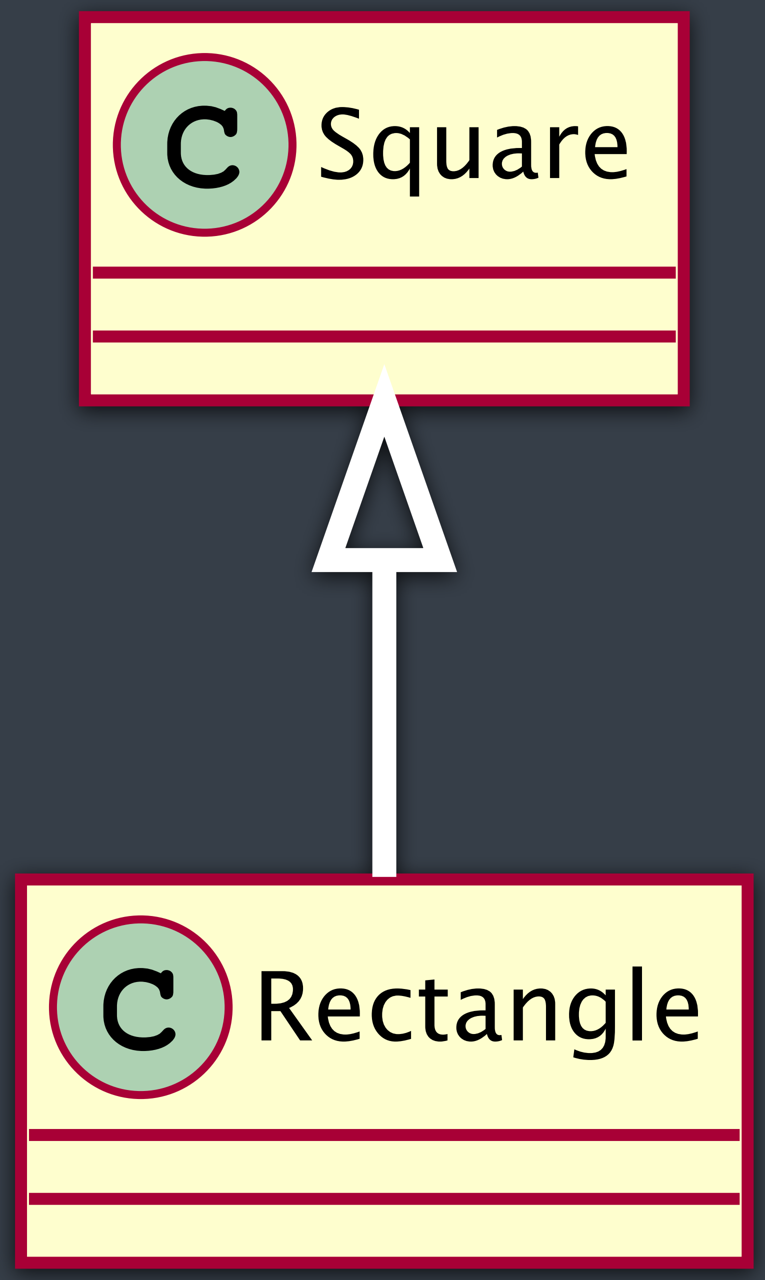
Simple problems are hard

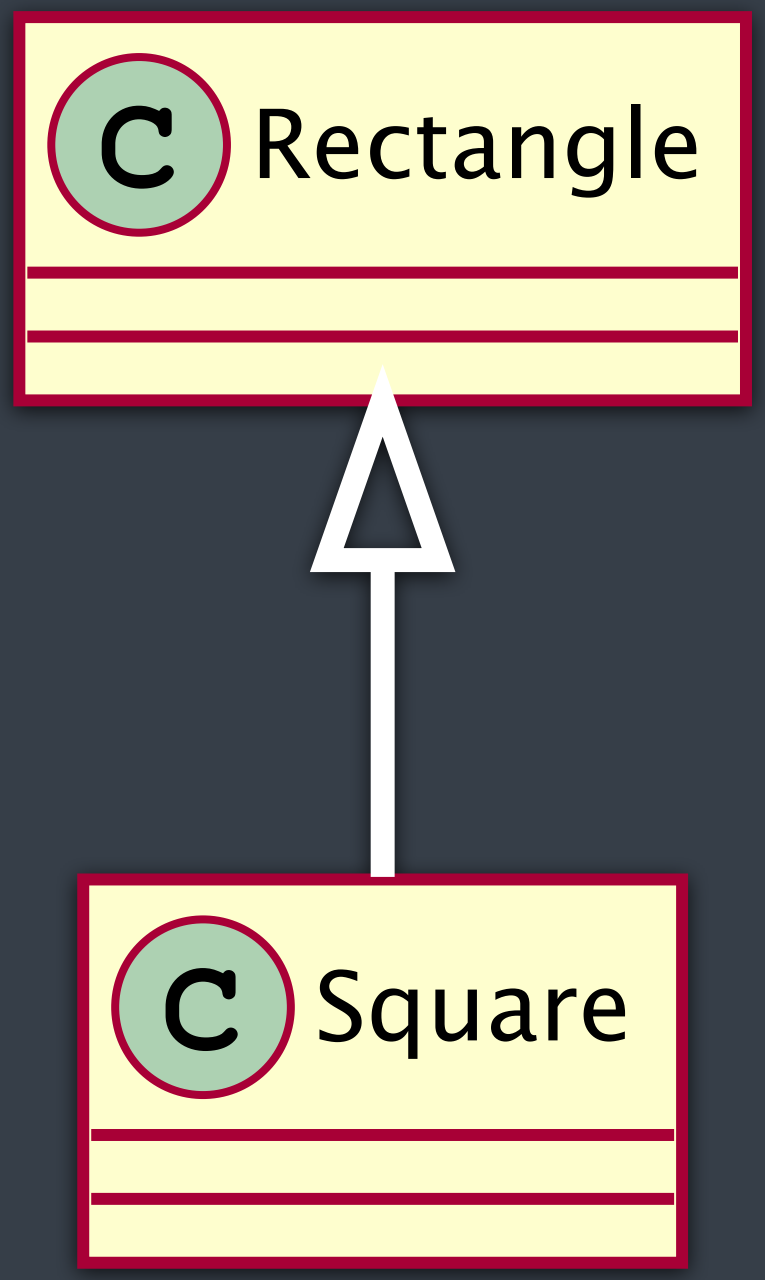
problem statement

design two classes: **Rectangle** and **Square**

operations: get/set dimensions, get area

two solutions





1. Rectangle is-a Square

```cpp
class Square {
    int size;

public:
    virtual int getSize() const { return size; }
    virtual void setSize(int x) { size = x; }
    virtual int getArea() const { return size * size; }
};

class Rectangle : public Square {
    int width;

public:
    virtual int getWidth() const { return width; }
    virtual int getHeight() const { return Square::getSize(); }
    void setSize(int x) override {
        Square::setSize(x);
        width = x;
    }
    virtual void setWidth(int x) { width = x; }
    virtual void setHeight(int x) { Square::setSize(x); }
    int getArea() const override { return width * getSize(); }
};
```

problems

mathematically incorrect

interface of **Rectangle** is polluted

LSP test broken

```
void increaseArea(Square& square) {
    auto oldArea = square.area();

    square.setSize(square.getSize() * 2);

    auto newArea = square.area();
    assert(newArea == 4 * oldArea);
}
```

2. Square is-a Rectangle

```cpp
class Rectangle {
   int width, height;
public:
    virtual int getWidth() const  { return width; }
    virtual int getHeight() const { return height; }
    virtual void setWidth(int x)  { width = x; }
    virtual void setHeight(int x) { height = x; }
    virtual int getArea() const   { return width*height; }
};

class Square: public Rectangle {
public:
    virtual int getSize() const { return Rectangle::getWidth(); }
    virtual void setSize(int x) {
        Rectangle::setWidth(x);
        Rectangle::setHeight(x);
    }
};
```

fixes

mathematically seems correct

interface is less polluted

problems

twice as much storage needed for **Square**

LSP test broken

```
void increaseAreaNew(Rectangle& r) {
    auto oldArea = r.area();

    square.setWidth(r.getWidth() * 2);

    auto newArea = r.area();
    assert(newArea == 2 * oldArea);
}
```

fix 1

make classes immutable

```cpp
class Rectangle {
protected:
    int width, height;
public:
    Rectangle(int w, int h) : width(w), height(h) {}
    virtual int getWidth() const { return width; }
    virtual int getHeight() const { return height; }
    virtual int getArea() const { return width*height; }
};

class Square: public Rectangle {
public:
    Square(int s) : Rectangle(s, s) {}
    virtual int getSize() const { return Rectangle::getWidth(); }
};
```

problems

twice as much storage needed for **Square**

inheritance doesn’t buy us anything

fix 2

remove inheritance

```cpp
class Rectangle {
protected:
    int width, height;
public:
    Rectangle(int w, int h) : width(w), height(h) {}
    int getWidth() const { return width; }
    int getHeight() const { return height; }
    int getArea() const { return width*height; }
};

class Square {
    int size;
public:
    Square(int s) : size(s) {}
    int getSize() const { return size; }
    int getArea() const { return size*size; }
};
```

The truth is that Squares and Rectangles, even immutable Squares and Rectangles, **ought not be associated by inheritance**

Robert C. Martin

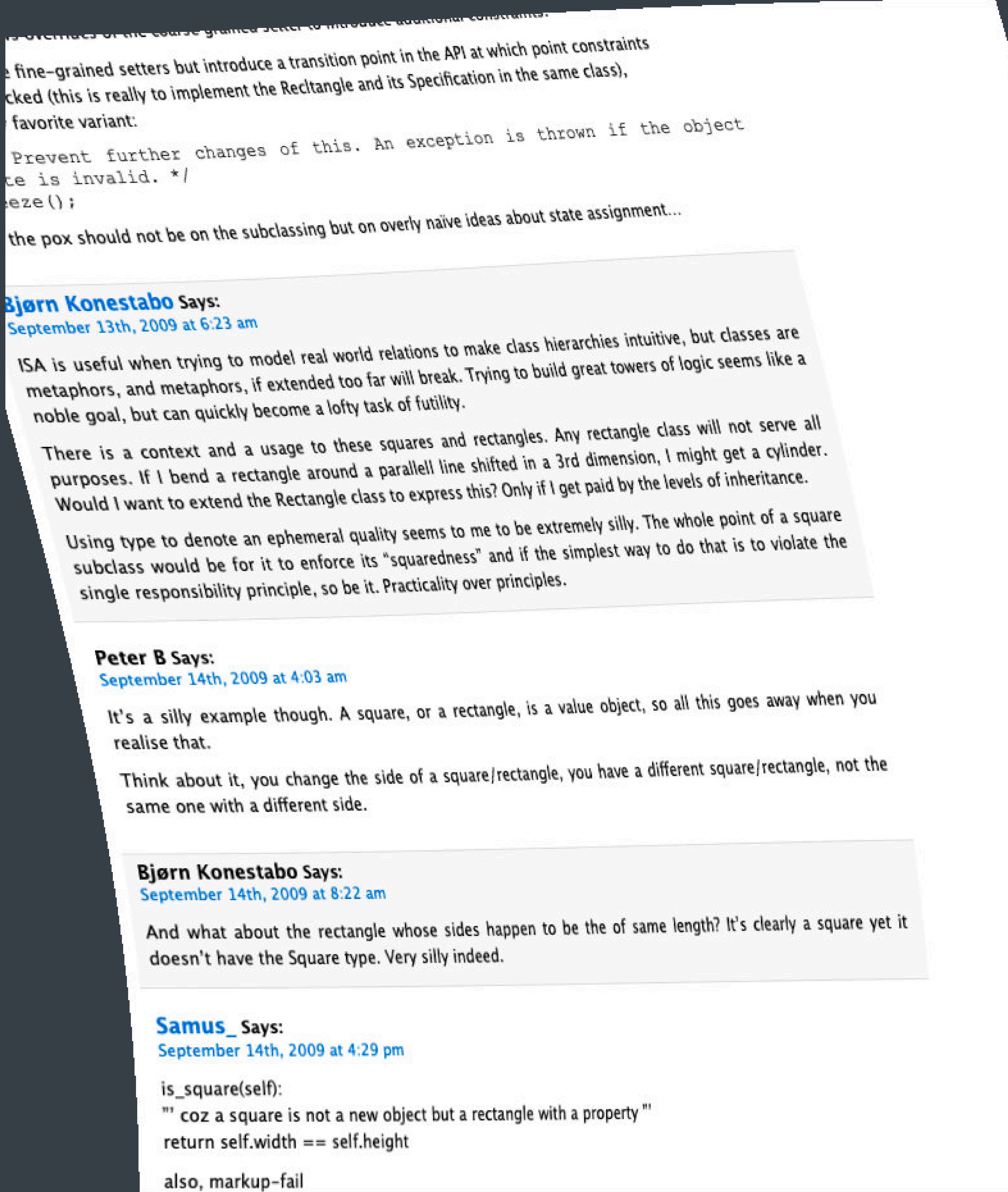
The class Square is not a square, it is a **program that represents a square**. The class Rectangle is not a rectangle, it is a program that represents a rectangle. [...] The fact that a square is a rectangle does not mean that their **representatives** share the ISA relationship.

Robert C. Martin

@LucT3o

ISA is useful when trying to model real world relations to make class hierarchies intuitive,
but **classes are metaphors**,
and metaphors,
if extended too far will break

Bjørn Konestabo


e fine-grained setters but introduce a transition point in the API at which point constraints
cked (this is really to implement the Recltangle and its Specification in the same class),
favorite variant:

```
 Prevent further changes of this. An exception is thrown if the object
te is invalid. */
eze();
```

the pox should not be on the subclassing but on overly naïve ideas about state assignment…

Bjørn Konestabo Says:
September 13th, 2009 at 6:23 am

ISA is useful when trying to model real world relations to make class hierarchies intuitive, but classes are metaphors, and metaphors, if extended too far will break. Trying to build great towers of logic seems like a noble goal, but can quickly become a lofty task of futility.

There is a context and a usage to these squares and rectangles. Any rectangle class will not serve all purposes. If I bend a rectangle around a parallell line shifted in a 3rd dimension, I might get a cylinder. Would I want to extend the Rectangle class to express this? Only if I get paid by the levels of inheritance.

Using type to denote an ephemeral quality seems to me to be extremely silly. The whole point of a square subclass would be for it to enforce its "squaredness" and if the simplest way to do that is to violate the single responsibility principle, so be it. Practicality over principles.

Peter B Says:
September 14th, 2009 at 4:03 am

It's a silly example though. A square, or a rectangle, is a value object, so all this goes away when you realise that.

Think about it, you change the side of a square/rectangle, you have a different square/rectangle, not the same one with a different side.

Bjørn Konestabo Says:
September 14th, 2009 at 8:22 am

And what about the rectangle whose sides happen to be the of same length? It's clearly a square yet it doesn't have the Square type. Very silly indeed.

Samus_ Says:
September 14th, 2009 at 4:29 pm

is_square(self):
"' coz a square is not a new object but a rectangle with a property "'
return self.width == self.height

also, markup-fail

why use inheritance?

~~model real-world concepts~~

~~reuse members~~

subtyping

implement an interface

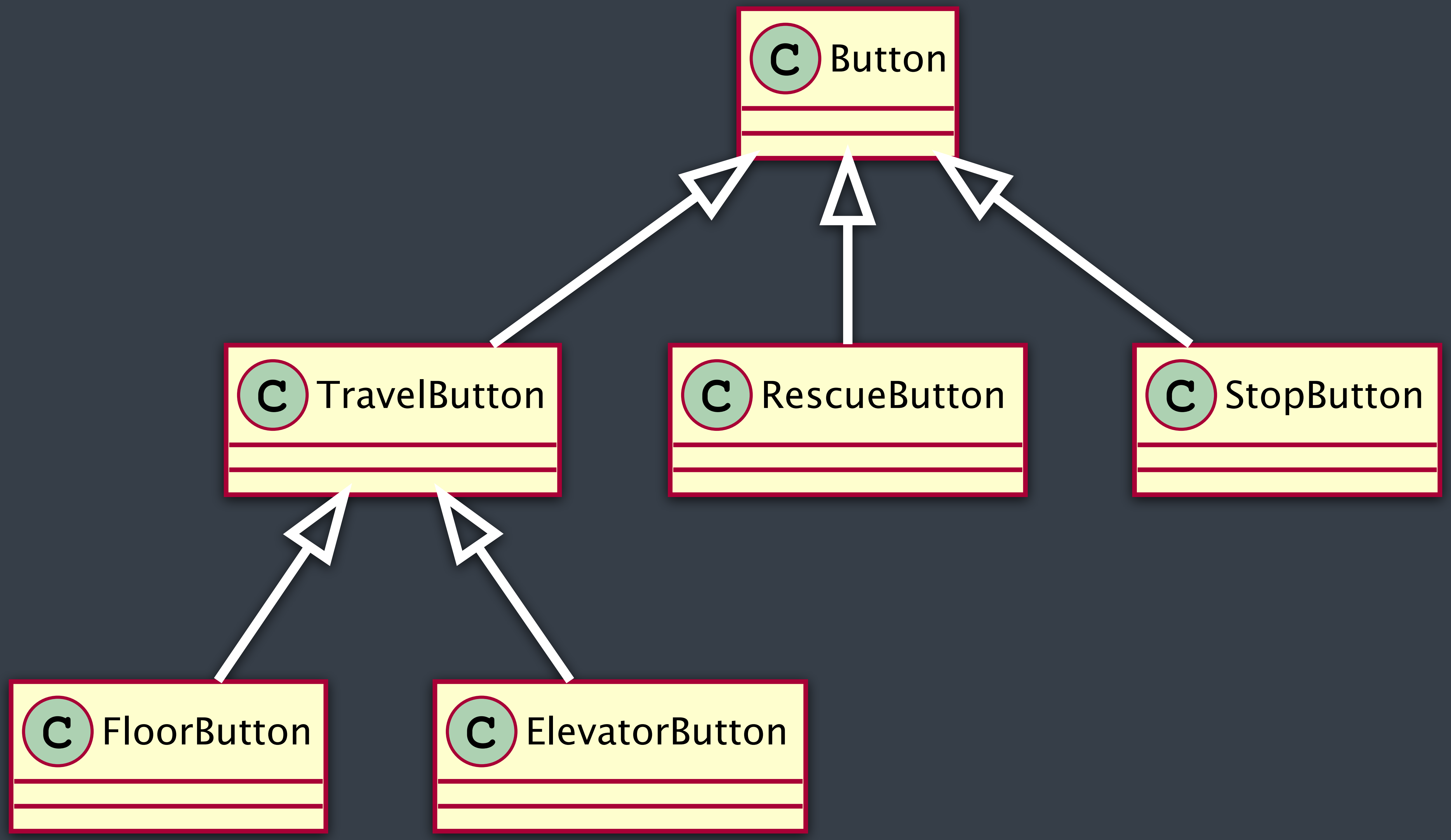
The grand IS-A confusion

2

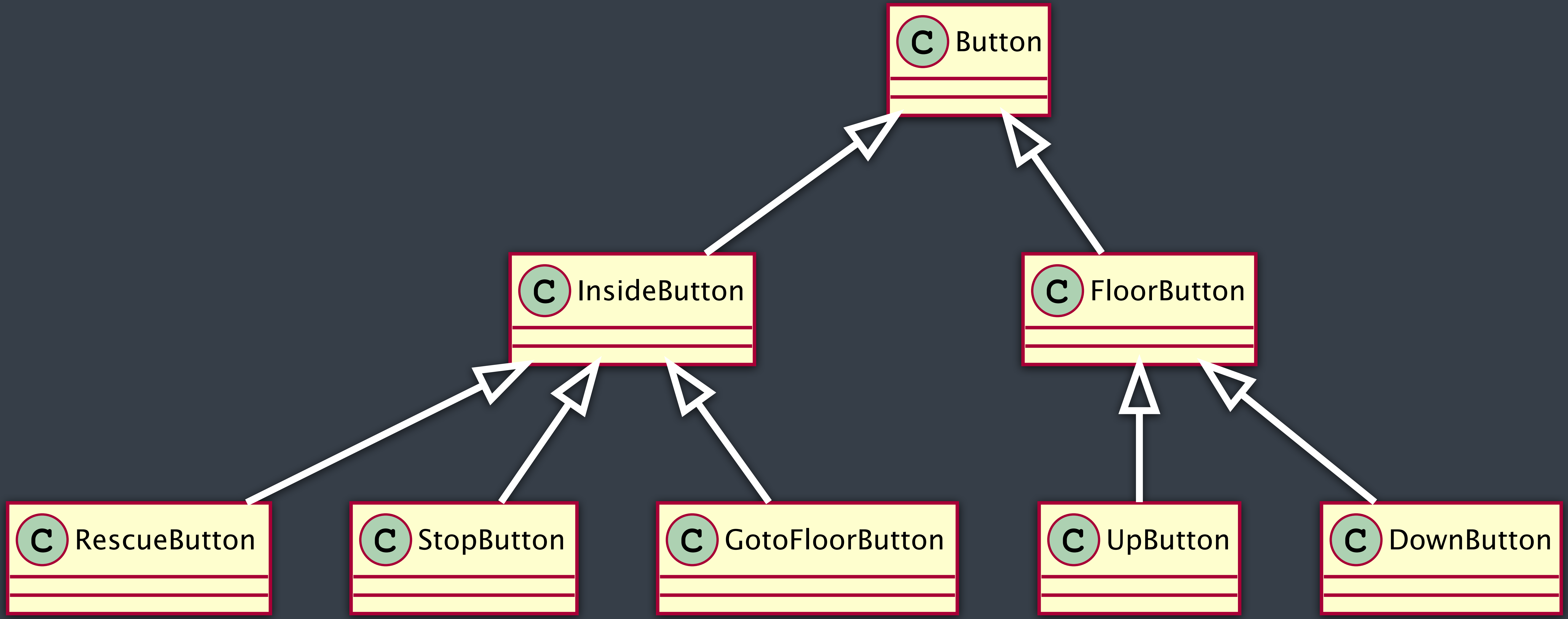
problem: elevator system

a lot of buttons

option 1

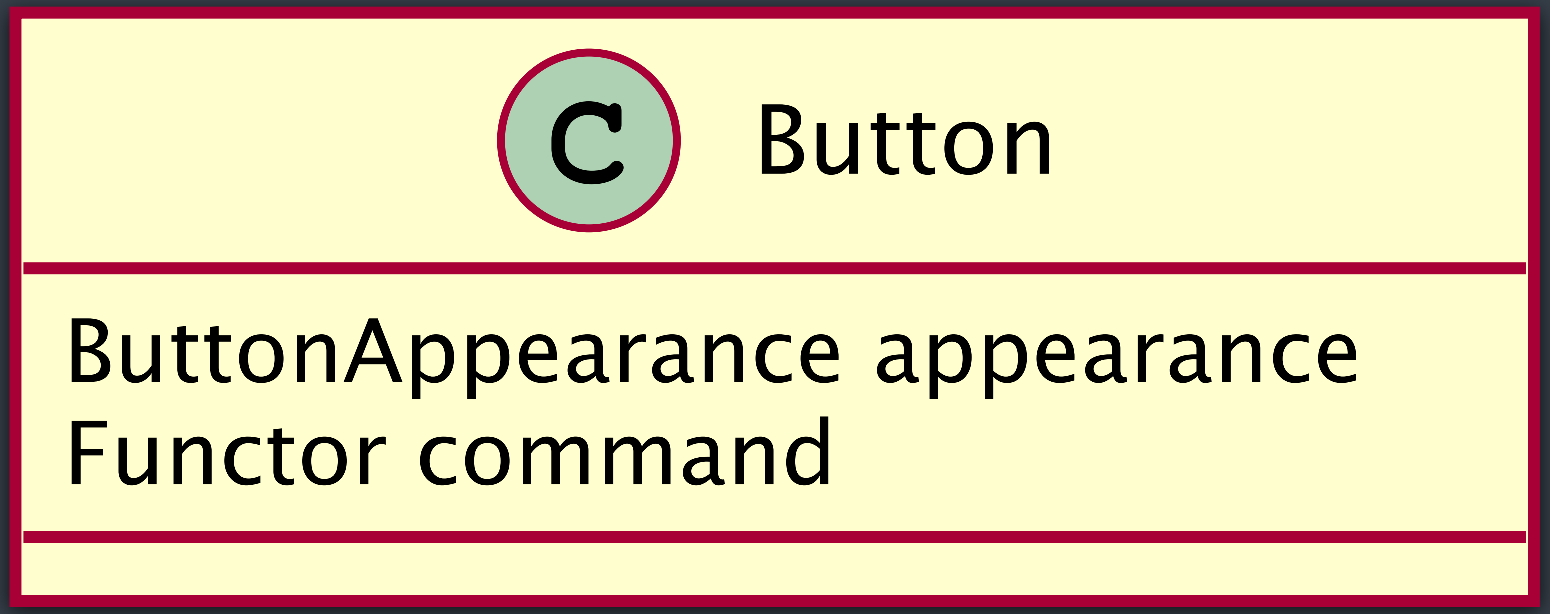

Button
TravelButton
RescueButton
StopButton
FloorButton
ElevatorButton

option 2

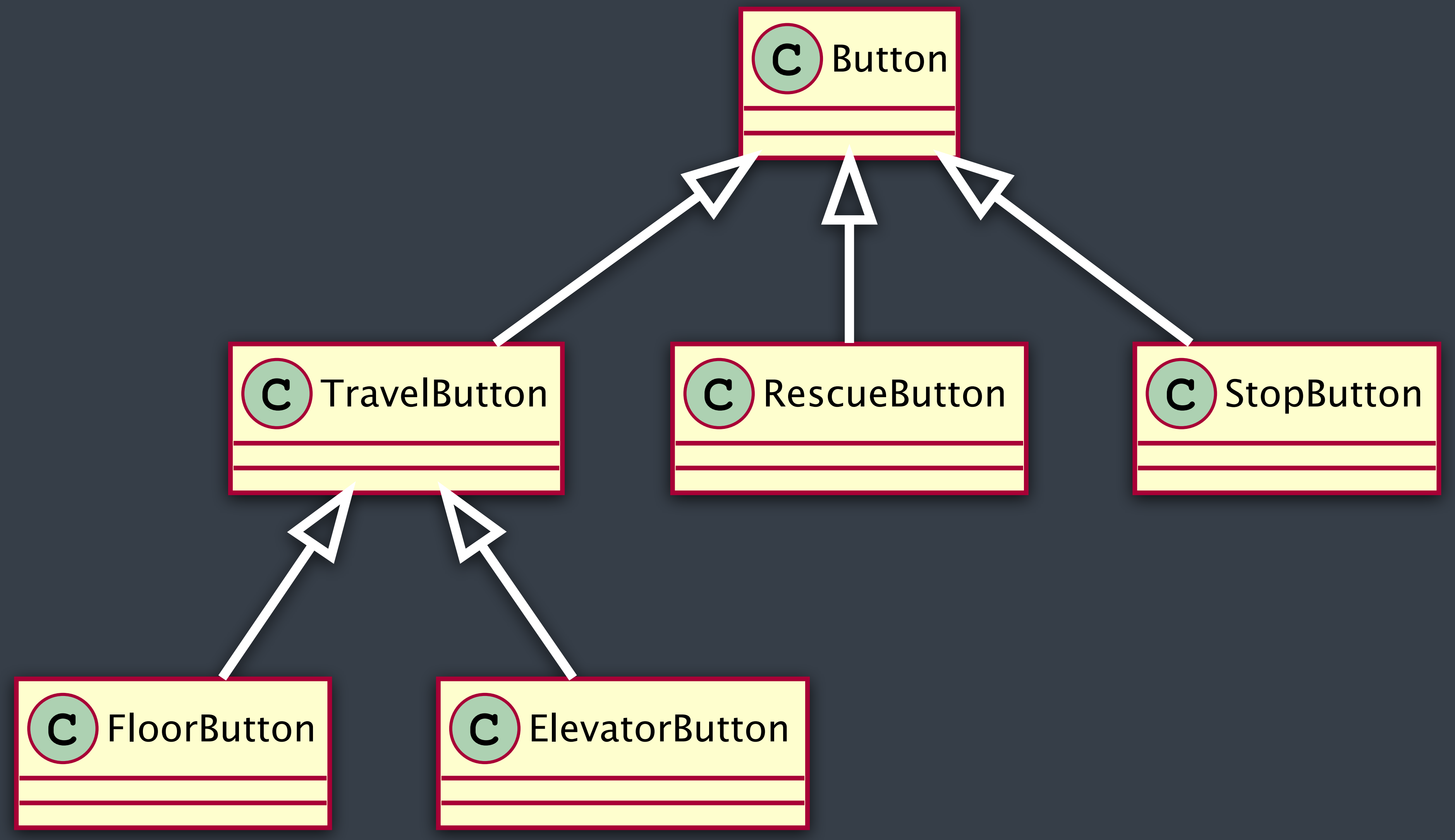


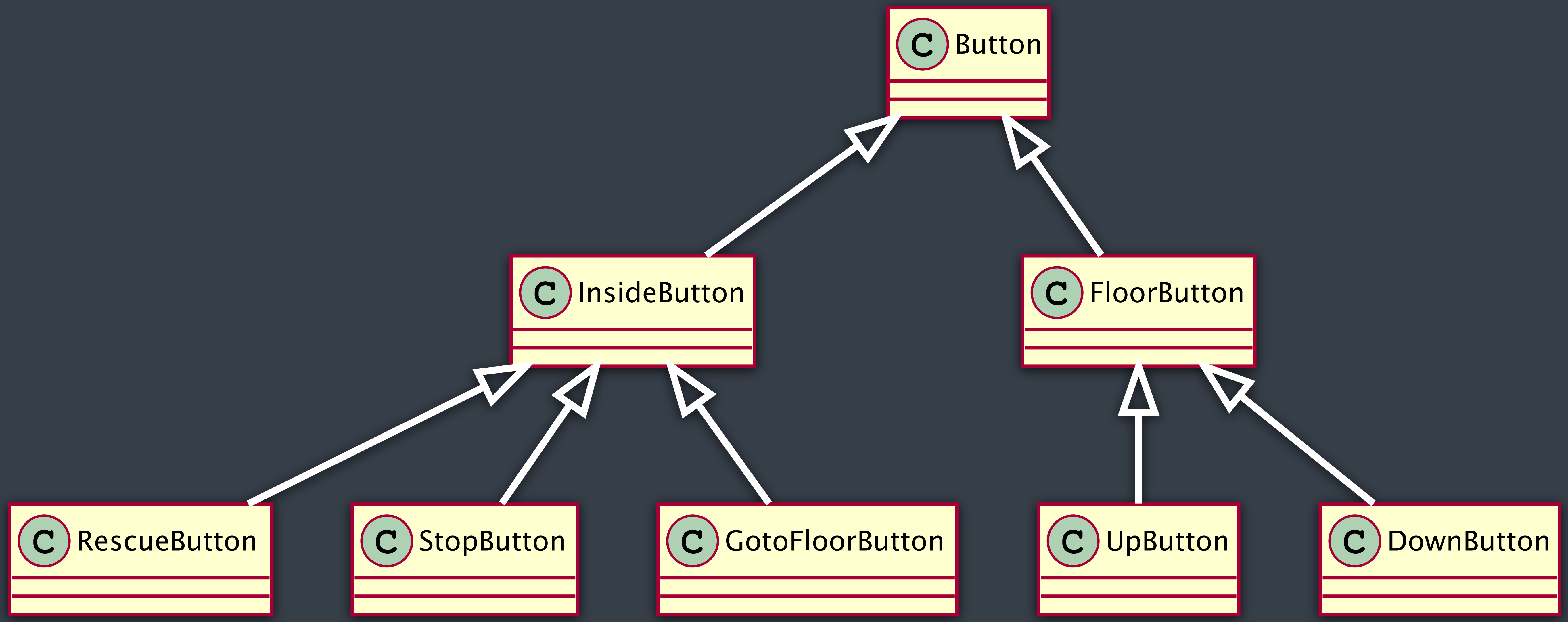
C Button
C InsideButton
C FloorButton
C RescueButton
C StopButton
C GotoFloorButton
C UpButton
C DownButton

option 3



what is **real-world**?





what is a **Button**?

elevator

arcade game

mechanical

keyboard

UI button

old bell button

...

in real world

concepts are **building blocks of thoughts**

fuzzy generalisations

by similitude

in real world

there is no “**the Button**”

in software

concepts are sets of instances

sharp distinctions

algebraically constructed

IS-A in **real world**

$\nexists$ inside-button **IS-A** button

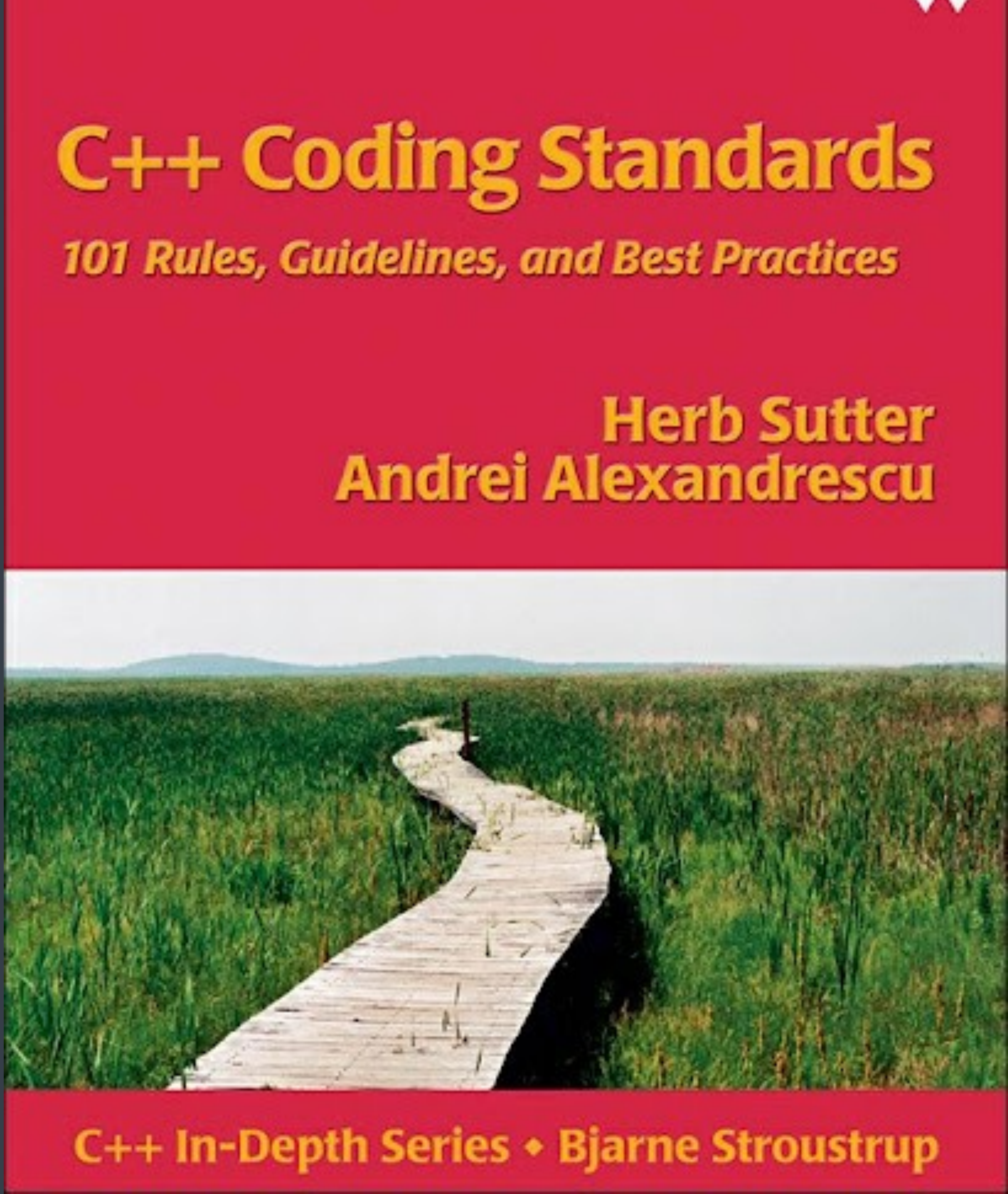
IS-A in **software**

1. reuse data layout & methods — composition & typing

2. substitutability — Liskov Substitution Principle

The "is-a" description of public inheritance is misunderstood when people use it to draw irrelevant real-world analogies: A square "is-a" rectangle (mathematically) but a Square is not a Rectangle (behaviorally). Consequently, instead of "is-a," we prefer to say "**works-like-a**" (or, if you prefer, "**usable-as-a**") to make the description less prone to misunderstanding.

Herb Sutter, Andrei Alexandrescu


C++ Coding Standards
101 Rules, Guidelines, and Best Practices
Herb Sutter
Andrei Alexandrescu
C++ In-Depth Series • Bjarne Stroustrup

public inheritance
is **substitutability**

it has nothing to do with “real-world”
IS-A — nothing but a metaphor

why use inheritance?

~~model real-world concepts~~

reuse members

subtyping

implement an interface

3

Liskov Substitution Principle

Liskov Substitution Principle

Subtype requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Barbara H. Liskov, Jeannette M., *A behavioral notion of subtyping*, ACM Transactions on Programming Languages and Systems, 1994

Liskov Substitution Principle

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Barbara H. Liskov, *Data Abstraction and Hierarchy*, 1988

Liskov Substitution Principle

If for each object o1 of type S there is an object o2 of type T such that for **all programs** P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Barbara H. Liskov, *Data Abstraction and Hierarchy*, 1988

sounds good, but…

it **doesn’t work** in a strict sense

hard to work in a relaxed sense

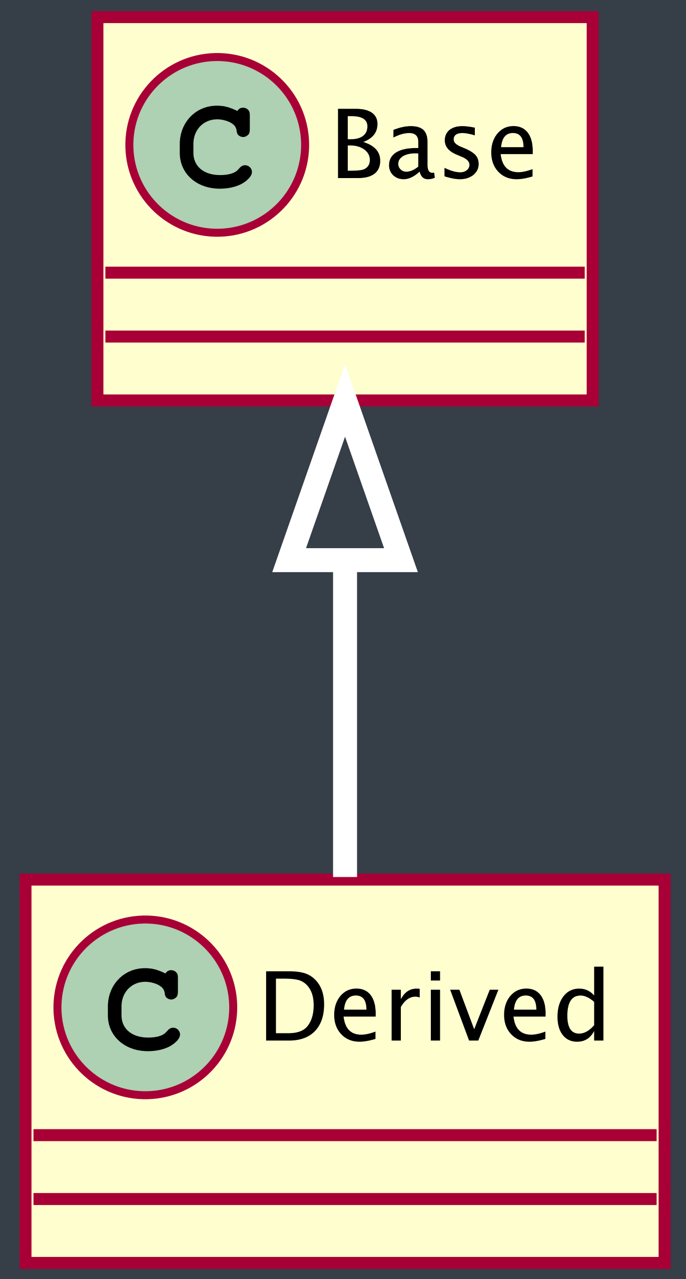
increases complexity, not reduce it

LSP, **strict** sense

$\phi(Base) = true$

$\phi(Derived) == true$?

$\phi(x) = true$, iff $x == Base$



Liskov Substitution Principle

Subtype requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Barbara H. Liskov, Jeannette M., *A behavioral notion of subtyping*, ACM Transactions on Programming Languages and Systems, 1994

```
void lspFailure(Base& poly) {
    assert(typeid(poly) == typeid(Base));
}
```

LSP and subtyping

mathematically, LSP doesn’t allow subtyping

relaxed LSP

$\phi(x) =$ behaves exactly like Base

returns the same things as Base does

calls exactly the same functions as Base does

same performance as Base

returns a subset of results that Base returns

has the same invariants as Base

...

what are the properties?

we have to survey all the code

⇒ **hard**

LSP can break

when changing Derived

if we don’t know all the properties of Base

when changing Base/clients

if we don’t know all the assumptions for all derived

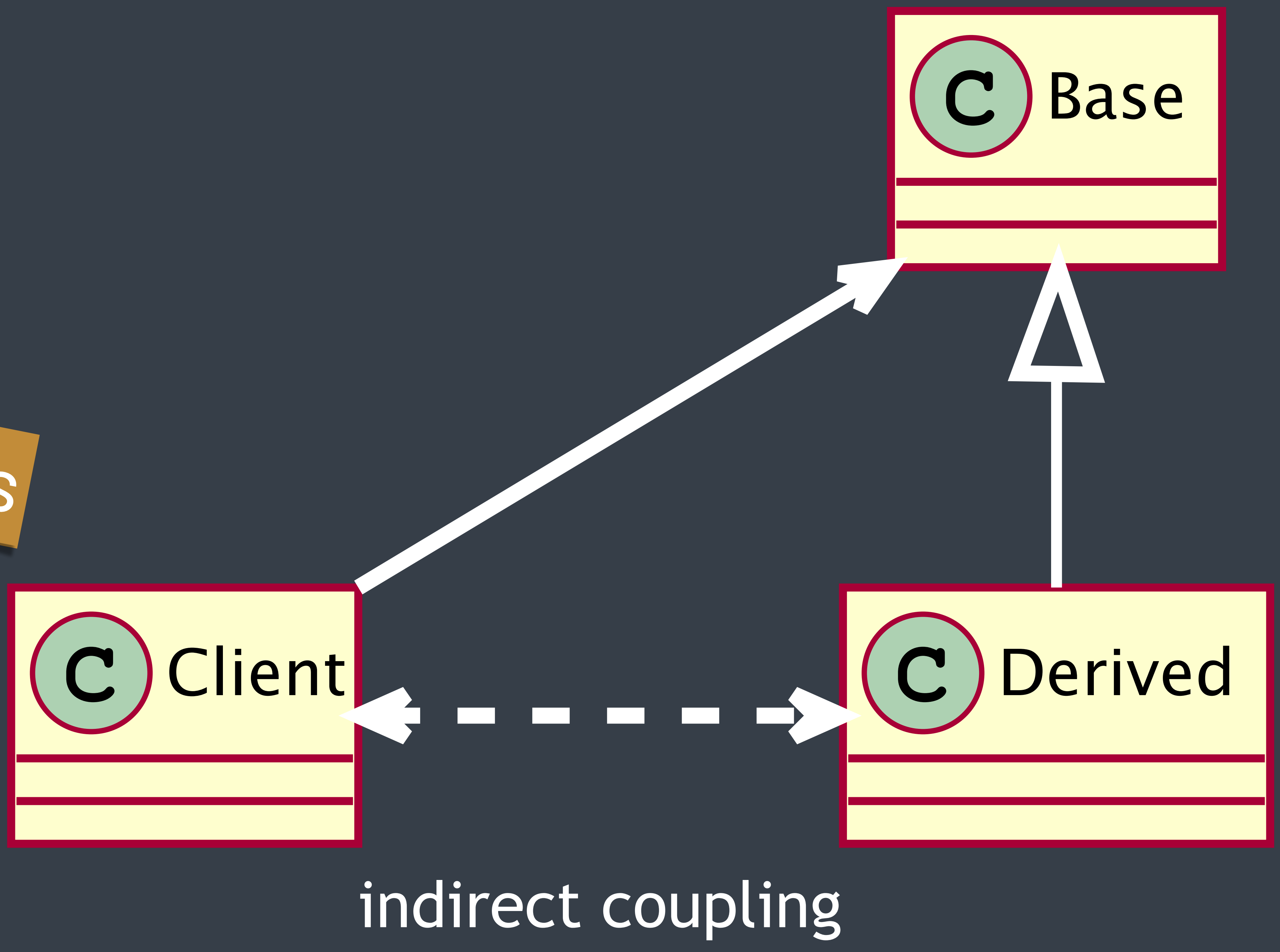
example 1

```
struct Button {
    virtual void push(bool on) { isPushed_ = on; }
    virtual bool isPushed() const { return isPushed_; }
private:
    bool isPushed_{false};
};

// OLD code, in a different module
void clientCode(Button& btn) {
    btn.push(true);
    assert(btn.isPushed()); // Should be ON
}

// NEW code
struct ButtonWithSafety : Button {
    // Only push the button if the safety button is also pushed
    void push(bool on) override { Button::push(on && safety_.isPushed()); }

    Button safety_;
};
```

example 2

```
// OLD code, in some distant module, not directly visible near Button
struct ButtonWithTimer : Button {
    void push(bool on) override {
        Button::push(on);
        // Button automatically unpressed after 1 second
        if (on)
            timer.start(1s, [this] { Button::push(false); })
    }
}

// NEW code, based on observed behaviour of Button
void clientCode(Button& btn) {
    btn.push(true);
    oldVal = btn.isPushed();
    std::this_thread::sleep_for(1s);
    assert(oldVal == btn.isPushed()); // FAILURE
}
```

source of problems



different modules

open-closed principle

increases complexity

NOT **an abstraction**

abstraction reduces complexity

LSP

it **doesn’t work** in a strict sense

hard to work in a relaxed sense

increases complexity, not reduce it

why use inheritance?

model real-world concepts

reuse members

~~subtyping~~

implement an interface

Inheritance and Friendship

4

real-life analogy

children are closer than friends

friendship impact

access to all members

(with a bit of care) does not change interface of class

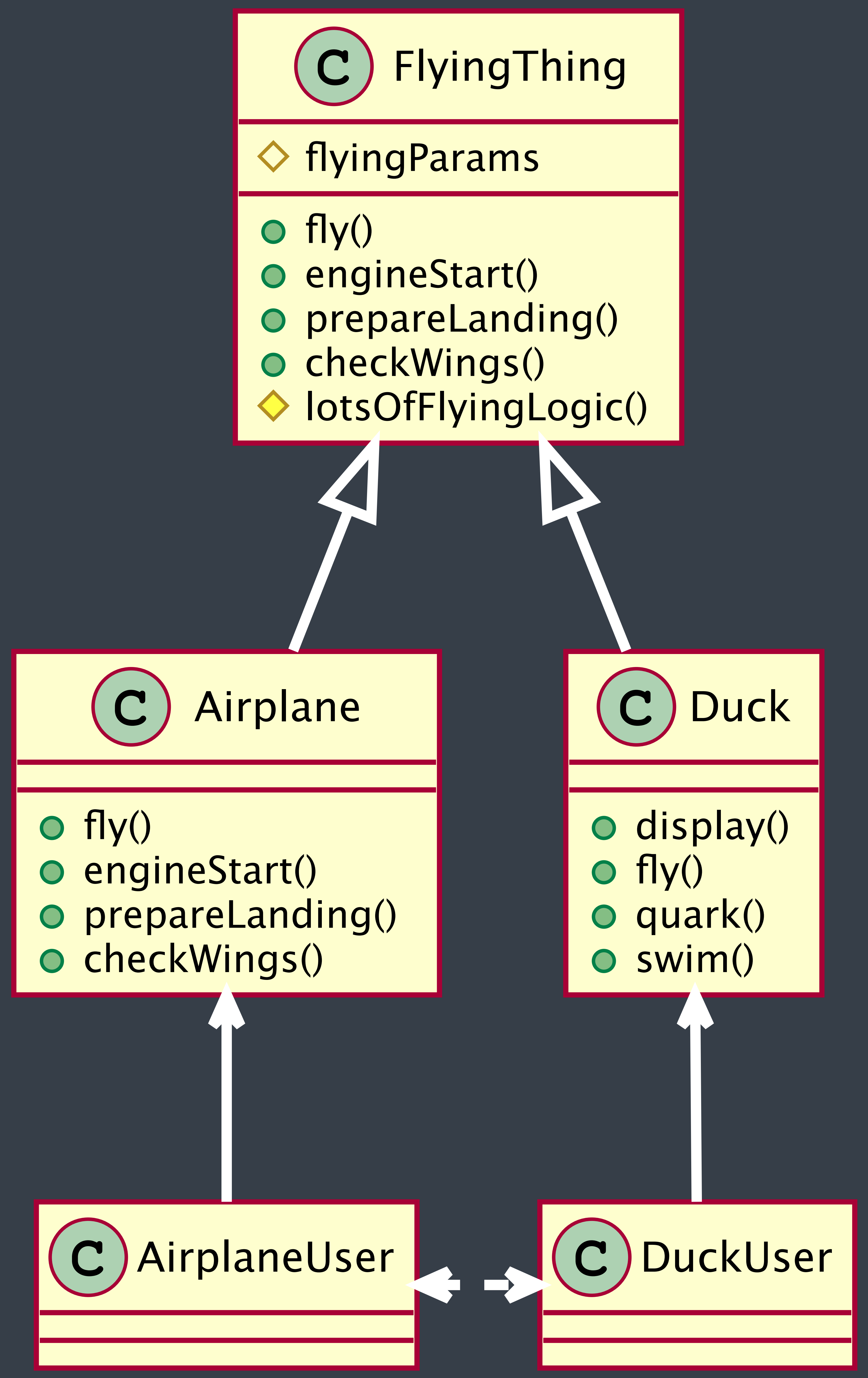
inheritance impact

access to most members

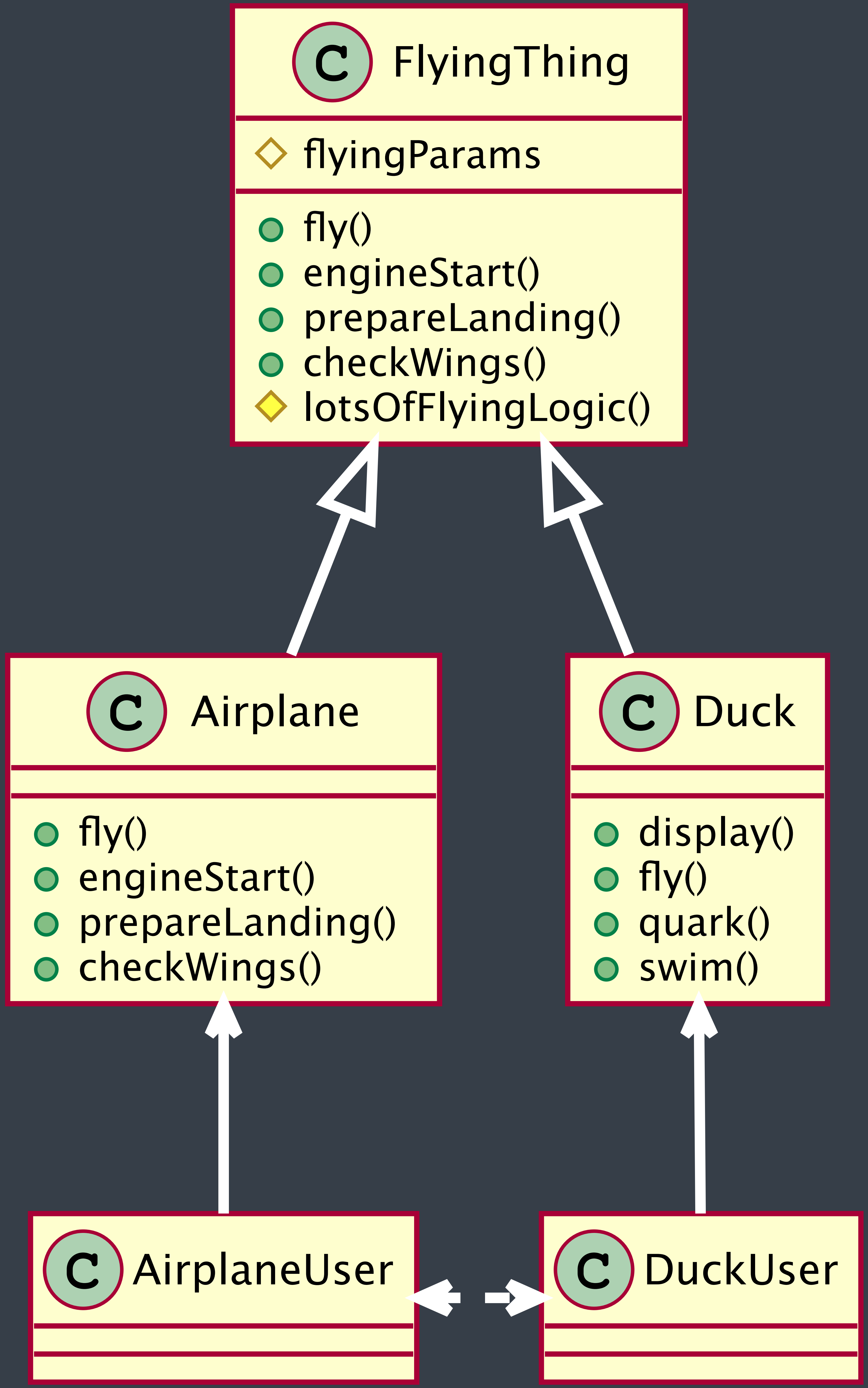
can change class invariants

=> affects all clients

example


C FlyingThing
◇ flyingParams
○ fly()
○ engineStart()
○ prepareLanding()
○ checkWings()
◆ lotsOfFlyingLogic()
C Airplane
○ fly()
○ engineStart()
○ prepareLanding()
○ checkWings()
C Duck
○ display()
○ fly()
○ quark()
○ swim()
C AirplaneUser
C DuckUser

change inertia



inheritance is

stronger than friendship

Inheritance vs Composition

5

A pox on the ISA relationship.
It’s been misleading and damaging for decades.
Inheritance is not ISA.
Inheritance is the **redeclaration of functions and variables** in a sub-scope.
No more. No less.

Robert C. Martin

inheritance

redeclaration

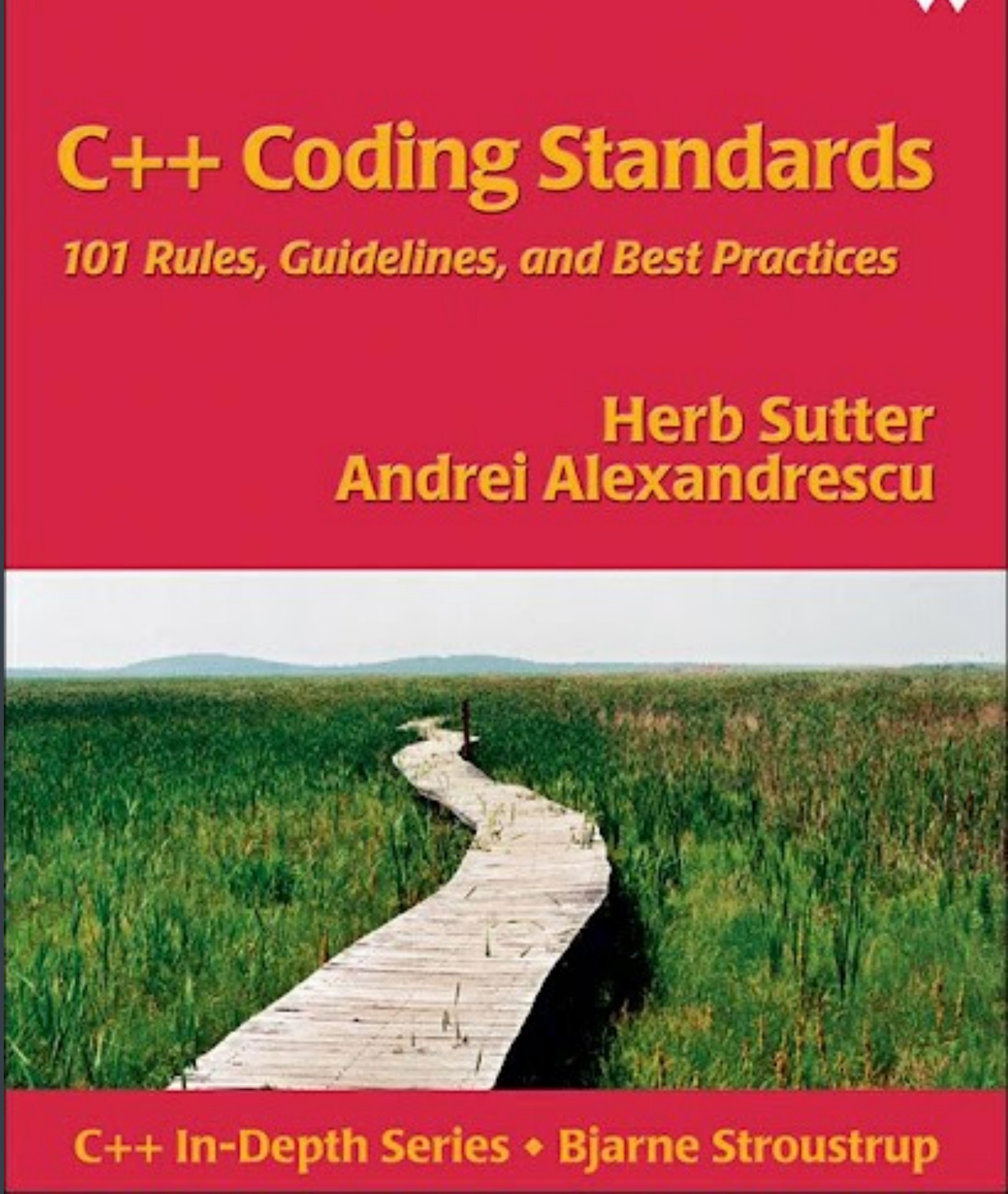
~~subtyping~~

composition

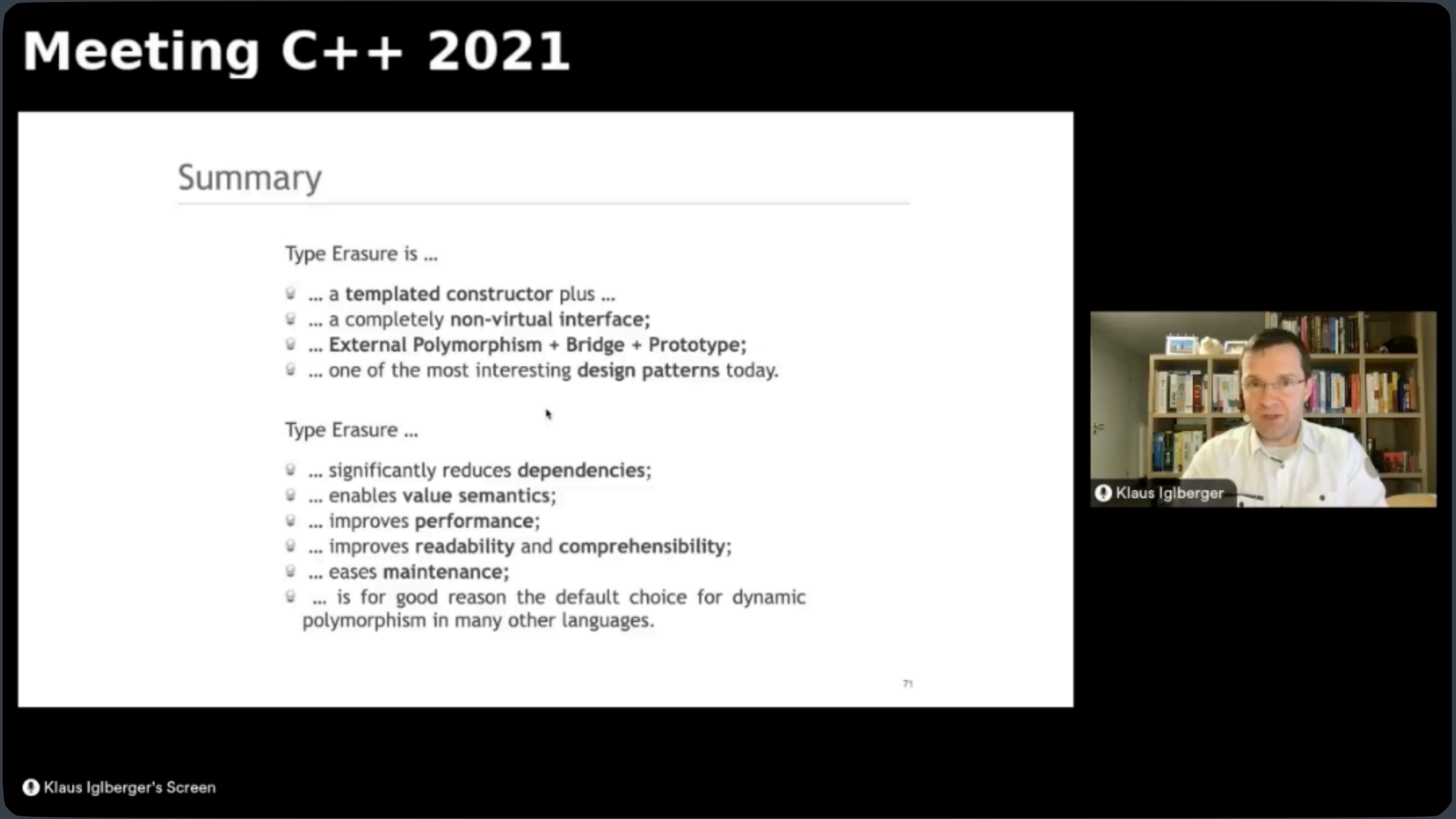
redeclaration can be **abstracted** out

Inheritance is often overused, even by experienced developers. A sound rule of software engineering is to minimize coupling: If a relationship can be expressed in more than one way, use the weakest relationship that's practical.

Herb Sutter, Andrei Alexandrescu


C++ Coding Standards
101 Rules, Guidelines, and Best Practices
Herb Sutter
Andrei Alexandrescu
C++ In-Depth Series • Bjarne Stroustrup

prefer **composition** to inheritance



Summary

Type Erasure is ...

- ... a **templated constructor** plus ...
- ... a completely **non-virtual interface;**
- ... **External Polymorphism + Bridge + Prototype;**
- ... one of the most interesting **design patterns** today.

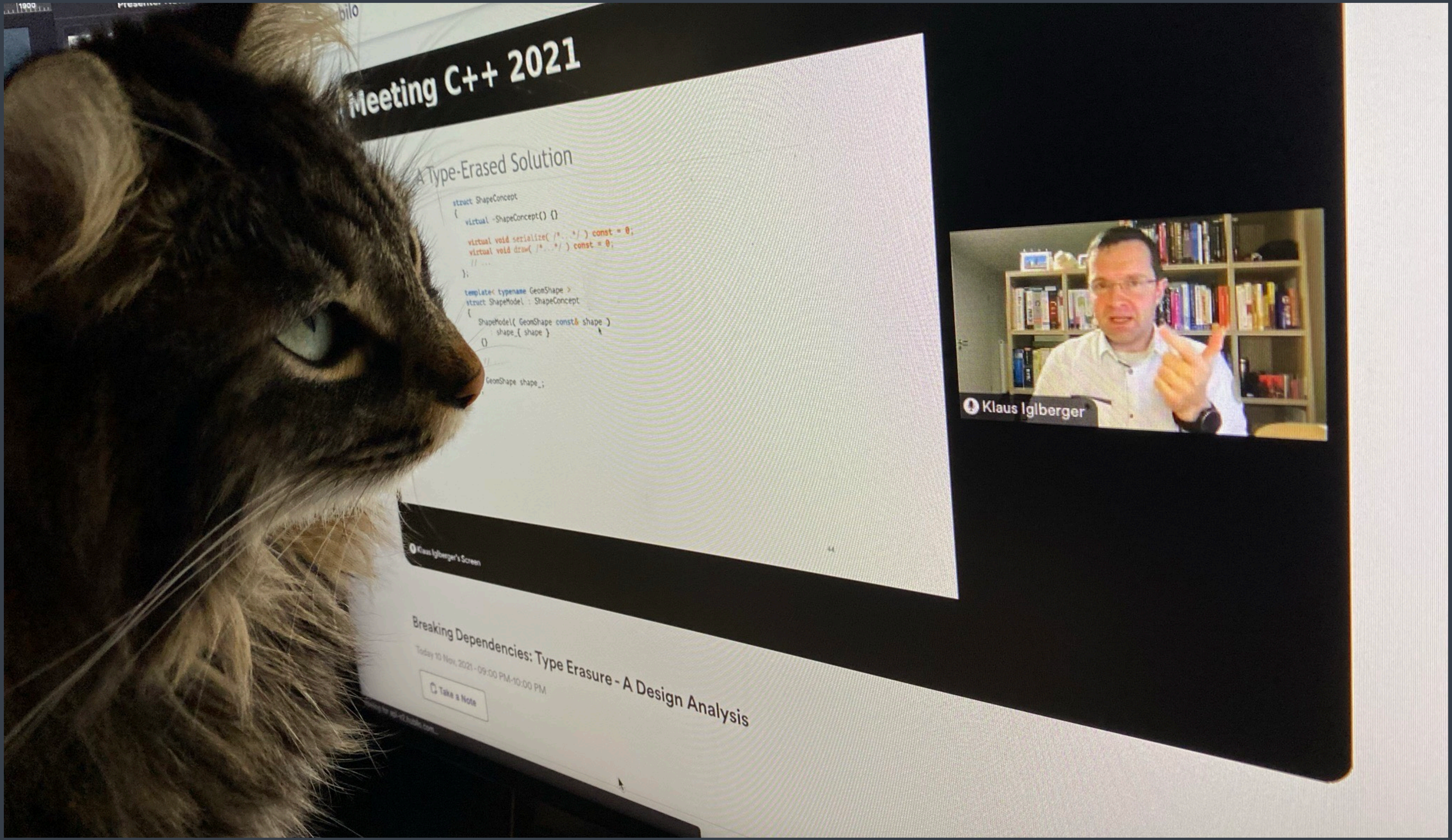
Type Erasure ...

- ... significantly reduces **dependencies;**
- ... enables **value semantics;**
- ... improves **performance;**
- ... improves **readability** and **comprehensibility;**
- ... eases **maintenance;**
- ... is for good reason the default choice for dynamic polymorphism in many other languages.

71

Klaus Iglberger, *Breaking Dependencies: Type Erasure - A Design Analysis*

@LucT3o


Meeting C++ 2021
A Type-Erased Solution
Klaus Iglberger
Klaus Iglberger's Screen
Breaking Dependencies: Type Erasure - A Design Analysis
Today 10 Nov, 2021 · 09:00 PM-10:00 PM
Take a Note

```
struct ShapeConcept
{
   virtual ~ShapeConcept() {}

   virtual void serialize( /*...*/ ) const = 0;
   virtual void draw( /*...*/ ) const = 0;
   // ...
};

template< typename GeomShape >
struct ShapeModel : ShapeConcept
{
   ShapeModel( GeomShape const& shape )
      : shape_{ shape }
   {}

   // ...

   GeomShape shape_;
```

Klaus Iglberger, *Breaking Dependencies: Type Erasure - A Design Analysis*

why use inheritance?

model real-world concepts

~~reuse members~~

subtyping

implement an interface

6

Putting things together

When to use and when to avoid inheritance

why use inheritance?

~~model real-world concepts~~

~~reuse members~~ composition

~~subtyping~~

implement an interface

inheritance

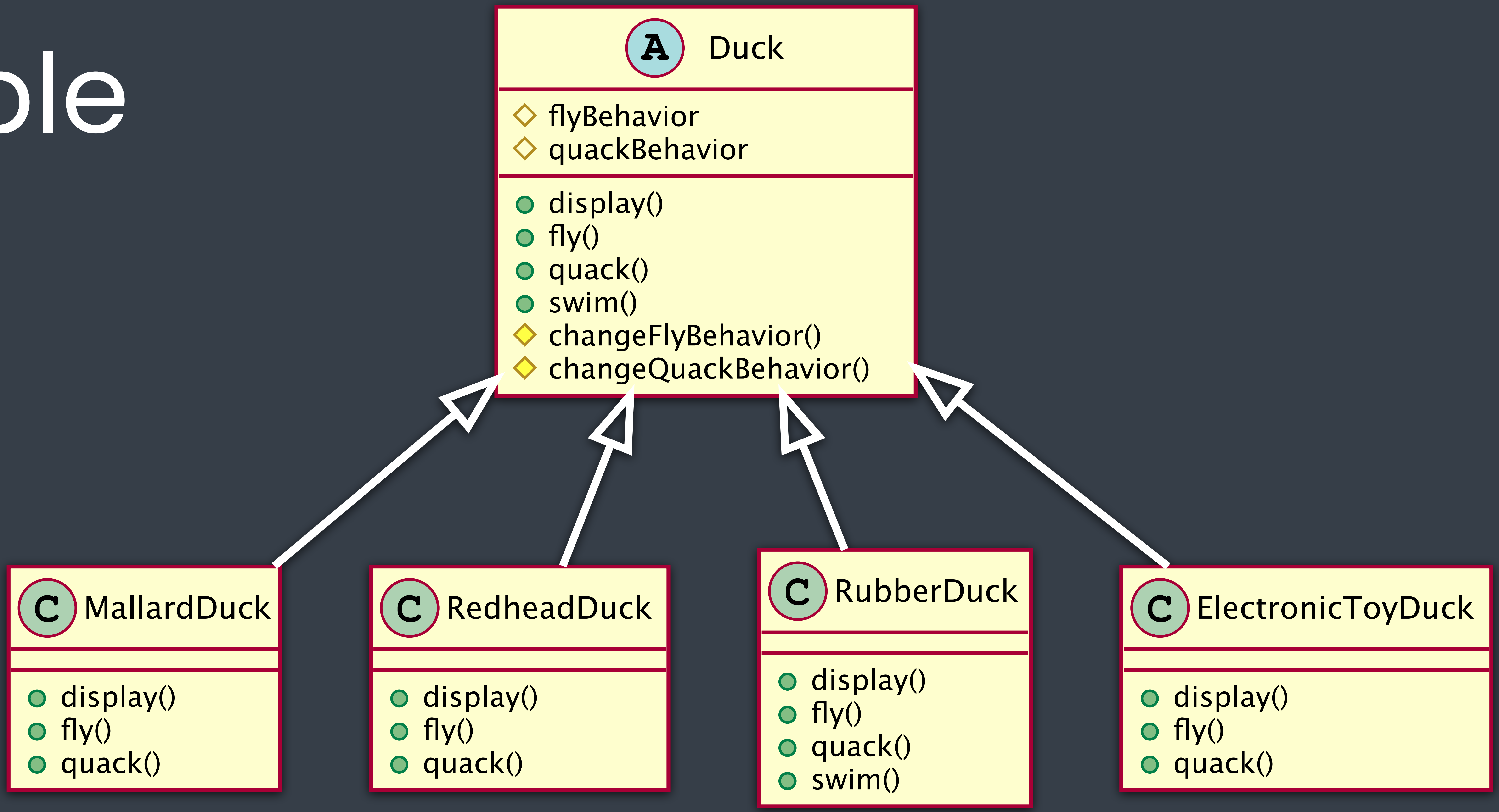
use with **interfaces**
replace by **composition**

interfaces

ok with **LSP**

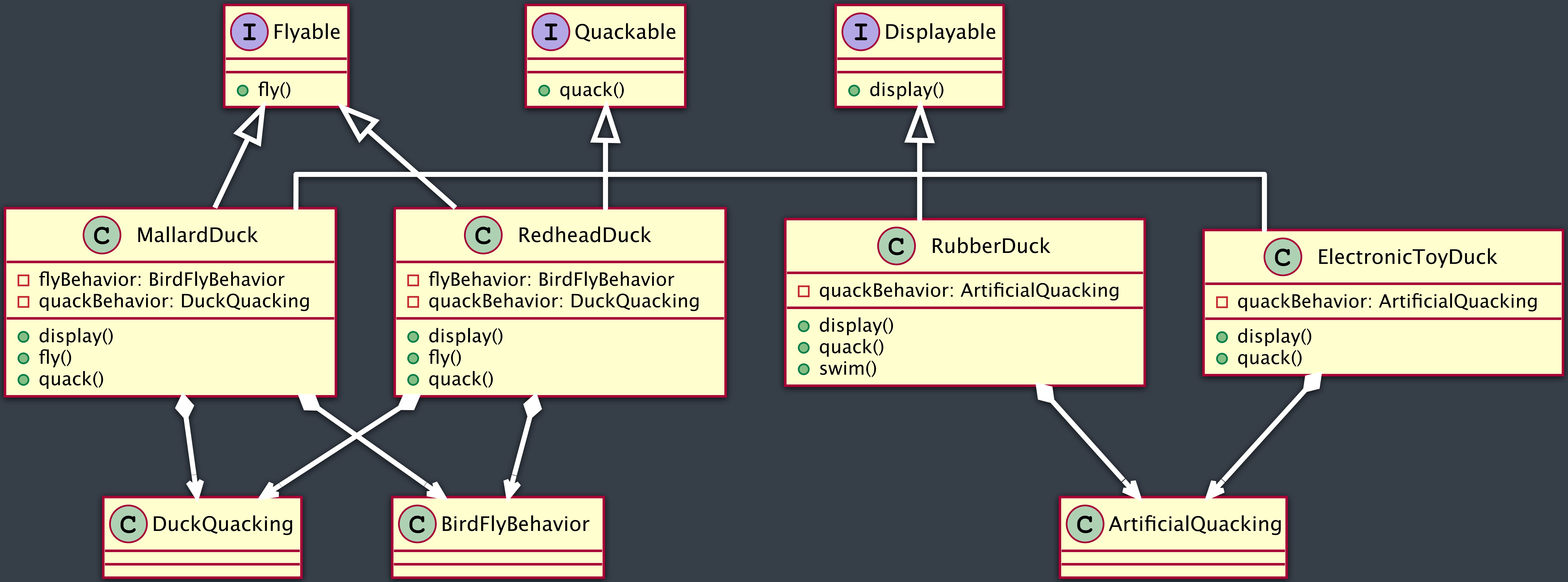
SOLID player

example



A Duck
flyBehavior
quackBehavior
display()
fly()
quack()
swim()
changeFlyBehavior()
changeQuackBehavior()
C MallardDuck
display()
fly()
quack()
C RedheadDuck
display()
fly()
quack()
C RubberDuck
display()
fly()
quack()
swim()
C ElectronicToyDuck
display()
fly()
quack()

A Duck
display()
fly()
quack()
swim()
C MallardDuck
flyBehavior: BirdFlyBehavior
quackBehavior: DuckQuacking
display()
fly()
quack()
C RedheadDuck
flyBehavior: BirdFlyBehavior
quackBehavior: DuckQuacking
display()
fly()
quack()
C RubberDuck
flyBehavior: NoFlyBehavior
quackBehavior: ArtificialQuacking
display()
fly()
quack()
swim()
C ElectronicToyDuck
flyBehavior: NoFlyBehavior
quackBehavior: ArtificialQuacking
display()
fly()
quack()
C BirdFlyBehavior
C DuckQuacking
C NoFlyBehavior
C ArtificialQuacking


I Flyable
fly()
I Quackable
quack()
I Displayable
display()
C MallardDuck
flyBehavior: BirdFlyBehavior
quackBehavior: DuckQuacking
display()
fly()
quack()
C RedheadDuck
flyBehavior: BirdFlyBehavior
quackBehavior: DuckQuacking
display()
fly()
quack()
C RubberDuck
quackBehavior: ArtificialQuacking
display()
quack()
swim()
C ElectronicToyDuck
quackBehavior: ArtificialQuacking
display()
quack()
C DuckQuacking
C BirdFlyBehavior
C ArtificialQuacking

interface guidelines

one for each type of client

invariants directed by the clients

interfaces **decouple**

concrete **classes** from **users**

proper abstraction

why use inheritance?

~~model real-world concepts~~

~~reuse members~~

composition

~~subtyping~~

implement an interface

inheritance is

… full of inconsistencies

… overrated

… overused

Thank You

@LucT3o

lucteo.ro

LUCIAN RADU TEODORESCU
GARMIN