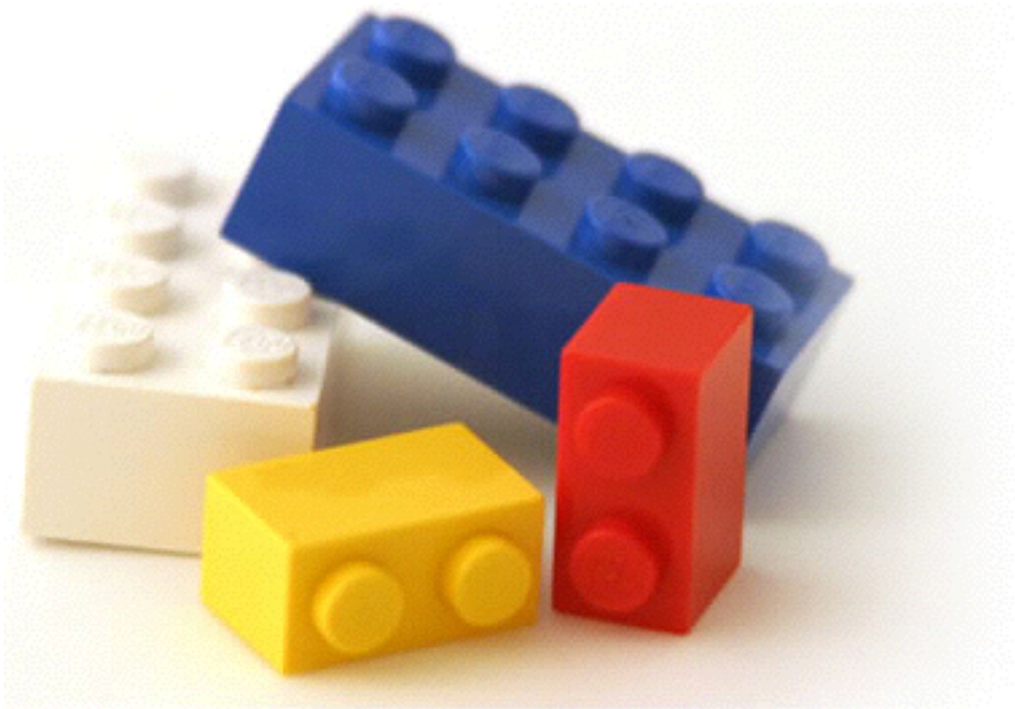
Object Oriented Design Principles

Pragmatic Programmer:
Eliminate Effects Between Unrelated Things –
design components that are:
self-contained,
independent,
and have a single, well-defined purpose

Hmm. . .

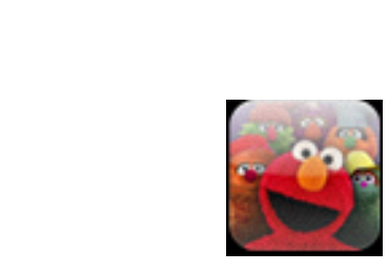
https://i.redd.it/pf4cx9cd2siy.png
posted by u/Mtelling

Software Design – Modularity



The goal of all software design techniques is to break a complicated problem into simple pieces.

Why Modularity?












Why Modularity?

- Minimize Complexity
- Reusability
- Extensibility
- Portability
- Maintainability
- …

What is a good modular Design?

- There is no one “right answer” with design but there are lots of wrong answers
- A bad design is hard to change, hard to maintain
- Applying heuristics/principles can provide insights and lead to a good design
- A MAJOR goal is to: **Minimise the Effects of Change**

Source: [*Gamma et all, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995*]

- Program an Interface not an Implementation
- Favor Composition Versus Inheritance
- Find what varies and encapsulate it

Source: [*R. Martin, "Agile Software Development, Principles, Patterns, and Practices", Prentice-Hall, 2002*]

- Dependency-Inversion Principle
- Liskov Substitution Principle
- Open-Closed Principle
- Interface-Segregation Principle
- Reuse/Release Equivalency Principle
- Common Closure Principle
- Common Reuse Principle
- Acyclic Dependencies Principle
- Stable Dependencies Principle
- Stable Abstraction Principle

Design Principles

Source: [*Larman, "Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development", Prentice-Hall, 20*

- Design principles are codified in the GRASP Pattern
- GRASP (Pattern of General Principles in Assigning Responsabilities)
- Assign a responsibility to the information expert
- Assign a responsability so that coupling remains low
- Assign a responsability so that cohesion remains high
- Assign responsabilities using polymorphic operations
- Assign a highly cohesive set of responsabilities to an artificial class that does not represent anything in the problem domain (when you want to
- Don't talk to strangers (Law of Demeter)

Source: [*Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", Commununication of ACM, 1972*]

- Information Hiding
- Modularity

Source: [*Hunt, Thomas, "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley, 1999*]

- DRY – Don't Repeat yourself
- Make it easy to reuse
- Design for Orthogonality
- Eliminate effects between unrelated things
- Program close to the problem domain
- Minimize Coupling between Modules
- Design Using Services
- Always Design for Concurrency
- Abstractions Live Longer than details

Source: [*Lieberherr,Holland, "Assuring Good Style for Object-Oriented Programs", IEEE Software, September 1989*]

- Law of Demeter

Source: [*Raymond, "Art of Unix Programming", Addison-Wesley, 2003*]

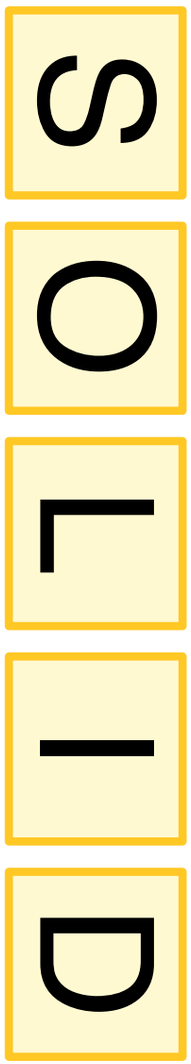
Pragmatic Programmer:

Eliminate Effects Between Unrelated Things –
design components that are:
self-contained,
independent,
and have a single, well-defined purpose

Principles & Heuristics for modular Design

http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29

- **S O L I D** (one nice combo of principles)
  - **S**ingle Responsibility Principle
    - (High Cohesion, Low Coupling)
  - **O**pen/Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principles



S O L I D

The Single Responsibility Principle

a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

http://en.wikipedia.org/wiki/Single_responsibility_principle

S O L I D

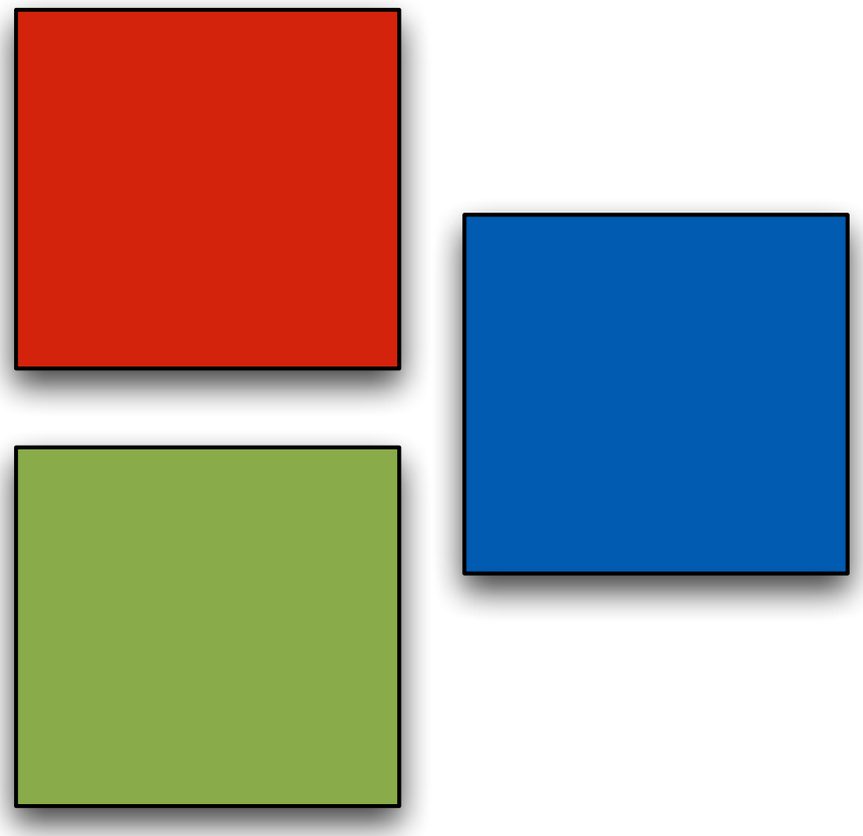
High Cohesion

http://en.wikipedia.org/wiki/Cohesion_(computer_science)

- Cohesion refers to how closely the functions in a module are related
- Modules should contain functions that logically belong together
  - Group functions that work on the same data
- Classes should have a **single responsibility.**

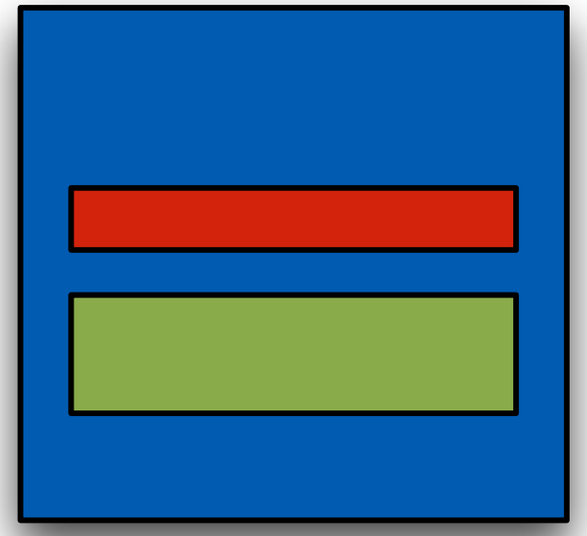
S O L I D

Cohesion (try to increase)



methods that serve the given class tend to be similar in many aspects

versus



- The functionalities embedded in a class, accessed through its methods, have little in common.
- Methods carry out many varied activities, often using coarsely-grained or unrelated sets of data.

The Or-Check

- A class description that describes a class in terms of *alternatives* is probably not a class, but a set of classes

"A ClassRoom is a location where students attend tutorials OR labs"

May need to be modeled as two classes: TutorialRoom and ComputerLab

S O L I D

Types of Cohesion

Coincidental cohesion (bad)
Coincidental cohesion is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. a “Utilities” class).

Logical cohesion (bad)
Logical cohesion is when parts of a module are grouped because they logically are categorized to do the same thing, even if they are different by nature (e.g. grouping all mouse and keyboard input handling routines).

Temporal cohesion
Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Procedural cohesion
Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).

Communicational cohesion
Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on the same record of information).

Sequential cohesion (very good)
Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).

Functional cohesion (best)
Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module (e.g. tokenizing a string of XML).

S O L I D

Functional Cohesion (best!)

- Functionally cohesive objects do ONE thing only.
- Good because they're easy to reuse, and understand
- Warning: functionally cohesive can proliferate and get very tiny (overly fine grained, overly numerous)
- Maximising functional cohesion will lead to classes that look like "Do-ers" (Logger, Driver…)

points from: http://it.toolbox.com/blogs/enterprise-solutions/design-principles-cohesion-16069

High or low cohesion?

```
public class EmailMessage {

    …

    public void sendMessage() {…}

    public void setSubject(String subj) {…}

    public void setSender(Sender sender) {…}

    public void login(String user, String passw) {…}

    ….

}
```

remember:
classes should be
“about” one thing

S O L I D

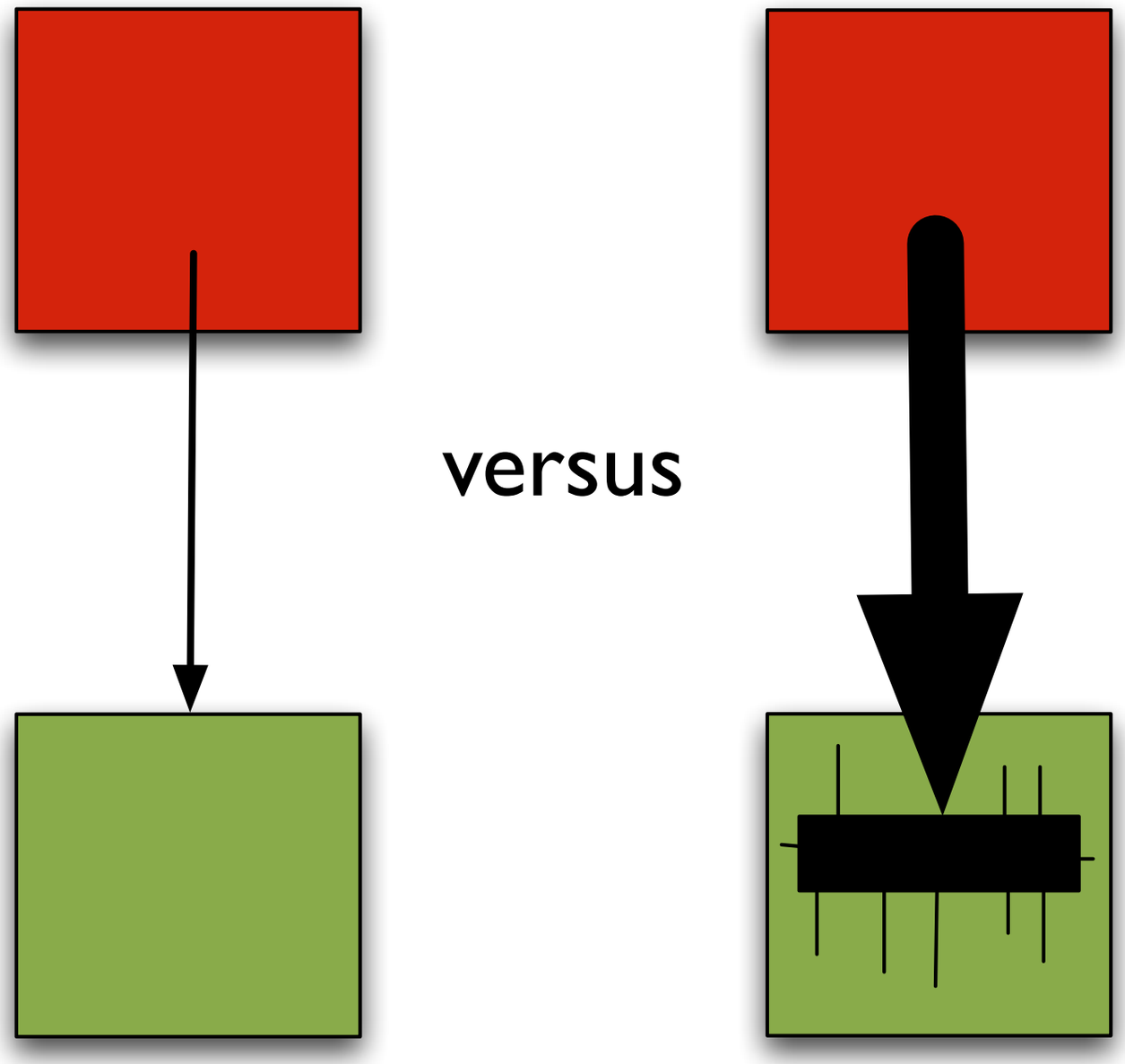
Loose Coupling

http://en.wikipedia.org/wiki/Coupling_(computer_science)

- Coupling assesses how tightly a module is related to other modules
- Goal is loose coupling:
  - modules should depend on as few other modules as possible
- Changes in modules should not impact other modules; easier to work with them separately

S O L I D

Coupling (try to decrease)



A change in one module usually forces a ripple effect of changes in other modules.

Assembly of modules might require more effort and/or time due to the increased inter-module dependency.

A particular module might be harder to reuse and/or test because dependent modules must be included.

S O L I D

Types of Coupling

Content coupling (high)
Content coupling is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module). Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.

Common coupling
Common coupling is when two modules share the same global data (e.g., a global variable). Changing the shared resource implies changing all the modules using it.

External coupling
External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.This is basically related to the communication to external tools and devices.

Control coupling
Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

Stamp coupling (Data-structured coupling)
Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it).
This may lead to changing the way a module reads a record because a field that the module doesn't need has been modified.

Data coupling
Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Message coupling (low)
This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing

No coupling
Modules do not communicate at all with one another.

S O L I D

Semantic coupling: *The most insidious kind of coupling occurs when one module makes use not of some syntactic element of another module but of some semantic knowledge of another module's inner workings.*

http://courses.cs.washington.edu/courses/cse403/07sp/assignments/Chapter5-Design.pdf, page 102

Semantic coupling is dangerous *because changing code in the used module can break code in the using module in ways that are completely* ***undetectable by the compiler****. When code like this breaks, it breaks in subtle ways that seem unrelated to the change made in the used module, which turns debugging into a Sisyphean task.*

The point of loose coupling *is that an effective module provides an additional level of abstraction—****once you write it, you can take it for granted****. It reduces overall program complexity and allows you to focus on one thing at a time. If using a module requires you to focus on more than one thing at once—knowledge of its internal workings, modification to global data, uncertain functionality—the abstractive power is lost and the module's ability to help manage complexity is reduced or eliminated.*

S O L I D

The OPEN/CLOSED principle

“software entities … should be open for extension, but closed for modification.”

http://en.wikipedia.org/wiki/Open/closed_principle

S O L I D

Open/Closed Principle

A class must be **closed** for internal change

But must be **open** for *extensions*

When designing classes, do not plan for brand new functionality to be added by modifying the core of the class.
Instead, design your class so that extensions can be made in a modular way, to provide new functionality by leveraging the power of the inheritance facilities of the language, or through pre-accommodated addition of methods.

S O L I D

Open/Closed Example

```
class Drawing {
  public void drawAllShapes(List<IShape> shapes) {
      for (IShape shape : shapes) {
          if (shape instanceof Square()) {
              drawSquare((Square) shape);
          } else if (shape instanceof Circle) {
              drawCircle((Circle) shape));
  } } }

  private void drawSquare(Square square) {..}
// draw the square… lots of stuff here
  private void drawCircle(Circle square) {..}
// draw the circle… lots of stuff here
}
```

This class assumes developers will modify the drawSquare and drawCircle methods directly to change their behaviour. This results in what looks like unplanned change!

```
class Drawing {
    public void drawAllShapes(List<IShape>
shapes) {
        for (IShape shape : shapes) {
            shape.draw(); } } }

interface IShape {
    public void draw();}

class Square implements IShape {
    public void draw() {  // draw the square  }}
```

this class has made specialising the shape draw method much more straightforward (also indicating that developers see this potential change coming!)

S O L I D

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 Liskov and Jeanette Wing, A Behavioral Notion of Subtyping, ACM Transactions on Programming Languages and Systems, Vol 16, No 6. November 1994, Pages 1811-1841.]

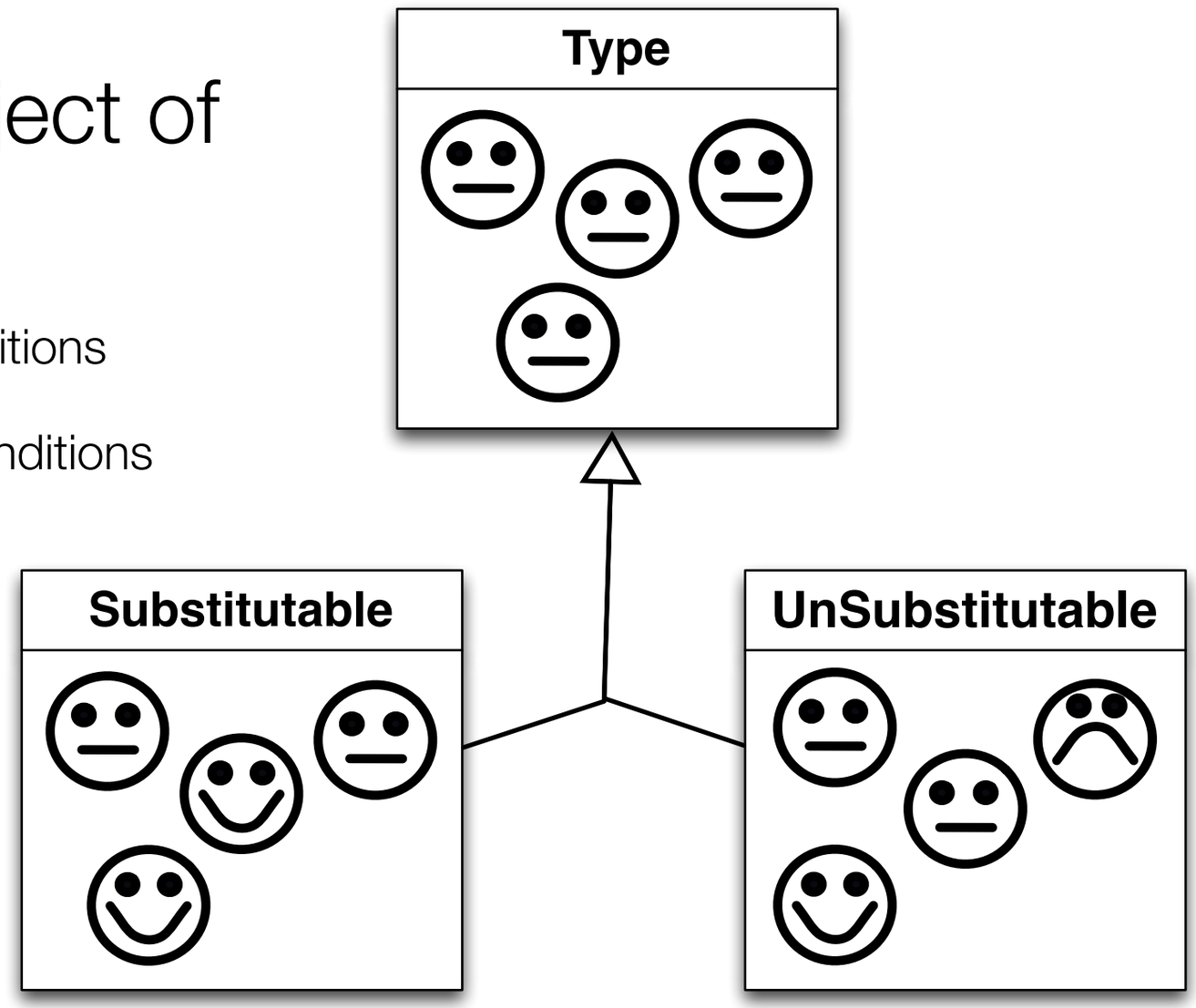
if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

http://en.wikipedia.org/wiki/Liskov_substitution_principle

S O L I D

Liskov Substitution Principle

- An object of a superclass should always be substitutable by an object of a subclass
  - Subclass has same or weaker preconditions
  - Subclass has same or stronger postconditions

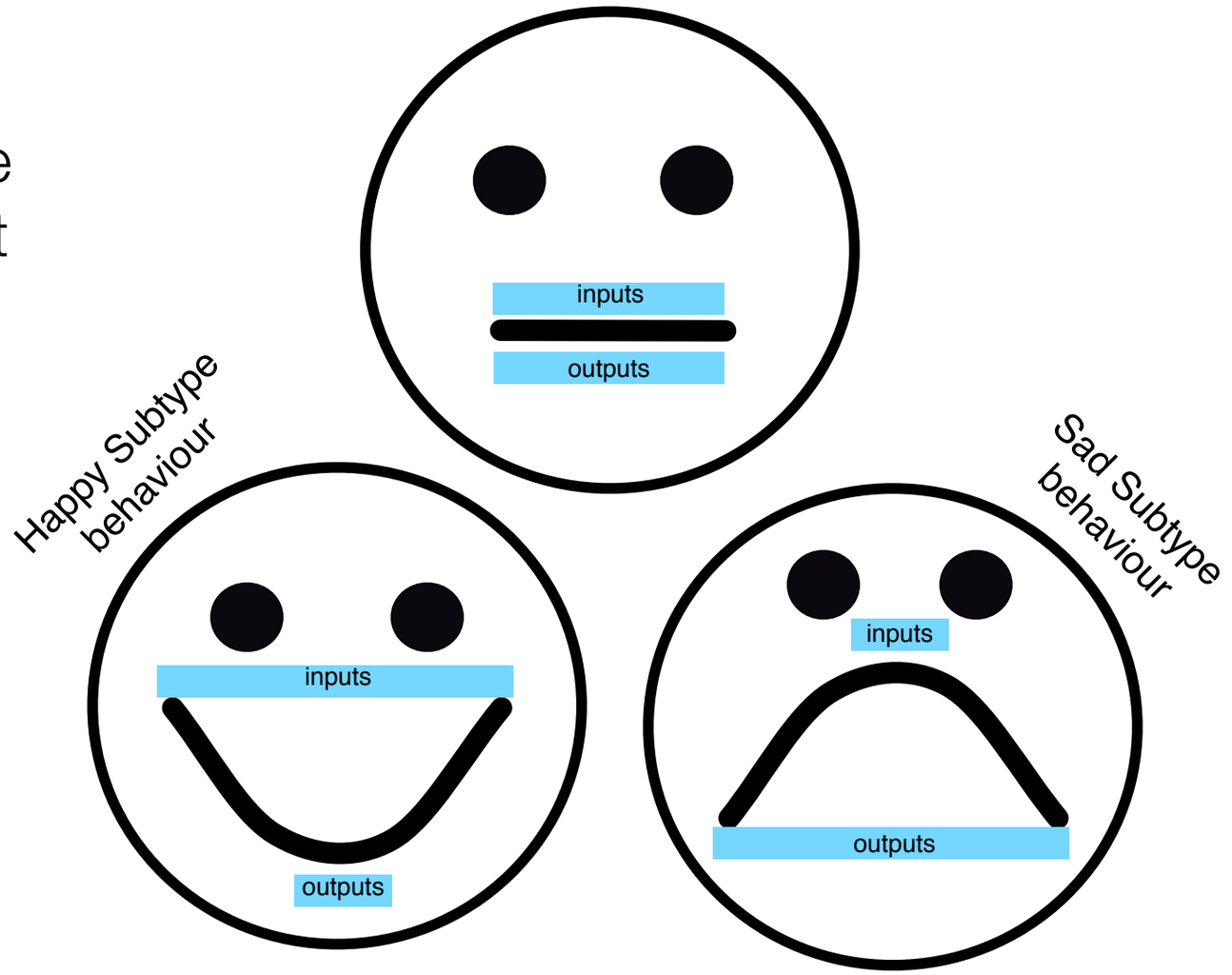


S O L I D

Liskov Substitution Principle

- When you are overriding a method, you want it to be useful in *all* of the situations in which its parent would be useful (more is okay too) (must accept at least the same range of inputs aka “same or weaker precondition”).

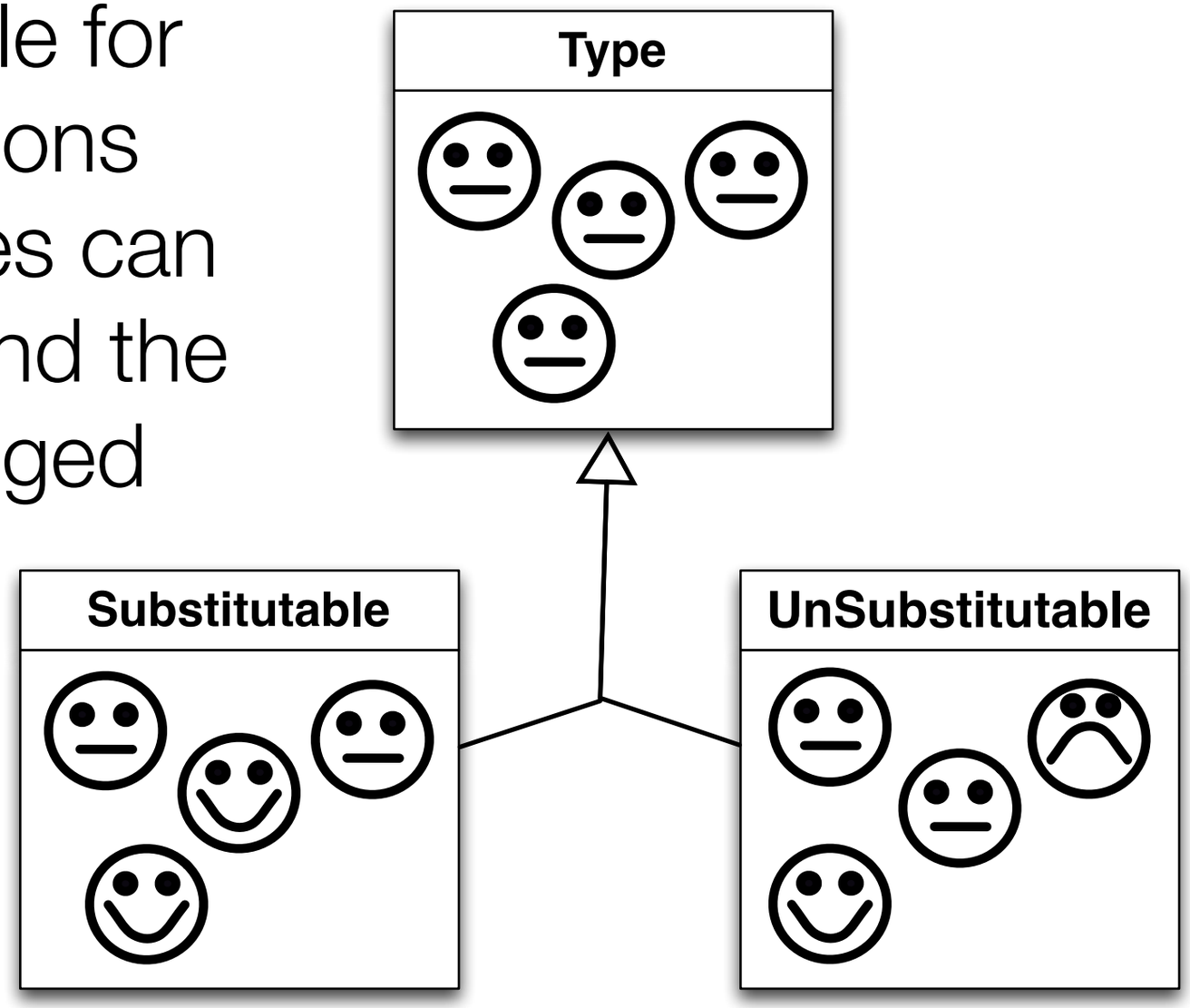
- And you don’t want it to break the caller by doing something extra or unexpected, or out of range (smaller range is fine) (must not produce a wider range of effects, aka “same or stronger postconditions”)



S O L I D

Liskov Substitution Principle

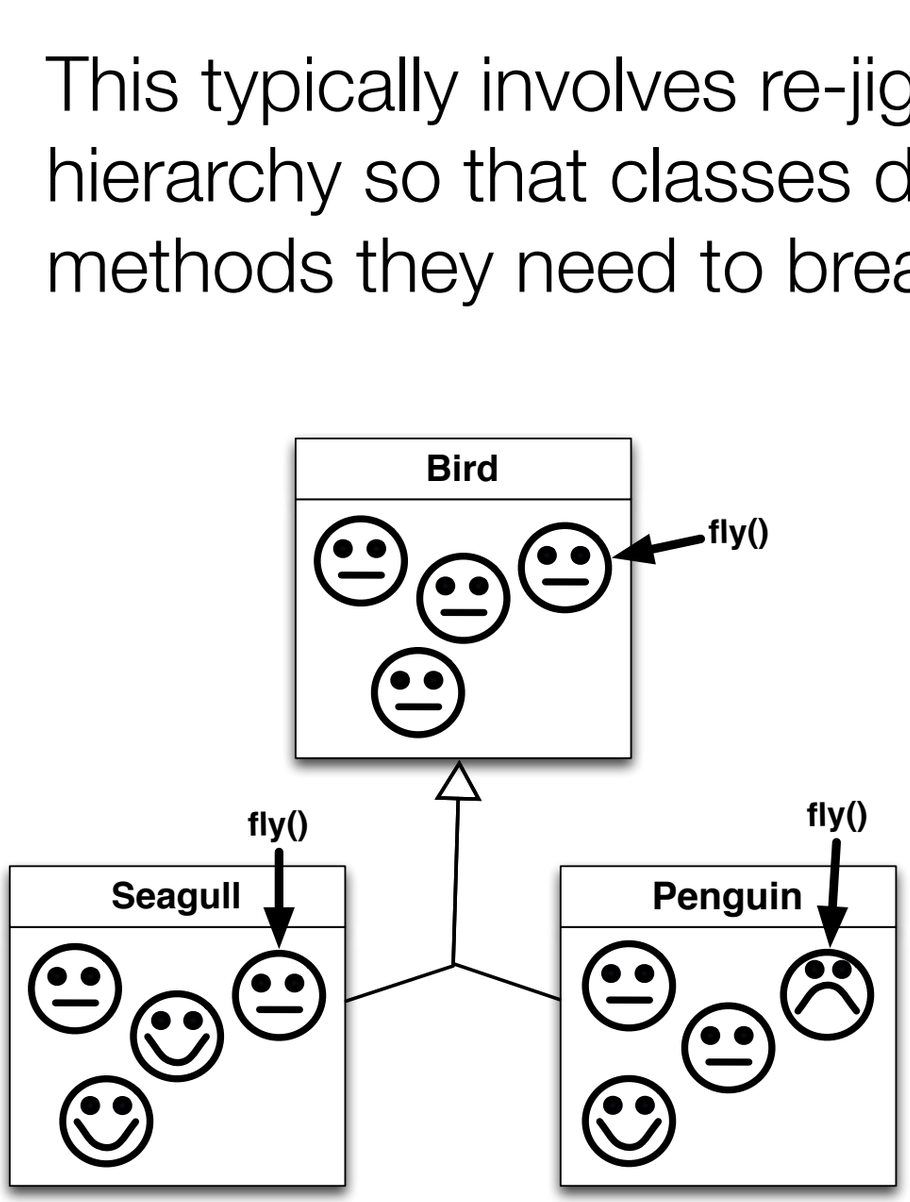
“It is only when derived types are completely substitutable for their base types that functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.”

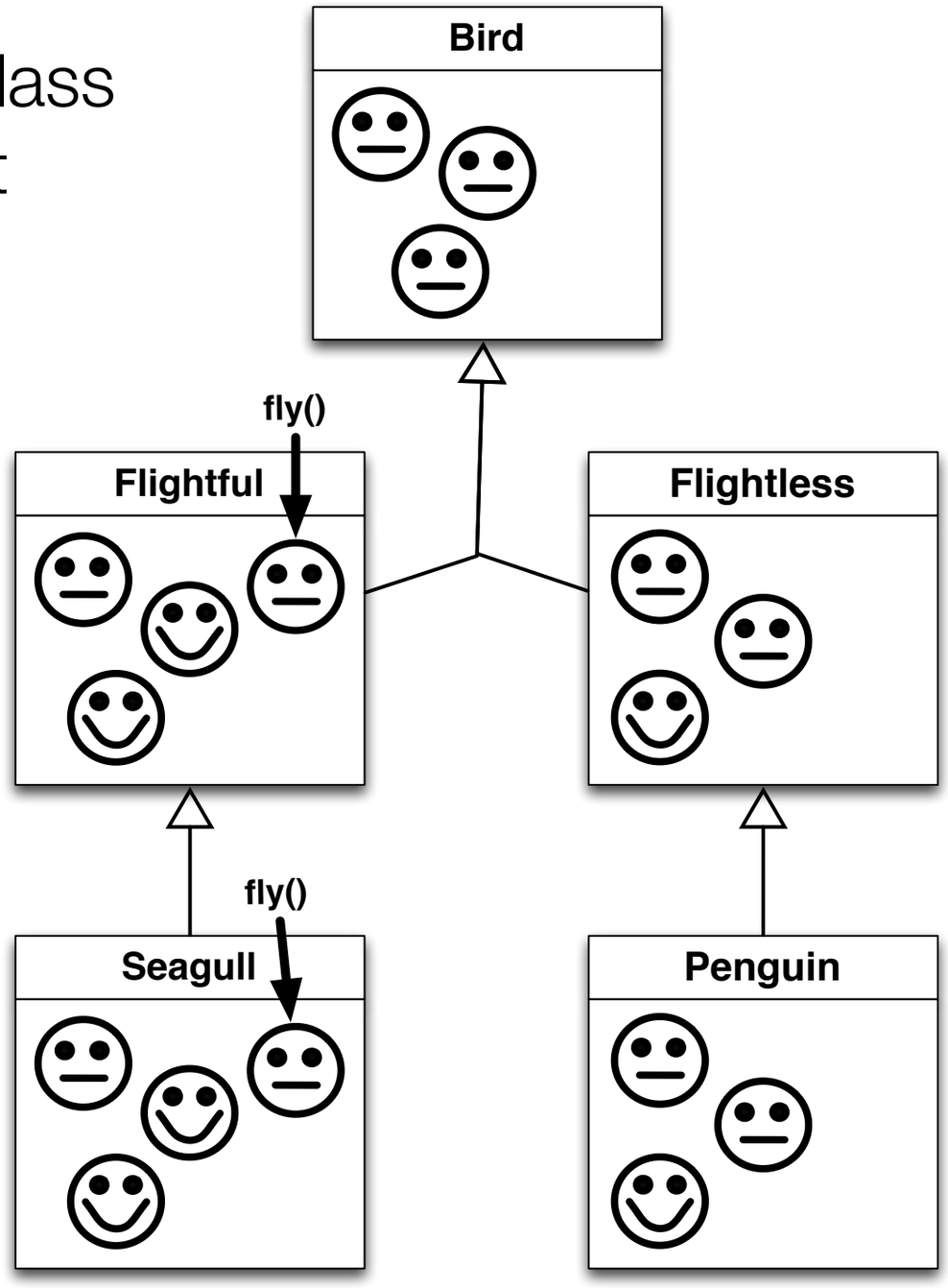


S O L I D

Fixing violations of LSP

This typically involves re-jigging the class hierarchy so that classes don't inherit methods they need to break





"flightful" and example stolen from here:
http://www.tomdalling.com/blog/software-design/solid-class-design-the-liskov-substitution-principle/

S O L I D

The Interface Segregation Principle

“many client-specific interfaces are better than one general-purpose interface.”

http://en.wikipedia.org/wiki/Interface_segregation_principle

S O L I D

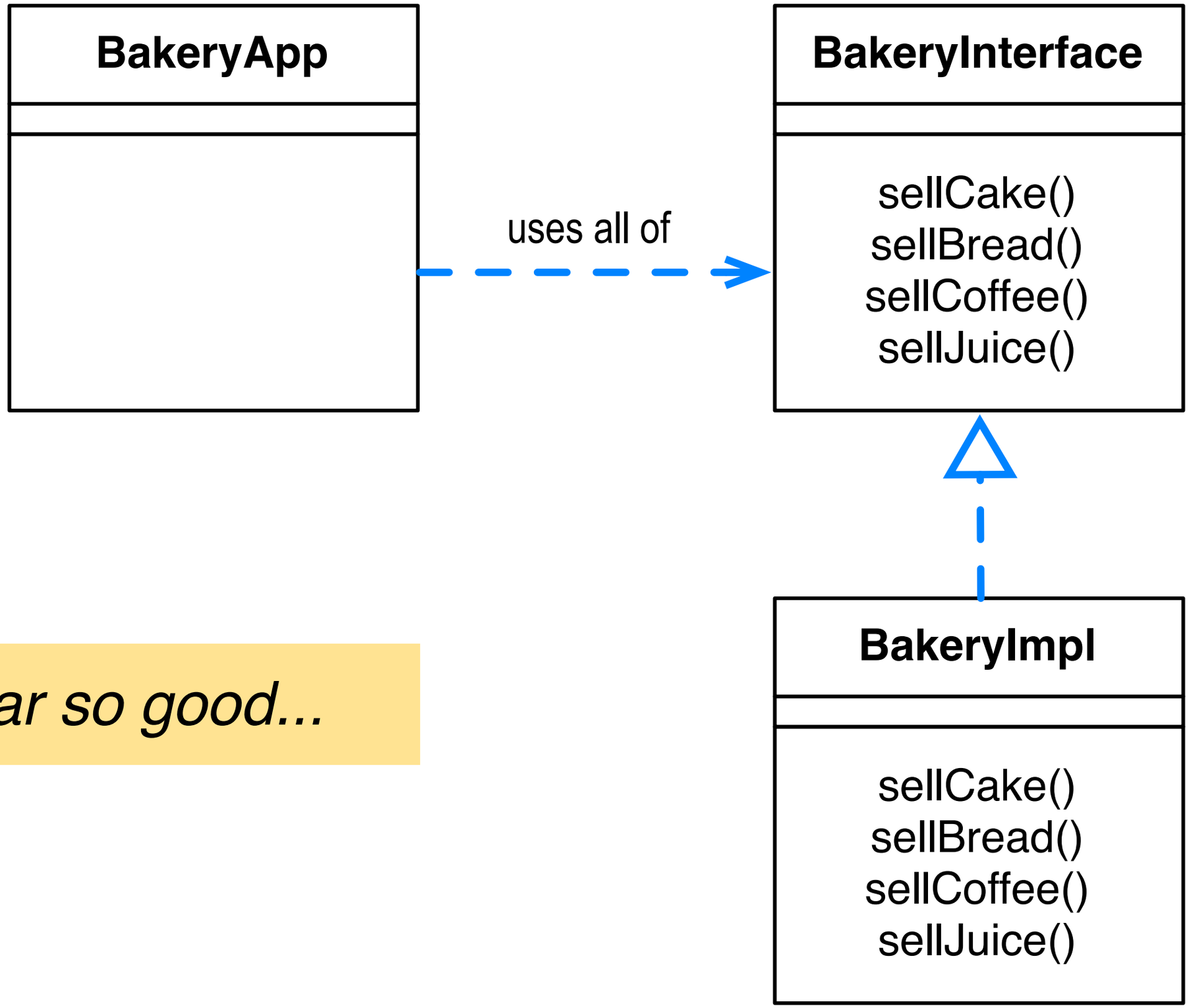
The Interface Segregation Principle

- *“no client should be forced to depend on methods it does not use”*

- A move towards role-based interfaces
- clients need only know about the methods that are of interest to them.
- This relates strongly to the concept of high cohesion

S O L I D

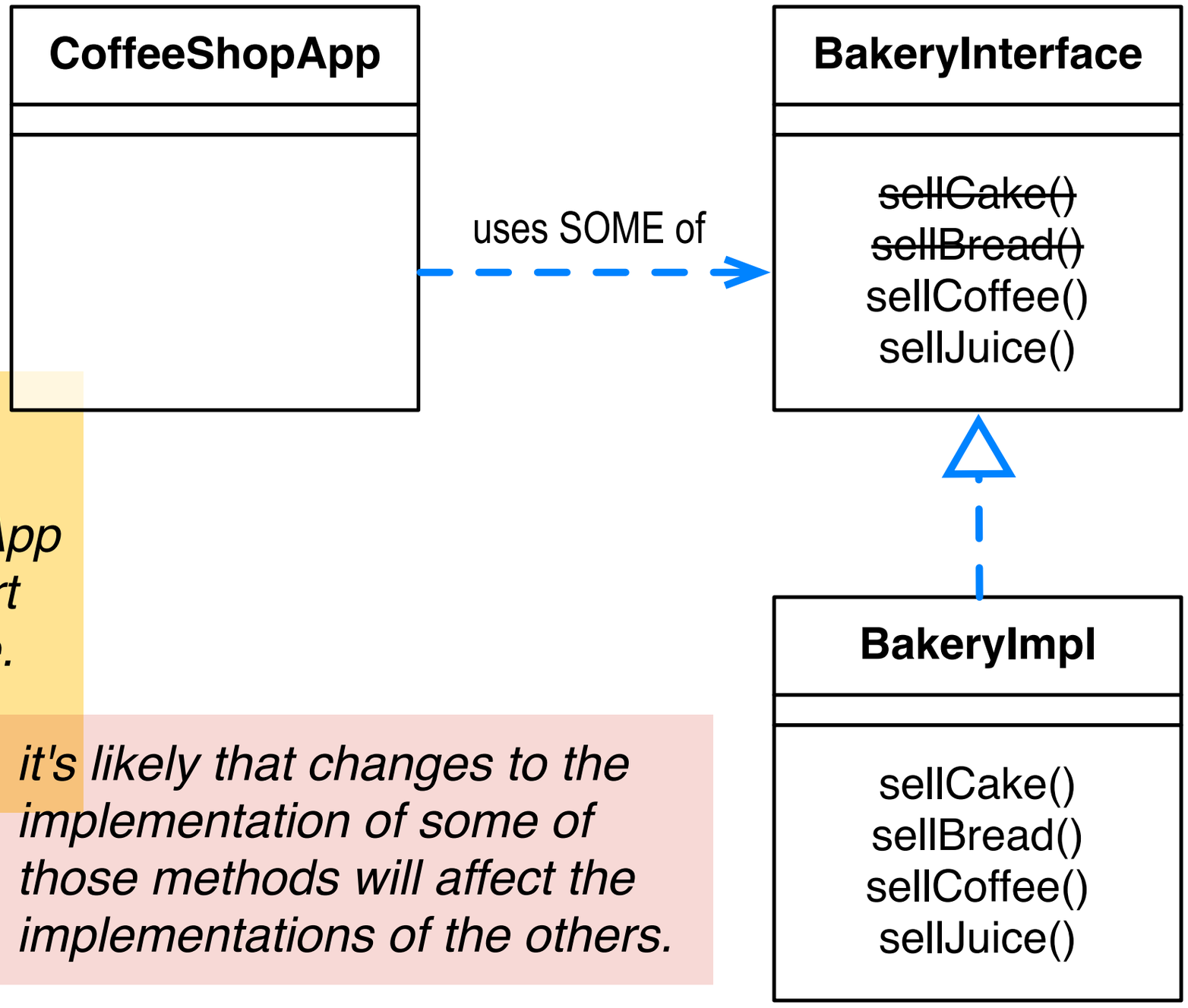
Example . . .



So far so good...

S O L I D

But what if we want a coffee shop?



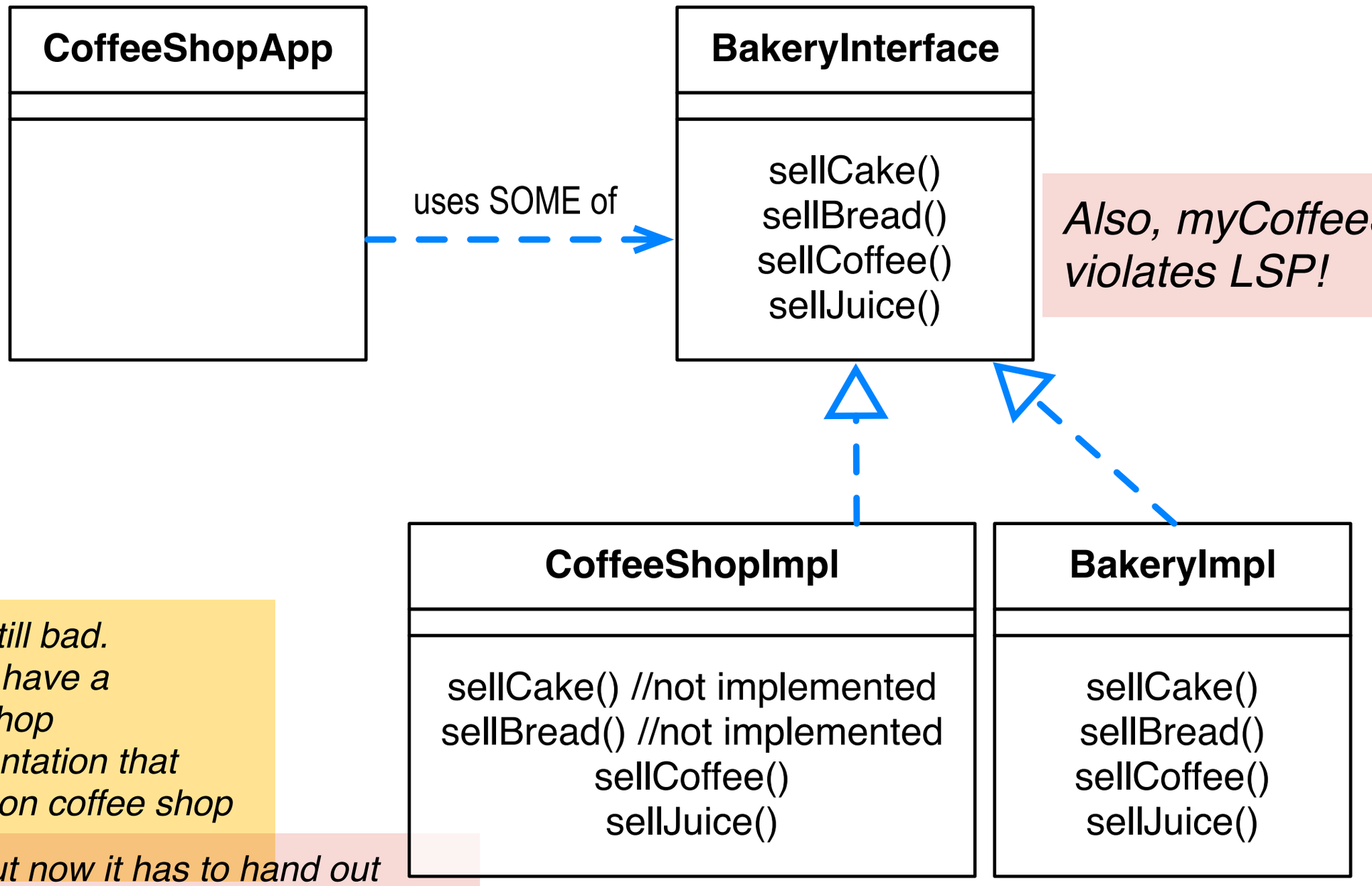
This is not as good!

Now the CoffeeShopApp client is only using part of the BakeryInterface. But it has to "know about" all of it.

it's likely that changes to the implementation of some of those methods will affect the implementations of the others.

S O L I D

How about a special implementation?



Also, myCoffeeShop app violates LSP!

This is still bad. Now we have a CoffeeShop implementation that focuses on coffee shop stuff,

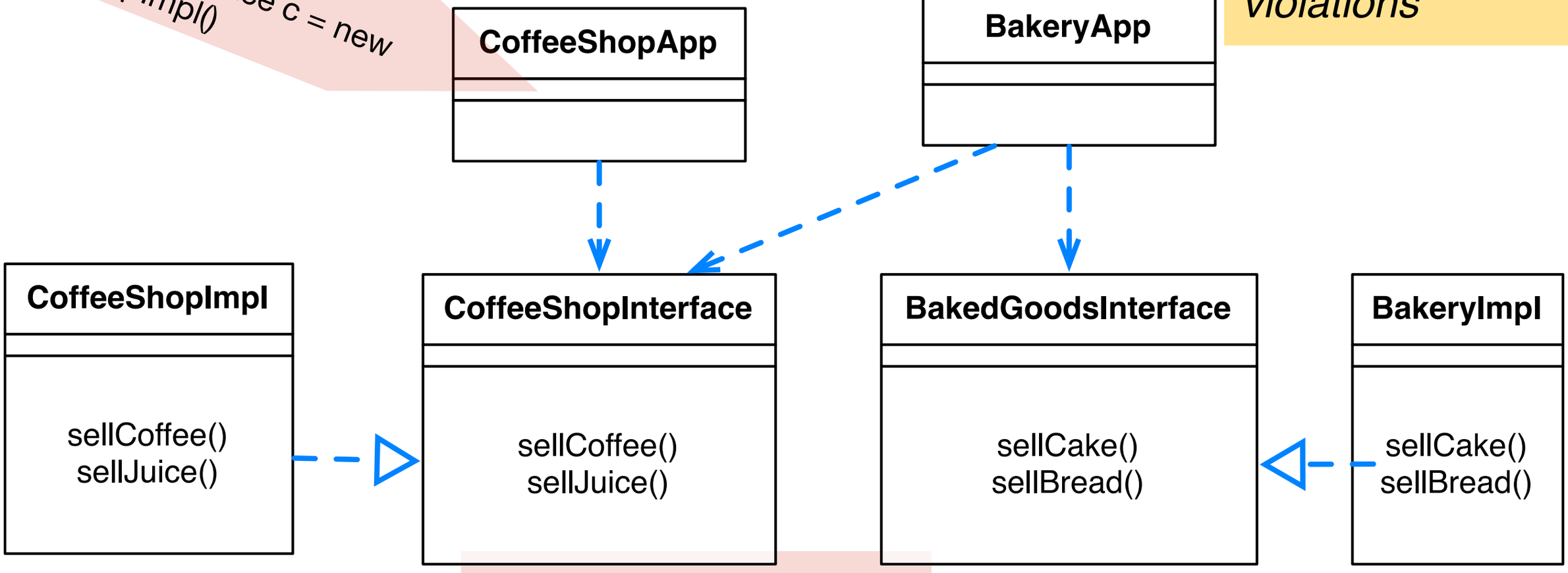
but now it has to hand out errors for the Bakery stuff if it's ever accidentally used!

S O L I D

A better arrangement

recall: this class would contain code that amounted to:
CoffeeShopInterface c = new CoffeeShopImpl()

We have no LSK violations



Now we have nice high cohesion in the interfaces (respecting Single Responsibility)

because of the arrangement, these are also open for extension, so we have also got the Open Closed principle checked off

S O L I D

Dependency Inversion

one should "Depend upon Abstractions. Do not depend upon concretions."

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.

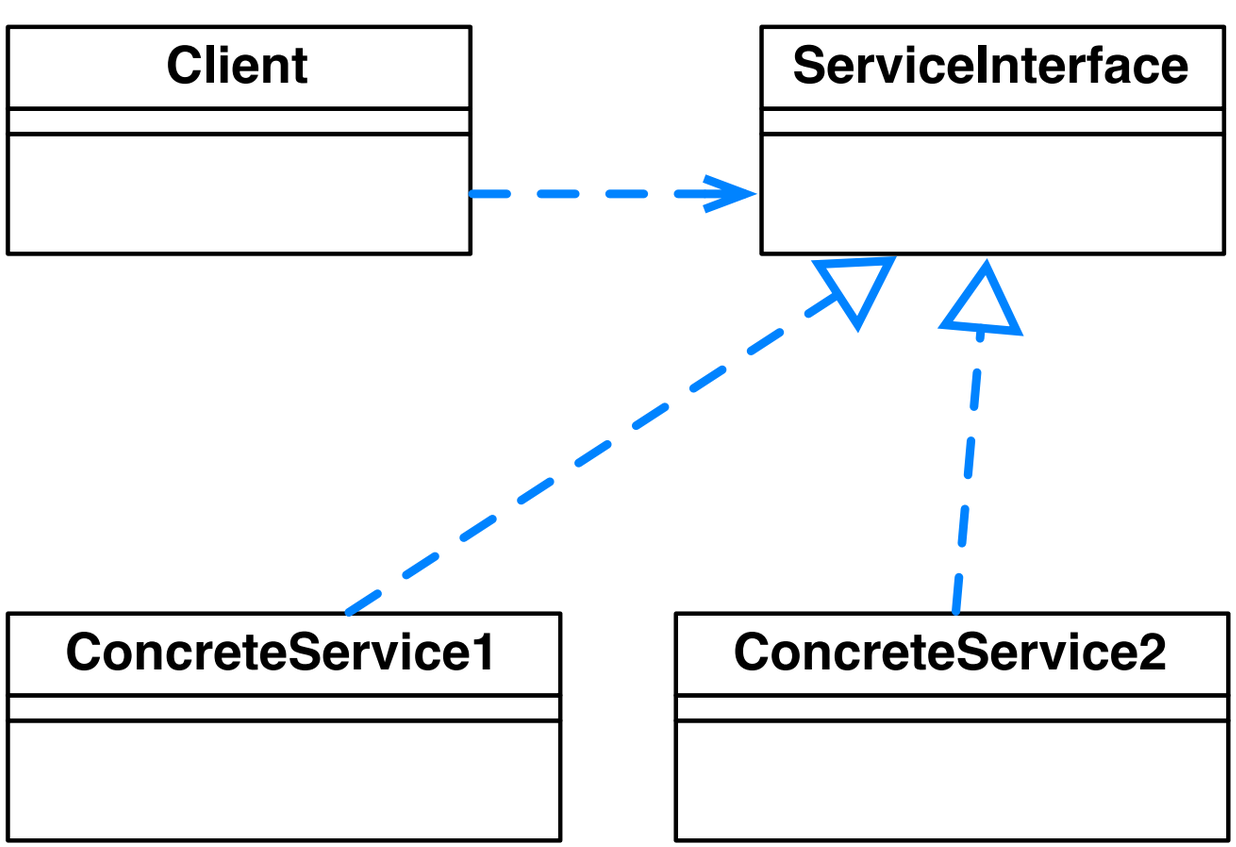
- We are going to think of this in about the same way as coarser grained information hiding, though there's much more to it than that.

http://en.wikipedia.org/wiki/Dependency_inversion_principle

S O L I D

Dependency Inversion - Class arrangement

means that the Client is "hooked up to" just one concrete service. Any changes to the concrete service will propagate to the client, meaning it will be harder to change or switch out the underlying service.



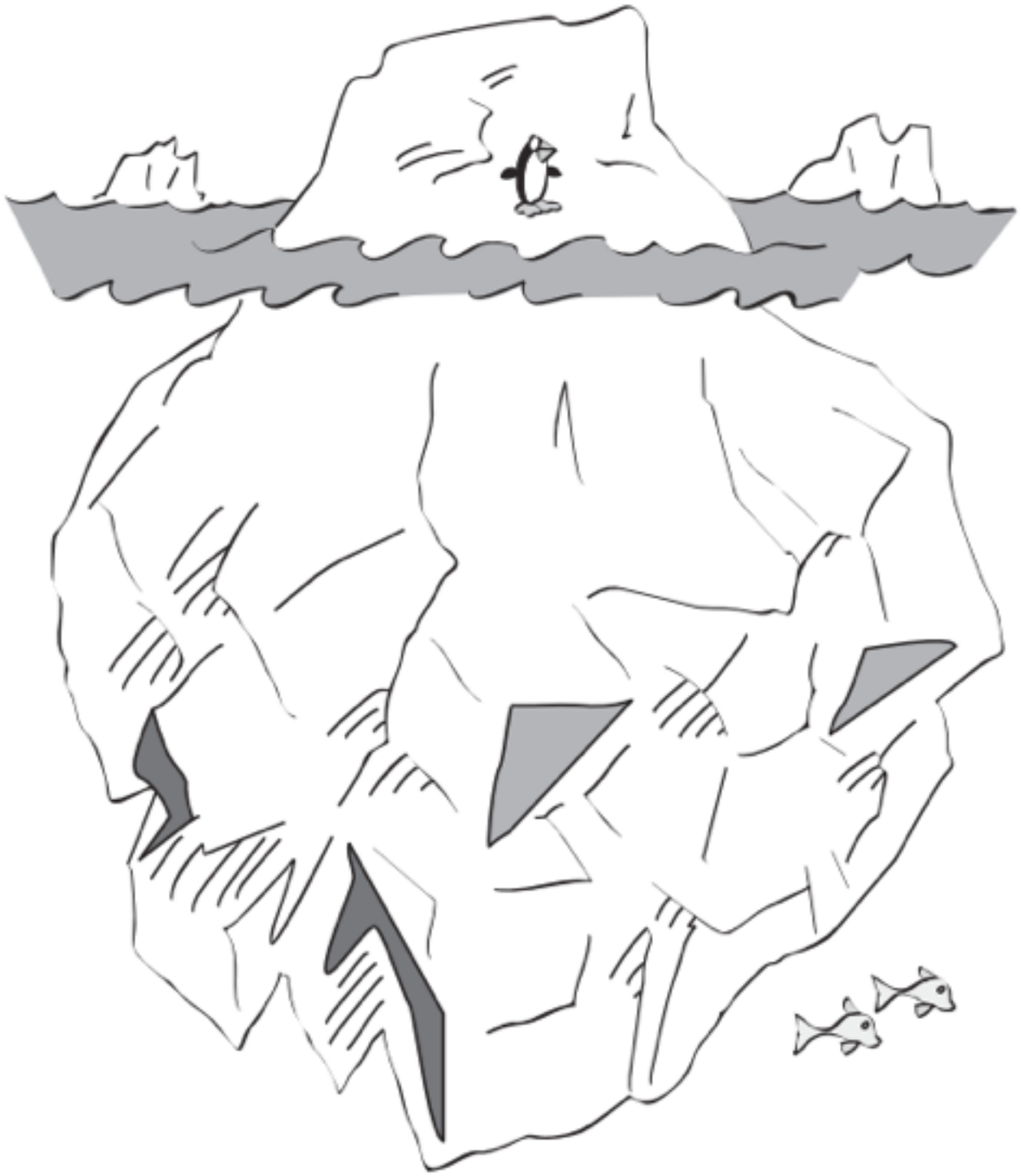
With this arrangement the client only knows about the interface, and is impervious to underlying changes (as long as the other principles are followed!!)

Declare abstraction;
Instantiate concretion.

```
ServiceInterface serv = new ConcreteService();
```

S O L I D

Information Hiding



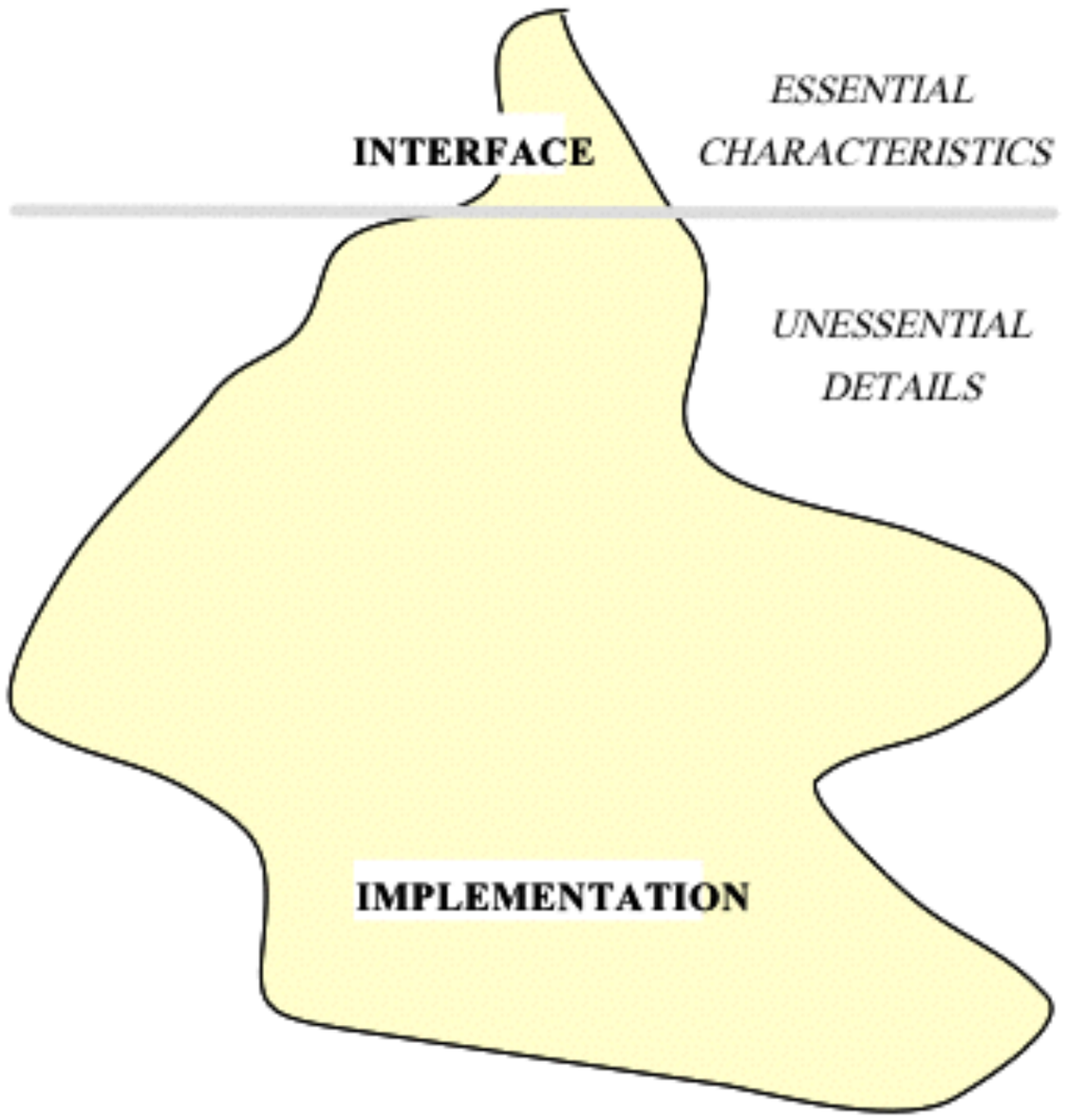
A good class is a lot like an iceberg: seven-eights is under water, and you can see only the one-eight that's above the surface.

from CodeComplete by Steve McConnell

S O L I D

Information Hiding

- Only expose **necessary** functions
- Abstraction hides complexity by emphasizing on essential characteristics and suppressing detail
- Caller should not assume anything about **how** the interface is implemented
- Effects of internal changes are localized



http://www.fatagnus.com/program-to-an-interface-not-an-implementation/

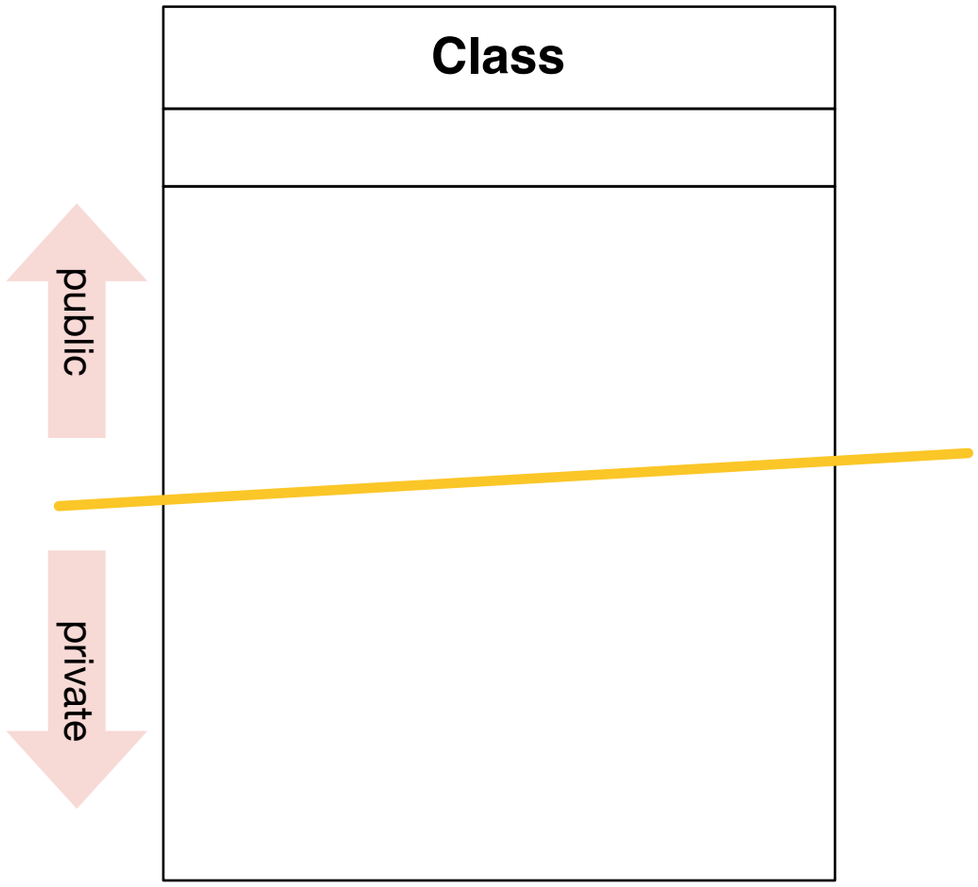
S O L I D

Information Hiding: Example

- Class DentistScheduler has
  - A public method ***automaticallySchedule()***
  - Private methods:
    - *whoToScheduleNext()*
    - *whoToGiveBadHour()*
    - *isHourBad()*
- To use DentistScheduler, just call ***automaticallySchedule()***
  - Don’t have to know how it’s done internally
  - Could use a different scheduling technique: no problem!

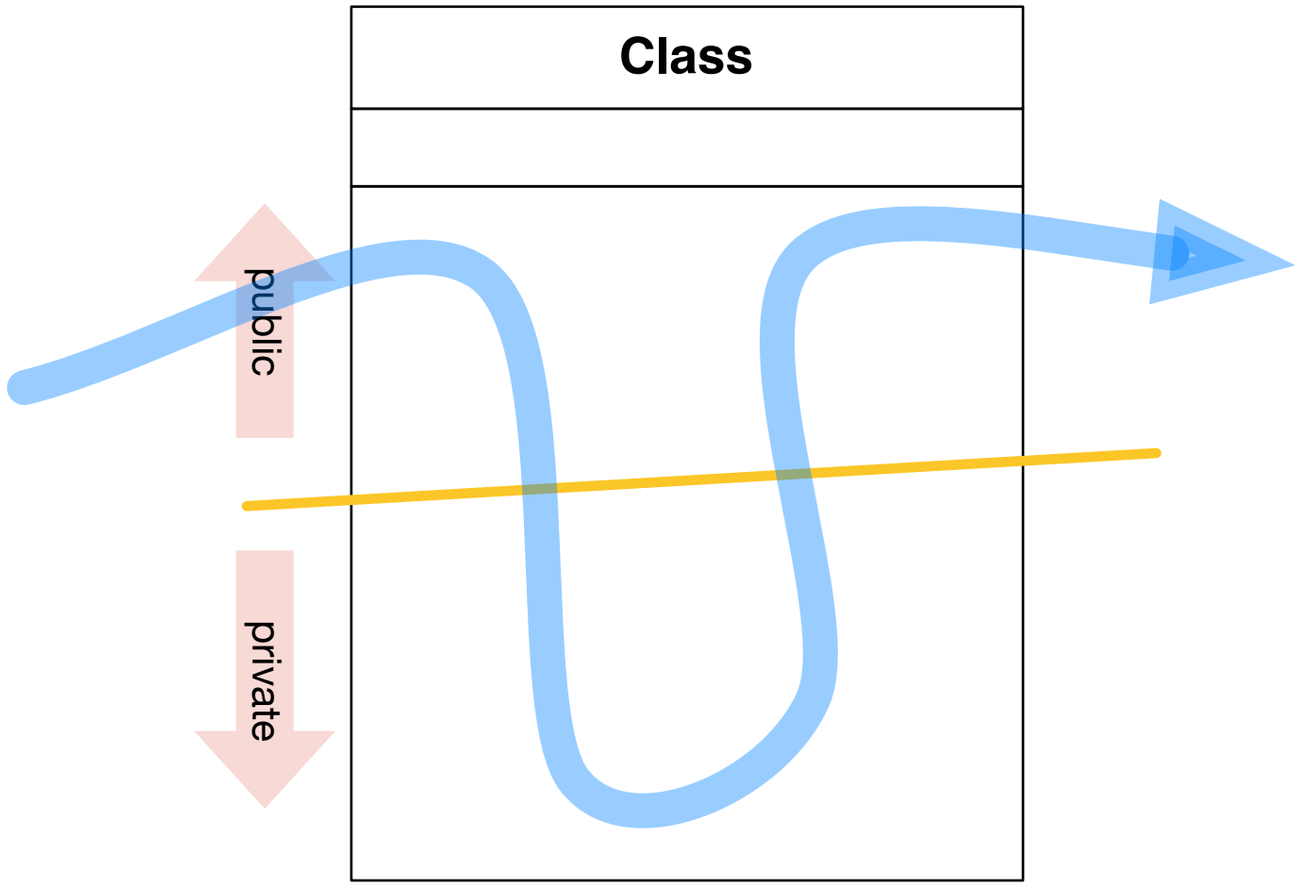
S O L I D

Typical class design



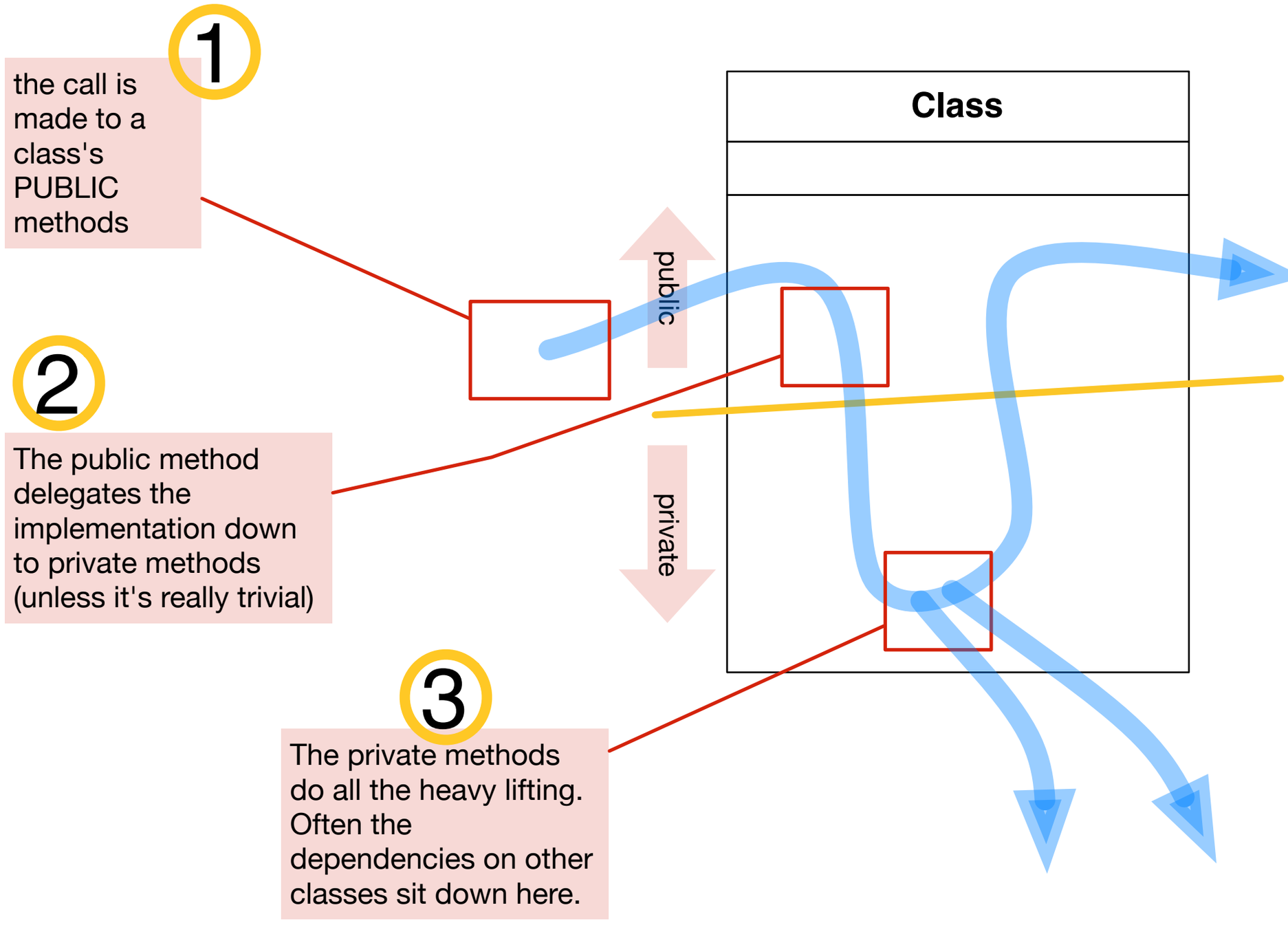
S O L I D

The public methods are the class's "interface"



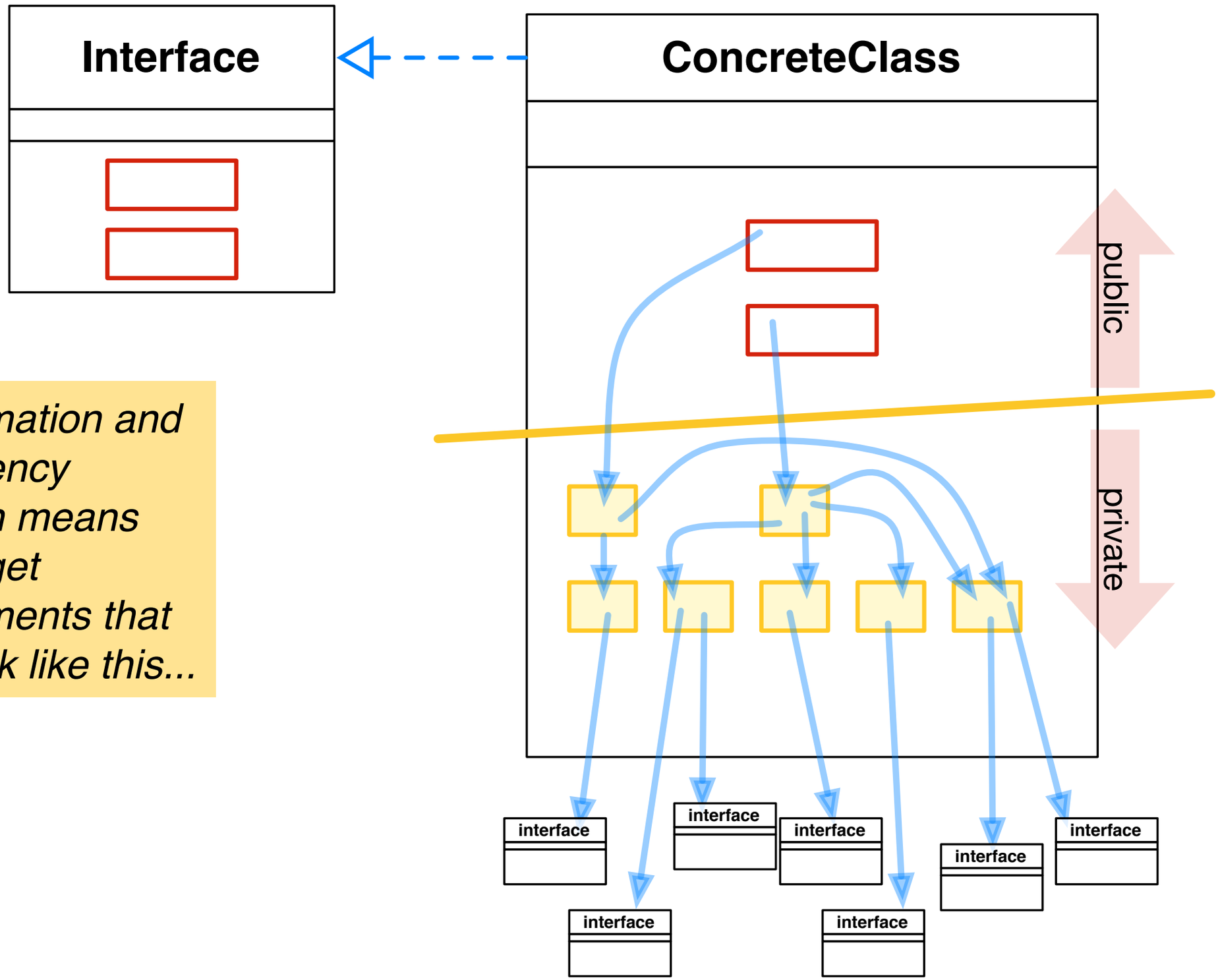
S O L I D

They make calls "down" to the implementation.



S O L I D

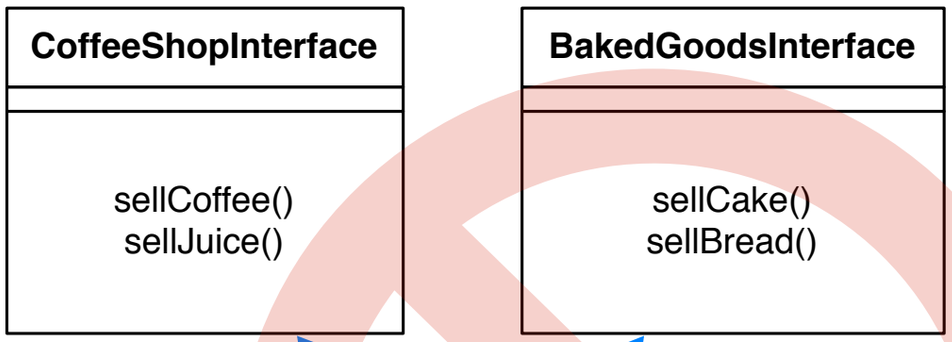
So putting these together...

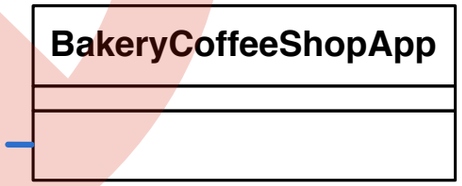


So information and Dependency Inversion means that we get arrangements that often look like this...

S O L I D

Using the bakery example . . .

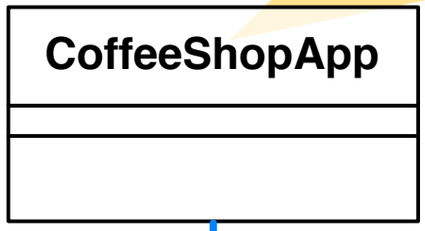


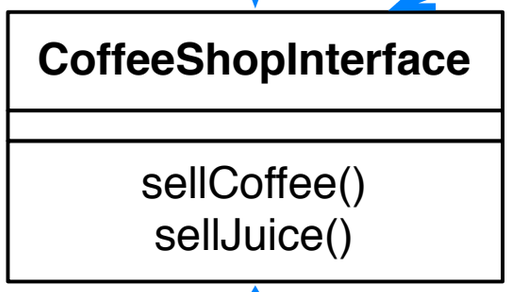


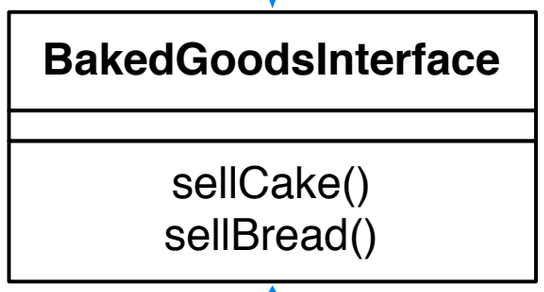
The BakeryCoffeeShop is depending on an IMPLEMENTATION, rather than a collection of nice, specific interfaces

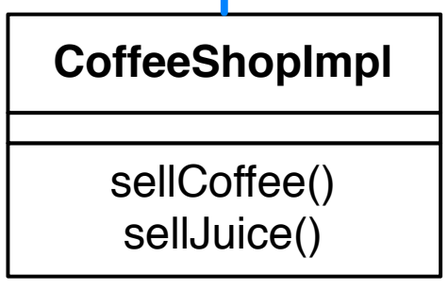
This is how you want the arrangement to look.

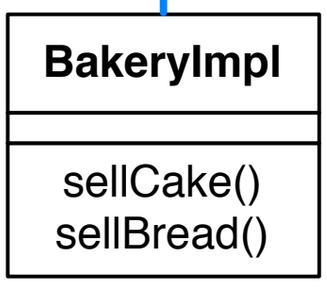
This gives the app more flexibility. It can depend upon additional interfaces, without having to wait for the perfectly combined underlying implementation.











S O L I D

Pragmatic Programmer:
Eliminate Effects Between Unrelated Things –
design components that are:
self-contained,
independent,
and have a single, well-defined purpose

Summary

- Classes should do ONE THING (sometimes (often) that aligns better with a role than with a noun) (pass the and/or-check)
- Don’t force “extenders” to change the implementation directly — allow them to truly extend.
- Don’t violate type substitutability! Pass the is-a test.
- Have small, role-specific interfaces (interfaces should also just do one thing)
- Depend upon abstractions, not implementations.