3.3 Class Design Principles

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
  - a.k.a. Design by Contract
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (ISP)

3.3 Single Responsibility Principle (SRP)

A class should have only one reason to change

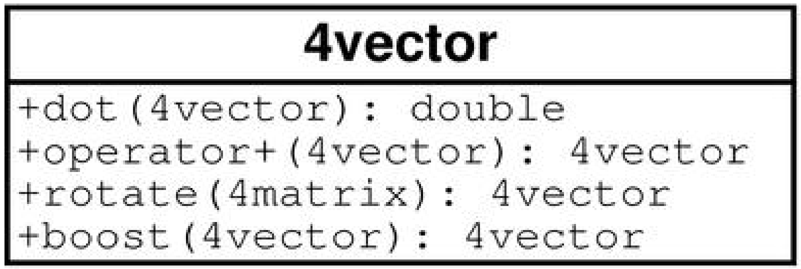
Robert Martin

Related to and derived from *cohesion*, i.e. that elements in a module should be closely related in their function

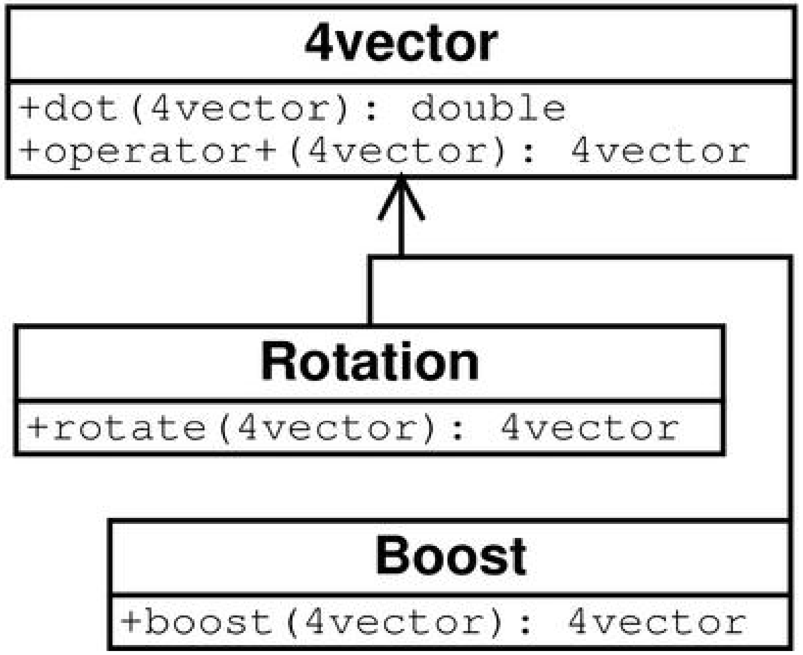
Responsibility of a class to perform a certain function is also a reason for the class to change

3.3 SRP Example

All-in-one wonder



Separated responsibilities



Always changes to 4vector

Changes to rotations or boosts don't impact on 4vector

3.3 SRP Summary

- Class should have only one reason to change
  - Cohesion of its functions/responsibilities
- Several responsibilities
  - mean several reasons for changes → more frequent changes
- Sounds simple enough
  - Not so easy in real life
  - Tradeoffs with complexity, repetition, opacity

3.3 Open/Closed Principle (OCP)

Modules should be open for extension,

but closed for modification

Bertrand Meyer

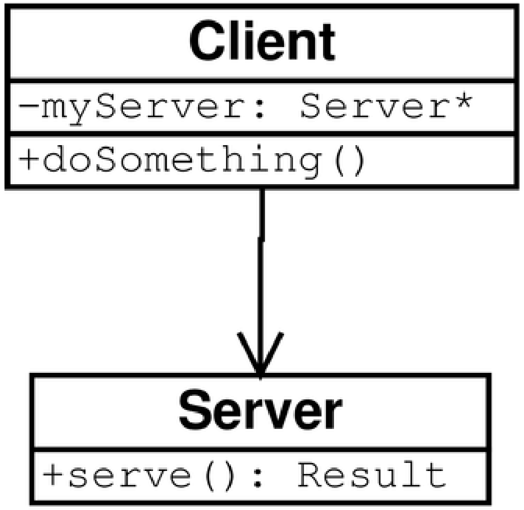
Object Oriented Software Construction

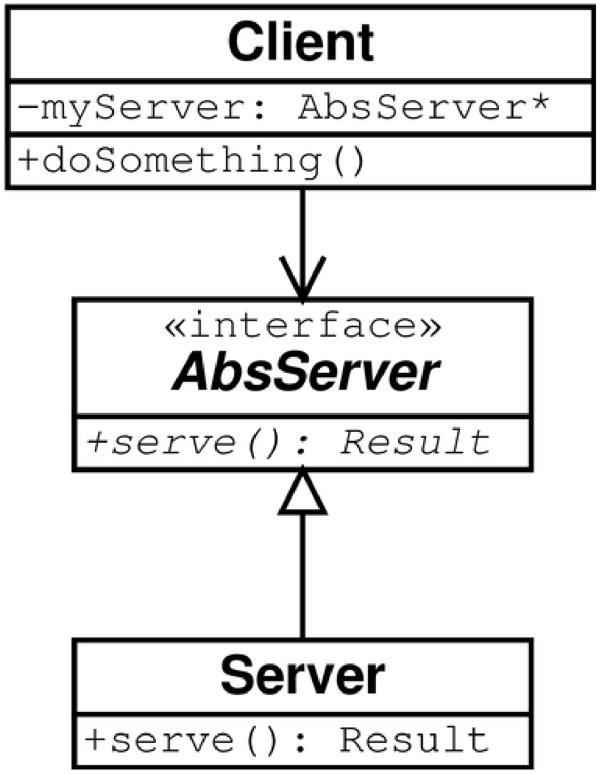
Module: Class, Package, Function

New functionality → new code, existing code remains unchanged

"Abstraction is the key" → cast algorithms in abstract interfaces
develop concrete implementations
as needed

3.3 Abstraction and OCP





Client is closed to changes of Server

Client is open for extension through new Server implementations

Without AbsServer the Client is open to changes in Server

3.3 The Shape Example - Procedural

Shape.h
```
enum ShapeType { isCircle, isSquare };
typedef struct Shape {
  enum ShapeType type
} shape;
```

Circle.h
```
typedef struct Circle {
  enum ShapeType type;
  double radius;
  Point center;
} circle;
void drawCircle( circle* );
```

Square.h
```
typedef struct Square {
  enum ShapeType type;
  double side;
  Point topleft;
} square;
void drawSquare( square* );
```

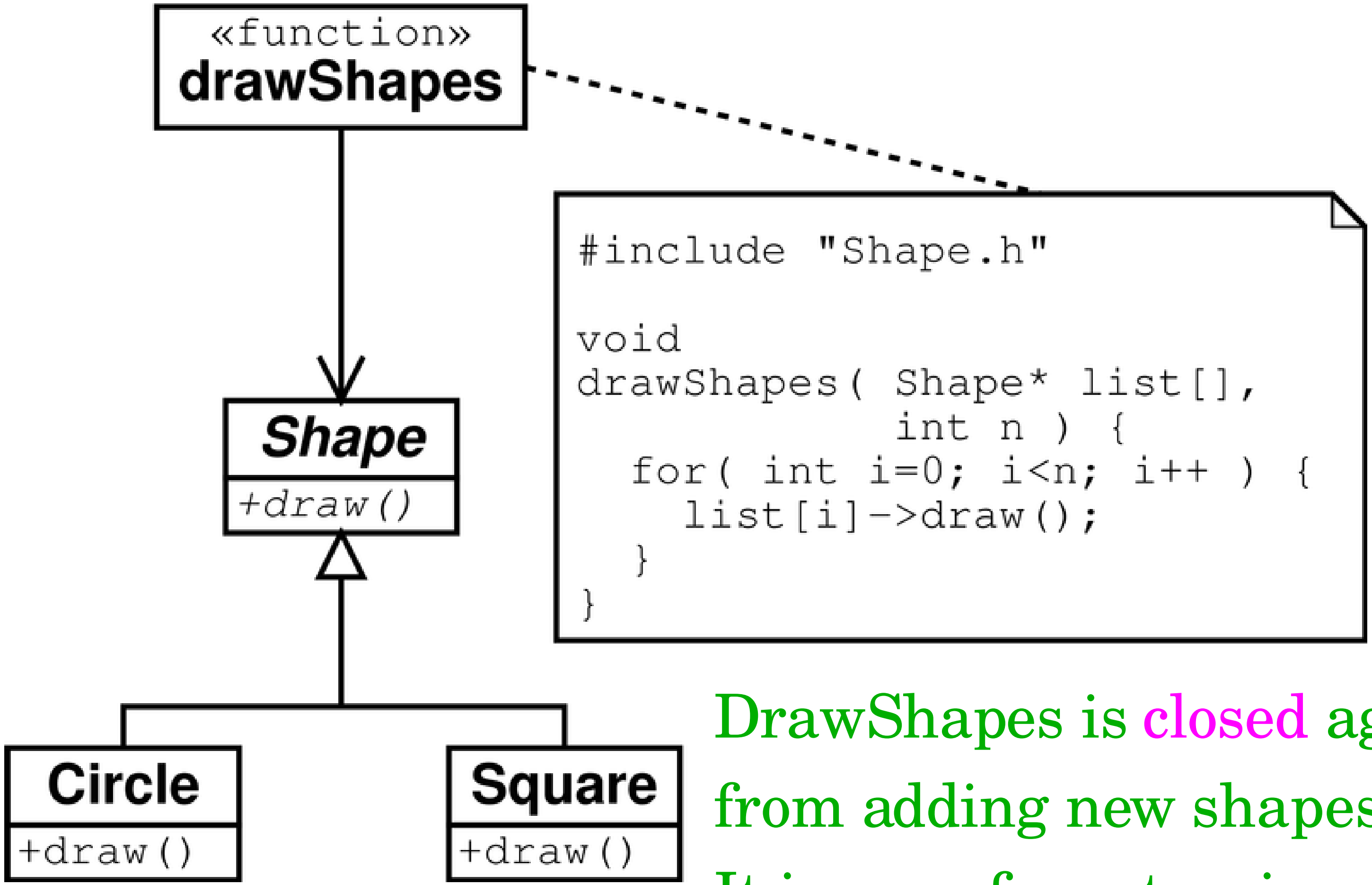
drawShapes.c
```
#include "Shape.h"
#include "Circle.h"
#include "Square.h"

void drawShapes( shape* list[], int n ) {
  int i;
  for( int i=0; i<n; i++ ) {
    shape* s= list[i];
    switch( s->type ) {
    case isSquare:
      drawSquare( (square*)s );
      break;
    case isCircle:
      drawCircle( (circle*)s );
      break;
    }
  }
}
```

RTTI a la C: Adding a new shape requires many changes

3.3 Problems with Procedural Implementation

- drawShapes is not closed
  - switch/case probably needed in several places
  - Adding a shape → modify switch/case
  - There may be many and the logic may be more complicated
  - Extending enum ShapeType → rebuild everything
- Rigid, fragile, highly viscous

3.3 The Shape Example OO



```
#include "Shape.h"

void
drawShapes( Shape* list[],
            int n ) {
  for( int i=0; i<n; i++ ) {
    list[i]->draw();
  }
}
```

DrawShapes is closed against changes from adding new shapes

It is open for extension, e.g. adding new functions to manipulate shapes

Just add new shapes or functions and relink

3.3 OCP Summary

- Open for extension
  - Add new code for new functionality, don't modify existing working code
  - Implementations of interfaces somewhere
- Closed for modification
  - Need to anticipate likely modifications to be able to plan ahead in the design
  - e.g. ordering shapes? No closure against this requirement ... but could be added in a design-preserving way (low viscosity)

3.3 OCP How-To

- How is the system going to evolve?
- How will its environment change?
- Isolate against *kinds of changes*, e.g.
  - database schema (data model)
  - hardware changes (sensors, ADCs, TDCs, etc)
  - data store technology (e.g. Objectivity vs ROOT)
- Plan ahead, but don't implement what is not already needed

3.3 Liskov Substitution Principle (LSP)

All derived classes must be substituteable for their base class

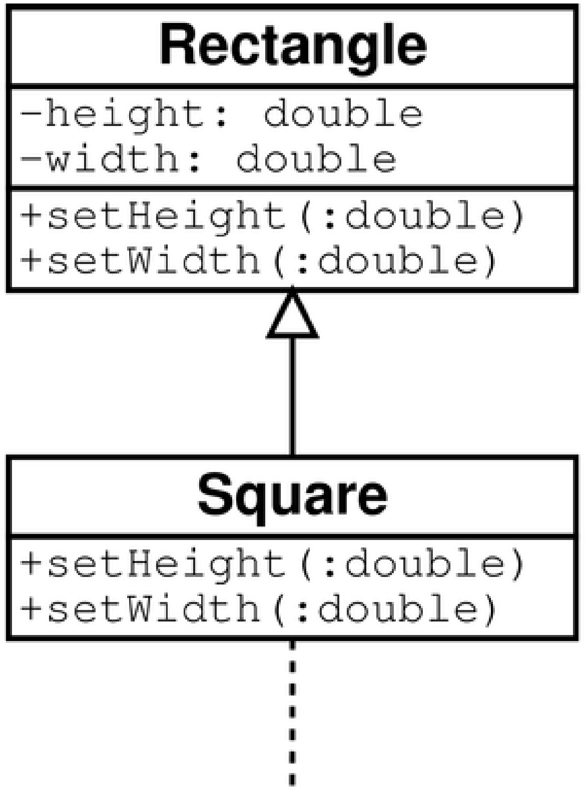
Barbara Liskov, 1988

The "Design-by-Contract" formulation:

All derived classes must honor the contracts of their base classes

Bertrand Meyer

3.3 The Square-Rectangle Problem



```
void Square::setHeight( double h ) {
  Rectangle::setHeight( h );
  Rectangle::setWidth( h );
}
void Square::setWidth( double w ) {
  setHeight( w );
}
```

Clients (users) of Rectangle expect that setting height leaves width unchanged (and vice versa)

Square does not fulfill this expectation

Client algorithms can get confused

Hack: attempt to identify subclasses and use if/switch (RTTI)

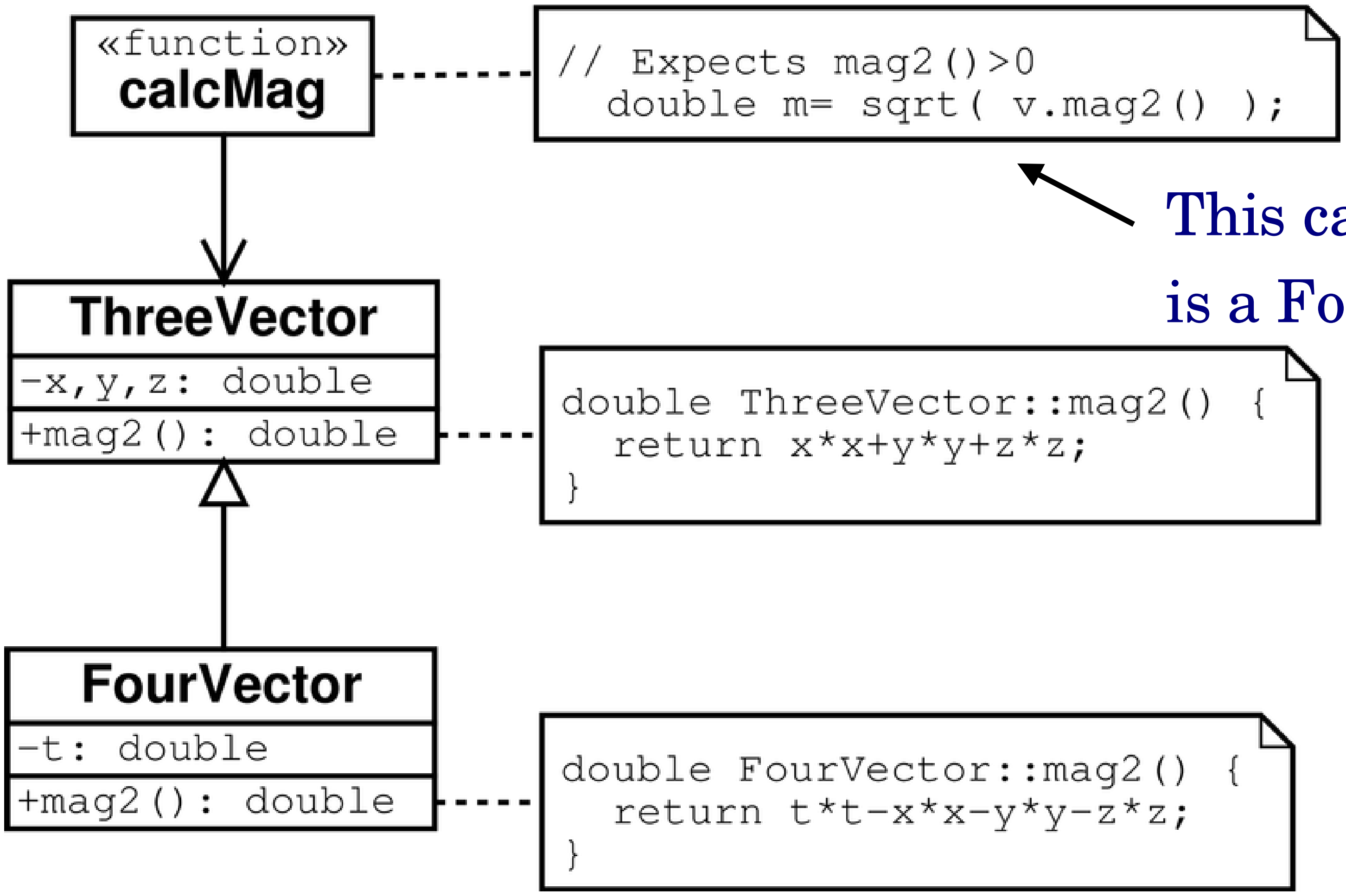
↑

This is evil!

3.3 Contract Violation

- The contract of Rectangle
  - height and width independent; set one while the other is unchanged, area = height*width
- Square breaks this contract
- Derived methods should not expect more and provide no less than the base class methods
  - Preconditions are not stronger
  - Postconditions are not weaker

3.3 The FourVector Example



```
// Expects mag2()>0
  double m= sqrt( v.mag2() );
```

```
double ThreeVector::mag2() {
  return x*x+y*y+z*z;
}
```

```
double FourVector::mag2() {
  return t*t-x*x-y*y-z*z;
}
```

This can crash when v is a FourVector

A 4-vector IS-A 3-vector with a time-component? Not in OO, 4-vector has different algebra → can't fulfill 3-vector contracts

3.3 LSP Summary

- Subclass must fully substitute base class
  - Guides design and choice of abstractions
- Good abstractions are not always intuitive
- Violating LSP may break OCP
  - Need RTTI and if/switch → lost closure
- Inheritance/polymorphism powerful tools
  - Use with care
- IS-A relation really means behaviour

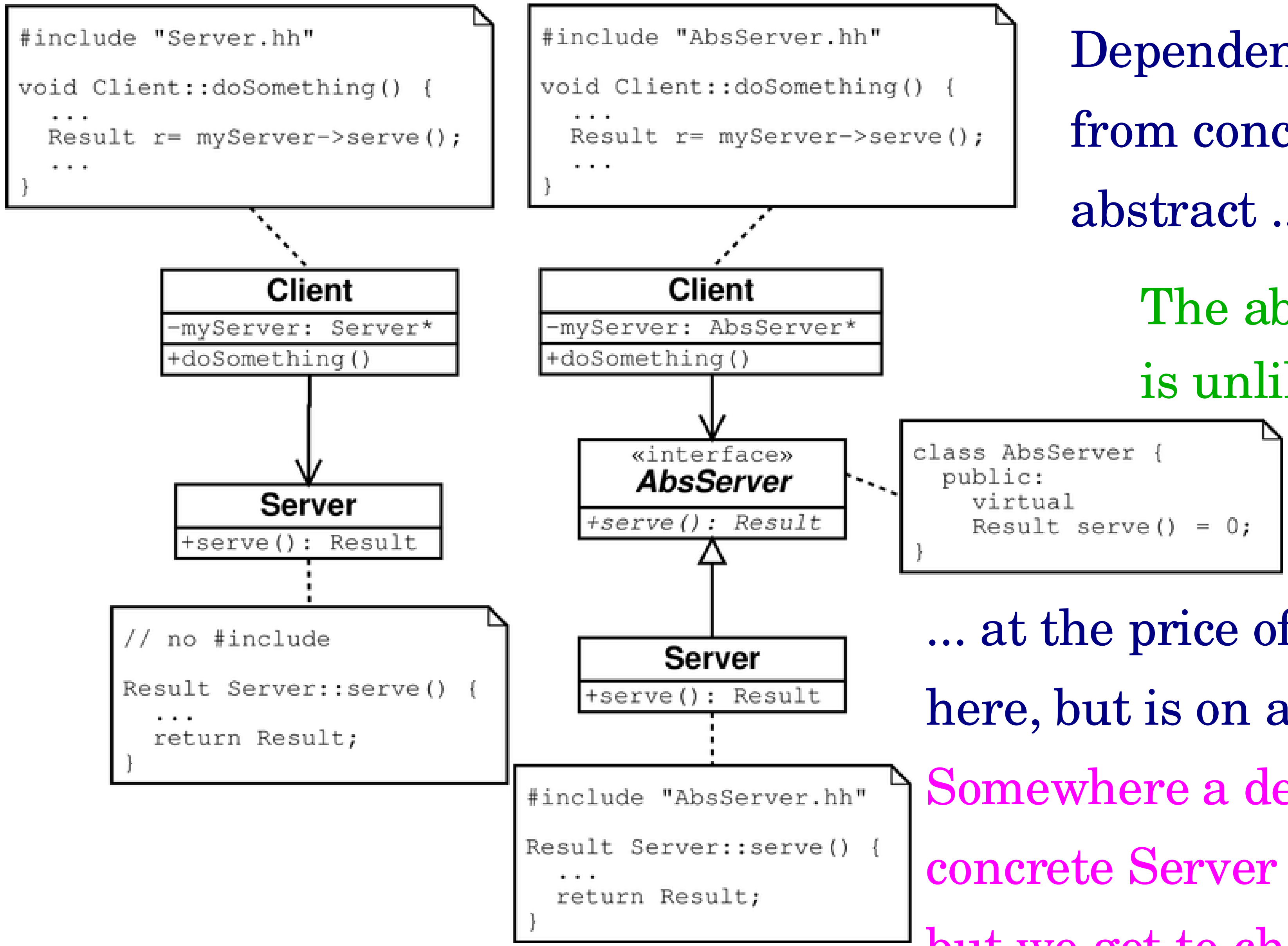
3.3 Dependency Inversion Principle (DIP)

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

Robert Martin

Why *dependency inversion*? In OO we have ways to invert the direction of dependencies, i.e. class inheritance and object polymorphism

3.3 DIP Example



```
#include "Server.hh"

void Client::doSomething() {
  ...
  Result r= myServer->serve();
  ...
}
```

```
// no #include

Result Server::serve() {
  ...
  return Result;
}
```

```
#include "AbsServer.hh"

void Client::doSomething() {
  ...
  Result r= myServer->serve();
  ...
}
```

```
class AbsServer {
  public:
    virtual
    Result serve() = 0;
}
```

```
#include "AbsServer.hh"

Result Server::serve() {
  ...
  return Result;
}
```

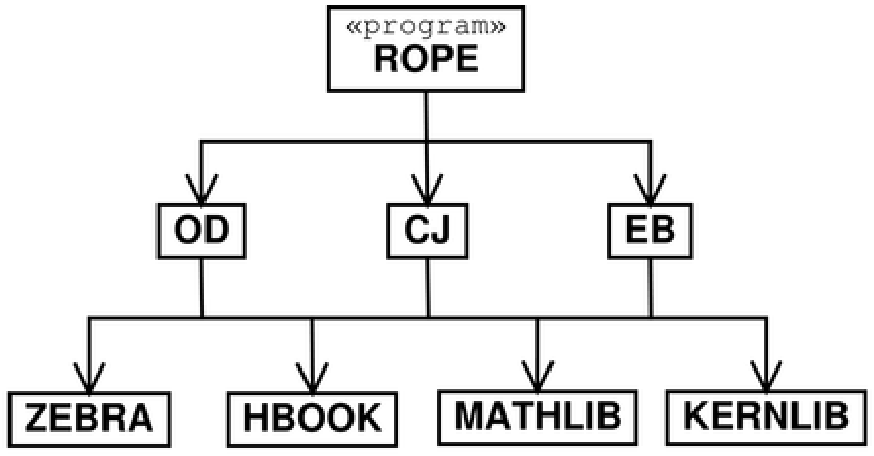
Dependency changed from concrete to abstract ...

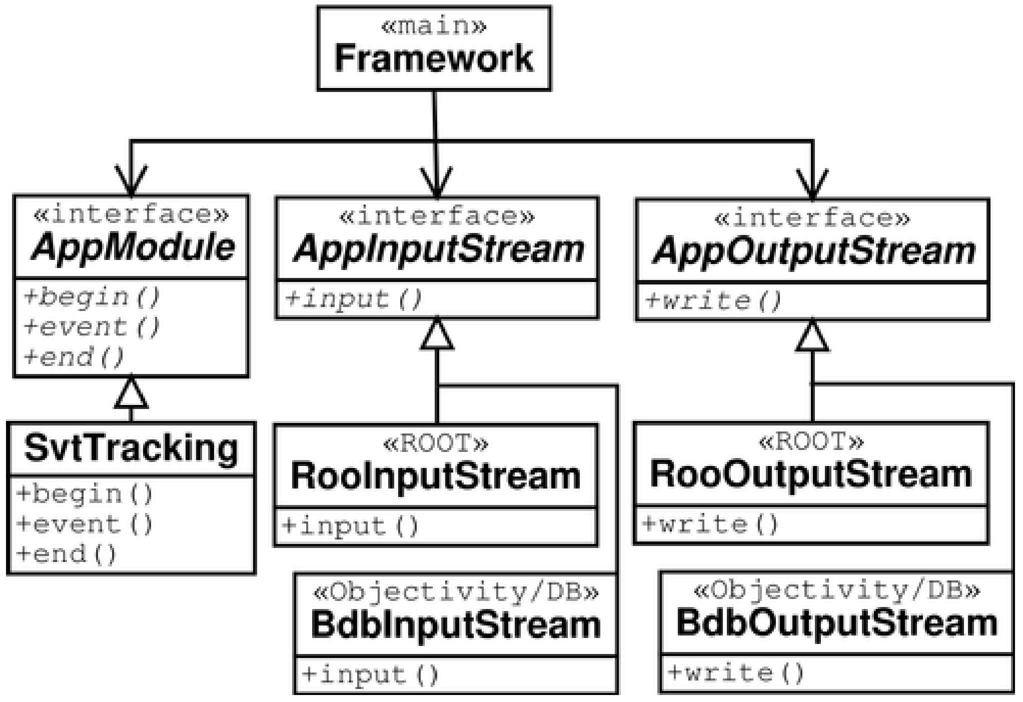
The abstract class is unlikey to change

... at the price of dependency here, but is on an abstraction.

Somewhere a dependency on concrete Server must exist, but we get to choose where.

3.3 DIP and Procedural Design





Procedural:

Call more concrete routines

Dependence on (reuseable) concrete modules

In reality the dependencies are cyclic → need multipass link and a “dummy library”

The BaBar Framework classes depend on interfaces

Can e.g. change data store technology without disturbing the Framework classes

3.3 DIP Summary

- Use DI to avoid
  - deriving from concrete classes
  - associating to or aggregating concrete classes
  - dependency on concrete components
- Encapsulate invariants: generic algorithms
  - Abstract interfaces don't change
  - Concrete classes implement interfaces
  - Concrete classes easy to replace
- Foundation classes (STL, CLHEP, MFC)?

3.3 Interface Segregation Principle (ISP)

Many client specific interfaces are better than one general purpose interface.
Clients should not be forced to depend upon interfaces they don't use.

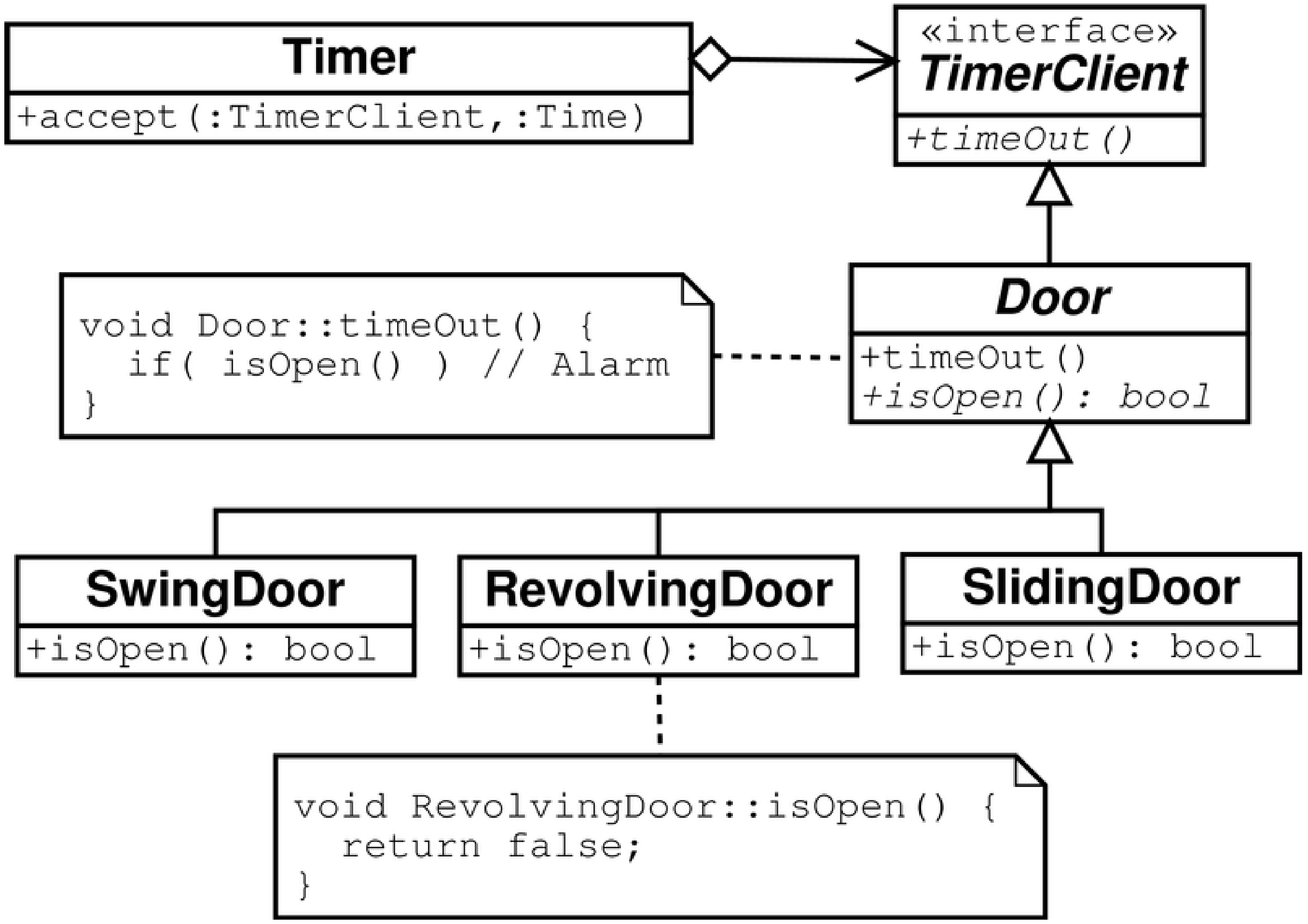
1) High level modules should not depend on low level modules.
Both should depend upon abstractions (interfaces)
2) Abstractions should not depend upon details.
Details should depend on abstractions.

Robert Martin

3.3 ISP Explained

- Multipurpose classes
  - Methods fall in different groups
  - Not all users use all methods
- Can lead to unwanted dependencies
  - Clients using one aspect of a class also depend indirectly on the dependencies of the other aspects
- ISP helps to solve the problem
  - Use several client-specific interfaces

3.3 ISP Example: Timed Door

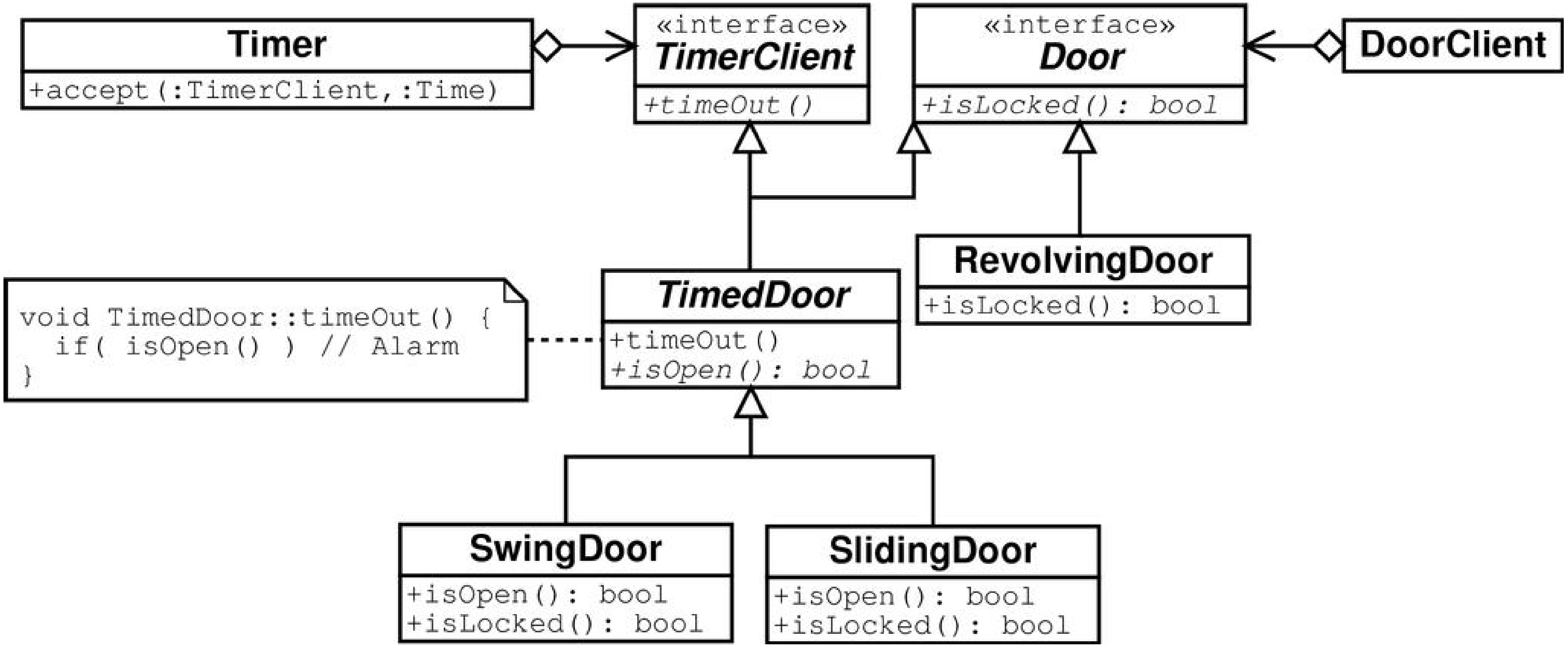


```
void Door::timeOut() {
  if( isOpen() ) // Alarm
}
```

```
void RevolvingDoor::isOpen() {
  return false;
}
```

There may be derived classes of Door which don't need the TimerClient interface. They suffer from depending on it anyway.

3.3 Timed Door ISP

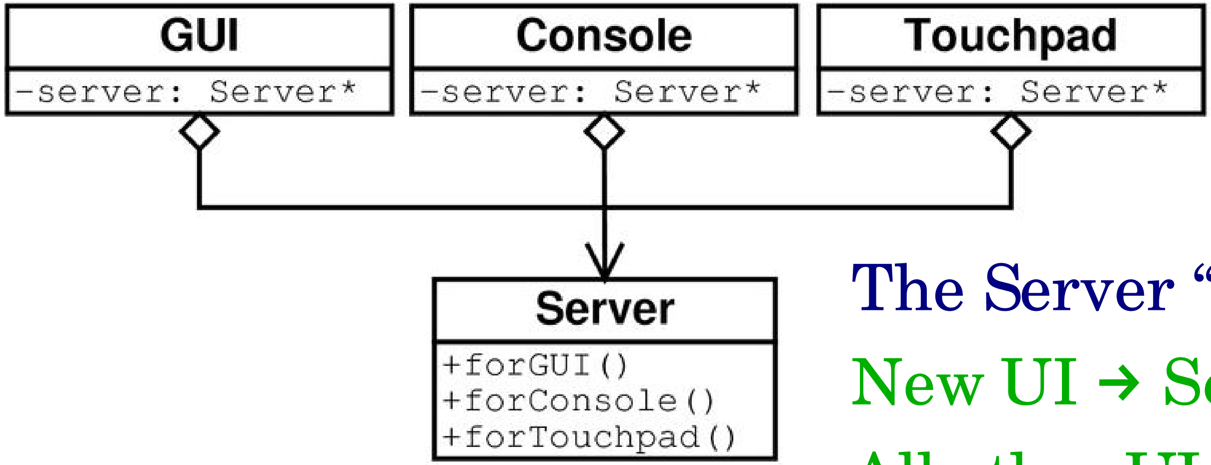


```
void TimedDoor::timeOut() {
  if( isOpen() ) // Alarm
}
```

RevolvingDoor does not depend needlessly on TimerClient

SwingDoor and SlidingDoor really are timed doors

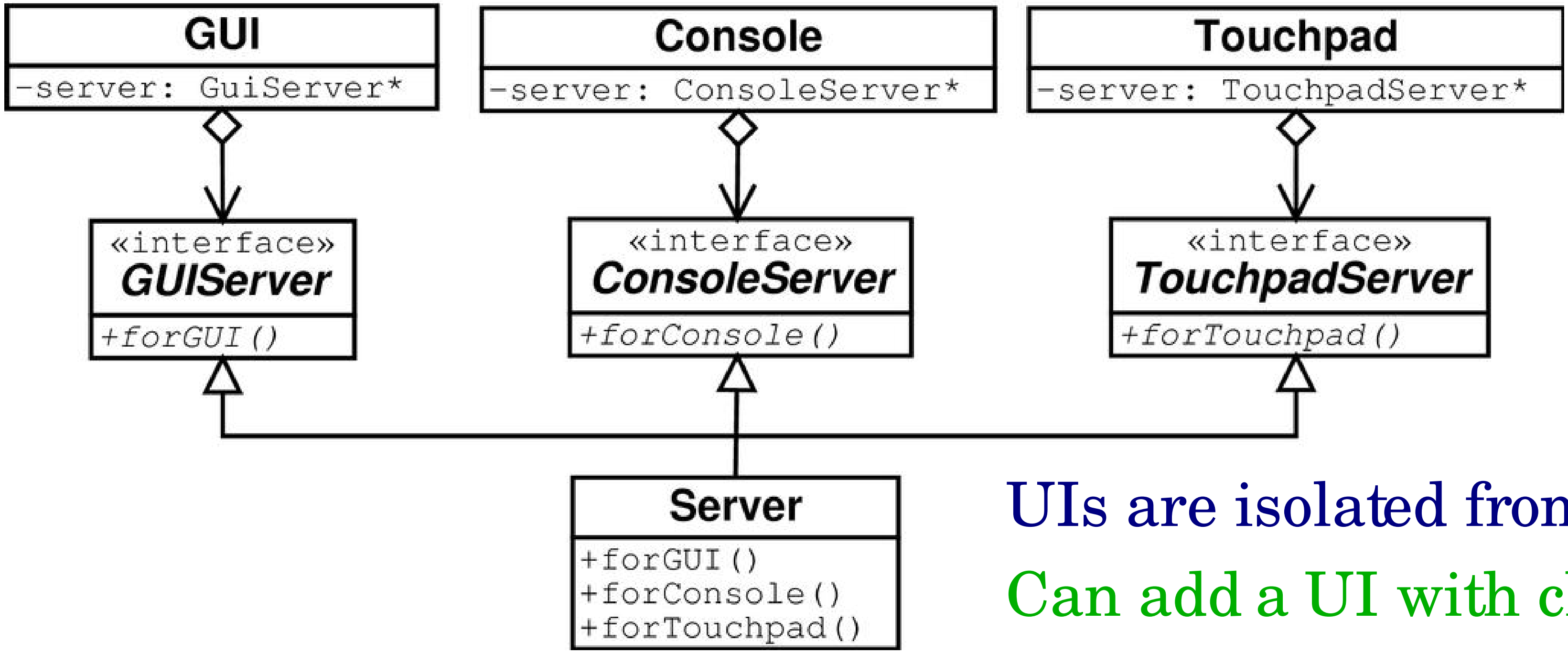
3.3 ISP Example: UIs



The Server "collects" interfaces

New UI → Server interface changes

All other UIs recompile



UIs are isolated from each other

Can add a UI with changes in Server → other UIs not affected

3.3 ISP Summary

- Class (Server) collects interfaces for various purposes (Clients) → fat interface
  - Use separate interfaces to hide parts of the Server interface for Clients
  - Similar to data hiding
  - Or split the Server in several parts
- Be careful with vertical multiple inheritance
  - You might drag in dependencies you don't want/need/like

3.3 Class Design Principles:

- Single Responsibility Principle (SRP)
  - Only one reason to change
- Open-Closed Principle (OCP)
  - Extend functionality with new code
- Liskov Substitution Principle (LSP)
  - Derived classes fully substitute their bases
- Dependency Inversion Principle (DIP)
  - Depend on abstractions, not details
- Interface Segregation Principle (ISP)
  - Split interfaces to control dependencies