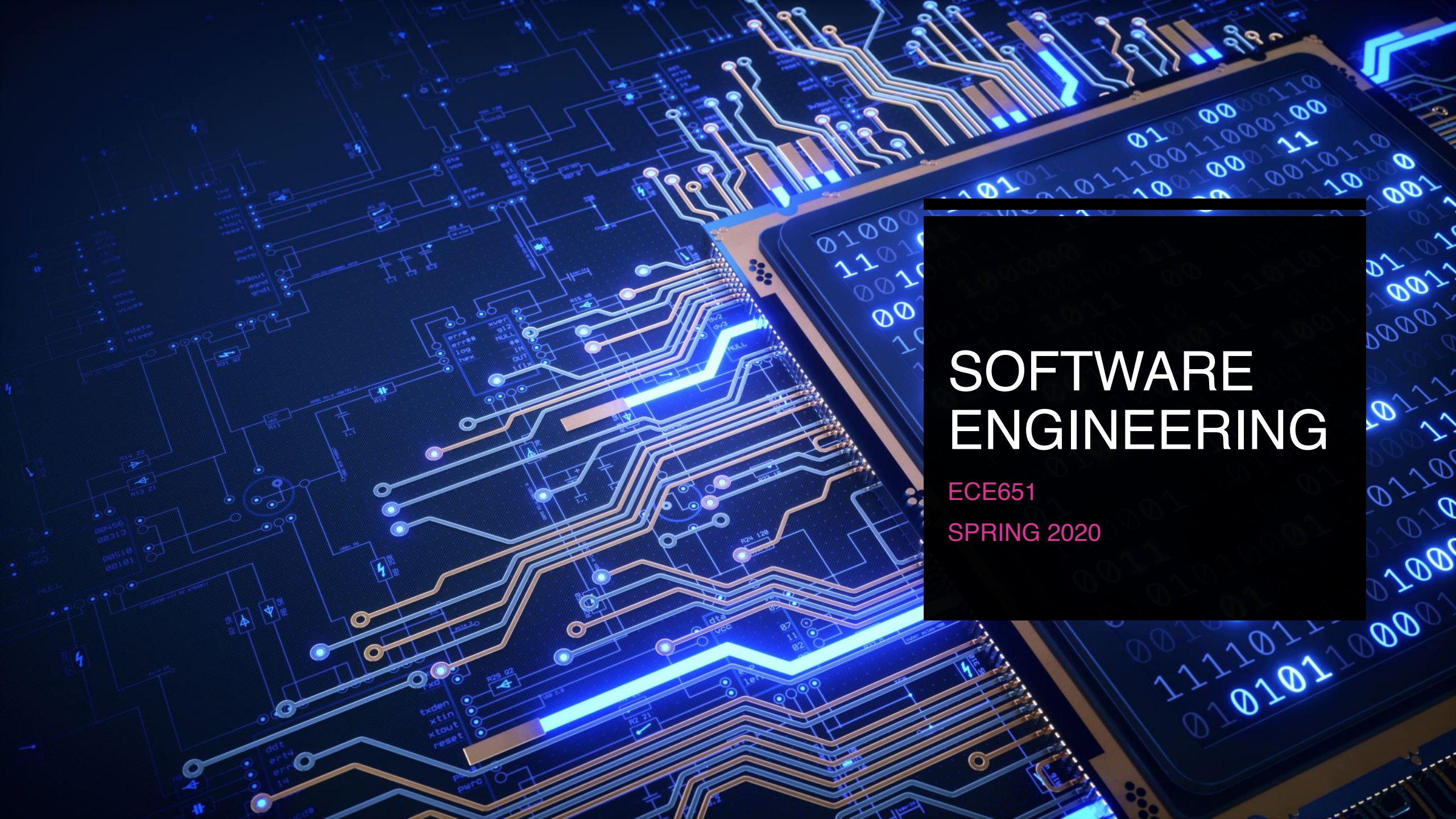


SOFTWARE ENGINEERING

ECE651

SPRING 2020

GOALS & DESIGN PRINCIPLES

THINK, PAIR, SHARE

- What characteristics of software would customers desire?
- Of these, which work well together, which are in opposition?

GOALS

- **Correct**: does what it is supposed to (no bugs)
- **Robust**: secure, dependable, resilient
- **Useable**: UI, compatibility,…
- **Many features**: does lots of "good" stuff
- **Good performance**: fast, low energy usage,…
  - **Scalable**
- **Maintainable/extensible**: Can add and change features
- **Quick delivery**: low development time
- **Low cost**: less spending is preferred

GOALS

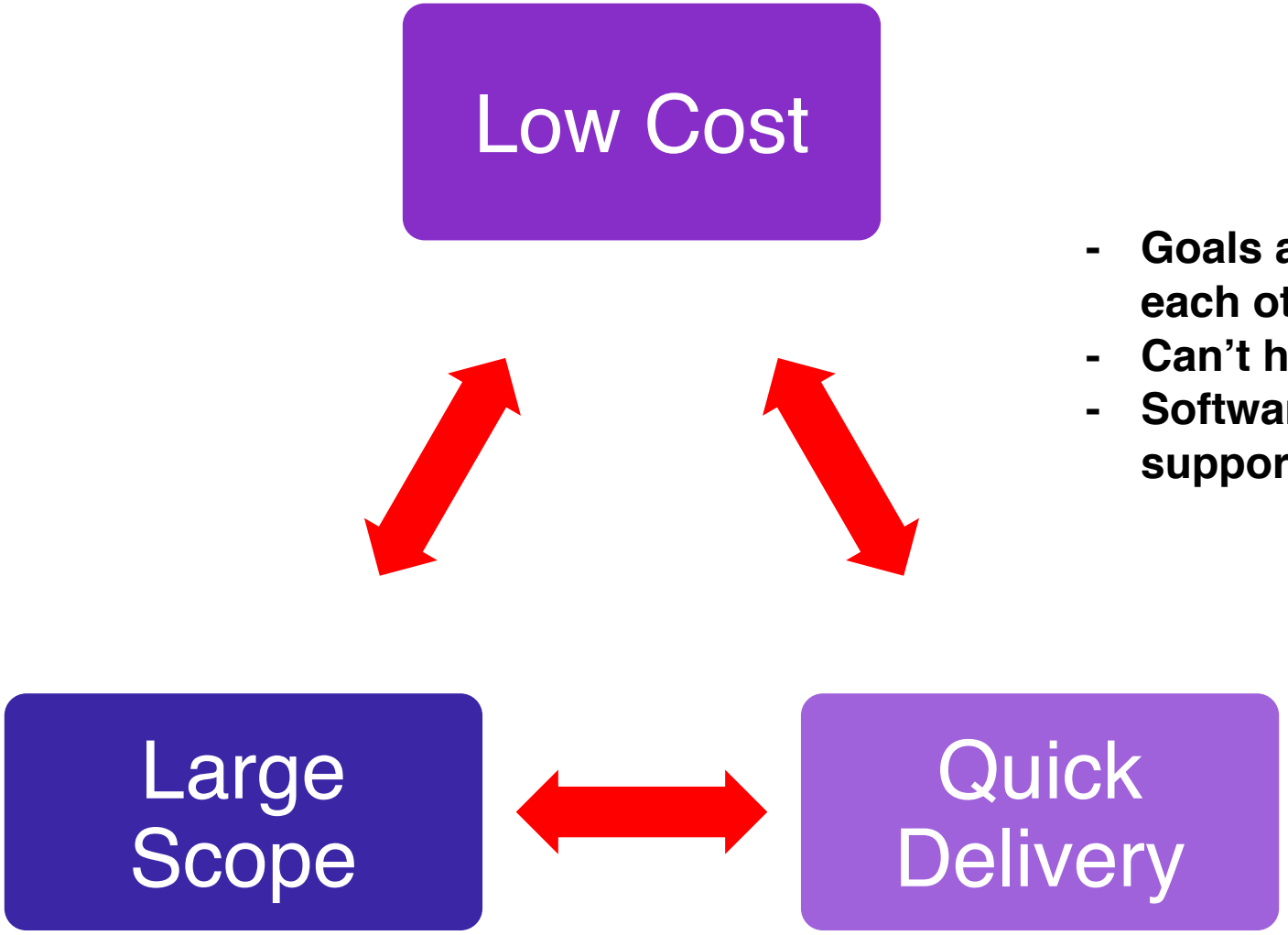
- **Correct**: does what it is supposed to (no bugs)

Must have

- **Robust**: secure, dependable, resilient
- **Useable**: UI, compatibility,…
- **Many features**: does lots of "good" stuff
- **Good performance**: fast, low energy usage
  - **Scalable**
- **Maintainable/extensible**: Can add and change features

Scope: What we must build

- **Quick delivery**: low development time
- **Low cost**: less spending is preferred



- **Goals are in opposition with each other**
- **Can’t have all 3**
- **Software engineering may support more of each**

- We have **10** developers to build our software product
  - Estimated time **1 year**
  - Developer salary: $125,000/year
  - Total cost: $1,250,000
- Can we just hire **20** developers and do it in **6 months**?
  - Each person does half as much work
  - Total cost still $1,250,000
  - Seems better, right?
- Could we take this further?
  - 40 developers in 3 months?
  - 3,650 developers in 1 day?
  - 87,600 developers in 1 hour?
  - 5,256,000 developers in 1 minute?

EMBARRASSINGLY PARALLEL PROBLEM

LITTLE OR NO EFFORT IS REQUIRED TO SEPARATE THE PROBLEM INTO A NUMBER OF **PARALLEL** TASKS

SOFTWARE DEVELOPMENT IS **NOT** EMBARRASSINGLY PARALLEL

- Significant communication & synchronization overheads
- 20 developers accomplish in 9 months
- 40 developers accomplish in 8 months
- Adding person power to a project often makes it later

“HOW DOES A SOFTWARE SYSTEM GET TO BE ONE YEAR LATE? ONE DAY AT A TIME”

– FREDERICK P. BROOKS, PHD

COROLLARY

- **Effective parallelism** between developers requires independent tasks
- Independent tasks are achieved by clearly **specified interfaces** that one developer can use while another implements

OBJECT-ORIENTED DESIGN FOUNDATIONS

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

DESIGN PRINCIPLES

- Least Surprise/Astonishment
- Don't Repeat Yourself (DRY)
- Low Coupling/High Cohesion
- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Separation
- Dependency Injection
- Design for testability

S.O.L.I.D.

We will keep coming back to these throughout the semester

LEAST SURPRISE/ ASTONISHMENT

- Reduce how much another developer is surprised by what happens
  - **No unexpected side-effects**
    - getSomething() should not change state
  - **No reliance on hidden knowledge to make it work**
    - "You have to call methodA right before methodB or it crashes"
    - "It got stuck in an infinite loop? Yeah, that happens if you pass a negative number to methodC…"
  - **Names make sense.**

DON’T REPEAT YOURSELF (DRY)

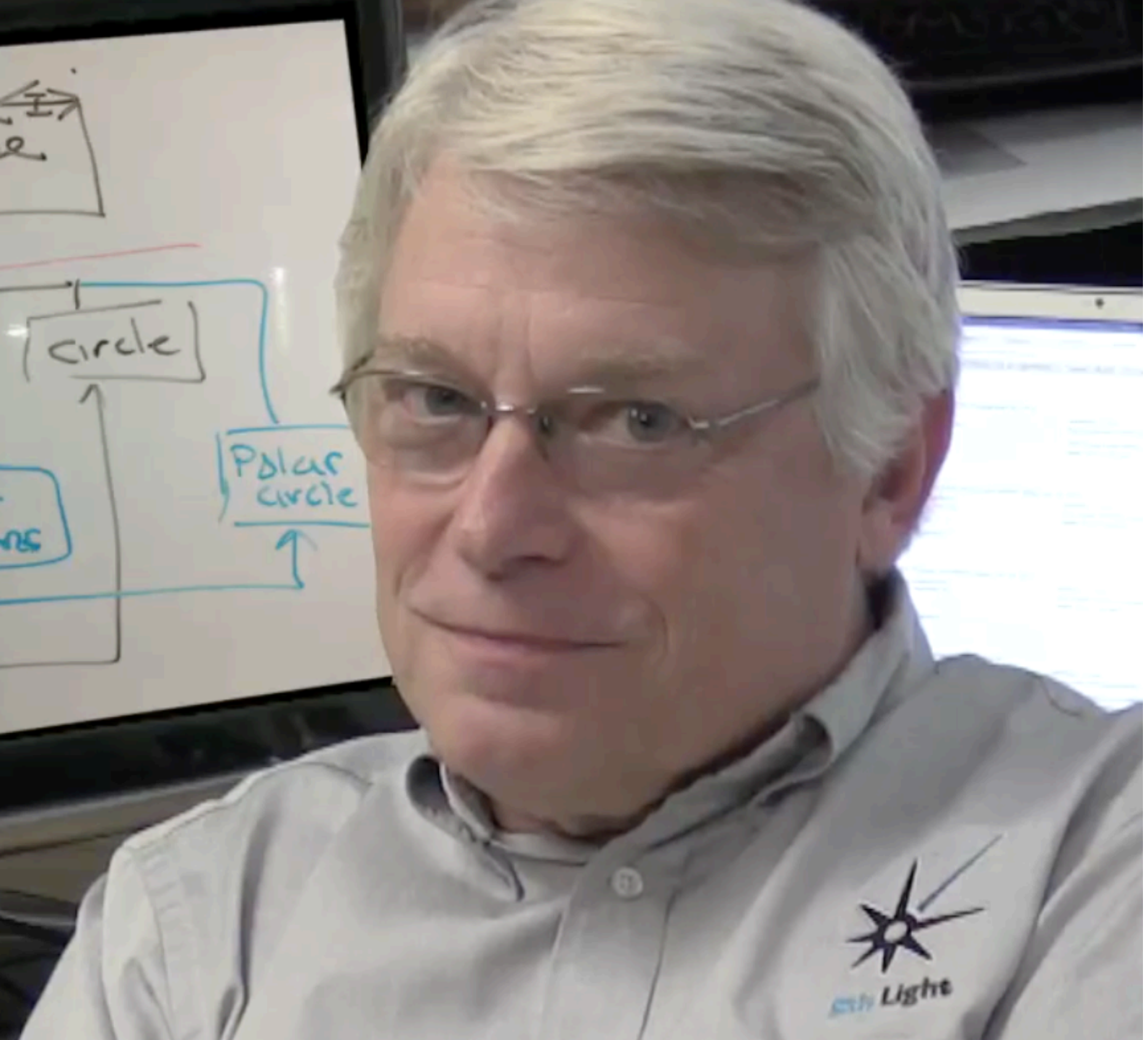
- Same code should only appear **once**
- If code needed more than once, use a function
- If constant needed more than once, give it a name
- **Gut Check**: If you change in place 1 and that requires you to make the same change in place 2, then you’ve violated DRY

LOW COUPLING/ HIGH COHESION

- **Coupling**: interdependence and coordination between entities (i.e., classes, functions, modules)
- **Cohesion**: how related are things inside one entity

SINGLE RESPONSIBILITY PRINCIPLE

- Software should do "one thing”



“UNCLE BOB”

- "Gather together the things that change for the same reasons. Separate those things that change for different reasons.”

SINGLE RESPONSIBILITY

- Suppose we have a program that does the following
  - Reads + parses some data
  - Performs some computation
  - Writes results
- How many responsibilities do we have?

SINGLE RESPONSIBILITY

- Suppose we have a program that does the following
  - Reads + parses some data
  - Performs some computation
  - Writes results
- How many responsibilities do we have?

At least two here:

1. Getting the data
   (open file, setup network connection, etc)
2. Parsing the input data format

SINGLE RESPONSIBILITY

- Suppose we have a program that does the following
  - Reads + parses some data
  - Performs some computation
  - Writes results
- How many responsibilities do we have?

At least two here:

1. Getting the data
   (open file, setup network connection, etc)
2. Parsing the input data format

Possibly many here, depending on how complex the computation is

SINGLE RESPONSIBILITY

- Suppose we have a program that does the following
  - Reads + parses some data
  - Performs some computation
  - Writes results
- How many responsibilities do we have?

At least two here:
1. Getting the data
   (open file, setup network connection, etc)
2. Parsing the input data format

Possibly many here, depending on how complex the computation is

At least one here

SINGLE RESPONSIBILITY

- Suppose we have a program that does the following
  - Reads + parses some data
  - Performs some computation
  - Writes results
- How many responsibilities do we have?

At least two here:
1. Getting the data
   (open file, setup network connection, etc)
2. Parsing the input data format

Possibly many here, depending on how complex the computation is

At least one here

Lets just look at these two responsibilities for one second

SINGLE RESPONSIBILITY

- Suppose we have a program that does the following
  - Reads + parses some data
  - Performs some computation
  - Writes results
- How many responsibilities do we have?

At least two here:
1. Getting the data
   (open file, setup network connection, etc)
2. Parsing the input data format

Possibly many here, depending on how complex the computation is

At least one here

Lets just look at these two responsibilities for one second

NOT SINGLE RESPONSIBILITY

Note: pseudo OO language,

```
DataSet readAndParse() {
    ConfigFileReader cfr = new ConfigFileReader("config.txt");
    String addr= cfr.getLineFor("datasource").getValue();
    InternetAddress addr = new InternetAddress(addr);
    Socket sock = new Socket(source);
    InputStream inp = sock.getInputStream();
    String line;
    DataSet ans = new DataSet();
    while((line = inp.readLine()) != null){
        String[] parts = splitUp(line);
        //some error checking on first part
        firstPart = doSomething(parts[0]);
        //some error checking on second part
        secondPart = doOtherThing(parts[1]);
        //some error checking on third part
        thirdPart = anotherFn(parts[2]);
        ans.add(new Data(firstPart, secondPart, thirdPart));
    }
    return ans;
}
```

Why is this bad?

NOT SINGLE RESPONSIBILITY

Note: pseudo OO language,

```
DataSet readAndParse() {
    ConfigFileReader cfr = new ConfigFileReader("config.txt");
    String addr= cfr.getLineFor("datasource").getValue();
    InternetAddress addr = new InternetAddress(addr);
    Socket sock = new Socket(source);
    InputStream inp = sock.getInputStream();
    String line;
    DataSet ans = new DataSet();
    while((line = inp.readLine()) != null){
        String[] parts = splitUp(line);
        //some error checking on first part
        firstPart = doSomething(parts[0]);
        //some error checking on second part
        secondPart = doOtherThing(parts[1]);
        //some error checking on third part
        thirdPart = anotherFn(parts[2]);
        ans.add(new Data(firstPart, secondPart, thirdPart));
    }
    return ans;
}
```

Three responsibilities

1. Configuration
2. Network connection setup
3. Parsing input

Why does this matter?

NOT SINGLE RESPONSIBILITY

Note: pseudo OO language,

```
DataSet readAndParse() {
    ConfigFileReader cfr = new ConfigFileReader("config.txt");
    String addr= cfr.getLineFor("datasource").getValue();
    InternetAddress addr = new InternetAddress(addr);
    Socket sock = new Socket(source);
    InputStream inp = sock.getInputStream();
    String line;
    DataSet ans = new DataSet();
    while((line = inp.readLine()) != null){
        String[] parts = splitUp(line);
        //some error checking on first part
        firstPart = doSomething(parts[0]);
        //some error checking on second part
        secondPart = doOtherThing(parts[1]);
        //some error checking on third part
        thirdPart = anotherFn(parts[2]);
        ans.add(new Data(firstPart, secondPart, thirdPart));
    }
    return ans;
}
```

What if we want to….

- connect to a different network address?
- read data from a file instead of network?
- support a new version of the data format?

SINGLE RESPONSIBILITY

Note: pseudo OO language,

```
DataSet readAndParse(Socket sock) {
    InputStream inp = sock.getInputStream();
    String line;
    DataSet ans = new DataSet();
    while((line = inp.readLine()) != null){
        String[] parts = splitUp(line);
        //some error checking on first part
        firstPart = doSomething(parts[0]);
        //some error checking on second part
        secondPart = doOtherThing(parts[1]);
        //some error checking on third part
        thirdPart = anotherFn(parts[2]);
        ans.add(new Data(firstPart, secondPart, thirdPart));
    }
    return ans;
}
```

This code is **better**, but **not great**

\+ Follows Single Responsibility

\- Tightly coupled to use of a Socket

REDUCE COUPLING

Note: pseudo OO language,

```
DataSet parseData(InputStream inp) {
    String line;
    DataSet ans = new DataSet();
    while((line = inp.readLine()) != null){
        String[] parts = splitUp(line);
        //some error checking on first part
        firstPart = doSomething(parts[0]);
        //some error checking on second part
        secondPart = doOtherThing(parts[1]);
        //some error checking on third part
        thirdPart = anotherFn(parts[2]);
        ans.add(new Data(firstPart, secondPart, thirdPart));
    }
    return ans;
}
```

This code is **even better:**

+ Follows Single Responsibility
+ Works on any InputStream: not coupled to data coming from network

SRP AT CLASS LEVEL (AND COHESION)

Note: pseudo OO language,

```
class DataParser {
   public:
      InternetAddress getSourceAddress() {…}
      InputStream getInputStream(InternetAddress addr) {…}
      DataSet parseData(InputStream inp) {…}

}
```

This does not follow SRP at the class level.
These belong in different classes.
Note that this class has low cohesion: three very different tasks grouped together
- Two of the tasks do not even go with the name.

SRP AT CLASS LEVEL (AND COHESION)

Note: pseudo OO language,

```
class DataParser {
   public:
      InternetAddress getSourceAddress() {…}
      String getSourceFileName() {…}
      InputStream getInputStream(InternetAddress addr) {…}
      InputStream getInputStream(String filename) {…}
      DataSet parseData(InputStream inp) {…}

}
```

As we make our changes, our class becomes a big confusing mess.

GUT CHECK

IF YOUR DESCRIPTION OF A CLASS HAS MULTIPLE “ANDS” THEN IT PROBABLY NEEDS TO BE REFACTORED.

OPEN/CLOSED PRINCIPLE

- **Software entities should be open to extension but closed to modification**
- We should be able to use software entities in new ways, without changing the code
  - Changing code risks introducing bugs
    - Sometimes might not even have access to code.
  - But use for things we did not think of when we wrote it

OPEN/CLOSED PRINCIPLE

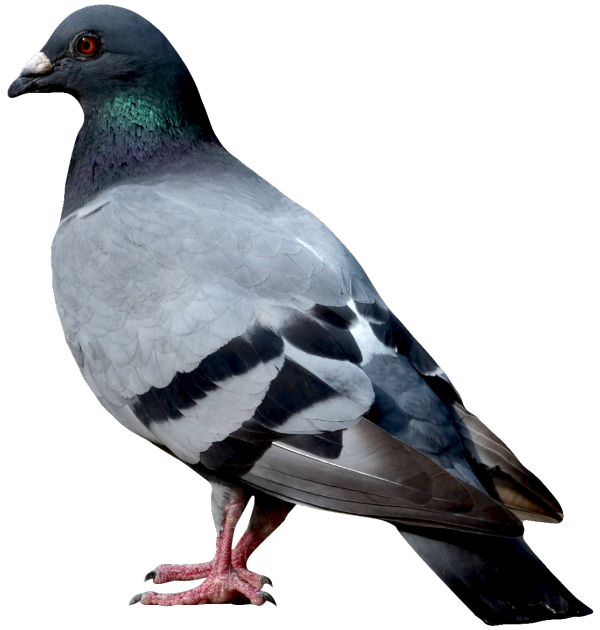
- Writing software is hard
  - Testing software is hard
  - Anytime we change something, we might break it

OPEN/CLOSED PRINCIPLE

- Writing software is hard
  - Testing software is hard
  - Anytime we change something, we might break it
- If we have written and tested code and are confident in that code, we want to NOT change it

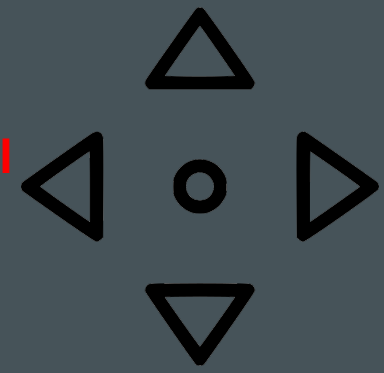
OPEN/CLOSED PRINCIPLE

- Writing software is hard
  - Testing software is hard
  - Anytime we change something, we might break it
- If we have written and tested code and are confident in that code, we want to NOT change it
- But we'd like to be able to use that code in new ways
  - As we add features
  - As we change behaviors
  - As we re-use code across projects



FLYING BIRDS PROGRAM

Use to Control bird motion



Source: Tom Dalling

```
 9  class Bird {
10  public:
11      virtual void setLocation(double longitude, double latitude) = 0;
12      virtual void setAltitude(double altitude) = 0;
13      virtual void draw() = 0;
14  };
```

VERSION 1: IS A SUCCESS!

BUT WHAT IF WE USED IF/ELSE FOR BIRDS?

- What if we want to check the bird type in 100 places?
- What if we want to add Hawk with this structure?

```
8  void someMethod(Bird b) {
9      if (b.getType() == Bird.PIGEON) {
10         //pigeon code
11     }
12     else if (b.getType() == Bird.PARROT) {
13         //parrot code
14     }
15     else if (b.getType() == Bird.EAGLE) {
16             //eagle code
17     }
18     //…
19 }
```

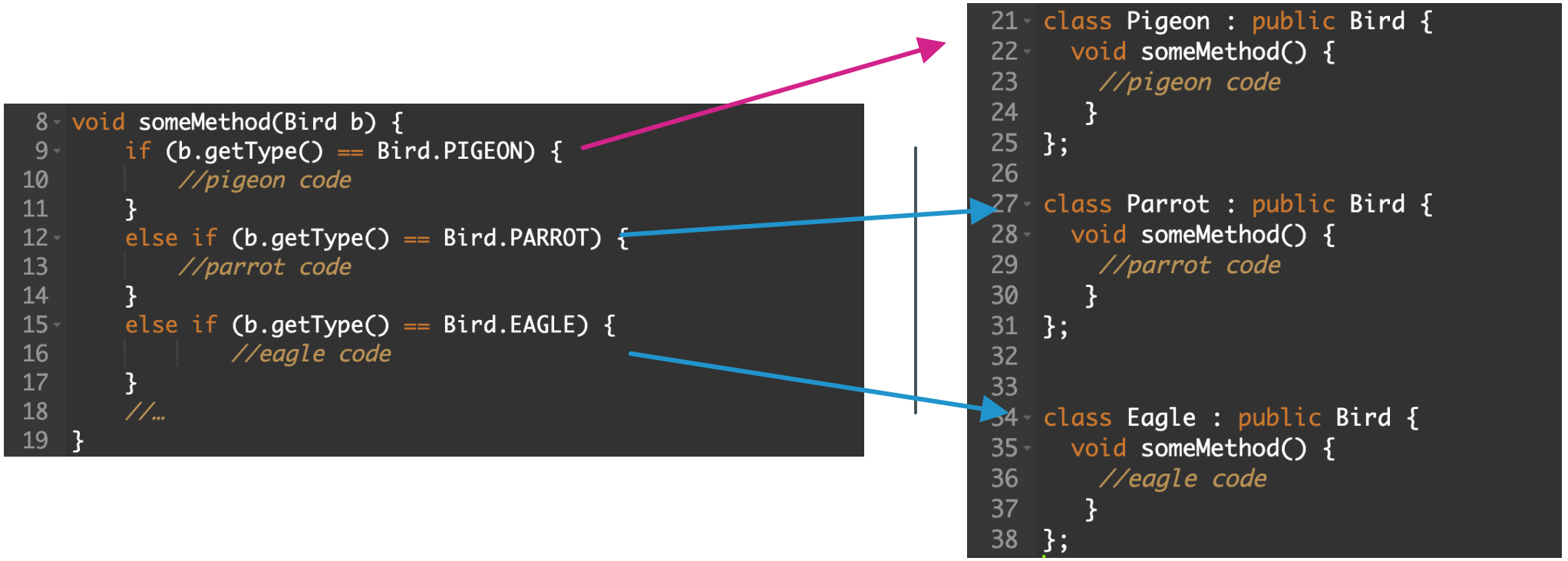
Note that if we add has by writing
else if (b.getType() == Bird.HAWK) { ....}
we risk breaking existing code :(

BUT WHAT IF WE USED IF/ELSE FOR BIRDS?

- What if we want to check the bird type in 100 places?
- What if we want to add Hawk with this structure?

```
 8  void someMethod(Bird b) {
 9      if (b.getType() == Bird.PIGEON) {
10          //pigeon code
11      }
12      else if (b.getType() == Bird.PARROT) {
13          //parrot code
14      }
15      else if (b.getType() == Bird.EAGLE) {
16              //eagle code
17      }
18      //…
19  }
```

If/Else and switch/case are the enemies of Open/Closed
- Especially on object types
- This does NOT mean "never use if"



WHAT SHOULD WE DO INSTEAD?

- Dynamic dispatch and polymorphism

Instead of someMethod(b) we do b.someMethod();

```
16  void Hawk::setAltitude(double altitude)
17  {
18      //code here
19  }
```

VERSION 2: ADDS A HAWK AND 10 OTHER BIRDS WITH NO ISSUE

Open/Closed Principle Satisfied

VERSION 3: ADDS A PENGUIN

DO YOU EXPECT ANY ISSUE?

PENGUIN IS-A BIRD, RIGHT?

- A penguin is-a bird, right?
- So use inheritance: `class Penguin : public Bird{ .... };`

PENGUIN IS-A BIRD, RIGHT?

- A penguin is-a bird, right?
- So use inheritance: `class Penguin : public Bird{ .... };`
- But birds fly and penguins dont
  - But we can make that work, right?

PENGUIN IS-A BIRD, RIGHT?

- A penguin is-a bird, right?
- So use inheritance:
- But birds fly and penguins dont
  - But we can make that work, right?

```
class Bird {
   virtual void fly(int altitude) {
          //whatever code to make it fly up to height
    }
};
class Penguin : public Bird
   virtual void fly(int altitude) {
        //do nothing: penguins dont fly
   }
};
```

Does this seem like a good idea or a bad idea?

PENGUIN IS-A BIRD, RIGHT?

- A penguin is-a bird, right?
- So use inheritance:
- But birds fly and penguins dont
  - But we can make that work, right?

```
class Bird {
   virtual void fly(int altitude) {
          //whatever code to make it fly up to height
    }
};
class Penguin : public Bird
   virtual void fly(int altitude) {
        //do nothing: penguins dont fly
   }
};
```

Does this seem like a good idea or a bad idea?

The rest of our code has certain expectations of what `b->fly(n)` means.
It expects certain results (the bird `b` flies at height `n`) and we now have birds that do not obey those rules

PENGUIN IS-A BIRD, RIGHT?

```
class Penguin : public Bird {...}
```

- Remember that inheritance gives sub-type polymorphism
  - A Penguin Is-A Bird
  - It has all the methods of Birds
  - We can use a Penguin wherever we need a Bird
    - (In C++, with pointers and references)
- The type system ensures that sub-type polymorphism is ok in terms of methods **existing**
- But what about method **behavior**?

PENGUIN IS-A BIRD, RIGHT?

```
class Penguin : public Bird {...}
```

- Remember that inheritance gives sub-type polymorphism
  - A Penguin Is-A Bird
  - It has all the methods of Birds
  - We can use a Penguin wherever we need a Bird
    - (In C++, with pointers and references)
- The type system ensures that sub-type polymorphism is ok in terms of methods **existing**
- But what about method **behavior**?

```
void goToNext(Bird * b, Nest *n) {
   b->fly(n->getHeight());
   //reasonable to expect b is flying
   //to that height
   ...
}
```

```
goToNest(somePenguin, itsNext); //broken
```

PENGUIN IS-A BIRD, RIGHT?

```
class Penguin : public Bird {...}
```

- Remember that inheritance gives sub-type polymorphism
  - A Penguin Is-A Bird
  - It has all the methods of Birds
  - We can use a Penguin wherever we need a Bird
    - (In C++, with pointers and references)
- The type system ensures that sub-type polymorphism is ok in terms of methods **existing**
- But what about method **behavior**?

```
void goToNext(Bird * b, Nest *n) {
  b->fly(n->getHeight());
  //reasonable to expect b is flying
  //to that height
  ...
}
```

```
goToNest(somePenguin, itsNext); //broken
```

We shouldn't really have "reasonable to expect"... What do we actually mean?

PENGUIN IS-A BIRD, RIGHT?

```
class Penguin : public Bird {...}
```

- Remember that inheritance gives sub-type polymorphism
  - A Penguin Is-A Bird
  - It has all the methods of Birds
  - We can use a Penguin wherever we need a Bird
    - (In C++, with pointers and references)
- The type system ensures that sub-type polymorphism is ok in terms of methods **existing**
- But what about method **behavior**?

```
void goToNext(Bird * b, Nest *n) {
  b->fly(n->getHeight());
  //reasonable to expect b is flying
  //to that height
  ...
}
```

```
goToNest(somePenguin, itsNext); //broken
```

We shouldn't really have "reasonable to expect"... What do we actually mean?
The **postcondition** of fly method is that the bird is flying to that altitude.

If you took Algorithms with Drew, this should be sooooo familiar! If not, more soon.

PRECONDITIONS & POST-CONDITIONS

Preconditions: things that are supposed to be true before the method is called

Post-conditions: things that are true after the method (assuming preconditions were satisfied)

POSTCONDITIONS

- Bird promised something
- Penguin did not deliver

- This is not ok
  - Code that uses Bird broken
  - Relies on promised behaviors
  - Behaviors not provided

```
class Bird {
   //postcondition: bird will be flying towards altitude
   virtual void fly(int altitude) {
          //whatever code to make it fly up to height
    }
};
class Penguin : public Bird
   //postcondition: bird's behavior unchanged
   virtual void fly(int altitude) {
        //do nothing: penguins dont fly
   }
};
```

LISKOV SUBSTITUTION PRINCIPLE

- Formally, the Penguin subclass violates the **Liskov Substitution Principle.**
- **Substitution:** being able to put one thing (Penguin) in place of another (Bird)

LISKOV SUBSTITUTION

- Ability to replace any instance of a superclass with an instance of one of its subclasses without negative side effects
- Not the same as subtype polymorphism because ”not breaking the program” is a requirement

PRECONDITIONS & POST-CONDITIONS

Preconditions: things that are supposed to be true before the method is called

Post-conditions: things that are true after the method (assuming preconditions were satisfied)

PRECONDITIONS & POST-CONDITIONS

```
template<typename T>
class LinkedList {
        //precondition:  0<=index< number of elements in list
        //postcondition: return value is indexth element
        T& operator[] (size_t index) {….}

        //precondition: none
        //postcondition: this list contains no values that == toRemove
        void removeAll(const T& toRemove) {…}
};
```

SUBCLASSES THAT DO NOT MAKE SENSE

```
template<typename T>
class LinkedList {
        //precondition:  0<=index< number of elements in list
        //postcondition: return value is indexth element
        virtual T& operator[] (size_t index) {….}

        //precondition: none
        //postcondition: this list contains no values that == toRemove
        virtual void removeAll(const T& toRemove) {…}
};
```

Suppose we made WeirdList, a LinkedList subclass that overrode these such that…

1. operator[] only works on the first 100 elements
2. removeAll does not remove the first element

Would code that uses LinkedList work correctly if we pass it a WeirdList?

SUBCLASSES THAT DO NOT MAKE SENSE

```
void myFunction(LinkedList & ll) {
   for (size_t i = 0; i < ll.getSize(); i++) {
        T& data = ll[i];          //we expect this to work: precondition is 0<=index<size, and we enforce that
         //other code that uses data
    }
}

void otherFn(LinkedList & ll){
    //some other code
    ll.removeAll(someValue);
    //you expect ll not to have someValue at all here
    //and this code might rely on it
}
```

Suppose we made WeirdList, a LinkedList subclass that overrode these such that…

1. operator[] only works on the first 100 elements
2. removeAll does not remove the first element

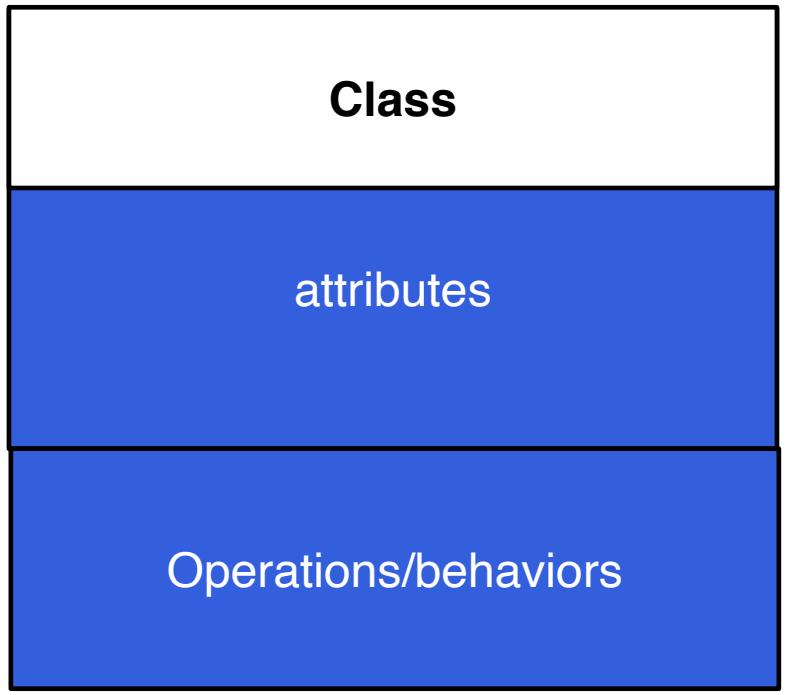
Would code that uses LinkedList work correctly if we pass it a WeirdList?

No: even though inheritance guarantees the methods exist, we have overridden them in ways that breaks the code.

LISKOV SUBSTITUTION

Preconditions: things that are supposed to be true before the method is called	• Must not be strengthened in a subclass
Post-conditions: things that are true after the method (assuming preconditions were satisfied)	• Must not be weakened in a subclass
Invariants: must be valid before and after a method call	• Invariants of the superclass must be preserved in the subclass
History: observable states of the object	• Any history must be valid for the superclass

QUICK PREVIEW: UML CLASS DIAGRAMS

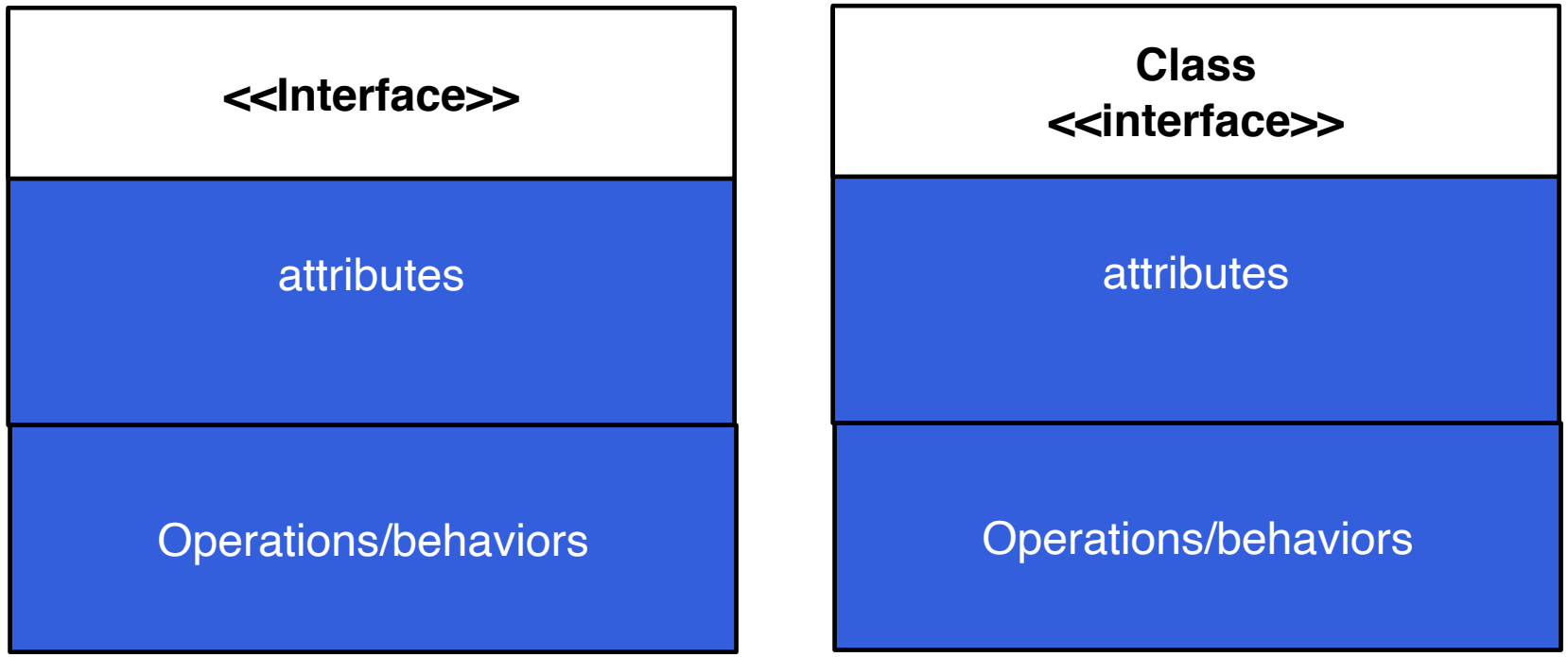


- We will discuss more in a forthcoming lecture
- Simplified class diagram to visually describe a system's organization
- Inheritance

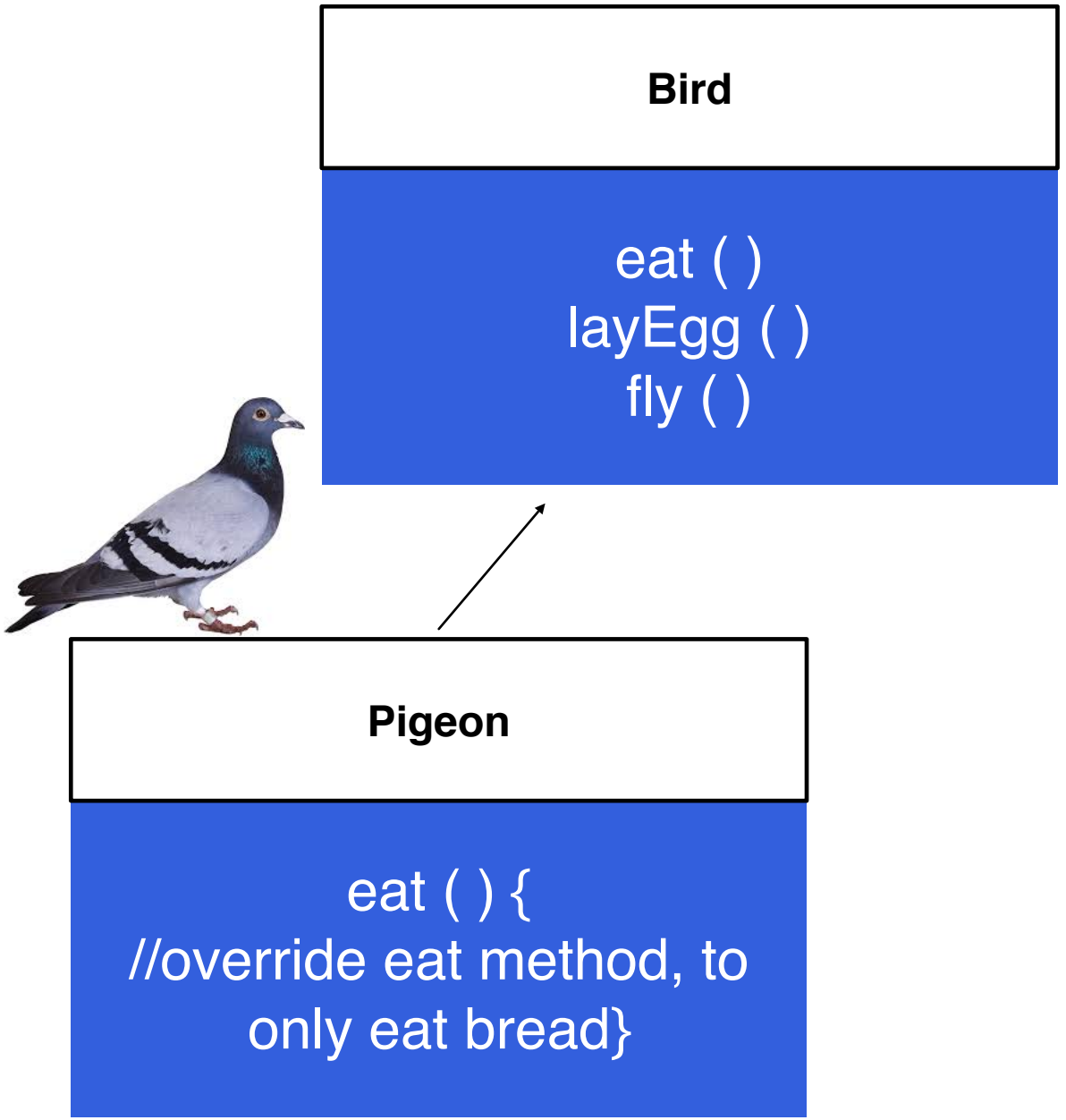
Generalization (class extends another class)

Realization (class implements an interface)

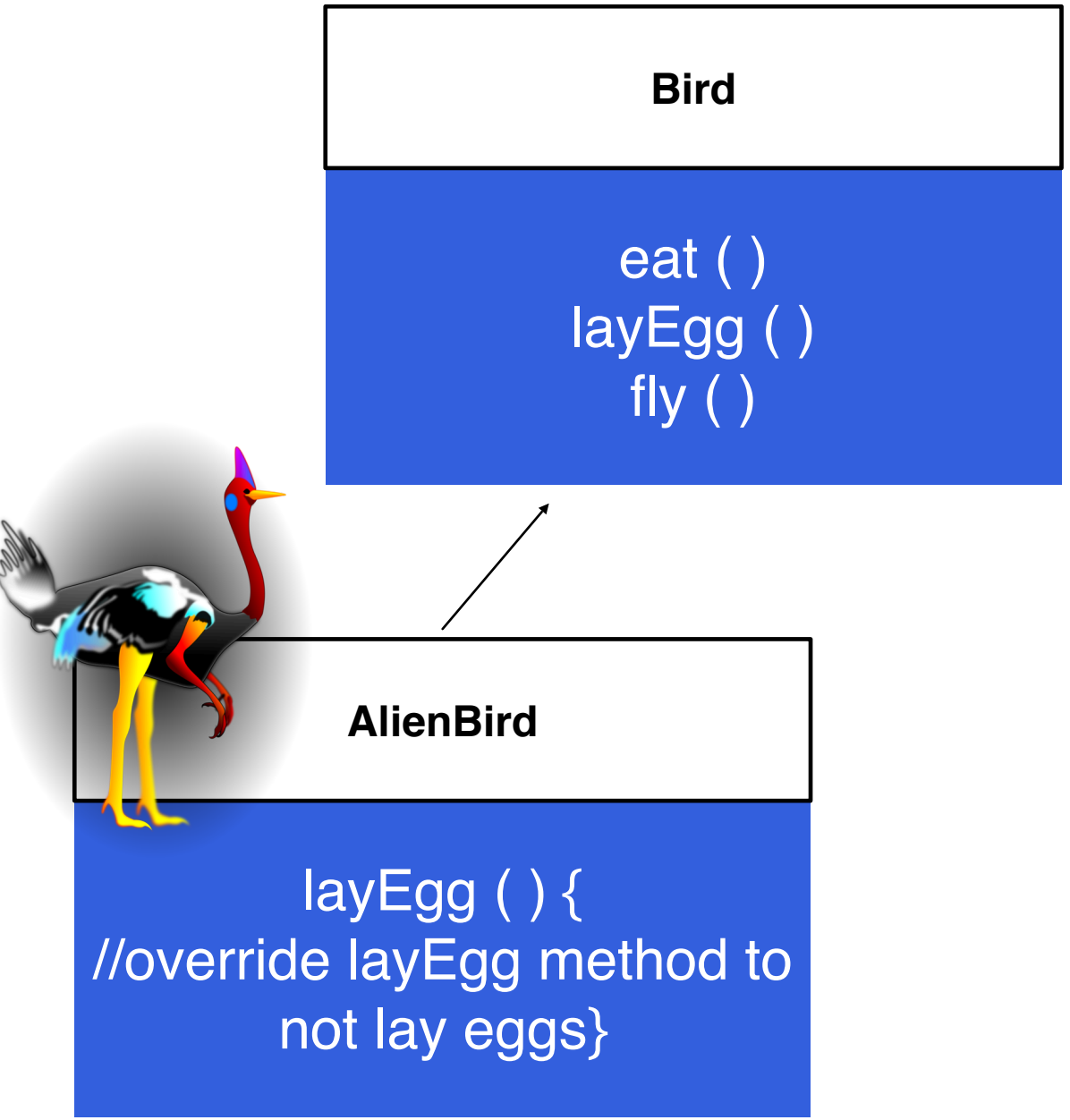
QUICK PREVIEW: UML CLASS DIAGRAMS (2)



- Interface representations



- Subclass should not strengthen preconditions
- Stated Differently: **Requirements on inputs to a subclass cannot be stricter than its base class**
- If attempt bird.eat(worm), then raises an error
- Violates LSP because Pigeon expectations for input (only bread) stricter than Bird



- Postconditions must not be weakened in a subtype
- Stated Differently: **Possible outputs from a subclass must be more than or equally as restrictive as from the base class**
- If attempt bird.hatch( ) exception raised because no return in alien subclass
- Violates LSP because there are less possible outputs. So post conditions are weakened

INVARIANTS MUST ALWAYS BE TRUE

Example Bird Invariant:

If *isFlying()* is true, then

get Altitude() > 0

All subclasses must ensure this relationship too!

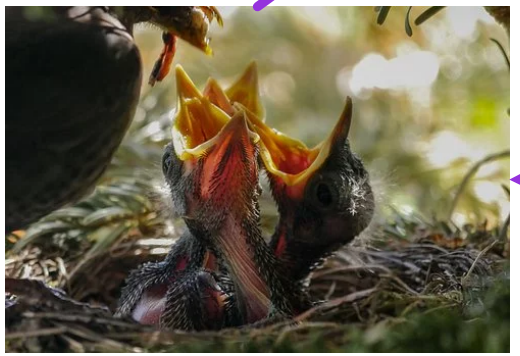
Egg

Hatching

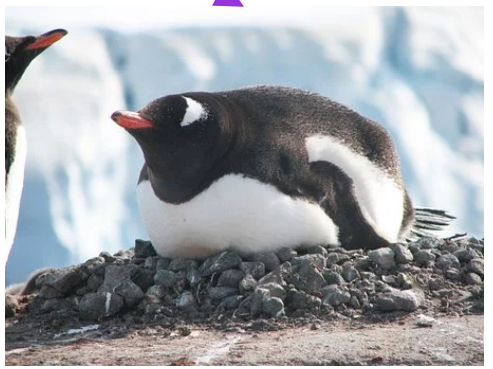
Baby

Adult

A sub-class that can go from Adult->Baby
Violates LSP



Parenting



Incubating

BARBARA LISKOV

MIT INSTITUTE PROFESSOR AND HEAD OF THE PROGRAMMING METHODOLOGY GROUP

INTERFACE SEGREGATION PRINCIPLE

- A client shouldn’t be forced to depend on interfaces they don’t use
- Better to have many small interfaces than a few very large interfaces
- “Too many is better than too few”

NOT INTERFACE SEGREGATION PRINCIPLE (ISP)

Note: pseudo OO language,

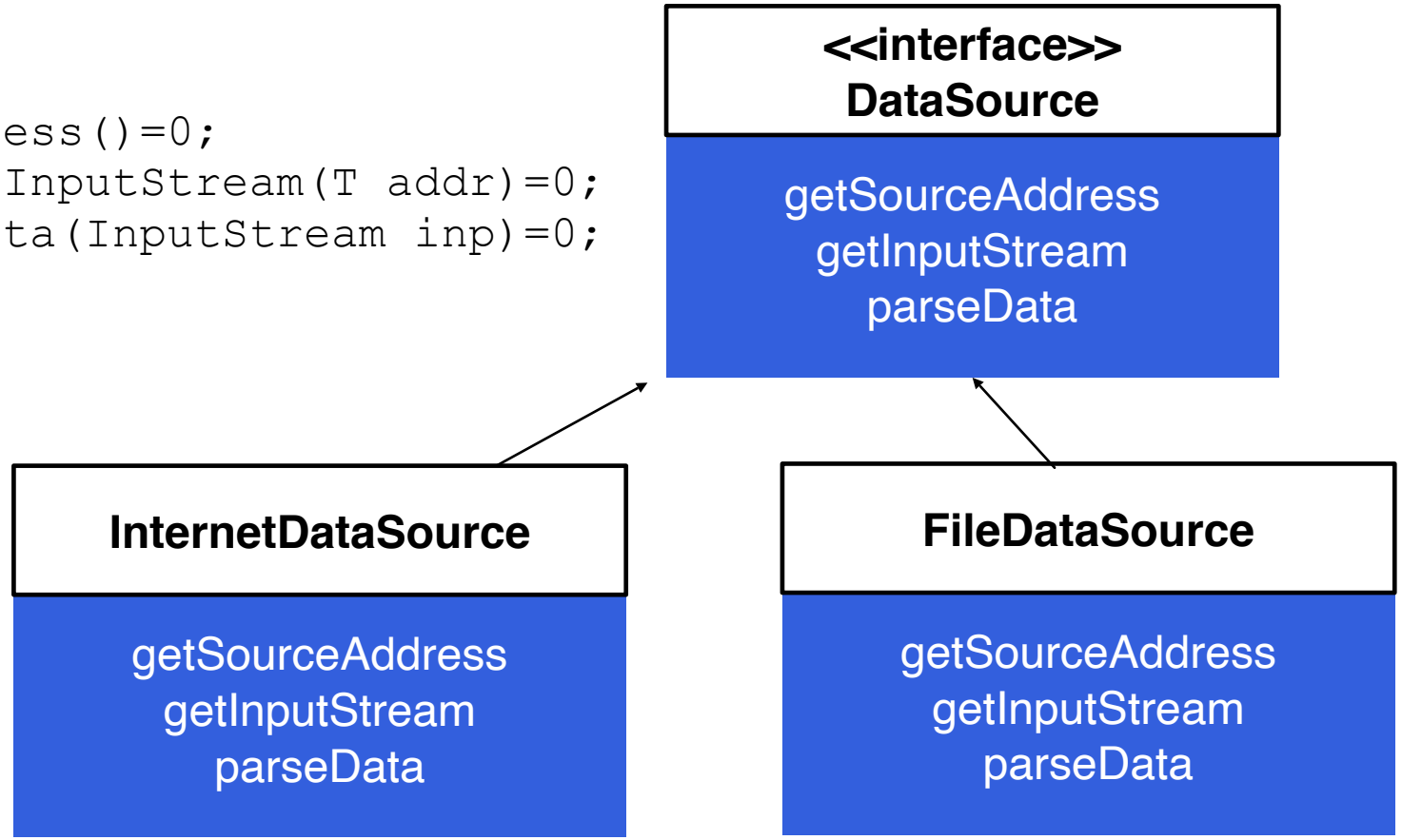
```
class DataParser {
   public:
      InternetAddress getSourceAddress() {…}
      InputStream getInputStream(InternetAddress addr) {…}
      DataSet parseData(InputStream inp) {…}

}
```

Let's go back to our DataParser for a moment.
Suppose we decide to make an abstract parent class for a data source
(as we will see soon, Java has an "interface" which is like a purely abstract class)

NOT INTERFACE SEGREGATION PRINCIPLE (ISP)

Note: pseudo OO language,

```
template<typename T>
class DataSource {
   public:
      virtual T getSourceAddress()=0;
      virtual InputStream getInputStream(T addr)=0;
      virtual DataSet parseData(InputStream inp)=0;
};
```

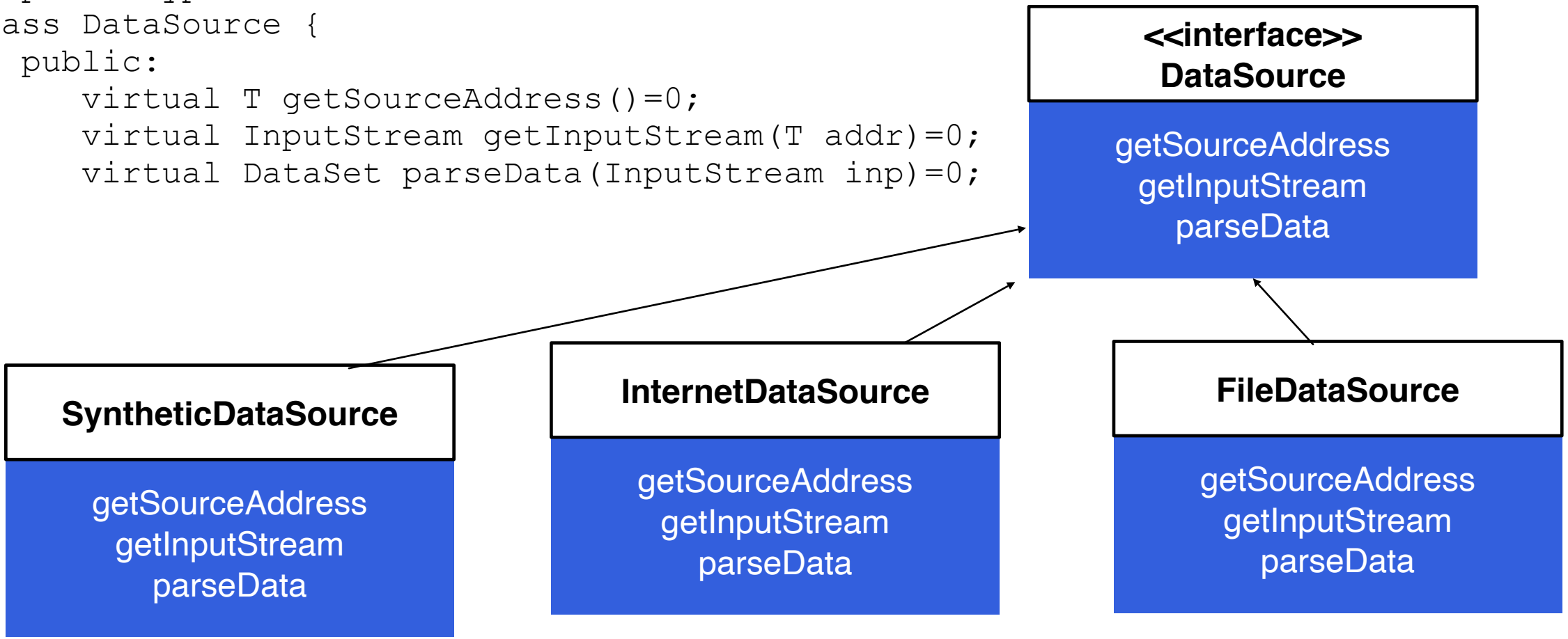


Now this could serve as the parent of e.g. InternetDataSource and FileDataSource…
Code that gets the data could then take a DataSource.

NOT INTERFACE SEGREGATION PRINCIPLE (ISP)

Note: pseudo OO language,

```
template<typename T>
class DataSource {
   public:
      virtual T getSourceAddress()=0;
      virtual InputStream getInputStream(T addr)=0;
      virtual DataSet parseData(InputStream inp)=0;
};
```

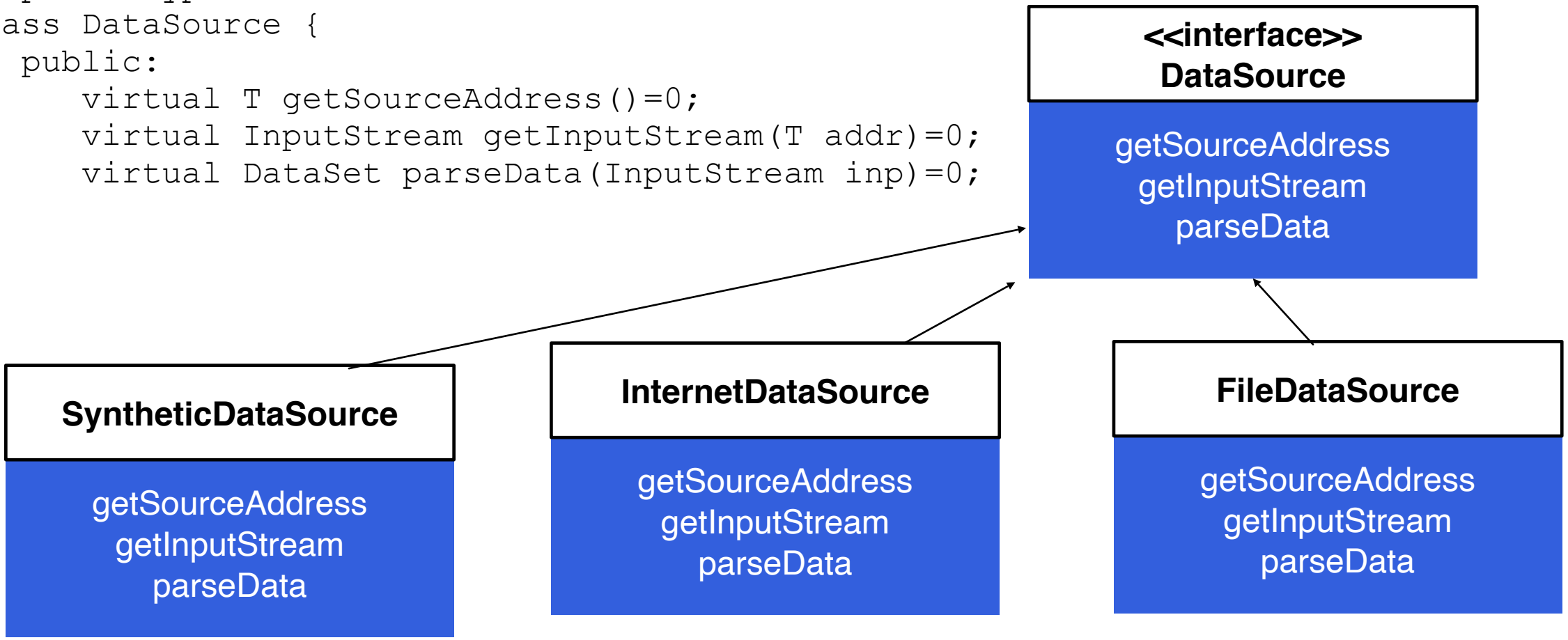


Now we want to make a synthetic data source (i.e., generate data according to some distribution)…
All our code takes a DataSource, so we implement that interface…

NOT INTERFACE SEGREGATION PRINCIPLE (ISP)

Note: pseudo OO language,

```
template<typename T>
class DataSource {
   public:
      virtual T getSourceAddress()=0;
      virtual InputStream getInputStream(T addr)=0;
      virtual DataSet parseData(InputStream inp)=0;
};
```



But getSourceAddress and getInputStream don't make sense!
We only want parseData…. But we have to implement the other two to have a concrete class..

NOT INTERFACE SEGREGATION PRINCIPLE (ISP)

Note: pseudo OO language,

```
class SyntheticDataSource: public DataSource<int> {
   public:
        virtual int getSourceAddress(){
           throw some_exception("this method is not appropriate to this type");
        }
        virtual InputStream getInputStream(int x){
           throw some_exception("this method is not appropriate to this type");
        }
        virtual DataSet parseData(InputStream inp) {
          //whatever code
        }
};
```

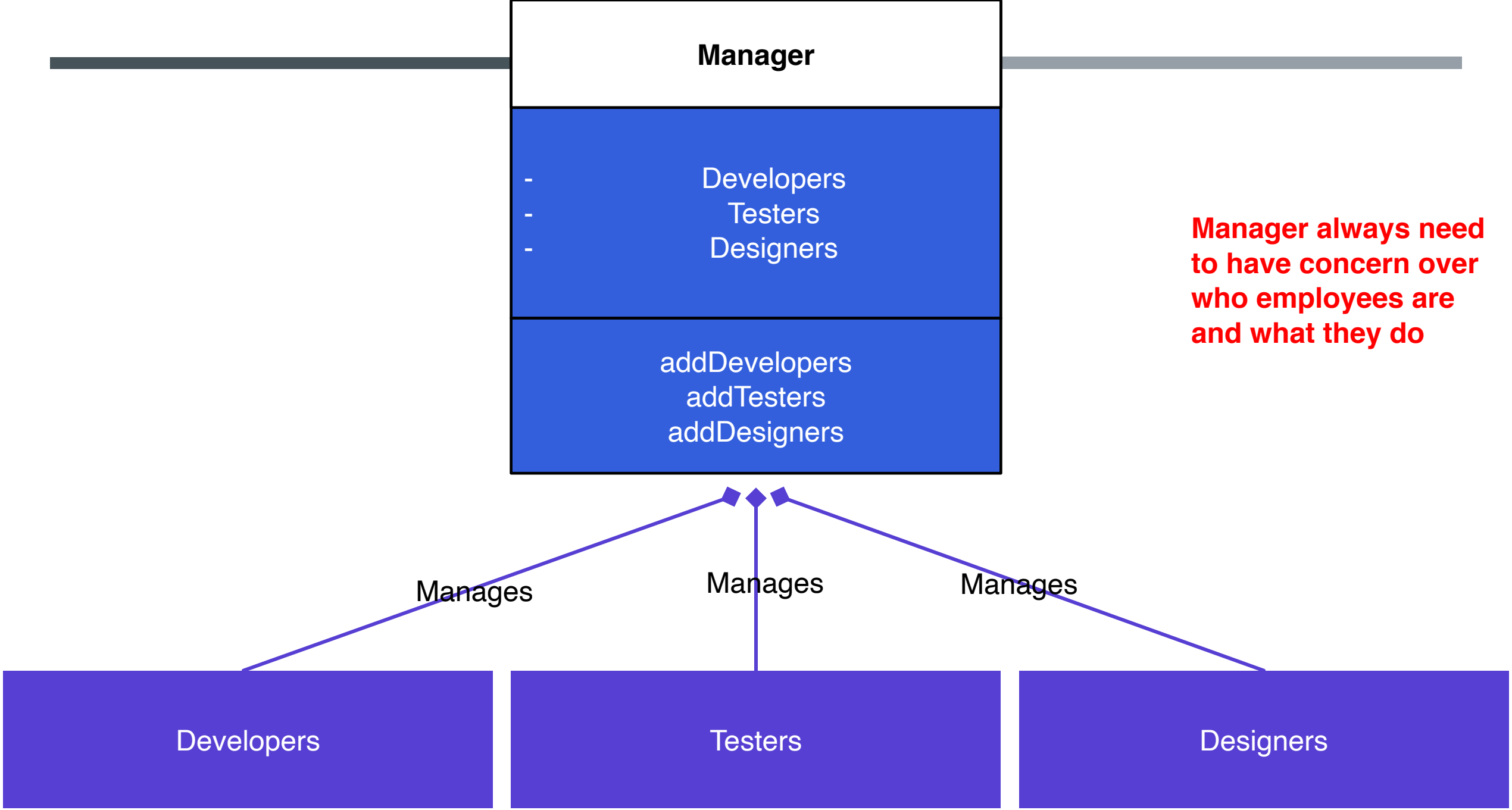
Then we end up with bogus implementations, as above.

Key indicator of ISP violations: forced to implement a method that does not make sense

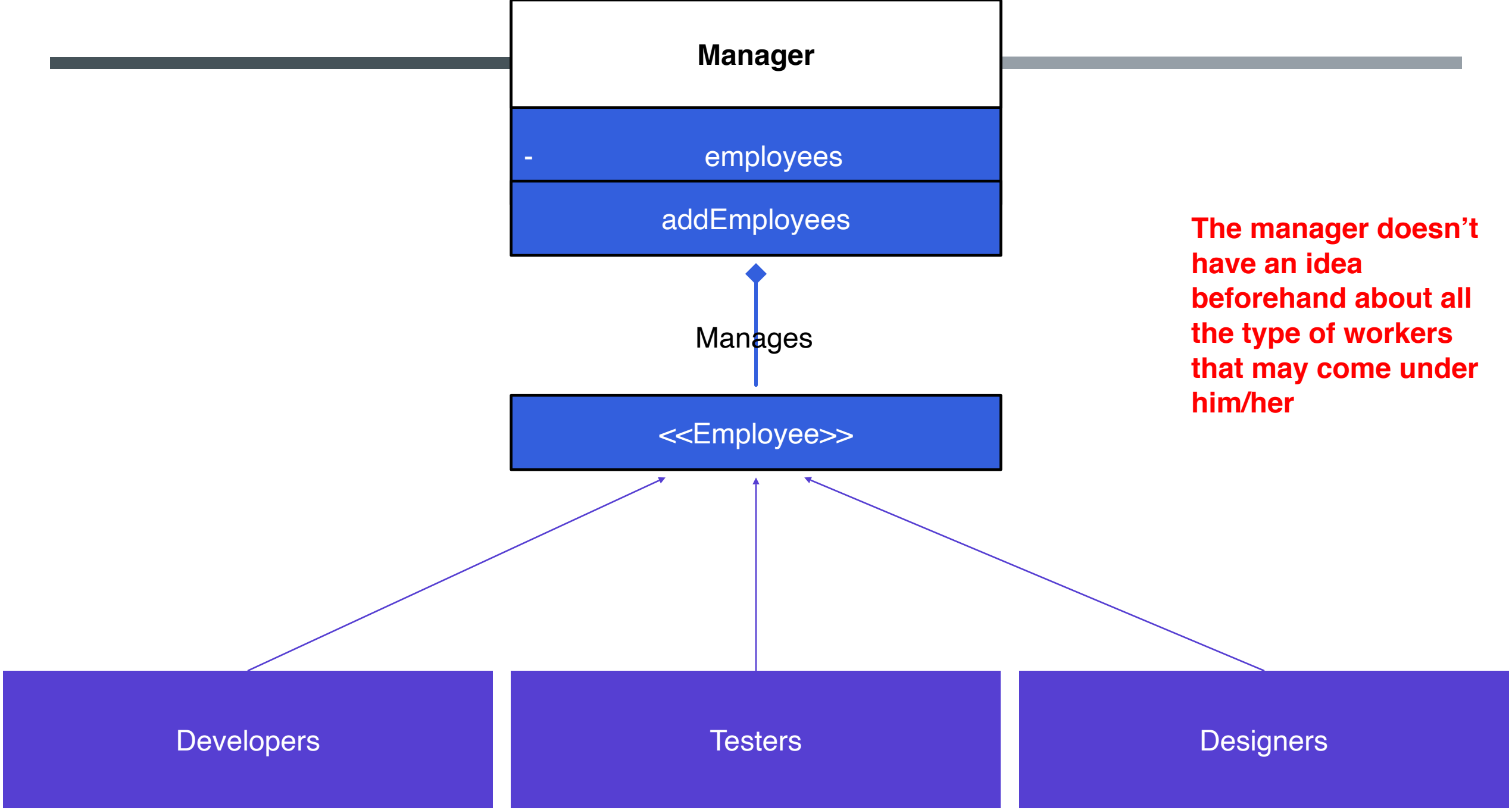
- Either it just throws an exception because it is not relevant for that type
- Or (even worse) you put a dummy implementation, e.g., return 0 and return NULL above.

DEPENDENCY INVERSION PRINCIPLE

- HIGH LEVEL OBJECTS SHOULD NOT DEPEND ON LOW LEVEL IMPLEMENTATIONS
- Higher classes are not dependent on the lower classes instead depend upon the abstraction of the lower classes



Manager always need to have concern over who employees are and what they do



The manager doesn't have an idea beforehand about all the type of workers that may come under him/her

DEPENDENCY INJECTION

AN OBJECT RECEIVES THE OTHER OBJECTS IT DEPENDS ON FROM OUTSIDE

DEPENDENCY INVERSION & DEPENDENCY INJECTION

```
class Manager  {
    std::vector<Developer> developers;
     void addDeveloper() {
            developers.push_back(Developer()); //not dependency injection
    }

}
```

- Notice that our first Manager which does not do Dependency Inversion does not do Dependency Injection
  - Manager creates the objects on which it depends (its employees)
  - Creates tight coupling between Manager and specific Employee types
  - Violates Open/Closed principle: add new employee types -> change Manager

DEPENDENCY INVERSION & DEPENDENCY INJECTION

```
class Manager  {
    std::vector<Employee *> employees;
     void addDeveloper() {
                employees.push_back(Developer()); //not dependency injection
    }

}
```

- Changing the data type helps a bit…
  - Other code in Manager that iterates over employees can remain unchanged with new Employee type
  - …but we still hard code the creation of specific types
    - Same problems as before: tight coupling + not open/closed

DEPENDENCY INVERSION & DEPENDENCY INJECTION

```
class Manager  {
    std::vector<Employee *> employees;
     void addEmployee(Employee * e) {  //dependency injection
                employees.push_back(e);
    }

}
```

- Now using dependency injection
  - This object (Manager) depends on other objects (Employees)
  - Those other objects (Employees) now come from outside this object
  - New employee types created? Can be added to Manager with no changes (open/closed principle)
    - Manager no longer depends on specific Employee types!

DEPENDENCY INJECTION: OFTEN IN CONSTRUCTOR

```
class Employee  {
   Employee * boss;
   Employee(Employee * b): boss(b) {}
}
```

- We frequently see dependency injection in constructors
  - Take parameters for other objects on which this object depends

DESIGNING FOR TESTING

- **Controllability**: how well can my test code control the state of the module I want to test?
- **Observability**: how well can my test code inspect the state (to see if correct) of the module I want to test?
- **Isolatability**: how well can I separate out the module I want to test it by itself?

TESTING AND SRP

HOW DOES FOLLOWING THE SINGLE RESPONSIBILITY PRINCIPLE HELP WITH TESTING?