Tell, Don’t Ask

- Ask for help, not information

Never ask an object for information that you need to do something; rather, ask the object that has the information to do the work for you.

In other words: Don’t use any getters/setters/properties.

Avoid getters and setters

- Wrong

```
Money a, b, c;
//...
a.setValue( a.getValue() +
                b.getValue() );
```

- Right

```
Money a, b, c;
//...
a.increaseBy( b );
```

Improve

```
if (aCargo.getStatus() ==
          HandlingStatus.MISDIRECTED)
    ...

if (aCargo.isMisdirected())
    ...
```

Compare

```
Dog dog = new Dog();
dog.setBall(
      new Ball());
Ball ball =
      dog.getBAll());
```

```
Dog dog = new Dog();
dog.take(new Ball());
Ball ball =
      dog.give();
```

```
Dog dog = new Dog();
Dog.setWeight("23Kg");
```

```
Dog dog =
      new Dog("23Kg");
```

Example

Wrong:

```
MyThing[] things =
    thingManager.getThingList();
for (int i = 0; i < things.length; i++) {
    MyThing thing = things[i];
    if (thing.getName().equals(thingName))
        return thingManager.delete(thing);
}
```

Right:

```
    return thingManager.deleteThingNamed
        (thingName);
```

Fail-Fast

- Compile time checking is best
  - Contrast this signature:

```
void assignCustomerToSalesman (
                long customerId,
                long salesmanId);
```

  - with

```
void assignCustomerToSalesman (
                Customer c,
                Salesman s);
```

Fail Fast

- Bad:

  In Java a Properties class maps String to String

put method does not make this check

save method does this check

Compare

```
void setAmount(int
  value, String
  currency) {
    this.value =
           value;
    this.currency =
           currency;
}
```

```
void setAmount(
       Money value) {
    this.value=value;
}
```


The code on the right

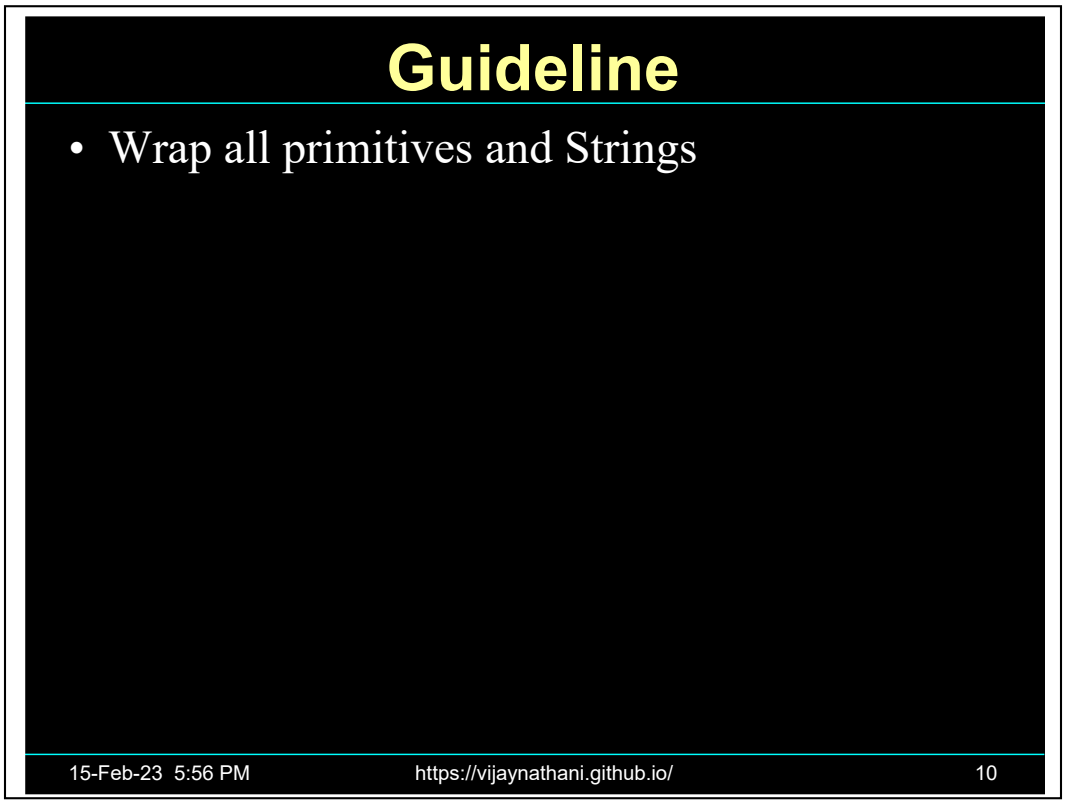
- Is flexible because any runner can be used
- Less prone to error, because the caller does not have to worry about exceptions

Improve

```
setOuterBounds ( x, y,
          width, height);
setInnerBounds ( x+2, y+2,
          width-4, height-4);
```

- Solution: Use Parameter Object

```
setOuterBounds (bounds);
setInnerBounds (bounds.expand
                    (-2));
```



An int on its own is just a scalar with no meaning. When a method takes an int as a parameter, the method name needs to do all the work of expressing the intent. If the same method takes an hour as a parameter, it's much easier to see what's happening. Small objects like this can make programs more maintainable, since it isn't possible to pass a year to a method that takes an hour parameter. With a primitive variable, the compiler can't help you write semantically correct programs. With an object, even a small one, you are giving both the compiler and the programmer additional information about what the value is and why it is being used.

Small objects such as hour or money also give you an obvious place to put behavior that otherwise would have been littered around other classes. This becomes especially true when you apply the rule relating to getters and setters and only the small object can access the value.

Direct access to Variables

- Compare

  ```
  doorRegister=1;
  ```

  with

  ```
  openDoor();
  ```

  with

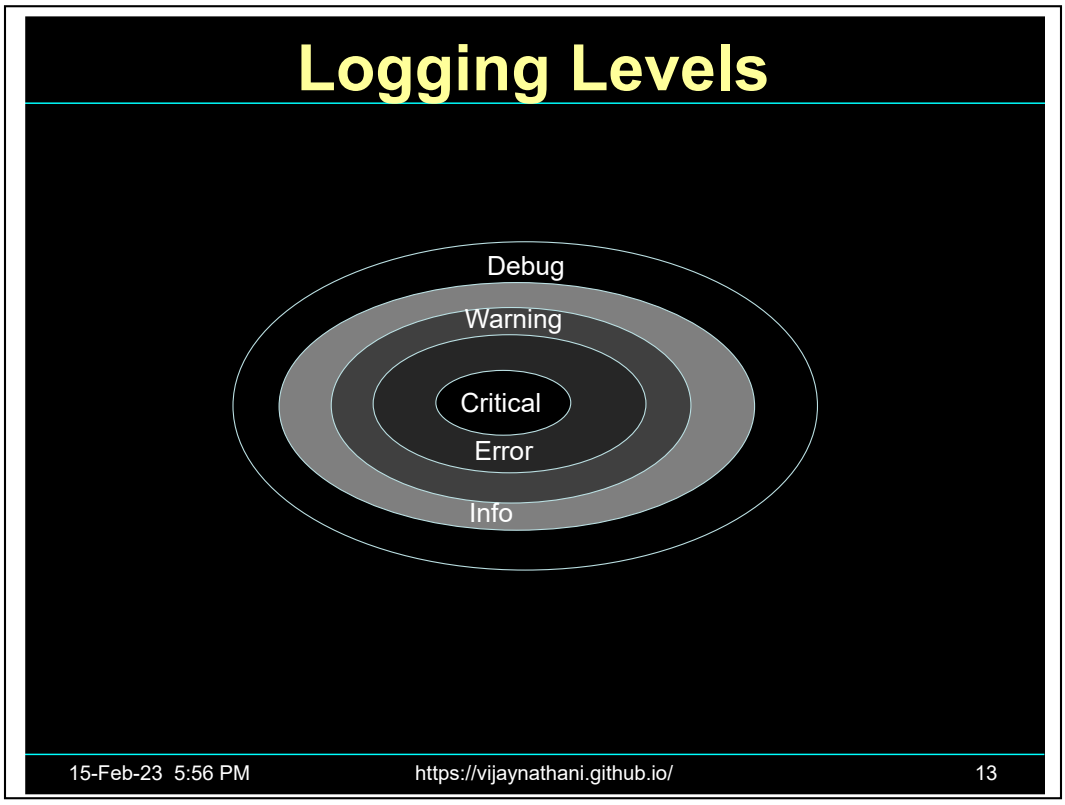
  ```
  door.open();
  ```


The last one is the best. It uses objects and functions.

Logging on hard disk

- Logging should be optimal
  - Too Much is bad
  - Too Little is also bad
- Log levels
  - Debug during development
  - Info, Warning, Error and Critical during Production.

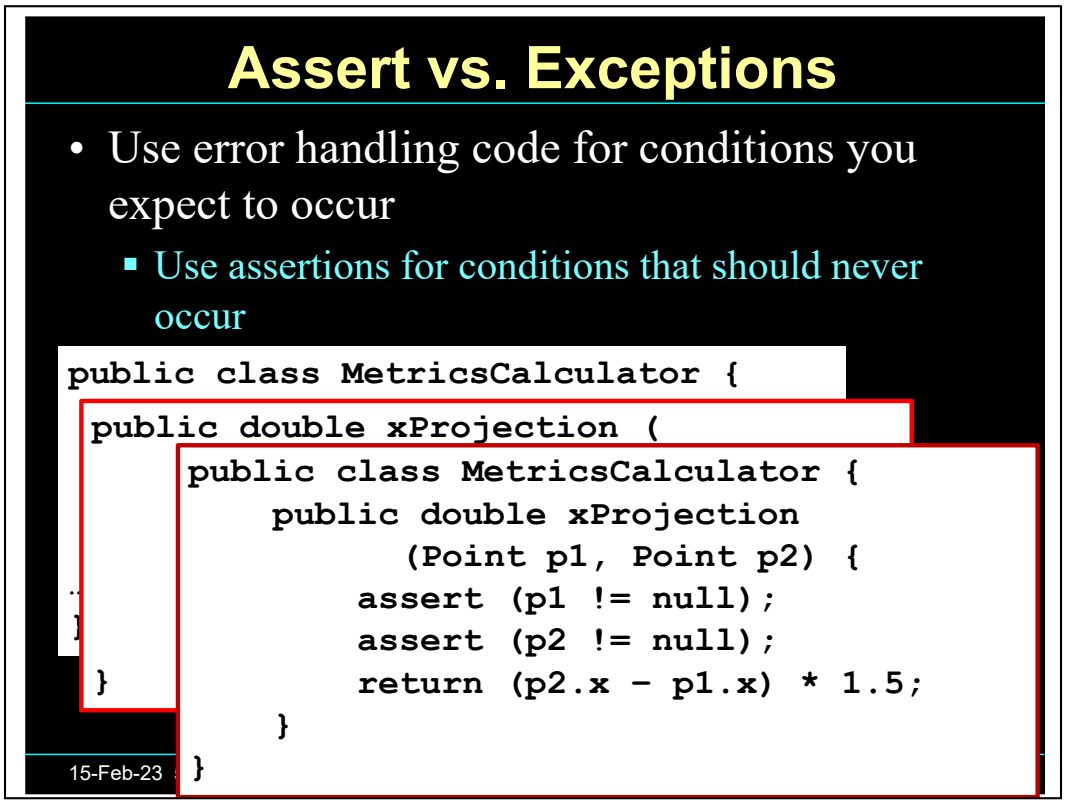
Reference: http://googletesting.blogspot.in/2013/06/optimal-logging.html



Code for logging

using boost library in C++: https://www.sentinelone.com/blog/getting-started-quickly-cplusplus-logging/

using log4j library in Java: https://www.edureka.co/blog/logger-in-java



Exceptions should be exceptional

Consider logging

•Error rates

•System metrics

==========

If a global debug flag is set, then do a lot more error checking / write detailed log. This flag will usually be false in production mode.

Log errors for technical

support personnel.

Use Assertions to document the

pre-conditions and post-conditions.

==========

Have alerts for systematic failure. For 24 X 7 no single point of failure (SPOF)

Does the performance graceful degrade

•under high loads

•With resource failure

For scaling scale-out is better than scale-up.

Interface Segregation Principle

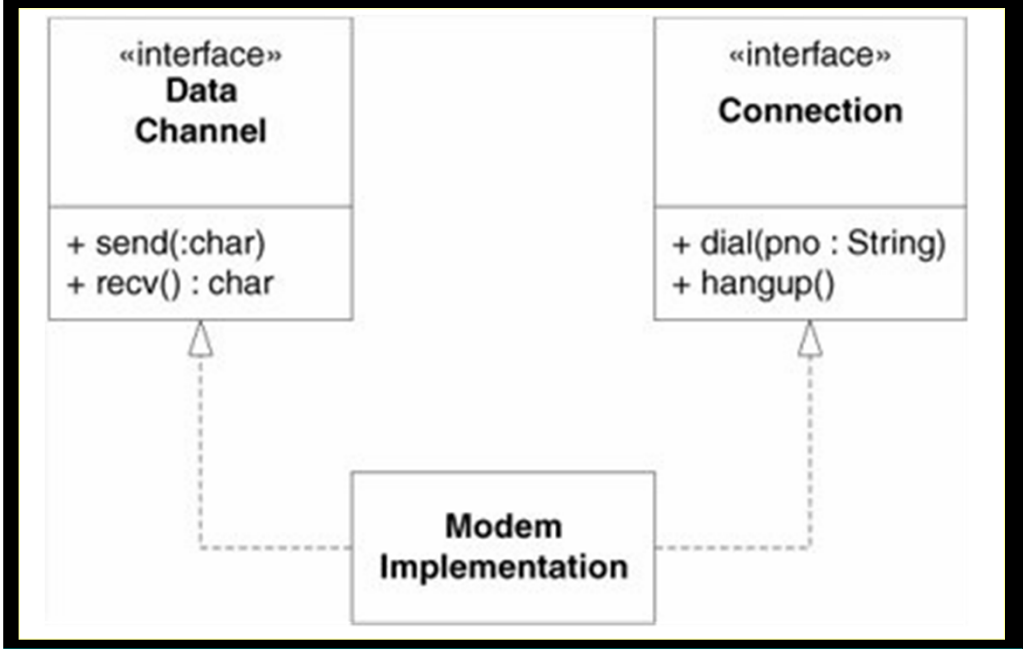
- Interfaces should be as fine-grained as possible.
  - Any problem:

```
public interface Modem {
  public void dial(String pno);
  public void hangup();
  public void send(Char c);
  public char recv();
}
```

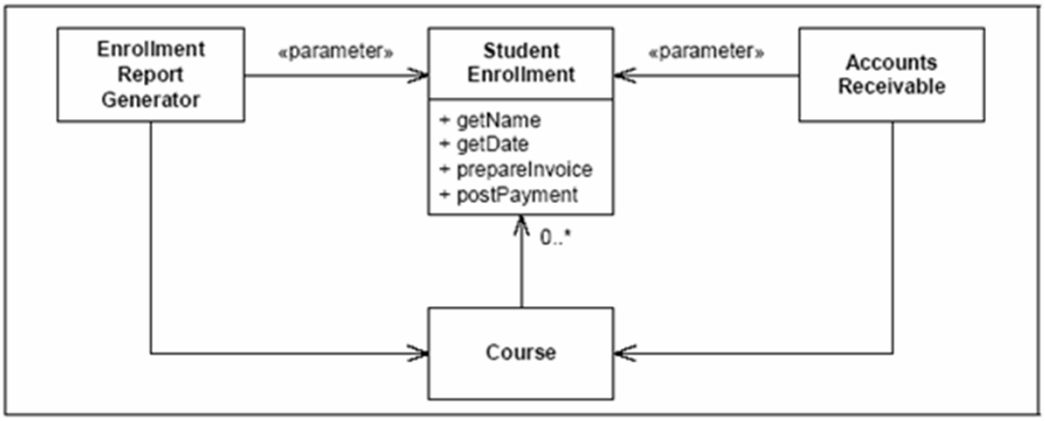

Clients should not be forced to depend upon the interfaces that they do no use. – Robert Martin

If a class implements an interface with multiple methods, but in one of the methods throws notSupportedException, then this principle is violated.

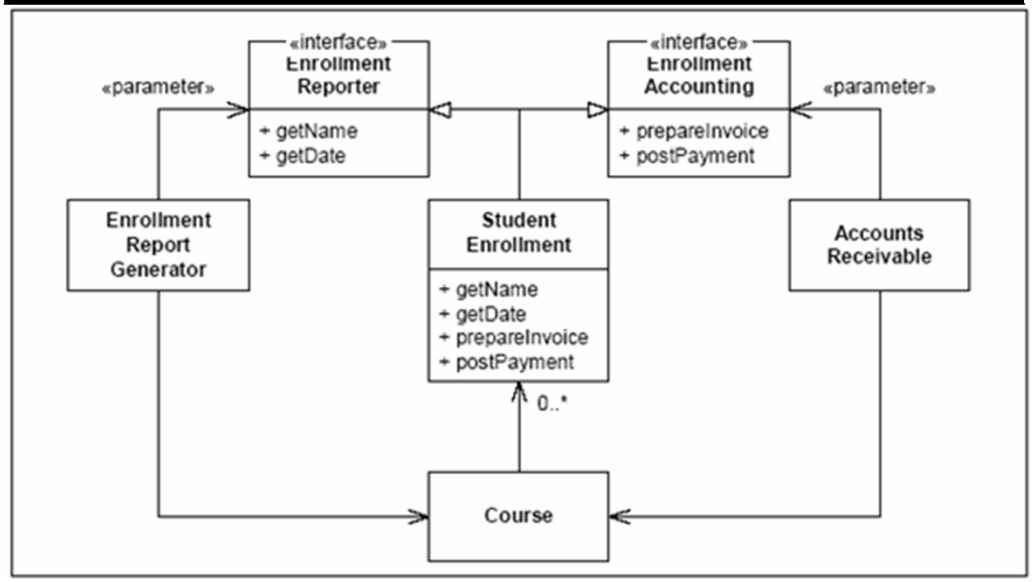
ISP implemented



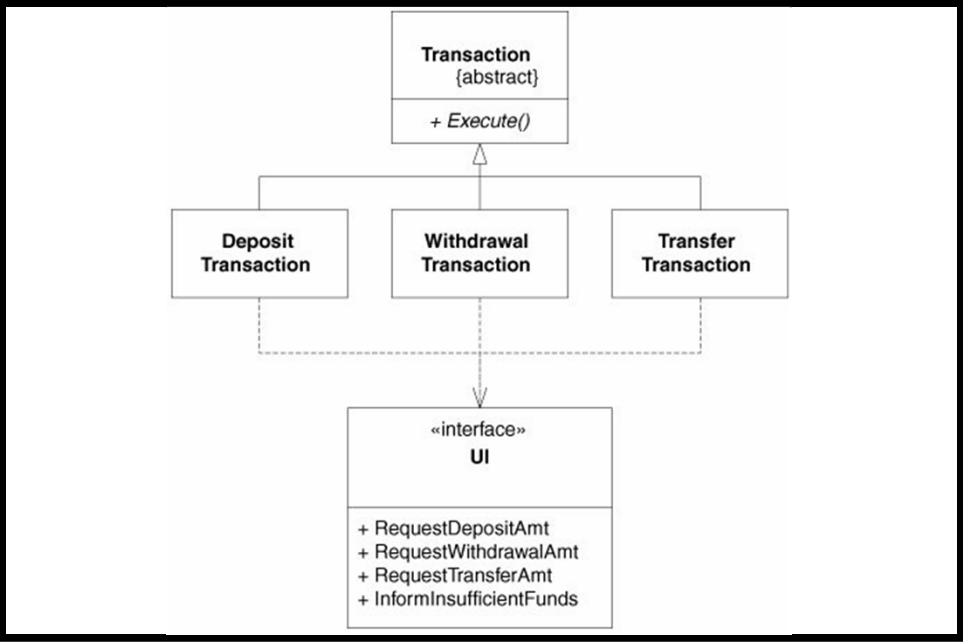
ISP violated



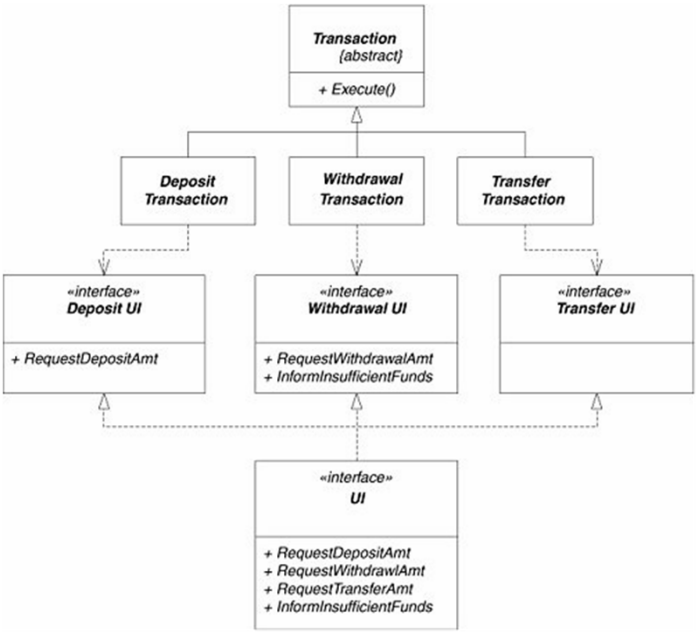
ISP implemented

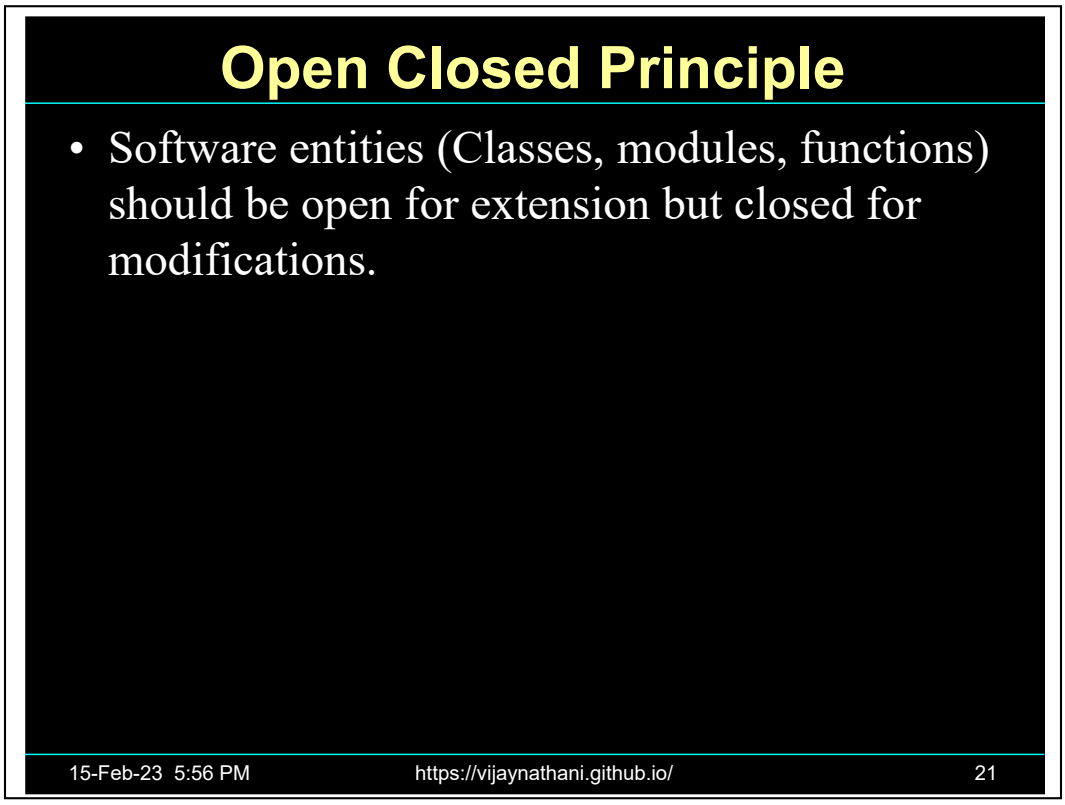


ISP violated



ISP implemented





It should be possible to change the environment of a class without changing the class.

A class should be closed for modification, but open for extension. When you change a class, there is always a risk that you will break something. But if instead of modifying the class you extend it with a sub-class, that's a less risky change.

Guidelines

- Keep things that vary separately from things that are common.

Q45 – LoanHandler

Q44 – FILE1, DATABASE1

Q39 - Scheduler

Q83 – Cooker

Q84 – ChooseFontDialog