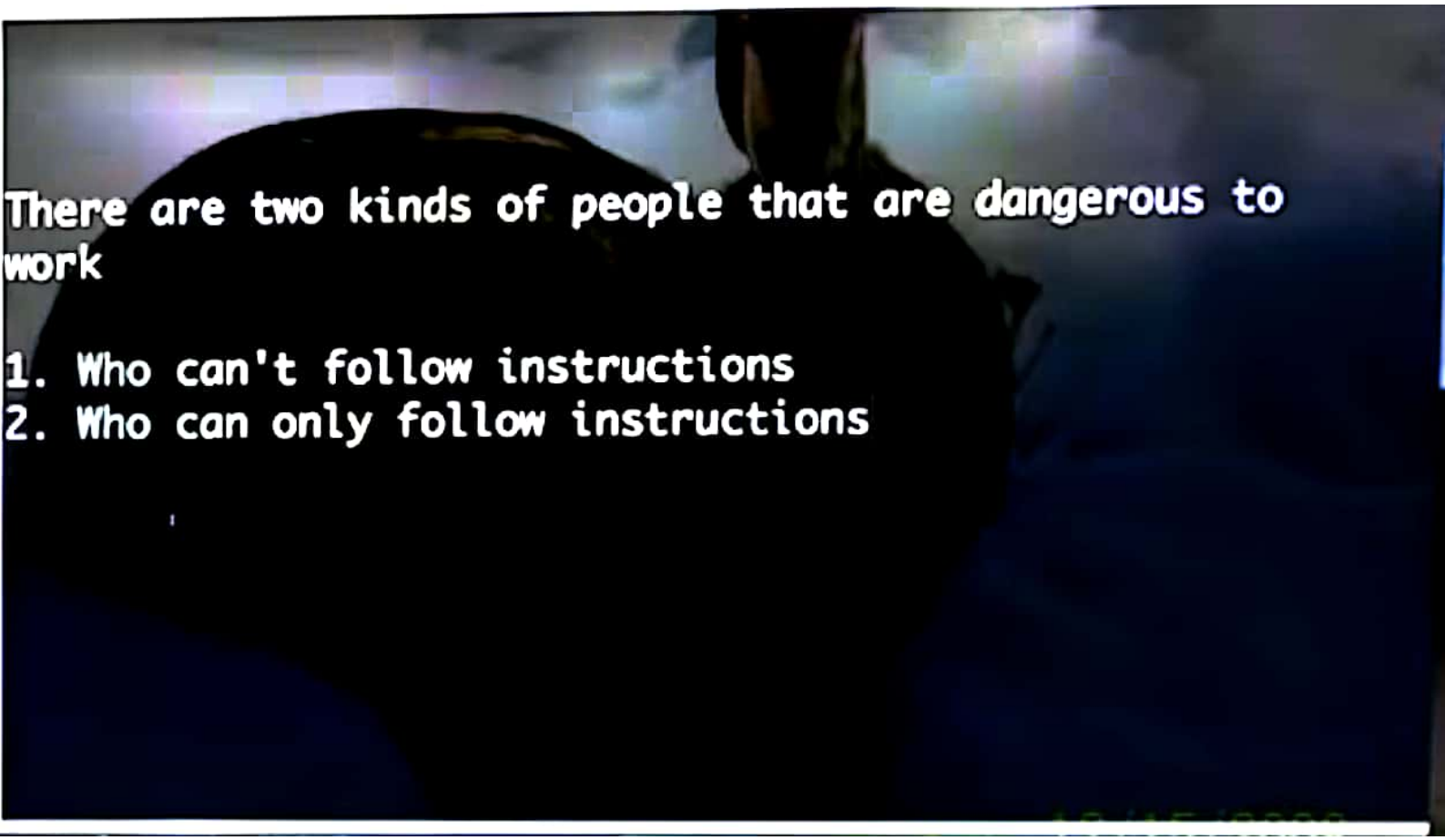
```
Core Design Principles for Software Developers
Venkat Subramaniam
Email: venkats@agiledeveloper.com
Twitter: @venkat_s
http://www.agiledeveloper.com download link

- What's a good design?
- How to evaluate quality of design?
- How to create good design?
- Keep it simple
- Complexity
  - inherent and accidental
- Think YAGNI
- Cohesion
- Coupling
- High cohesion and low coupling
- Dealing with coupling
- Keep it DRY
- Focus on Single Responsibility
```

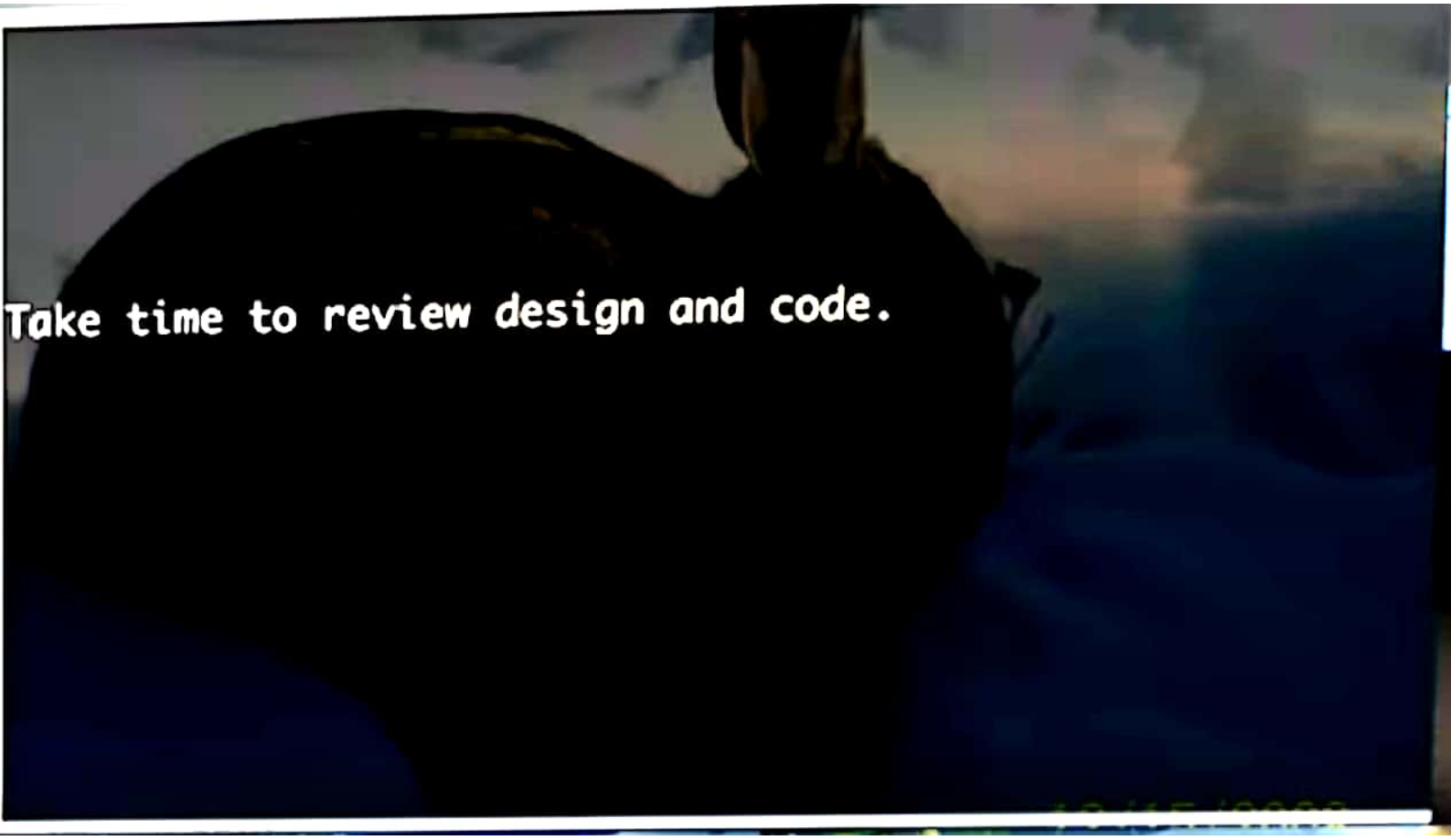
Almost impossible to get it right the first time.

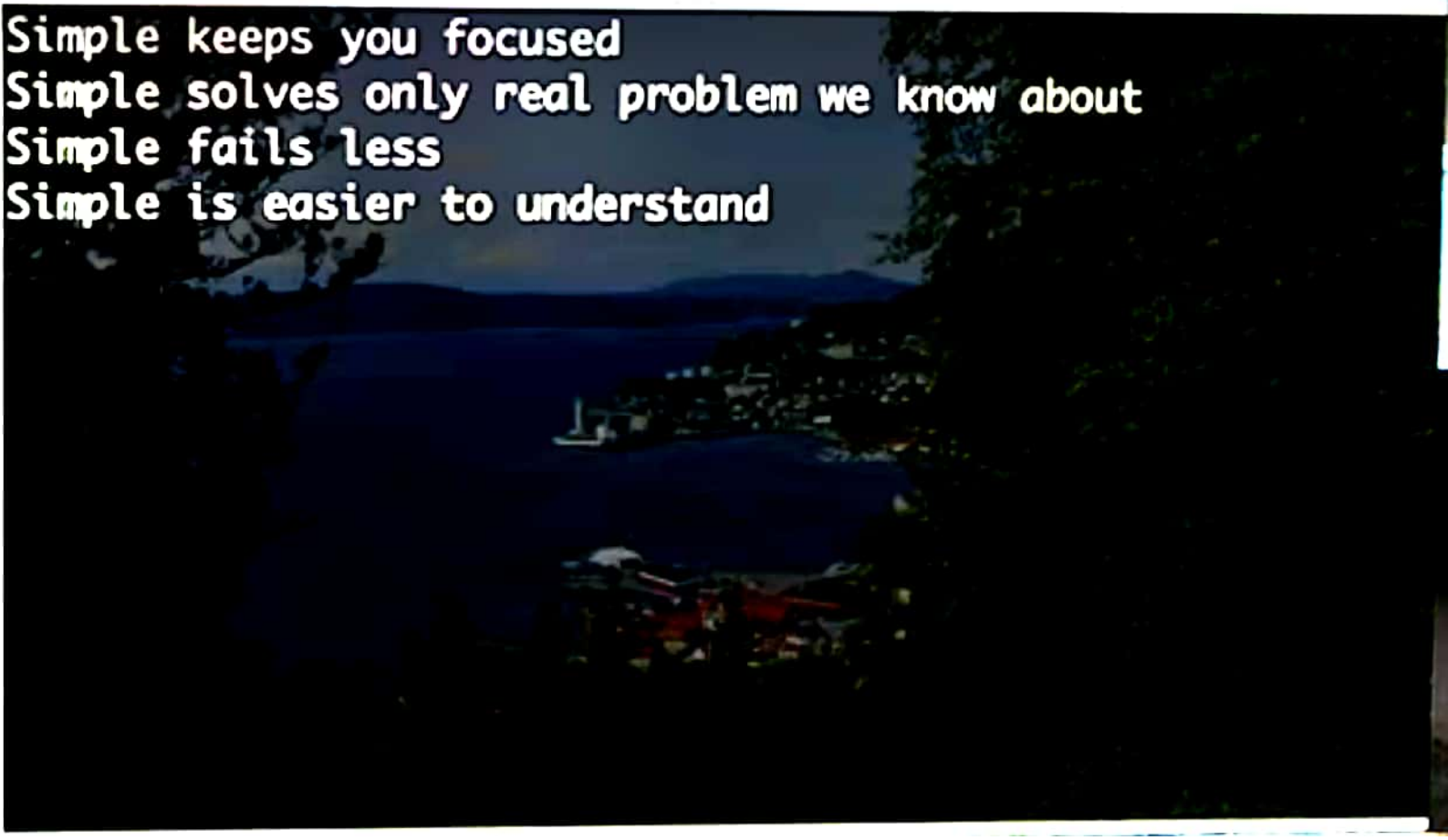
Software is never written, it is always rewritten.

To create good design first step is let go of the ego.

Be unemotional.

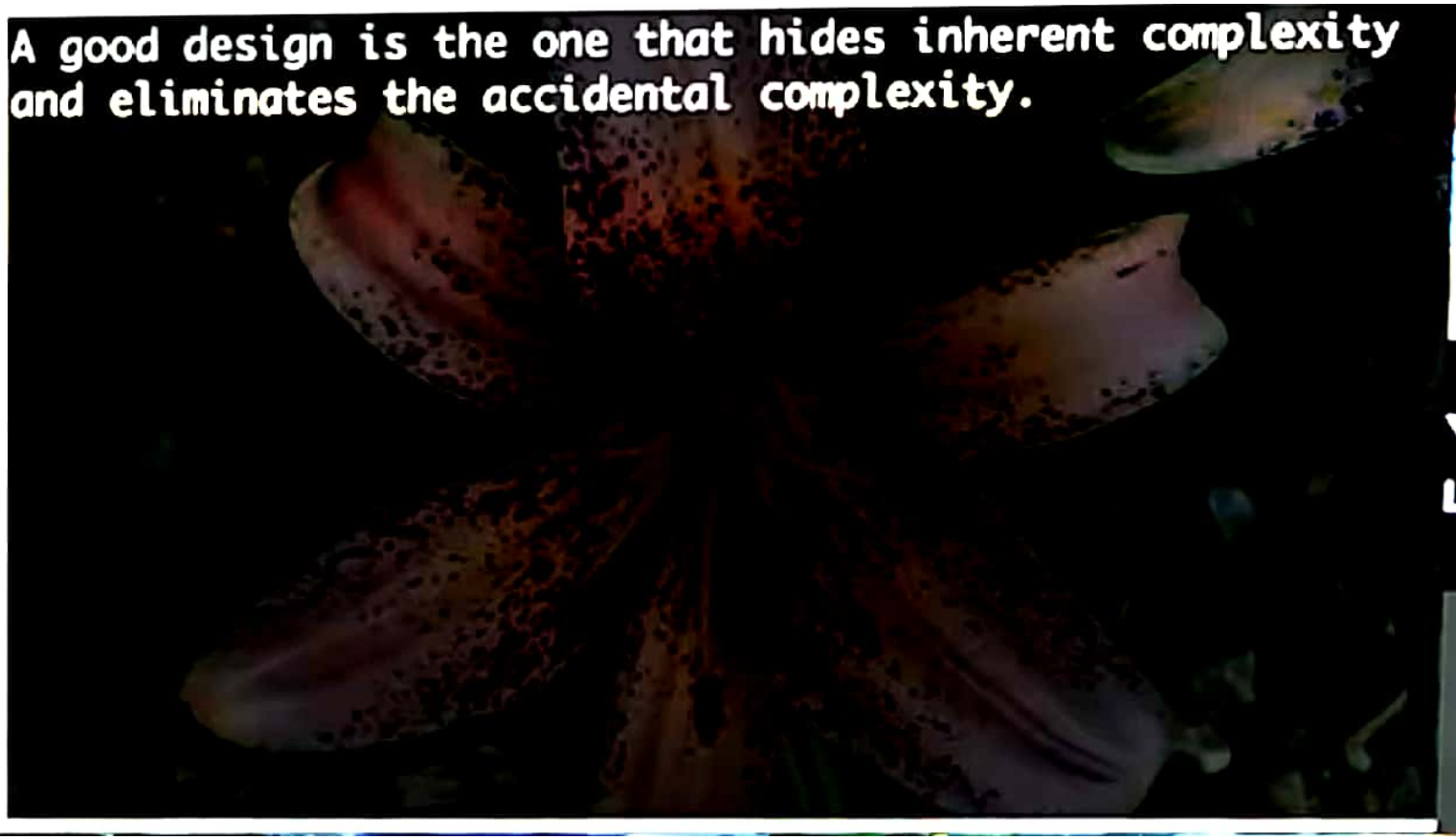






Simple keeps you focused
Simple solves only real problem we know about
Simple fails less
Simple is easier to understand

Simple is not necessarily familiar



When should I implement something?

How much do you know?

Cost of implementing

Now		Later	
$N	>	$L	- postpone
$N	==	$L	- postpone
$N	<	$L	

- how probable? high

How much do you know?

Cost of implementing

Now		Later	
$N	>	$L	- postpone
$N	=	$L	- postpone
$N	<	$L	

- how probable? high - do it now
  low - postpone

Why don't we postone?

We are scared

Manual testing
  end lets change - are you crazy

Automated testing
  end lets change - give it try

```
Why don't we postone?

We are scared

Manual testing
  end lets change - are you crazy

Automated testing
  end lets change - give it try

  If you want to postone we need to good automated testing.
```

We want software to change, but not too expensive

If a code is cohesive, it has to change less frequently

coupling what you depend on.

Worst form of coupling - Inheritance

Try to see if you can remove coupling.

"knock out before you mock out"

Can't remove all the depencies.

1. get rid of it
2. Make

Try to see if you can remove coupling.

"knock out before you mock out"

Can't remove all the depencies.

1. get rid of it
2. Make it loose instead of tight

Depending on a class is tight coupling
Depending on an interface is loose coupling

Use caution.

```
DRY

Don't Repeat yourself

Don't duplicate -code and also effort

Every piece of knowledge in a system should have
a single unambiguous authorative represention.

It reduces the cost of development

Why should we care?

The future you will thank you.
```

```
//comment...
...
...
...

//comment...
...

...
...



//comment...
...
```

```
Long methods are bad:
hard to test
hard to read
hard to remember
obscured business rules
hard to reuse
leads to duplication
many reasons to change
can't be optimized by anything
lot of variables and ...
not developers friendly
mixed levels
low cohesion
high coupling
obsolete comments
```

SLAP

Single Level of Abstraction

Don't comment what, instead comment why

A good code is like a joke

OCP

Software module should be open for extension but closed from modification.

Abstraction and polymorphism are the key to make this happen.

OCP

Software module should be open for extension but closed from modification.

Abstraction and polymorphism are the key to make this happen.

two options - to make an enhancement
1. change existing code
2. add a small new module of code

Who can make code extensible?

We need to know software and domain.

There are three kinds of people we work with:

1. know domain really well, knows no software
2. know no domain, know software really well
3. know domain really well and know software really well

Inheritance overused

Inheritance should be used only for substitutability.

If an object of B should be used anywhere an object of A is used then use inheritance.

If an object of B should use an object of A, then use composition / delegation.

Inheritance demands more from a developer than composition or delegation does.

If an object of B should be used anywhere an object of A is used then use inheritance.

If an object of B should use an object of A, then use composition / delegation.

Inheritance demands more from a developer than composition or delegation does.

Services of the derived class should require no more and promise no less than the corresponding services of the base class.

Inheritance demands more from a developer than composition or delegation does.

Services of the derived class should require no more and promise no less than the corresponding services of the base class.

Why?

The user of a base class should be able to use an instance of a derived class without knowing the difference.

Good

public vs protected in base vs. derived

derived function can't throw any new checked exception not thrown by the base (unless the new exception extends the old one...)

Collection of derived does not extend from collection of base

Bad

```java
class A {
  public void f1() {}
  public int f2() { return 0; }
}

class B {
  private final A _a = new A();
  //should have f1, f2 (same as in A) and f3

  public void f3() {}

  public void f1() {
    _a.f1();
  }

  public int f2() {
    return _a.f2();
  }
}

//If we use inheritance in this case we violate LSP.
//we are not using inheritance.

//But we're violating two principles:
//

public class Sample {
  public static void main(String[] args) {
    System.out.println("OK");
```

```
  public void f3() {}

  public void f1a() {
    _a.f1a();
  }

  public int f2() {
    return _a.f2();
  }
}

//If we use inheritance in this case we violate LSP.
//we are not using inheritance.

//But we're violating two principles:
//DRY
//OCP

//Should I violate LSP or should I violate DRY and OCP?

public class Sample {
  public static void main(String[] args) {
    System.out.println("OK");
  }
}
```

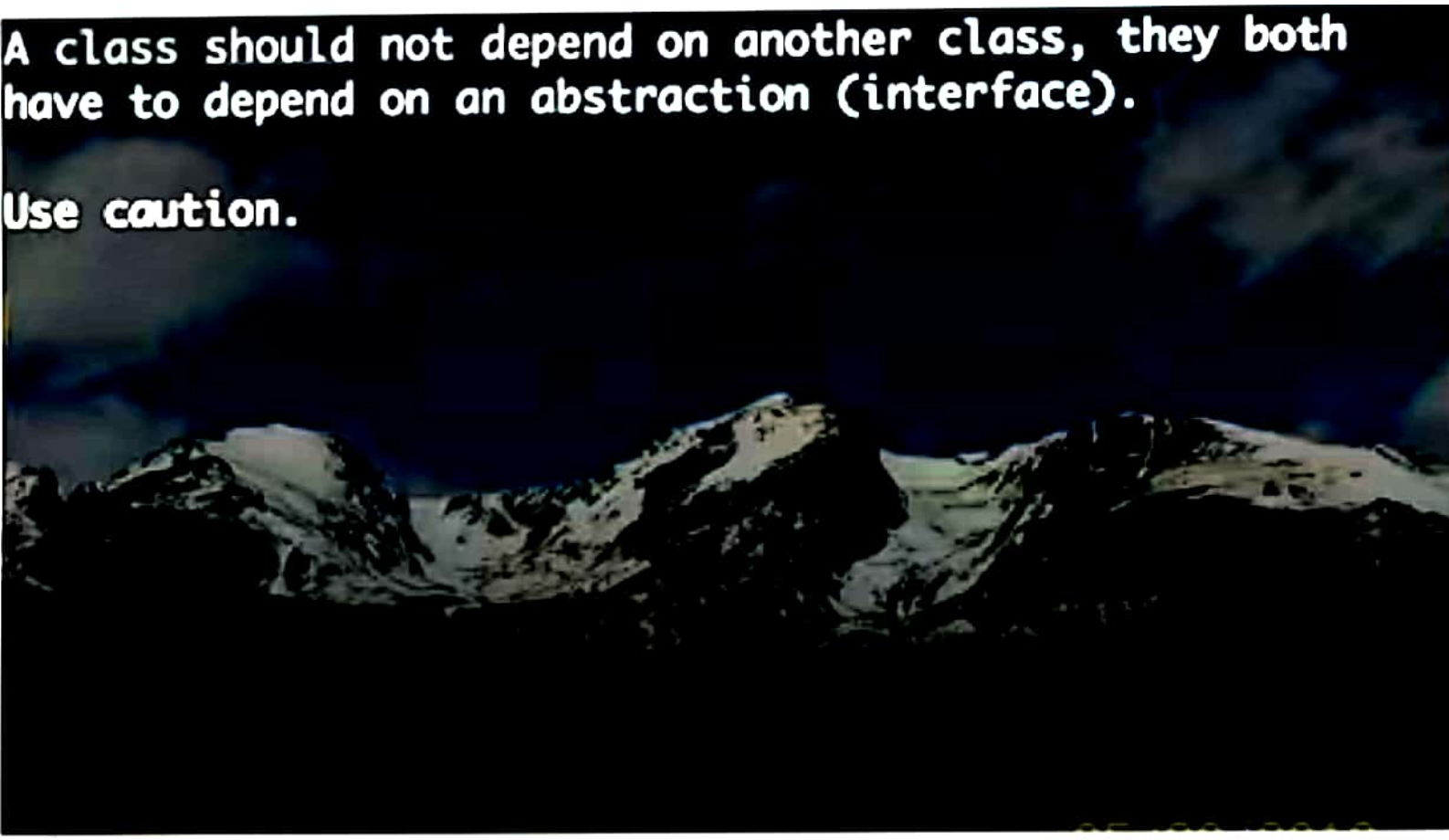
Use composition or delegation instead of inheritance unless you want substitutability.

```
class Worker {
  def work() { println 'working...' }
}

class Manager {
  @Delegate Worker victim = new Worker()
}

def bob = new Manager()
bob.work()
```

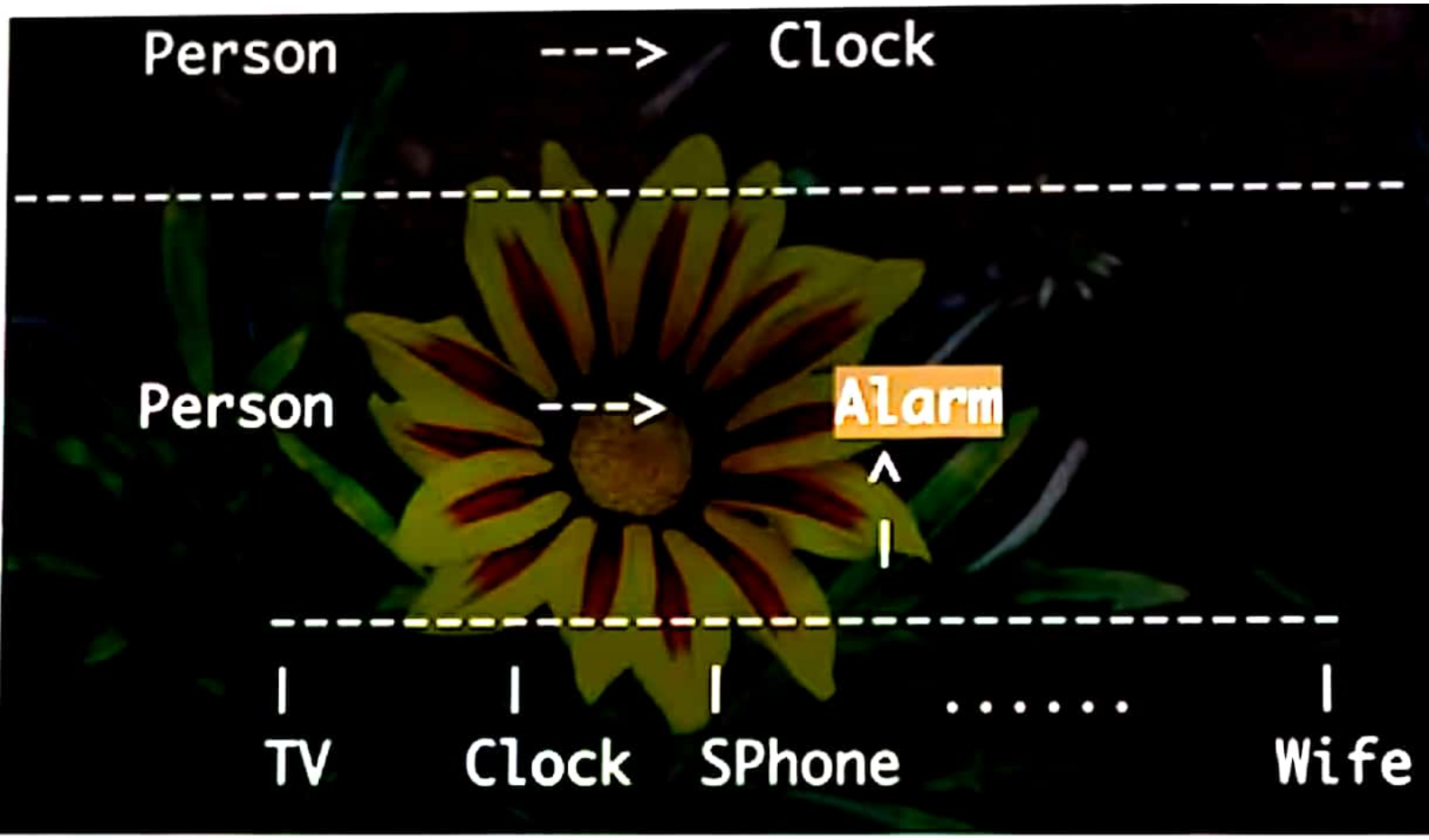
```
working...
```

A class should not depend on another class, they both have to depend on an abstraction (interface).

Use caution.



```
Venkat      --->   Clock

-------------------------------------

Venkat      --->     Alarm
                      ^
                      |
        ------------------------------
        |      |      |      ......
        TV     Comp   SPhone
```



```
Person      --->   Clock

-----------------------------------------------------


Person      --->    Alarm
(FunctionalInterface)

Library of lambda expression that conform
to the Alarm fuctional interface
```

```
Clock
  set time
  get time
  set alarm
  get alarm
  set radio
  listen radio

class Clock implements TimePiece, Alarm,
Radio {

}
```

```
4   set alarm
5   get alarm
6   set radio
7   listen radio
8
9 class Clock implements TimePiece, Alarm,
  Radio {
10
11 }
12
13 User 1: TimePiece send a Clock
14
```

```
class Clock implements TimePiece, Alarm, Radio {

}

User 1: TimePiece send a Clock

User 2: Alarm send a Clock

User 3: Radio send a Clock
```