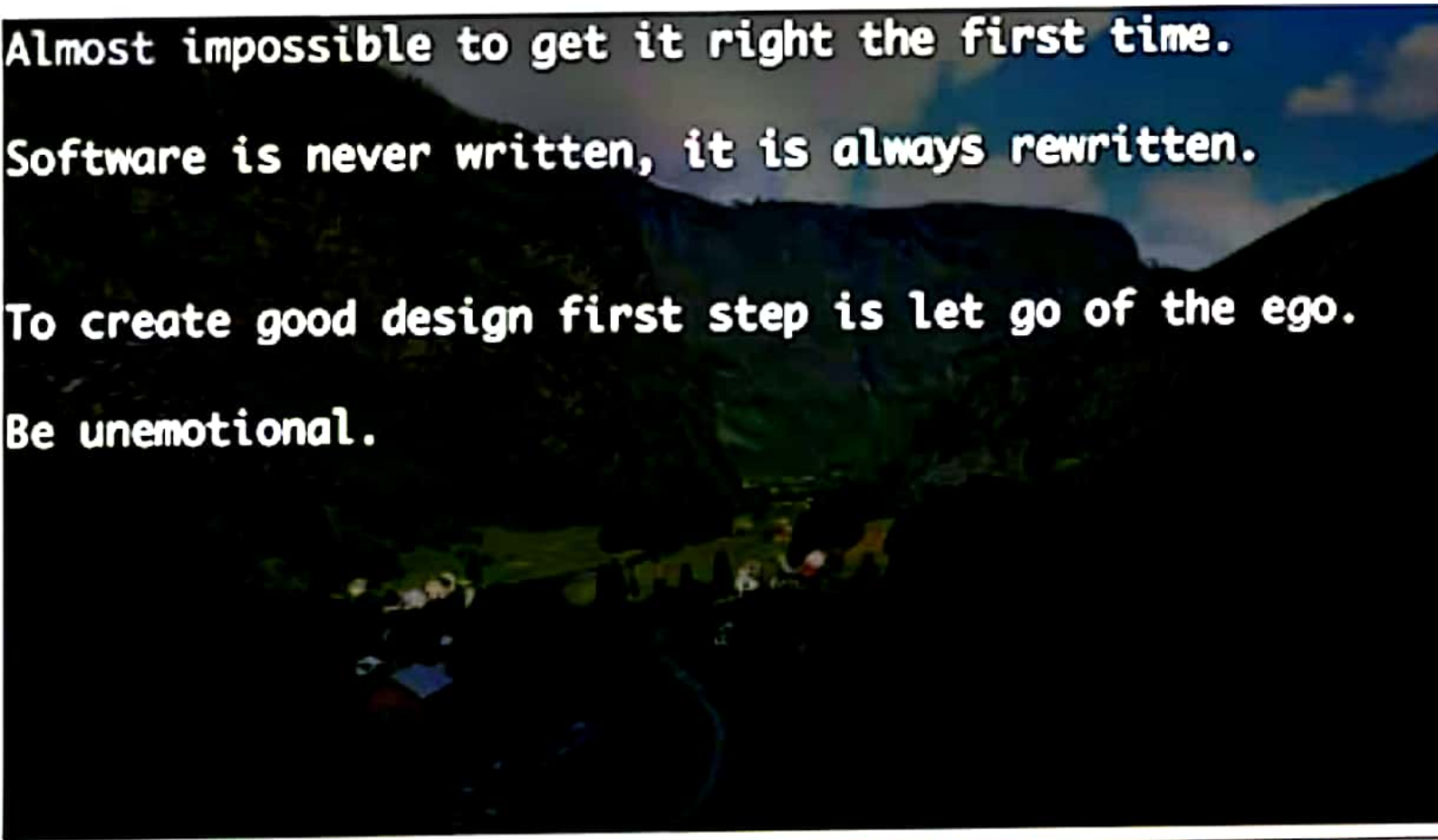


\*\*\*

- 1 Core Design Principles for Software Developers
- 2 Venkat Subramaniam
- 3 Email: [venkats@agiledeveloper.com](mailto:venkats@agiledeveloper.com)
- 4 Twitter: @venkat\_s
- 5 <http://www.agiledeveloper.com> download link
- 6
- 7 - What's a good design?
- 8 - How to evaluate quality of design?
- 9 - How to create good design?
- 10 - Keep it simple
- 11 - Complexity
- 12   - inherent and accidental
- 13 - Think YAGNI
- 14 - Cohesion
- 15 - Coupling
- 16 - High cohesion and low coupling
- 17 - Dealing with coupling
- 18 - Keep it DRY
- 19 - Focus on Single Responsibility

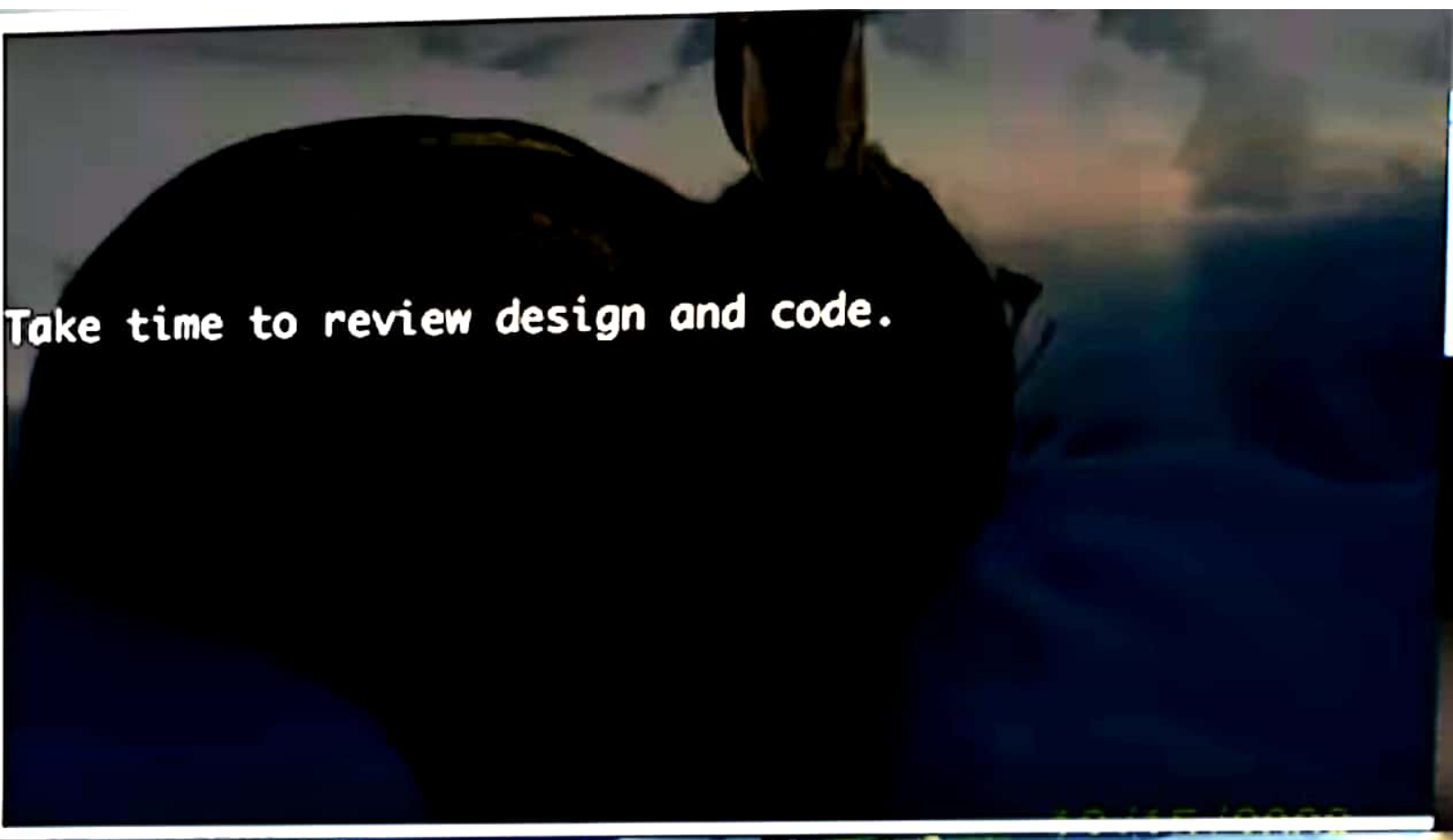
A dark, atmospheric landscape with a river and mountains under a cloudy sky. The scene is dimly lit, with the river reflecting the light from the sky. The mountains are silhouetted against the sky, and the overall mood is somber and contemplative.

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.

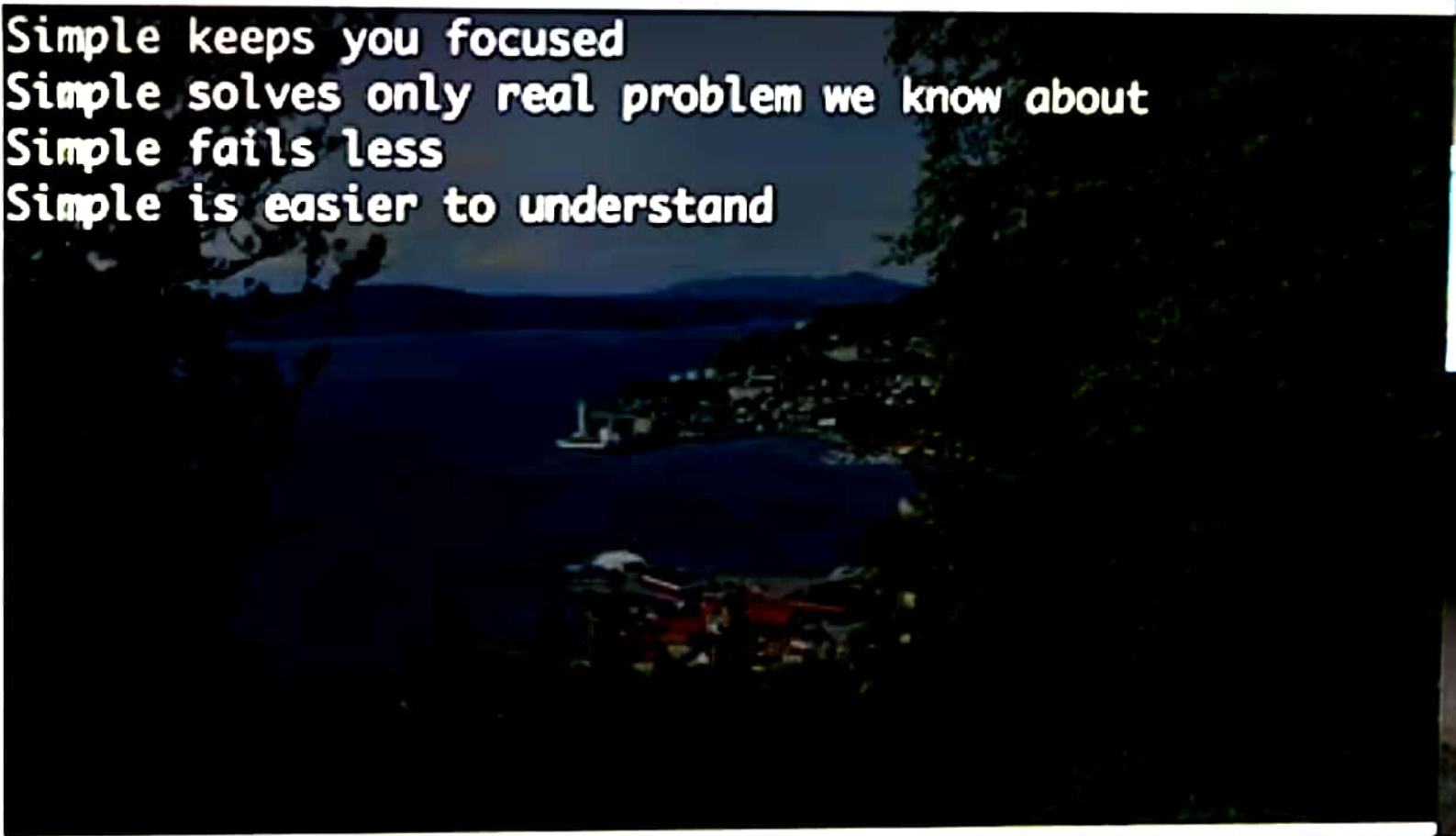


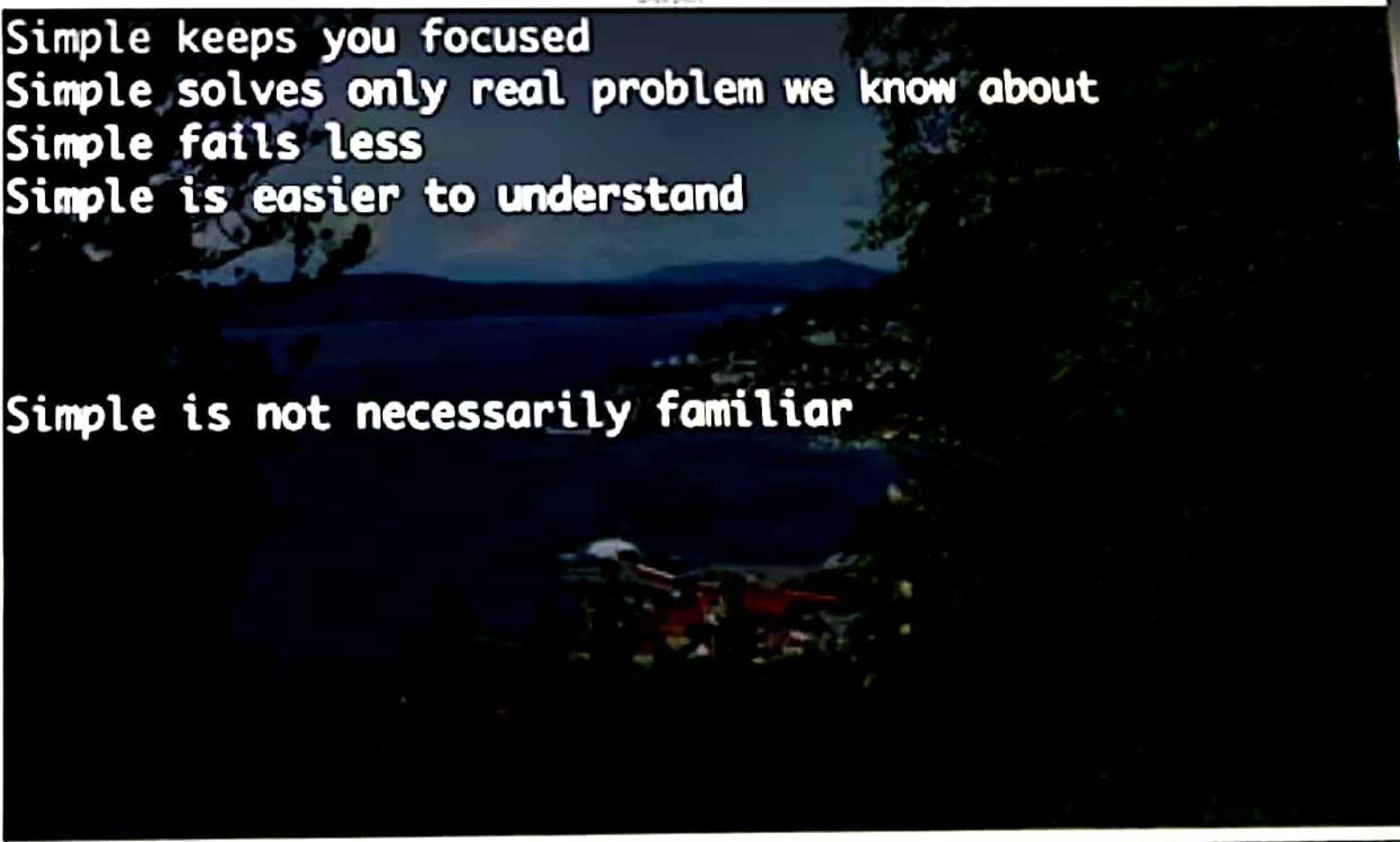
There are two kinds of people that are dangerous to work

1. Who can't follow instructions
2. Who can only follow instructions



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

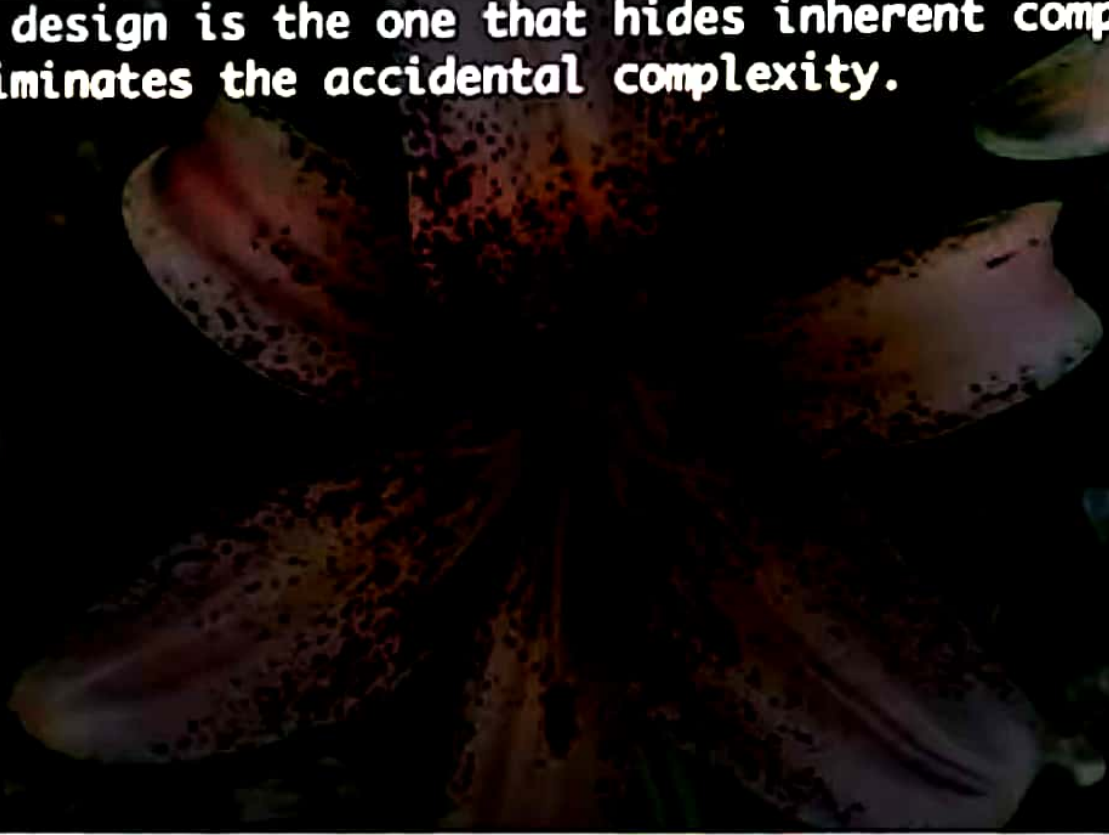




Simple keeps you focused  
Simple solves only real problem we know about  
Simple fails less  
Simple is easier to understand

Simple is not necessarily familiar

**A good design is the one that hides inherent complexity  
and eliminates the accidental complexity.**





When should I implement something?

How much do you know?

Cost of implementing

Now

\$N

>

Later

\$L

- postpone

\$N

=

\$L

- postpone

\$N

<

\$L

- how probable? high



How much do you know?

Cost of implementing

Now

\$N

\$N

\$N

Later

\$L

\$L

\$L

- postpone

- postpone

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

Why don't we postpone?

We are scared

Manual testing  
end lets change - are you crazy

Automated testing  
end lets change - give it try

1 Why don't we postpone?

2

3 We are scared

4

5 Manual testing

6 end lets change - are you crazy

7

8 Automated testing

9 end lets change - give it try

10

11 If you want to postpone 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 dependencies.

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 dependencies.

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.



1 DRY  
2  
3 Don't Repeat yourself  
4  
5 Don't duplicate -code and also effort  
6  
7 Every piece of knowledge in a system should have  
8 a single unambiguous authoritative representation.  
9  
10 It reduces the cost of development  
11  
12 Why should we care?  
13  
14 The future you will thank you.



Email: [venkats@agiledeveloper.com](mailto: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*
- ✓ *Long methods*
- ✓ *SLAP*

//comment...

...  
...  
...

//comment...

...  
...  
...

//comment...

...

\*\*\*

- 1 Long methods are bad:
- 2 hard to test
- 3 hard to read
- 4 hard to remember
- 5 obscured business rules
- 6 hard to reuse
- 7 leads to duplication
- 8 many reasons to change
- 9 can't be optimized by anything
- 10 lot of variables and ...
- 11 not developers friendly
- 12 mixed levels
- 13 low cohesion
- 14 high coupling
- 15 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

- ...
- 1
- 2
- 3

# Extensible



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

\*\*\*  
1 Inheritance overused

2

3 Inheritance should be used only for substitutability.

4

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

6

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

8

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

10

11

12

...

4

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

6

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

8

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

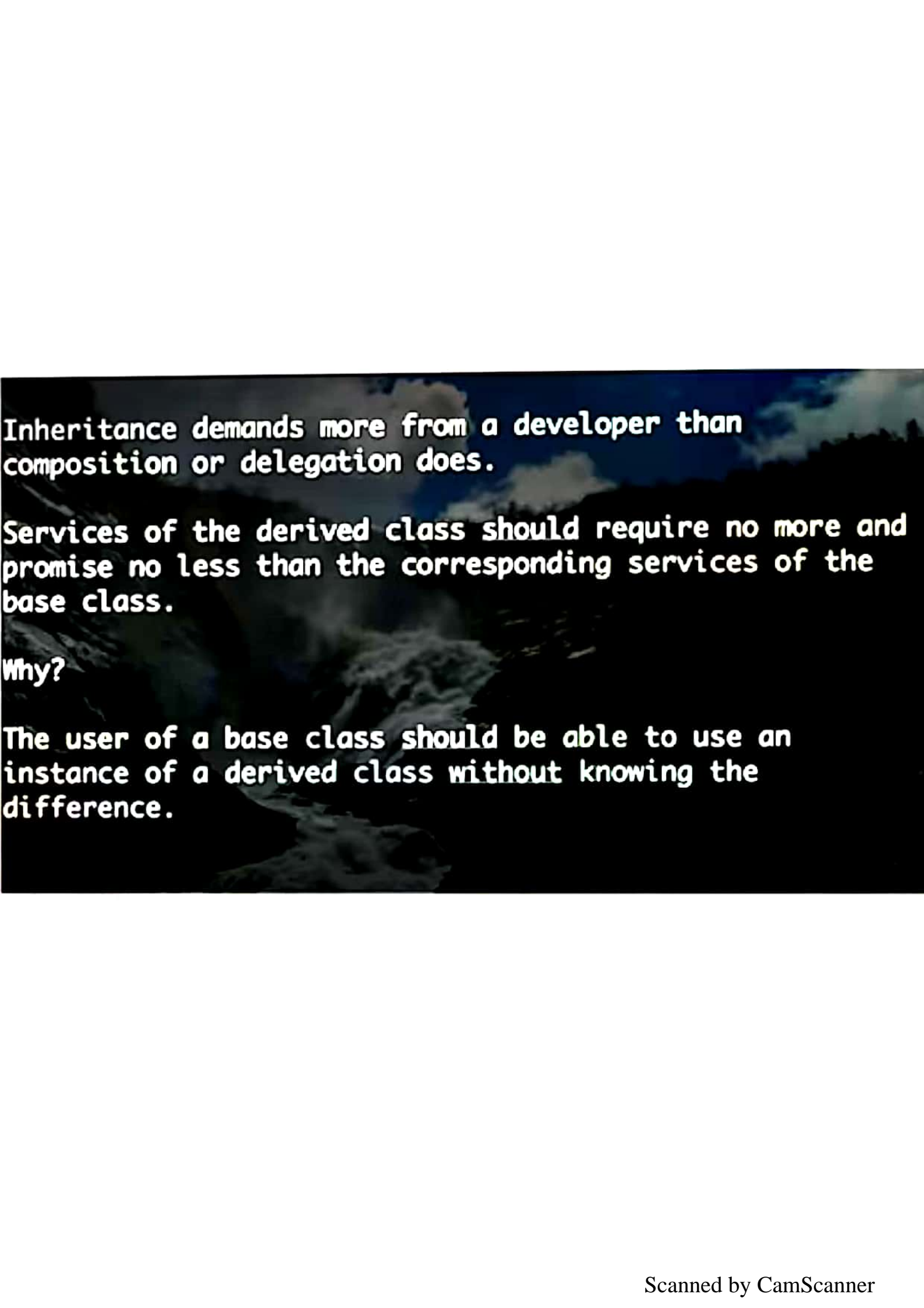
10

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

12

13





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**

```

1 class A {
2     public void f1() {}
3     public int f2() { return 0; }
4 }
5
6 class B {
7     private final A _a = new A();
8     //should have f1, f2 (same as in A) and f3
9
10    public void f3() {}
11
12    public void f1() {
13        _a.f1();
14    }
15
16    public int f2() {
17        return _a.f2();
18    }
19 }
20
21 //If we use inheritance in this case we violate LSP.
22 //we are not using inheritance.
23
24 //But we're violating two principles:
25 //1
26
27 public class Sample {
28     public static void main(String[] args) {
29         System.out.println("OK");
30     }
31 }

```

```

10 public void f3() {}
11
12 public void f1a() {
13     _a.f1a();
14 }
15
16 public int f2() {
17     return _a.f2();
18 }
19 }
20
21 //If we use inheritance in this case we violate LSP.
22 //we are not using inheritance.
23
24 //But we're violating two principles:
25 //DRY
26 //OCP
27
28 //Should I violate LSP or should I violate DRY and OCP?
29 |
30
31 public class Sample {
32     public static void main(String[] args) {
33         System.out.println("OK");
34     }
35 }
36

```

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

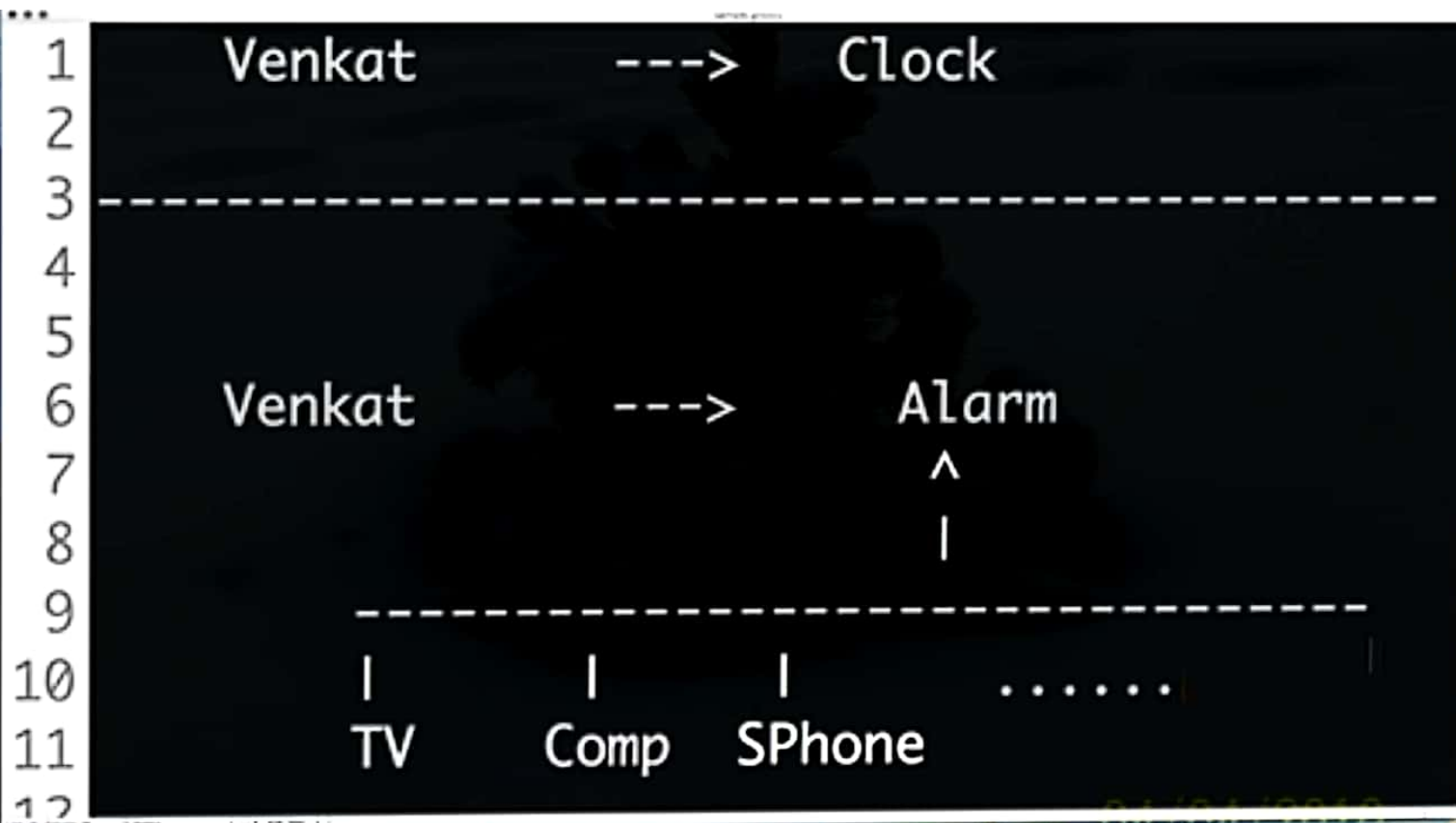
```
class Worker {  
  def work() { println 'working...' }  
}  
  
class Manager {  
  @Delegate Worker victim = new Worker()  
}  
  
def bob = new Manager()  
bob.work()
```

working...

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

Use caution.





Person

---->

Clock

Person

---->

Alarm

^

|

|

TV

|

Clock

|

SPhone

.....

|

Wife

Person

--->

Clock

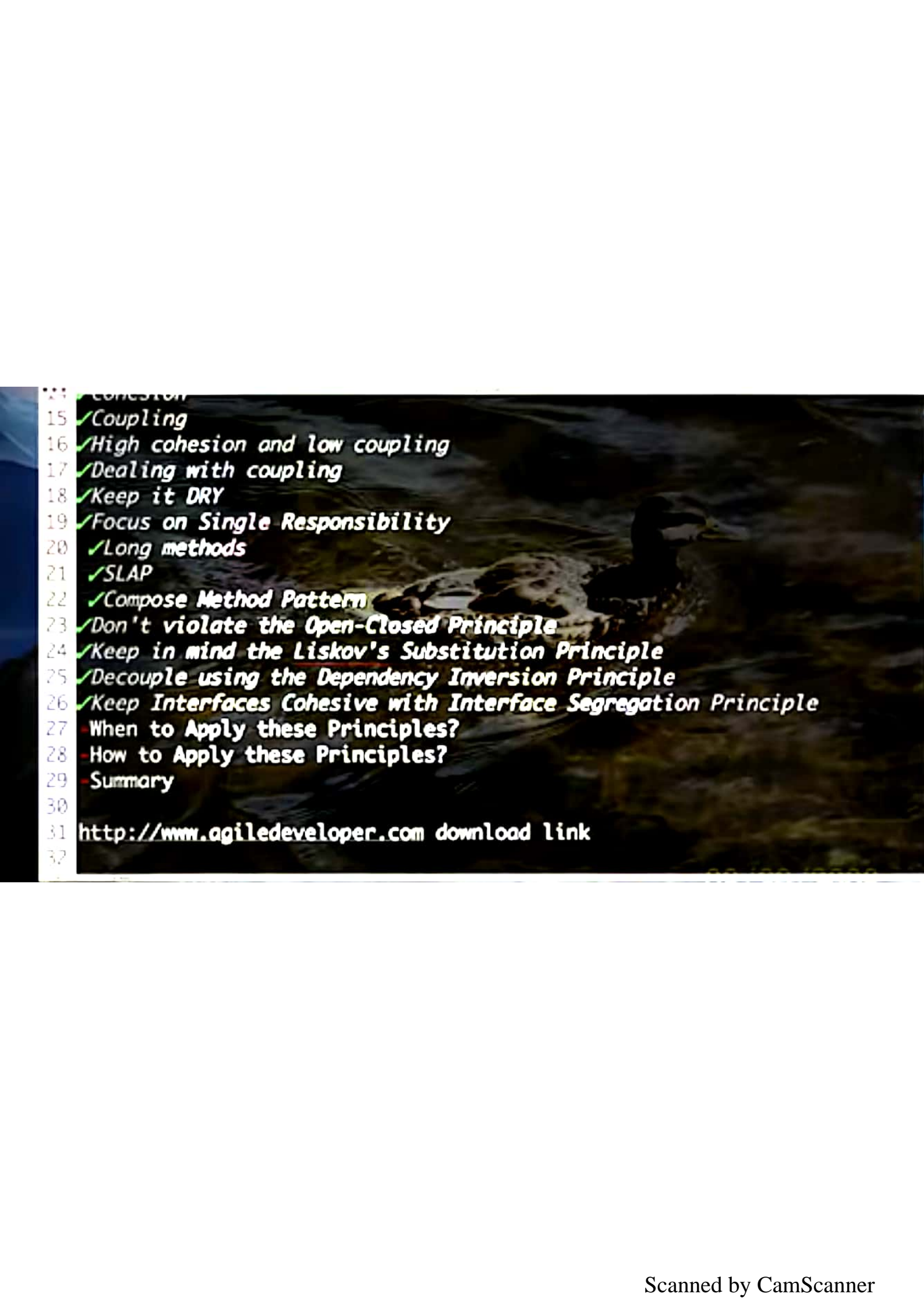
Person

--->

Alarm

(FunctionalInterface)

Library of lambda expression that conform  
to the Alarm functional interface



14 ✓ Cohesion  
15 ✓ Coupling  
16 ✓ High cohesion and low coupling  
17 ✓ Dealing with coupling  
18 ✓ Keep it DRY  
19 ✓ Focus on Single Responsibility  
20 ✓ Long methods  
21 ✓ SLAP  
22 ✓ Compose Method Pattern  
23 ✓ Don't violate the Open-Closed Principle  
24 ✓ Keep in mind the Liskov's Substitution Principle  
25 ✓ Decouple using the Dependency Inversion Principle  
26 ✓ Keep Interfaces Cohesive with Interface Segregation Principle  
27 - When to Apply these Principles?  
28 - How to Apply these Principles?  
29 - Summary  
30  
31 <http://www.agiledeveloper.com> download link  
32

```
1 Clock
2   set time
3   get time
4   set alarm
5   get alarm
6   set radio
7   listen radio
8
9 class Clock implements TimePiece, Alarm,
  · Radio {
10
11 }
```



```
...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
```



```
... 8
9 class Clock implements TimePiece, Alarm,
  · Radio {
10
11 }
12
13 User 1: TimePiece send a Clock
14
15 User 2: Alarm send a Clock
16
17 User 3: Radio send a Clock
18
```

1 DRY  
2 YAGNI  
3 -  
4 SRP  
5 OCP  
6 LSP  
7 ISP  
8 DIP

