Good

- Immutable (constants)
- Boy Scout rule
- DRY principle
- 3A Pattern
- SRP principle
- Principle of Least Astonishment
- Cyclomatic Complexity < 10
- Fun size < 10 lines
- Comments are failure
- Readable checkin labels
- NULL Pattern
- Write Test Before Bug Fix
- Specification Pattern
    - Separate domain logic (fun) from domain rules (isRule1)
- Seperation of Concerns
- Boundary control Entity Pattern
- Nesting {if < 4 , switch,while,do, for <3}
- Design By Contract (DBC)

Bad

- Mutable (variable)
- Commented Code (XXX)
- Dead Code (XXX)
- Duplicate Code (XXX)
- Arrow Code
- Bool, int, null for error handling
- God Class
- Swiss Knife
- Bool, null, optional parameter
- Out parameter

Good

- SRP
- Low coupling
- Program to an interface (implements)
- Upcasting
- LSP
- ISP
- DIP
- Favour composition (ref)
- size of class
  - Max - 12 interface methods
  - Avg - 4 interface methods
- dont talk to strangers (Law of Demeter)
- YAGNI
- Functional programming
- Declrative programming
- AAA security pattern

Bad

- Coupling
  - Unidirectional Tight Coupling (A->B) **< 7**
  - (big no) Bi-directional/Cyclic coupling (A->B, B->A)
  - (no,no,no) Many to many coupling
- Static Methods
- Down casting
- Type checking
- Inheritance (extends) **< 4**
- Flag
- God Class
- Static polymorphism
- AOP
- Functional Interface -> Lilliput classes

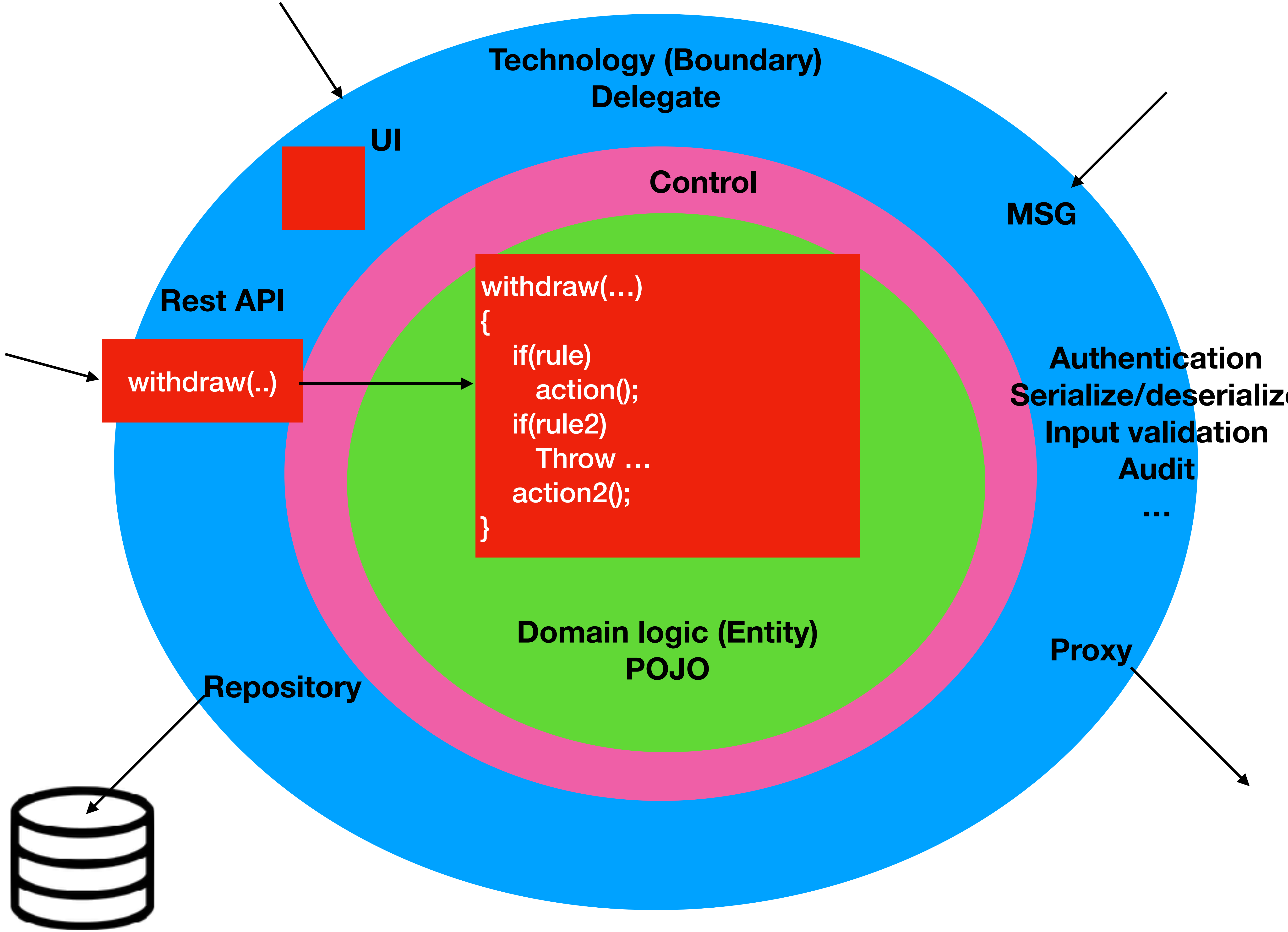
- Cyclomatic complexity
- Coupling

```
Tiger t = new Tiger;
//up - abstraction (program interface)
Animal a = t;

//down - anti abstraction (anti interface)
Tiger t2 = (Animal) a;
```

```
double withdraw(double amount)

{

    if(! IsSufficientBalance(amount))

        ….

    balance -= amount

    //..

}
```

- If condition
    - Domain (Domain rules - requirements) >,<,>=,<=,…
        - If sal > 5000, type of investor rule,
        - Specification pattern (write a separate method/class for all domain rules)
    - technical (programmer introduced if) ==, !=
        - **Error Handling conditions (avoid using exception handling)**
        - Validation Conditions (null pattern, annotations, …)
        - Flow navigation Condition (polymorphism)
            - Replace a flag with Object per flag value
            - Replace flag with Sub class per flag value
            - Lookup (Map)
- Coupling
    - Interface (OO interface, fun interface)
    - Mediator
    - Wrapper

- Seperation of concerns
  - Separate technology logic from domain logic
  - Separate domain logic from domain rules
  - Separate Error Handling logic from domain logic
  - Separate steps(action) from flow



Technology (Boundary)
Delegate

UI

Control

MSG

Rest API

withdraw(..)

```
withdraw(…)
{
    if(rule)
       action();
    if(rule2)
       Throw …
    action2();
}
```

Authentication
Serialize/deserializ
Input validation
Audit
…

Domain logic (Entity)
POJO

Repository

Proxy

void fun()

{

Setting Expectations

1. Preconditon - Fun’s Expectation from caller
    1. What should you pass to this fun
2. Post Condition - What caller can expect from fun

Contract

Implementation

Logic

}

```
double getExpenseLimit() {
  Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject != null);
  return (_expenseLimit != NULL_EXPENSE) ?
    _expenseLimit:
    _primaryProject.getMemberExpenseLimit();
}
```

```
fun2()
{
    fun(10);
}
…

fun(int i)
{
     Assert (i > 0);
    //if(i <=0 ) <— input validation
     //   Throw …

    Logic
}
```

```
void fun()
{
    int i;
..

    I = I + 2;
…
    I = 6

  ..
}
```

```
void fun()
{
    Const int i=4;
..

    Const int j  = I + 2;
…
    Const int k = 6;

  ..
}
```

C++ | C#

```
Emp* e;
fun(&e);
```

```
void fun(Emp** r)
{
  *e = new Emp()
}
```

```
void fun(ref Emp r)
{

}
```

```
class Stack
{
    void push(int) { … }
    int pop() { … }
}
```

```
class StackTest
{
    void usage1()
    {
        //arrange
        Stack s = new Stack();
       //act
       s.push(10);
        //assert
        val = s.pop();
        asset(val,10);
    }
    void usage2() {…}
    void usage3() {…}
    void usage4() {…}
    …
}
```

1. Regression (*). - agility
2. Documentation
3. Class Design
4. Find bugs

```
class CA
{
    void f1() { … }
    void f2() { … }
    void f3() { … }
    void f4() { … }
    void f5() { … }
    void f6() { … }
    void f7() { … }
}
```

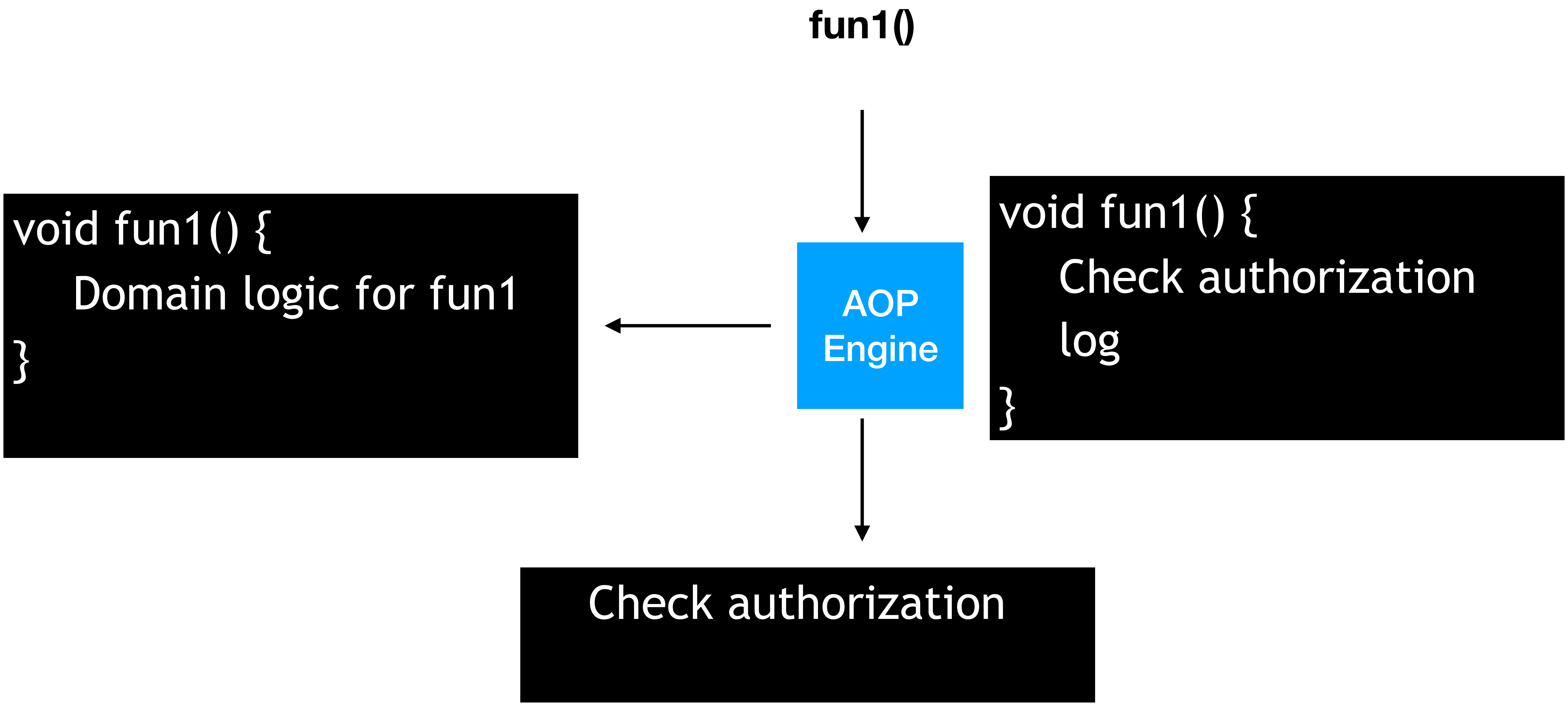
```
class CATest
{
    Does it depend on methods in CA ?
}
```

Cross cutting concerns

```
void fun1() {
    Check authorization
    Domain logic for fun1
    log
}
void fun2() {
    Check authorization
    Domain logic for fun2
    log
}
void fun3() {
    Check authorization
    Domain logic for fun3
    log
}
```

```
log(f)
{
   Make Log
   f()
}
```

- Aspect orient programming (AOP)



fun1()

```
void fun1() {
    Domain logic for fun1
}
```

AOP Engine

```
void fun1() {
    Check authorization
    log
}
```

```
Check authorization
```

- DEAD

1. **A**uthentication (who are you) - first defence
    - By knowledge (what you know) - pwd, secret,
    - By Possession (what you have) - otp, email, rosa tokens
    - Bio (what you are) - face, finger, voice, dna, …
2. **A**uthorization (what can you do)
    - Role based
3. **A**udit log (what did you do) - last defence
    - Write log
4. Input validation (70%)
    - Range, null, type check, …
5. Exception Handling
    - try/catch
6. Asset Handling (credit card, Personal, …)
    - Transit (wire) -> **HTTPS**
    - Rest (storage) -> Encryption
7. Session Handling
8. Key management, pwd