

System Design

Author: Daria Shutina

System Design

24-02-01

Architecture

Requirements

24-02-07

Approaches to decomposition

Basic principles of decomposition

Principles of OO design

SOLID

Practice 1

Interview part

24-02-01

Architecture

First, the system is decomposed into some components. We need to understand how the components interact with each other (e.g. how they share resources). Then we define structures that are necessary for the implementation.

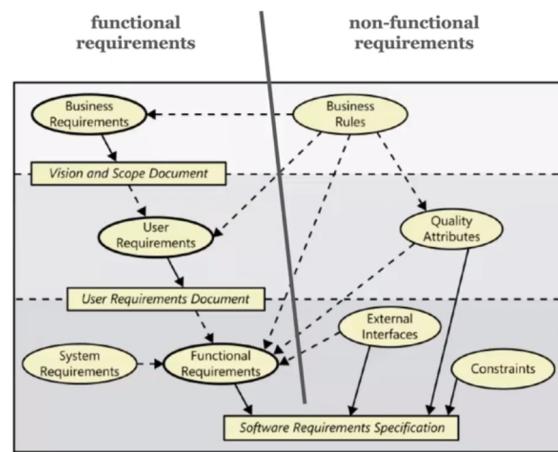
Architecture is the way system components are organized together. It is represented as a diagram with blocks and arrows. It is a tool for organizing the code, so it is important for the development team, not for the customer.

In the real world, architecture is not a static image. It constantly changes in order to meet current requirements (customer preferences, time or performance constraints).

Requirements

Working with requirements

- requirements engineering
 - extraction
 - analysis
 - documentation
 - verification
- requirements management



24-02-07

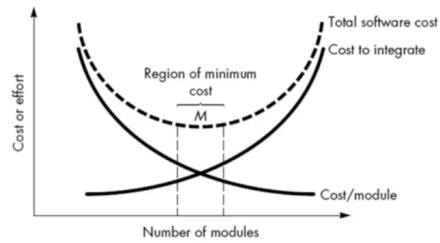
Approaches to decomposition

- **Bottom-up:** you start with small components that solve a specific task. Then you build low-level logic between components.
The problem is there can be inconsistent components on the finish line (e.g. when implementation approach is modified). Also, you need a lot of testing code to check if the low-level logic works correctly.
- **Top-down:** start from creating a prototype you want to achieve, and then decompose it into components.
In this approach, you think about the interfaces first, so you have to encapsulate tasks in these modules. That way you limit the complexity for each component.

Basic principles of decomposition

Modularity

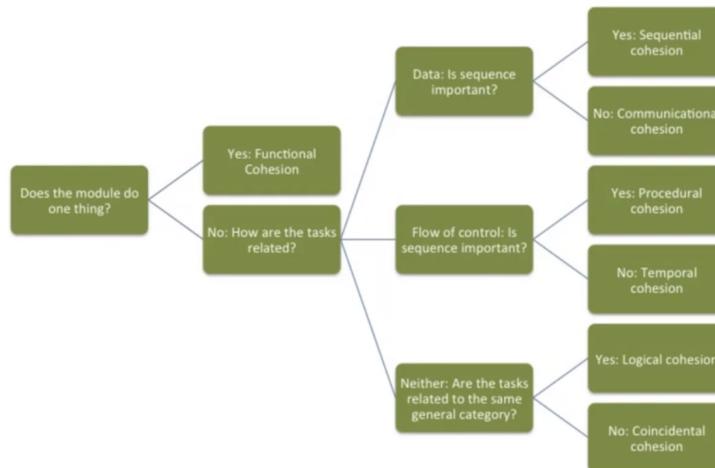
- Dividing the system into components
 - nameable
 - addressable
- Potentially allows you to create arbitrarily complex systems
 - interface + implementation
 - clear decomposition
 - minimization
 - one module — one functionality
 - no side effects
 - independence from other modules
 - principle of data hiding



Information hiding

- The contents of the modules should be hidden from each other
 - all modules are independent
 - exchange only information necessary for work
 - access to module operations and data structures is limited
- Provides the ability to develop modules by various independent teams
- Provides simple modification of the system

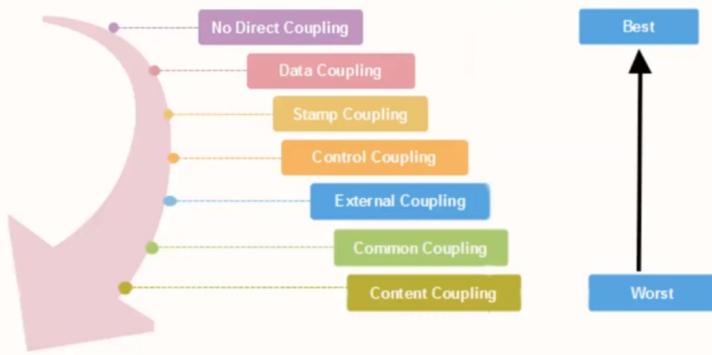
Cohesion



Cohesion refers to the degree to which the elements inside one module belong together.

The scheme shows how the cohesion changes depending on the state of the system. Logical cohesion is the worst option, and sequential cohesion (modules create a sequence) is the best one.

Coupling



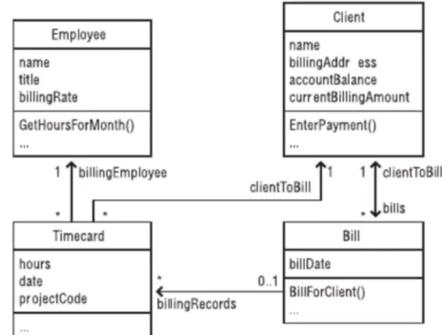
Coupling describes how strong connections between modules are. It shows how many other modules need changes if one module is changed. Ideally, none (none direct coupling is the best option).

Content coupling is the worst option, when one module knows not only logic, but also internals (e.g. private fields) of the other module.

Principles of OO design

Define real world objects

- Defining objects and their attributes
- Defining actions that can be performed on each object
- Defining relationships between objects
- Defining the interface of each object



Define consistent abstractions

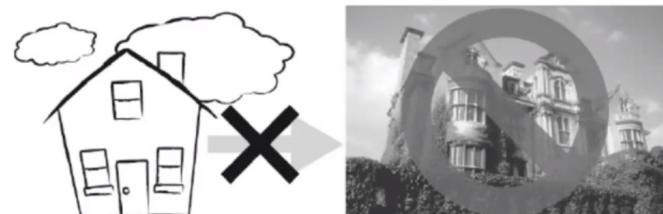
- Highlighting essential characteristics of an object and ignoring non-essential ones
- Defining its conceptual boundaries from an observer's point of view
 - defining interfaces
- Managing complexity through capturing external behavior
- Different levels of abstraction required



When constructing a house, do not think what color or material the door needs.

Encapsulate implementation details

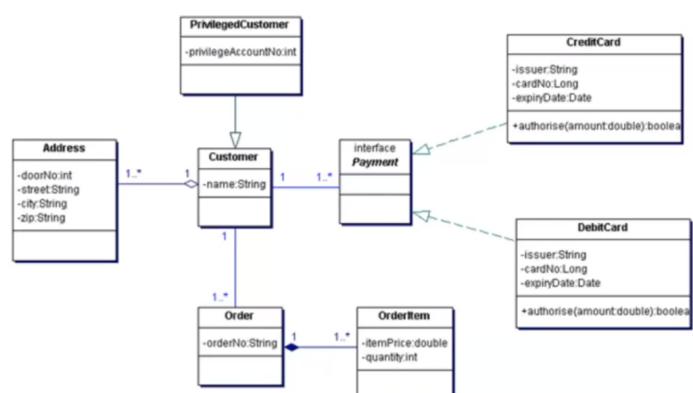
- Separating internal structure and external behavior from each other
- Isolating interface contracts from implementation
- Managing complexity by hiding implementation details



In the example, you can create a structure which describes a house, but you cannot understand from the structure what the house is made of.

Choose between inheritance and composition

- Inheritance
 - “is-a” relation
 - creating new entities
 - expands the idea of abstraction
 - encapsulation violation
- Composition (aggregation)
 - “has-a” relation
 - flexible dependency management at runtime
 - a lot of objects



Hide “extra” information

- Isolating “personal” information
 - secrets that hide complexity
 - secrets that hide the sources of change



Identify areas of likely change

- Identify elements that seem likely to change
- Separate elements that seem likely to change
- Isolate elements that seem likely to change
 - interfaces, interfaces, interfaces!
- Sources of change
 - business rules
 - equipment dependence
 - input-output
 - non-standard language capabilities
 - complex aspects of design
 - status variables
 - data structures
 - ...

Additional principles

- High cohesion and low coupling
- Formalize class contracts
 - define pre- and post-conditions, invariants
- Design testable systems
- Consider using “brute force”
- Draw diagrams

SOLID

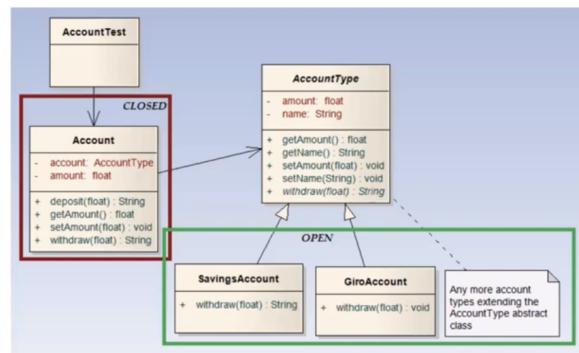
Single responsibility principle

- each object must have one responsibility
- this responsibility must be completely encapsulated in a class



Open/closed principle

- software entities (classes, modules, functions, etc.) must be open for extension, but closed for modification
 - reuse via inheritance
 - immutable interfaces



If we want to add new features to the class, we need to extend it, not modify it.

The optimal solution is to use interfaces, each interface is responsible for a specific feature.

Liskov substitution principle

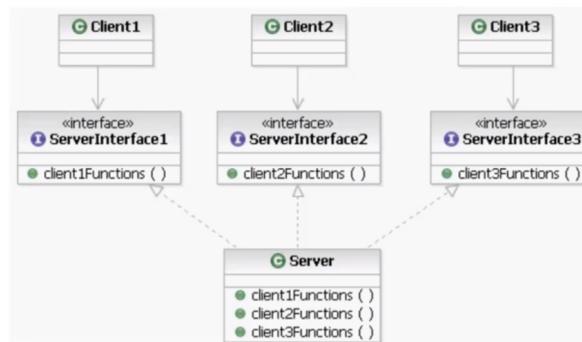
- Functions that use a base type should be able to use subtypes of the base type without knowing it



You should be able to take any subclass and put it where you expect the base class, and everything should work fine.

Interface segregation principle

- Clients should not be dependent on methods they do not use
 - interfaces that are too generic need to be divided into smaller and specific ones



Another example. We have interface `Payments` where there are some payment methods:

```

1 public interface Payments {
2     void payWebMoney();
3     void payCreditCard();
4     void payPhoneNumber();
5 }
```

We want to create a class `InternetPaymentService` -- a service which is responsible for the payment and it does not accept payments by phone. If we implement it from the `Payments` interface, the service will have to implement unnecessary methods:

```

1 public class InternetPaymentService implements Payments{
2     @Override
3     public void payWebMoney() { ... }
4
5     @Override
6     public void payCreditCard() { ... }
7
8     @Override
9     public void payPhoneNumber() {
10         // unnecessary method
11     }
12 }
```

The better solution is to create three interfaces instead of `Payment`:

```

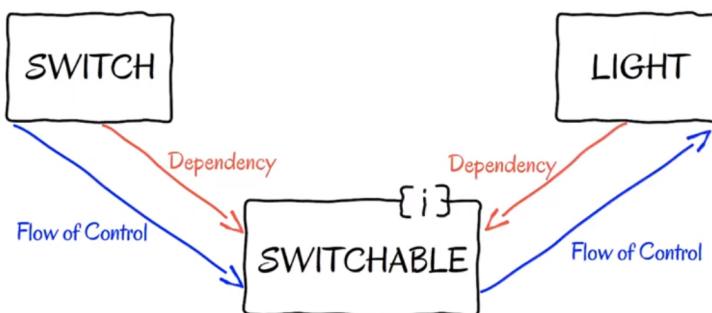
1 public interface WebMoneyPayment {
2     void payWebMoney();
3 }
4
5 public interface CreditCardPayment {
```

```

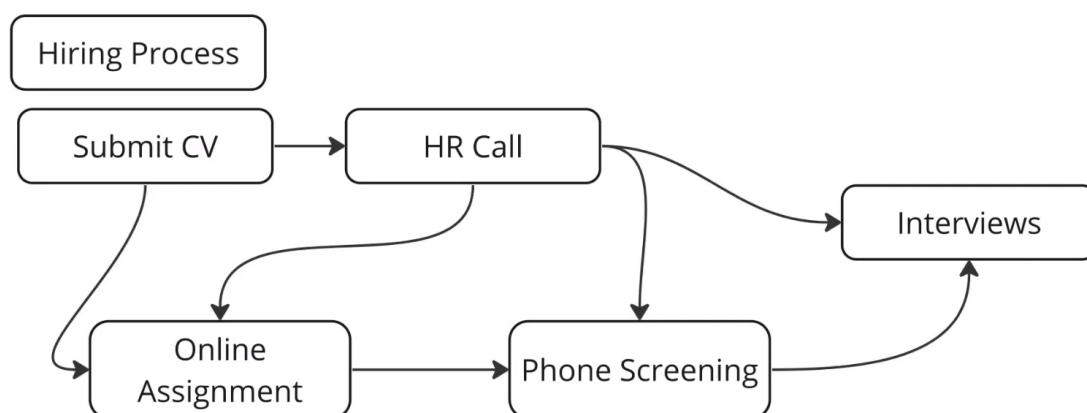
6   void payCreditCard();
7 }
8
9 public interface PhoneNumberPayment {
10    void payPhoneNumber();
11 }
12
13 // now only necessary methods will be implemented
14 public class InternetPaymentService implements WebMoneyPayment,
15                                              CreditCardPayment {
16    @Override
17    public void payWebMoney() { ... }
18
19    @Override
20    public void payCreditCard() { ... }
21 }
```

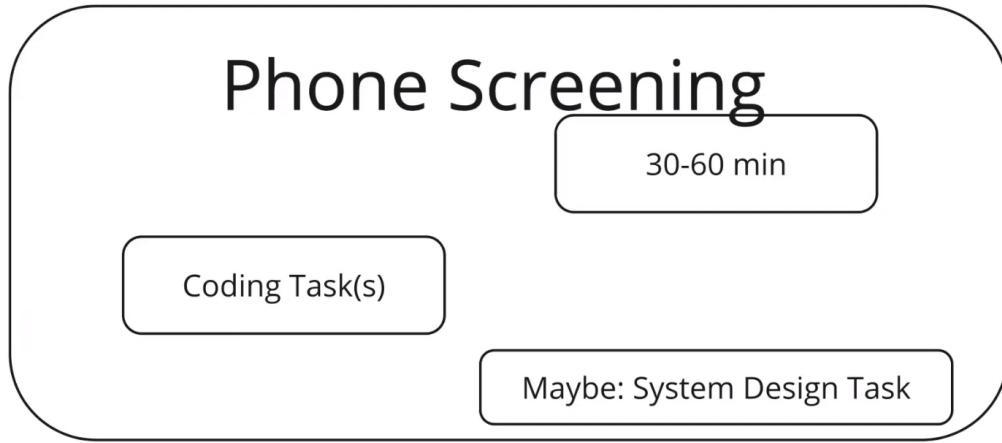
Dependency inversion principle

- Modules at higher levels should not depend on modules at lower levels.
Both types of modules must depend on abstractions
- Abstractions should not depend on details. Details must depend on abstractions



Practice 1





Interview part

- **Behavior**

Besides common questions like who you are and what you expect, you will be asked about previous experience: deadlines, possible conflicts, motivation.

When answering this question, you can use **S.T.A.R. approach**.

- situation - what was the context/background of the situation you were in?
where did it occur and what happened?
- task - what was your role? what was the goal? what were the consequences if nothing happened?
- action - what did you personally own? how did you do it? who else was involved?
what obstacles did you meet?
- result - what results did you achieve? what did you learn? how did you measure success of the project?

- **Coding**

1. Most probably the interviewer explains the task verbally. So, clarify all moments you are unsure about, e.g. lower and upper bounds, types of elements (null?), whether the object has some specific attributes (array \Rightarrow is it sorted?).

Give some examples where the task is solved. Also, think about time and memory complexity.

2. When the interviewer is okay with your clarifications, start writing the code. Pay attention to formatting, names of variables.

You can also write some tests where, on each step of the implementation, it is explained what happens with the object.

Task #1
Find max in array on ints. not sorted.

```

1 < let(array) < 10k
-10k <= array[i] <= 10k
empty? null?
[1,3,2] => 3
[1,1,1] => 1

```

Clarify the task

```

Approach #1
sort array
return the last
TC = O(n log n), MC = O(n) // only input array

```

Approach #2

```

max = array[0]
iterate over array
update max
TC = O(n), MC = O(n) // only input array

```

Discuss approaches for the implementation

```

// [1,3,2]
int find_max(int[] array) {
    int max = array[0]; // max = 1

    for(int i = 0; i < array.length; ++i){ // i = 0; 1; 2; 3;
        if(max < array[i]) { // 1 < 1; 1 < 3; 2 < 3
            max = array[i]; // max = 3;
        }
    }
    return max; // max = 3
}

```

In the comments you can add an example of how your code works

- **System Deep Dive**

Prepare a story about the project that you did or took part in. Explain what problem you had, then what architecture and what technologies you used to solve it.

- **System Design**

During the system design interview, you are given a task to design a system (it can be used for payments, reservations, etc). First of all, you clarify requirements, boundaries, discuss possible solutions. If it is okay for the interviewer, you can use [diagrams](#) for that.

For example, you are asked to create a URL shortener: a user submits a long URL, and a service returns a short one.

Numbers used below are mostly your assumptions.

1. Functional requirements:

It is a web service. No login, no registration - the service is open to everyone.

The length of a short URL is 6 symbols. 26 lowercase letters + 10 digits can be used.

Use case #1. User sends a long URL, the service returns a short one.

Use case #2. User sends a short URL, the service returns the original one.

2. Non-functional requirements:

Daily users? 10k per day.

How many URLs can we store? 4 billions.

$4B / 10k = 400k; 365 < 400 \Rightarrow$ We can store URLs for ~ 1000 years. Quite a lot.

Storage per row = (short URL, long URL, id) = (6 bytes, 5k bytes, 8 bytes) = 5 kilobytes per row

All storage: 5 kilobytes * 4B \approx 16 TB

3. API - HTTPS/REST is used.

- Get short URL

Request: POST

Body: json { "url": "<long_url>" }

Response: 200 OK

Body: json { "url": "<short_url>" }

If there was a error, the response is 400 Bad Request

- Get original URL

Request: GET <short_url>

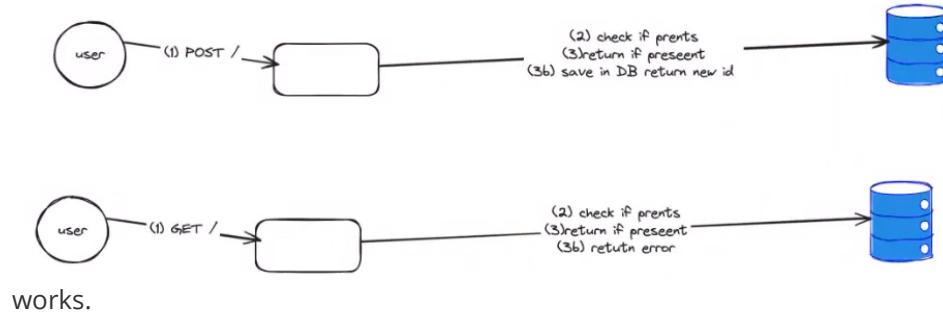
Response: 301 Redirect

Header: Original url

In case of error, the response is 400 Bad Request

If the entry is not found, response with 404 Not Found

4. Draw diagrams describing how the API



works.