System Design

Author: Daria Shutina

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.

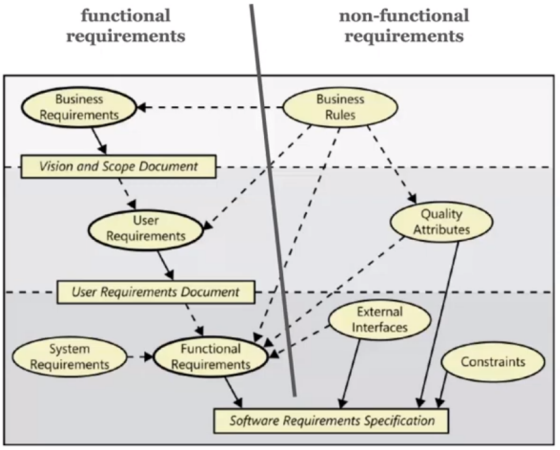
Architecture is the way system components are organized together. It is represented as a diagram with blocks and arrows. It is a tool for organizing the code, so it is important for the development team, not for the customer.

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



02 - Decomposition

The idea is to break a complex system into more simple ones, then iterate the process again and again until the pieces become atomic.

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 is the low-level logic works correctly.

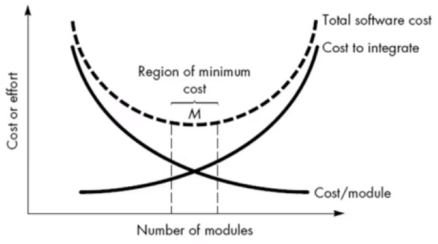
- **Top-down:** start from creating a prototype you want to achieve, ans then decompose it into components.

  In this approach, you think about the interfaces first, so you have to encapsulate tasks in this modules. That way you limit the complexity for each component.

Basic principles of decomposition

Modularity

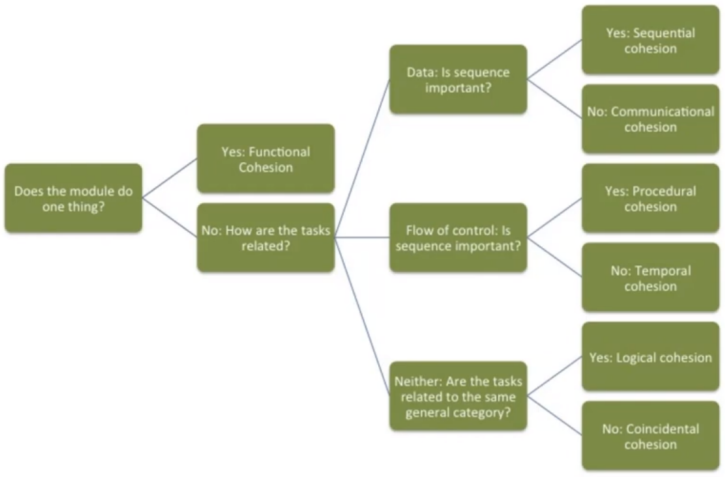
- Dividing the system into components
  - nameable
  - addressable
- Potentially allows you to create arbitrarily complex systems
  - interface + implementation
  - clear decomposition
  - minimization
  - one module — one functionality
  - no side effects
  - independence from other modules
  - principle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

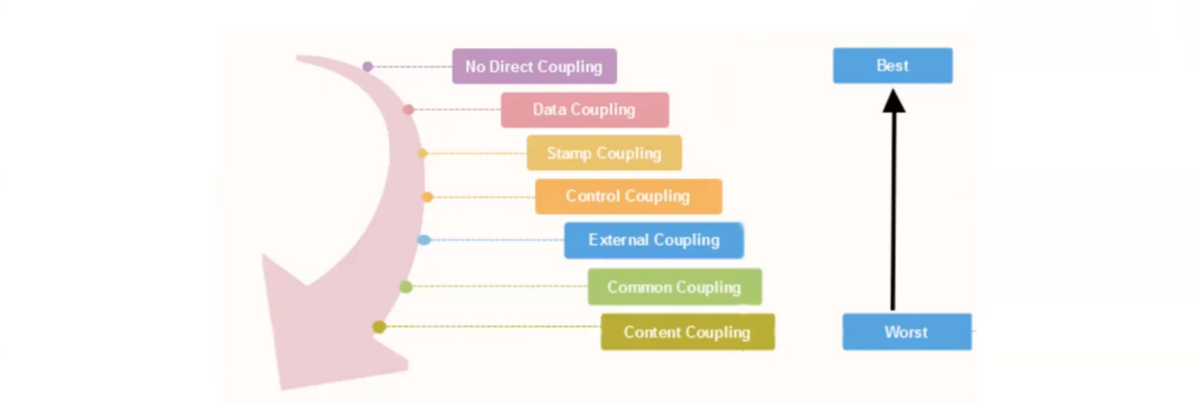
Cohesion



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



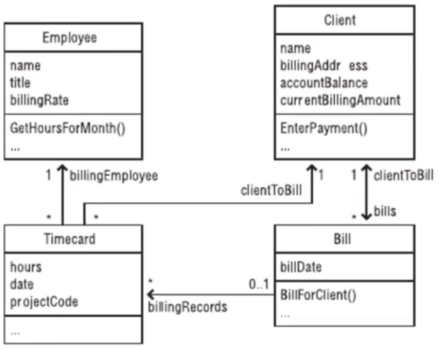
Coupling describes how strong connections between modules are. It shows how many other modules need changes if one module is changed. Ideally, none (none direct coupling is the best option).

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



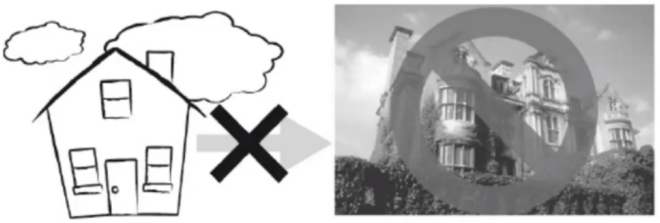
Define consistent abstractions

- 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.

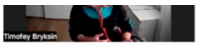
Encapsulate implementation details

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

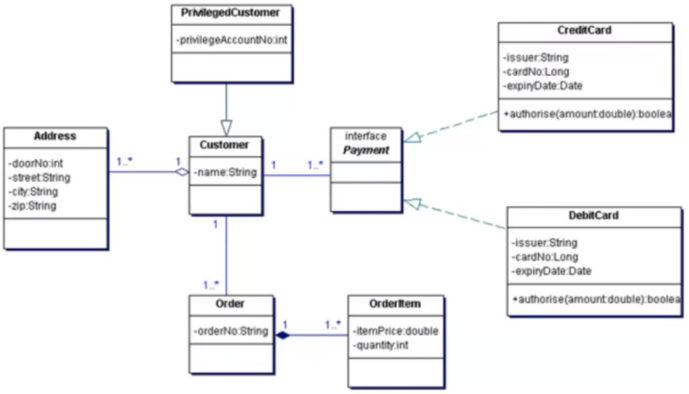


In the example, you can create a structure which describes a house, but you cannot understand from the structure what the house is made of.

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

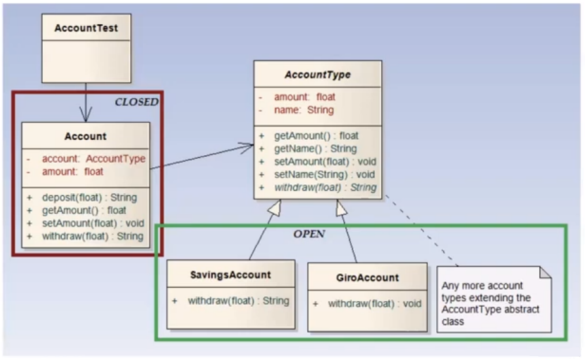
SOLID

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.

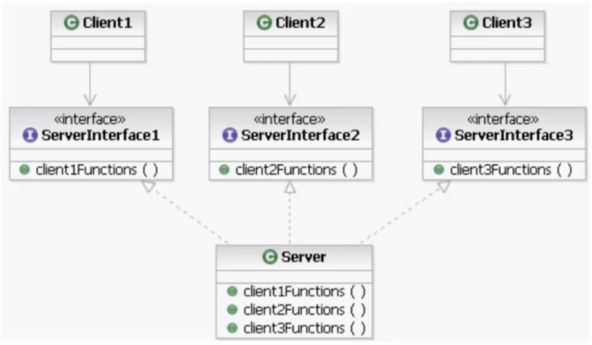
Liskov substitution principle

- 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:

```
public interface Payments {
    void payWebMoney();
    void payCreditCard();
    void payPhoneNumber();
}
```

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:

```
public class InternetPaymentService implements Payments{
    @Override
    public void payWebMoney() { ... }

    @Override
    public void payCreditCard() { ... }

    @Override
    public void payPhoneNumber() {
        // unnecessary method
    }
}
```

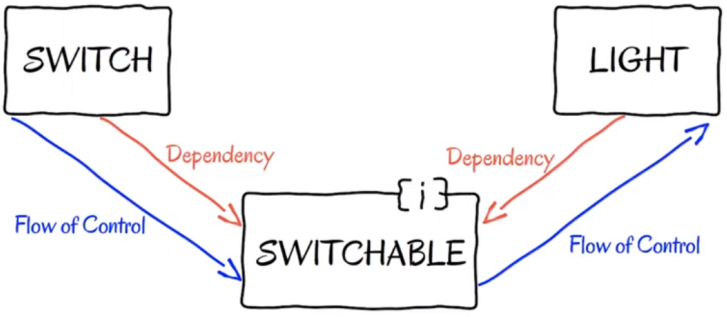
The better solution is to create three interfaces instead of `Payment`:

```
public interface WebMoneyPayment {
    void payWebMoney();
}

public interface CreditCardPayment {
```

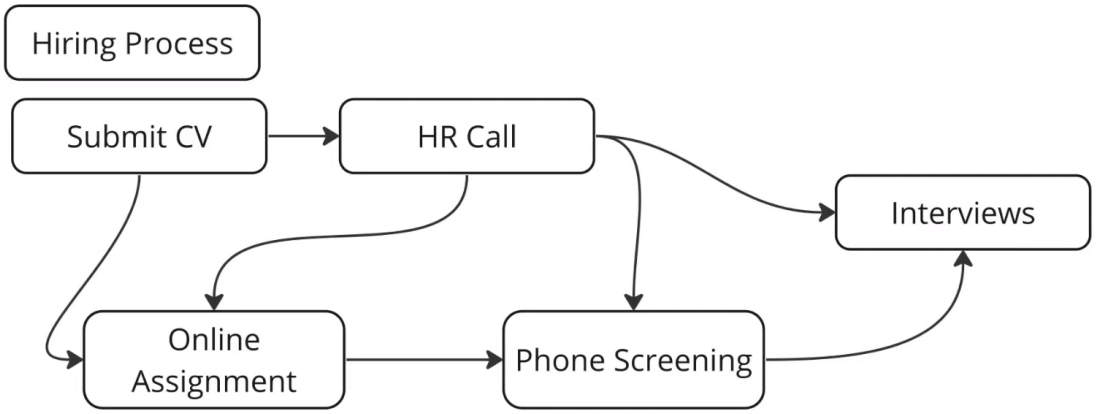
```
    void payCreditCard();
}

public interface PhoneNumberPayment {
    void payPhoneNumber();
}

// now only necessary methods will be implemented
public class InternetPaymentService implements WebMoneyPayment,
                                                CreditCardPayment {
    @Override
    public void payWebMoney() { ... }

    @Override
    public void payCreditCard() { ... }
}
```

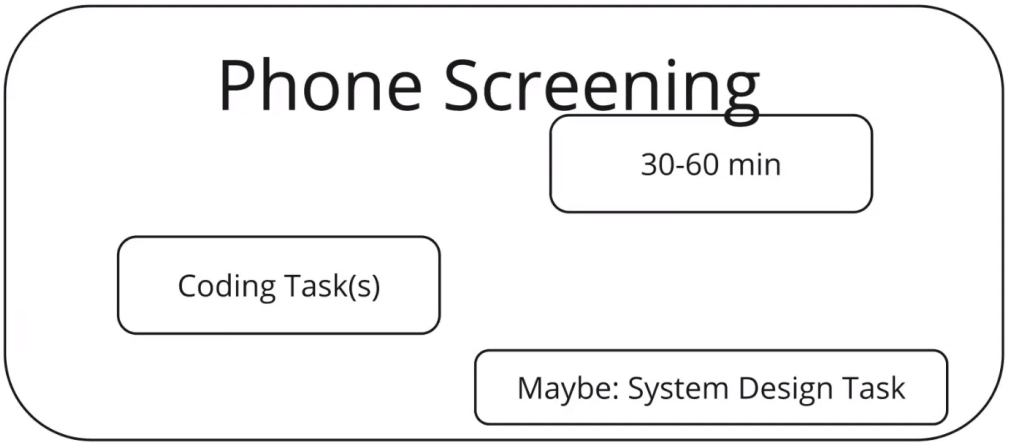
Dependency inversion principle

- 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?

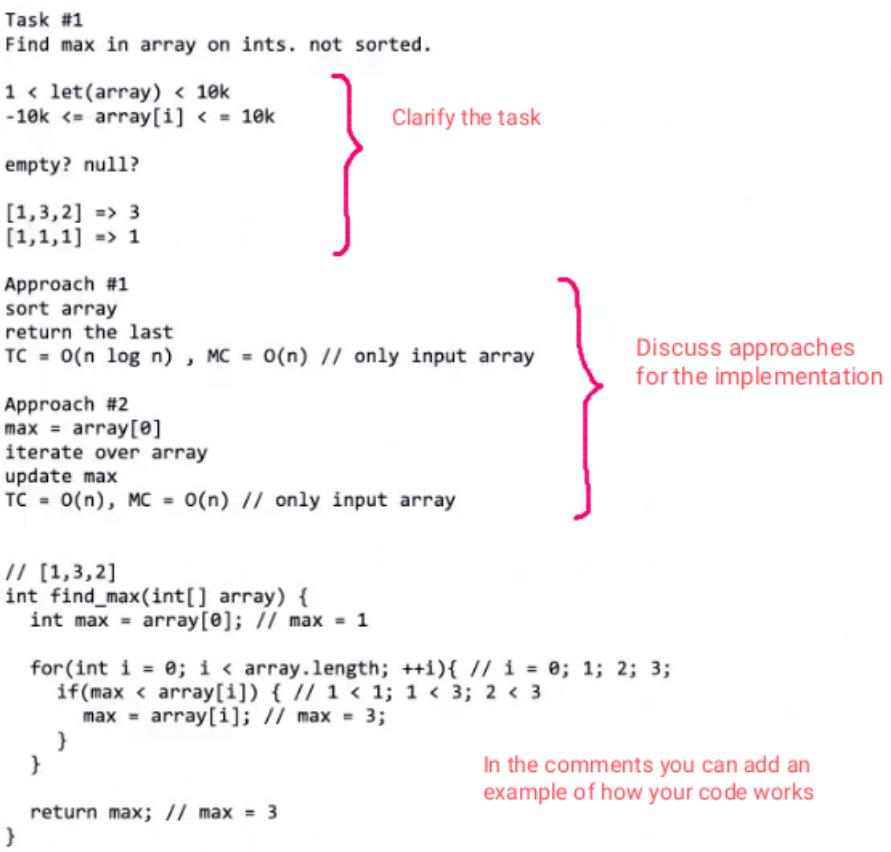
- **Coding**

  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.



- **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 ⇒ 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 ≈ 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>`

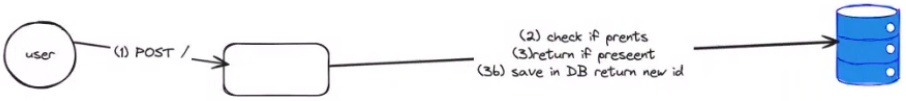
      Response: `301 Redirect`

      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.

03 - Architectural patterns and styles

An **architectural style** is a set of decisions that

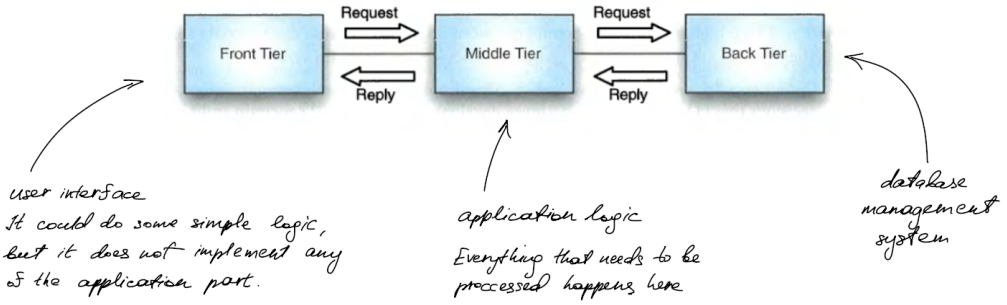
1. are applicable in the selected development context
2. set restrictions on the architectural solutions specific to certain systems in this context
3. lead to the desired positive qualities of the resulting system

An **architectural pattern** is a named set of key design decisions for the efficient organization of subsystems, applicable to repeatable technical design tasks in various contexts and domains.

Styles are more like guidelines - they are very abstract, - while patterns are usually much more concrete. One style can be implemented using different patterns.

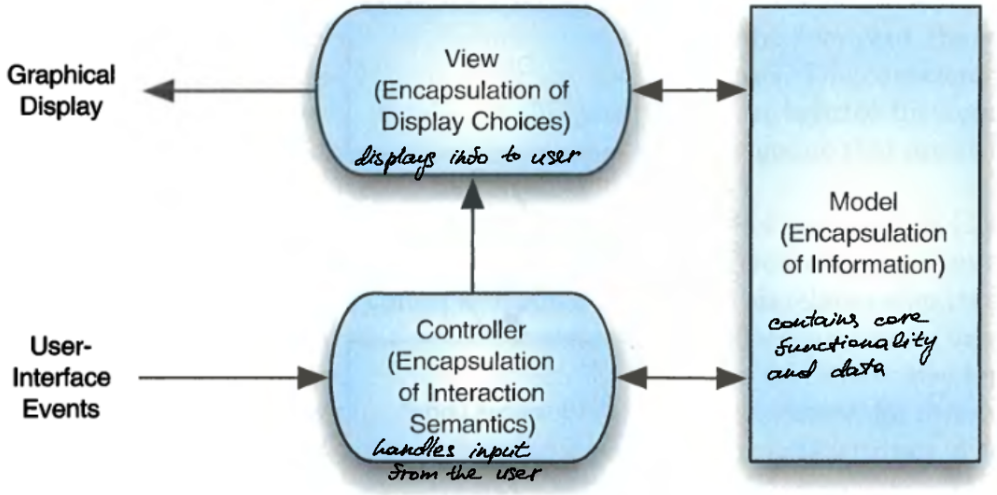
Patterns

- **Three-tier architecture**



- **Model-View-Controller**

  We have an application which knows how to handle, store and show data. Instead of creating one class responsible for all logic (which violates Single Responsibility principle), we separate the logic between three main classes.

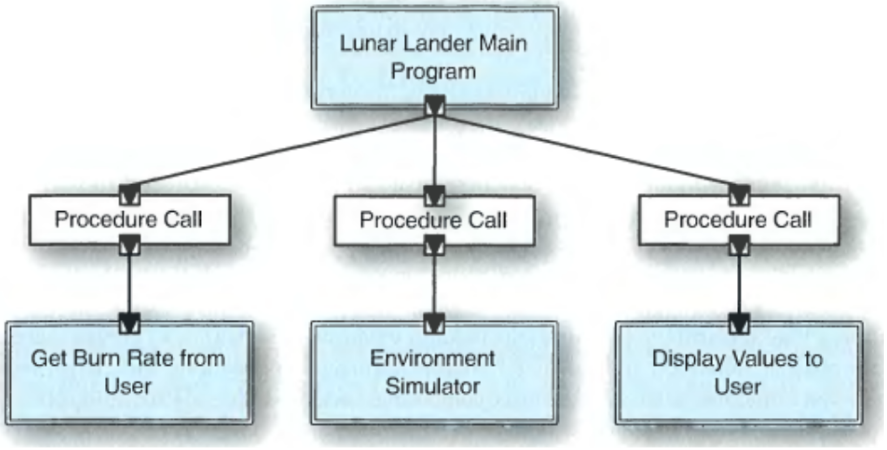


Styles

- **Main procedure and subroutines (Master-Slave)**

  This style is suitable for procedural languages (C, Pascal, etc). It has the master component and a bunch of identical slave components that are called to execute specific tasks.

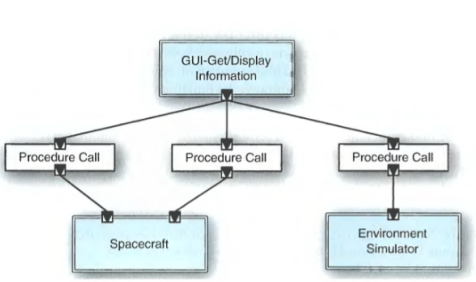
The master component knows about the slave components. They, in turn, do not know about each other. The master component implements the algorithm and distributes the work between slave components.

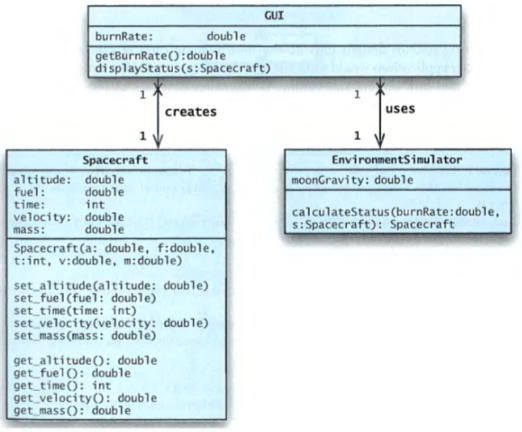


This approach is popular, since it is simple and straightforward. The drawback is it does to scale. If you need several leading components, you need to think of a super parent for them. And it makes the system too complex.

- **Object-oriented design**

  Here we build a system from standalone components which communicate with each other but does not depend on each other.





- **Layered architecture**

  The system is separated into several layers. Each layer represents specific responsibility and provides services to the next higher layers. The layers with lower logic do not know about the higher ones. There can be modifications like each layer knows only about the next layer and not about layers beneath.

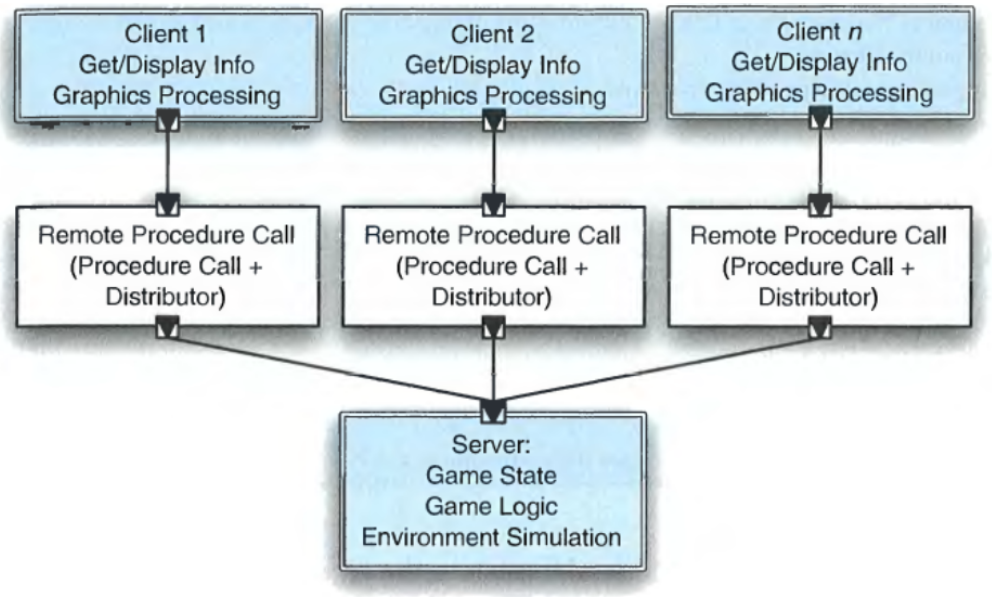
  The only remark is when you have a lot of layers and decide to send a message from the top one to the bottom one, the system becomes highly ineffective.

  This approach is widely used for desktop and web applications. A common example is when the app has four layers:

  - Presentation (UI layer)
  - Application (service layer)
  - Business logic (domain layer)
  - Data access (persistence layer)

- **Client-server**

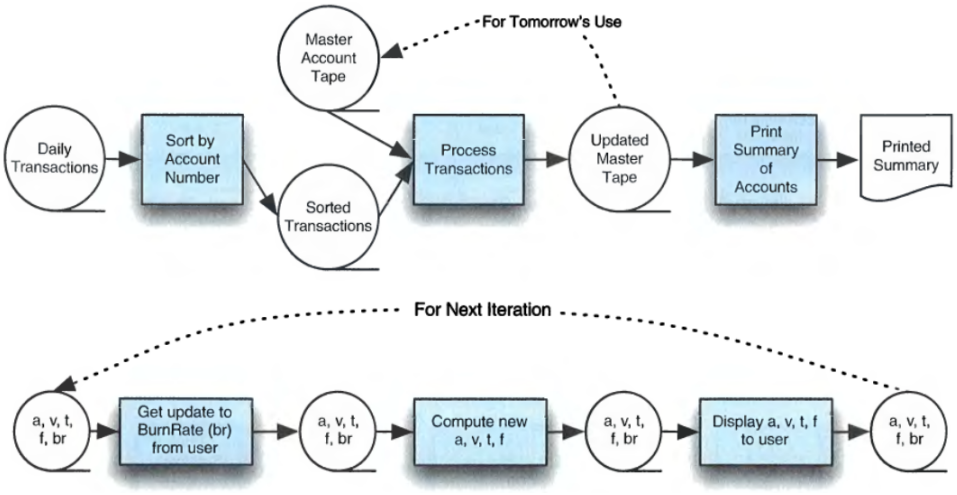
  Consists of a server and multiple clients. A server component provides services to client components.



  This approach is widely used in online applications.

- **Batch-processing**

  The algorithm is split into specific parts, and data is moved between the parts in order to be handled.



  It works only for linear execution where there is not a lot of interactions.

- **Pipes and filters**

  This approach is used to structure systems which produce and process a stream of data.

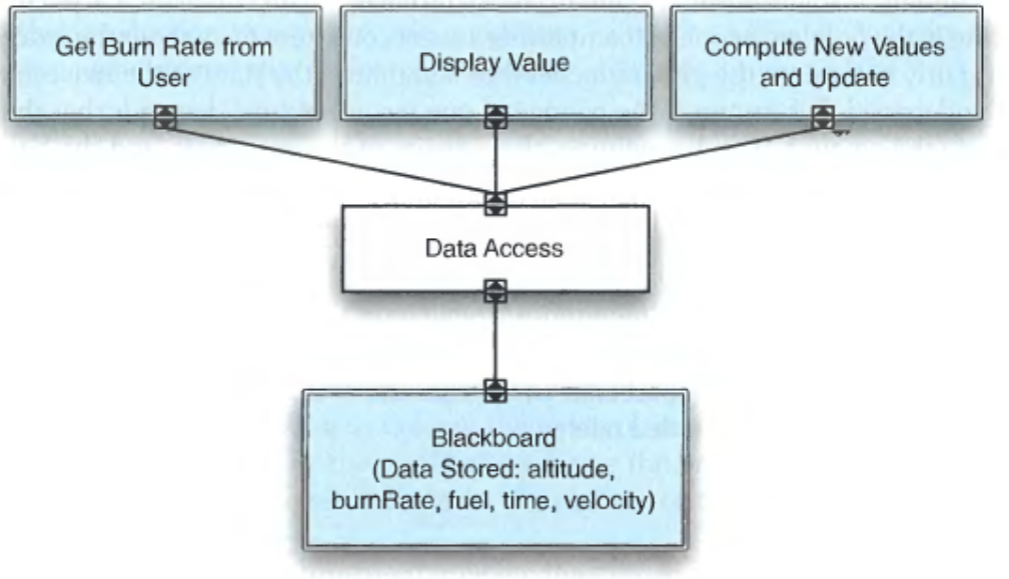
  Data is moved between different components called *fields*. Channels of communication between fields are called *pipes* (or *streams*). Pipes are used for buffering or synchronization.

The idea is to enable parallel execution. Fields and pipes construct a graph of execution. Things can be done in parallel, because components are independent.

- **Blackboard**

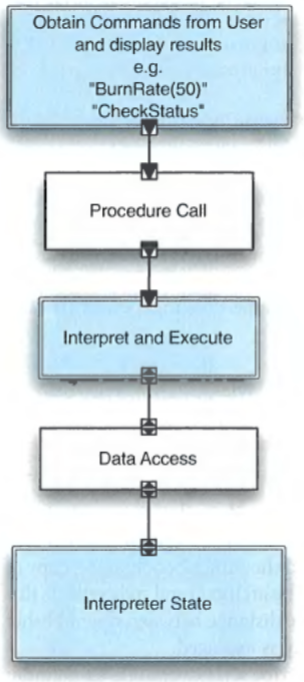
  Blackboard is a structured global memory containing some objects.

  All components have access to the blackboard. Components may produce new objects which are added to the blackboard (that is where the pattern name comes from).



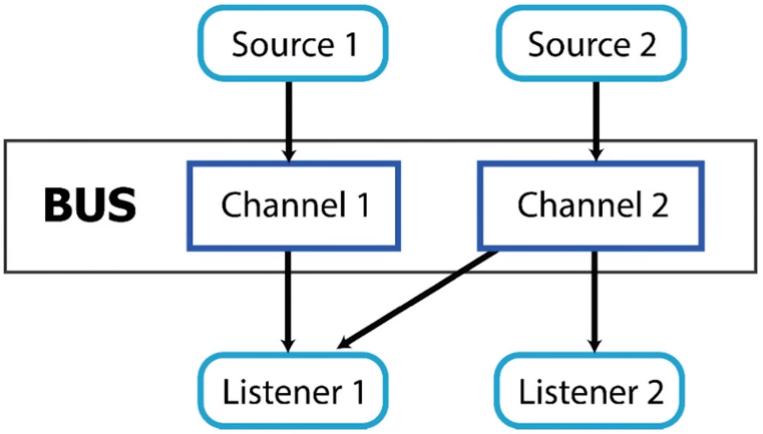
- **Interpreter**

  This approach is used to design a component that interprets programs written in a dedicated language.



- **Event-driven**

  There are four main entities: event source, event listener, channel and event bus.



The source publishes message into particular channels on an event bus. Listeners subscribe to particular channels. Therefore, they are notified of messages that are published to channels they have subscribed to.

This approach is used in android development and notification services.