04834580 Software Engineering (Honor Track) 2025-26

Design Principles

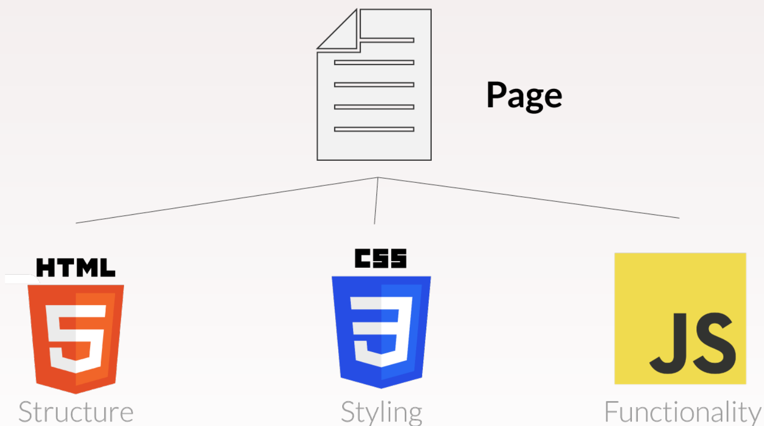
Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Separation of Concerns

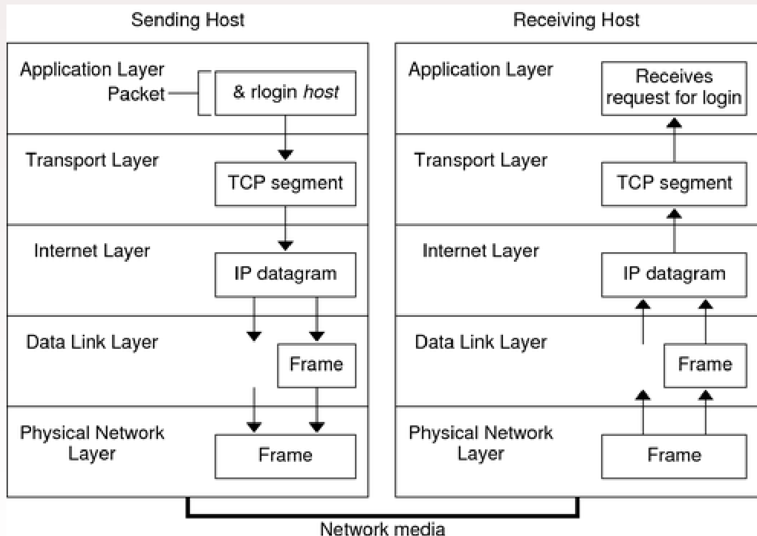
Edsger W. Dijkstra in “On the role of scientific thought” [1]:

> *We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained — on the contrary! — by tackling these various aspects simultaneously. It is what I sometimes have called “the separation of concerns”.*

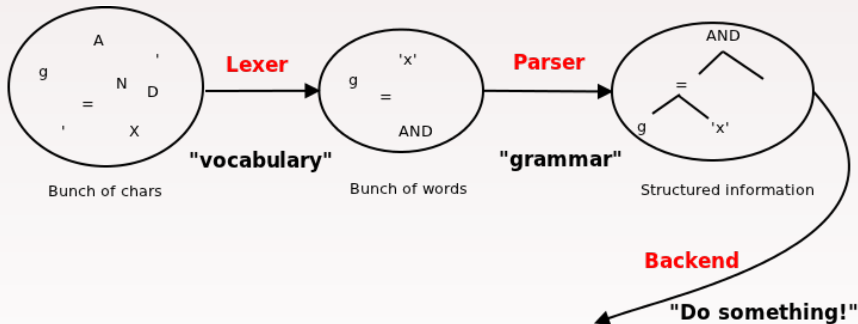
Separation of Concerns: HTML, CSS, JavaScript



Separation of Concerns: TCP/IP Protocol Stack



Separation of Concerns: Parsers



SOLID Principles

Five object-oriented design principles compiled by Robert C. Martin [2]:

- **S** — Single Responsibility: a class has one reason to change
- **O** — Open/Closed: open for extension, closed for modification
- **L** — Liskov Substitution: subtypes must be substitutable for their supertypes
- **I** — Interface Segregation: clients should not depend on interfaces they do not use
- **D** — Dependency Inversion: depend on abstractions, not concretions

Single Responsibility Principle

Robert C. Martin, the originator of the term [2, 3], expresses the principle as

> *A class should have only one reason to change.*

Illustrating Example:

> *As an example, consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for different causes. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should, therefore, be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.*

Single Responsibility Principle: Negative Example

北京大学

```java
class Report {
    private String title;
    private List<String> sections;

    public Report(String title) { this.title = title; ... }
    public void addSection(String text) { sections.add(text); }

    // Content concern: what the report says
    public String getContent() {
        return title + "\n" + String.join("\n", sections);
    }

    // Formatting concern: how the report looks
    public void printAsHTML() {
        System.out.println("<html><h1>" + title + "</h1>...</html>");
    }

    // Persistence concern: where data is stored
    public void saveToDatabase() { /* SQL code */ }
}
```

Three reasons to change: report content, output format, and persistence logic.

Single Responsibility Principle: Improved Example

北京大学

```java
class Report {                          // only responsible for content
    private String title;
    private List<String> sections;
    public String getContent() { return title + "\n" + String.join("\n", sections); }
}

class HTMLReportFormatter {             // only responsible for HTML output
    public void print(Report r) {
        System.out.println("<html><h1>" + r.getTitle() + "</h1>...</html>");
    }
}

class ReportRepository {                // only responsible for persistence
    public void save(Report r) { /* SQL code */ }
}
```

Each class now has exactly one reason to change.

Open/Closed Principle

Formulated by Bertrand Meyer [4]:

> *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

Open/Closed Principle: Negative Example

北京大学

```java
class Drawing {
    public void drawShape(String shape) {
        if ("circle".equals(shape)) {
            System.out.println("Drawing a circle");
        } else if ("rectangle".equals(shape)) {
            System.out.println("Drawing a rectangle");
        }
        // Adding new shapes requires modifying this method:
        // else if ("triangle".equals(shape)) {
        //     System.out.println("Drawing a triangle");
        // }
    }
}
```

Open/Closed Principle: Improved Example

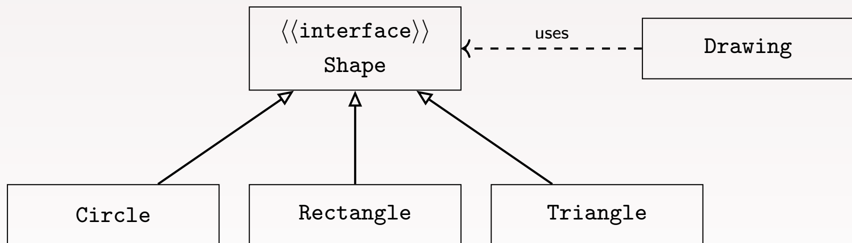
北京大学

```java
interface Shape {
    void draw();
}

class Circle implements Shape {
    @Override
    public void draw() { System.out.println("Drawing a circle"); }
}

class Rectangle implements Shape {
    @Override
    public void draw() { System.out.println("Drawing a rectangle"); }
}

// Drawing is closed for modification: adding Triangle never touches this class
class Drawing {
    public void drawShape(Shape shape) { shape.draw(); }
}
```

Open/Closed Principle: Class Diagram

北京大學

Adding a new shape (e.g., `Triangle`) requires only a new class — `Drawing` is never modified:



Liskov Substitution Principle

Barbara Liskov and Jeannette Wing formulated this principle in "A behavioral notion of subtyping" [5]:

> *Subtype Requirement: Let* $\phi(x)$ *be a property provable about objects* x *of type* T*. Then* $\phi(y)$ *should be true for objects* y *of type* S *where* S *is a subtype of* T*.*

Informally: *objects of a subclass must behave correctly wherever objects of the superclass are expected.*

Interface Segregation Principle

Robert C. Martin [2]:

> *Clients should not be forced to depend upon interfaces that they do not use.*

Prefer several small, focused interfaces over one large, general-purpose ("fat") interface.

```java
interface Worker {
    void work();
    void eat();     // not applicable to all workers
    void sleep();   // not applicable to all workers
}

class Human implements Worker {
    public void work()  { /* works */ }
    public void eat()   { /* eats */ }
    public void sleep() { /* sleeps */ }
}

class Robot implements Worker {
    public void work()  { /* works */ }
    public void eat()   { throw new UnsupportedOperationException(); }
    public void sleep() { throw new UnsupportedOperationException(); }
}
```

Robot is forced to depend on methods it cannot use — a fat interface creates unnecessary coupling.

Interface Segregation Principle: Improved Example

北京大学

```
interface Workable { void work(); }
interface Feedable { void eat();  }
interface Restable { void sleep(); }

class Human implements Workable, Feedable, Restable {
    public void work()  { /* works */ }
    public void eat()   { /* eats */ }
    public void sleep() { /* sleeps */ }
}

class Robot implements Workable {
    public void work() { /* works */ }
    // Not forced to implement eat() or sleep()
}
```

Each class implements only the interfaces relevant to it. New roles can be added without touching existing code.

Dependency Inversion Principle

北京大学

Robert C. Martin [2]:

> *A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *B. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.*

Dependency Inversion Principle: Negative Example

北京大学

```java
class MySQLDatabase {
    public void save(String data) { /* MySQL-specific code */ }
}

class UserService {
    private MySQLDatabase db = new MySQLDatabase(); // tightly coupled

    public void createUser(String name) {
        db.save(name);
    }
}
```

`UserService` cannot be tested without a real MySQL database, and switching databases requires modifying `UserService` directly.

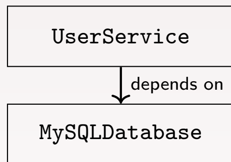
Dependency Inversion Principle: Improved Example

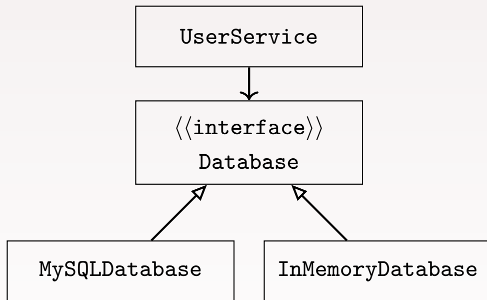
北京大学

```java
interface Database {                          // the abstraction
    void save(String data);
}

class MySQLDatabase implements Database {         // detail depends on abstraction
    public void save(String data) { /* MySQL-specific code */ }
}

class InMemoryDatabase implements Database {      // easy to swap in for tests
    public void save(String data) { /* in-memory code */ }
}

class UserService {
    private Database db;                          // depends on abstraction

    public UserService(Database db) { this.db = db; } // injected by caller

    public void createUser(String name) { db.save(name); }
}
```

`UserService` now works with any `Database` implementation, including a lightweight in-memory mock for testing.

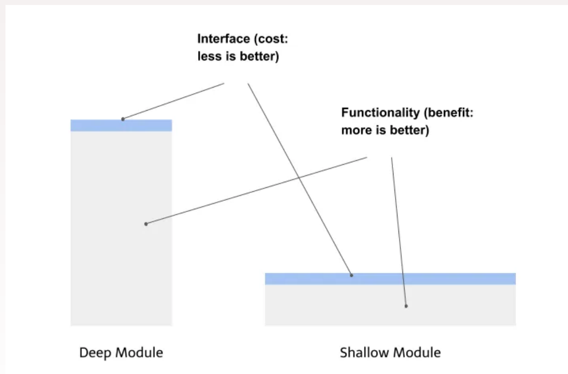
Dependency Inversion Principle: Diagram





Shallow And Deep Interfaces

Deep modules provide a lot of functionality behind a simple interface, while shallow modules have a relatively complicated interface.

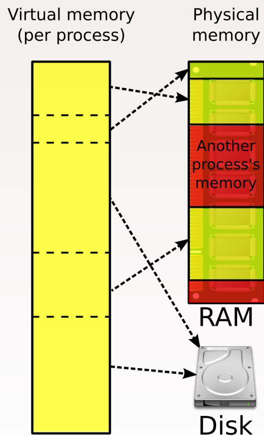


```
int open(const char* path, int flags, mode_t permissions)
int close(int fd)
ssize_t read(int fd, void* buffer, size_t const)
ssize_t write(int fd, const void* buffer, size_t count)
off_t lseek(int fd, off_t offset, int referencePosition)
```

Abstracts:

- On-disk representation, disk block allocation
- Directory management, path lookup
- Permission management
- Disk scheduling
- Block caching

Shallow And Deep Interfaces: Virtual Memory



Linus Torvalds in TED interview [6]:

> *sometimes you can see a problem in a different way and rewrite it so that a special case goes away and becomes the normal case, and that's good code*

```
void remove(list *l, list_item *target)
{
  list_item *cur = l->head, *prev = NULL;
  while (cur != target) {
    prev = cur;
    cur = cur->next;
  }
  if (prev)
    prev->next = cur->next;
  else
    l->head = cur->next;
}
```

```
void remove(list *l, list_item *target)
{
  list_item **p = &l->head;
  while (*p != target)
    p = &(*p)->next;
  *p = target->next;
}
```

By storing a pointer-to-pointer, `*p` is always the link that needs updating — the head-node special case vanishes entirely.

YAGNI (You Aren't Gonna Need It)

Ron Jeffries, a co-founder of extreme programming (XP):

> *Always implement things when you actually need them, never when you just foresee that you will need them.*

YAGNI: Example

北京大学

Building a greeting feature, you speculatively add a strategy registry "just in case" multiple greeting styles are needed later:

```
interface GreetStrategy { String greet(String name); }

class GreetStrategyRegistry {
    private Map<String, GreetStrategy> strategies = new HashMap<>();
    public void register(String id, GreetStrategy s) { strategies.put(id, s); }
    public String greet(String id, String name) { return strategies.get(id).greet(name); }
}
```

The registry is never used with more than one strategy. The actual requirement was:

```
String greet(String name) { return "Hello, " + name + "!"; }
```

The unused abstraction must now be maintained, documented, and tested — at zero delivered value.

DRY (Don't Repeat Yourself)

David Thomas and Andrew Hunt in *The Pragmatic Programmer* [7]:

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

Violation: duplicated logic that can diverge

```
// In UserRegistrationService:
if (email == null
    || !email.contains("@"))
  throw new IllegalArgumentException(
    "Invalid email");

// In UserUpdateService (copy-pasted):
if (email == null
    || !email.contains("@"))
  throw new IllegalArgumentException(
    "Invalid email");
```

Better: single authoritative source

```
class EmailValidator {
  static void validate(String email) {
    if (email == null
        || !email.contains("@"))
      throw new IllegalArgumentException(
        "Invalid email");
  }
}

// In UserRegistrationService:
EmailValidator.validate(email);

// In UserUpdateService:
EmailValidator.validate(email);
```

KISS (Keep It Simple, Stupid!)

北京大学

Attributed to Kelly Johnson (Lockheed's "Skunk Works"), adopted in software engineering:

> *Most systems work best if they are kept simple rather than made complex.*

Unnecessarily clever:

```java
// Finding the larger of two numbers
int max = (a + b + Math.abs(a - b)) / 2;
```

Simple and readable:

```java
int max = Math.max(a, b);
```

Over-engineered for a simple need:

```java
interface SortStrategy {
    List<Integer> sort(List<Integer> data);
}
class SortContext {
    private SortStrategy strategy;
    public SortContext(SortStrategy s) {
        this.strategy = s;
    }
    public List<Integer> execute(
            List<Integer> data) {
        return strategy.sort(data);
    }
}
```

What was actually needed:

```java
Collections.sort(data);
```

References I

[1] Edsger W Dijkstra and Edsger W Dijkstra.
On the role of scientific thought.
Selected writings on computing: a personal perspective, pages 60–66, 1982.

[2] Micah Martin and Robert C Martin.
Agile principles, patterns, and practices in C.
Pearson Education, 2006.

[3] Robert C. Martin.
The single responsibility principle.
`https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html`, 2014.

[4] Bertrand Meyer.
Object-oriented software construction, volume 2.
Prentice hall Englewood Cliffs, 1997.

References II

北京大学

[5] Barbara H Liskov and Jeannette M Wing.
A behavioral notion of subtyping.
ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6):1811–1841, 1994.

[6] Linus Torvalds.
The mind behind linux.
https://www.ted.com/talks/linus_torvalds_the_mind_behind_linux, 2016.

[7] David Thomas and Andrew Hunt.
The Pragmatic Programmer: your journey to mastery.
Addison-Wesley Professional, 2019.