04834580 Software Engineering (Honor Track) 2025-26

Abstraction Mechanisms

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

Abstraction

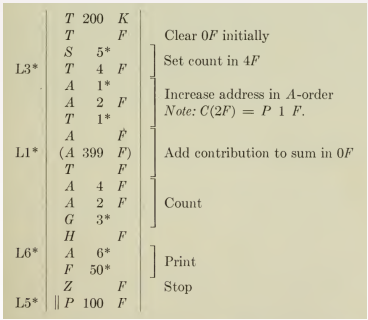
北京大学

Definition ([1])

- The act or process of leaving out of consideration one or more properties of a complex object so as to attend to others.
- A general concept formed by extracting common features from specific examples.

From “The Preparation of Programs for an Electronic Digital Computer” (1957) on “automatic programming” [2]:

> *There has been much controversy [...] Some went so far as to assert that programmers should be compelled to write order in a form as near as possible to that which they take inside the machine, and that attempts to make the machine assist with the clerical tasks of programming lead only to a wasteful dissipation of effort.*



A **procedure** (or function) is the oldest abstraction mechanism in programming:

- Hides a sequence of steps behind a name and a parameter list
- Callers depend only on *what* the procedure does, not *how*
- Enables reuse — call the same procedure from many places

The first subroutine libraries date back to the 1950s [2].

Procedural Abstraction: Before

北京大学

```c
int main(void) {
    double prices[] = {10.0, 25.5, 3.0, 42.0, 18.5};
    int n = 5;

    // compute the average price
    double sum = 0;
    for (int i = 0; i < n; i++)
        sum += prices[i];
    double avg = sum / n;

    // compute standard deviation
    double sq_sum = 0;
    for (int i = 0; i < n; i++)
        sq_sum += (prices[i] - avg) * (prices[i] - avg);
    double stddev = sqrt(sq_sum / n);

    printf("avg=%.2f  stddev=%.2f\n", avg, stddev);
}
```

Average and standard deviation are interleaved in one long function — hard to reuse or test independently.

Procedural Abstraction: After

北京大学

```c
double average(const double *a, int n) {
    double sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum / n;
}

double stddev(const double *a, int n) {
    double avg = average(a, n);
    double sq_sum = 0;
    for (int i = 0; i < n; i++)
        sq_sum += (a[i] - avg) * (a[i] - avg);
    return sqrt(sq_sum / n);
}

int main(void) {
    double prices[] = {10.0, 25.5, 3.0, 42.0, 18.5};
    printf("avg=%.2f  stddev=%.2f\n",
           average(prices, 5), stddev(prices, 5));
}
```

Each function has a single responsibility and can be reused and tested independently.

Information Hiding

David Parnas in "On the criteria to be used in decomposing systems into modules" [3]:

> *We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.*

A **module** groups related procedures and data, exposing only a public interface and hiding internal details.

Modules: Example (C Header)

北京大学

stats.h — public interface:

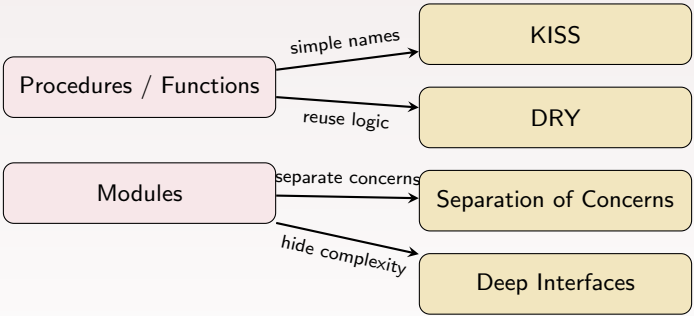
```c
#ifndef STATS_H
#define STATS_H

// Clients depend only on these
// declarations:
double average(const double *a,
               int n);
double stddev(const double *a,
              int n);

#endif
```

stats.c — hidden implementation:

```c
#include "stats.h"
#include <math.h>

double average(const double *a,
               int n) {
    double s = 0;
    for (int i = 0; i < n; i++)
        s += a[i];
    return s / n;
}

double stddev(const double *a,
              int n) {
    double avg = average(a, n);
    double sq = 0;
    for (int i = 0; i < n; i++)
        sq += (a[i]-avg)*(a[i]-avg);
    return sqrt(sq / n);
}
```

The implementation can change (e.g. use Welford's algorithm) without affecting any client code.

Procedural Abstraction and Design Principles



Functional Abstraction

John Backus in his 1977 Turing Award lecture [4]:

> *An alternative functional style of programming [...] can often be simpler and more powerful than conventional programs.*

Key idea: functions are **values** that can be passed to and returned from other functions.

- **First-class functions** — functions can be stored in variables and data structures
- **Higher-order functions** — functions that take or return other functions
- **Closures** — functions that capture their lexical environment

Higher-Order Functions: Abstracting Iteration

北京大学

Without higher-order functions:

```
# Filter even numbers
evens = []
for x in numbers:
    if x % 2 == 0:
        evens.append(x)

# Square each number
squares = []
for x in numbers:
    squares.append(x ** 2)

# Sum all numbers
total = 0
for x in numbers:
    total += x
```

With higher-order functions:

```
from functools import reduce

evens = list(
    filter(lambda x: x % 2 == 0,
           numbers))

squares = list(
    map(lambda x: x ** 2,
        numbers))

total = reduce(
    lambda a, x: a + x,
    numbers, 0)
```

The iteration pattern is abstracted away — only the *what* remains.

A higher-order function abstracts over *behavior*, making code open for extension without modification:

```
def process_records(records, transform, predicate):
    """Process records: filter by predicate, then apply transform."""
    return [transform(r) for r in records if predicate(r)]

# Different behaviors, same structure:
active_names = process_records(users,
    transform=lambda u: u.name,
    predicate=lambda u: u.is_active)

high_prices = process_records(products,
    transform=lambda p: f"{p.name}: ${p.price:.2f}",
    predicate=lambda p: p.price > 100)
```

Adding new processing logic requires no changes to `process_records` — only new lambdas.

A **closure** is a function together with its referencing environment [5]. It "closes over" variables from the enclosing scope:

```
def make_counter(start=0):
    count = start                    # captured by the closure
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

c1 = make_counter()
c2 = make_counter(10)
print(c1(), c1(), c1())    # 1 2 3
print(c2(), c2())           # 11 12
```

Each closure has its own private copy of `count` — encapsulation without classes.

Closures: Configurable Behavior

北京大学

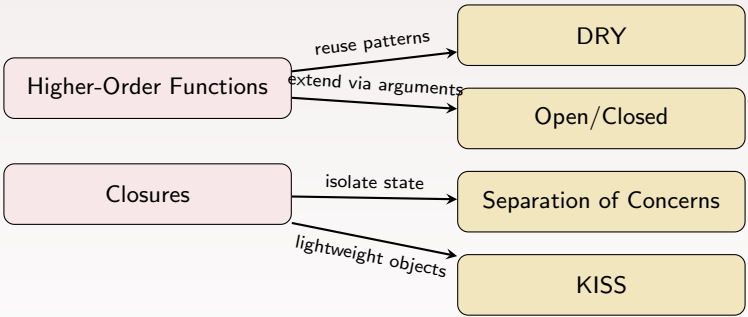
Closures can act as lightweight, stateful objects:

```
def make_validator(min_len, max_len, pattern):
    """Return a validation function with captured configuration."""
    import re
    regex = re.compile(pattern)
    def validate(value):
        if len(value) < min_len:
            return f"Too short (min {min_len})"
        if len(value) > max_len:
            return f"Too long (max {max_len})"
        if not regex.match(value):
            return f"Does not match {pattern}"
        return None  # valid
    return validate

validate_username = make_validator(3, 20, r'^[a-zA-Z0-9_]+$')
validate_email    = make_validator(5, 254, r'^[^@]+@[^@]+\.[^@]+$')
```

Configuration is captured once; validation functions are reused many times.

Functional Abstraction and Design Principles

北京大学



Object-Oriented Abstraction

Grady Booch [6]:

> *Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.*

Three pillars of OO abstraction:

- **Encapsulation** — bundle data with the operations that act on it
- **Polymorphism** — use a uniform interface for different underlying types
- **Inheritance** — define new abstractions by extending existing ones

Encapsulation: Hiding Representation

北京大学

Leaking internals:

```
class Temperature {
    public double celsius;
}

// Client code depends on
// representation:
Temperature t = new Temperature();
t.celsius = 100.0;
double f = t.celsius * 9/5 + 32;
```

Changing to Fahrenheit storage breaks every client.

Encapsulated:

```
class Temperature {
    private double celsius;

    public Temperature(double c) {
        this.celsius = c;
    }
    public double getCelsius() {
        return celsius;
    }
    public double getFahrenheit() {
        return celsius * 9/5 + 32;
    }
}
```

Internal representation can change freely.

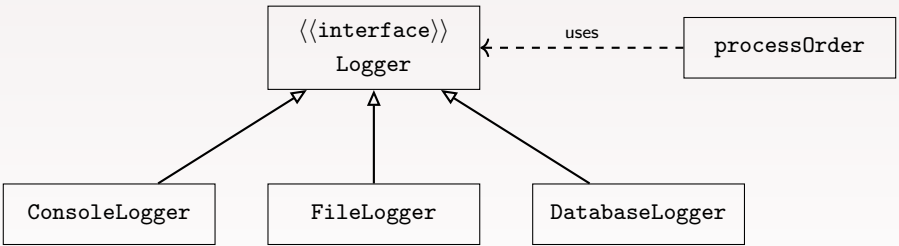
Polymorphism: Uniform Interface

北京大学

```java
interface Logger {
    void log(String message);
}

class ConsoleLogger implements Logger {
    public void log(String message) {
        System.out.println("[CONSOLE] " + message);
    }
}

class FileLogger implements Logger {
    private PrintWriter writer;
    public FileLogger(String path) throws IOException {
        this.writer = new PrintWriter(new FileWriter(path, true));
    }
    public void log(String message) { writer.println("[FILE] " + message); }
}

// Client code works with any Logger --- no conditionals needed:
void processOrder(Order order, Logger logger) {
    logger.log("Processing order " + order.getId());
    // ...
}
```

Polymorphism: Class Diagram

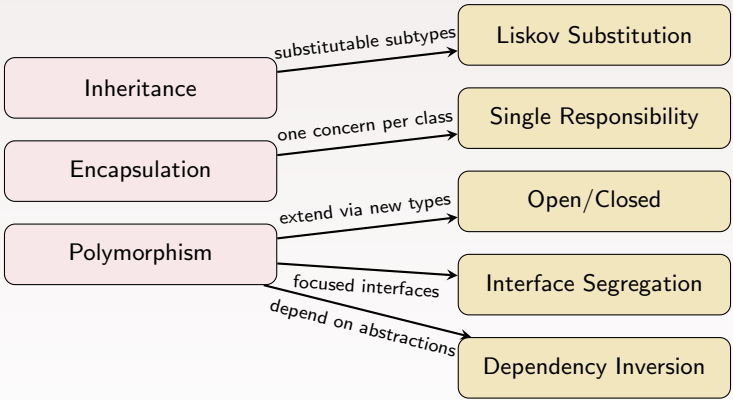
Adding a new logger (e.g., `DatabaseLogger`) requires only a new class — `processOrder` is never modified:



Inheritance: Reusing and Extending

北京大学

```
abstract class Shape {
    private String color;
    public Shape(String color) { this.color = color; }
    public String getColor() { return color; }

    public abstract double area();           // subclasses must implement

    public String describe() {               // shared behavior
        return color + " shape with area " + String.format("%.2f", area());
    }
}

class Circle extends Shape {
    private double radius;
    public Circle(String c, double r) { super(c); this.radius = r; }
    public double area() { return Math.PI * radius * radius; }
}

class Rectangle extends Shape {
    private double w, h;
    public Rectangle(String c, double w, double h) { super(c); this.w=w; this.h=h; }
    public double area() { return w * h; }
}
```


Inheritance
Encapsulation
Polymorphism
substitutable subtypes
one concern per class
extend via new types
focused interfaces
depend on abstractions
Liskov Substitution
Single Responsibility
Open/Closed
Interface Segregation
Dependency Inversion

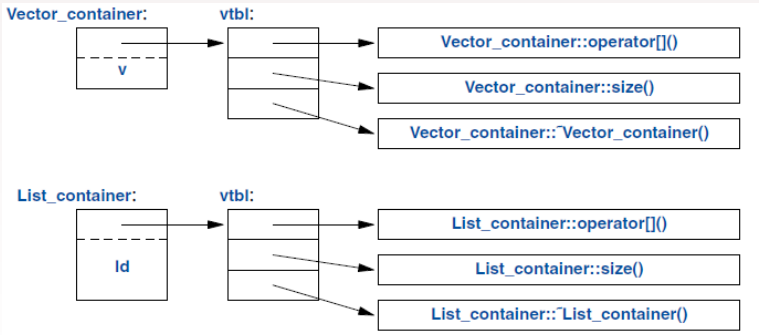
Definition

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time.

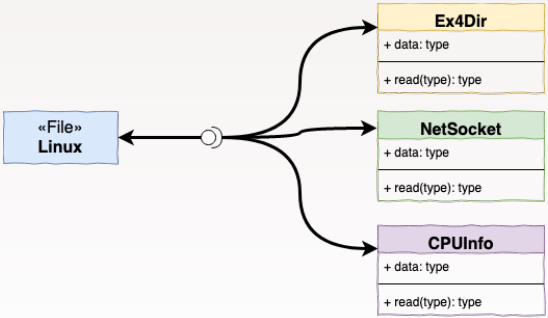
Dynamic Dispatch: Virtual Method Table

北京大学

```
class Container {
public:
    virtual double& operator[](int) = 0;
    virtual int size() const = 0;
    virtual ~Container() {}
};
```



```
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)
```



A generic struct is used to store pointers to implementation of I/O functions:

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
   //...
}
```

An implementation struct is filled with concrete code:

```
const struct file_operations ext4_dir_operations = {
  .llseek         = ext4_dir_llseek,
  .read         = generic_read_dir,
  //...
};
```

Cost of Abstraction

A program in Java:

```java
private static double average(int[] data) {
  int sum = 0;
  for (int i = 0; i < data.length; i++) {
    sum += data[i];
  }
  return sum * 1.0d / data.length;
}
```

Time: **30 ms**

The same program using Scala abstractions:

```scala
def average(x: Array[Int]): Double = {
  x.reduce(_ + _) * 1.0 / x.size
}
```

Time: **518 ms**

Leaky Abstractions

北京大学

Joel Spolsky: "All non-trivial abstractions, to some degree, are leaky." [7]

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str1 = "Hello";
    std::string result1 = str1 + ", World!";
    std::cout << "result1: " << result1 << std::endl;

    // ERROR: Invalid operands to binary +
    std::string result2 = "Hello" + ", World!";

    return 0;
}
```

- TCP guarantees reliable delivery — until a cable is cut and latency spikes
- SQL abstracts away storage — until a missing index makes a query 1000x slower
- Iterators abstract away collections — until modifying a collection during iteration crashes

Good abstractions *minimize* leaks; understanding the layer below helps you diagnose them when they occur.

A program with Scala abstractions:

```scala
def average(x: Array[Int]): Double = {
  x.reduce(_ + _) * 1.0 / x.size
}
```

Time: **518 ms**

The same program in Rust:

```rust
fn average(xs: &[i32]) -> f64 {
  xs.iter().fold(0, |x, y| x + y) as f64 / xs.len() as f64
}
```

Time: **18 ms**

A polymorphic operation implementing a trait:

```
trait Foo {
    fn method(&self) -> String;
}
impl Foo for u8 {
    fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
    fn method(&self) -> String { format!("string: {}", *self) }
}
```

The operation is chosen at compile time:

```
fn do_something<T: Foo>(x: T) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something(x);
    do_something(y);
}
```

The operation is chosen at runtime:

```
fn main() {
    let items: Vec<Box<dyn Foo>> = vec![
        Box::new(5u8),
        Box::new(String::from("Hello")),
    ];

    for item in items {
        item.method();
    }
}
```

Overloading in Java & C++

北京大学

```java
interface Animal {
    void makeSound();
}
class Dog implements Animal {
    void makeSound() {
        System.out.println("Woof! Woof!");
    }
}
class Cat implements Animal {
    void makeSound() {
        System.out.println("Meow! Meow!");
    }
}
class SoundPlayer {
    void playSound(Dog dog) {
        dog.makeSound();
    }
    void playSound(Cat cat) {
        cat.makeSound();
    }
}
```

```java
List<Animal> animals = new ArrayList<>();
animals.add(new Dog());
animals.add(new Cat());
SoundPlayer soundPlayer = new SoundPlayer();

for (Animal animal : animals) {
  animal.makeSound();
  soundPlayer.playSound(animal);
}
```

What is the difference?

References I

[1] Jeff Kramer.
Is abstraction the key to computing?
Communications of the ACM, 50(4):36–42, 2007.

[2] Maurice V Wilkes, David J Wheeler, Stanley Gill, and FJ Corbató.
The preparation of programs for an electronic digital computer, 1958.

[3] David Lorge Parnas.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM, 15(12):1053–1058, 1972.

[4] John Backus.
Can programming be liberated from the von Neumann style? a functional style and its algebra of programs.
Communications of the ACM, 21(8):613–641, 1978.

[5] Harold Abelson and Gerald Jay Sussman.
Structure and interpretation of computer programs.
MIT Press, 1996.

[6] Grady Booch, Robert A Maksimchuk, Michael W Engle, Bobbi J Young, Jim Connallen, and Kelli A Houston.
Object-oriented analysis and design with applications.
ACM SIGSOFT software engineering notes, 33(5):29–29, 2008.

[7] Joel Spolsky.
The law of leaky abstractions.
https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/, 2002.