YZM1022

Advanced Programming

Week 7: SOLID Principles and Testing

Instructor: Ekrem Çetinkaya

Date: 08.04.2026

Today's Agenda

SOLID Principles

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Clean Code & Testing

- Naming, small functions, DRY, comments
- pytest: tests, fixtures, parametrization
- How design and testability reinforce each other

Today's Running Example - Orders and Notifications

We will use **one example** across all five principles so you can focus on the *design decision*, not on learning a new problem each time.

- **Orders** are the first object type with many responsibilities: calculating totals, reserving inventory, charging payment, sending confirmation.
  - This makes them a perfect target for **Single Responsibility Principle** violations.
- **Notifications** are the second object type which can be email / SMS / push — a *family of interchangeable behaviors*.
  - Adding a new channel without touching existing code demonstrates **Open/Closed Principle**; depending on an abstraction instead of `EmailSender` directly demonstrates **Dependency Inversion Principle**.

> Responsibilities won't *disappear* when we refactor, they will *shift* to better places.

SOLID Principles

SOLID is a set of **guidelines** for object oriented design enabling easier changes, clearer tests, less accidental coupling.

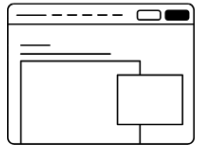
- Each letter gets: **rule -> violation -> refactor**

S ingle responsibility

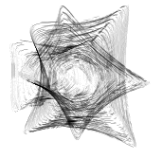
O pen/closed

L iskov substition

I nterface seggregation



D ependency inversion



S - Single Responsibility Principle

Rule: A class should have **one, and only one, reason to change**.

- This means a class must have a single, well-defined, and coherent job within the system.

Why it matters

- **Fewer surprise breakages:** When a requirement shifts (e.g., changing the database schema), you only touch the database class. Unrelated logic (like email formatting) remains safe.
- **Easier to test:** Smaller classes with a single focus require fewer test cases and minimal mocking.
- **High cohesion:** Code that changes together stays together, making it easier to read and maintain.
- **Clearer ownership:** Different teams or developers can work on different responsibilities without causing merge conflicts.

S - Single Responsibility Principle - Bad example 1

`UserManager` mixes user storage, email sending, DB writes, and report generation.

```python
class UserManager:
    def __init__(self): self.users = []

    def add_user(self, user): self.users.append(user)

    def send_email(self, user, message):
        print(f"Sending email to {user.email}: {message}")

    def save_to_database(self, user):
        print(f"Saving {user.name} to database")

    def generate_report(self):
        return f"Total users: {len(self.users)}"
```

Four reasons to change: user storage, email logic, DB logic, report format.

S - Single Responsibility Principle - Bad example 2

Log analysis doing too much in one class

```
class LogAnalyzer:
    def __init__(self, log_file_path):
        self.log_file_path = log_file_path
        self.logs = []

    def read_logs(self):           # file I/O concern
        with open(self.log_file_path) as f:
            self.logs = f.readlines()

    def parse_entries(self):       # parsing concern
        return [l.split(' - ') for l in self.logs]

    def filter_errors(self, entries):   # filtering concern
        return [e for e in entries if e[1] == 'ERROR']

    def send_alert(self, errors):  # notification concern
        print(f"Alert: {len(errors)} errors found")

    def generate_report(self, entries):  # presentation concern
        return "<html>" + "".join(f"<p>{e}</p>" for e in entries) + "</html>"
```

Five reasons to change in one class - a new email lib, a changed log format, a new report template, a new filter rule, or a new storage all modify this file.

S - Single Responsibility Principle - Good example 1

Each class has **one** reason to change.

```python
class UserRepository:
    def __init__(self):
        self.users = []
    def add_user(self, user): self.users.append(user)
    def get_all_users(self): return self.users

class EmailService:
    def send_email(self, user, message):
        print(f"Sending email to {user.email}: {message}")

class DatabaseService:
    def save_user(self, user):
        print(f"Saving {user.name} to database")

class UserReportGenerator:
    def __init__(self, repo: UserRepository):
        self.repo = repo
    def generate_report(self):
        return f"Total users: {len(self.repo.get_all_users())}"
```

S - Single Responsibility Principle - Good example 2

Same pipeline, one concern per class (read -> parse -> filter -> alert -> report).

```
class LogFileReader:
    def read_logs(self, path):
        return open(path).readlines()

class LogParser:
    def parse(self, raw):
        return [dict(zip(["ts","level","msg"], l.split(" - "))) for l in raw]

class LogFilter:
    def by_level(self, logs, level):
        return [l for l in logs if l["level"] == level]

class AlertService:
    def send_alert(self, errors):
        print(f"Alert: {len(errors)} errors")

class ReportGenerator:
    def html_report(self, logs):
        return "<html>" + "".join(f"<p>{l}</p>" for l in logs) + "</html>"
```

S - Single Responsibility Principle - Coordinator Usage

When we want to **combine** responsibilities, we can use coordinators.

- `LogAnalysisCoordinator` **sequences** the pipeline-it does not embed any of the rules.

```
class LogAnalysisCoordinator:
    def __init__(self, reader, parser, filter_svc, alert_svc, reporter):
        self.reader = reader
        self.parser = parser
        self.filter_svc = filter_svc
        self.alert_svc = alert_svc
        self.reporter = reporter

    def run(self, file_path):
        raw = self.reader.read_logs(file_path)
        entries = self.parser.parse_log_entries(raw)
        errors = self.filter_svc.filter_by_level(entries, 'ERROR')
        if errors:
            self.alert_svc.send_error_alert(errors)
        return self.reporter.generate_html_report(entries)
```

Practice

Identify violations of the Single Responsibility Principle in the following class and refactor it into multiple classes with single responsibilities.

```
class OrderProcessor:
    def __init__(self):
        self.orders = []

    def add_order(self, order):
        self.orders.append(order)

    def calculate_total(self, order):
        return sum(item.price * item.quantity for item in order.items)

    def send_confirmation_email(self, order):
        print(f"Email sent to {order.customer.email}")

    def update_inventory(self, order):
        for item in order.items:
            print(f"Reducing inventory for {item.name}")

    def process_payment(self, order, payment_method):
        print(f"Processing {payment_method} payment for ${self.calculate_total(order)}")
```

Hint: Identify at least 4 different responsibilities. Consider what would cause each method to change.

Solution - Split Services

Five responsibilities -> five classes, each with **one reason to change**.

```
class OrderRepository:
    def __init__(self): self.orders = []
    def add(self, order): self.orders.append(order)

class PriceCalculator:
    def total(self, order):
        return sum(i.price * i.quantity for i in order.items)

class EmailService:
    def confirm(self, order):
        print(f"Email sent to {order.customer.email}")

class InventoryService:
    def update(self, order):
        for i in order.items:
            print(f"Reducing stock for {i.name}")

class PaymentProcessor:
    def charge(self, order, method, amount):
        print(f"{method} charge: ${amount}")
```

Solution - Coordinating Service

One **orchestrator** with one job: sequence the steps.

```
class OrderProcessingService:
    def __init__(self, repo, calc, email, inv, payment):
        self.repo, self.calc = repo, calc
        self.email, self.inv, self.payment = email, inv, payment

    def process(self, order, method):
        self.repo.add(order)
        total = self.calc.total(order)
        self.payment.charge(order, method, total)
        self.inv.update(order)
        self.email.confirm(order)
        return order

svc = OrderProcessingService(
    OrderRepository(), PriceCalculator(),
    EmailService(), InventoryService(), PaymentProcessor()
)
```

O - Open/Closed Principle

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

- You should be able to add new behavior without editing stable, tested core code.

Why it matters

- **Zero regression risk:** If you don't touch existing code, you can't break it. New features are added by writing *new* code, not by altering old code.
- **Scalability:** The system can grow infinitely without the core logic becoming a massive, unreadable file.
- **How to achieve it:** Use **polymorphism**, **interfaces**, **composition**, or **strategy objects** instead of growing `if/elif` or `switch` chains.

O - Open/Closed Principle - Bad Example 1

Growing `if/elif` for each customer tier

```
class DiscountCalculator:
    def calculate_discount(self, customer_type, amount):
        if customer_type == "regular":
            return amount * 0.05
        elif customer_type == "premium":
            return amount * 0.10
        elif customer_type == "vip":
            return amount * 0.20
        # Adding new customer type requires modifying this method
        elif customer_type == "corporate":
            return amount * 0.15
        return 0
```

Every new tier requires editing (and retesting) this function.

O - Open/Closed Principle - Bad example 2

```
class PaymentProcessor:
    def process_payment(self, amount, payment_type, payment_data):
        if payment_type == "credit_card":
            card_number = payment_data['card_number']
            print(f"Processing ${amount} via credit card ending in {card_number[-4:]}")

        elif payment_type == "paypal":
            email = payment_data['email']
            print(f"Processing ${amount} via PayPal for {email}")

        elif payment_type == "bank_transfer":
            account = payment_data['account_number']
            print(f"Processing ${amount} via bank transfer to {account}")

        # Adding crypto requires modifying this method!
        elif payment_type == "bitcoin":
            wallet_address = payment_data['wallet_address']
            print(f"Processing ${amount} via Bitcoin to {wallet_address}")

        else:
            raise ValueError(f"Unsupported payment type: {payment_type}")
```

O - Open/Closed Principle - Good example 1

New tier = **new class**, not a new `elif`. (Connects to Strategy pattern.)

```python
class DiscountStrategy:
    def calculate(self, amount) -> float: ...

class RegularDiscount(DiscountStrategy):
    def calculate(self, a): return a * 0.05

class PremiumDiscount(DiscountStrategy):
    def calculate(self, a): return a * 0.10

class VIPDiscount(DiscountStrategy):
    def calculate(self, a): return a * 0.20

# ✓ New tier: zero changes to existing code
class CorporateDiscount(DiscountStrategy):
    def calculate(self, a): return a * 0.15

class DiscountCalculator:
    def __init__(self, strategy: DiscountStrategy):
        self.strategy = strategy
    def apply(self, amount):
        return self.strategy.calculate(amount)
```

O - Open/Closed Principle - Good example 2

New processor = **new class**, not edits to existing code.

```python
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount, payment_data): pass

class CreditCardProcessor(PaymentProcessor):
    def process_payment(self, amount, data):
        print(f"Card ending in {data['card_number'][-4:]}: ${amount}")

class PayPalProcessor(PaymentProcessor):
    def process_payment(self, amount, data):
        print(f"PayPal ({data['email']}): ${amount}")
```

O - Open/Closed Principle - Good example 2

```
# New method: add a class, touch nothing else
class BitcoinProcessor(PaymentProcessor):
    def process_payment(self, amount, data):
        print(f"Bitcoin ({data['wallet_address']}): ${amount}")

class PaymentService:
    def __init__(self, processor: PaymentProcessor):
        self.processor = processor
    def charge(self, amount, data):
        return self.processor.process_payment(amount, data)

# Usage
svc = PaymentService(CreditCardProcessor())
svc.charge(99, {"card_number": "4111111111111234"})
```

L - Liskov Substitution Principle

Rule: Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application.

- A **subtype must be usable anywhere** the base type is expected and should be **no surprises** for callers.

Why it matters

- **True polymorphism:** Polymorphism only works when **all** subtypes honor the original contract. If a caller has to check `isinstance(obj, SpecificSubclass)` to avoid a crash, LSP is violated.
- **Predictability:** Subclasses shouldn't weaken preconditions (require more from the caller) or strengthen postconditions (guarantee less to the caller).

L - Liskov Substitution Principle - Bad example 1

`Square` breaks `Rectangle`'s setter contract and callers can't treat them interchangeably.

```
class Rectangle:
    def __init__(self, w, h): self._w, self._h = w, h
    def set_width(self, w):  self._w = w
    def set_height(self, h): self._h = h
    def area(self): return self._w * self._h

class Square(Rectangle):
    def set_width(self, w):
        self._w = self._h = w   # silently changes both
    def set_height(self, h):
        self._w = self._h = h   # silently changes both

def test_rectangle(r):
    r.set_width(5); r.set_height(10)
    assert r.area() == 50  # passes for Rectangle, fails for Square
```

L - Liskov Substitution Principle - Bad example 2

Bird hierarchy where `fly()` lies

```
class Bird:
    def fly(self):
        print("Flying high in the sky")

class Eagle(Bird):
    def fly(self):
        print("Eagle soaring majestically")

class Penguin(Bird):
    def fly(self):
        raise NotImplementedError("Penguins cannot fly!")  # LSP violation

class Ostrich(Bird):
    def fly(self):
        raise NotImplementedError("Ostriches cannot fly!")  # LSP violation

def make_birds_fly(birds):
    for bird in birds:
        bird.fly()  # Will crash with penguins and ostriches

make_birds_fly([Eagle(), Penguin(), Ostrich()])  # Throws error
```

L - Liskov Substitution Principle - Good example 1

`Rectangle` and `Square` are **siblings** under `Shape` and neither inherits from the other.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self) -> float: pass

class Rectangle(Shape):
    def __init__(self, w, h): self._w, self._h = w, h
    def area(self): return self._w * self._h

class Square(Shape):
    def __init__(self, s): self._s = s
    def area(self): return self._s ** 2

def total_area(shapes: list[Shape]) -> float:
    return sum(s.area() for s in shapes)

# Both substitutable — client code only calls .area()
print(total_area([Rectangle(5, 10), Square(4)]))  # 66.0
```

L - Liskov Substitution Principle - Good example 2

Split *can fly* vs *cannot fly* so `move()` is always meaningful.

```
from abc import ABC, abstractmethod

class Bird(ABC):
    @abstractmethod
    def move(self) -> str: pass
    @abstractmethod
    def make_sound(self) -> str: pass

class FlyingBird(Bird):
    def move(self): return self.fly()
    @abstractmethod
    def fly(self) -> str: pass

class FlightlessBird(Bird):
    def move(self): return self.walk()
    @abstractmethod
    def walk(self) -> str: pass
```

L - Liskov Substitution Principle - Good example 2

```
class Eagle(FlyingBird):
    def fly(self): return "Eagle soaring majestically"
    def make_sound(self): return "Screech!"

class Penguin(FlightlessBird):
    def walk(self): return "Penguin waddling on ice"
    def make_sound(self): return "Squawk!"

def make_birds_move(birds: list[Bird]):
    for bird in birds:
        print(bird.move())

make_birds_move([Eagle(), Penguin()])
# Eagle soaring majestically
# Penguin waddling on ice
```

I - Interface Segregation Principle

Rule: Clients should not be forced to depend upon interfaces that they do not use.

- Prefer **small, role-based interfaces** over large, monolithic ones.

Why it matters

- **No fat interfaces:** Classes shouldn't be forced to implement methods they don't need (avoiding forced `NotImplementedError` stubs for unused capabilities).
- **Decoupling:** Changes to one part of a large interface won't force recompilation or updates in clients that only care about another part.
- **Easier testing:** When you need to mock a dependency, you only have to implement the 2 methods the client actually uses, rather than a monolithic interface with 15 methods.

I - Interface Segregation Principle - Bad example 1

Monolithic `Worker` ABC forces `Robot` to fake human biology

```python
from abc import ABC, abstractmethod

class Worker(ABC):
    @abstractmethod
    def work(self): pass
    @abstractmethod
    def eat(self): pass
    @abstractmethod
    def sleep(self): pass
    @abstractmethod
    def code(self): pass

class Robot(Worker):
    def work(self): print("working")
    def eat(self):  raise NotImplementedError("Robots don't eat!")
    def sleep(self): raise NotImplementedError("Robots don't sleep!")
    def code(self):  raise NotImplementedError("This robot can't code!")
```

I - Interface Segregation Principle - Bad example 2

`SimpleTextEditor` forced to add capabilities it cannot provide

```python
from abc import ABC, abstractmethod

class DocumentManager(ABC):
    @abstractmethod
    def open_document(self, path): pass
    @abstractmethod
    def save_document(self, doc, path): pass
    @abstractmethod
    def print_document(self, doc): pass
    @abstractmethod
    def scan_document(self): pass
    @abstractmethod
    def fax_document(self, doc, number): pass

class SimpleTextEditor(DocumentManager):
    def open_document(self, path): print(f"Opening: {path}")
    def save_document(self, doc, path): print(f"Saving: {path}")
    # Forced methods:
    def print_document(self, doc): raise NotImplementedError
    def scan_document(self): raise NotImplementedError
    def fax_document(self, doc, number): raise NotImplementedError
```

I - Interface Segregation Principle - Good example 1

```python
from typing import Protocol

class Workable(Protocol):
    def work(self) -> None: ...

class Eatable(Protocol):
    def eat(self) -> None: ...

class Programmable(Protocol):
    def code(self) -> None: ...

class Robot:
    def work(self): print("Robot working")
    # No eat/code – not forced to lie

class Human:
    def work(self): print("Human working")
    def eat(self):  print("Human eating")

class Developer(Human):
    def code(self): print("Developer coding")
```

I - Interface Segregation Principle - Good example 1

Clients depend only on what they need

```
# Only needs Workable — works for both Robot and Human
def make_workers_work(workers: list[Workable]):
    for worker in workers:
        worker.work()

# Only needs Eatable — Robot excluded at type-check time
def feed_workers(workers: list[Eatable]):
    for worker in workers:
        worker.eat()

robot = Robot()
alice = Developer()

make_workers_work([robot, alice])  # both work
feed_workers([alice])              # robot not accepted
```

I - Interface Segregation Principle - Good example 2

Six focused Protocols, each class implements only what it supports.

```python
from typing import Protocol

class DocumentReader(Protocol):
    def open_document(self, path) -> None: ...

class DocumentWriter(Protocol):
    def save_document(self, document, path) -> None: ...

class DocumentPrinter(Protocol):
    def print_document(self, document) -> None: ...

class DocumentScanner(Protocol):
    def scan_document(self) -> None: ...
```

I - Interface Segregation Principle - Good example 2

```python
# Only open + save — no force for print/scan/fax
class SimpleTextEditor:
    def open_document(self, path):
        print(f"Opening: {path}")
    def save_document(self, document, path):
        print(f"Saving to: {path}")

# Only print + scan
class MultiFunctionPrinter:
    def print_document(self, document): print("Printing")
    def scan_document(self): print("Scanning")

# Client depends only on what it actually calls
def process_docs(reader: DocumentReader, writer: DocumentWriter, files):
    for f in files:
        doc = reader.open_document(f)
        writer.save_document(doc, f"out_{f}")
```

D - Dependency Inversion Principle

Rule: High-level modules should not depend on low-level modules. Both should depend on **abstractions** (e.g., interfaces).

- Furthermore, abstractions should not depend on details. Details should depend on abstractions.

Why it matters

- **Decoupling policy from detail:** Your core business logic (high-level) shouldn't care whether data is saved to PostgreSQL or a text file (low-level). It just talks to a `Repository` interface.
- **Flexibility:** Swapping out a database, an email provider, or a third-party API becomes trivial because the core application relies on an abstraction, not the concrete implementation.
- **Testability:** It enables **dependency injection**, allowing you to easily pass mock objects during testing.

D - Dependency Inversion Principle - Bad example

Bad Example: Depending on Concretions

```python
class EmailService:
    def send_email(self, message):
        print(f"Sending email: {message}")

class SMSService:
    def send_sms(self, message):
        print(f"Sending SMS: {message}")

class NotificationManager:
    def __init__(self):
        self.email_service = EmailService()  # Hard dependency
        self.sms_service = SMSService()      # Hard dependency

    def send_notification(self, message, method):
        if method == "email":
            self.email_service.send_email(message)
        elif method == "sms":
            self.sms_service.send_sms(message)
```

D - Dependency Inversion Principle - Good example

Abstract `NotificationService` so the `NotificationManager` becomes dependent only on the interface.

```python
from abc import ABC, abstractmethod

class NotificationService(ABC):
    @abstractmethod
    def send(self, message): pass

class EmailService(NotificationService):
    def send(self, message): print(f"Email: {message}")

class SMSService(NotificationService):
    def send(self, message): print(f"SMS: {message}")

class SlackService(NotificationService):
    def send(self, message): print(f"Slack: {message}")
```

D - Dependency Inversion Principle - Good example

```
class NotificationManager:
    def __init__(self, services: list[NotificationService]):
        self.services = services   # injected — no hard coupling

    def notify(self, message):
        for svc in self.services:
            svc.send(message)

# Wiring happens at call site, not inside the class
manager = NotificationManager([EmailService(), SMSService()])
manager.notify("Server is down!")

class FakeNotifier(NotificationService):
    def __init__(self): self.sent = []
    def send(self, msg): self.sent.append(msg)
```

Practice

Apply the Dependency Inversion Principle to refactor the following class. The `OrderProcessor` class is tightly coupled to specific implementations of payment and shipping services.

```
class PayPalPayment:
    def process_payment(self, amount):
        print(f"Processing ${amount} via PayPal")

class FedExShipping:
    def ship_order(self, order):
        print(f"Shipping order {order.id} via FedEx")

class OrderProcessor:
    def __init__(self):
        self.payment_service = PayPalPayment()
        self.shipping_service = FedExShipping()

    def process_order(self, order):
        self.payment_service.process_payment(order.total)
        self.shipping_service.ship_order(order)
```

Solution - Abstractions + Injection

```
class PaymentService:
    def process_payment(self, amount): ...

class ShippingService:
    def ship_order(self, order): ...

class PayPalPayment(PaymentService):
    def process_payment(self, a): print(f"PayPal: ${a}")

class FedExShipping(ShippingService):
    def ship_order(self, o): print(f"FedEx: {o.id}")

class OrderProcessor:
    def __init__(self, pay: PaymentService, ship: ShippingService):
        self.pay, self.ship = pay, ship
    def process(self, order):
        self.pay.process_payment(order.total)
        self.ship.ship_order(order)

proc = OrderProcessor(PayPalPayment(), FedExShipping())
test = OrderProcessor(MockPayment(), MockShipping())
```

Solution - Factory (optional)

We can use **factory as wiring layer**. This will let us register once and swap freely later on.

```
class ServiceFactory:
    def __init__(self):
        self._payments = {}
        self._shippings = {}
    def register_payment(self, name, cls):
        self._payments[name] = cls
    def register_shipping(self, name, cls):
        self._shippings[name] = cls
    def create(self, pay, ship):
        return OrderProcessor(self._payments[pay](), self._shippings[ship]())

factory = ServiceFactory()
factory.register_payment("paypal", PayPalPayment)
factory.register_payment("stripe", StripePayment)
factory.register_shipping("fedex", FedExShipping)

proc = factory.create("paypal", "fedex")
```

Code Review Checklist - Design

Use this as a **quick check** before commiting your code. Not every point applies to every change, but they help catch structural issues early.

1. SOLID Principles:

- **S:** Does each class have only one reason to change?
- **O:** Can new functionality be added without modifying existing code?
- **L:** Can subclasses substitute the parent without surprises?
- **I:** Are interfaces small and focused on specific client needs?
- **D:** Do high-level modules depend on abstractions rather than concretions?

2. Clean Code:

- **Naming:** Are variables and functions meaningful, searchable, and pronounceable?
- **Functions:** Are they small and kept to a single level of abstraction?
- **Comments:** Do they explain *why* the code exists, rather than *what* it does?
- **DRY:** Is accidental duplication eliminated where appropriate?
- **Error Handling:** Are errors handled gracefully with meaningful messages?

Code Review Checklist - Testing & Quality

Use these criteria to ensure the code is safe to merge and maintainable long-term.

1. Testing:

- **Coverage:** Are critical paths and edge cases properly exercised?
- **Independence:** Can tests run in any order without shared state?
- **Clarity:** Does the test name read like a specification?
- **Speed:** Are the tests fast enough to run on every single commit?
- **Maintainability:** Will the tests be easy to update when requirements change?

2. General Quality:

- **Performance:** Are there any obvious bottlenecks or N+1 database queries?
- **Security:** Are all user inputs validated and sanitized?
- **Documentation:** Does complex or non-obvious logic have a brief explanation?
- **Dependencies:** Are new external dependencies minimal and well-justified?
- **Configuration:** Are magic numbers and strings replaced by named constants?

Clean Code Practices

Code is **read far more than it is written**. We must optimize for **clarity** and build systems out of **small, composable pieces**.

Core Principles of Clean Code:

- **Names:** Reveal intent and context.
- **Functions:** Do one thing and stay at one level of abstraction.
- **DRY (Don't Repeat Yourself):** Remove accidental duplication to prevent inconsistent updates.
- **Comments:** Explain the *why* behind a decision, only when the code cannot express it naturally.

Meaningful Names - Bad

Avoid: one-letter variables, cryptic abbreviations, `do_stuff`, `proc`.

```
# Unclear variable names
d = 10           # days? distance? data?
usr_lst = []     # user list?
calc_amt = lambda x, y: x * y * 0.1

# Unclear function names
def do_stuff(data):
    return [x for x in data if x > 0]

def proc(u):     # process user?
    u.active = True
    u.last_login = datetime.now()
```

Meaningful Names - Good

Goal: a reader understands intent without looking at the body.

```
# Descriptive variables
days_since_last_update = 10
active_users = []
calculate_discount_amount = lambda price, qty: price * qty * 0.1

# Descriptive function names
def filter_positive_numbers(numbers):
    return [n for n in numbers if n > 0]

def activate_user_account(user):
    user.active = True
    user.last_login = datetime.now()
```

Small Functions - Bad

Rule: either **orchestrate** (`validate -> save -> notify`) **or** do low-level steps-not both.

```python
def process_user_registration(user_data):
    # low-level validation
    if not user_data.get('email'):
        raise ValueError("Email is required")
    if '@' not in user_data['email']:
        raise ValueError("Invalid email")

    # mid-level DB work
    user = User(email=user_data['email'], name=user_data['name'])
    db.session.add(user)
    db.session.commit()

    # high-level notification
    email_body = f"Welcome {user.name}! Your account has been created."
    send_email(user.email, "Welcome!", email_body)
    return user
```

Small Functions - Good

```
def process_user_registration(user_data):
    validate_user_data(user_data)           # delegates details
    user = create_user_account(user_data)
    send_welcome_email(user)
    return user

def validate_user_data(user_data):
    if not user_data.get('email'):
        raise ValueError("Email is required")
    if '@' not in user_data['email']:
        raise ValueError("Invalid email")

def create_user_account(user_data):
    user = User(email=user_data['email'], name=user_data['name'])
    db.session.add(user)
    db.session.commit()
    return user

def send_welcome_email(user):
    body = f"Welcome {user.name}! Your account has been created."
    send_email(user.email, "Welcome!", body)
```

Comments - Bad

Comments should not narrate what the code already says. Use naming + structure for that.

- **Comments** should be used to answer *why*, not *what*.

```
# Increment i by 1       <- obvious
i += 1

# Check if user is active   <- obvious
if user.is_active:
    # Send notification to user   <- obvious
    send_notification(user)

def calculate_price(base_price, discount_rate):
    # Multiply base price by discount rate   <- obvious
    discount = base_price * discount_rate
    # Subtract discount from base price      <- obvious
    return base_price - discount
```

Comments - Good

```
# Business rule: Premium users get free shipping on orders over $50
if user.is_premium and order.total > 50:
    order.shipping_cost = 0

def calculate_hash(data):
    # SHA-256 required by PCI DSS compliance for financial transactions
    return hashlib.sha256(data.encode()).hexdigest()

# TODO: Optimize this query when user base exceeds 1M records
def get_user_statistics():
    return db.session.query(User).all()

# HACK: API v1 returns timestamps in a non-standard format
# Remove when API v2.0 is released (Q2 2024)
timestamp = parse_weird_timestamp_format(api_response['date'])
```

Comments

Self-documenting code needs no comment; adding one only adds noise to maintain.

```
def calculate_discounted_price(original_price, discount_percentage):
    discount_amount = original_price * (discount_percentage / 100)
    return original_price - discount_amount

def is_weekend(date):
    return date.weekday() >= 5

def send_welcome_email(user):
    template = get_email_template('welcome')
    email_service.send(user.email, template.render(user=user))
```

Introduction to Testing with `pytest`

`pytest` is the industry standard testing framework for Python. It offers minimal boilerplate, rich failure introspection, and a powerful fixture system, making it the best default for modern Python projects.

Why choose `pytest`

- **No Boilerplate:** Unlike the built-in `unittest` module, you don't need to create classes or inherit from `TestCase`. Tests are just **ordinary functions** prefixed with `test_`.
- **Plain Asserts:** You use the standard Python `assert` keyword (e.g., `assert result == 5`) instead of learning custom methods like `assertEqual` or `assertTrue`. `pytest` automatically inspects the variables when an assertion fails to give you detailed error messages.
- **Powerful Fixtures:** A modular and scalable way to set up and tear down test state (like database connections or mock servers) without cluttering your test logic.
- **Design Synergy:** Good design (specifically **SOLID** and **DIP**) makes **isolated** tests much easier to write. If your classes depend on abstractions rather than concretions, you can easily pass mock objects into them during testing.

Why Test?

Testing is not just about finding bugs; it's about building a safety net that allows you to move fast and change code without fear.

Benefits of Automated Testing:

- **Confidence:** You know your code works correctly today, and will continue to work tomorrow.
- **Living Documentation:** Tests provide executable examples of how your code is meant to be used.
- **Refactoring Safety:** You can clean up or optimize code without fear of breaking existing functionality.
- **Bug Prevention:** Catch edge cases and regressions early in the development cycle, before they reach production.
- **Design Feedback:** If a unit of code is hard to test, it's usually a sign that the design is flawed (e.g., too tightly coupled).

Types of Tests:

- **Unit Tests:** Test individual functions or methods in isolation (fast, highly specific).
- **Integration Tests:** Test how multiple components or systems work together (e.g., database queries).
- **End-to-End (E2E) Tests:** Test complete user workflows from the UI down to the database (slow, broad).

Test Structure - Arrange, Act, Assert

A well-structured test follows the **AAA pattern**. This keeps your tests readable and ensures they only verify one specific behavior at a time.

- **Arrange:** Set up the initial state, create objects, and prepare the data needed for the test.
- **Act:** Execute the specific function or method you are testing.
- **Assert:** Verify that the outcome matches your expectations

```
def test_calculate_discount():
    # Arrange
    price, pct = 100, 20
    # Act
    result = calculate_discount(price, pct)
    # Assert
    assert result == 80

def test_user_activation():
    user = User(name="John", email="j@j.com", active=False)  # Arrange
    activate_user(user)                                        # Act
    assert user.active is True                                 # Assert
    assert user.activation_date is not None

def test_empty_cart():
    assert ShoppingCart().calculate_total() == 0
```

Test Organization

- **Mirror Source Layout:** Keep tests in a dedicated `tests/` directory at the root of your project, structured identically to your `src/` directory.
- **Shared Setup:** Place shared test data, mock objects, and database connections in a `conftest.py` file. `pytest` automatically discovers fixtures defined here and makes them available to all tests in that directory.

```
project/
├── src/
│   ├── calculator/
│   │   ├── basic_operations.py
│   │   └── advanced_operations.py
│   └── user_management/
│       ├── models.py
│       └── services.py
├── tests/
│   ├── conftest.py              # Shared fixtures
│   ├── unit/
│   │   ├── test_calculator/
│   │   │   ├── test_basic_operations.py
│   │   │   └── test_advanced_operations.py
│   │   └── test_user_management/
│   │       ├── test_models.py
│   │       └── test_services.py
│   └── integration/
│       └── test_user_workflow.py
```

Naming Conventions

Test names are **executable specifications**.

- When a test fails in CI/CD, the name is often the only context you get.
- Write test names like a requirement sentence.

Standard `pytest` Discovery Rules:

- **Files:** Must start with `test_` (e.g., `test_calculator.py`) or end with `_test.py`.
- **Functions:** Must start with `test_` (e.g., `test_calculate_discount_with_valid_code`).
- **Classes:** Must start with `Test` (e.g., `TestDiscountCalculator`).

Best Practice for Naming Functions:
Use the format `test_<method_name>_<expected_behavior>_<under_condition>`.

Naming Conventions

```
# Good - reads like a spec
def test_calculator_adds_positive_numbers_correctly():
    assert add(2, 3) == 5

def test_user_login_with_invalid_password_raises_authentication_error():
    with pytest.raises(AuthenticationError):
        login("user@example.com", "wrong_password")

# Bad - tells you nothing about the scenario
def test_add():
    assert add(2, 3) == 5

def test_login():
    with pytest.raises(AuthenticationError):
        login("user@example.com", "wrong_password")
```

`pytest` Basics - Test Functions

Unlike older frameworks, `pytest` doesn't require you to learn custom assertion methods.

- You just write a normal Python function starting with `test_` and use the built-in `assert` keyword.
- **Introspection:** When a test fails, `pytest` rewrites the Python bytecode to show you exactly what the variables were at the time of failure, making debugging significantly easier.
- **Exceptions:** Use `pytest.raises(ExceptionType)` to verify that your code correctly throws an error under specific conditions.

```
def test_addition():
    result = add(2, 3)
    assert result == 5

def test_string_concatenation():
    result = concatenate("Hello", "World")
    assert result == "HelloWorld"

def test_list_contains_item():
    items = ["apple", "banana", "cherry"]
    assert "banana" in items

def test_exception_handling():
    with pytest.raises(ValueError):
        divide_by_zero(10, 0)
```

`pytest` Basics - Fixtures

Fixtures are reusable setup helpers injected directly into test functions. They replace the `setUp` and `tearDown` methods from older frameworks like `unittest`.

- **Dependency Injection:** You request a fixture simply by adding it as a parameter to your test function. `pytest` handles the instantiation automatically.
- **Setup and Teardown:** By using the `yield` keyword instead of `return`, a fixture can run setup code before the test, pause while the test executes, and then run teardown code (like closing a database connection) after the test finishes.

`pytest` Basics - Fixtures

```python
import pytest

@pytest.fixture
def sample_user():
    return User(name="Test User", email="test@example.com")

@pytest.fixture
def empty_database():
    db.create_all()
    yield db          # test runs here
    db.drop_all()     # teardown runs after

def test_user_creation(sample_user):
    assert sample_user.name == "Test User"
    assert sample_user.email == "test@example.com"

def test_db_insert(empty_database):
    empty_database.session.add(User(name="John", email="j@j.com"))
    empty_database.session.commit()
    found = User.query.filter_by(email="j@j.com").first()
    assert found.name == "John"
```

`pytest` - Sample Output

One of `pytest`'s strongest features is its clear and actionable console output.

Passing Run (`-q` or `--quiet` mode):
Provides a minimal summary, good for CI/CD pipelines when everything is fine.

```
$ pytest -q tests/test_wallet.py

tests/test_wallet.py::test_deposit_increases_balance PASSED
tests/test_wallet.py::test_withdraw_insufficient_funds_raises PASSED
2 passed in 0.04s
```

Failing Run (Default or `-v` verbose mode):
When a test fails, `pytest` automatically provides a detailed traceback and a visual diff of the variables involved, showing exactly *where* and *why* the assertion failed.

```
$ pytest tests/test_wallet.py::test_withdraw_insufficient_funds_raises -v
...
E   Failed: DID NOT RAISE <class 'ValueError'>
FAILED tests/test_wallet.py::test_withdraw_insufficient_funds_raises
1 failed in 0.06s
```

`pytest` Fixtures - Function and Class Scope

Fixtures can be scoped to control how often they are executed. Choosing the right scope balances **test isolation** against **execution speed**.

- `function` **scope (Default):** The fixture is executed once per test function. This provides the best isolation, ensuring tests don't accidentally share state and interfere with each other.
- `class` **scope:** The fixture is executed once per test class. All methods in the class share the same setup. This is faster for expensive setups, but you must be careful not to mutate shared state.

`pytest` Fixtures - Function and Class Scope

```python
import pytest
from database import DatabaseConnection
from email_service import EmailService

@pytest.fixture(scope="function")
def clean_database():
    db = DatabaseConnection(":memory:")
    db.create_tables()
    yield db
    db.close()

@pytest.fixture(scope="class")
def email_service():
    svc = EmailService()
    svc.configure_test_mode()
    yield svc
    svc.cleanup()

class TestUserOperations:
    def test_create_user(self, clean_database):
        user = create_user(clean_database, "test@example.com")
        assert user.email == "test@example.com"
```

`pytest` Fixtures - Module and Session Scope

For very expensive operations (like spinning up a Docker container or loading a large dataset), you can expand the scope further.

- **`module` scope:** Executed once per test file (`.py`). Useful for setting up resources that all tests in a specific file need, like a temporary directory or a mock server.
- **`session` scope:** Executed exactly once per `pytest` invocation. Ideal for global, read-only configurations or heavy infrastructure setup (like establishing a database connection pool).

`pytest` Fixtures - Module and Session Scope

```python
import pytest, tempfile, os

@pytest.fixture(scope="module")
def temp_directory():
    with tempfile.TemporaryDirectory() as d:
        yield d          # cleaned up automatically

@pytest.fixture(scope="session")
def app_config():
    return {"TESTING": True, "DATABASE_URL": "sqlite:///:memory:"}

def test_file_round_trip(temp_directory):
    path = os.path.join(temp_directory, "test.txt")
    write_file(path, "Hello World")
    assert read_file(path) == "Hello World"

def test_uses_test_db(app_config):
    assert app_config["TESTING"] is True
```

Testing Patterns - Data-Driven Testing with Parametrize

Writing separate test functions for every possible input combination leads to massive code duplication. `pytest` solves this with **Data-Driven Testing**.

`@pytest.mark.parametrize` allows you to define one test body and run it multiple times with different sets of inputs and expected outputs which makes it easy to test edge cases without cluttering your test suite.

Testing Patterns - Data-Driven Testing with Parametrize

```python
import pytest

@pytest.mark.parametrize("price, discount, expected", [
    (100, 10,  90),
    (50,  20,  40),
    (200, 25, 150),
    (0,   10,   0),
])
def test_calculate_discount(price, discount, expected):
    assert calculate_discount(price, discount) == expected

@pytest.mark.parametrize("email", [
    "valid@example.com", "user.name@domain.co.uk",
])
def test_valid_emails(email):
    assert is_valid_email(email) is True

@pytest.mark.parametrize("Bad", ["nodomain", "@nodomain", "user@", ""])
def test_invalid_emails(Bad):
    assert is_valid_email(Bad) is False
```

Parametrized Testing - Advanced Patterns

You can combine positive and negative test cases in a single parametrized test by passing a boolean flag (e.g., `should_pass`) and using conditional logic inside the test.

```
import pytest

@pytest.mark.parametrize("username, email, should_pass", [
    ("alice",  "alice@example.com", True),
    ("",       "alice@example.com", False),   # empty name
    ("alice",  "not-an-email",      False),   # Bad email
    ("alice!", "alice@example.com", False),   # invalid chars
])
def test_user_registration(username, email, should_pass):
    if should_pass:
        user = register_user(username, email)
        assert user.username == username
    else:
        with pytest.raises((ValueError, ValidationError)):
            register_user(username, email)
```

Parametrized Testing - Advanced Patterns

Using `indirect=True`:
Sometimes, the parameters aren't meant for the test function itself, but for a *fixture* the test uses. Setting `indirect=True` tells `pytest` to pass the parameter to the fixture first, allowing dynamic setup (e.g., testing against multiple database engines).

```
@pytest.mark.parametrize("db", ["sqlite", "postgresql"], indirect=True)
def test_insert(db):
    assert db.insert({"name": "T"}).id is not None
```

Testing Exceptions with `pytest.raises`

Testing that your code *fails correctly* (negative testing) is just as important as testing that it succeeds.

- **Basic Exception Testing:** Use the `with pytest.raises(ExceptionType):` context manager to assert that a specific block of code raises the expected error.
- **Matching Error Messages:** Use the `match="regex"` parameter to ensure the exception message contains specific text, preventing false positives.
- **Inspecting Exceptions:** Use `as exc_info` to capture the exception object and assert against its custom attributes.

Testing Exceptions with `pytest.raises`

```python
import pytest

def test_divide_by_zero():
    with pytest.raises(ZeroDivisionError):
        divide(10, 0)

def test_invalid_age_message():
    with pytest.raises(ValueError, match="Age must be between 0 and 150"):
        create_user("John", age=-5)

def test_auth_error_attributes():
    with pytest.raises(AuthenticationError) as exc_info:
        login("Bad@user.com", "wrong")
    assert exc_info.value.error_code == "AUTH_FAILED"
    assert exc_info.value.retry_allowed is False

def test_valid_input_does_not_raise():
    result = process_valid_data("Good_input")   # no raises = test passes
    assert result is not None
```

Testing Exceptions - Multiple Types and Async

`pytest` handles complex exception scenarios seamlessly:

- **Multiple Exception Types:** You can pass a tuple of exception classes if a function might legitimately raise one of several errors.
- **Inspecting Complex Errors:** For validation libraries (like Pydantic), you can capture the error and inspect the nested validation details.
- **Async Testing:** `pytest` works perfectly with `asyncio` (via the `pytest-asyncio` plugin), allowing you to test timeouts and async exceptions.

```python
# Accept either exception type
def test_file_errors():
    with pytest.raises((FileNotFoundError, PermissionError)):
        read_protected_file("/nonexistent/path")

# Collect multiple validation errors at once
def test_multiple_validation_errors():
    with pytest.raises(ValidationError) as exc_info:
        validate_user_data({"username": "", "email": "Bad", "age": -1})
    errors = exc_info.value.errors
    assert {"username", "email", "age"} <= set(errors)

# Async code
@pytest.mark.asyncio
async def test_async_timeout():
    with pytest.raises(asyncio.TimeoutError):
        await fetch_with_timeout(timeout=0.001)
```

Test Coverage

Test Coverage measures the percentage of your source code that is executed when your test suite runs. It helps identify untested branches and dead code.

- **Branch Coverage:** It's not enough to just execute a line of code; you need to ensure all logical paths (`if` / `else`) are tested.
- **The Metric, Not the Goal:** 100% coverage doesn't guarantee your code is bug-free, but low coverage guarantees you have blind spots.
- **Tooling:** Use the `pytest-cov` plugin to generate detailed coverage reports right in your terminal or as HTML.

Test Coverage

```
def process_payment(amount, method, user):
    if amount <= 0:                    # branch A (positive check)
        raise ValueError("Must be positive")
    if user.is_premium:                # branch B (discount)
        amount *= 0.9
    if method == "card":
        return charge_card(amount, user.card)
    else:
        raise ValueError(f"Unknown: {method}")  # branch C

# Tests that hit all branches
def test_negative():
    with pytest.raises(ValueError): process_payment(-1, "card", u)

def test_premium_discount():
    r = process_payment(100, "card", premium_user)
    assert r.charged == 90

def test_unknown_method():
    with pytest.raises(ValueError): process_payment(100, "btc", u)
```

```
pytest --cov=src --cov-report=term-missing tests/
# src/payments.py  18 stmts  2 miss  Branch 8  BrPart 1  Cover 85%
```

Practice

Write tests for `Calculator`. Cover each operation, edge cases, errors, and use `parametrize`.

```
class Calculator:
    def add(self, a, b): return a + b
    def subtract(self, a, b): return a - b
    def multiply(self, a, b): return a * b
    def divide(self, a, b):
        if b == 0: raise ValueError("Cannot divide by zero")
        return a / b
    def power(self, base, exp): return base ** exp
```

Solution - Fixture + Add/Subtract

```python
import pytest

@pytest.fixture
def calc():
    return Calculator()

@pytest.mark.parametrize("a, b, expected", [
    (2, 3, 5), (0, 5, 5), (-2, 3, 1), (-2, -3, -5),
])
def test_addition(calc, a, b, expected):
    assert calc.add(a, b) == expected

@pytest.mark.parametrize("a, b, expected", [
    (5, 3, 2), (0, 5, -5), (-2, -3, 1),
])
def test_subtraction(calc, a, b, expected):
    assert calc.subtract(a, b) == expected
```

Solution - Divide + Power + Error

```python
@pytest.mark.parametrize("a, b, expected", [
    (6, 2, 3.0), (7, 2, 3.5), (-6, 3, -2.0),
])
def test_division(calc, a, b, expected):
    assert calc.divide(a, b) == expected

def test_divide_by_zero(calc):
    with pytest.raises(ValueError, match="Cannot divide by zero"):
        calc.divide(5, 0)

def test_power(calc):
    assert calc.power(2, 3) == 8
    assert calc.power(2, 0) == 1
    assert calc.power(2, -1) == pytest.approx(0.5)
```

Testing Best Practices

Habits for Maintainable Tests:

- **Organize Logically:** Mirror your `src/` directory structure in your `tests/` directory. Keep shared fixtures centralized in `conftest.py`.
- **Enforce Isolation:** Tests must not depend on the order they are run. Use fixtures to guarantee a fresh, known state before every test execution.
- **Test Behavior, Not Implementation:** Focus on the inputs and outputs (the contract) rather than the internal logic. This allows you to refactor the implementation without breaking the tests.
- **Descriptive Naming:** Name tests so that when they fail in a CI/CD pipeline, the failure message reads like a broken specification.

A Healthy Testing Workflow:

1. **Red-Green-Refactor:** Write a failing test first (Red), write the minimal code to make it pass (Green), then clean up the code (Refactor).
2. **Prioritize Paths:** Start by testing the *Happy Path* (normal operation), then systematically test edge cases (boundaries), and finally verify error handling (exceptions).
3. **Continuous Execution:** Run `pytest` frequently during development, and automate it to run on every commit (via pre-commit hooks) and every pull request (via CI/CD).

Thank You!

Contact Information

- **Email:** ekrem.cetinkaya@yildiz.edu.tr

- **Office Hours:** Wednesday 13:30-15:30 - Room C-120
- **Book a slot before coming:** Booking Link
- **Course Repository:** GitHub

Next Week

Week 8: Midterm