T05 System Design III

Security, Reliability, and Availability

Prof. Pramod Bhatotia
Systems Research Group
https://dse.in.tum.de/

Tutorial outline

- **Part I: Lecture summary**
  - **Q&A for the lecture material**
- **Part II:** Programming basics
- **Part III:** Homework programming exercises (Artemis)

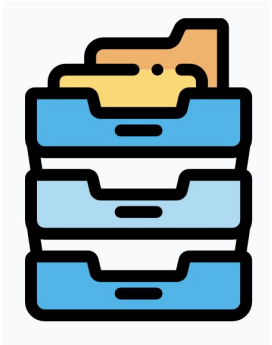
Lecture overview

- **Part I:** Security
  - Security engineering
  - Software security in the cloud
- **Part II:** Reliability and availability
  - Single-node reliable systems and associated issues
  - Replication as the general recipe for fault-tolerance
  - Fault-tolerance for stateful services
- **Part III:** Pattern implementation
  - Adapter pattern
  - Observer pattern
  - Strategy pattern

Security CIA properties

- **Confidentiality**
  - Confidentiality of a service **limits access of information to privileged entities**
  - Confidentiality requires **authentication and access rights** according to a policy
- **Integrity**
  - Integrity of a service **limits the modification of information to privileged entities** (or an attacker cannot modify protected data)
  - Integrity property also requires **authentication and access rights** according to a policy
- **Availability**
  - Availability of a service guarantees that the **service remains accessible** (or, availability prohibits an attacker from hindering computation
  - The availability property guarantees that **legitimate uses of the service remain possible**

Security design principles



#1:

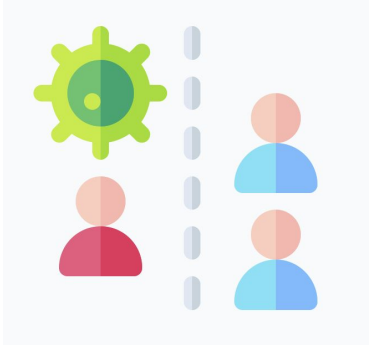
Compartmentalization

Break a complex system into small components

#2:

Principle of least privileges

A component has the least amount of privileges needed to function



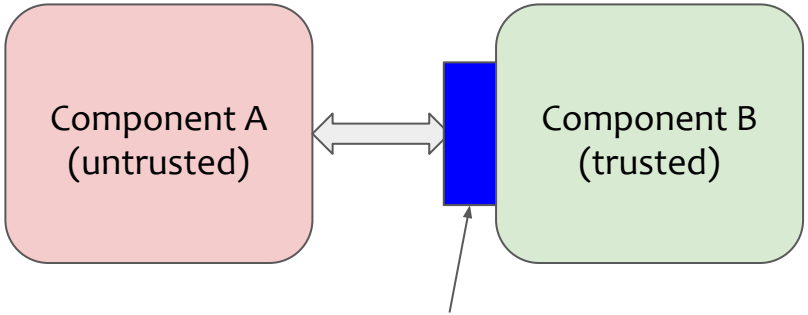
#3:

Isolation via privilege mediation

Component separation and confinement of their interactions to a well-defined API

A high-level recipe for secure system design

- Break system into compartments
- Ensure each compartment is isolated
- Ensure each compartment runs with least privilege
- Treat compartment interface as the trust boundary



Always aim to reduce the **trusted computing base (TCB)** → **Lower TCB is better!**

Authentication vs authorization

- **Authentication**
  - **Determining the identity of a user**
  - May also involve verification of IP address, machine, time, etc…
  - Determine whether a user is allowed to access the system at all
  - Used for audit, not just authorization
- **Authorization**
  - Assuming identity of user is known, **determine whether some specific action is allowed**
  - Access control allows authorization

Two mechanisms for access control

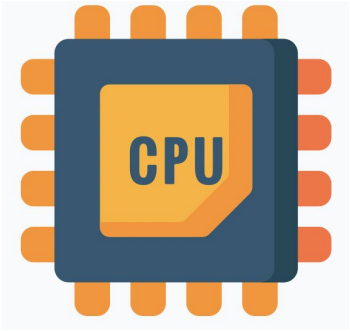
- **Access control lists (ACLs)**
  - ACL is like a guest list with the names
  - A list of permissions associated with a system resource (object)
  - An ACL specifies which users or system processes are granted access to objects and what operations are allowed on given objects

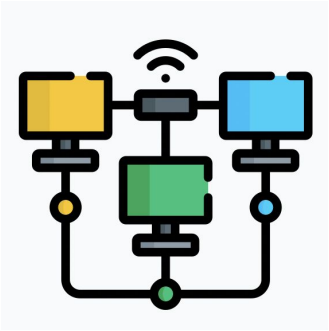
- **Capabilities**
  - Capabilities are like a ticket
  - Unique, unforgeable token that gives a process permission to access an entity or object in system
  - Core operations: *Delegate*, *Revoke*, *Grant*

Challenges of system security in the cloud

- **Outsourced infrastructure**
  - Third-party infrastructure
- **Multitenancy**
  - Computing infrastructure is shared across multiple tenants
- **Identity management/authorization**
  - Require establishing trust
- **Compliance (e.g., GDPR)**
  - Data and code might be handled in a different administrative jurisdiction
- **Misconfiguration and software bugs**
  - Large trusted computing base (infrastructure) operated/developed by multiple third-parties

Secure systems stack in the cloud





Compute

(Data in use)

Protection the code and data, while computing on the CPU, accessing over the RAM

- Virtualization
- Confidential computing

Network

(Data in motion)

Protecting the data being transmitted over the untrusted network connection

- Service authenticity (e.g., attestation
- Secure communication (e.g., TLS)
- Service availability (e.g., protect from DOS attacks)

Storage

(Data at rest)

Protecting the data storage on untrusted persistent storage, e.g., disk/SSDs

- CIA properties
- Freshness property

Lecture overview

- ~~**Part I:** Security~~
  - ~~Security engineering~~
  - ~~Software security in the cloud~~
- **Part II:** Reliability and availability
  - Single-node reliable systems and associated issues
  - Replication as the general recipe for fault-tolerance
  - Fault-tolerance for stateful services
- **Part III:** Pattern implementation
  - Adapter pattern
  - Observer pattern
  - Strategy pattern

Faults

- **Faults** are deviation from the expected behavior
- **Faults happen** due to a variety of factors:
  - Hardware failure
  - Software bugs
  - Operator errors/mis-configurations
  - Network errors/outages
- **Faults are the norm, not an exception!**
- Types of faults:
  - Transient faults → E.g., bit flips
  - Permanent faults
- Potential consequences:
  - Fail-stop (crash faults) → Process crashes
  - Byzantine faults → Arbitrary behavior, e.g., data corruption, security attacks

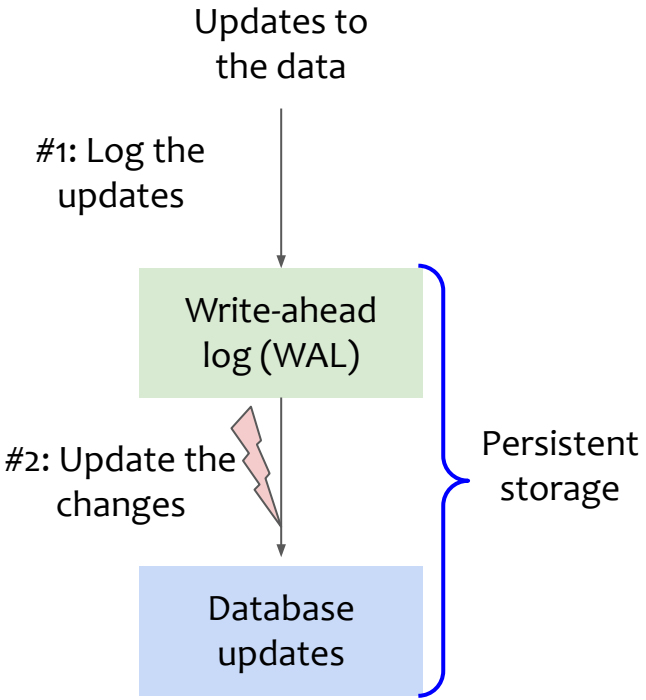
We need to design fault-tolerant systems!

Roadmap: How to make systems fault-tolerant?

1. Single-node fault-tolerant systems
   - Write-ahead logging for system reliability
   - Issues with a single-node fault tolerance approach
2. Replication as the general recipe
   - Replication for stateless services
   - Issues with replication for stateful services
3. Replication for stateful services
   - Primary-backup replication
   - State machine replication

Write-ahead logging

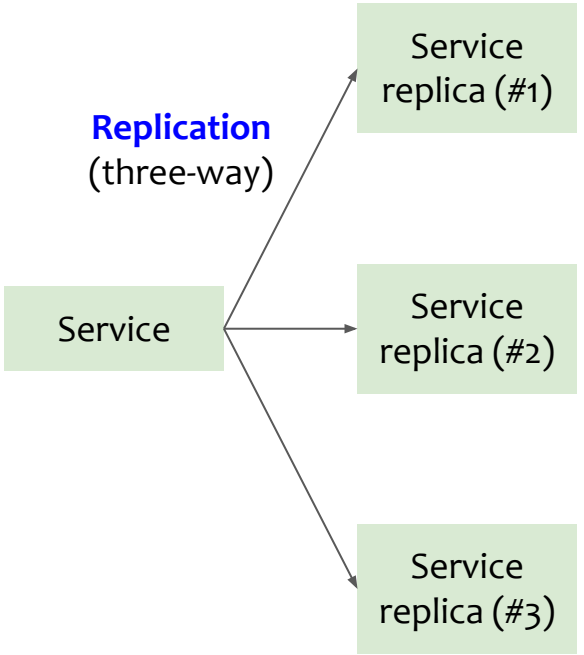
- **Write-ahead logging** is a standard way to ensure data integrity and reliability
  - **Any changes made on the database are first logged in an append-only file** called write-ahead Log or Commit Log
  - Thereafter, **the actual blocks having the data (row, document) on the persistent storage are updated**
  - **System recovery** relies on **"replaying the logs"** for uncommitted operations after a crash on reboot
- **Types of logs**
  - **Undo log** allows a database to undo a transaction
  - **Redo log** allows to redo a transaction



https://blog.cykerway.com/posts/2018/11/18/database-undo-log-and-redo-log.html

Replication

- **Replication** is the primary basis for a fault-tolerant system design
  - **Keeping the application state across multiple machines** on several different nodes, potentially different locations/data centers
  - **Replication provides redundancy:** if some nodes are unavailable, the application can still function from the remaining nodes
  - **A side advantage:** Replication can also help **improve performance** (primarily for read-only workloads)



Replication for stateful services

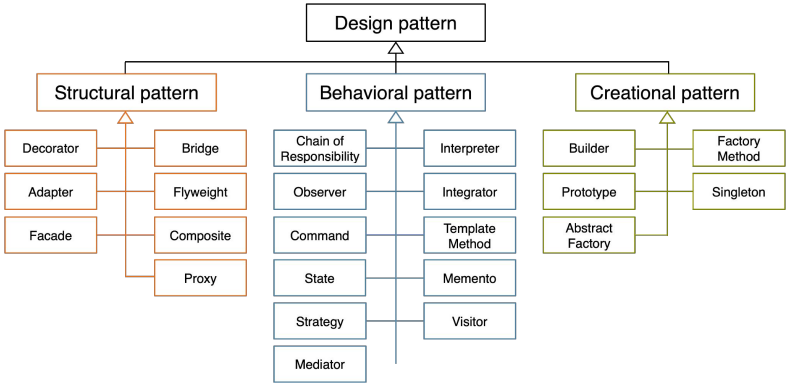
- Each replica stores the copy of the entire state (for fault-tolerance), and a state replication protocol ensures that the replicas are consistent
  - The protocol ensures that every write to the state/data is processed by every replicas, otherwise the replicas would no longer contain the same data
- Two prominent approaches
  - **Primary-backup replication**
    - Aka passive replication, or active-passive replication
    - Based on **replicating the state (or side-effects)**
  - **State machine replication**
    - Aka active replication
    - Based on **replicating the operations**

Lecture overview

- ~~**Part I:** Security~~
  - ~~Security engineering~~
  - ~~Software security in the cloud~~
- ~~**Part II:** Reliability and availability~~
  - ~~Single-node reliable systems and associated issues~~
  - ~~Replication as the general recipe for fault-tolerance~~
  - ~~Fault-tolerance for stateful services~~
- **Part III:** Pattern implementation
  - Adapter pattern
  - Observer pattern
  - Strategy pattern

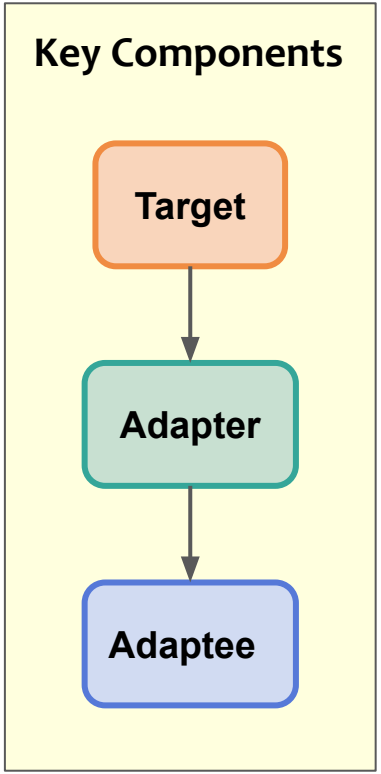
Pattern implementation

- **What are design patterns?**
  - Reusable solutions to common problems in software design
  - Best practices that can be adapted to specific situations
  - Improve code maintainability, flexibility, and extensibility
- **Design patterns taxonomy:**
  - **Structural:** Concerned with the composition of classes and objects
  - **Behavioural:** Define the ways objects interact and communicate with one another
  - **Creational:** Deal with the process of object creation



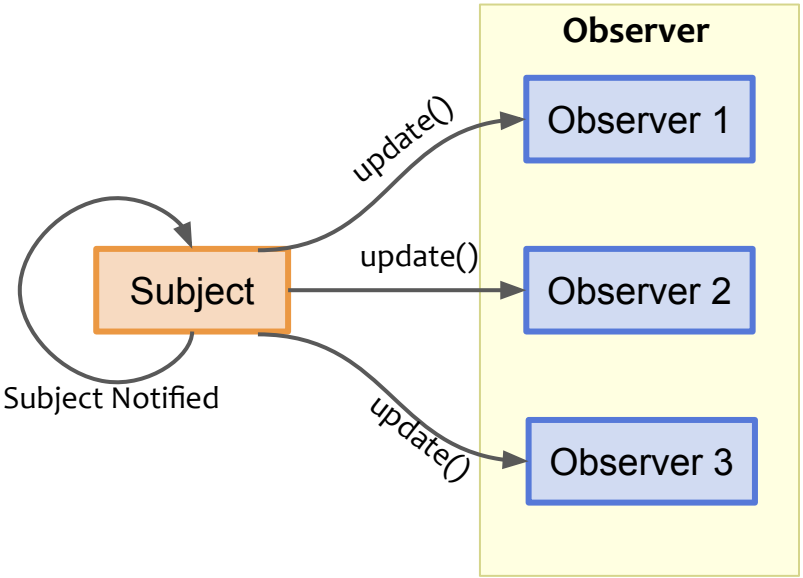
Adapter pattern

- **Target:** The desired interface that the client wants to use
  - Defines the methods that the client will call
- **Adaptee:** The existing interface that needs to be adapted
  - Provides the functionality that needs to be used by the client, but its interface is incompatible with the Target
- **Adapter:** The class that converts the Adaptee's interface into the Target's interface
  - Implements the Target interface and holds a reference to the Adaptee, making it possible to call the Adaptee's methods using the Target's interface



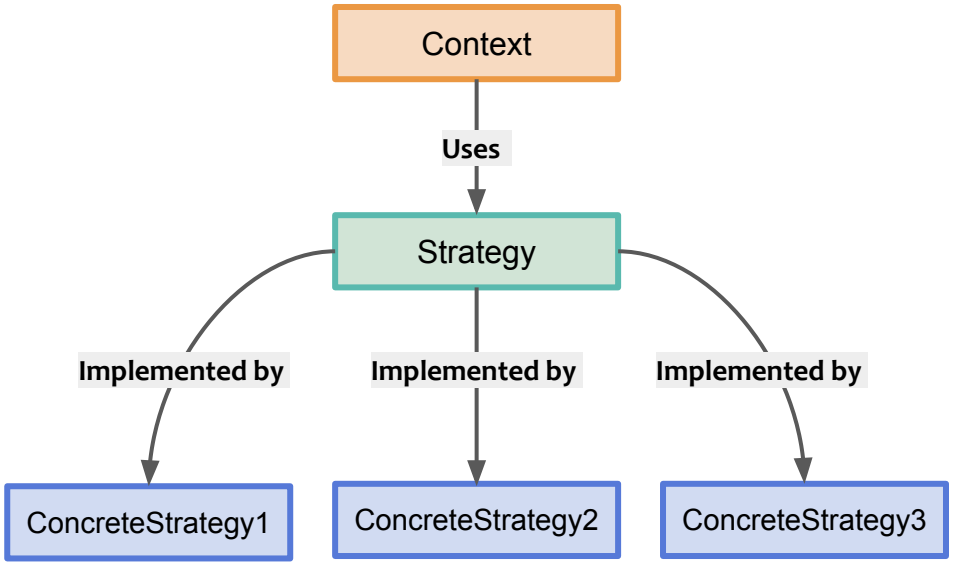
Observer pattern

- **Subject:** The object that has one-to-many dependencies with other objects (observers)
  - Maintains a list of observers and provides methods to add, remove, and notify observers
- **Observer:** An interface that defines the methods to be implemented by objects that need to be notified when the Subject's state changes
  - Specifies the methods to update the observer's state when notified by the Subject



Strategy pattern

- **Context:** The class that uses a Strategy to perform an operation
  - Contains a reference to a Strategy object and delegates the execution of the algorithm to it
- **Strategy:** An interface that defines the methods to be implemented by different algorithms
  - Specifies the methods to execute the algorithm
- **Concrete Strategies:** Classes that implement the Strategy interface and define a specific algorithm
  - Provides a specific implementation of the algorithm



Clues for the use of design patterns

- **For Adapter Pattern:**
  - Need to convert the interface of a class into another interface, clients expect
  - Want to reuse an existing class, but its interface is incompatible with the rest of the system
  - Need to provide a unified interface to a set of classes with diverse interfaces
- **For Observer Pattern:**
  - Need to establish a one-to-many relationship between objects
  - Need to update a set of dependent objects automatically when an object's state changes
  - Want to decouple a subject from its observers, promoting loose coupling
- **For Strategy Pattern:**
  - Need to define a family of algorithms and make them interchangeable
  - Want to provide a way to select an algorithm at runtime
  - Need to encapsulate algorithms and avoid code duplication

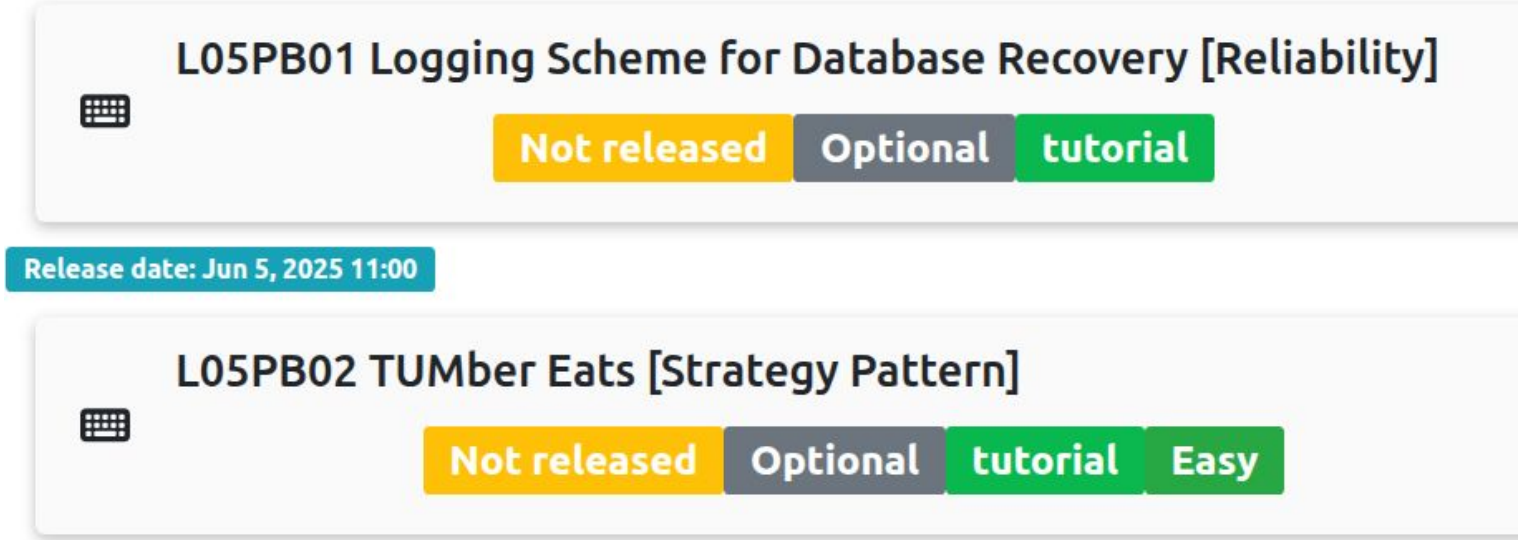
Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- **Part II:** **Programming basics**

- **Part III:** Homework programming exercises (Artemis)

Programming Basics (PB) exercises



L05PB01 Logging Scheme for Database Recovery [Reliability]

- Goals:
  - Understand the roles of undo and redo logging in upholding database integrity and consistency
- Tasks:
  - Implement a basic logging scheme using a Python-based key-value (KV) store and file I/O operations
  - Initial state: a predefined KV-store
  - Create undo log: operations are logged for recovery
  - Simulate recovery: by applying the undo log to revert to the initial state and then reapplying the redo log to return to the post-operations state
  - Verify Integrity: compare the final state of the KV store with the expected state

L05PB01 Logging Scheme for Database Recovery [Reliability]

- The lecture introduces two types of logs:
  - Undo Log: Records operations in a reversible manner, allowing the KV-store to be reverted to its previous state by counteracting recent changes
  - Redo Log: Keeps track of all executed operations, enabling the "replaying" of transactions to restore the KV-store to its post-transaction state
- Purpose of undo/redo logging
  - Database integrity and consistency, especially during system failures
  - Ensures that the database can revert to a **reliable** state, post-failures

L05PB01 Logging Scheme for Database Recovery [Reliability]

```python
def main():
    # Step 1 - Define the initial KV-store state
    print("Initial KV-store state determined.")

    initial_kv_store = {
        "name": "John",
        "age": 25,
        "city": "New York",
        "salary": 50000,
        "status": "active"
    }

    # Step 2 - Define a list of operations to be applied to the KV-store
    operations = [{'action': 'delete', 'key': 'status'},
                  {'action': 'delete', 'key': 'salary'},
                  {'action': 'delete', 'key': 'city'},
                  {'action': 'set', 'key': 'name', 'value': 'Mark'},
                  {'action': 'set', 'key': 'age', 'value': 28},
                  {'action': 'set', 'key': 'country', 'value': 'USA'},
                  {'action': 'delete', 'key': 'position'},
                  {'action': 'delete', 'key': 'department'},
                  {'action': 'set', 'key': 'gender', 'value': 'male'},
                  {'action': 'delete', 'key': 'email'}]

    print("Operations:", operations)

    kv_store = KVStore(initial_kv_store)

    print("Operations applied to kv store.")
    for op in operations:
        kv_store.apply_operation(op)

    comparison_kv_store = kv_store.state.copy()
```

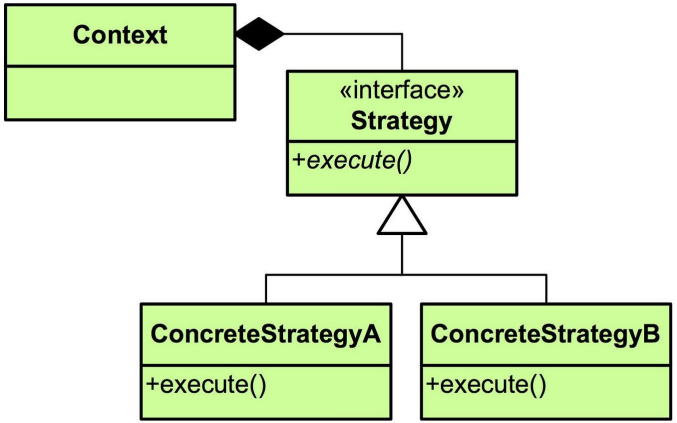
```python
# Step 3 - Generate and write Undo Log
print("Undo log generation..")
undo_operations = Logger.generate_undo_log(operations, initial_kv_store)
Logger.write_undo_log("undo_log.log", undo_operations)

print("KV store after operations:", comparison_kv_store)

# Step 4 - Apply Undo Log to the KV-store
print("After applying undo log:")
Logger.apply_log("undo_log.log", kv_store)
print(kv_store)

# Step 5 - Apply Redo Log to the KV-store
print("After applying redo log:")
Logger.apply_log("redo_log.log", kv_store)
print(kv_store)

# Step 6 - Comparison of initial state and the state after the log files
if kv_store.compare_states(comparison_kv_store):
    print("Success! The final state matches the initial state.")
else:
    print("Error: The final state does not match the initial state.")
```

L05PB02 TUMber Eats [Strategy Pattern]

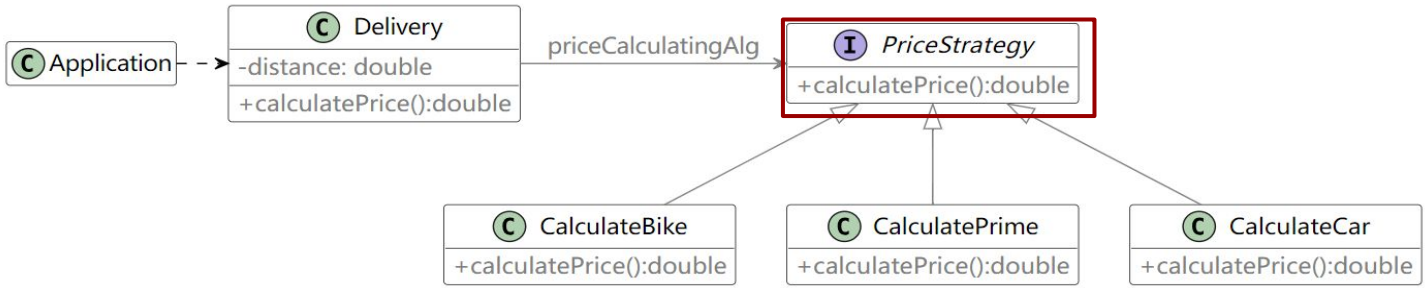
- Goals:
  - You are now contributing to *TUMber Eats*
  - You are required to implement price-calculating algorithms for deliveries
  - The prices vary depending on the on the distance and if the orders are from normal or VIP customers
- Tasks:
  - Adapt the application using the strategy pattern
    - Adapt the modifier of *PriceStrategy*
    - Declare the method to calculate the price
    - Make the price calculations implement the *PriceStrategy*
    - Add a *PriceStrategy* attribute in the delivery class and implement getter/setter as well as a method to call the strategy method
  - Implement a new algorithm to calculate the price for a delivery by car

L05PB02 TUMber Eats [Strategy Pattern]

- The strategy pattern is a behavioral design pattern
  - It lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable
- Strategy: An interface that defines the strategy
  - Needs to be inherited by different concrete strategies
  - The *execute* methods are designed specifically according to the requirements
- Context **:** A class that uses the strategy
  - Has an attribute for storing a reference to a strategy object
  - A setter for replacing values of the strategy



L05PB02 TUMber Eats [Strategy Pattern]

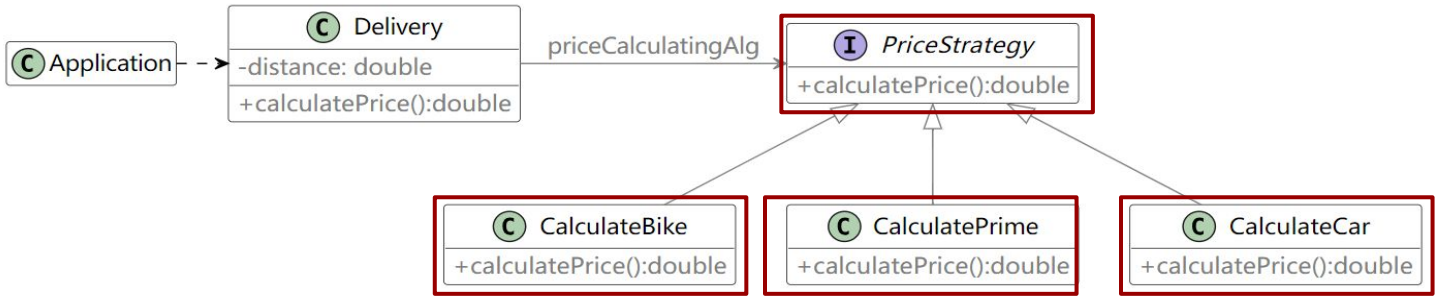


1. Adapt the modifier

2. Declare strategy method

```
public interface PriceStrategy {
    double calculatePrice(double distance);

}
```

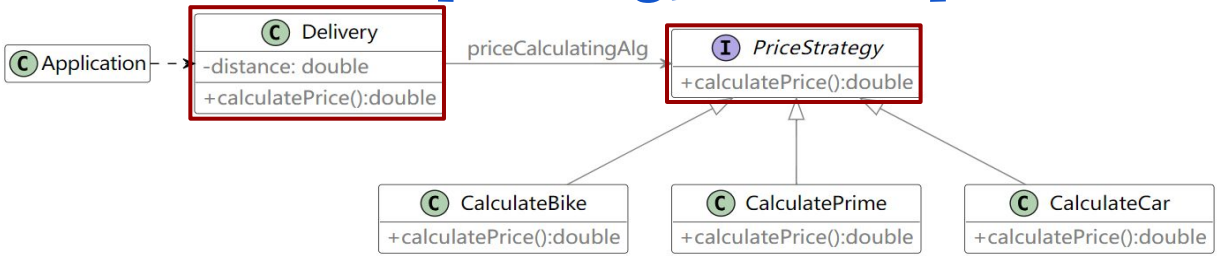
L05PB02 TUMber Eats [Strategy Pattern]



3. Implement strategy pattern

```
public class CalculateBike  implements PriceStrategy {
 @Override
 public double calculatePrice(double distance) {
  DistanceChecking.checkDistance(distance);
      Return 2.0 + distance/2.8;
  }
}
```

L05PB02 TUMber Eats [Strategy Pattern]

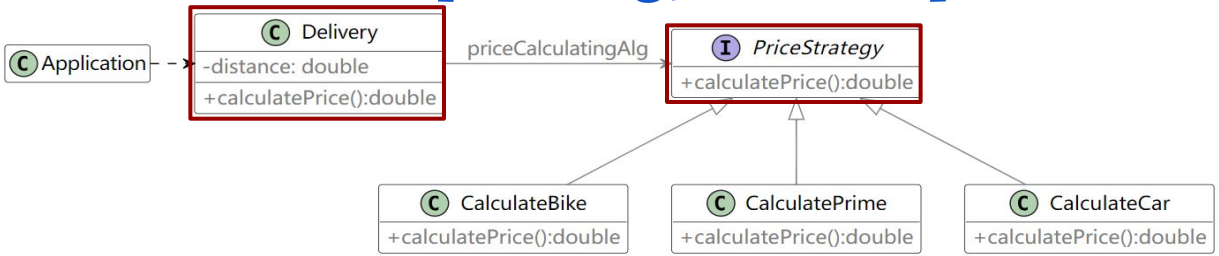



```
public class Delivery {
   public static final String ILLEGAL_DISTANCE_MESSAGE = "The input for the distance is
invalid";
   private double distance;
   private PriceStrategy priceCalculatingAlg;

   public Delivery() {
   }
   public Delivery(double distance) {
       this.distance = distance;
       this.priceCalculatingAlg = priceCalculatingAlg;

   } ...
```

4. Add strategy attribute

5. Initialize in the constructor

L05PB02 TUMber Eats [Strategy Pattern]

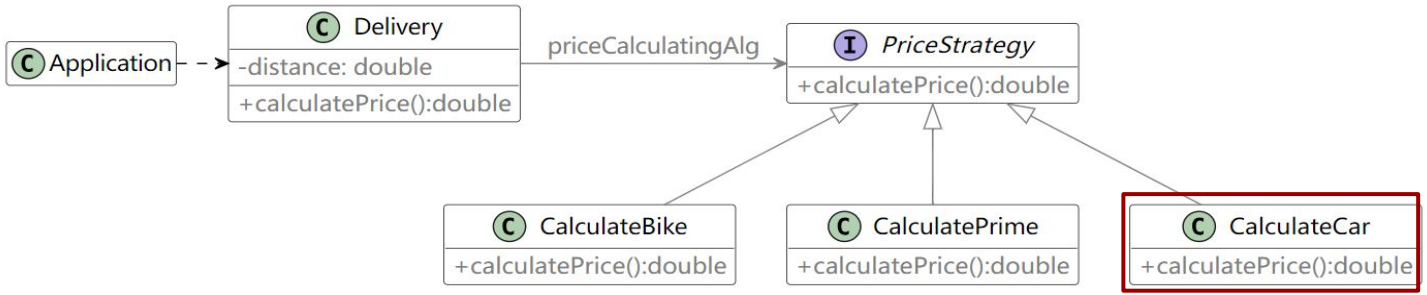



```
... }
    public PriceStrategy getPriceCalculatingAlg() {
        return priceCalculatingAlg;
    }
    public void setPriceCalculatingAlg(PriceStrategy priceCalculatingAlg) {
        this.priceCalculatingAlg = priceCalculatingAlg;
    }
    public double calculatePrice() {
        return priceCalculatingAlg.calculatePrice(this.distance);
    }
}
```

6. Implement setter and getter method

7. Implement the calculatePrice() method

L05PB02 TUMber Eats [Strategy Pattern]




```
public class CalculateCar implements PriceStrategy {
 @Override
 public double calculatePrice(double distance) {
   DistanceChecking.checkDistance(distance);
   return 3.2 + distance / 2.6;
 }
}
```

8. Implement concrete strategy

Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~
- ~~**Part II:** Programming basics~~
- **Part III: Homework programming exercises (Artemis)**

Programming (P) exercises

L05P01 Database Sharding and Recovery [Reliability, Availability]

- Goals:
  - Implement Undo/Redo Logging (for reliability and resilience)
  - Implement replication for a sharded database (for availability)
- Tasks:
  - Part 1:
    - Use `log_and_apply_operations()` to log operations
    - Implement `apply_operation()`
    - Generate and write undo Log
    - Write the undo log
    - Apply the Undo log
    - Apply the Redo log
    - Compare the initial state and the state after applying the undo and redo log
  - Part 2:
    - Implement `doesDBContainKey(...)`, `doesDBContainKeys(...)` and `empty_nodes_check_remaining(...)`
    - Implement `create_replicates(...)`, `recover_node(...)` and `recover_nodes(...)` for replication of the database

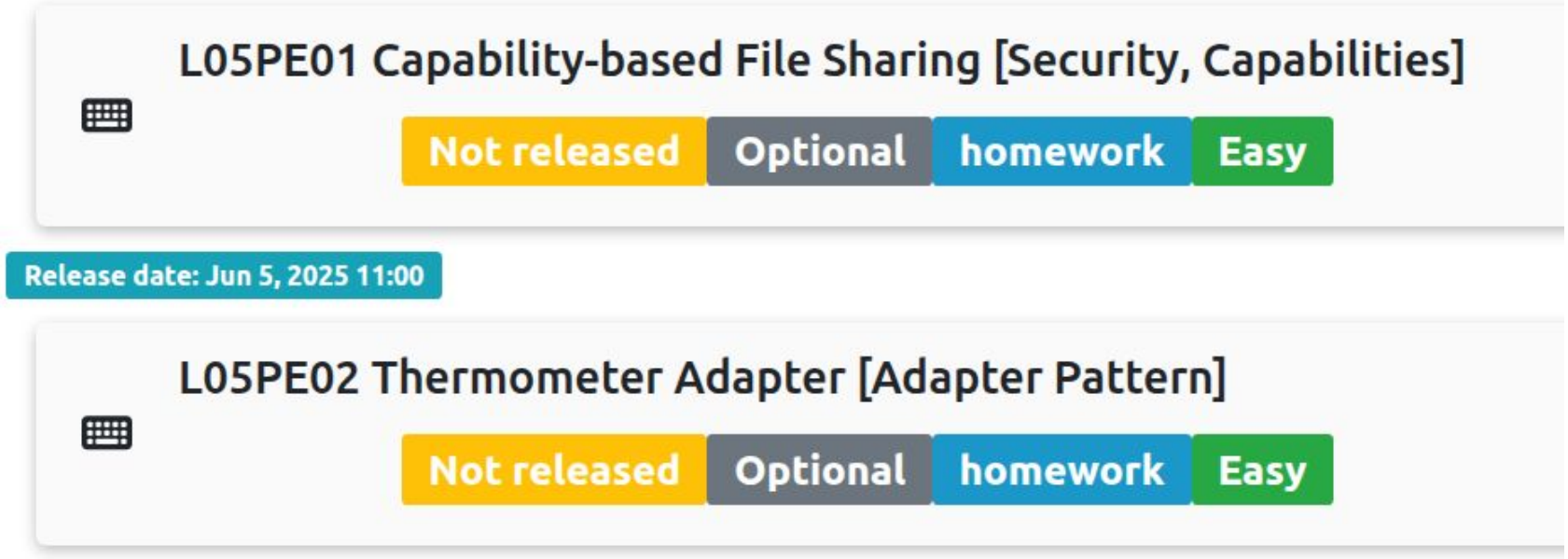
L05P02 Searching in e-BOOKist [Strategy Pattern]

- Goals:
  - Implement Linear and Binary Search
  - Implement the Strategy Pattern for the search in a book
- Tasks:
  - Part 1:
    - Implement Linear and Binary Search
  - Part 2:
    - Create a `SearchStrategy` interface
    - Create and implement the `Context` class (with getters & setters for all attributes)
    - Implement `Context#isChaptersSortedByName()` and `Context#search()`
    - Create and implement the `Policy` class
    - Implement `Client` class

L05P03 Old Beer Factory Digitalization [Facade, ACL]

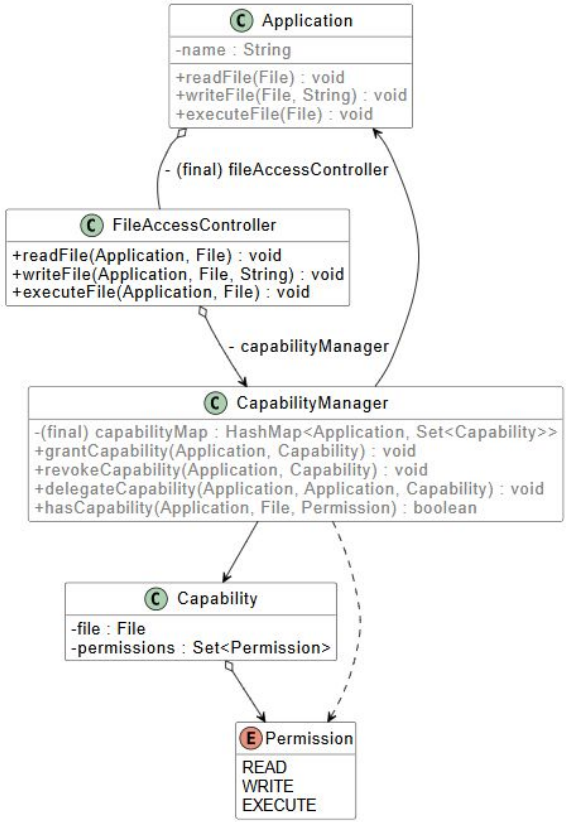
- Goals:
  - Implement AccessControlList (ACL)
  - Implement Facade Pattern
  - Implement REST APIs in the Microservices
  - In general: finish the Inventory and Shipping system of a brewery and additionally implement an ACL for it
- Tasks:
  - Part 1:
    - Implement `grantAccess(...)` and `hasAccess(...)` in class `AccessControlList`
  - Part 2:
    - Implement `addProduct(...)`, `sellProduct(...)`, `checkProduct(...)` and `shippingRecord(...)` in class `FactoryFacade`
  - Part 3:
    - Implement `addProduct(...)`, `removeProduct(...)` and `checkProduct()` in class `InventoryController`
    - Implement `makeShipping(...)` and `shippingRecord()` in class `ShippingControler`

Programming Extras (PE) exercises



L05PE01 Capability-based File Sharing [Security, Capabilities]

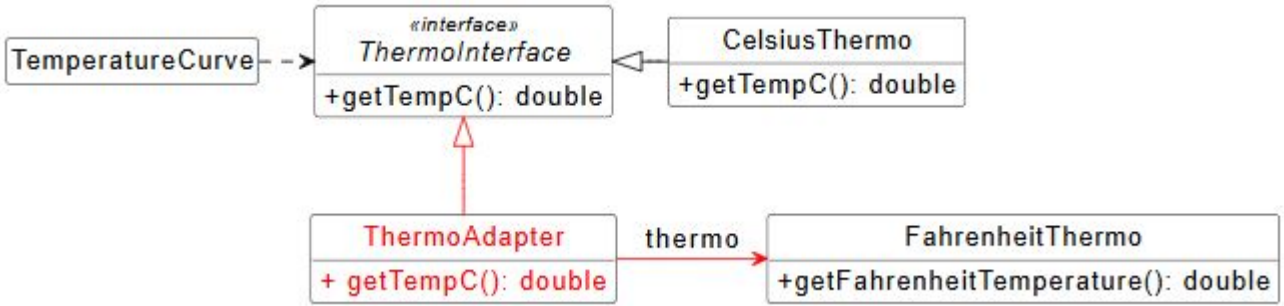
- Goals:
  - Implement Capability-Based Access Control (CBAC)
- Tasks:
  - Part 1:
    - Implement `Application` class + its constructor according to the UML class diagram
    - Implement `readFile(...)`, `writeFile(...)` and `executeFile(...)` in class `Application`
  - Part 2:
    - Implement the `CapabilityManager` class according to the UML class diagram
    - Implement `hasCapability(...)`, `grantCapability(...)`, `revokeCapability(...)` and `delegateCapability(...)` in class `CapabilityManager`



L05PE01 UML class diagram

L05PE02 Thermometer Adapter [Adapter Pattern]

- Goals:
  - Implement Adapter Pattern to translate temperature from Fahrenheit to Celsius
- Tasks:
  - Add `ThermoAdapter` class according to the UML class diagram
  - Add `thermo` attribute and instantiate it
  - Implement delegation of `getTempC()`-method call to `thermo` attribute
  - Replace `CelsiusThermo` implementation in `TemperatureCurve` class with `ThermoAdapter`



L05PE02 UML class diagram