T03 System Design I

Modularity and Data Management

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

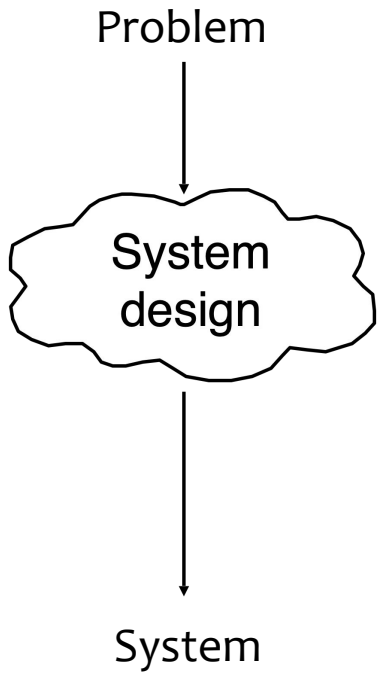
- **Part I:** System design challenges
- **Part II:** Modularity
  - Subsystem decomposition: Modules
  - Differentiate between coupling and cohesion
  - Design pattern: Facade pattern
  - Interface design
- **Part III:** Software architecture: Layered architectures
  - Open vs closed layered architectures
  - Different layers, different abstraction
  - Pulldown the complexity downward/upward
  - Ubiquitous adoption of layered architectures in systems
- **Part IV:** Data management
  - Key Value (KV) store, Filesystems, Shared log, Databases
- **Part V:** Pattern implementation (MVC pattern)

The scope of system design

- Bridge the gap between a problem and a system in a manageable way
- The system design should address both
  - **Functional requirements**
  - **Non-functional requirements**

Collectively also known as **FURPS requirements**

Functionality
Usability
Reliability
Performance
Supportability



Approach:

- Understand the **functional requirements**
- Identify **non-functional requirements**

FURPS is a broader taxonomy of functional and non-functional requirements: https://en.wikipedia.org/wiki/FURPS

A three-part series: System design in our course

L03: Design I

- **Modularity**
  - How to design modular systems?
- **Data management**
  - How to manage your data?

L04: Design II

- **Performance**
  - How to design performant systems?
- **Concurrency (Scale-up)**
  - How to scale-up systems?
- **Scalability (Scale-out)**
  - How to scale-out systems?

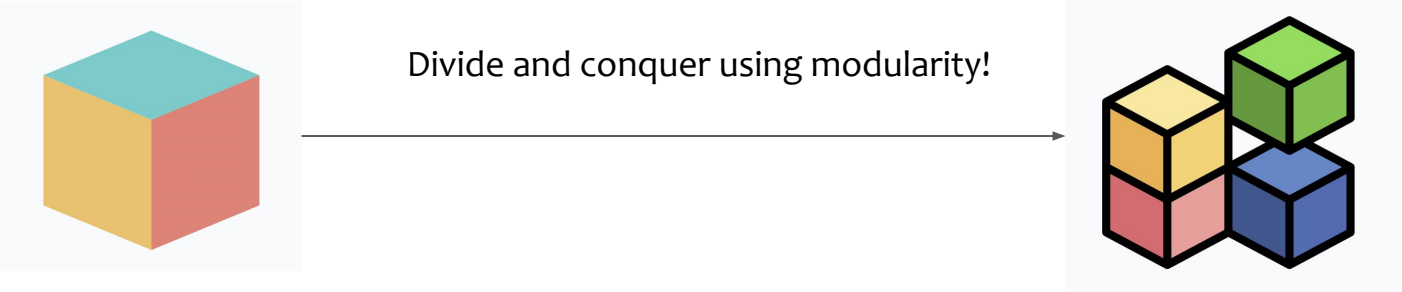
L05: Design III

- **Security**
  - How to secure your systems?
- **Fault tolerance**
  - How to make systems **reliable** & **available**?

System implementation

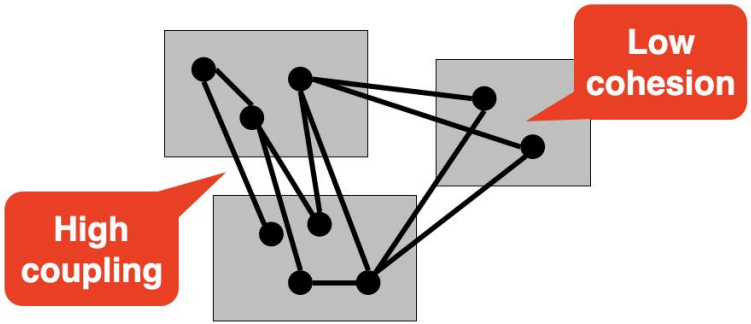
How to approach system design?

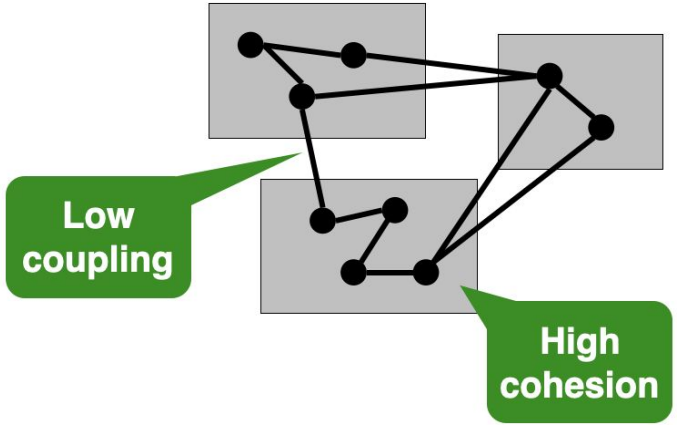
- Given a complex “problem statement”, how do we design systems?
  - **Use divide & conquer**
    - Model the **new system design as a set of subsystems**
    - Address **the major design goals first**



A good system design

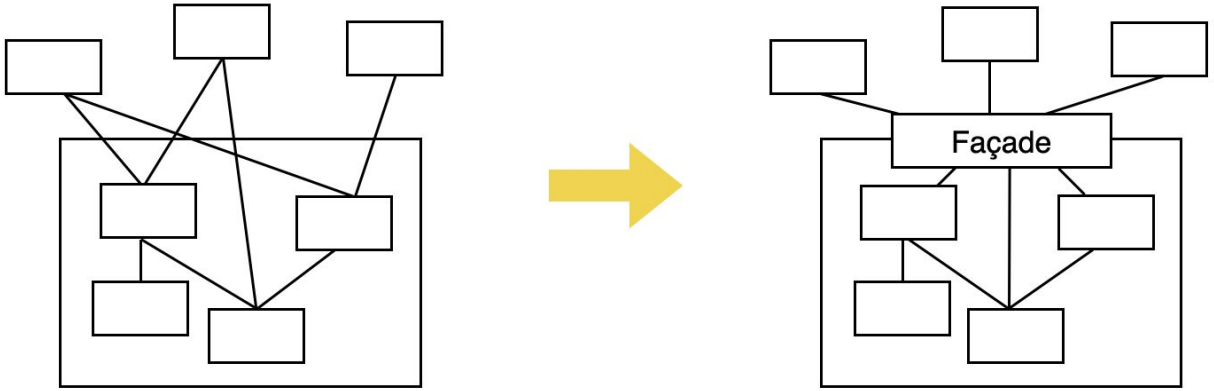
- A good system design aims to achieve **high cohesion** and **low coupling**
  - High cohesion strives for tightly dependent objects in one module
  - Low coupling strives to minimize interdependence of objects between different modules





Facade design pattern: Reduces coupling

- Provides **a unified interface for a module/subsystem**
  - Consists of **a set of public operations**
    - Each public operation is delegated to one or more operations in the classes behind the façade
  - **Defines a higher-level interface** that makes the subsystem easier to use (i.e., it abstracts away the gory details)
    - Allows to hide design spaghetti from the caller



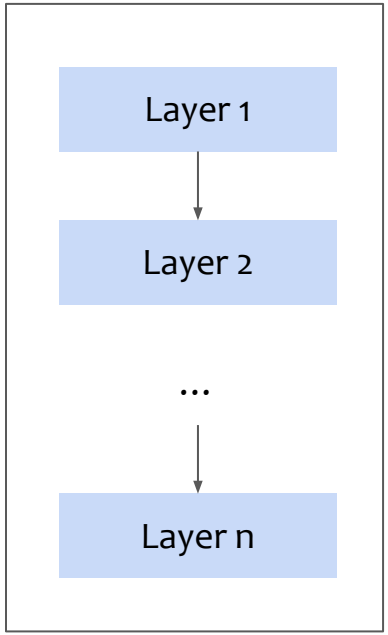
Interface design

- **An interface** is a boundary that separates different subsystems and defines how they interact with each other
- **Interface parts:**
  - **Formal parts:** Explicitly specified parts in the code (Specified as the API/method signature)
  - **Informal parts:** Not explicitly specified in the code, but captures the high-level behavior (Usually, specified in the comments)
- **A developer needs to know both formal and informal parts** of the interface before invoking/using them
- **The best modules are deep** (powerful functionality with a simple interface)
- Information hiding leads to **good interface design and reduce complexity**

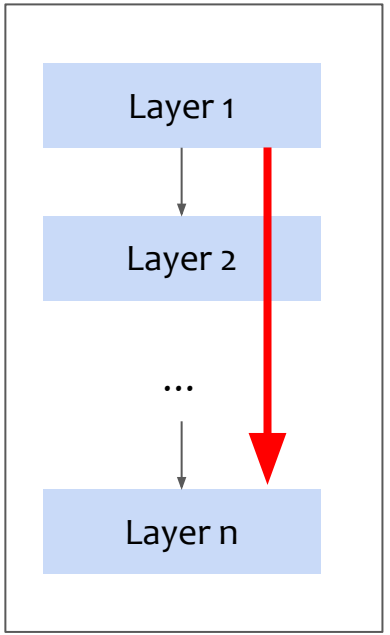
Layered architecture

- **A Layered architecture** achieves modularity by dividing the system into distinct layers (subsystems/components)
  - A layer only **depends on services from lower layers**
  - A layer **has no knowledge of higher layers**
- In a well-designed system, each layer **provides a different abstraction** from the layers above or below it
- If **two adjacent layers provide the same abstraction**, it is **NOT a good decomposition** of system into layers
- **Avoid config parameters!!** as each layer must work independently of who gonna use it.

Layered architecture: Open vs closed



Closed layered architecture:
If each layer can only call operations from the layer directly below
Pros: Cleaner system interfaces and design, portability, maintainability



Open layered architecture:
A layered architecture is open if a layer can call operations from any layer below
Pros: High-performance

Data management

- **State management:** An application developer models the data as objects or data structures, and uses APIs to manipulate those data structures

- A **data management system** offers
  - **Data model:** An abstraction for storing and retrieving data objects
  - **Application programming interface (APIs):** Interface for manipulating data objects

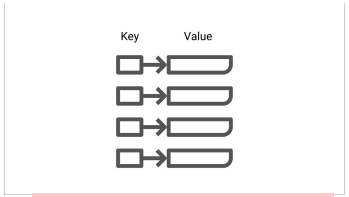
- Data management systems present **different trade-offs** in terms of interface, performance, programmability, reliability, and security properties
  - A complex application may make use of different data management systems

Data management systems

Application

State management via APIs

Data management systems



A: Key-value stores

B: Filesystems

C: Shared logs

D: Databases

Key-value store (KVS)

- **KVS** stores data as a set of unique identifiers, each having an associated value
- Oten used as **cache** to accelerate application responses
- **Simple acces model:** Do not require to support complex queries
- **Scalable data management systems**
- **Types of KVS**
  - **In memory:** Maintain the entire state in the main memory **(fast)**
  - **Persistent:** Maintain the state in memory **(fast)** and on disk/state **(non-volatile)**

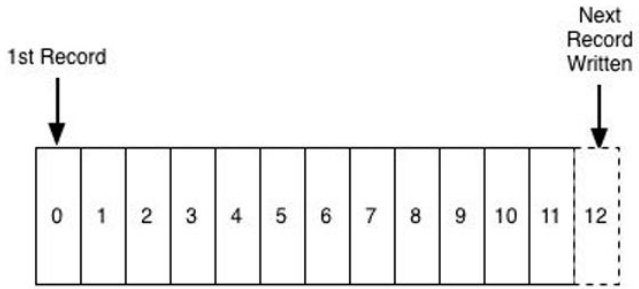
Key (Name)	Value (Age)
Pramod	37
Martin	38

Filesystems

- **A filesystem** is a data structure that stores data & information on storage devices
  - Persistent data management
  - Namespace: hierarchical directory structure with directories and files
  - Metadata: file sizes, last modified, etc.
  - APIs for access the filesystem: creating, manipulating, reading, writing, deleting, etc.
  - Security: Access control lists (ACLs)
- Usage as **unstructured data store**
- **APIs**:
  - **Open and close**: Opening and closing a file
  - **Read, write and position:** Reading, writing, and seeking a location for random access
  - **Metadata management:** File metadata (last access, size, access permissions, etc.)

Shared logs

- **A log** is an **append-only file, totally-ordered** sequence of records ordered by time
- **Records are appended** to the end of the log
- Reads proceed **left-to-right**
- Each entry is assigned a **unique sequential log entry number**
- They are widely used for storing **a stream of bytes** providing **fast sequential R/W access**
- Large-scale data analytics systems also use shared logs
  (e.g., Distributed systems, iot, database systems)



Databases

- **A relational database** is a collection of information that **organizes data in predefined relationships** where data is stored in one or more tables (or "relations") of columns and rows
- Relationships are a **logical connection** between different tables
- Each table stores information in columns (attributes) and rows (records/tuples)
- **Advantages**
  - Structured data management
  - Complex analytics via SQL query language
  - Built-in ACID transactions

Transaction processing

- In **transaction processing,** work is divided into individual, indivisible operations, called transactions (TXs)
- **Advantages:** TXs shield programmers from low-level management of
  - Concurrent processing of records
  - Integrity of data
  - Manages the prioritization of transaction execution

ACID properties of transactions

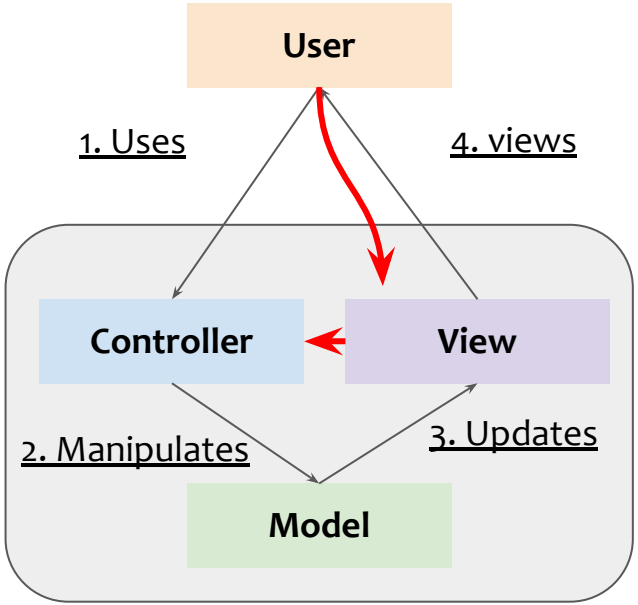
- **Atomicity**
  - All changes to data are performed as if they are a single operation
  - That is, all the changes are performed, or none of them are
- **Consistency**
  - Data is in a consistent state when a transaction starts and when it ends
- **Isolation**
  - The intermediate state of a transaction is invisible to other transactions
  - As a result, transactions that run concurrently appear to be serialized
- **Durability**
  - After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure

https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions

Model-view-controller (MVC)

- **Model-View-Controller (MVC)** is a design pattern that separates an application into three main logical components:
  - the model, the view, and the controller

- **The key idea:** Each of these components are built to handle specific aspects of an app
  - *Some of your code holds **the data of your app (model)**, some of your code makes your **app look nice (view)**, and some of your code controls **how your app functions (control)***



Push- vs pull-based MVC

- **Pull notification variant:** view and controller obtain the data from the model
  - Pulling the new updates “on-demand”
  - **Pros:** Saves bandwidth
  - **Cons:** Missing intermediate updates
- **Push notification variant:** the model sends the changed state to view and controller
  - Pushing the updates for “all changes”
  - **Pros:** Not missing any updates
  - **Cons:** Wastes bandwidth

Select the variant that suits your application requirements!

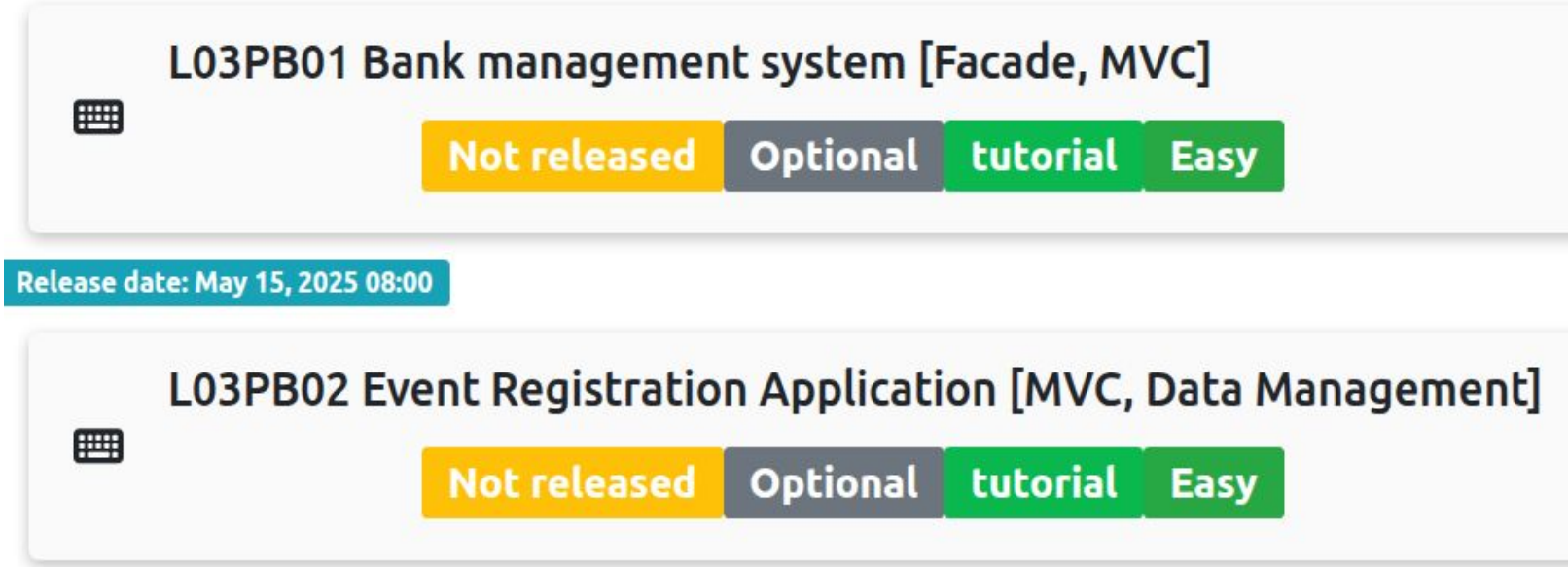
Tutorial outline

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

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

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

Programming Basics (PB) exercises



L03PB01 Bank management system [Facade, MVC]

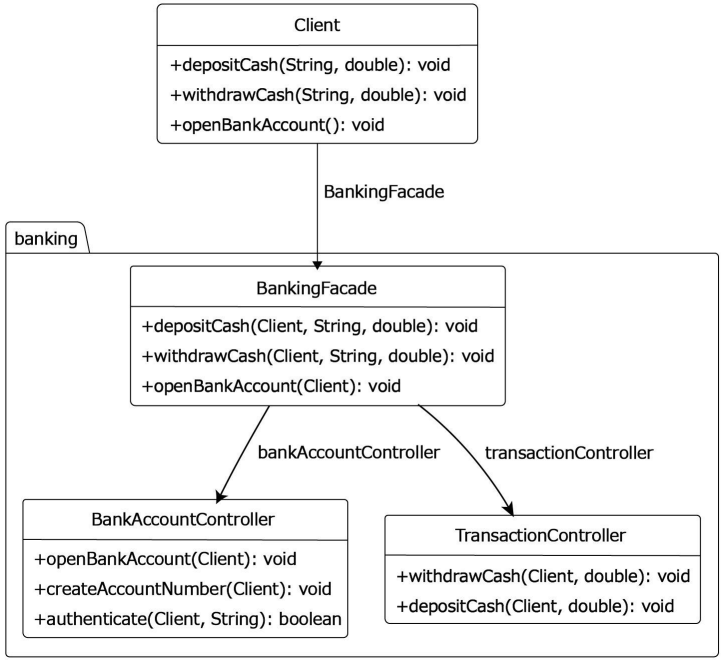
- Tasks:
  - Designing a simplified bank management system which includes subsystems such as account management and transaction handling.
  - A façade pattern needs to be applied to make these interactions between clients and subsystems simpler.
  - The implementation is done in Java.

- Goals:
  - **Understand** the concept of the facade pattern
  - **Experience** how the facade pattern simplifies interactions, enhances privacy, and promotes modular design in a banking system

L03PB01 Bank management system [Facade, MVC]

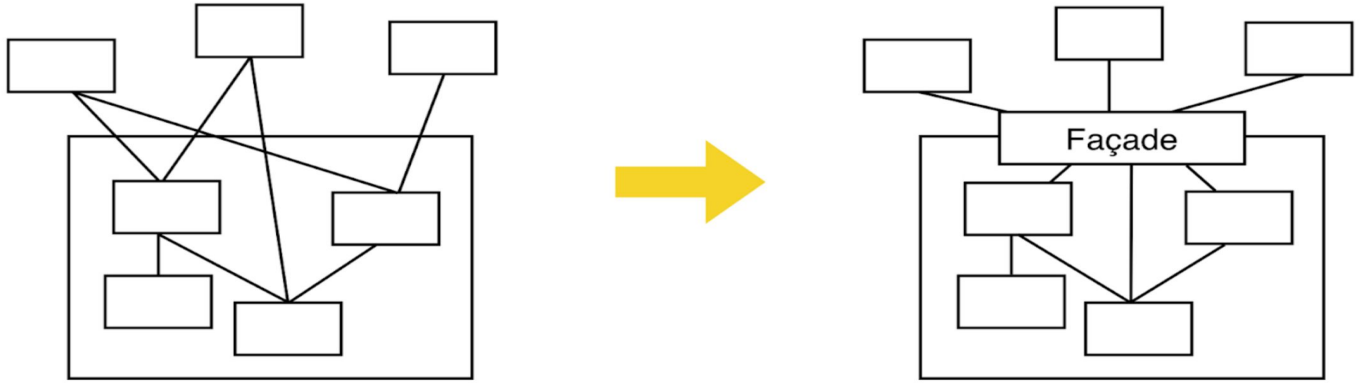
To do:

1. Implement the class *BankingFacade* to abstract the banking subsystem.
2. Remove the associations between the client and the controllers: *BankAccountController* and *TransactionController*.
3. Implement a new algorithm for the methods *depositCash*, *withdrawCash* and *openBankAccount* by invoking methods on the *BankingFacade*.



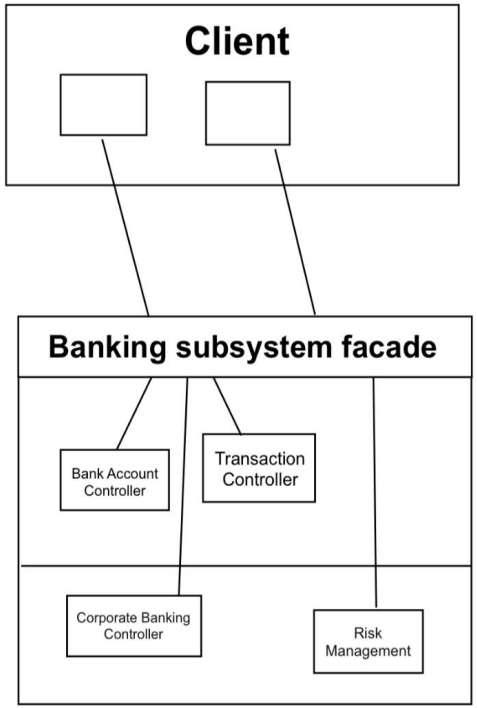
L03PB01 Bank management system [Facade, MVC]

- Provides a unified interface for a subsystem with a set of public operations
- Each public operation is delegated to one or more operations in the classes behind the façade
- Defines a higher-level interface that makes the subsystem easier to use
- Allows to hide design complexities from the user



L03PB01 Bank management system [Facade, MVC]

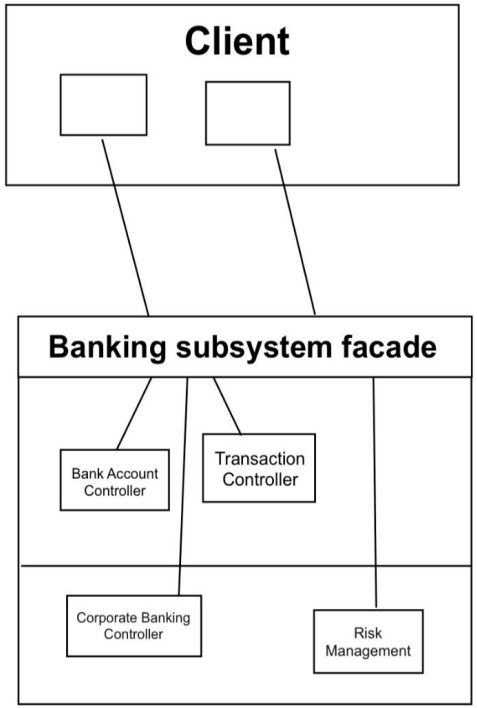
Why is the use of a façade particularly appropriate in this context:



L03PB01 Bank management system [Facade, MVC]

Why is the use of a façade particularly appropriate in this context:

- The façade pattern **encapsulates** the subsystems, preventing direct access by clients.
- Complexities of managing bank accounts is abstracted from the users.
- Additionally, it helps to **protect** sensitive information from being exposed to unauthorized parties.



L03PB01 Bank management system [Facade, MVC]

1. Implement the class *BankingFacade* to abstract the subsystem controllers

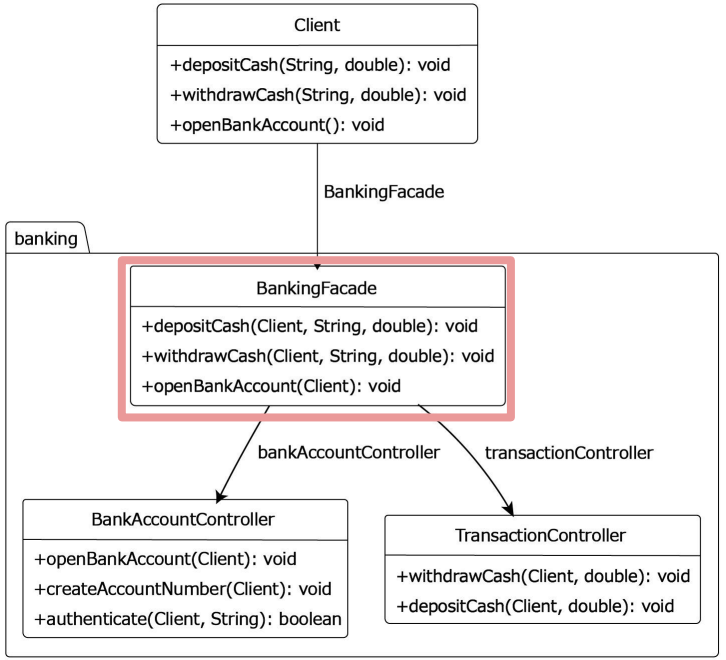
```
public class BankingFacade {

    private final BankAccountController bankAccountController;

    private final TransactionController transactionController;


    public BankingFacade() {
        this.transactionController = new TransactionController();
        this.bankAccountController = new BankAccountController();
    }

    public void depositCash(Client client, String pinCode, double cash) {
        if (!bankAccountController.authenticate(client, pinCode)) {
            System.out.println("Incorrect PIN Code. Access denied.");
        } else if (cash <= 0) {
            System.out.println("Bad request!");
        } else {
            this.transactionController.depositCash(client, cash);
            System.out.println("Deposit of $" + cash + " completed. " +
                    "Your current balance: " + client.getBalance());
        }
    }
```



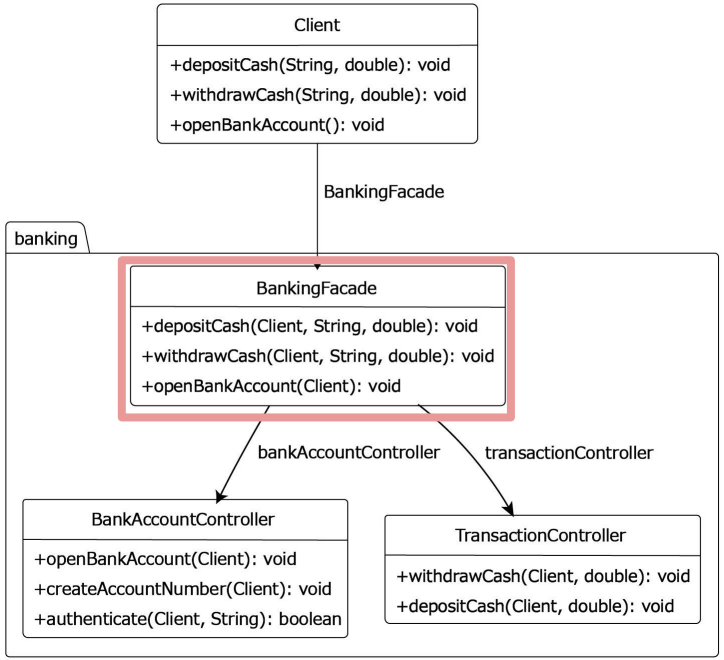
L03PB01 Bank management system [Facade, MVC]

1. Implement the class *BankingFacade* to abstract the subsystem controllers

```
    public void withdrawCash(Client client, String pinCode, double cash) {
        if (!bankAccountController.authenticate(client, pinCode)) {
            System.out.println("Incorrect PIN Code. Access denied.");
        } else if (client.getBalance() < cash) {
            System.out.println("Bad request!");
        } else {
            this.transactionController.withdrawCash(client, cash);
            System.out.println("Withdraw of $" + cash + " completed. " +
                    "Your current balance: " + client.getBalance());
        }
    }

    public void openBankAccount(Client client) {
        this.bankAccountController.openBankAccount(client);
    }

}
```

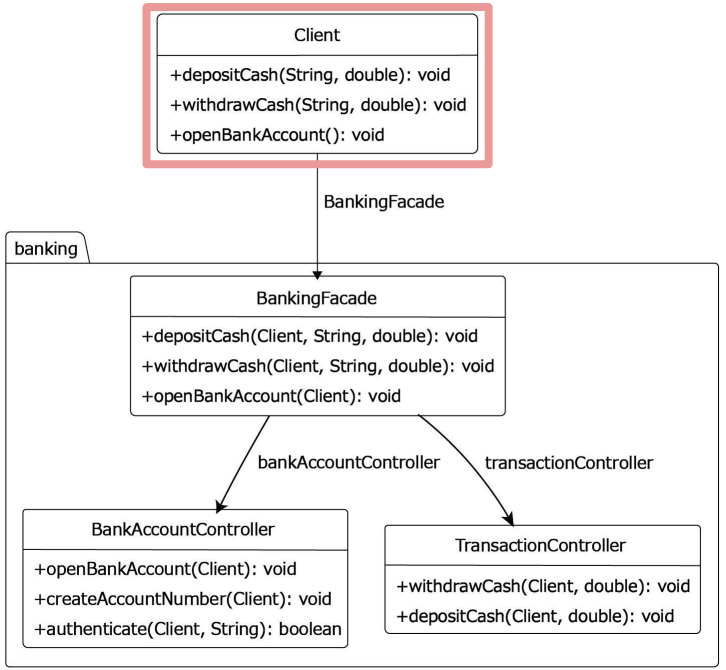


L03PB01 Bank management system [Facade, MVC]

2. Remove the associations between the *Client* and *BankAccountController* and *TransactionController*

```
public class Client {

    // TODO 2 remove all associations to the different controllers
   // private final TransactionController transactionController;
   // private final BankAccountController bankAccountController;
    private final BankingFacade bankingFacade;
```



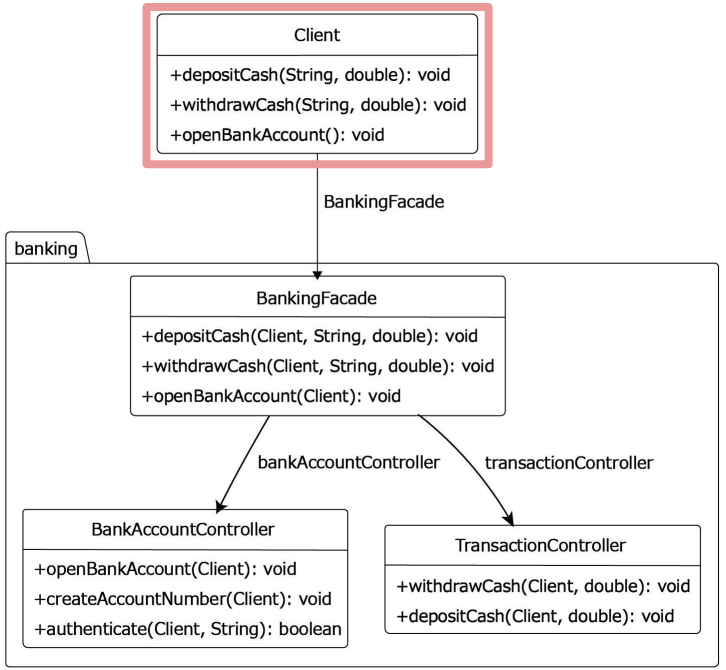
L03PB01 Bank management system [Facade, MVC]

3. Implement an algorithm for the methods *depositCash*, *withdrawCash* and *openBankAccount* in the class *Client*

```
public void depositCash(String pinCode, double cash) {
    this.bankingFacade.depositCash(this, pinCode, cash);
}

public void withdrawCash(String pinCode, double cash) {
    this.bankingFacade.withdrawCash(this, pinCode, cash);
}

public void openBankAccount() {
    bankingFacade.openBankAccount(this);
}
```



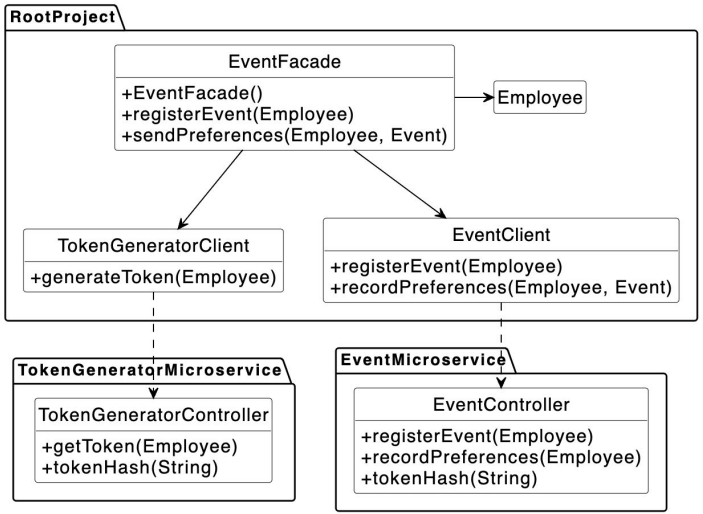
L03PB02 Event Registration Application [MVC, Data Mgmt]

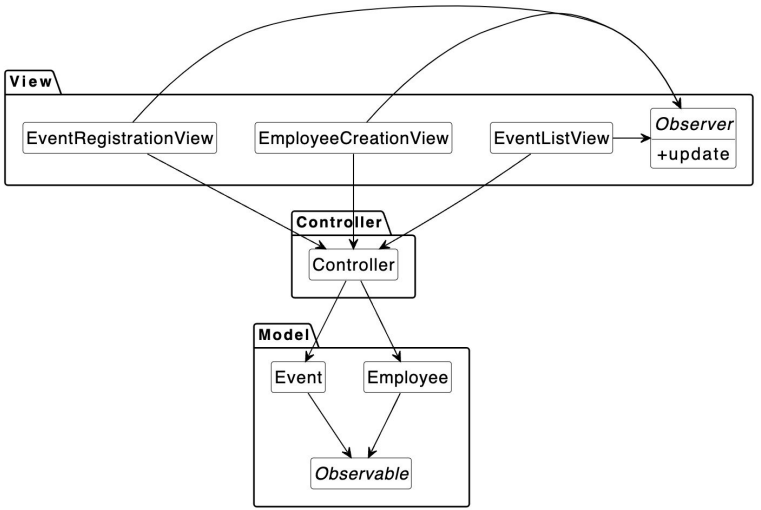
- Tasks:
  - Implement an event registration application for Your company.
  - Use the facade pattern with access policy to simplify the usage and guarantee that allowed employees can create / register for events.
  - Use the model view controller pattern to implement an UI for the users.

- Goals:
  - **Deepen the understanding** of the facade pattern
  - **Understand** the model view controller pattern
  - **Experience** how different patterns work together and how each pattern solves a specific problem

L03PB02 Event Registration Application [MVC, Data Mgmt]

Example UML Class diagrams:

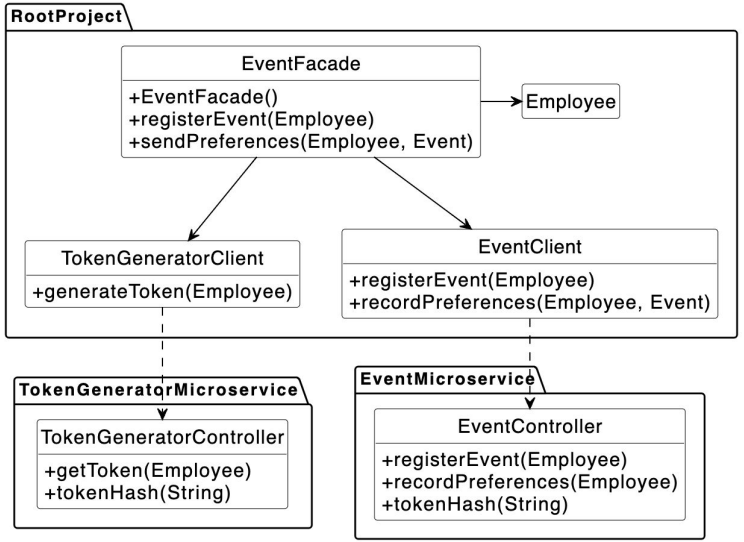




L03PB02 Event Registration Application [MVC, Data Mgmt]

To do:

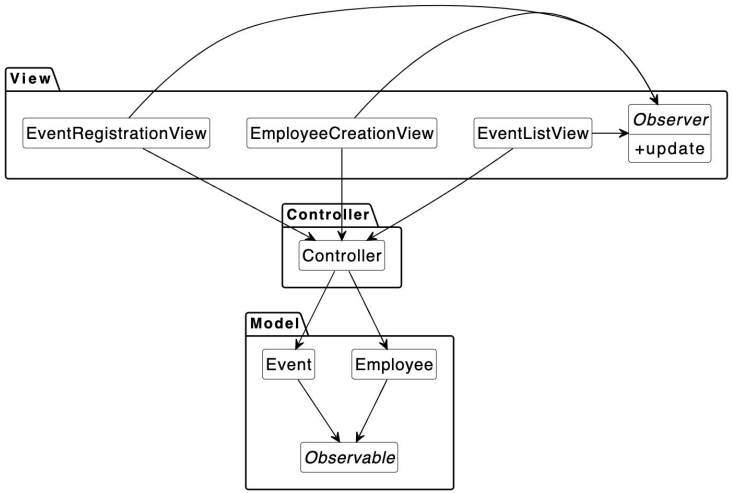
1. Implement the *registerEvent* method inside the class *EventFacade*.
2. The method should make sure that only managers can create events and should handle token generation / validation.
3. Implement the *registerEvent* endpoint inside the *EventController* class.
4. The endpoint should create a token specific to the employee, verify its validity and if it is valid, add the employee to the event.



L03PB02 Event Registration Application [MVC, Data Mgmt]

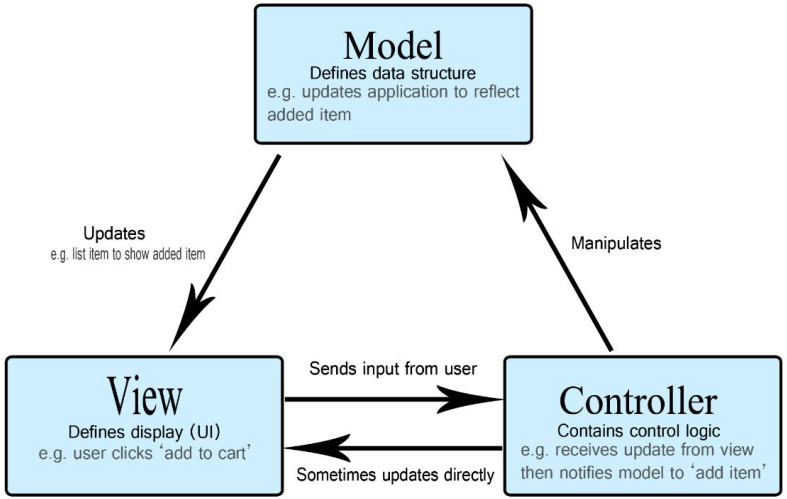
To do:

1. Implement the *addNewEmployee* method inside the class *Controller*.
2. The method should add the employee to the *EventRegistrationView* and notify the employees observers.
3. Implement the *save* method inside the *EmployeeCreationView* class.
4. The method should update the name of the employee and use the controller to add the employee instance.



L03PB02 Event Registration Application [MVC, Data Mgmt]

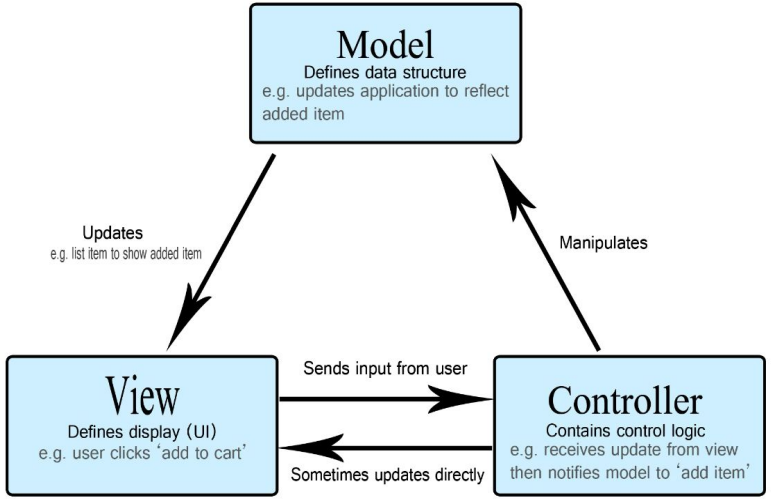
- Provides a clear separation between the display / UI, the data structure and the control logic of an application
- The view is responsible for what the user can see and interact with
- The model contains all the data the user can see or manipulate
- The controller receives / sends events from / to the view and updates the model



L03PB02 Event Registration Application [MVC, Data Mgmt]

Example: A manager has updated the event description and clicks on the save button

1. The *view* sends an event to the *controller* that the description of the event should be updated
2. The *controller* validates the description that it's not too short or too long
3. The *controller* updates the *model* and tells the *view* that the update was successful
4. The *model* sends an update to the *view* so that the event description is displayed correctly



L03PB02 Event Registration Application [MVC, Data Mgmt]

1. Implement the *registerEvent* method inside the class *EventFacade*.

```
public void registerEvent(Employee employee) {
    try {
        if (employee.getRole() == Employee.Role.MANAGER) {
            String token = tokenGeneratorClient.generateToken(employee);
            employee.setToken(token);
            try {
                if (eventClient.registerEvent(employee).equals("Registration is successful")) {
                    employee.setIsRegistered(true);
                } else {
                    System.out.println("Registration failed");
                }
            } catch (Exception e) {

            }
        } else {
            System.out.println("Only managers can register the event!");
        }
    } catch (Exception e) {

    }
}
```

L03PB02 Event Registration Application [MVC, Data Mgmt]

2. Implement the *registerEvent* endpoint inside the *EventController* class.

```
@PostMapping("registerEvent")
public String registerEvent(@RequestBody Employee employee) {
    String token = tokenHash(employee.getName() + employee.getId());
    if (token.equals(employee.getToken())) {
        registeredList.add(employee);
        return "Registration is successful";
    } else {
        return "Registration failed";
    }
}
```

L03PB02 Event Registration Application [MVC, Data Mgmt]

3. Implement the *addNewEmployee* method inside the class *Controller*.

```
public void addNewEmployee(EmployeeAdapter employee) {
    this.eventRegistrationView.addNewEmployee(employee);
    employee.notifyObservers();
}
```

L03PB02 Event Registration Application [MVC, Data Mgmt]

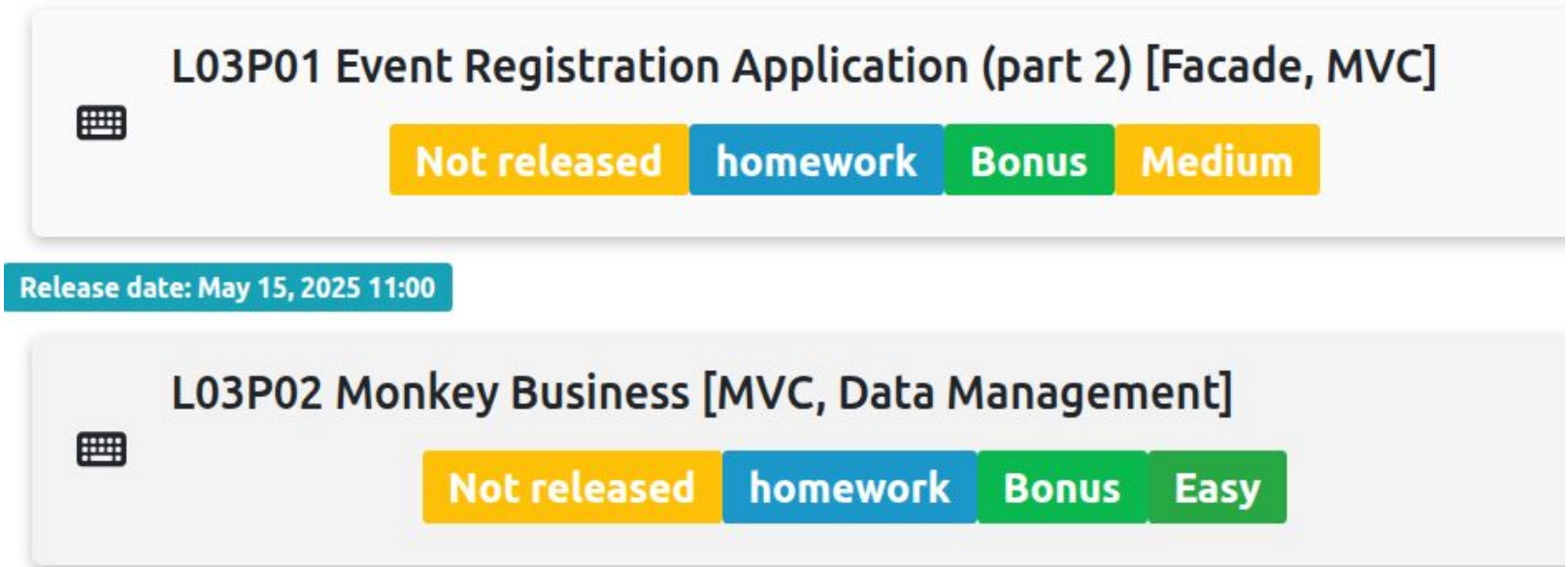
4. Implement the *save* method inside the *EmployeeCreationView* class.

```
private void save() {
    this.employee.setName(employeeNameTextField.getText());
    this.controller.addNewEmployee(employee);
}
```

Tutorial outline

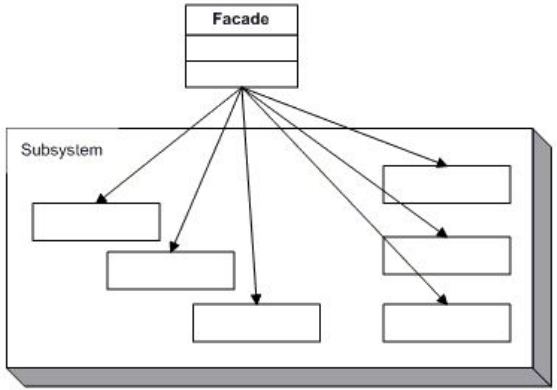
- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~
- ~~**Part II:** Programming basics~~
- **Part III: Homework programming exercises (Artemis)**

Programming (P) exercises



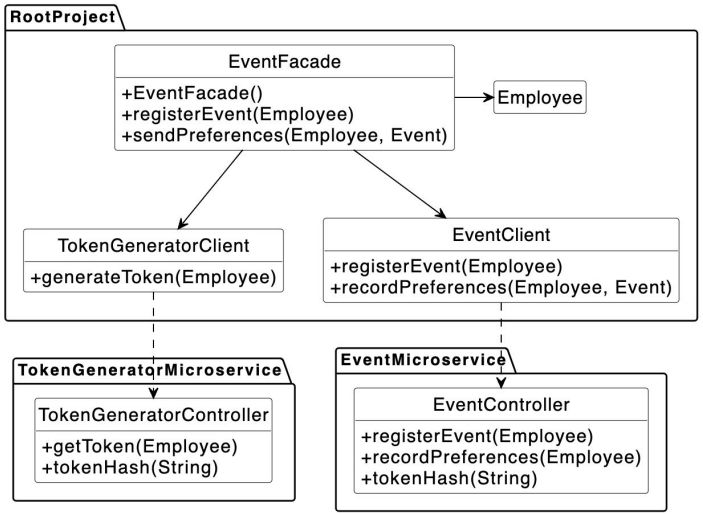
L03P01 Event Registration App (part 2) [Facade, MVC]

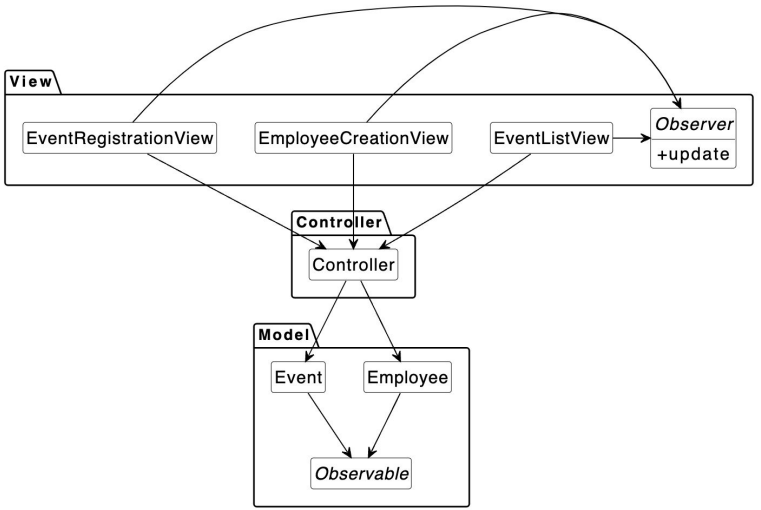
- Tasks and Goals:
  - Develop an application to manage registration and attendee preferences for a private dinner by understanding the Facade pattern and Model View Controller
  - Features:
    - Token generation
    - Identity verification
  - Components:
    - Facade with Access Policy
    - Model-View-Controller (MVC)



L03P01 Event Registration App (part 2) [Facade, MVC]

Example UML Class diagrams:

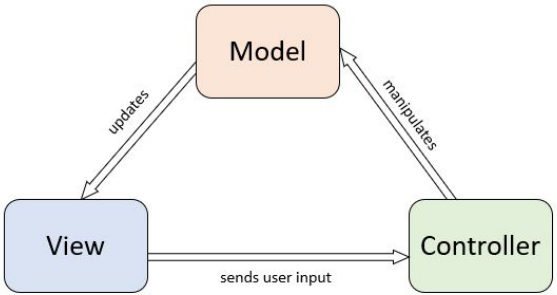




L03P02 Monkey Business [MVC, Data Management]

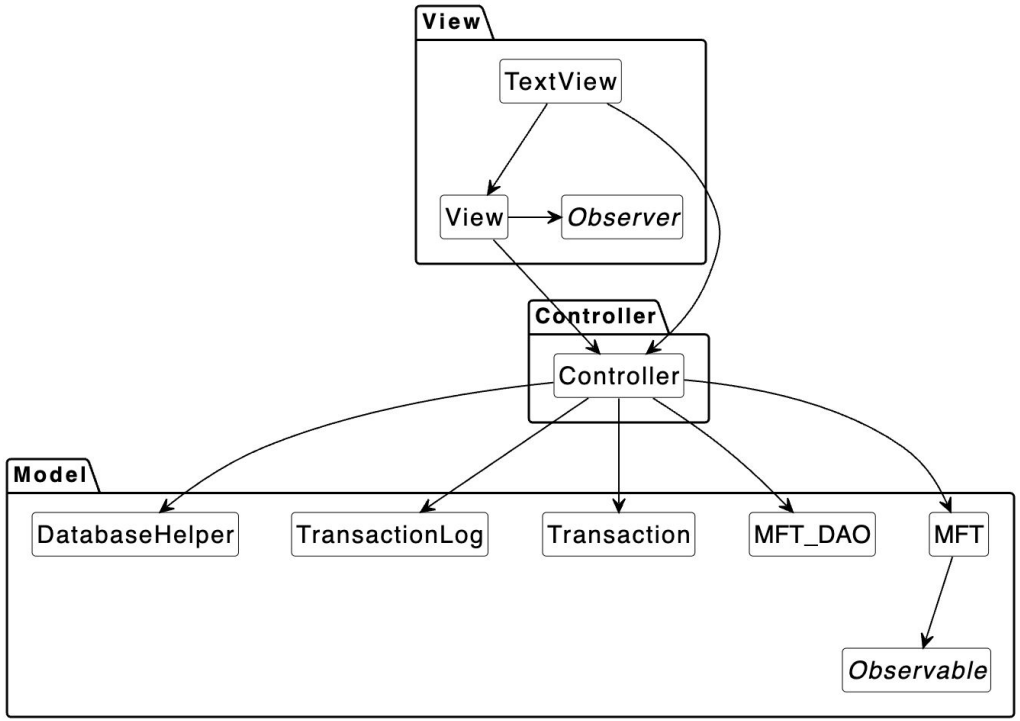
- Tasks and goals:
  - Create a platform for listing, buying, and updating Maybe Fungible Tokens (MFTs)
  - Adoption of Model-View-Controller (MVC) architectural style with a database and transaction log
  - Ensure synchronized views for MFT ownership and pricing updates

- Software:
  - Java
  - In-memory **SQLite** Database

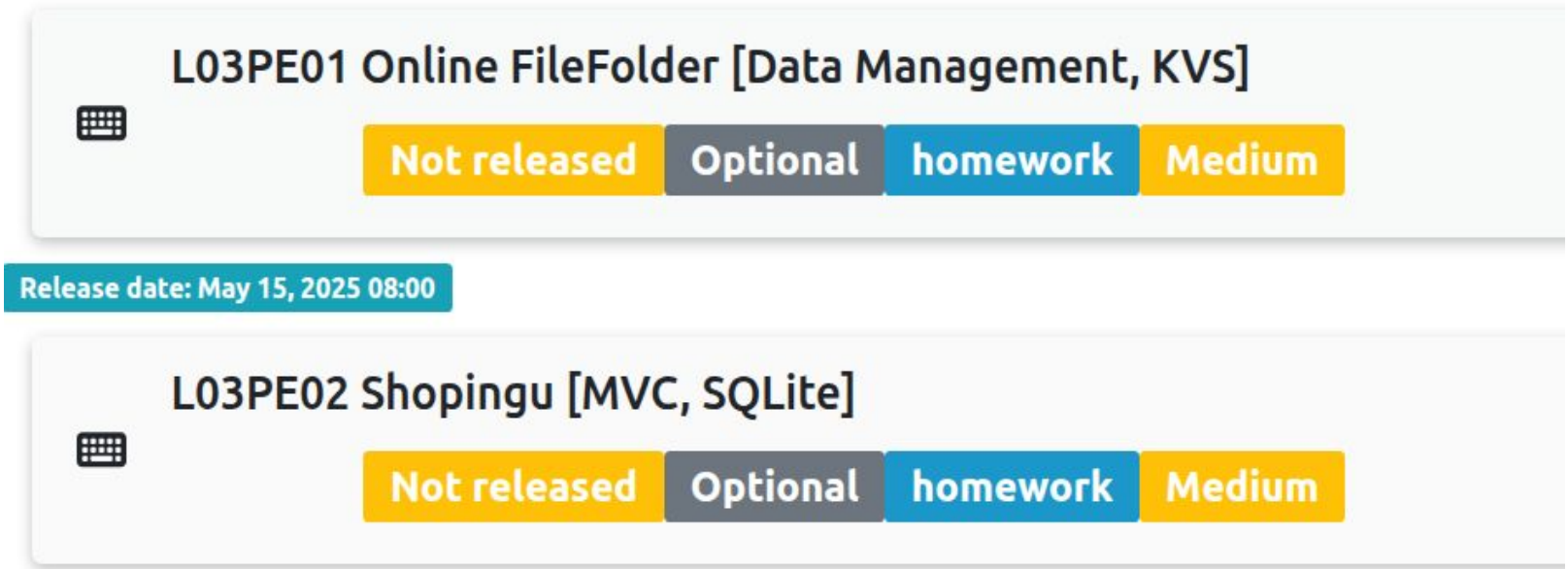


L03P02 Monkey Business [MVC, Data Management]

Example UML Class diagram:



Programming Extras (PE) exercises



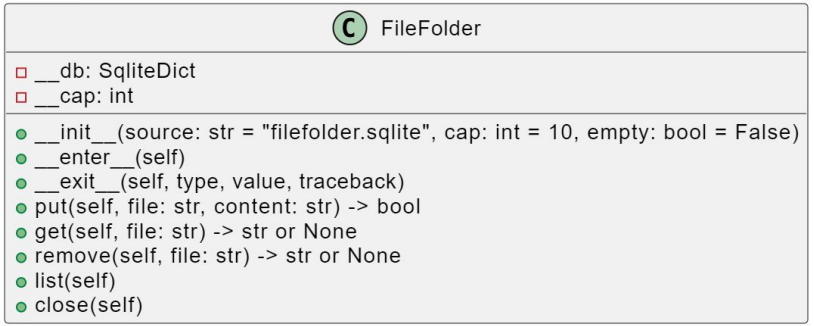
L03PE01 Online FileFolder [Data Management, KVS]

Objectives:

- Implement a basic key-value based file storage utilizing *sqlitedict*.
- Implement CRUD-like methods (**get()**, **put()**, **remove()**, **list()**)

Optional challenge:

- Create a program that uses your KVS through with aforementioned methods.

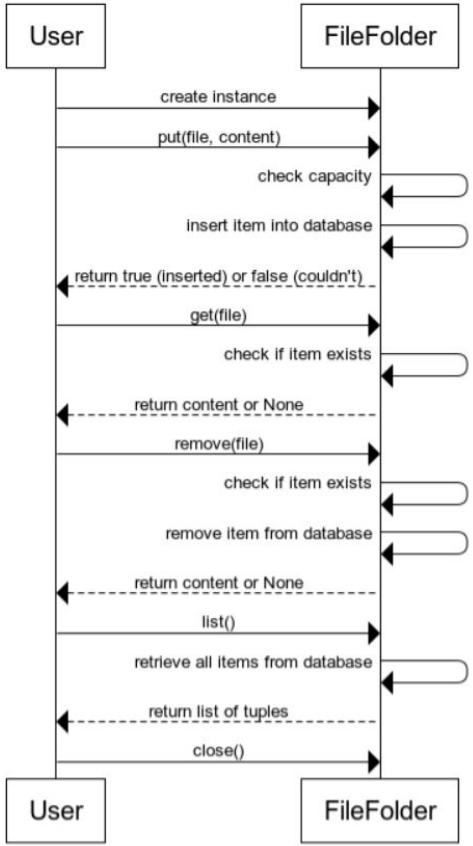


L03PE01 Online FileFolder [Data Management, KVS]

User creates instance of FileFolder (our KVS system)

Users can:

- *Put* ‘files’ (if capacity allows)
- *Get* file contents (if the file exists)
- *Remove* files (is file exists)
- *List* the current files
- *Close* the FileFolder.



L03PE01 Online FileFolder [Data Management, KVS]

SqliteDict provides a dictionary-like interface for SQLite databases.

Example usage:

```
from sqlitedict import SqliteDict # import

#specifies the path to the SQLite file
self.__db = SqliteDict(source)

# clears the sqlite database
self.__db.clear()

# ensures that all changes made to the db are saved permanently
self.__db.commit()

# inserts or updates item in the SqliteDict database with content.
self.__db[item] = content

# deletes entry with the key 'item' from the database.
del self.__db[item]

# retrieves an iterable of key-value pairs from the database.
self.__db.items()

# ensures that all resources associated with the database connection are properly released.
self.__db.close()
```

L03PE02 Shopingu [MVC, SQLite]

Tasks and Goals:

- Create a platform for listing, buying, and managing products in a shopping application for penguins.
- Adoption of Model-View-Controller (MVC) architectural style with JavaFX for the View and an in-memory SQLite database for persistent storage
- Ensure synchronized and consistent views for product listings and updates across admin and customer views