BÁO CÁO BÀI TẬP LỚN

Môn học: Thiết kế xây dựng phần mềm

Giảng viên hướng dẫn: TS. Nguyễn Thị Thu Trang

Nhóm sinh viên

1. Phạm Minh Khiêm - 20170084
2. Lê Vũ Lợi - 20173240
3. Phạm Trung Kiên - 20170088

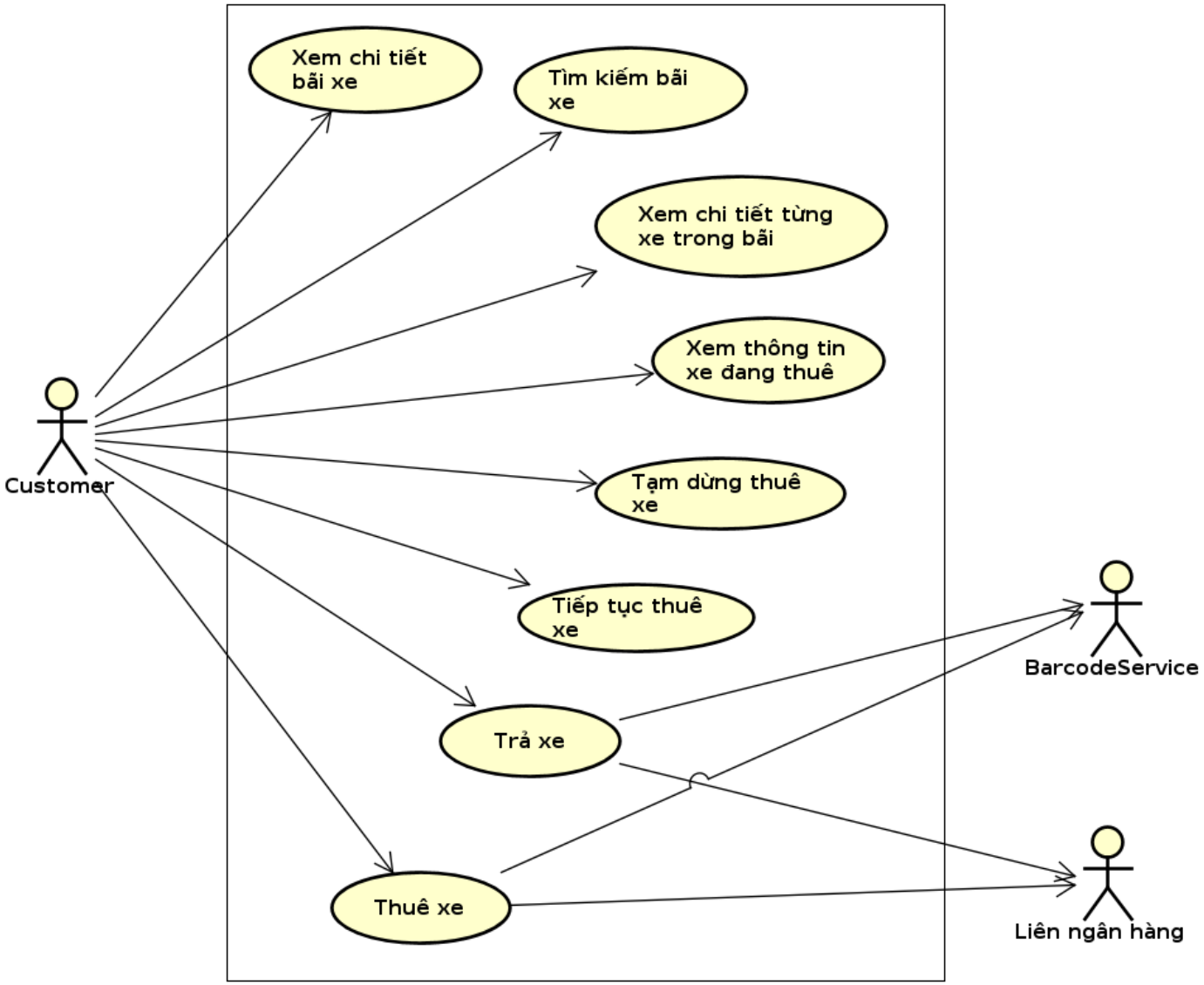
Ngày 24 tháng 12 năm 2020

Phân công công việc

- Lợi: đặc tả usecase Xem chi tiết xe đang thuê, code frontend, thiết kế chi tiết lớp, làm slide. (35%)
- Khiêm: đặc tả usecase Trả xe, code backend, thiết kế giao diện người dùng. (40%)
- Kiên: đặc tả usecase Thuê xe, thiết kế dữ liệu.(25%)

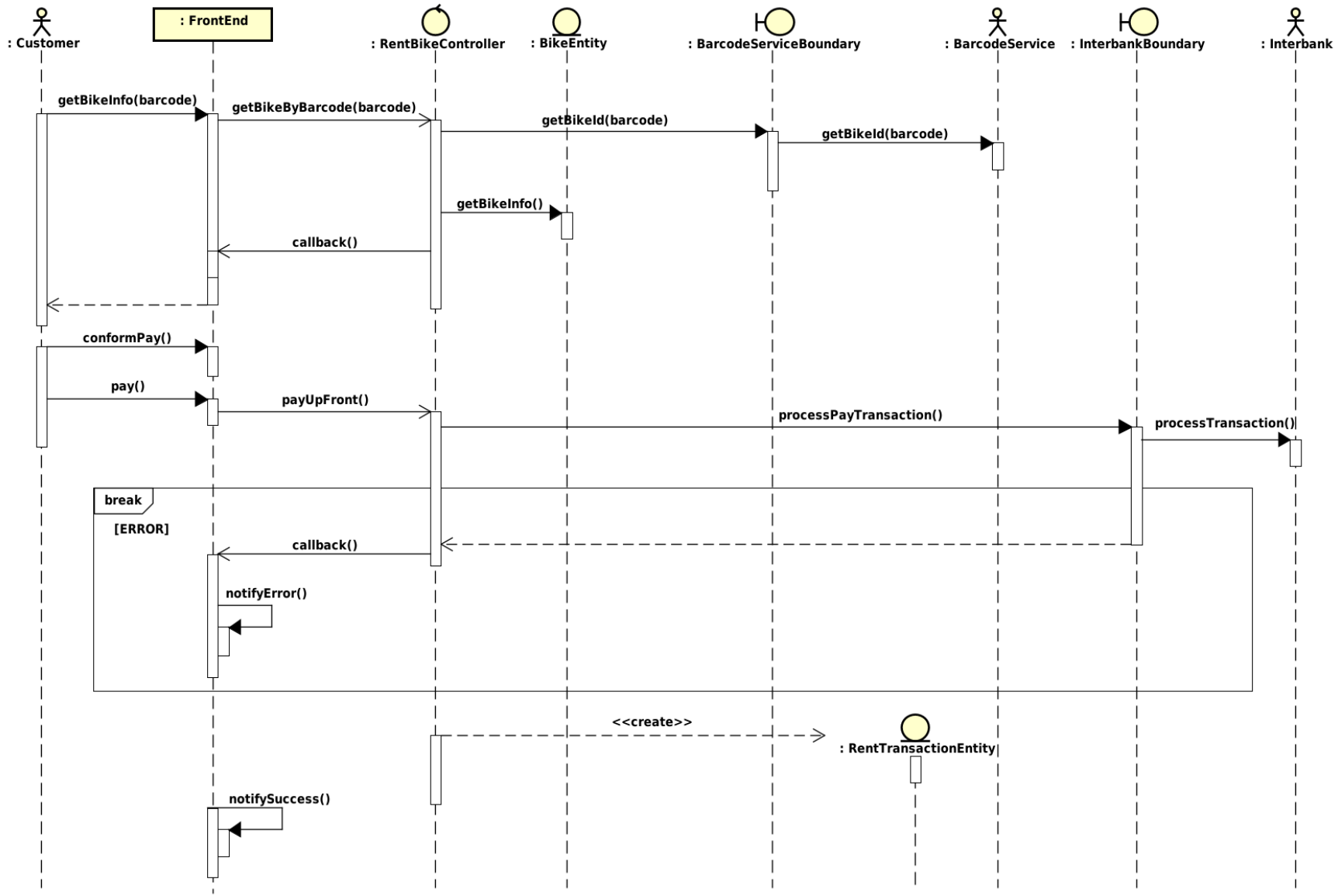
Hệ thống cho thuê xe đạp theo giờ

Biểu đồ usecase tổng quan

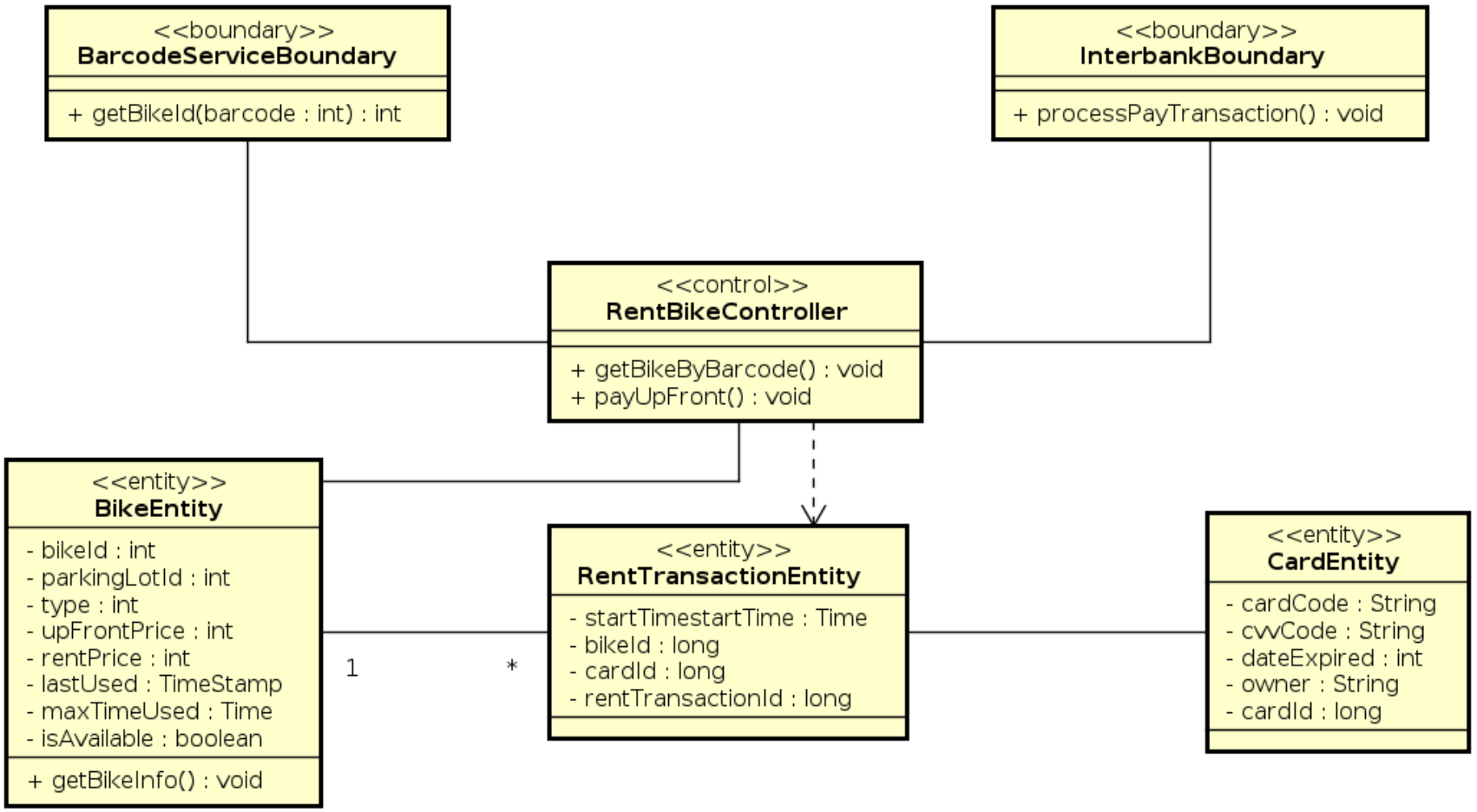


Phân tích usecase thuê xe

Biểu đồ trình tự



Phân tích usecase thuê xe
Biểu đồ lớp



- **Cohesion**: Các phương thức và thuộc tính của một lớp nếu chỉ được gọi đến ngay bên trong lớp thì sẽ được đặt là private. Điều này làm tăng cohesion bên trong mỗi lớp, đồng thời cũng giảm coupling giữa các lớp với nhau. Ví dụ:
  - Lớp *InterbankController* có phương thức *processTransaction* được để là private. Phương thức này được gọi bởi 2 phương thức public khác là *processPayTransaction* và *processReturnTransaction*.
  - Lớp *PaymentController* có các phương thức *payUpFront* và *finalPay* được để là private, lần lượt sử dụng bởi 2 phương thức public khác là *payUpFrontControl* và *finalPayControl*.
  - Trong lớp *HttpConnector*, thuộc tính *client* được để private và chỉ phục vụ mục đích sử dụng bên trong lớp này.

- **Coupling**: Mỗi lớp trong hệ thống đều chỉ cung cấp các API public ra bên ngoài, các dữ liệu và logic được ẩn đi ở bên trong. Điều này làm giảm sự phụ thuộc giữa các lớp với nhau, ví dụ:
  - Trong lớp *PaymentController*, nếu cần thay đổi phương thức *payUpFront*, chỉ cần có một thay đổi nhỏ kéo theo trong phương thức *payUpFrontControl* mà không làm ảnh hưởng đến bất kỳ thành phần nào khác của hệ thống.
  - Giữa lớp *PaymentController* và *InterbankSystemController*, sự phụ thuộc là gián tiếp thông qua interface *InterbankInterface*. Như vậy, khi thay có thay đổi bên *InterbankSystemController*, các thay đổi này sẽ không làm ảnh hưởng đến *PaymentController*.

- **Single responsibility**
  - Trách nhiệm của hệ thống được phân chia cho từng package và cho từng lớp trong package, cụ thể:
    - Package *controller*: làm nhiệm vụ thực thi các logic của toàn bộ hệ thống.
    - Package *utils*: chứa các chức năng tiện ích có thể được dùng bởi nhiêu lớp của nhiều gói khác nhau.
    - Package *entity*: chứa các đối tượng mapping với các bảng trong cơ sở dữ liệu.
    - Package *repository*: chứa các interface điều khiển việc truy vấn cơ sở dữ liệu.
    - Package *Interbank subsystem*: chứa các lớp phục vụ cho việc giao tiếp giữa hệ thống với backend server.

- **Open/closed principle**
  - Nguyên tắc này được thực hiện thông qua việc sử dụng các interface.
  - *Interbank subsystem implement* các phương thức định nghĩa trong *Interbank interface*. Các lớp của hệ thống chỉ phụ thuộc vào Interbank interface chứ không phụ thuộc trực tiếp vào *Interbank subsystem*. Do đó, các sửa chữa và mở rộng sau này đối với *Interbank subsystem* có thể được thực hiện dễ dàng.

- **Liskov substitution principle**
  - Nguyên tắc này yêu cầu một lớp con kế thừa từ một lớp cha sẽ có thể sử dụng để thay thế lớp cha trong bất kì tình huống nào.
  - Để đảm bảo yêu cầu này thì đối với các phương thức ghi đè, tham số truyền vào cho lớp con phải có miền giá trị rộng hơn lớp cha, đồng thời giá trị trả về trong phương thức ở lớp con phải có miền giá trị nhỏ hơn.
  - Trong thiết kế của hệ thống, lớp *PaymentController* có một thuộc tính tên *interbankSubsystem* với kiểu dữ liệu là *InterbankInterface*. Trong mọi tình huống, kiểu dữ liệu của *interbankSubsystem* có thể chuyển thành *InterbankController* hay bất kì lớp nào khác implement *InterbankInterface*.

- **Interface segregation principle**
  - Một interface không nên chứa quá nhiều phương thức, nếu không các implementations của interface đó có thể bị trùng lặp mã nguồn.
  - Trong thiết kế của hệ thống, *InterbankInterface* có 3 phương thức. Các implementations khác của interface nhiều khả năng sẽ khác nhau ở cả 3 phương thức. Do vậy, việc phân chia interface của *InterbankInterface* là hợp lý theo nguyên tắc *Interface segregation*

- **Dependency inversion principle**
  - Các lớp ở tầng trên không nên sử dụng trực tiếp dịch vụ cung cấp bởi các lớp tầng dưới mà nên thông qua một interface.
  - Trong thiết kế hiện tại, các lớp sử dụng Java persistence API (trong gói *repository*) sẽ không cần phải thay đổi nếu hệ thống chuyển từ MySQL sang SQLServer. Các lớp sử dụng gói repository như *PaymentController*, *BikeController* khi cần kết nối đến cơ sở dữ liệu sẽ thông qua đối tượng của gói *repository*.

- **Thư viện Simple Logging Facade for Java - SLF4J**
  - *Singleton design pattern*: `LoggerFactory.getLogger(Class<> clazz)` - trả về một logger tương ứng với tên lớp được truyền vào.
  - *Facade design pattern*: Toàn bộ logic phức tạp của thư viên được đóng gói và cung cấp một API cho người dùng để có thể dễ dàng sử dụng.
  - *Factory method*: SLF4J sử dụng *LoggerFactory* để tạo các logger. Người dùng có thể hoàn toàn không quan tâm đến đến các logger được tạo ra mà chỉ cần biết cách để sử dụng chúng.

- **Thư viện Java Persistence API - JPA**
  - *Singleton design pattern*: Các đối tượng *repository* được gắn với annotation **@Autowired**. Nếu có 2 lớp cùng sử dụng đối tượng của một lớp *repository*, đối tượng repository này sẽ chỉ được tạo ra một lần cho cả 2 lớp.