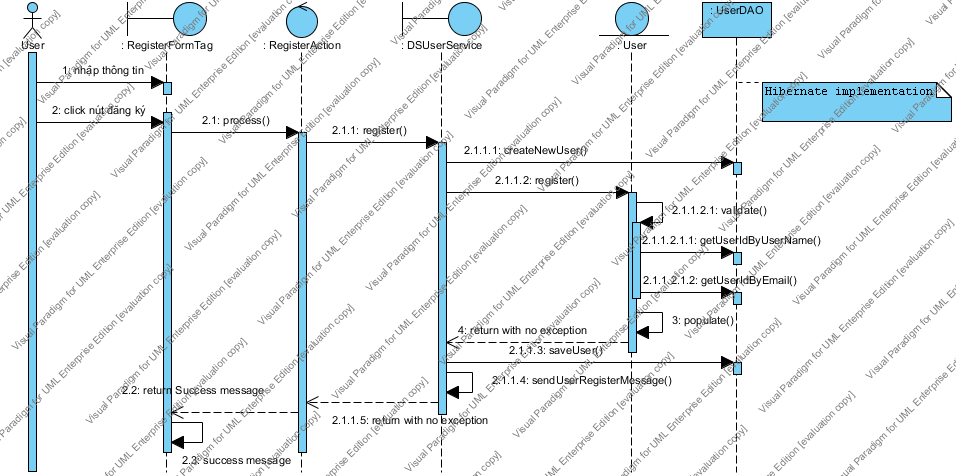
- Sequence Diagram cho tác xác định được các lớp và tương tác giữa các lớp (các lớp gọi các phương thức của nhau), nhưng chưa xác định được các thuộc tính

- Để xác định chính xác các lớp cần sang class diagram: khi tác xem xét các phương thức vừa tạo ở sequence diagram: từ đó có thể suy ra chính xác mối quan hệ: dependency (các lớp tự tạo reference hoặc có reference qua tham số của các method), association (lưu giữ reference dưới dạng các instance variable), aggregation (chặt hơn association, quan hệ kết tập, từng thành phần tạo nên tính chất của đối tượng cha), composite (chặt hơn aggregation, thành phần không tồn tại riêng rẽ vì tồn tại riêng rẽ không còn ý nghĩa)

# Register\_NormalFlow



- RegisterFormTag 🡪 system boundary: UI - Custom Tag

- RegisterAction 🡪 control : Struts 2 Action 🡪 chỉ là plain old java object, không bị phụ thuộc vào framework hay servlet api

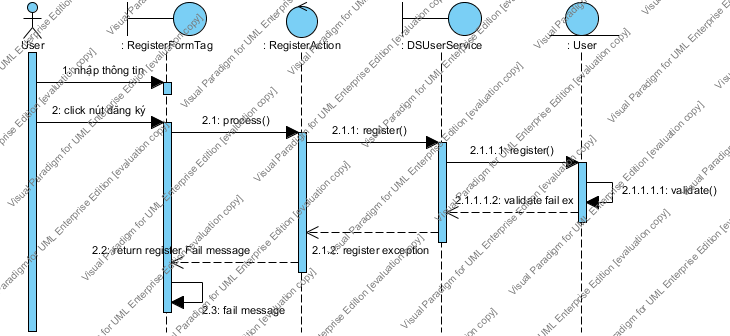
- Presentation validation: được thực hiện bởi validation framework 🡪 được thực hiện ngay trên value object (VO được chuyển từ view – customtag lên action qua các interceptor của struts 2 framework, trong quá trình chuyển sẽ thực hiện validate ngay trên các VO dựa trên các file xml definition 🡪 validate trên VO có thể được dùng lại hơn là action).

- Từ DSUserService 🡪 system boundary chúng ta chuyển sang Bussiness logic tier

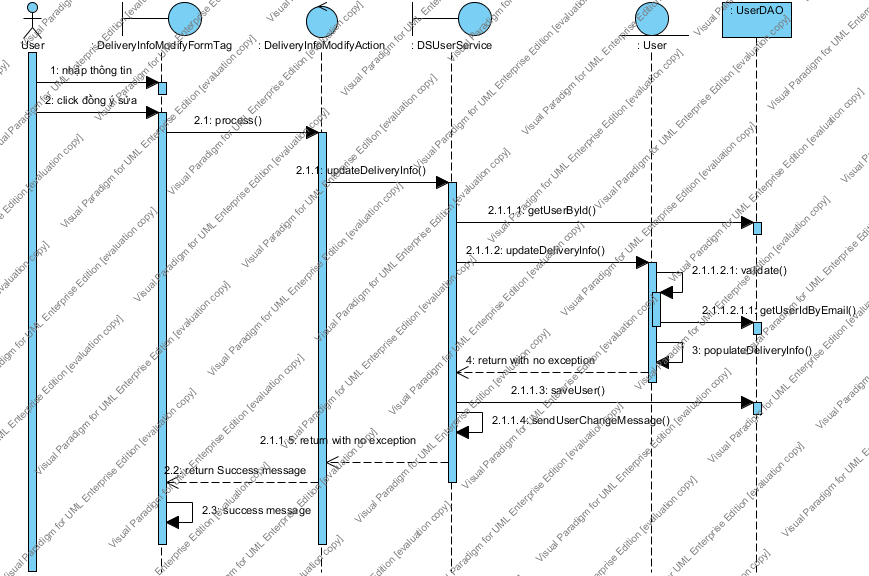
- DSUserService, UserDAO: interface 🡪 strategy design pattern

- Các thao tác logic, validate được thực hiện trên entity User 🡪 entity

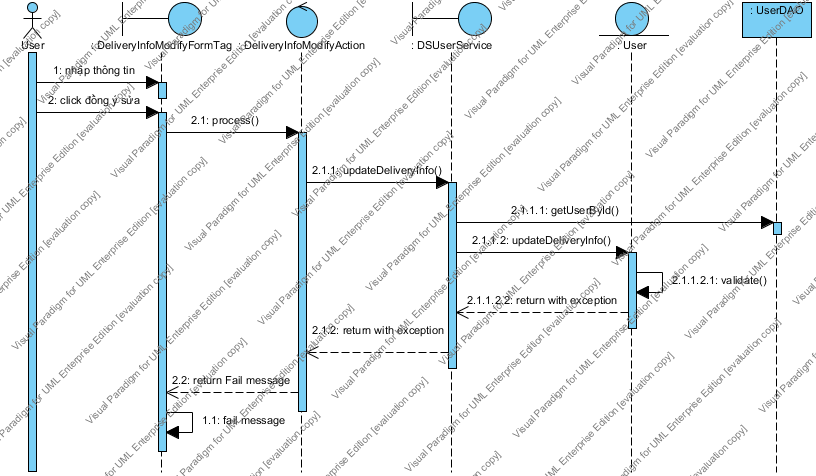
# Register\_ValidateFail



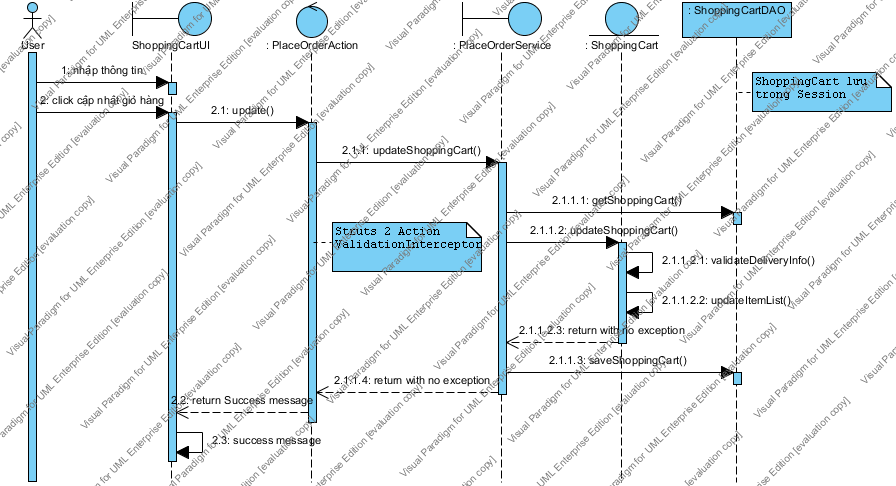
# ThayDoiThongTinCaNhan\_NormalFlow



# ThayDoiThongTinCaNhan\_ValidateFail



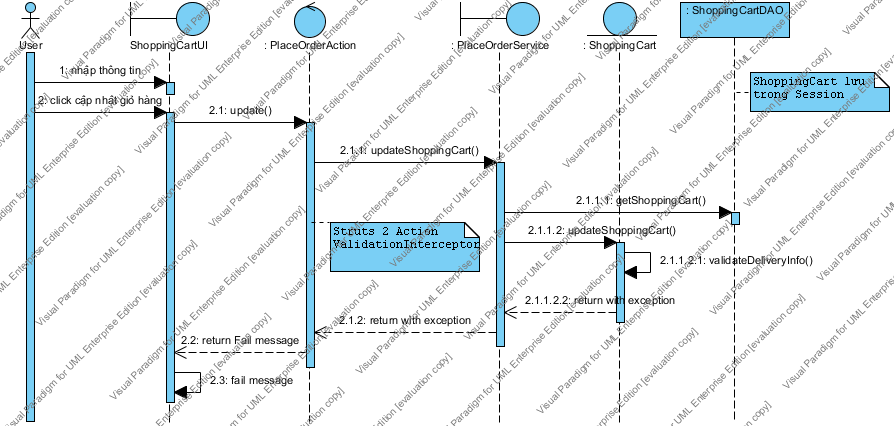
# CapNhatGioHang\_NormalFlow



- ShoppingCart ở đây chưa đề cập đến implementation : có thể lưu trong session, có thể trong bộ nhớ kiểu threadlocal, có thể lưu trong database nhưng ở đây, sequence diagram về cấu trúc không khác gì các sequence khác

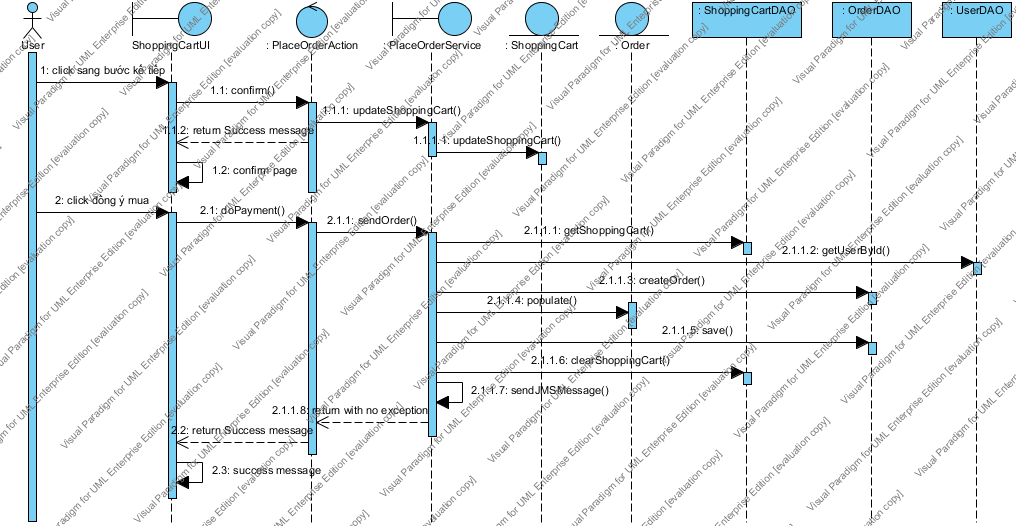
- Thực hiện stategy design pattern : ngay cả trên bussiness logic như PlaceOrderService hay thậm chí entity ShoppingCart không hề biết thông tin được lưu ở đâu, developer cho đến giờ có thể thực hiện lưu trong CSDL hoặc khác, các phần còn lại của hệ thống không bị phụ thuộc vào DAO và vì thế dễ thực hiện hơn: được thực hiện 1 cách riêng rẽ, có thể unit test (không cần CSDL, không phụ thuộc vào servlet API) và nếu có thay đổi thì các lớp này cũng có thể được dùng lại

# CapNhatGioHang\_ValidateFail



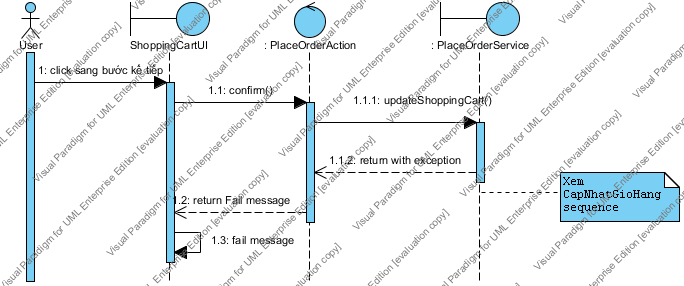
Sequence Diagram

# GuiDonHang\_NormalFlow

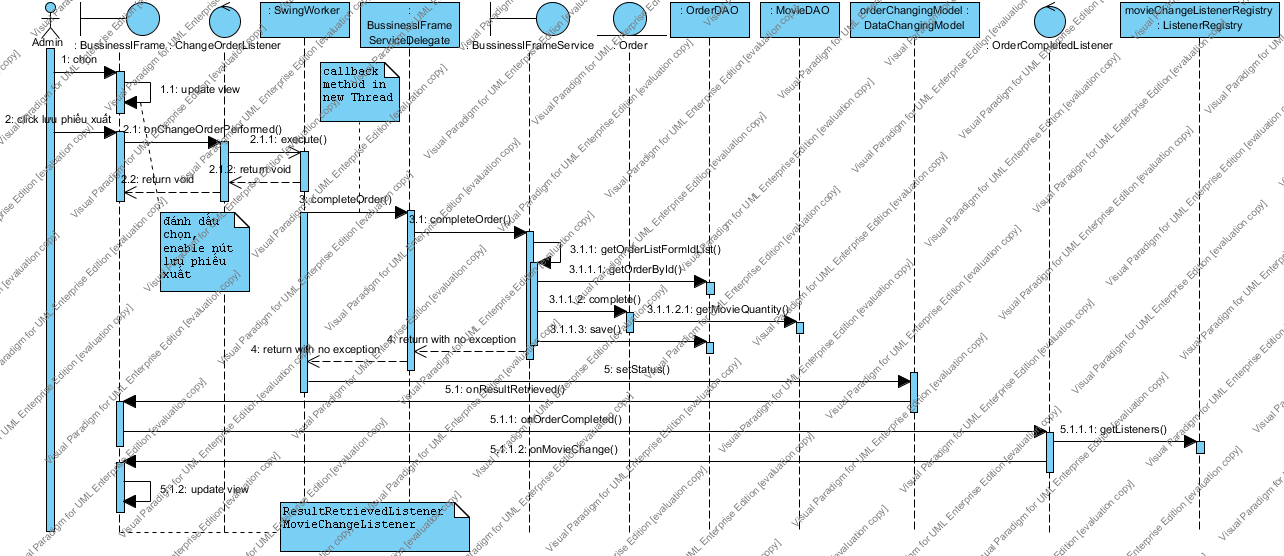


Sequence Diagram

# GuiDonHang\_ConfirmFail



# LuuPhieuXuat\_NormalFlow



- MVC thực chất là việc tách các chức năng hay thay đổi thành các lớp riêng rẽ và làm giảm sự phụ thuộc giữa các lớp này: View gửi message tới control, control cập nhật model, model notify View

- Áp dụng design pattern này có thể thấy lợi ích rất rõ ràng: tách các thành phần hay thay đổi, thực thi các thành phần riêng rẽ. Khi 1 thành phần thay đổi thì các phần còn lại không bị thay đổi theo.

- Tuy nhiên việc áp dụng 1 design pattern không thể cứng nhắc, mà sẽ áp dụng 1 cách linh hoạt tùy vào yêu cầu của bài toán. Ví dụ: không phải lúc nào cũng tách hoàn toàn MVC riêng rẽ. Và việc xem xét các thành phần cần xem xét dưới các góc nhìn khác nhau. Ví dụ: Với 1 JFrame 🡪 là một đối tượng chứa nhiều đối tượng khác, form, input control... bản thân các input control nếu phức tạp có thể thực thi dưới dạng mvc (chủ yếu để tách view ra khỏi control và model do view dễ thay đổi) , bản thân JFrame là một view, tuy nhiên có thể implement ActionListener để trở thành một control, đối với component mà nó chứa, tuy nhiên với đối tượng như JButton hay TextField không phải lúc nào cũng mvc.

- MVC: Presentation tier: View 🡪 BussinessIFrame, Model 🡪 Value Object, DataChangingModel (View căn cứ model này để render: thông báo lỗi, thành công, cập nhật bảng...), Control 🡪 ChangeOrderListener, OrderCompletedListener

Bussines Logic: View 🡪 BussinessIFrameService (interface), Control 🡪 Service Implementation, Model 🡪 remain

- SwingWorker: Multithreading, inner class ChangeOrderListener, execute() mở new thread và trả về ngay void, sử dụng SwingWorker vì ta đang làm việc với Swing, các đối tượng Swing không threadsafe.

- Việc sử dụng multithreading sẽ giúp không khóa event dispatch thread, nếu thread này quá bận, user sẽ loss control đối với chương trình

- ChangeOrderListener được Spring inject BussinessIFrameServiceDelegate (interface) 🡪 ở đây spring giúp thực thi Service Locator design pattern, đây là design pattern giúp tăng tính flexible của hệ thống, do việc locate (xác định) đối tượng service có thể thay đổi, việc sử dụng service locator có thể giúp developer thay đổi thông tin này chỉ ở 1 chỗ, thậm chí nếu sử dụng spring, việc thay đổi được thực hiện thông qua file xml config.

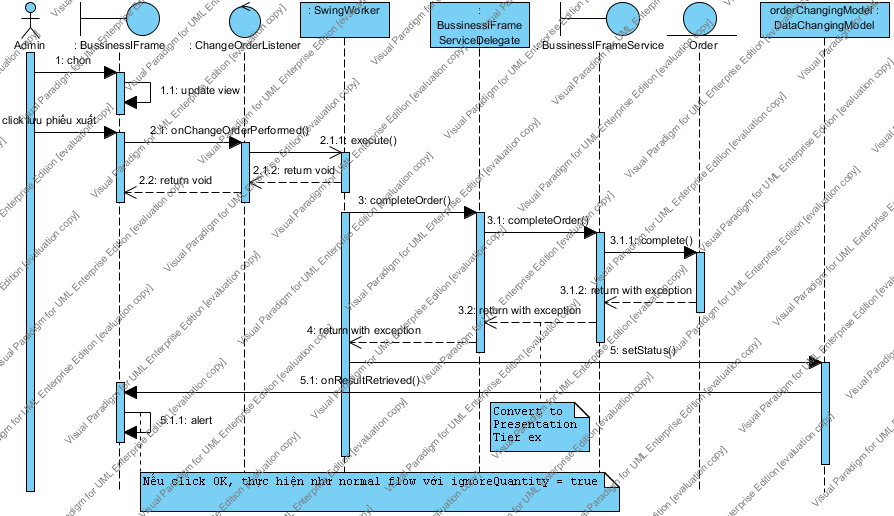
- movieChangeListenerRegistry: do thực thi MDI, có nhiều cửa sổ cần cập nhật đây là nơi chứa các listener lắng nghe sự kiện. Cửa sổ nào cần cập nhật, khi bật lên (bắt sự kiện opened) sẽ tự đăng ký với Registry, khi tắt đi cũng sẽ thông báo Registry xóa thông tin listener. Việc này giúp tạo ra sự độc lập cho mỗi cửa sổ cần cập nhật dữ liệu

- Việc sử dụng interface BussinessIFrameServiceDelegate giúp giảm sự phụ thuộc của Presentation Tier vào Bussiness Logic tier, do Presentation Tier chỉ làm việc với delegate interface (gọi các phương thức của interface chứ không gọi phương thức của BussinessService, ví dụ khi gọi service là remote service, có thể xuất hiện remote exception, nhưng khi service là local, không còn exception này, việc của delegate là đảm bảo presentation tier chỉ phải làm việc với 1 interface duy nhất: delegate interface). Khi Bussiness Service thay đổi, chỉ cần thay đổi Delegate Implementation.

- BussinessIFrameService (Interface) cũng là cố gắng làm giảm sự phụ thuộc giữa Presentation tier và BL Tier, khi có sự thay đổi, gắng không thay đổi interface, việc sử dụng Interface giúp tạo ra sự độc lập giữa P Tier và BL Tier, developer khi thực thi có thể xây dựng P tier riêng rẽ, không cần có BL Tier bằng cách xây dựng các Service Implementation giả, hoặc sử dụng Mock Object khi Junit test. Hơn nữa việc sử dụng Interface giúp làm việc với spring framework.

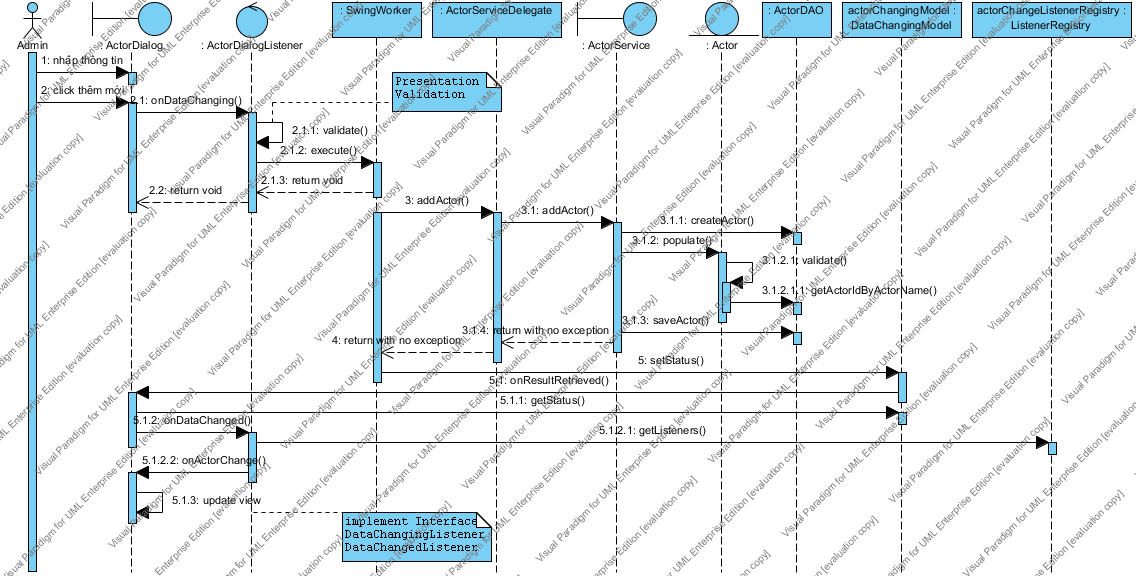
Sequence Diagram

# LuuPhieuXuat\_NotEnough



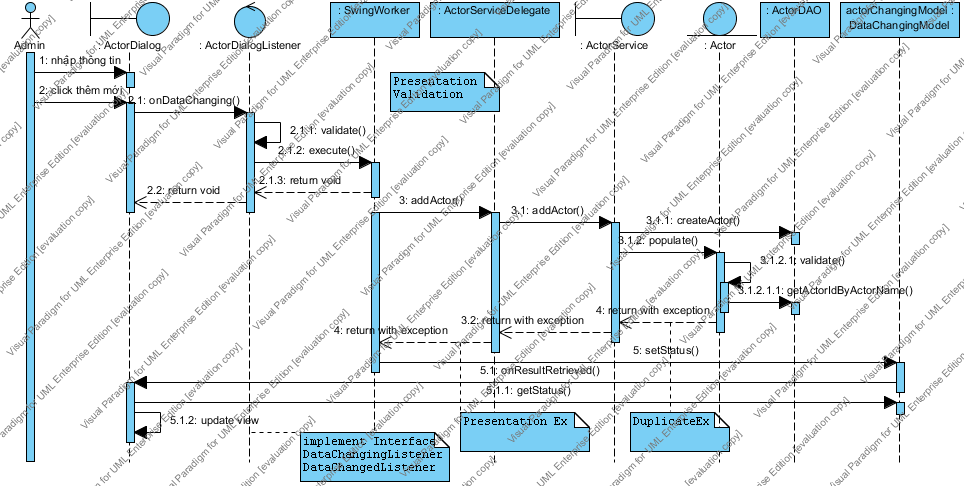
Sequence Diagram

# ThemDienVien\_NormalFlow



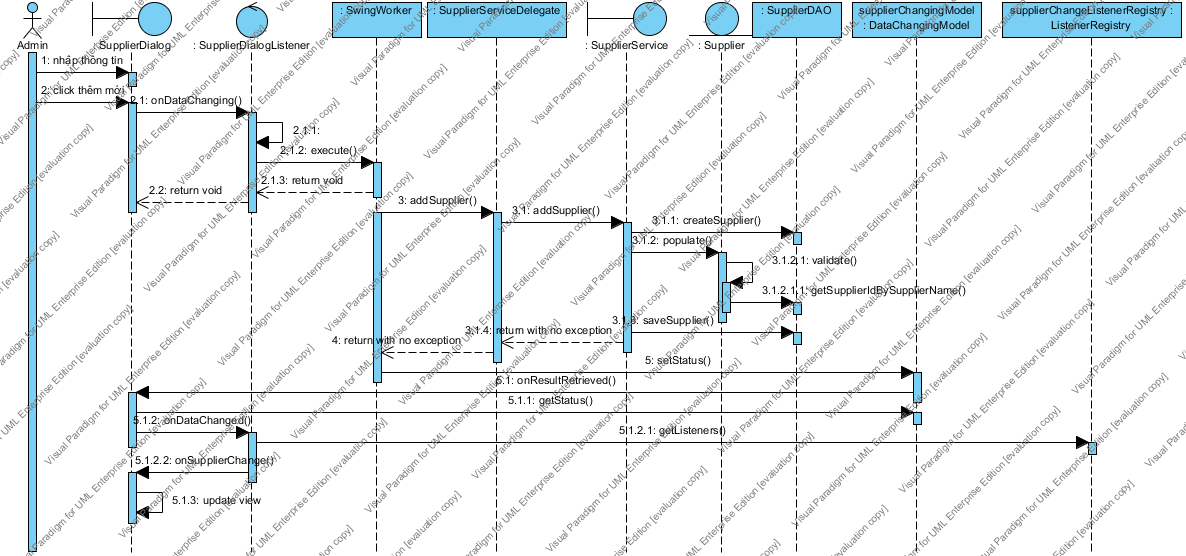
Sequence Diagram

# ThemDienVien\_Duplicate



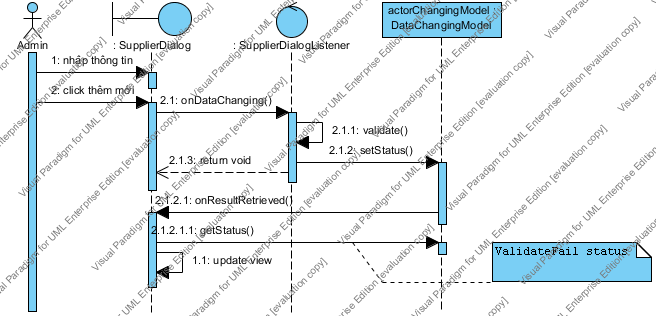
Sequence Diagram

# ThemNhaCungCap\_NormalFlow



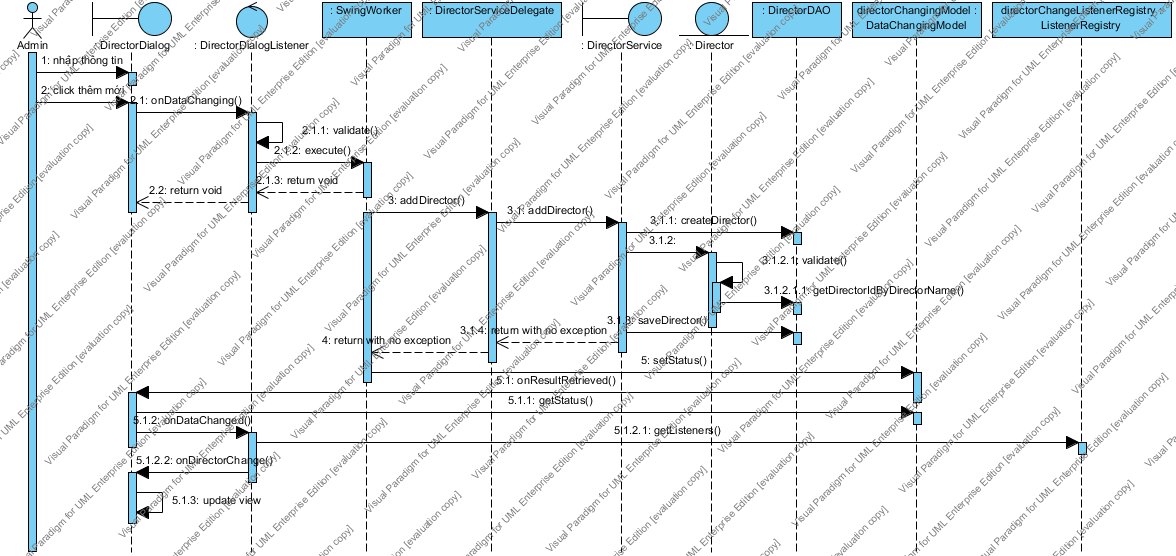
Sequence Diagram

# ThemNhaCungCap\_ValidateFail



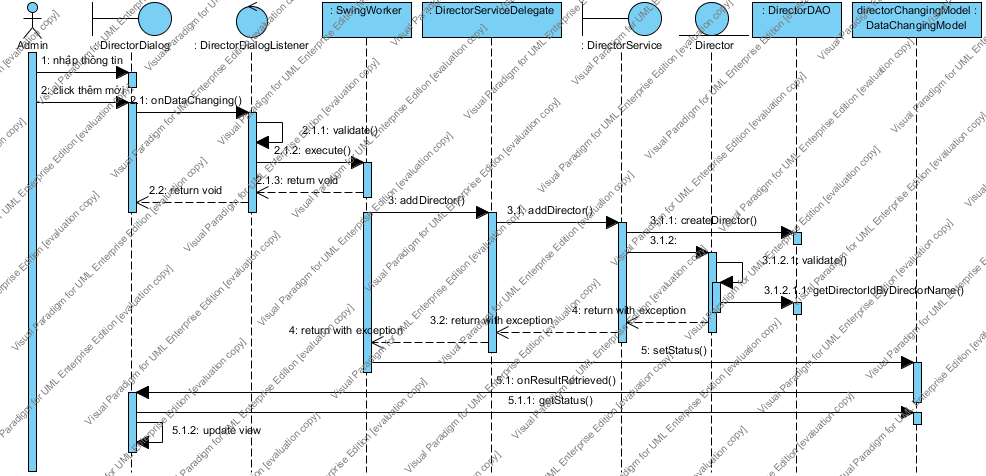
Sequence Diagram

# ThemDaoDien\_NormalFlow



Sequence Diagram

# ThemDaoDien\_Duplicate



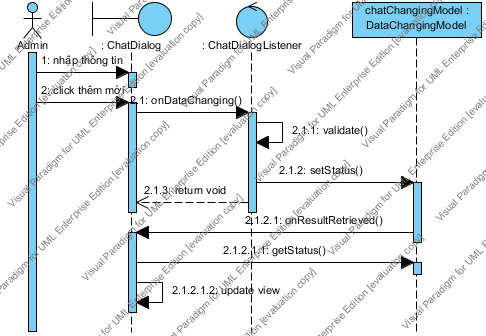
Sequence Diagram

# ThemNickChat\_NormalFlow



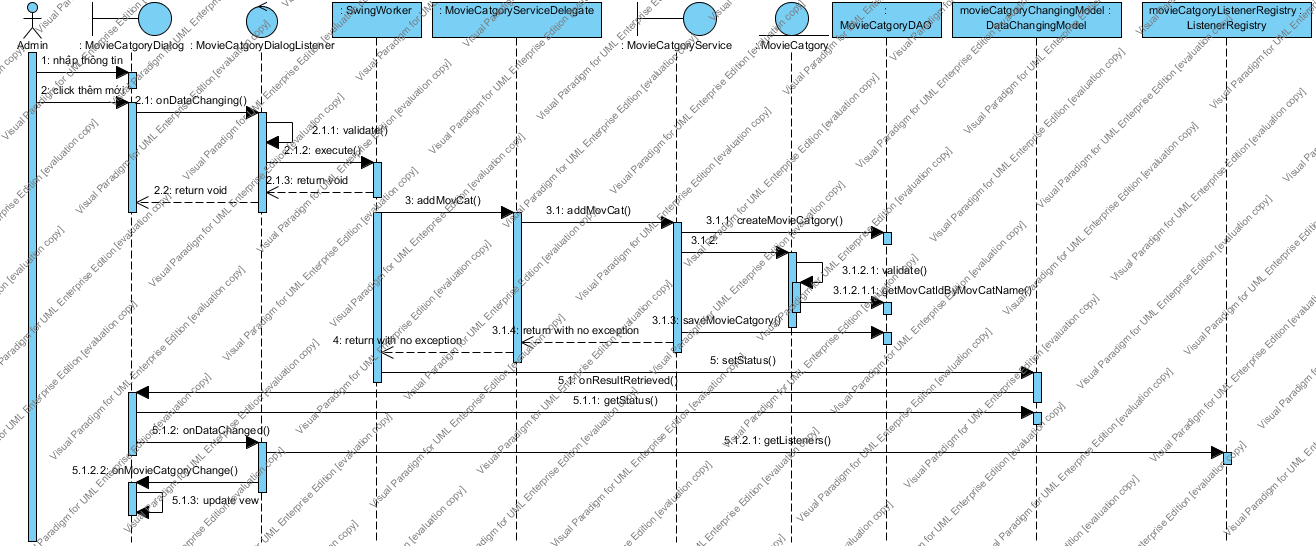
Sequence Diagram

# ThemNickChat\_ValidateFail



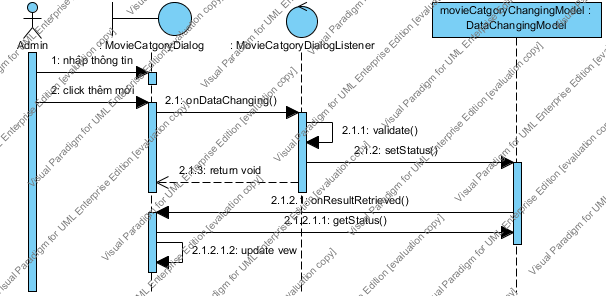
Sequence Diagram

# ThemLoaiPhim\_NormalFlow



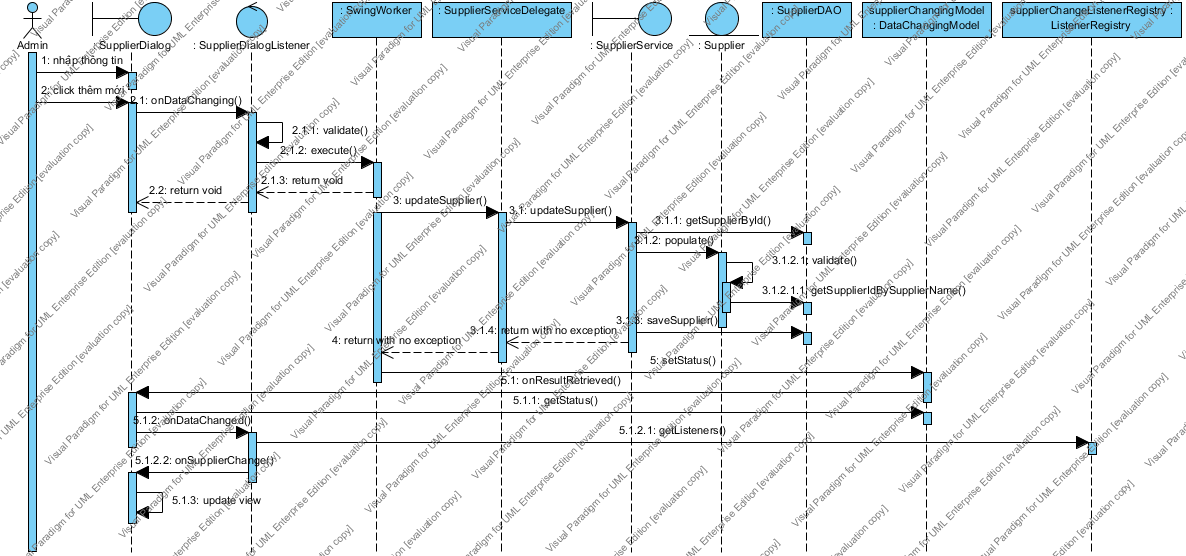
Sequence Diagram

# ThemLoaiPhim\_ValidateFail



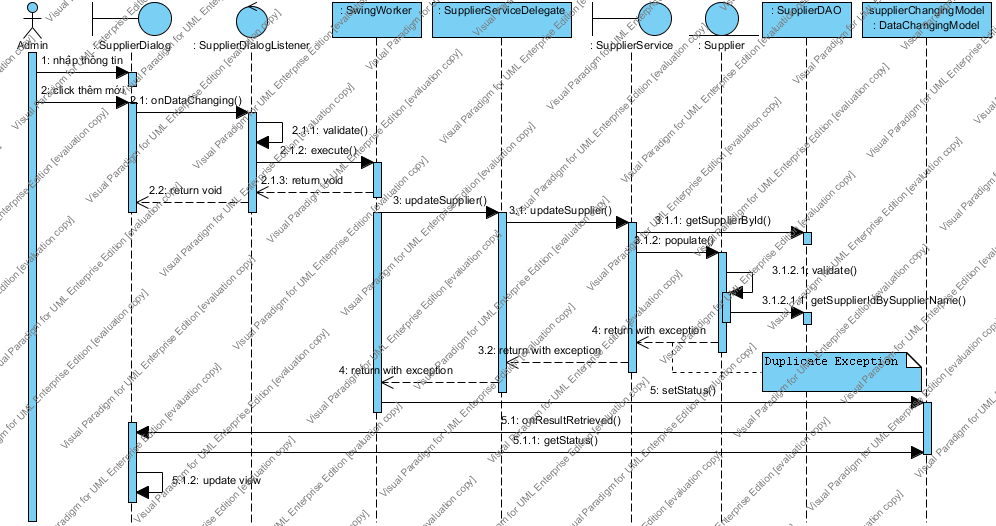
Sequence Diagram

# SuaNhaCungCap\_NormalFlow



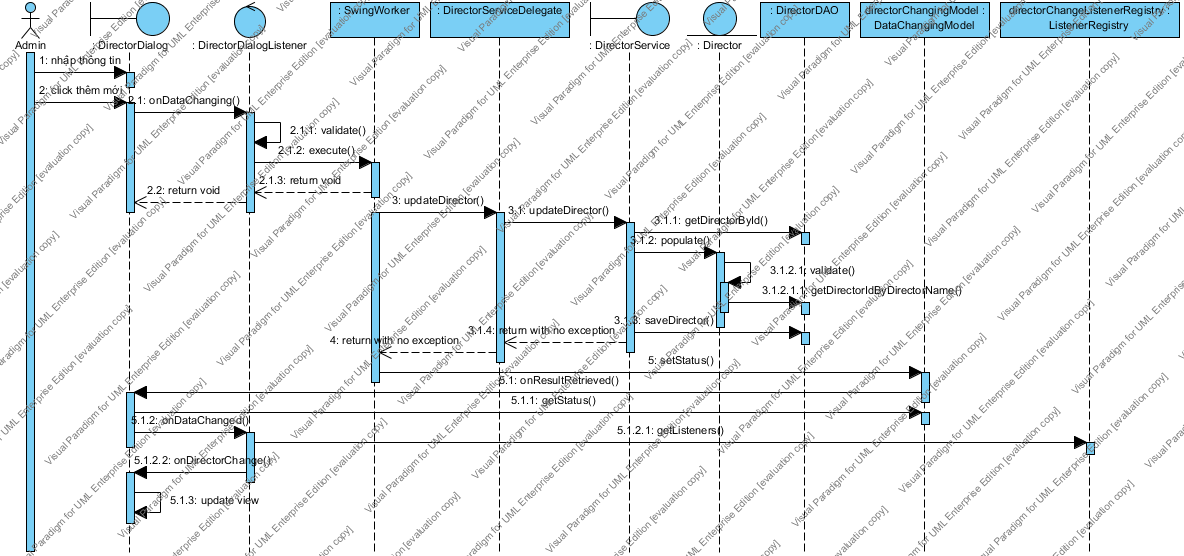
Sequence Diagram

# SuaNhaCungCap\_Fail



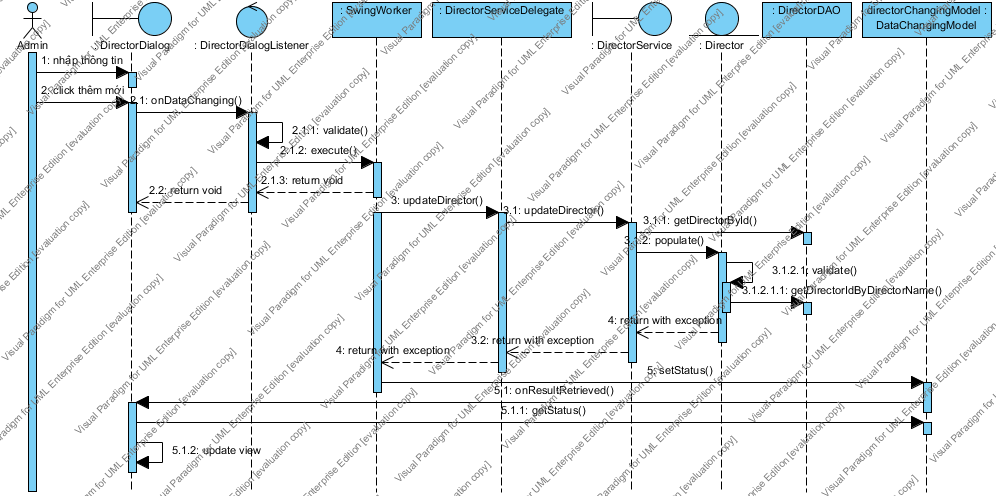
Sequence Diagram

# SuaDaoDien\_NormalFlow



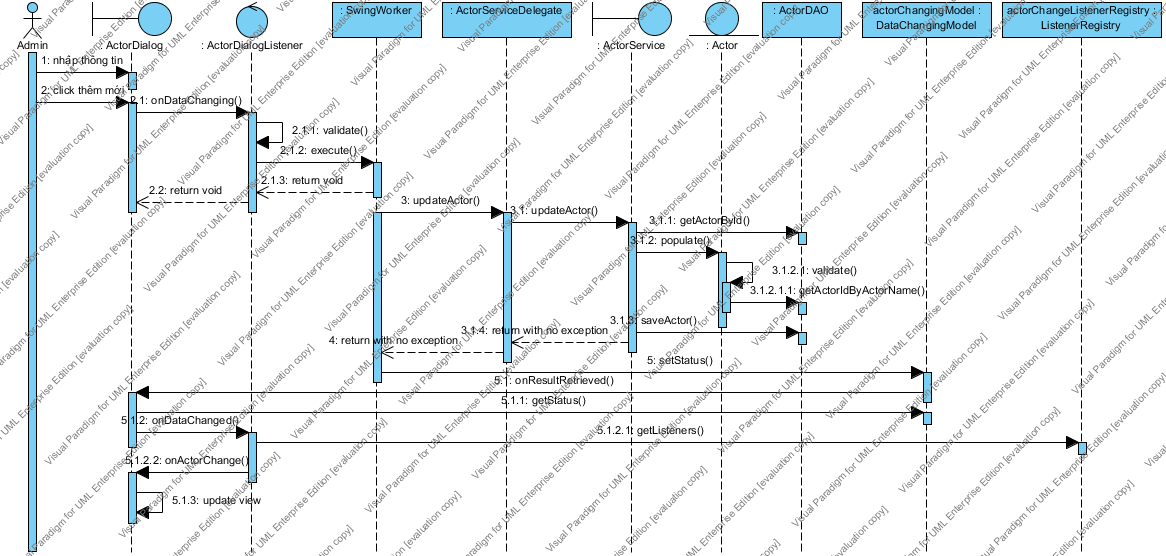
Sequence Diagram

# SuaDaoDien\_Fail



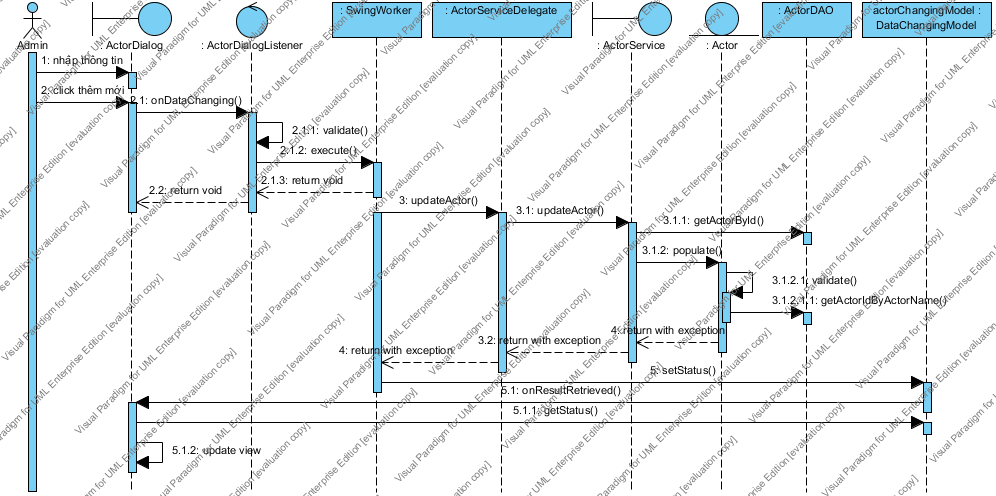
Sequence Diagram

# SuaDienVien\_NormalFlow



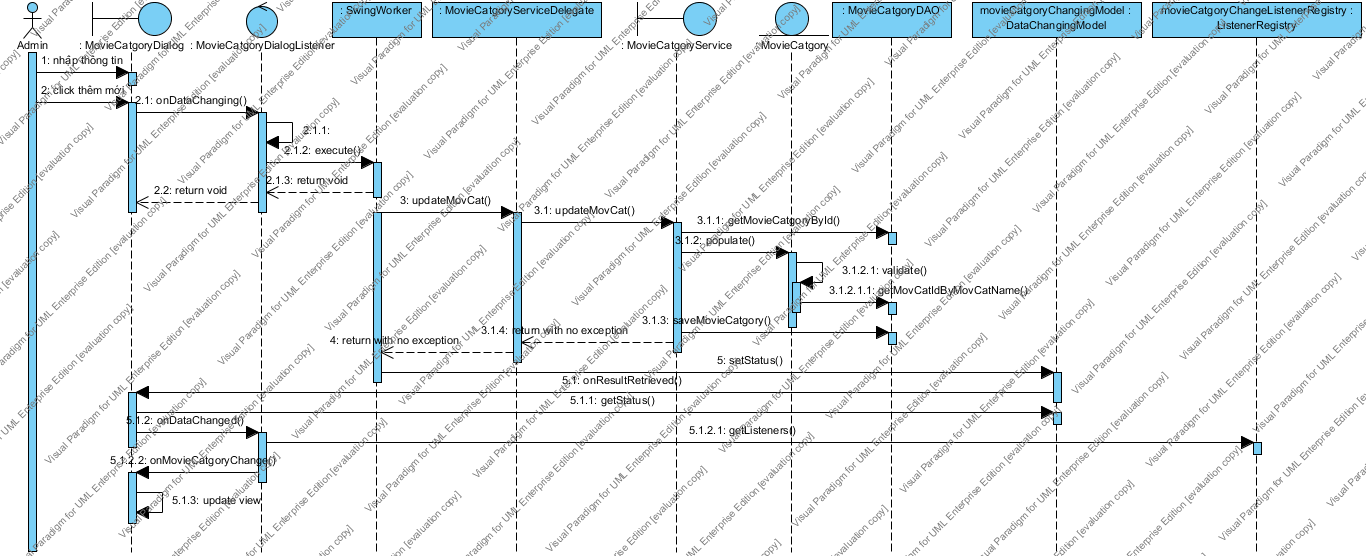
Sequence Diagram

# SuaDienVien\_Fail



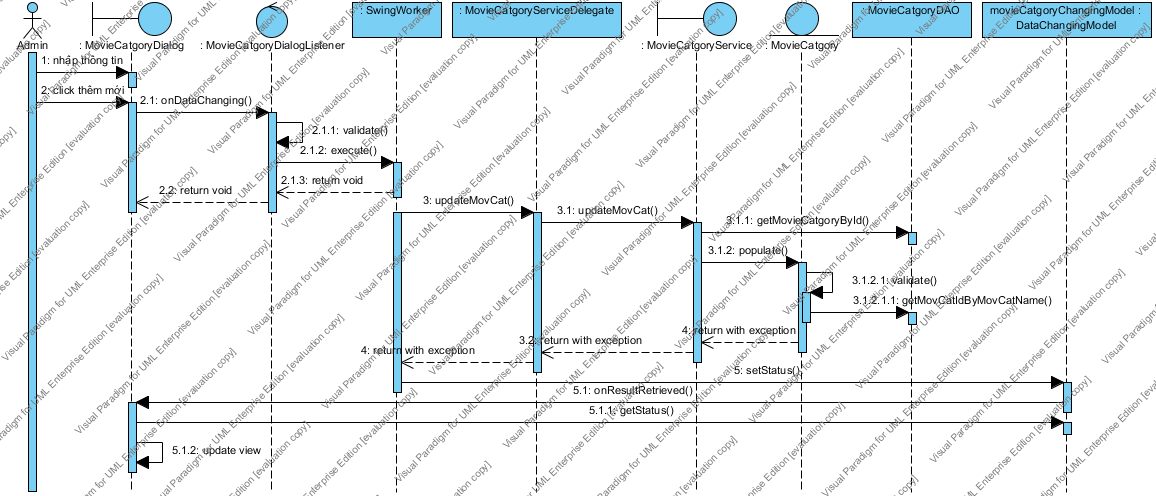
Sequence Diagram

# SuaLoaiPhim\_NormalFlow



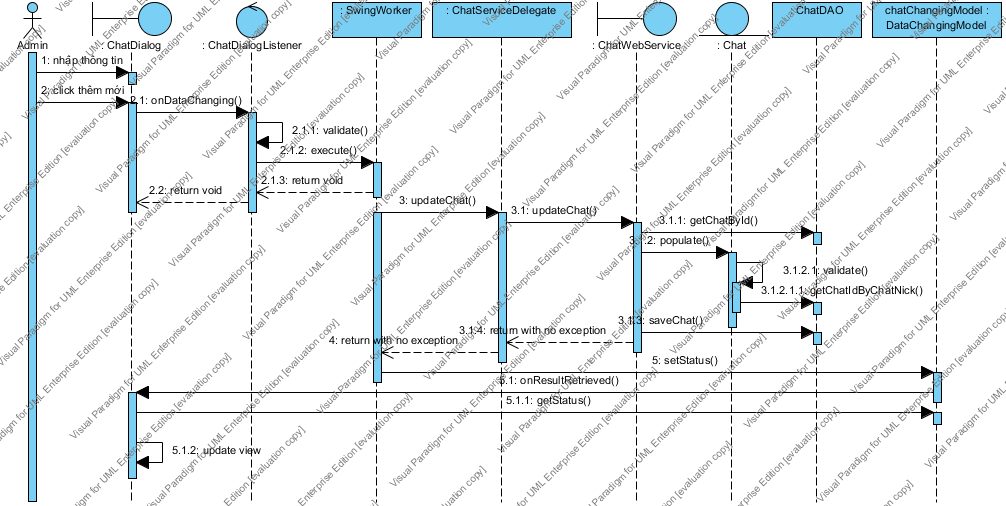
Sequence Diagram

# SuaLoaiPhim\_Fail



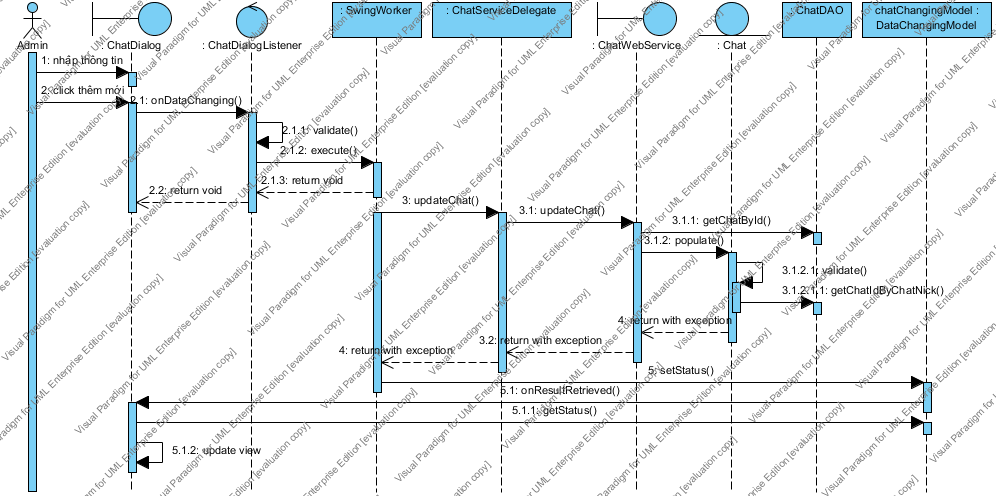
Sequence Diagram

# SuaNickChat\_NormalFlow



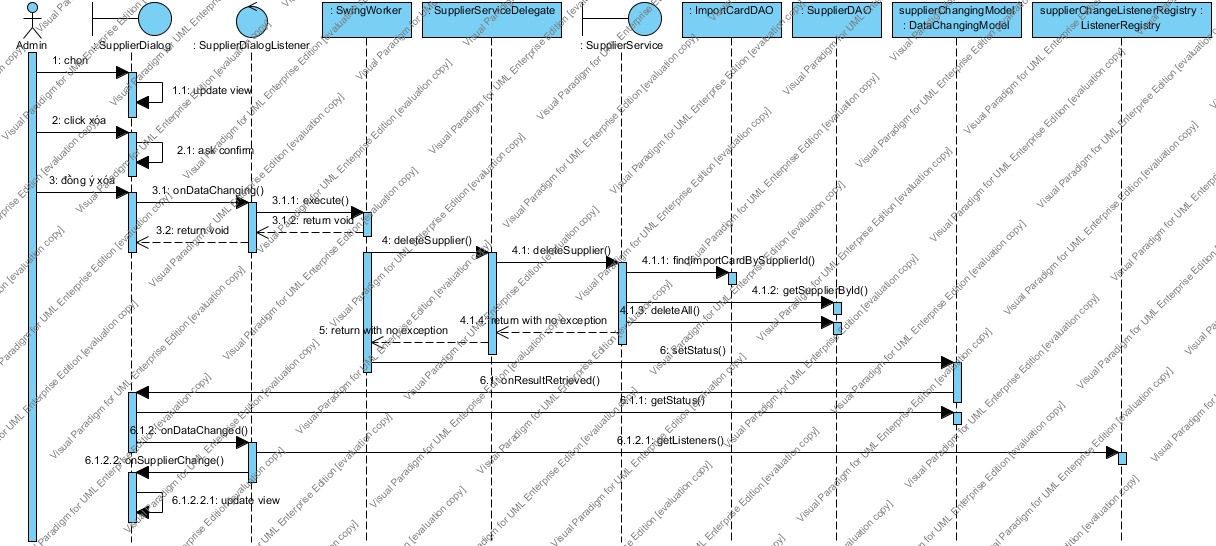
Sequence Diagram

# SuaNickChat\_Fail



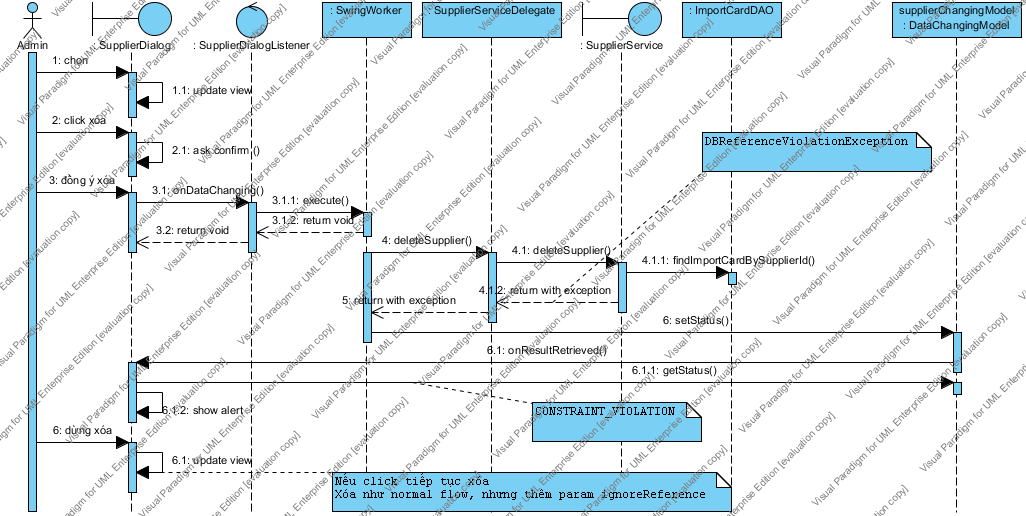
Sequence Diagram

# XoaNhaCungCap\_NormalFlow



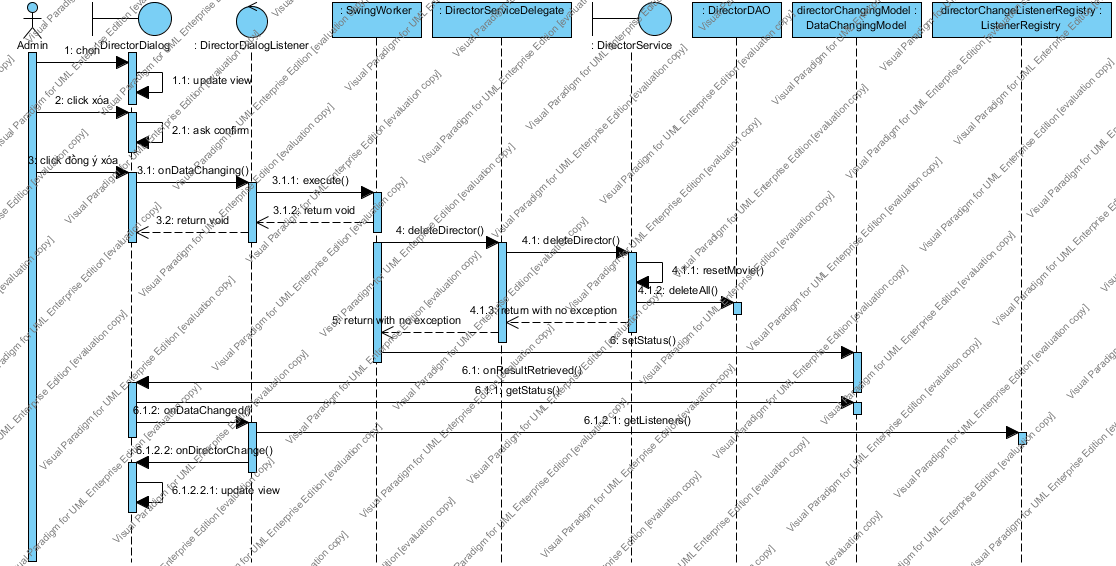
Sequence Diagram

# XoaNhaCungCap\_Fail



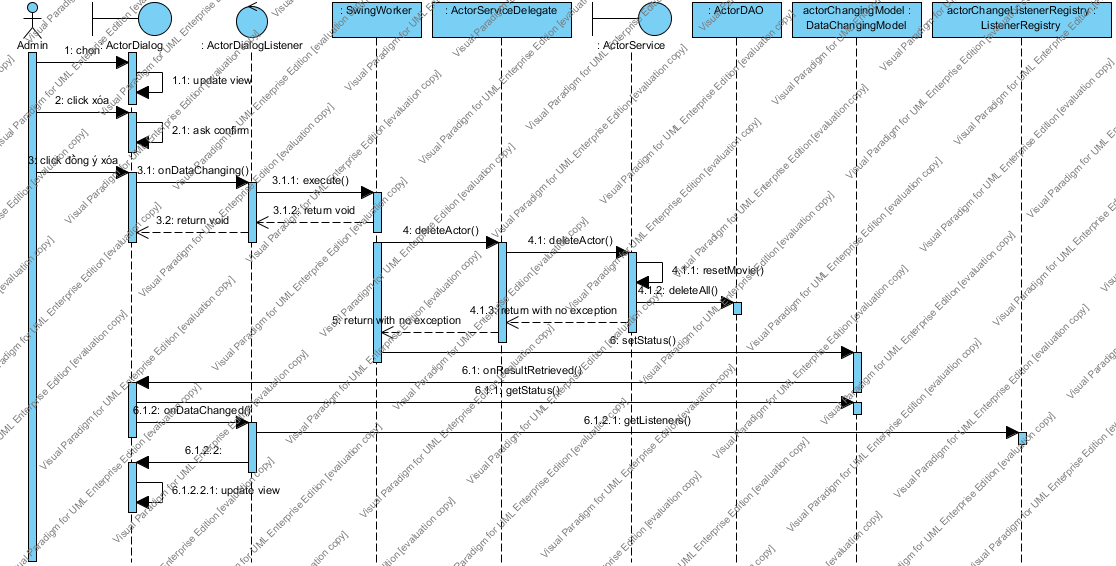
Sequence Diagram

# XoaDaoDien



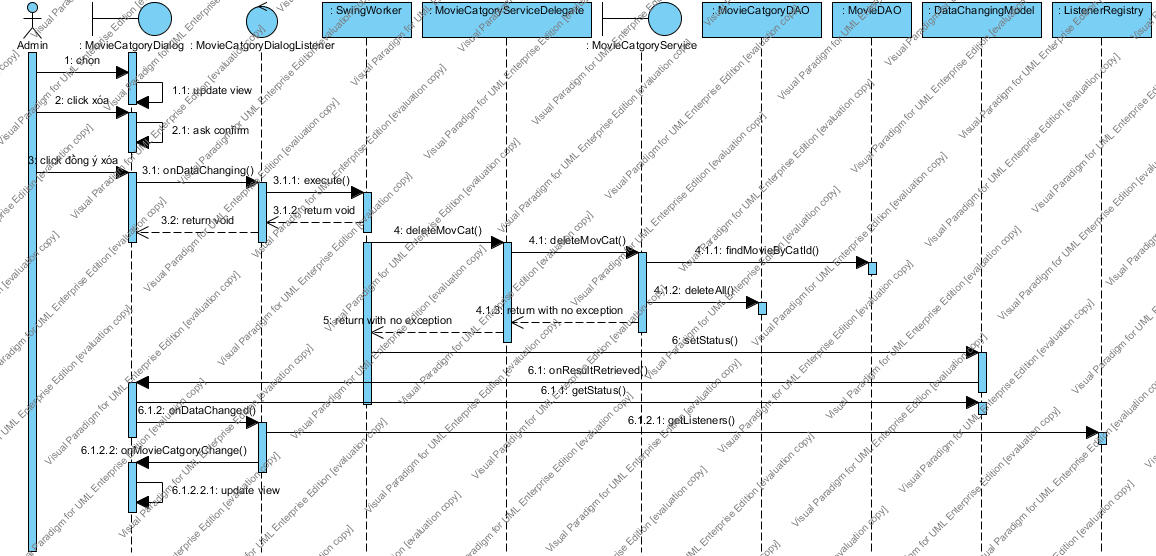
Sequence Diagram

# XoaDienVien



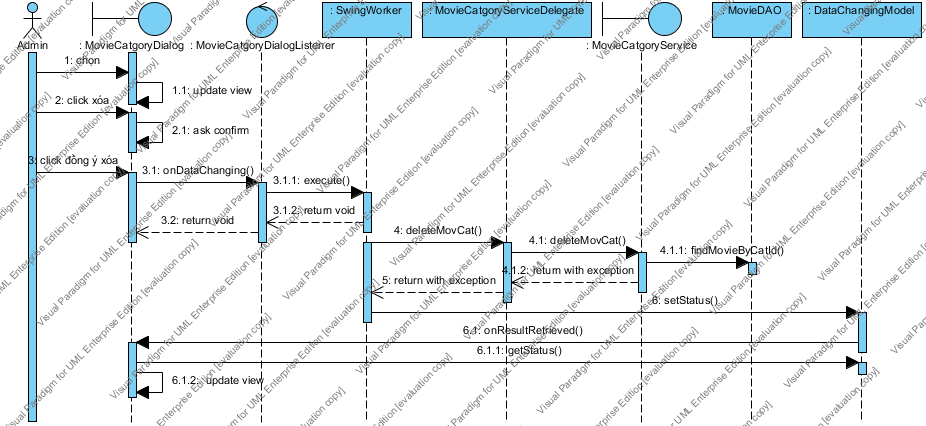
Sequence Diagram

# XoaLoaiPhim\_NormalFlow



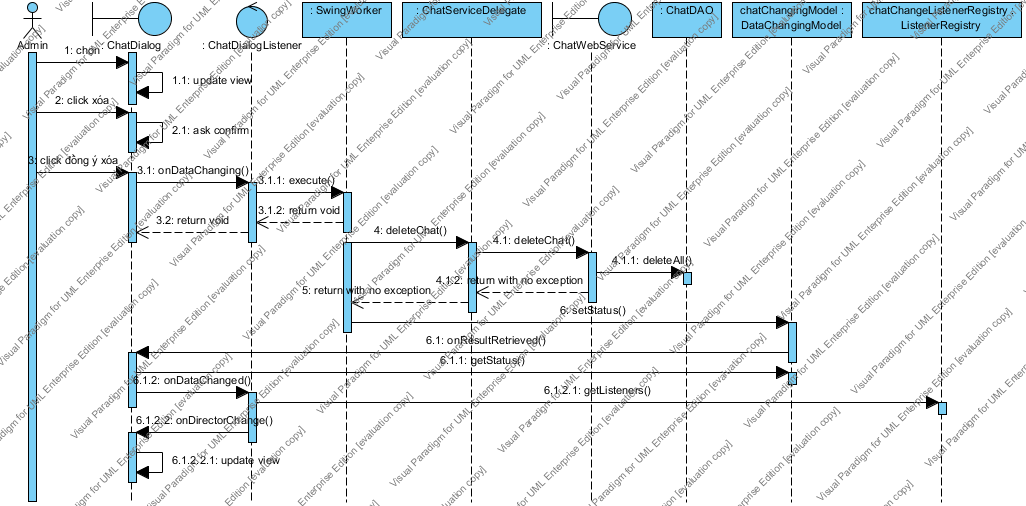
Sequence Diagram

# XoaLoaiPhim\_Fail



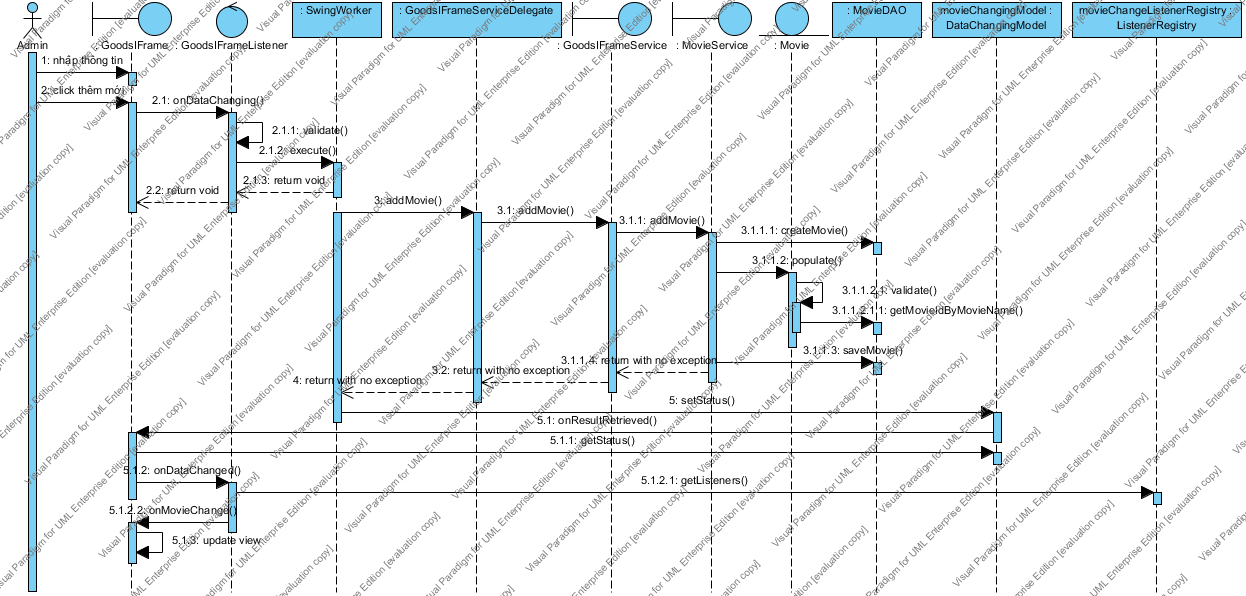
Sequence Diagram

# XoaNickChat



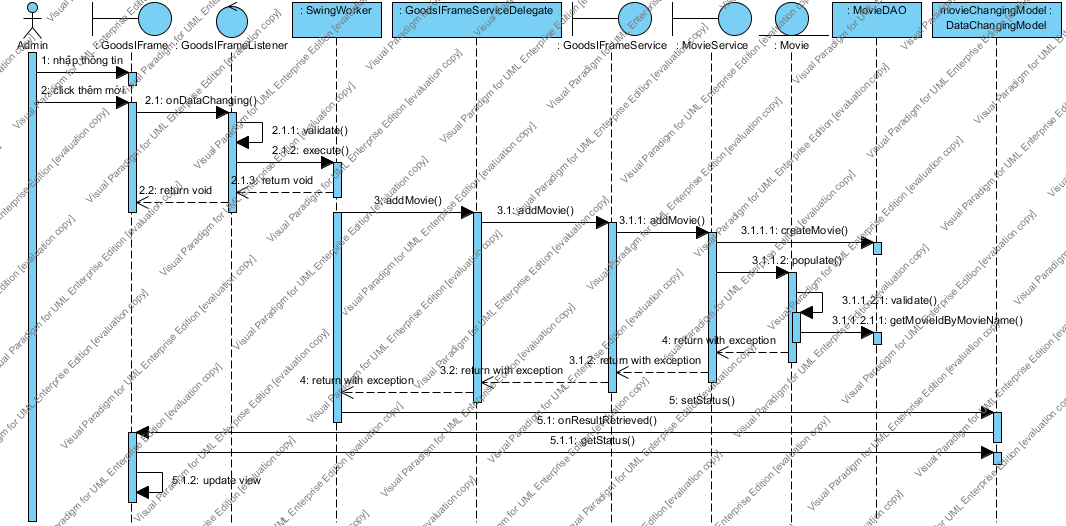
Sequence Diagram

# ThemPhim\_NormalFlow



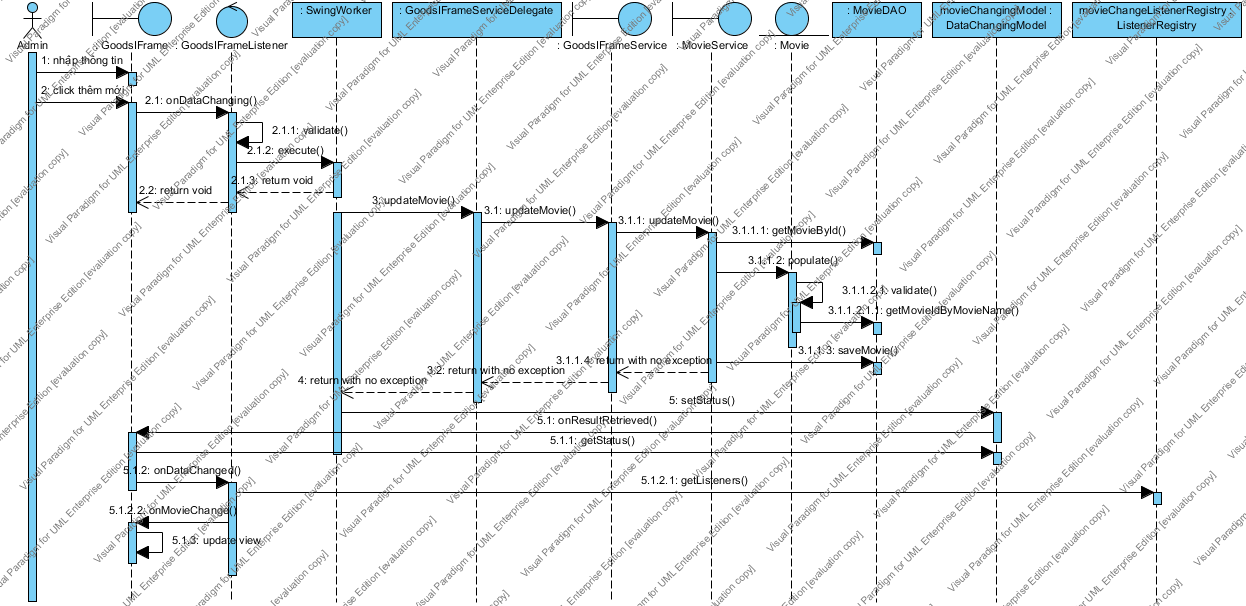
Sequence Diagram

# ThemPhim\_Fail



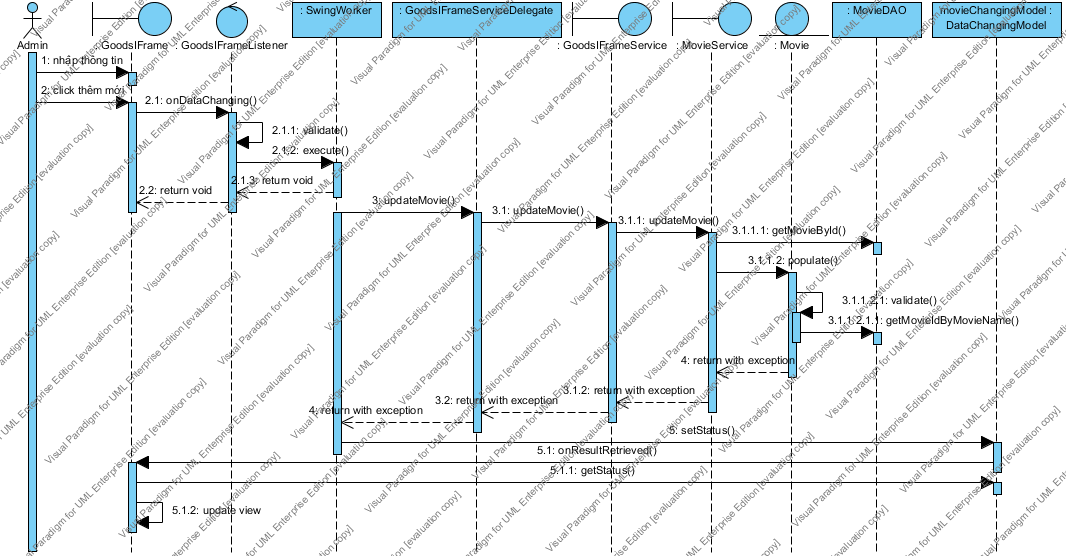
Sequence Diagram

# SuaPhim\_NormalFlow



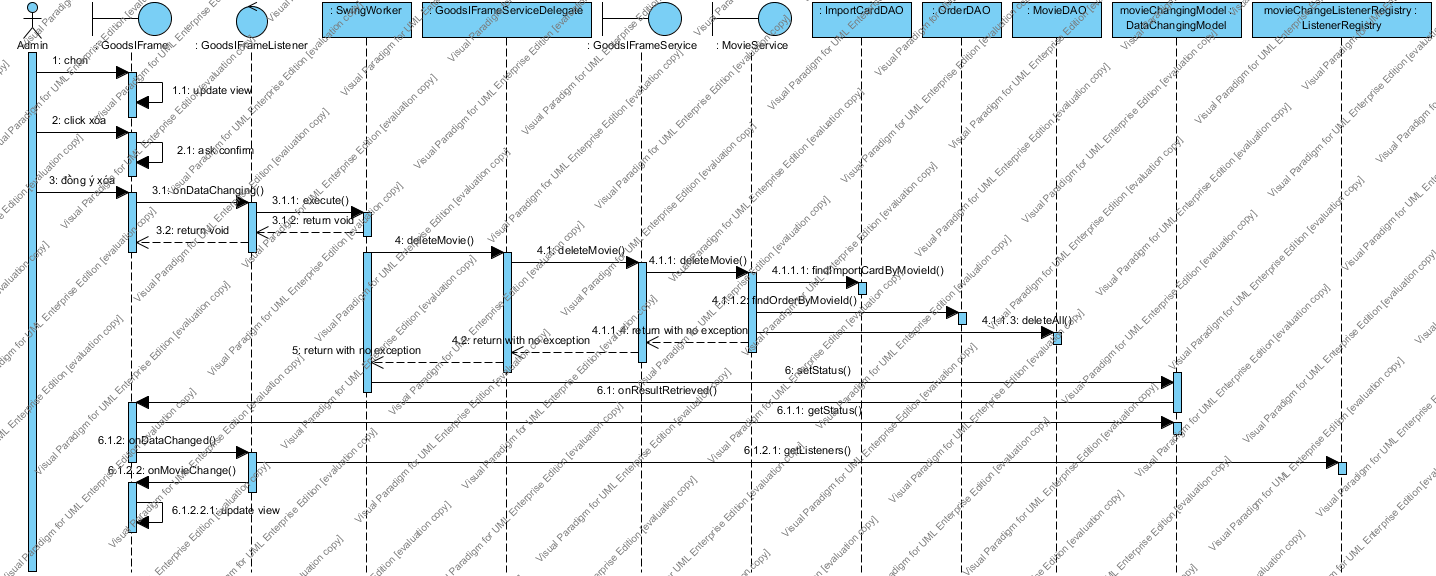
Sequence Diagram

# SuaPhim\_Fail



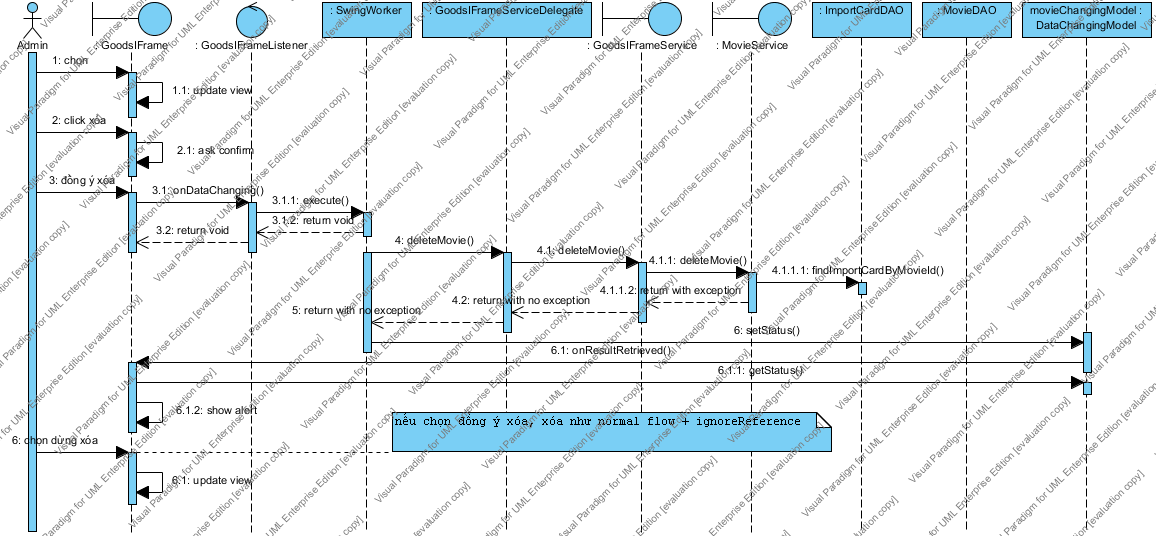
Sequence Diagram

# XoaPhim



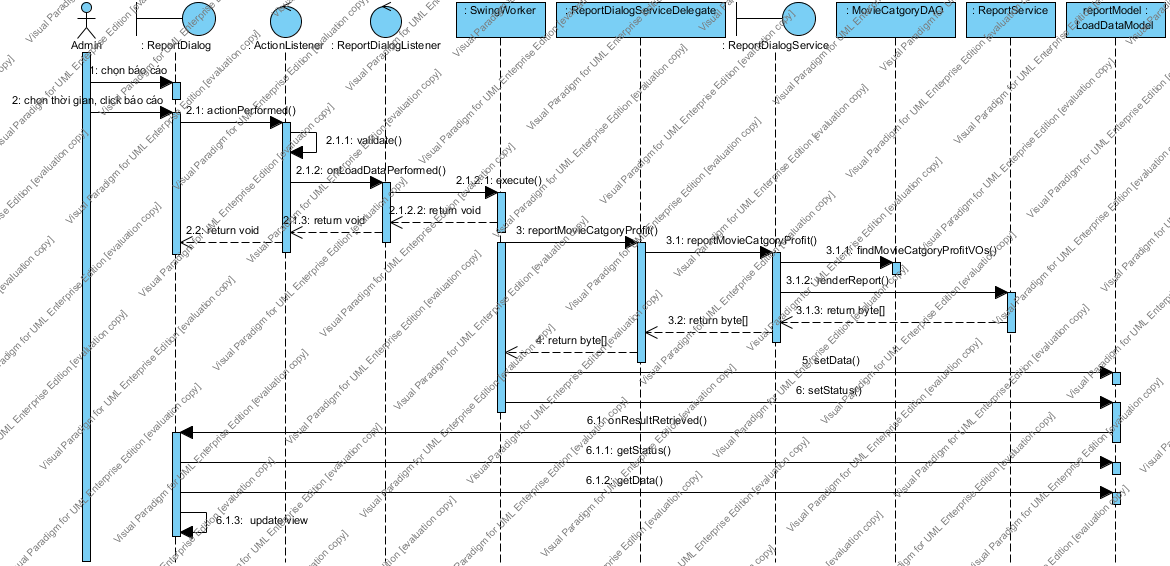
Sequence Diagram

# XoaPhim\_Fail



Sequence Diagram

# XuatDuLieuBaoCaoDoanhThuTheoLoaiPhim

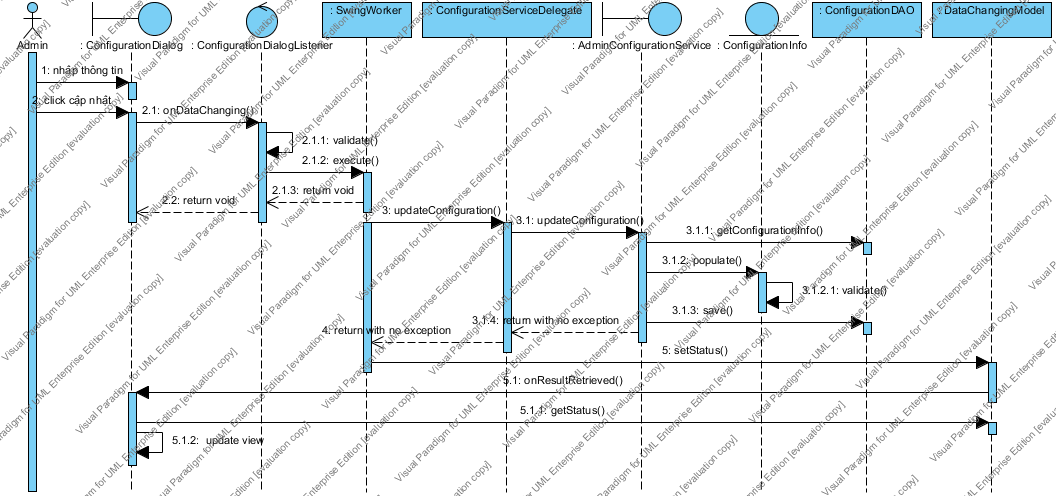


- Về cấu trúc, sequence trên không khác so với các sequence trước (vẫn có các MVC rõ ràng, sử dụng SwingWorker cho multithreading, sử dụng Delegate Interface và Service Interface), tuy nhiên, ở đây chúng ta có thể thấy rõ hơn ưu điểm của thiết kế này, đó là sự tách biệt giữa Presentation tier và Bussiness logic tier. Presentation tier hoàn toàn không biết việc lấy dữ liệu ở đâu và thiết kế bao cáo ra sao: Việc thiết kế báo cáo, lấy dữ liệu, fill data lên báo cáo hoàn toàn được thực hiện phía Bussiness logic tier, việc tập trung này giúp cho báo cáo được lưu tập trung 1 nơi, nếu có thay đổi báo cáo chỉ sửa 1 nơi.

- Do sử dụng web service nên dữ liệu trả về dạng byte[], như thế dễ dàng marsalling dạng xml.

- Có thể thấy có thêm ReportService 🡪 đây là interface cho phép encapsulate các thao tác với Jasper Report, ReportDialogService chịu trách nhiệm lấy dữ liệu cho báo cáo theo yêu cầu bài toán, nhưng không biết gì về render báo cáo, việc này được tách ra 1 lớp riêng, thiết kế này giúp cho việc quản lý các thao tác với Jasper Report api dễ dàng hơn, tập trung 1 nơi, việc sử dụng strategy design pattern (sử dụng interface và spring xml config) giúp dễ dàng thay đổi nếu có thay đổi trong report api

# SuaCaiDatHeThong



- Configuration được lưu theo cơ chế hoàn toàn khác, và có nguy cơ gắn với servlet api. Tuy nhiên nhìn vào thiết kế này chúng ta có thể thấy developer sẽ quen thuộc vì hoàn toàn giống thiết kế đối với các modul trước.

# SuaCaiDatHeThong\_ValidateFail

