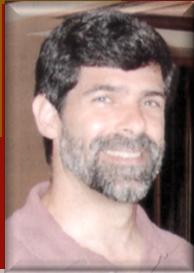


# **CSSE 374:** **Logical Architecture and** **Refinement Package** **Design**



**Shawn Bohner**  
**Office: Moench Room F212**  
**Phone: (812) 877-8685**  
**Email: bohner@rose-hulman.edu**



---

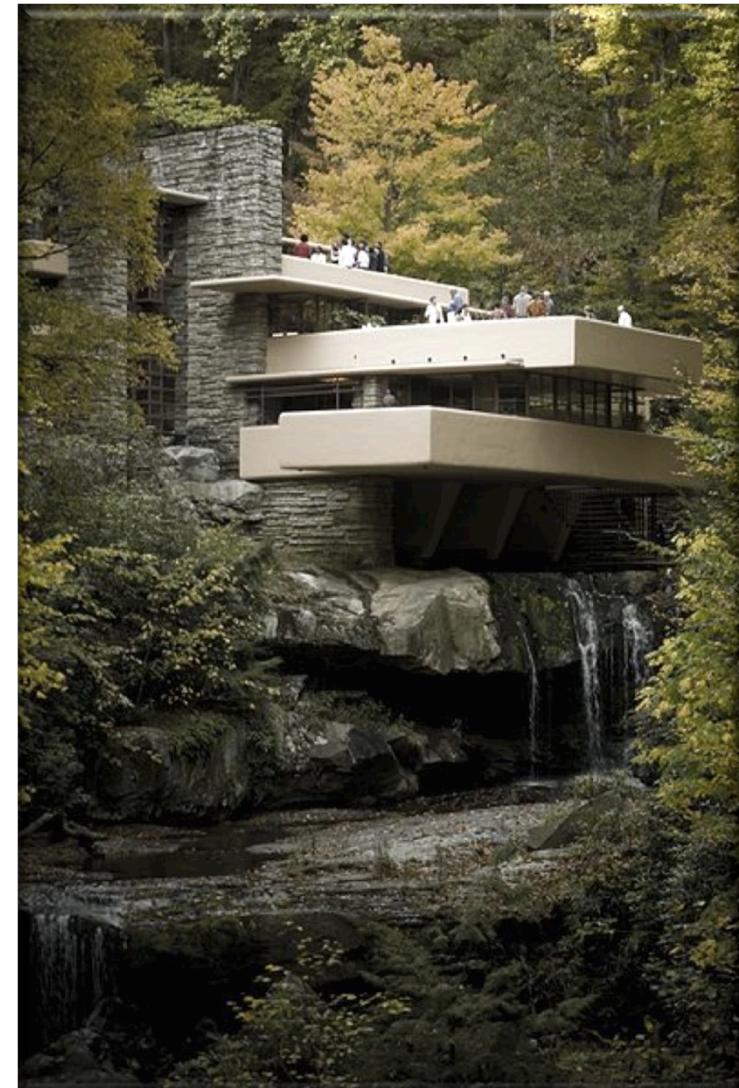
**Q1**

**ROSE-HULMAN**  
INSTITUTE OF TECHNOLOGY

# Learning Outcomes: Analysis of Design

Analyze and explain the feasibility & soundness of a software design.

- Logical Architecture Refinements
- Package Design
- Design Studio - Team 2.2



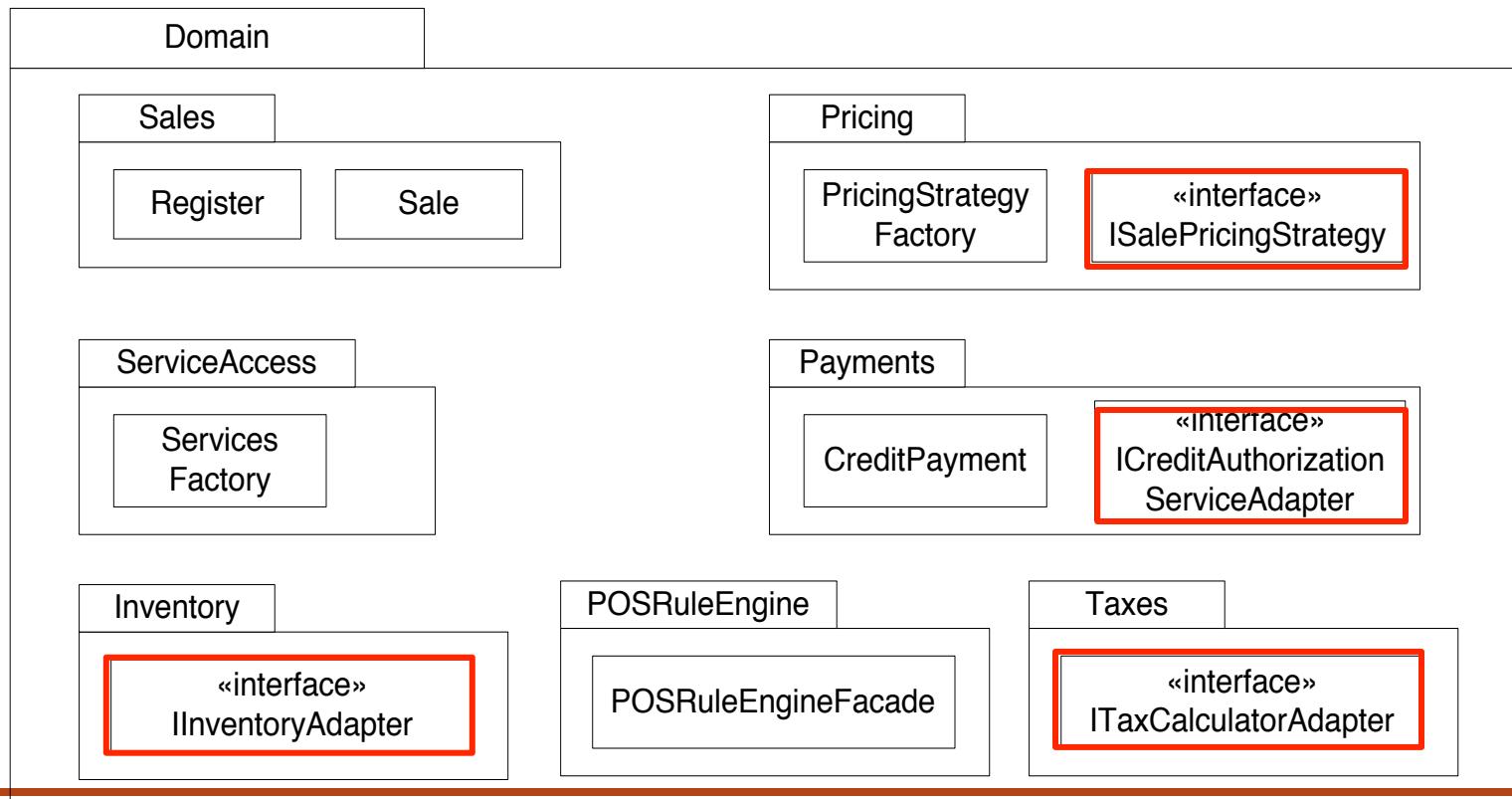
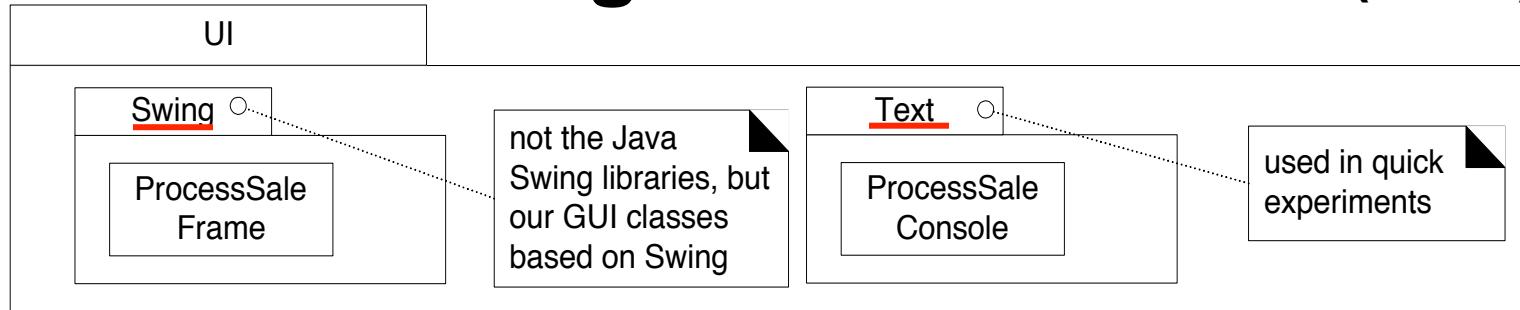


# **The Logical Architecture is a static depiction. When is it useful to show dynamic information to support this level of Design?**

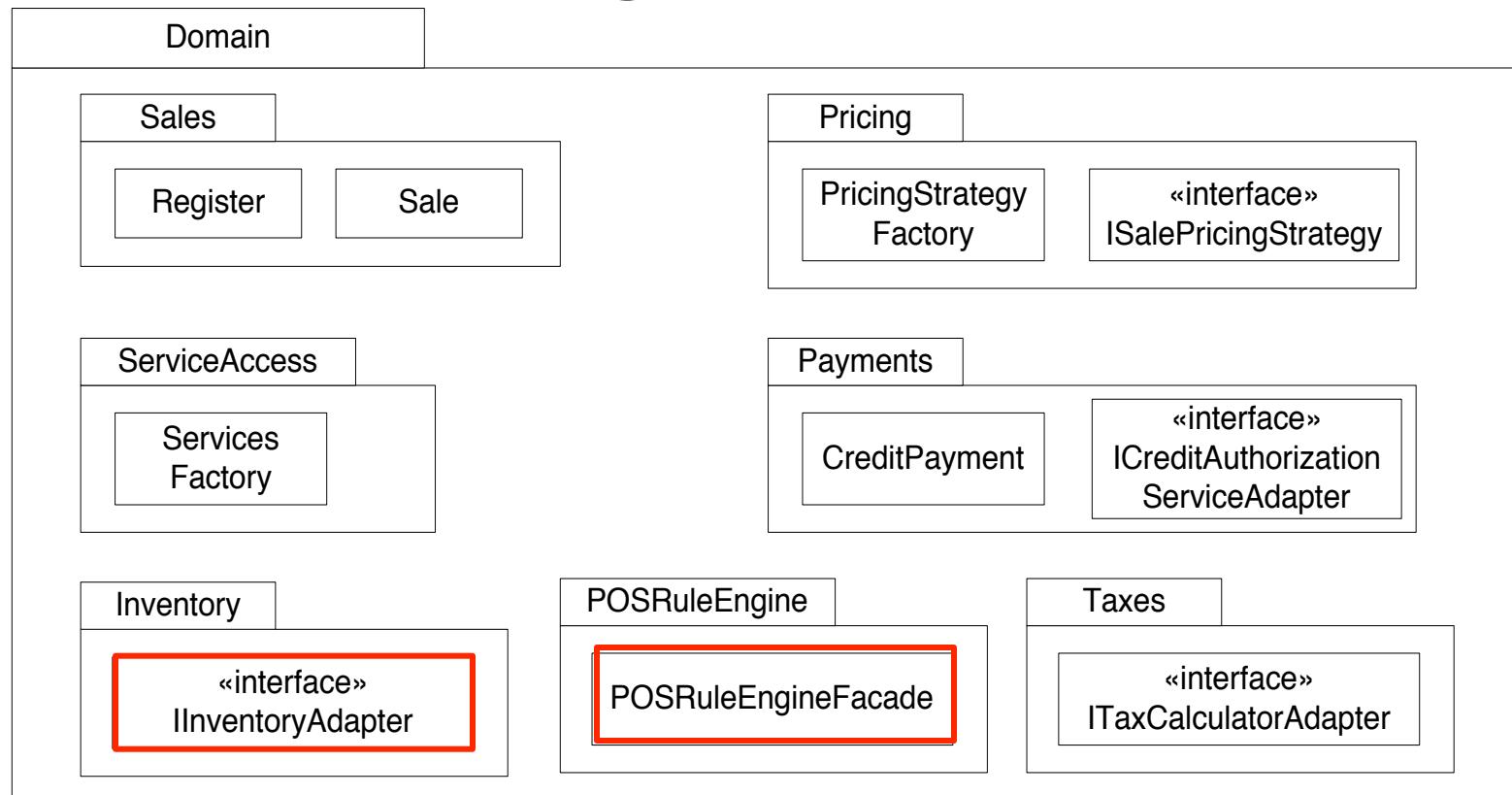
- Think for 15 seconds...
- Turn to a neighbor and discuss it for a minute



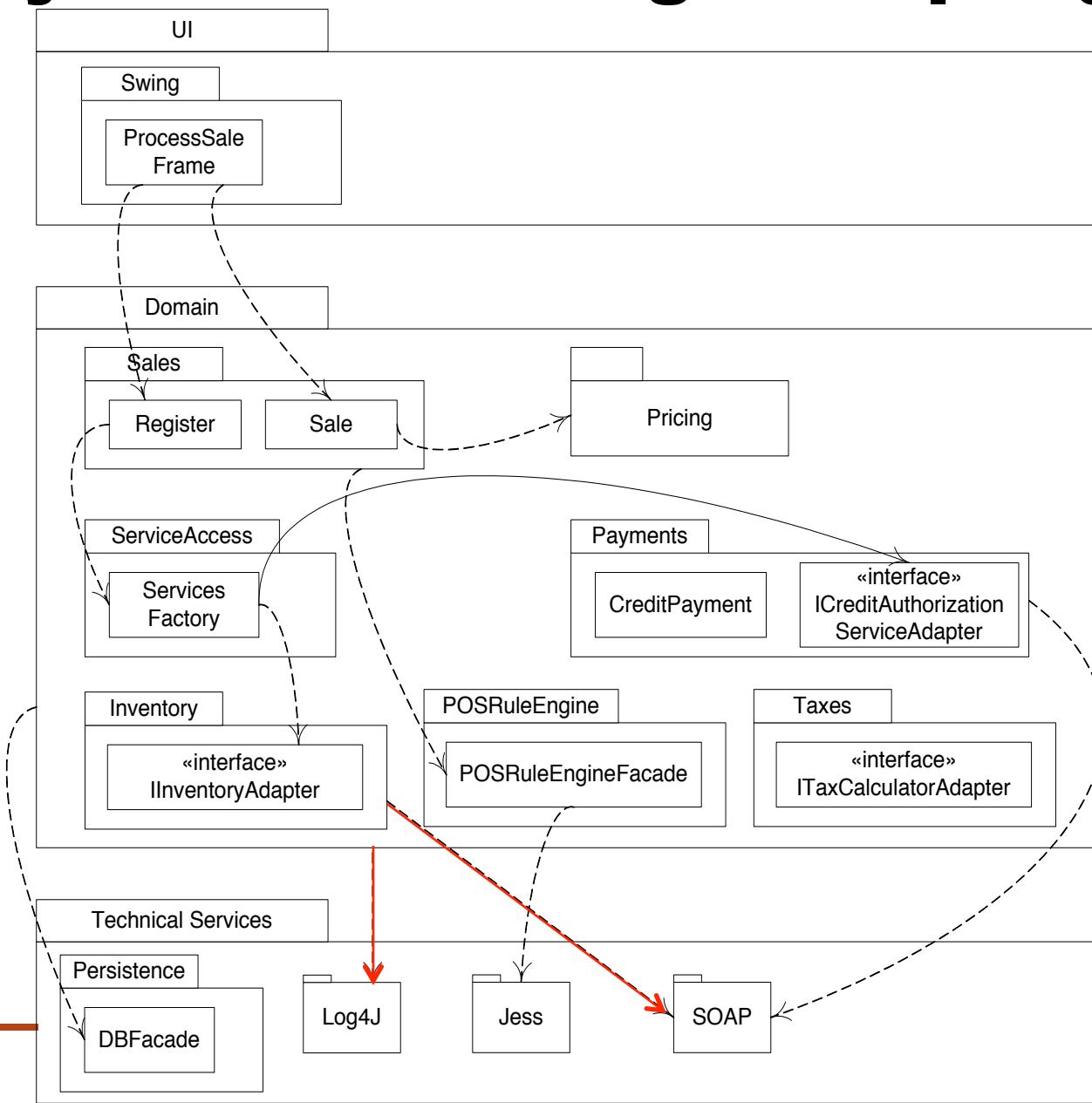
# NextGen POS Logical Architecture (1 of 2)



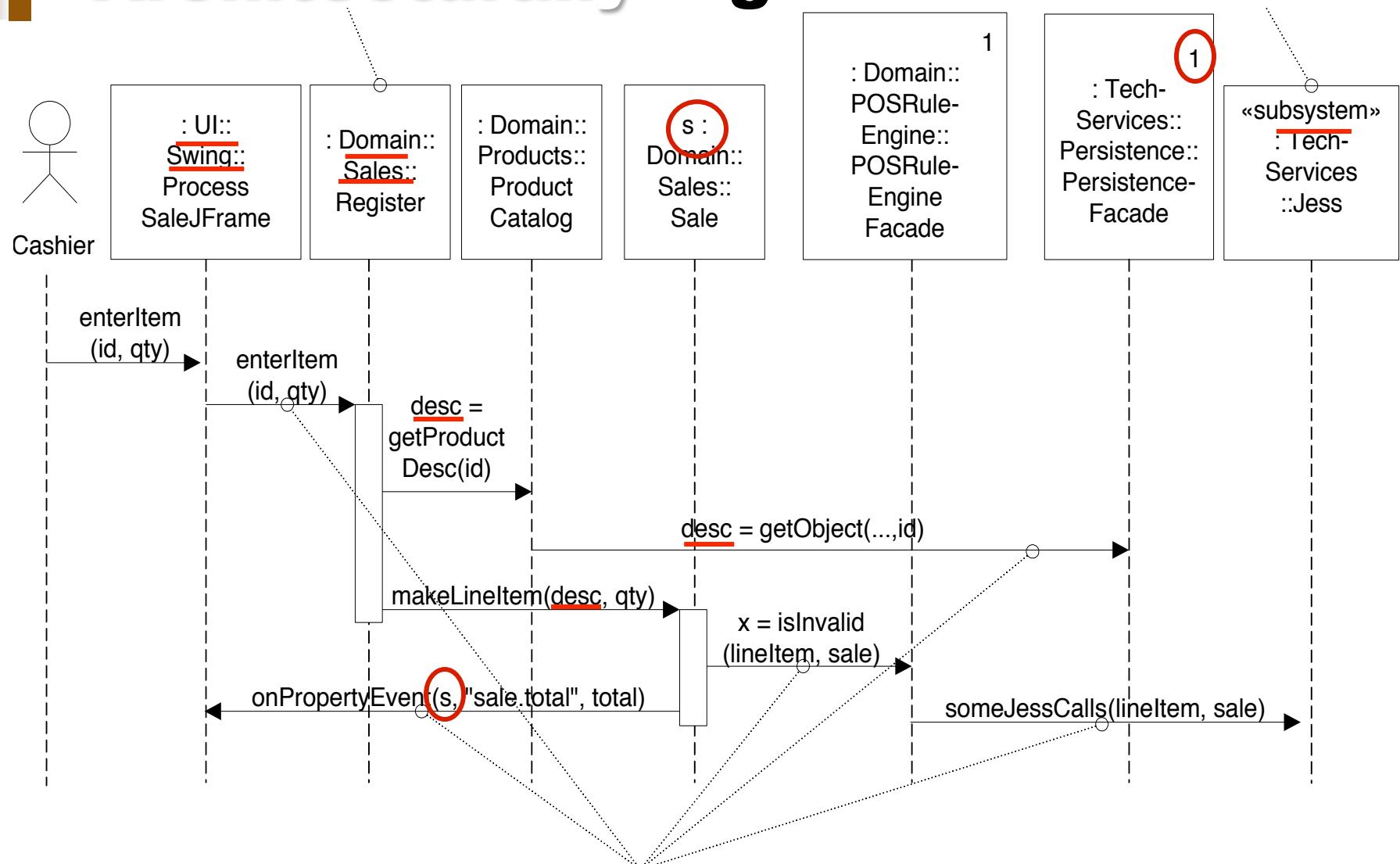
# NextGen POS Logical Architecture (2 of 2)



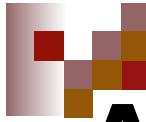
# Inter-Layer/Intra-Package Coupling



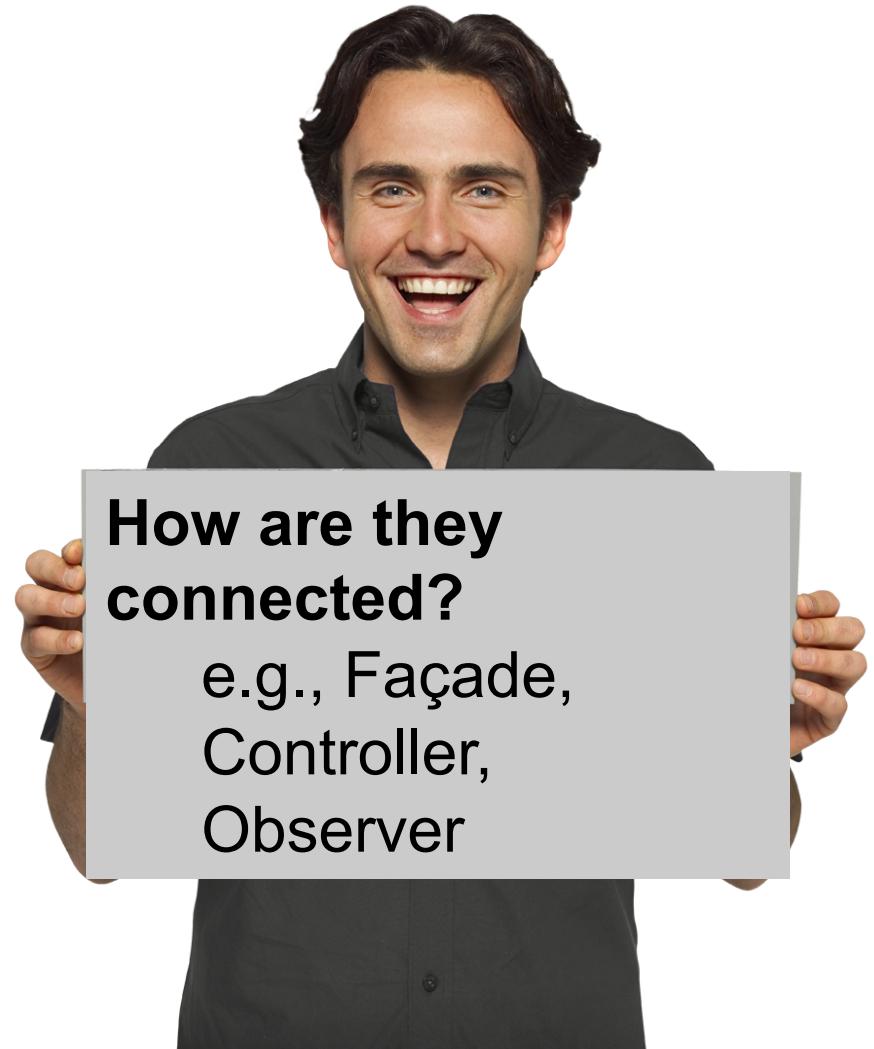
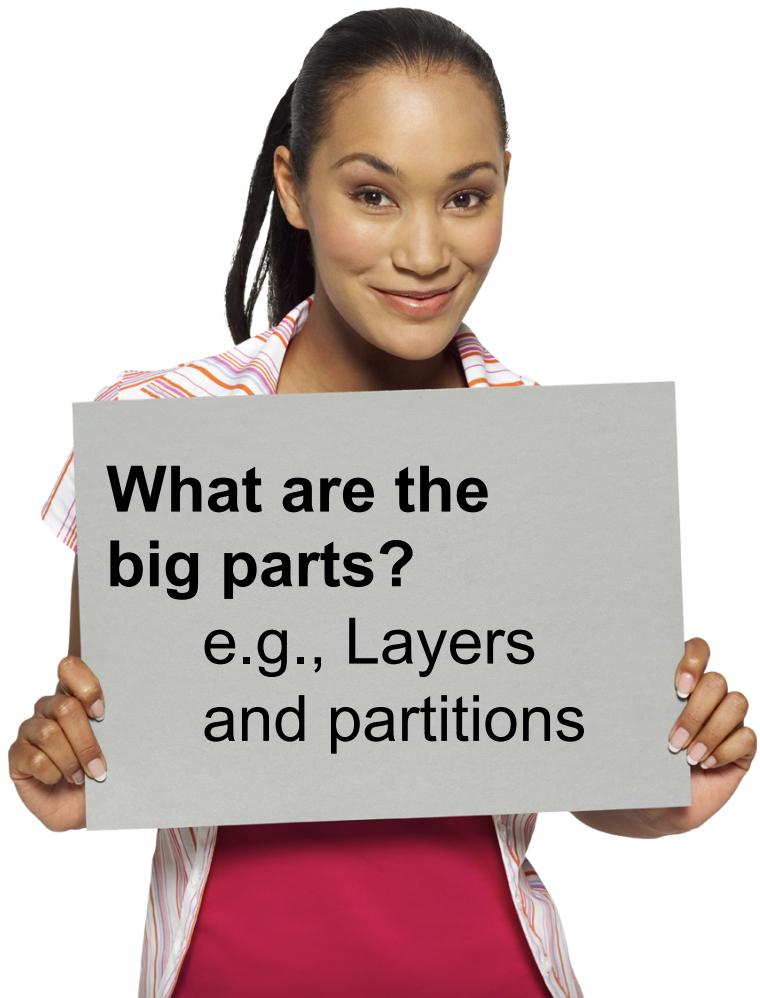
# Architecturally Significant Scenarios

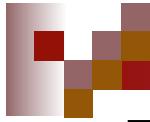


Points of crossing interesting boundaries or layers. These are especially noteworthy for people who need to understand the system, and thus are highlighted in this diagram. This diagram supports communicating the *logical view of the architecture* (a UP term) because it emphasizes architecturally significant information.

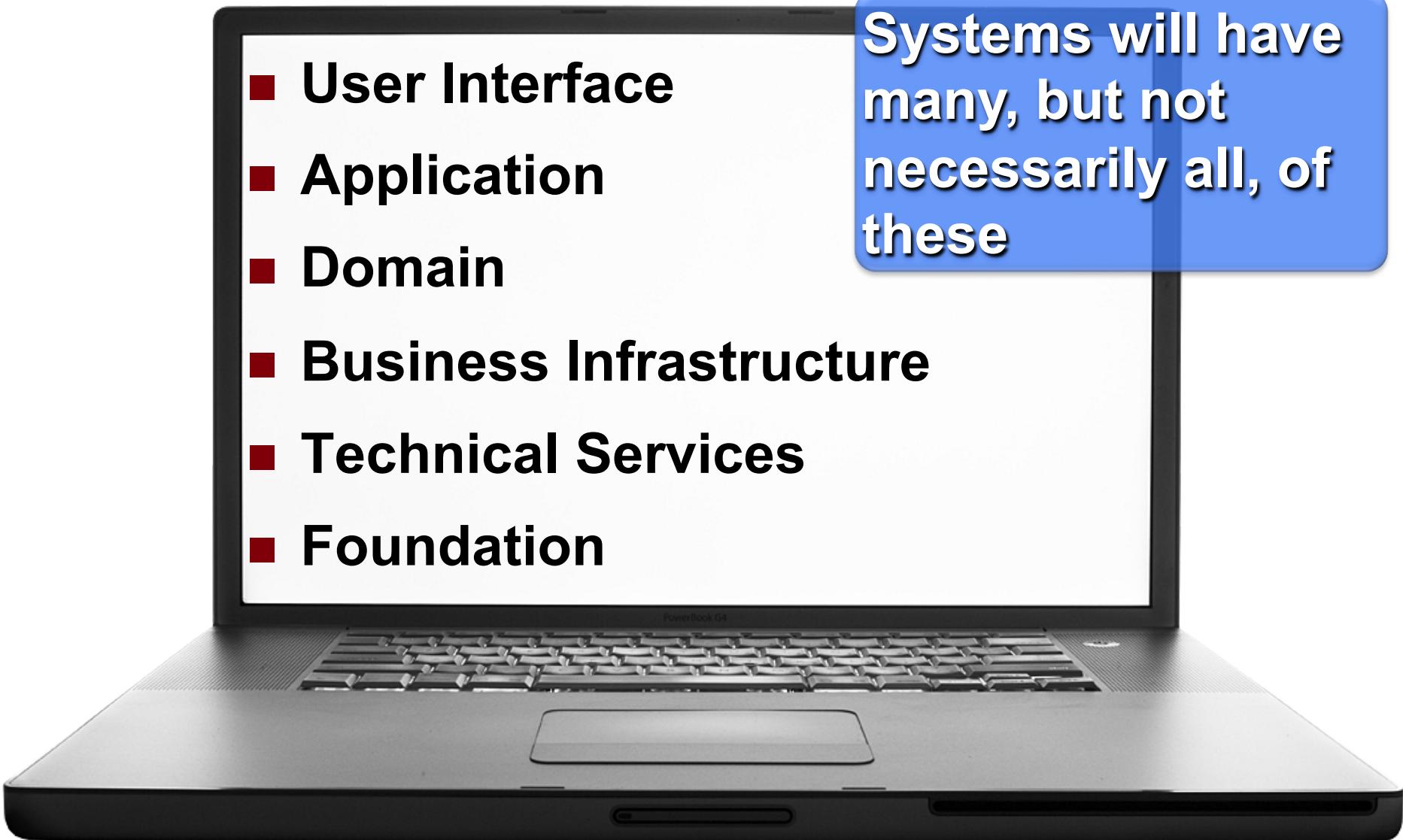


# Architectural Level Design Decisions





# Recall: Common Layers



# Simple Packages vs. Subsystems

- *Simple package*: just groups classes
  - Pricing
  - Sales
- *Subsystem*: discrete, reusable “engine”
  - Persistence
  - POSRuleEngine

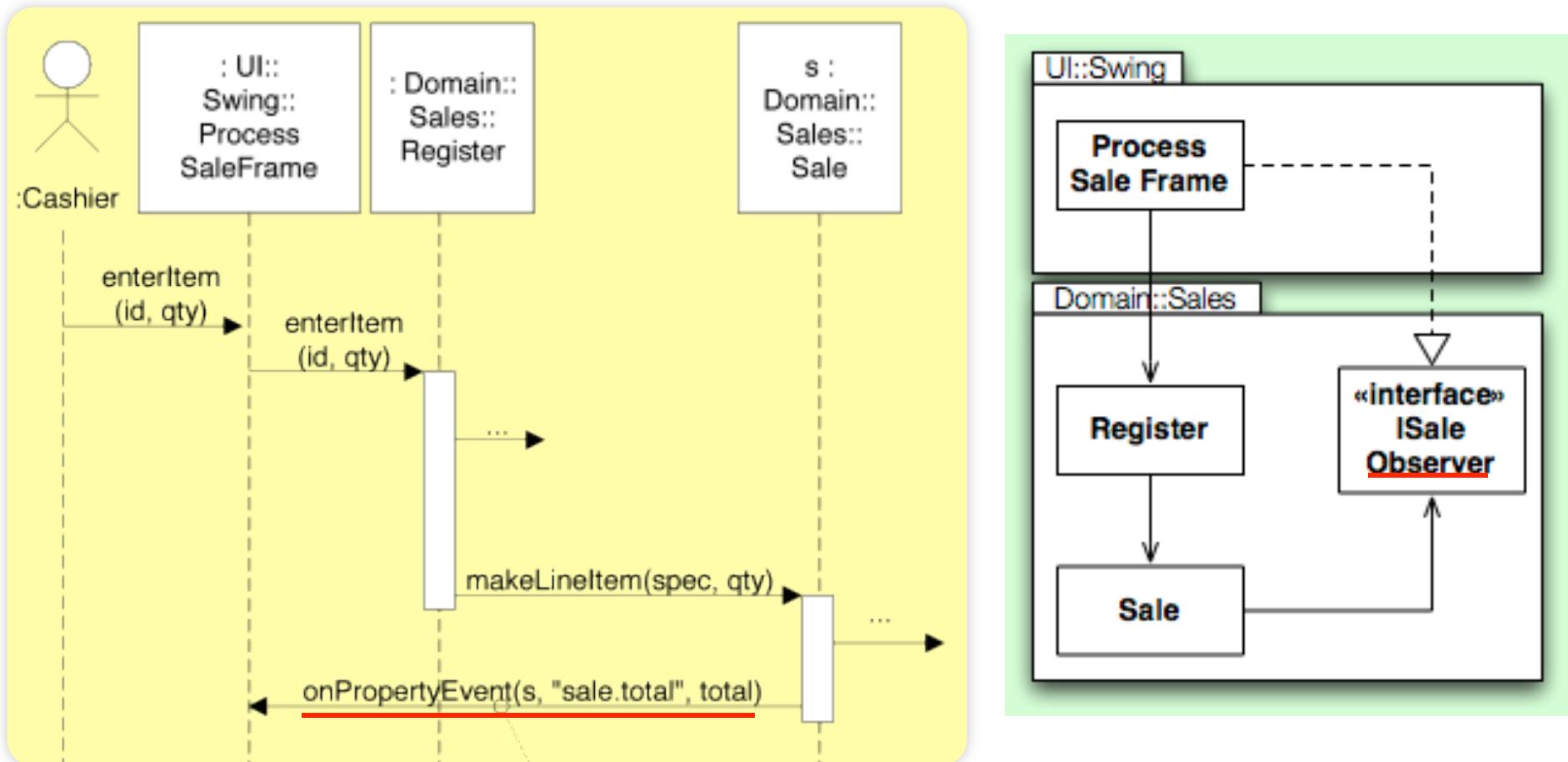


# **Subsystems Often Provide a Façade**

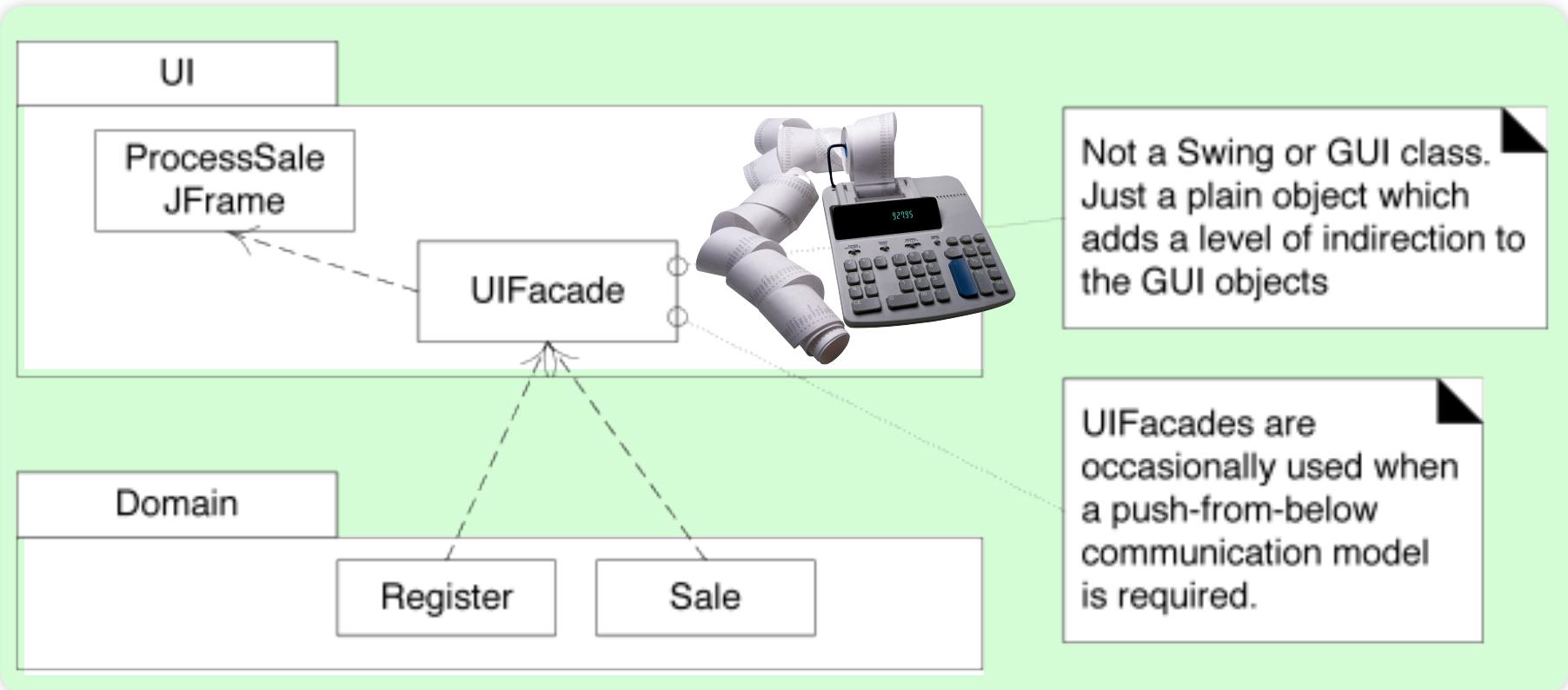
- Serves as a single variation point
- Defines the subsystems services
- Exposes just a few high-level operations
  - High cohesion
  - Allows different deployment architectures



# Upward Collaboration with *Observer*



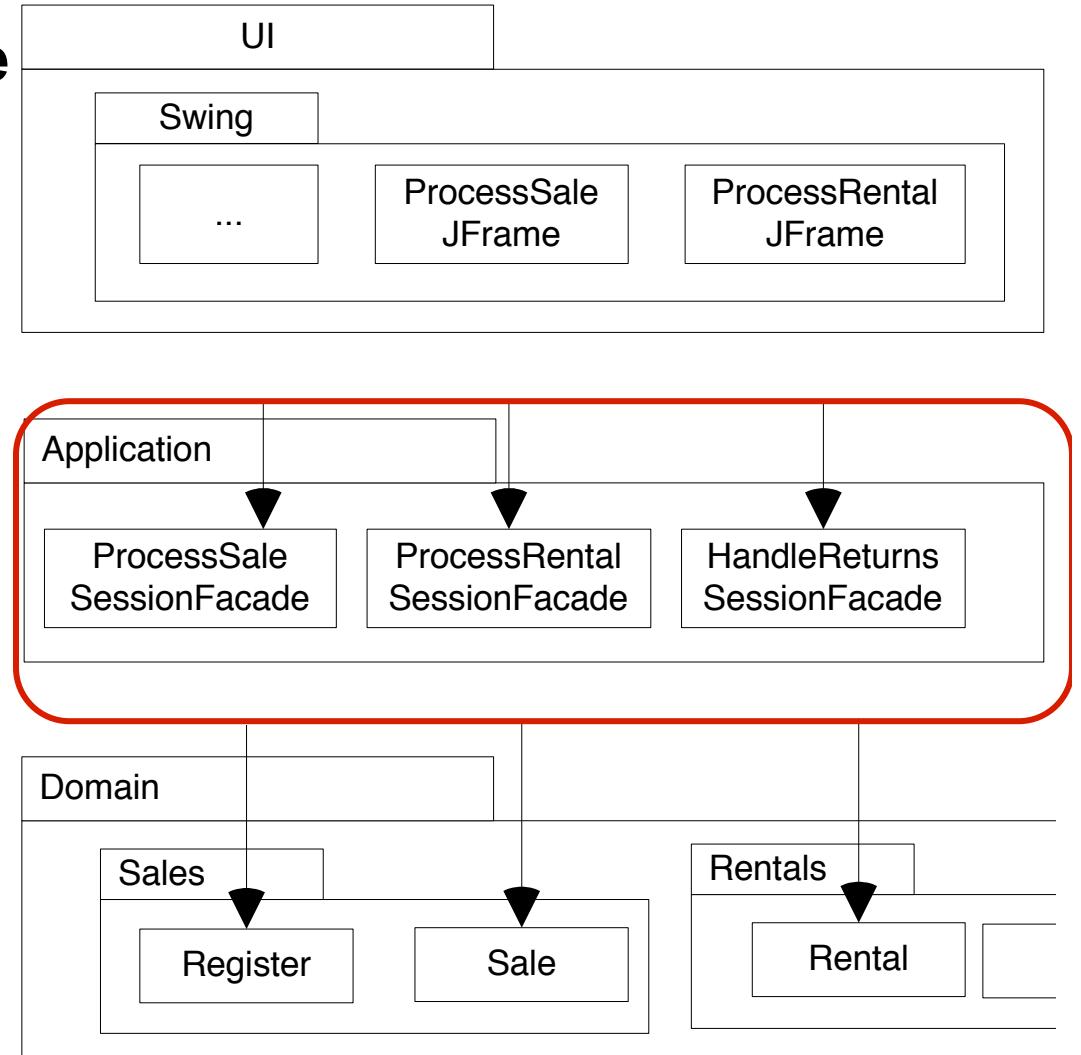
# Alt: Upward Collaboration with UI Façade



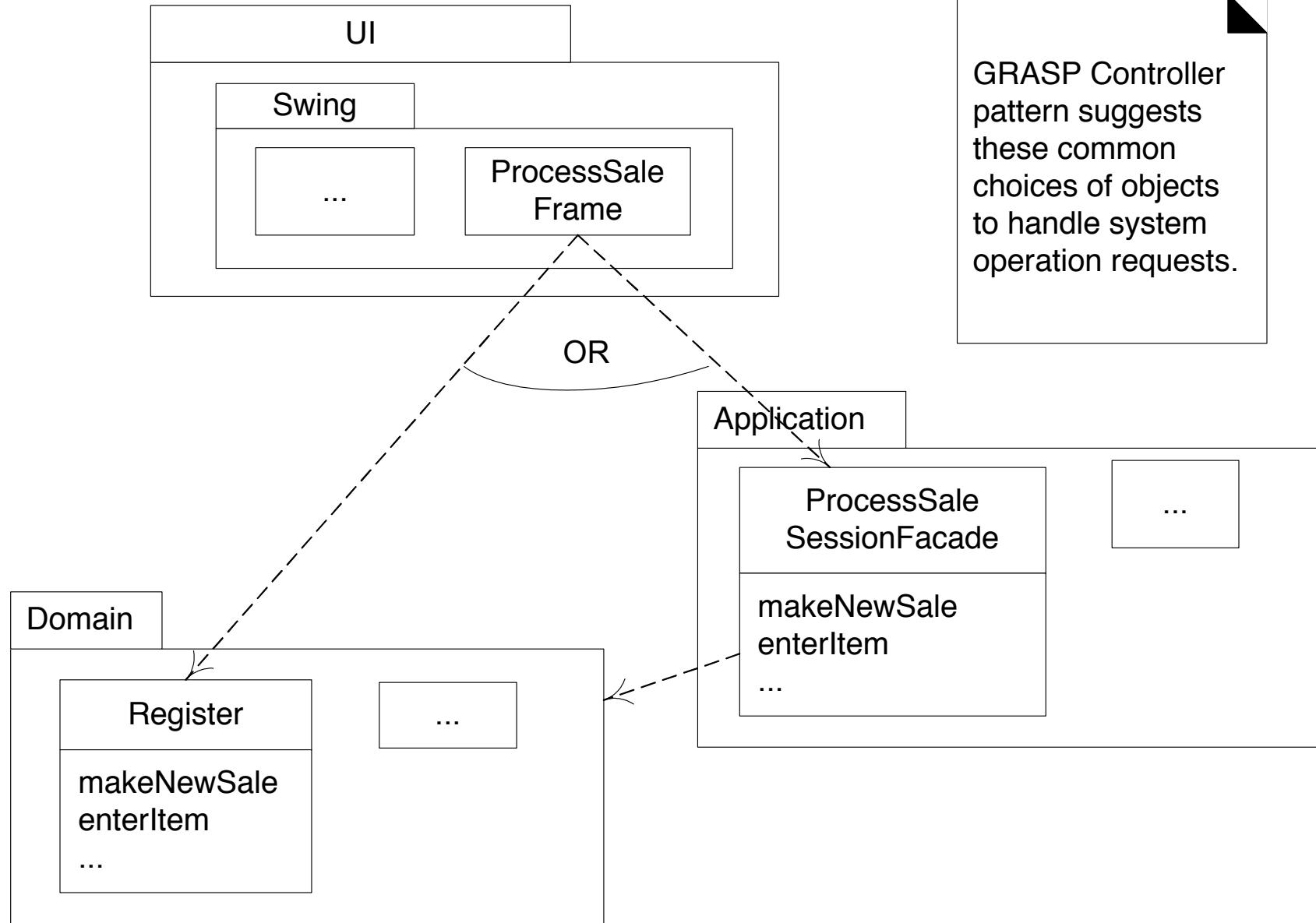
When might this be useful?

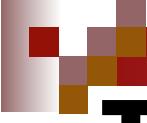
# Application Layer

- Maintains session state
- Houses Controllers
- Enforces order of operations
- Useful when:
  - Multiple UIs
  - Distributed systems with UI and Domain separated
  - Insulating Domain from session state
  - Strict workflow



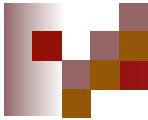
# Controller Choices





# Typical Coupling Between Layers

- From higher layers to Technical Services and Foundation
- From Domain to Business Infrastructure
- From UI to Application & Application to Domain
- Desktop apps: UI uses Domain objects directly
  - E.g., Sales, Payment
- Distributed apps: UI gets data representation objects
  - E.g., SalesData, PaymentData



# Liabilities with Layers

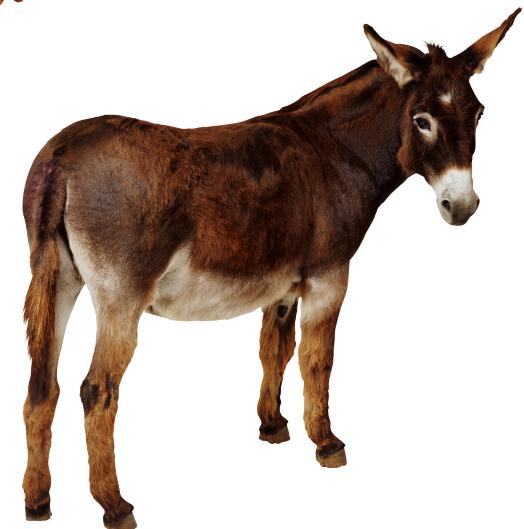
## ■ Performance

- e.g., game applications that directly communicate with graphics cards or real-time system interrupts



## ■ Poor architectural fit sometimes

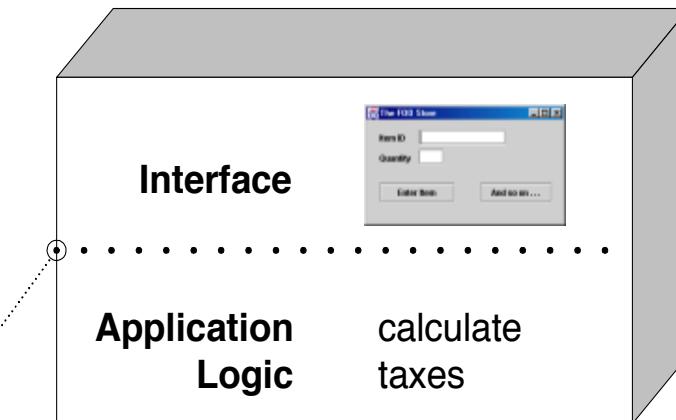
- Batch processing (use “Pipes and Filters”)
- Expert systems (use “Blackboard”)



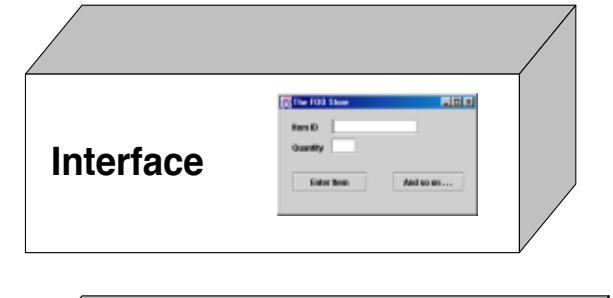


# 3-Tiered Architecture Depictions in UML

UML notation:  
a node. This is  
a processing  
resource such  
as a computer.



classic 3-tier architecture deployed  
on 2 nodes: "thicker client"



classic 3-tier architecture  
deployed on 3 nodes: "thinner client"



# Physical Package Design

- Goal: define physical packages so they can be:

- Developed independently
- Deployed independently

Multiple logical packages might be developed together physically

- Packages should depend on other packages that are more stable than themselves
  - Avoids *version thrashing*

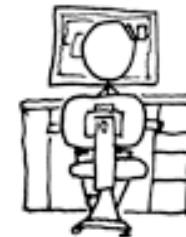


# Straying outside the guidelines...

I COULD RESTRUCTURE  
THE PROGRAM'S FLOW  
OR USE ONE LITTLE  
'GOTO' INSTEAD.



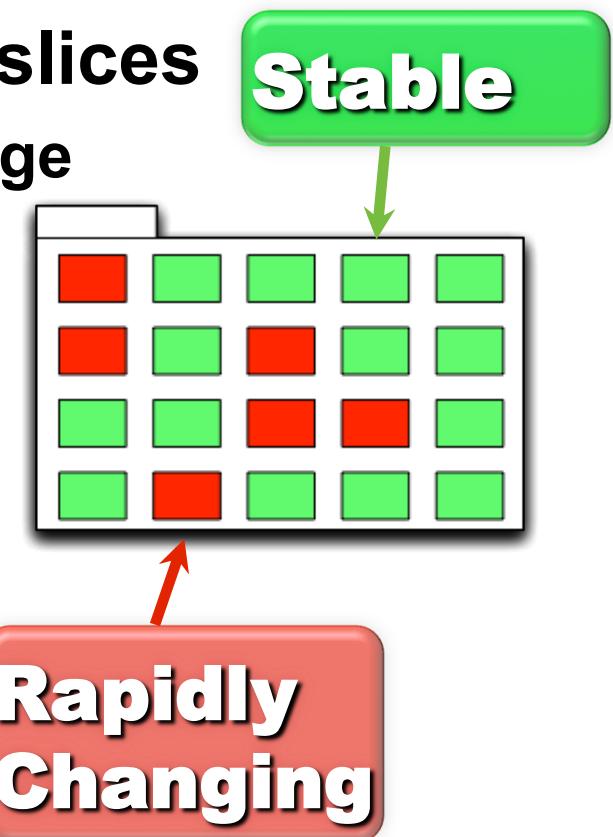
EH, SCREW GOOD PRACTICE.  
HOW BAD CAN IT BE?



# Package Organization Guidelines 1/3

**Guideline:** Most Responsible are most stable.

- Package functionally cohesive slices
  - Limit strong coupling within package
  - Loose coupling between packages
- Package a family of interfaces
  - Factor out independent types
- Package by clusters of unstable classes
- Make the most depended-on packages the most stable





# Package Organization Guidelines 2/3

Increase stability by:

1. Using only (or mostly) interfaces and abstract classes
2. Not depending on other packages
3. Encapsulating dependencies  
(e.g., with Façade)
4. Heavy testing before first release
5. Fiat

Iron-fisted rule,  
not the Italian car brand ☺



# Package Organization Guidelines 3/3

## Guideline: Factor out the independent types

- Grouping by common functionality may not provide right level of granularity in packages
- e.g., Common Utilities

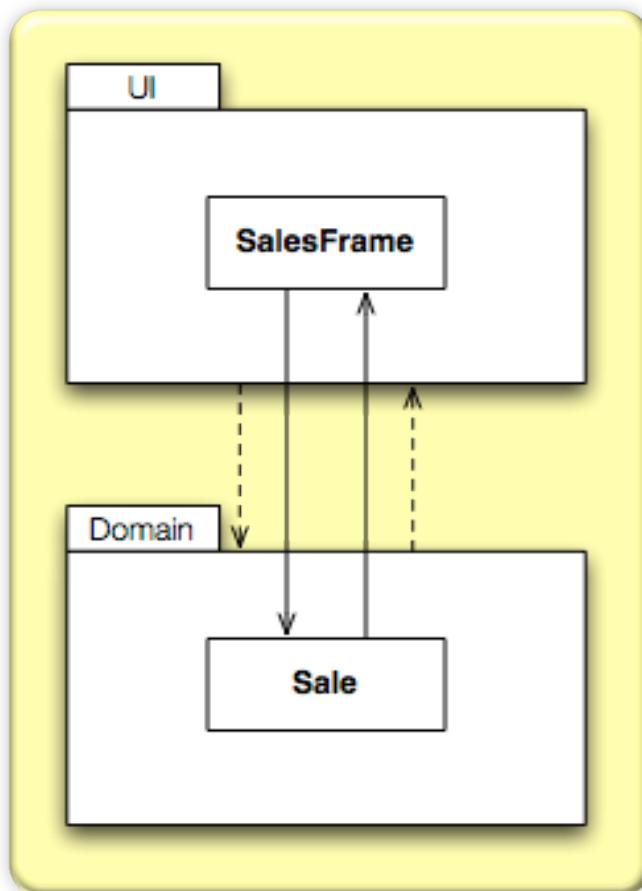
## Guideline: Use factories to reduce dependencies on concrete packages

- E.g., instead of exposing all the subtypes, expose an abstract superclass and a factory

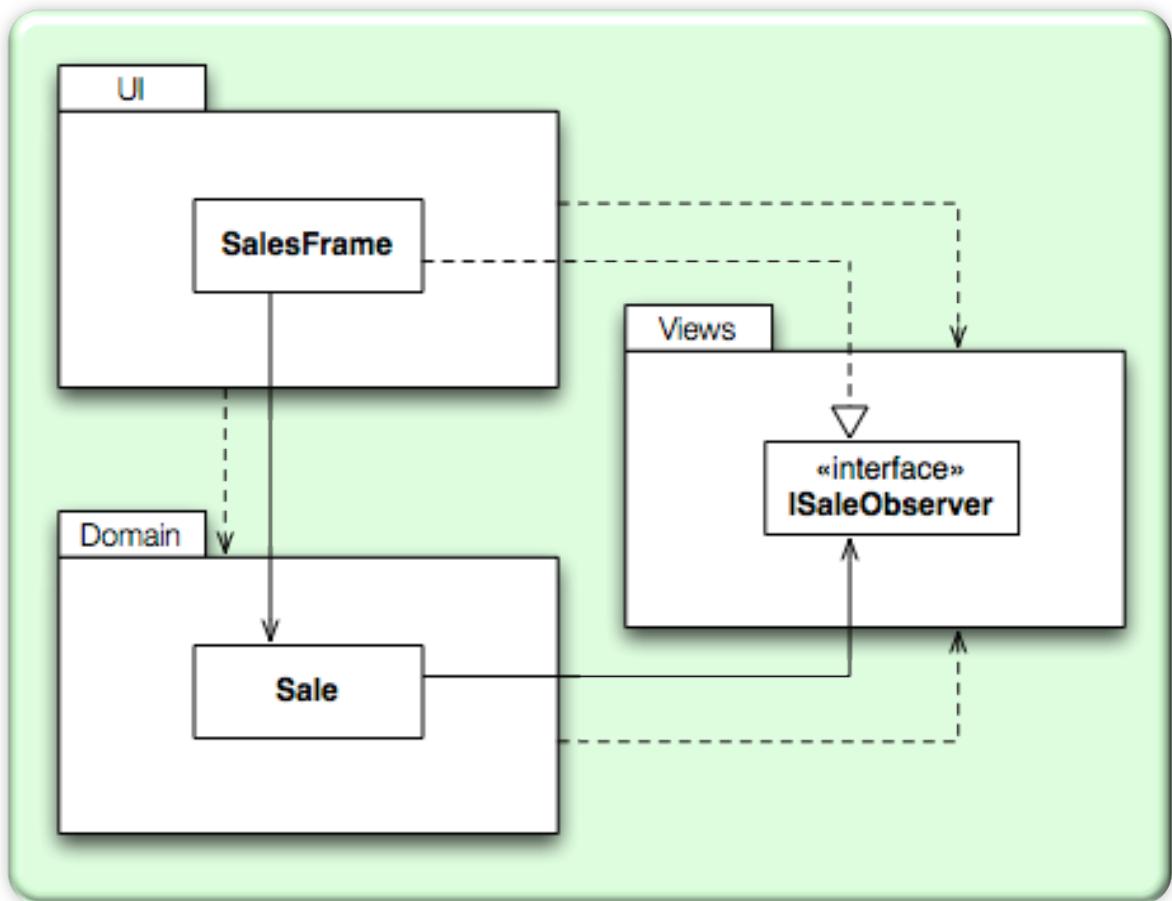
## Guideline: No cycles between packages

- Cycles often force packages to be developed and released together

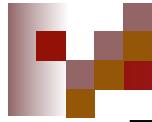
# Breaking Dependency Cycles Between Packages



**Cyclic  
Coupling**

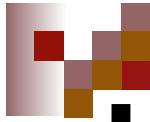


**Better: Cycle Removed!**



# Design Studio Calendar

	Monday	Tuesday	Thursday
8th week		Team 2.4	Team 2.1
9th week	Today Team 2.2	Team 2.3	Team 2.5
10th week	Team 2.4	Team 2.1	Course Wrap-up



# Homework and Milestone Reminders

- **Read Chapter 36**
- **Milestone 5 – Final Junior Project System and Design**
  - Preliminary Design Walkthrough on Friday, February 11th, 2011 during weekly project meeting
  - Final due by 11:59pm on Friday, February 18<sup>th</sup>, 2011
- **Team 2.3 Design Studio**