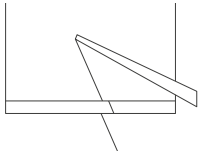
Bring ideas to life
VIA University College



Separation of Concerns

Logical Architecture and Package Diagrams
SWE 1

Software Architecture

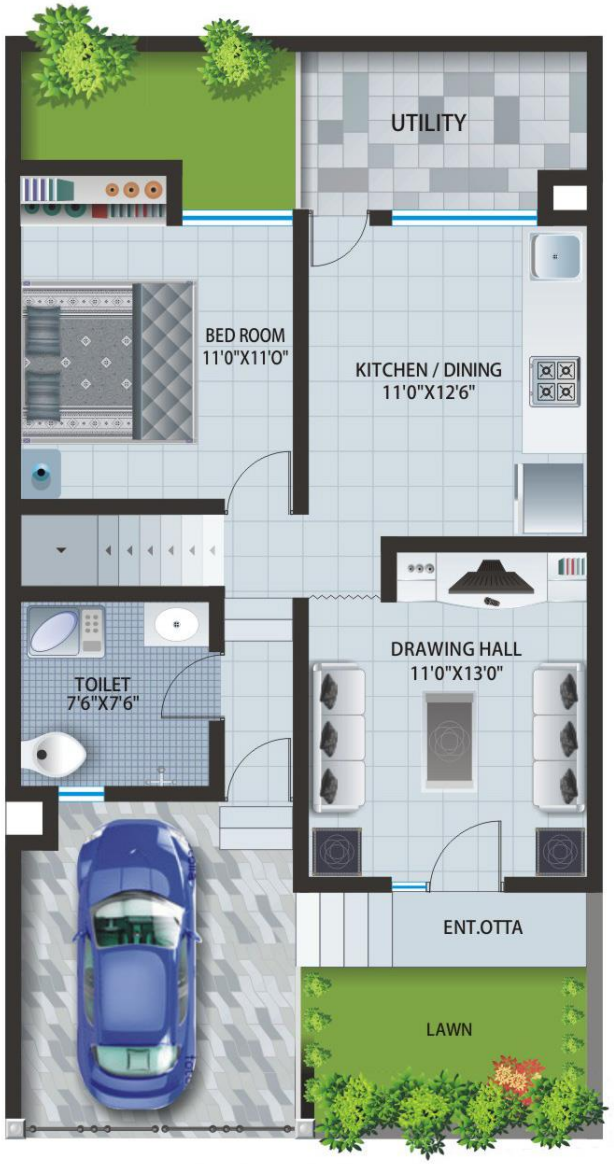
Think of a house – what should be in

- Kitchen
- Bathroom
- Toilet
- Living rooms

And not least – how are these rooms connected together?

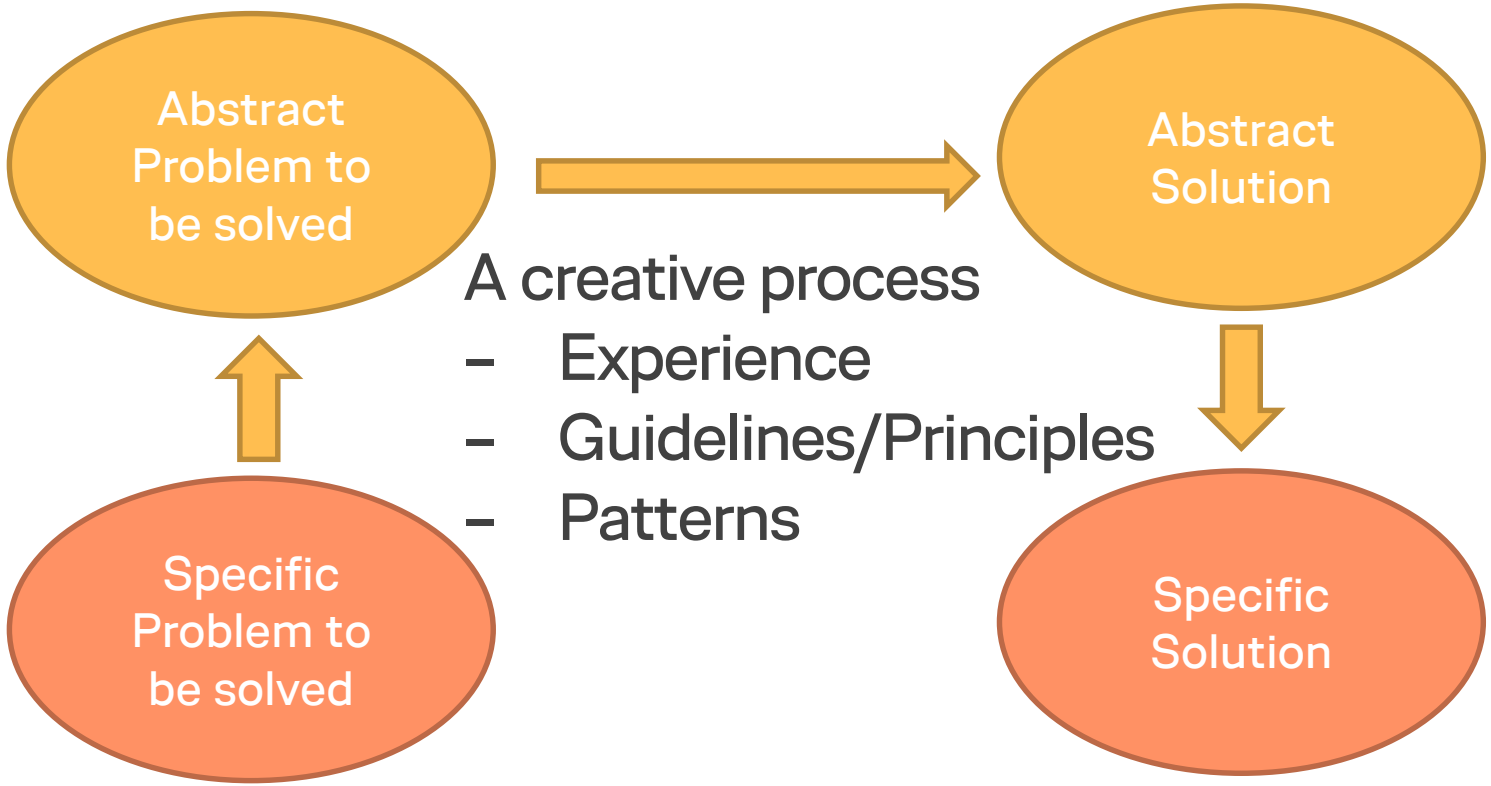
In software design – what classes belongs to

- User Interface
- Business logic
- Persistence



GROUND FLOOR PLAN

OOA/D Work Pattern



4+1 View

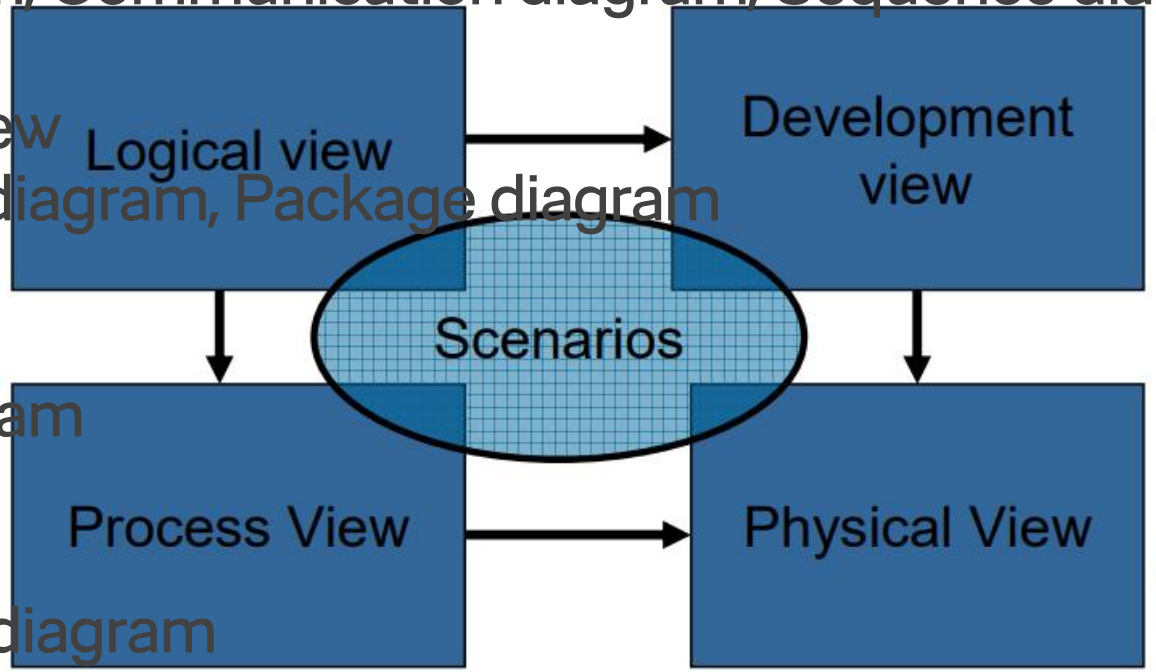
Logical View
- Class diagram, Communication diagram, Sequence diagram

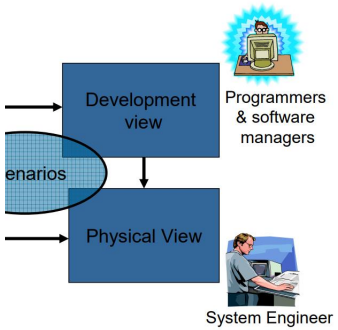
Development View
- Component diagram, Package diagram

Process View
- Activity diagram

Physical View
- Deployment diagram

Scenarios
- Use-case diagram – to demonstrate the architecture





Logical Architecture (LA)

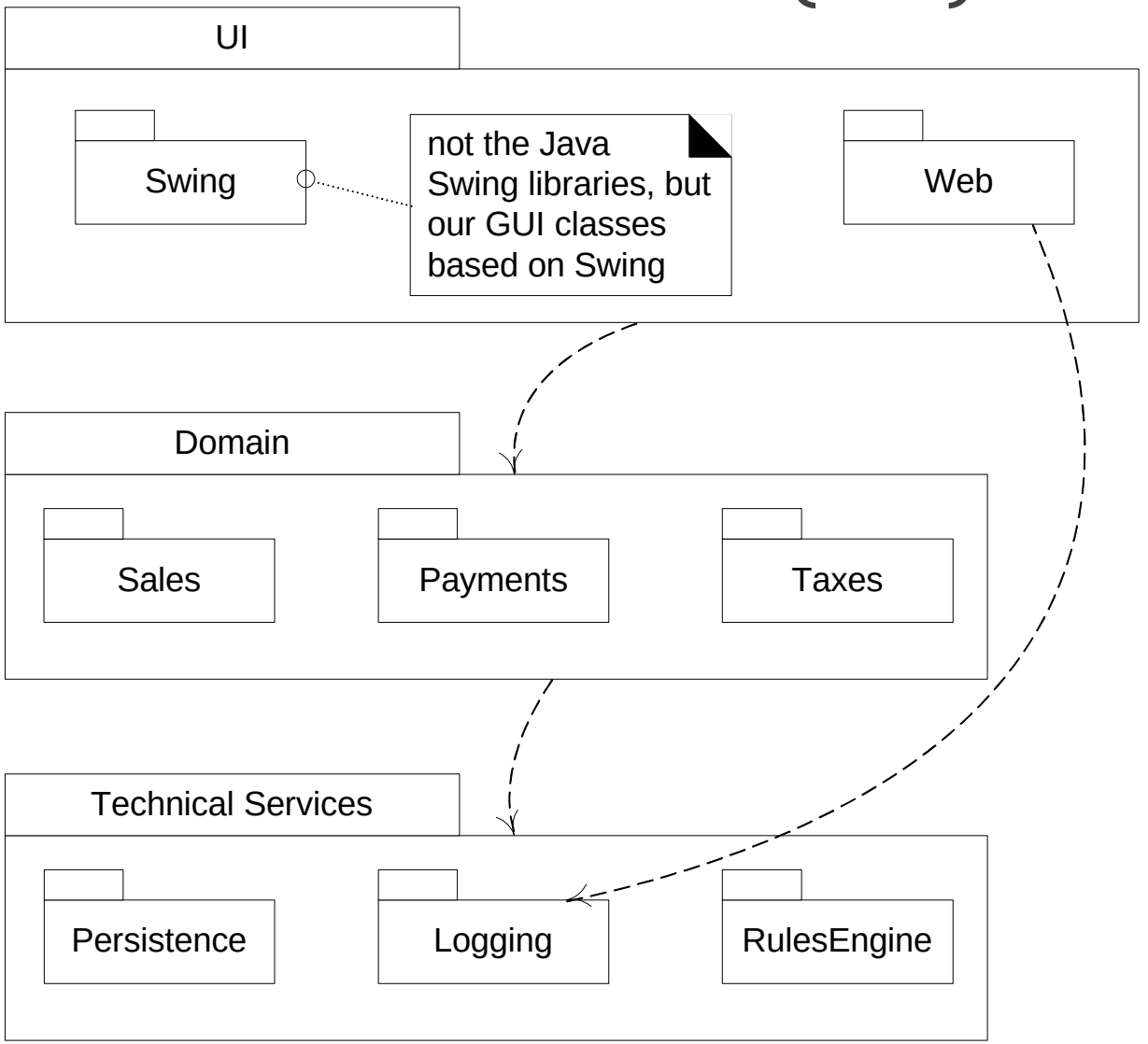
Is a large-scale organisation of software classes into

- Packages (namespaces)
- Logical Layers
  - Groups of classes, packages, subsystems
  - Has a cohesive responsibility for a major aspect of the system
    - UI, Domain objects, Application/business logic, Technical services etc.

It is **NOT**

- In which process the things are executed
- Grouped by physical computers

Logical Architecture (LA)



[Larman, 2005] Figure 13.2

Package Diagrams

Often used to show the Logical architecture of a system

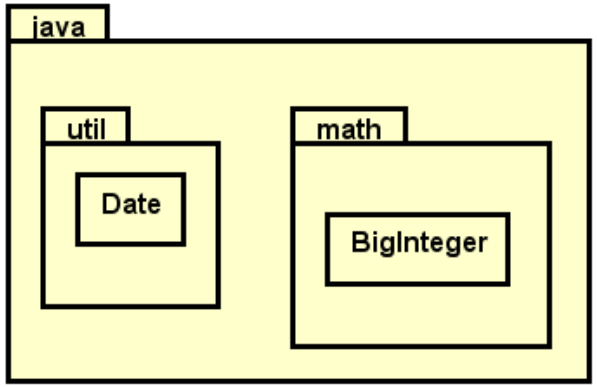
- Layers, sub-systems, packages etc.
- Groups elements together
- More general the a Java package
- Dependencies between packages show the systems large-scale coupling
  - See [Larman, 2005] figure 13.2

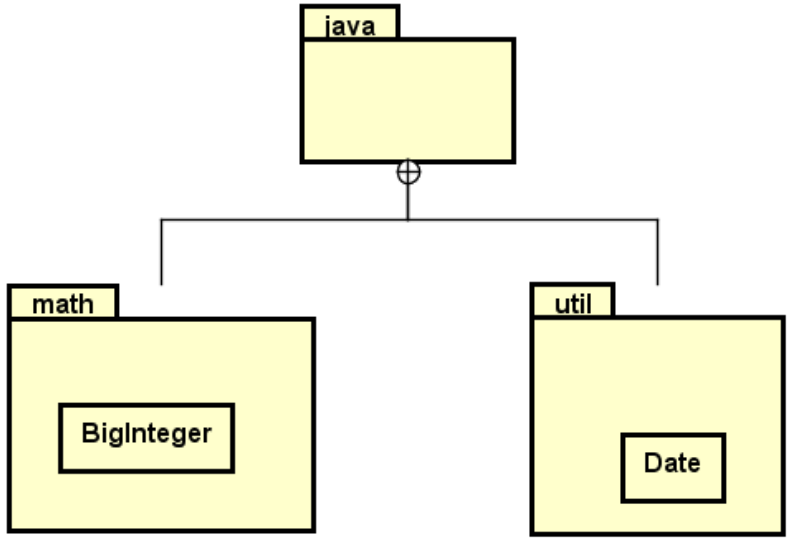
A UML Package represent a Namespace

- Different classes with same name can be defined in different packages

Package Diagrams

Nested classes alternatives





java::util::Date

java::math::BigInteger

Design with Layers

Why layers?

- Organise classes etc. into discrete layers
  - Separate
  - Related responsibility
  - With clean cohesive separation of concerns
    - **Lower level** layers are low-level general services
    - **Higher level** layers are more application specific

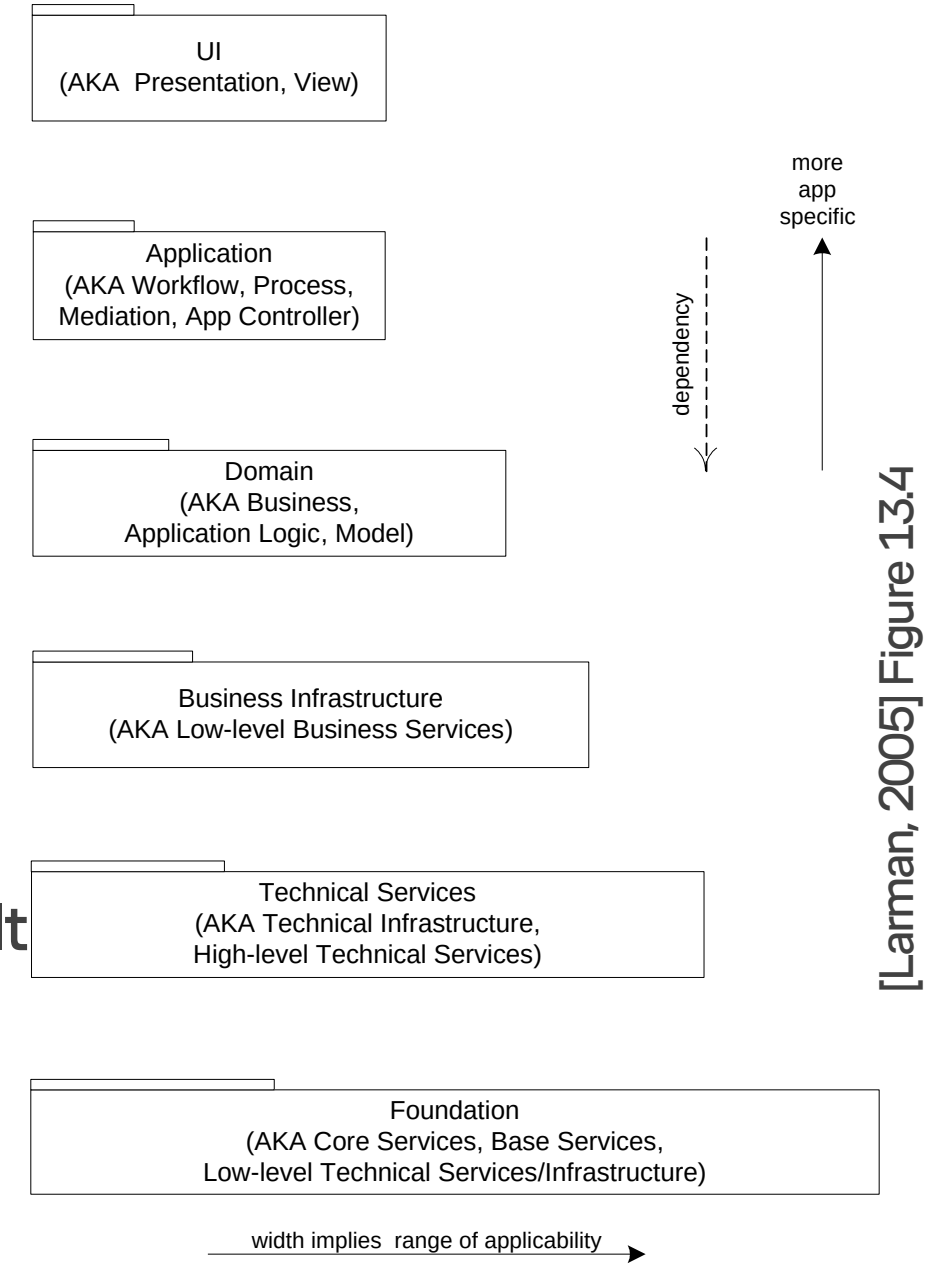
Collaboration between layers

- From higher to lower level
- Avoid lower to high level dependencies

Design with Layers

A layered design addresses these **problems**

- Source code changes ripple to the hole system if many parts are highly coupled
- Application/business logic is spread all over including UI
  - Application/business logic can not be reused
- Technical services are spread all over
  - Can't be reused
- High coupling between concerns makes it difficult to
  - Divide work between developers
  - To test

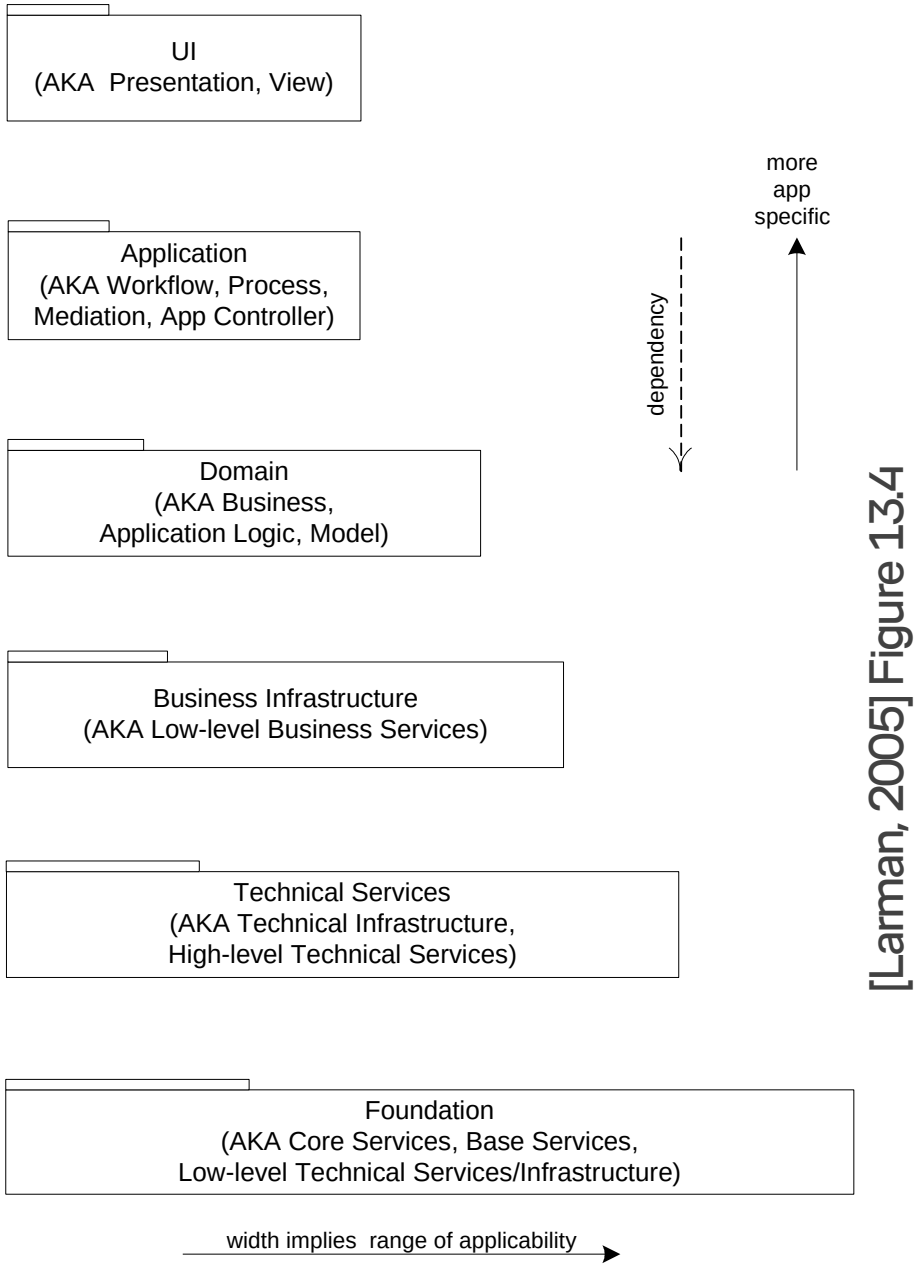


[Larman, 2005] Figure 13.4

Design with Layers

A layered design **benefits**

- Separation of concerns
- Separation of high from low level services
- Related complexity is encapsulated and decomposable
- Some layers can be replaced with new implementations
  - E.g. New UI
- Low level layers contains reusable functionalities
- Some layers can de distributed
- Development in teams because of logical segmentation



[Larman, 2005] Figure 13.4

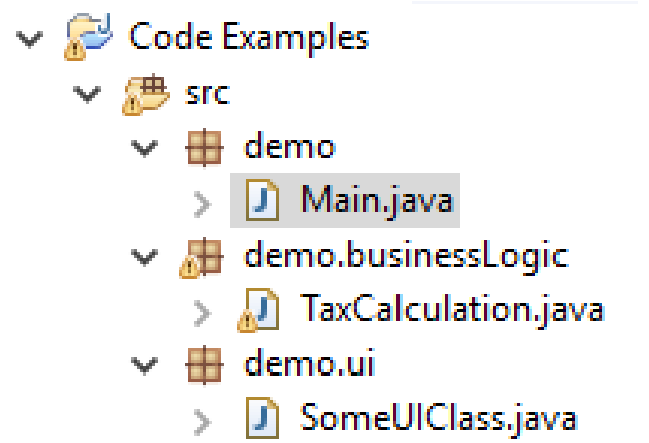
Design with Layers

Cohesive Responsibilities – Separation of concerns

- Objects in one layers is strongly related to the other objects in the same layer
  - Should not be mixed with responsibilities of other layers
  - E.g. Objects in business logic layer should focus on business logic/calculations, not how the UI works

Packages in Code

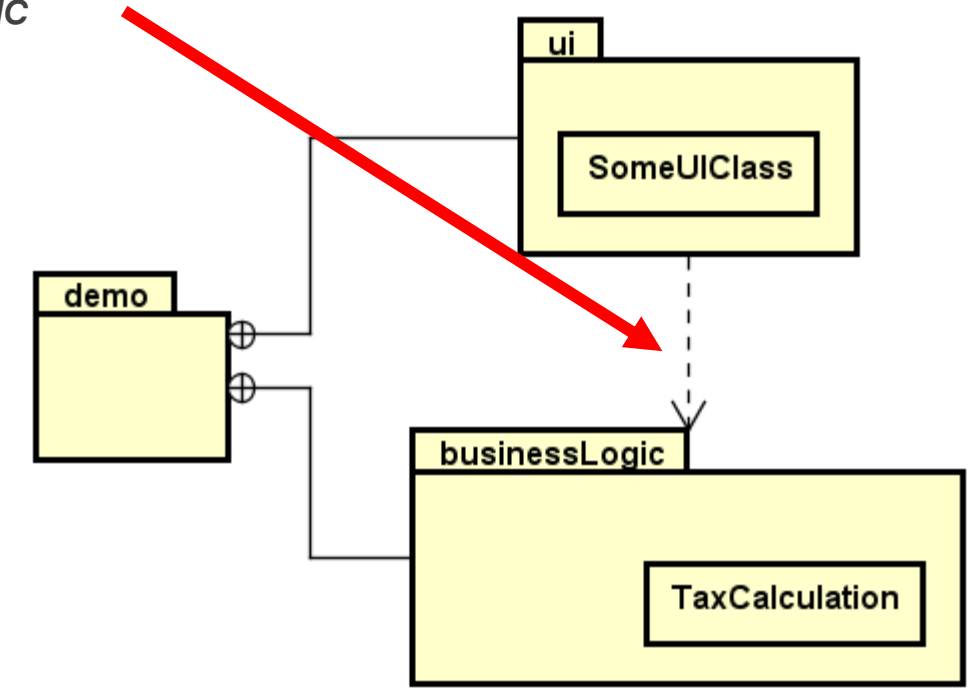
Java has *package*, C# and C++ has *namespaces*




```java
package demo.ui;
import demo.businessLogic.*;

public class SomeUIClass {

}
```

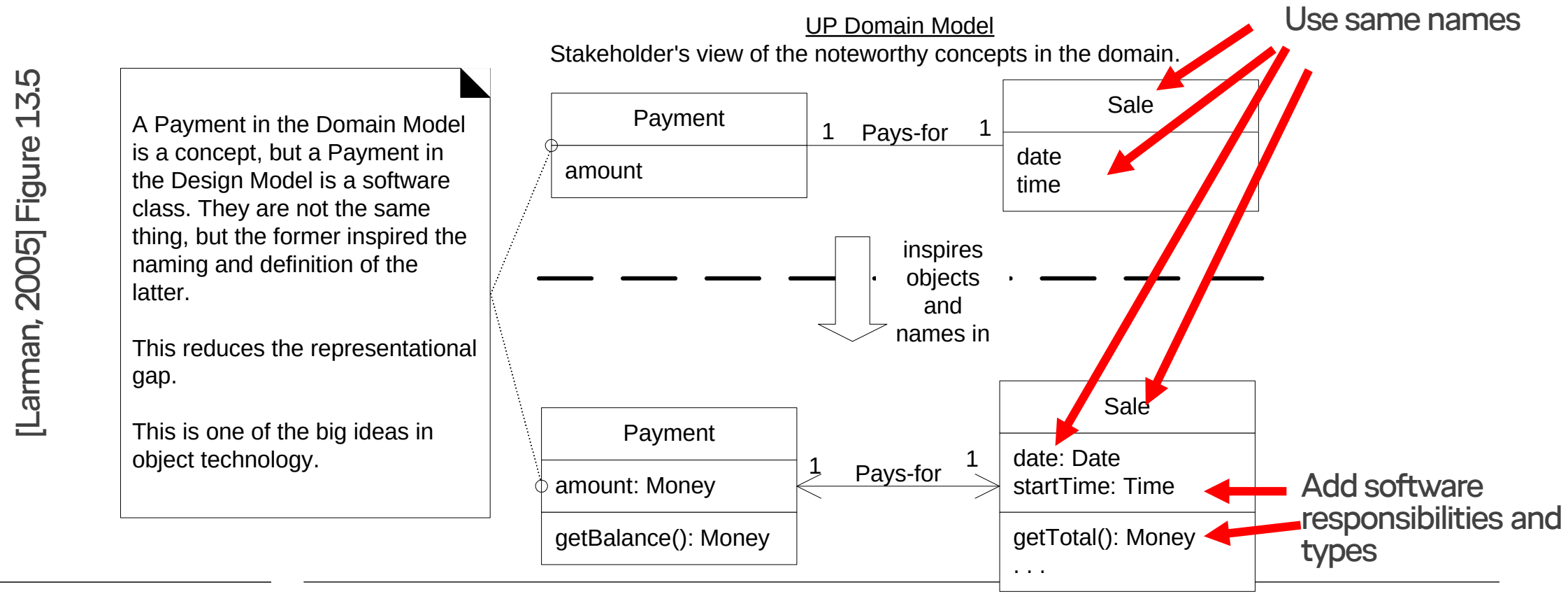
This is the dependency to *businessLogic*

```java
package demo.businessLogic;

public class TaxCalculation {

}
```



From Domain Model to Domain Layer (design)

1. Create classes with names and information similar to Domain Classes
2. Add application logic/software responsibilities to them



Model – View separation Principle/Pattern

Should other packages/layers have access to UI objects?
How should non-UI objects communicate with UI objects?

Principle

1. Do not couple/mix non-UI objects to UI objects!
   - Non-UI objects can't be reused
   - UI objects are application dependent
2. Do not put application logic into UI objects
   - UI objects should only initialise UI elements
   - UI objects should only receive UI events
   - UI objects should delegate request to the application logic in lower layers

Model – View separation Principle/Pattern

Model/domain classes/object should not have direct knowledge of view/UI objects!

- The Observer pattern can handle this for problem ☺ - more later

Domain classes/objects must encapsulate the information and the behaviour related to the application/business logic

GUI/Window classes are **only** responsible for in- and output and catching user events

View: UI/Presentation

- Changes more often
- Needs often an expert in a GUI framework and in user experience

Model: Domain/Business Rules: The part that fulfils the requirements

- Change once in a while
- Needs a Domain expert

Benefits of Model – View separation

Cohesive model definitions that focus on domain process
Separates development of UI and model
Minimise impact of UI requirement changes on domain/model objects
Allow new/different views to easily be connected to existing domain/model layer
Allow different simultaneous views on the same model
Allow execution of model layer without/independent on UI layer
- Makes it easy to test

Allow easy porting of model layer to another UI-Framework

Circle Example

We should implement a system that can handle circles and do some calculations on them (area, circumference etc.). It should be possible to draw the circles on a screen

What classes, attributes and operations will you have?

Exercise – Flipped Classroom

Find out what the
Observer Design pattern (behavioural pattern) is
and how it works

Prepare a small presentation to show for the rest of the class including

- Class diagram that shows the Observer pattern structure
- A sequence diagram that shows the behaviour of the Observer pattern on a simple example
- A simple Java implementation that shows how to use the Observer pattern

Diagrams must be made in Astah, even that you can find them else where.
(You need more Astah training!!! ☺)

Exercise 2

Think about how the Observer pattern can help to decouple UI layer from Model layer

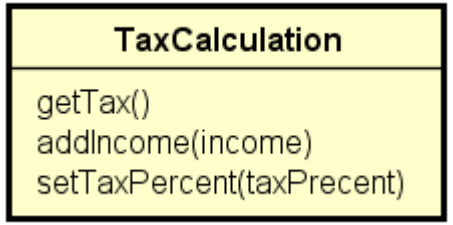
Design a simple tax calculation system with max 4 classes that use packages for View and Model layers

Prepare a small presentation to show for the rest of the class including

- Class diagram that shows the View and the Model and what classes are in these packages
- A sequence diagram that shows the behaviour of the system
- A simple Java implementation that shows the implementation of the system

The View must use the console as UI

Must be part of the system →



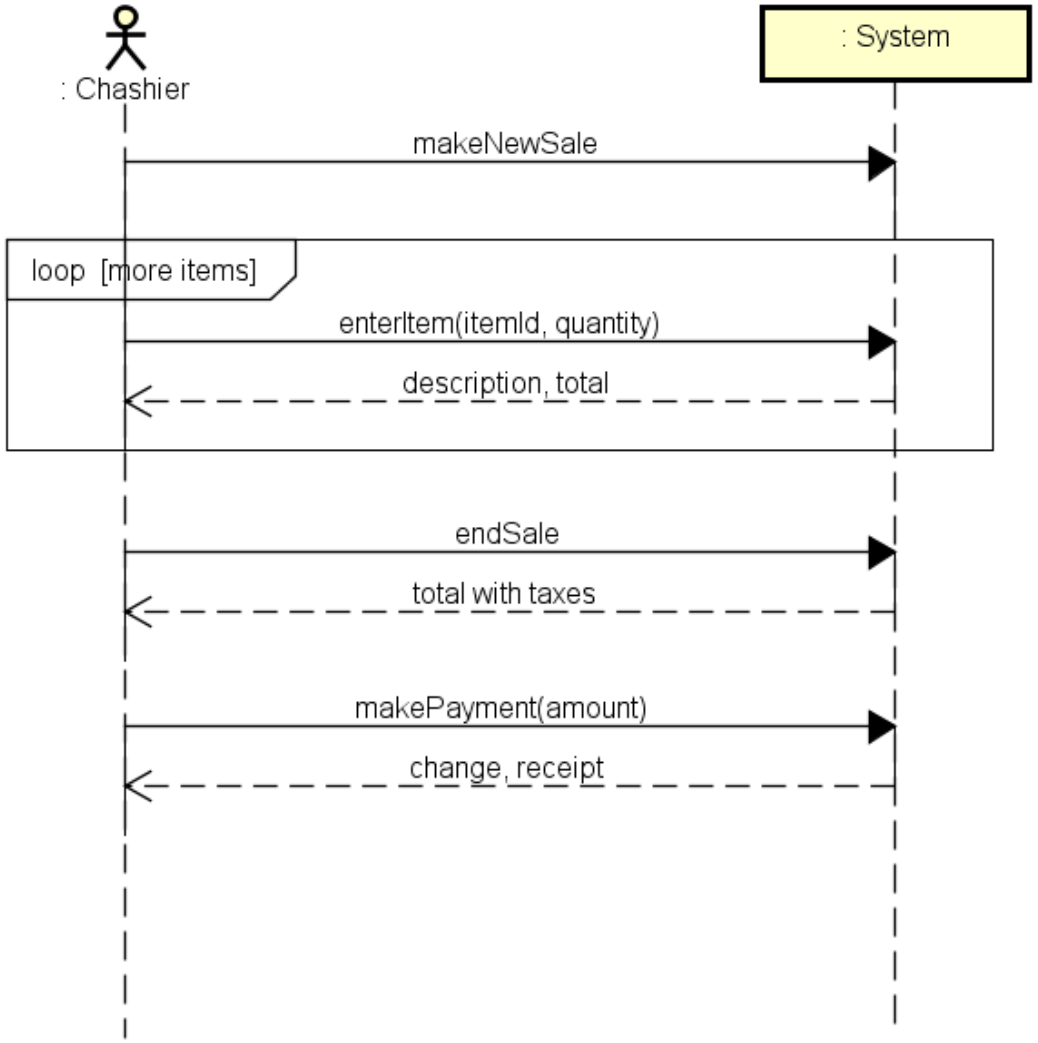
Diagrams must be made in Astah!

System Sequence Diagrams (SSD) and Layers

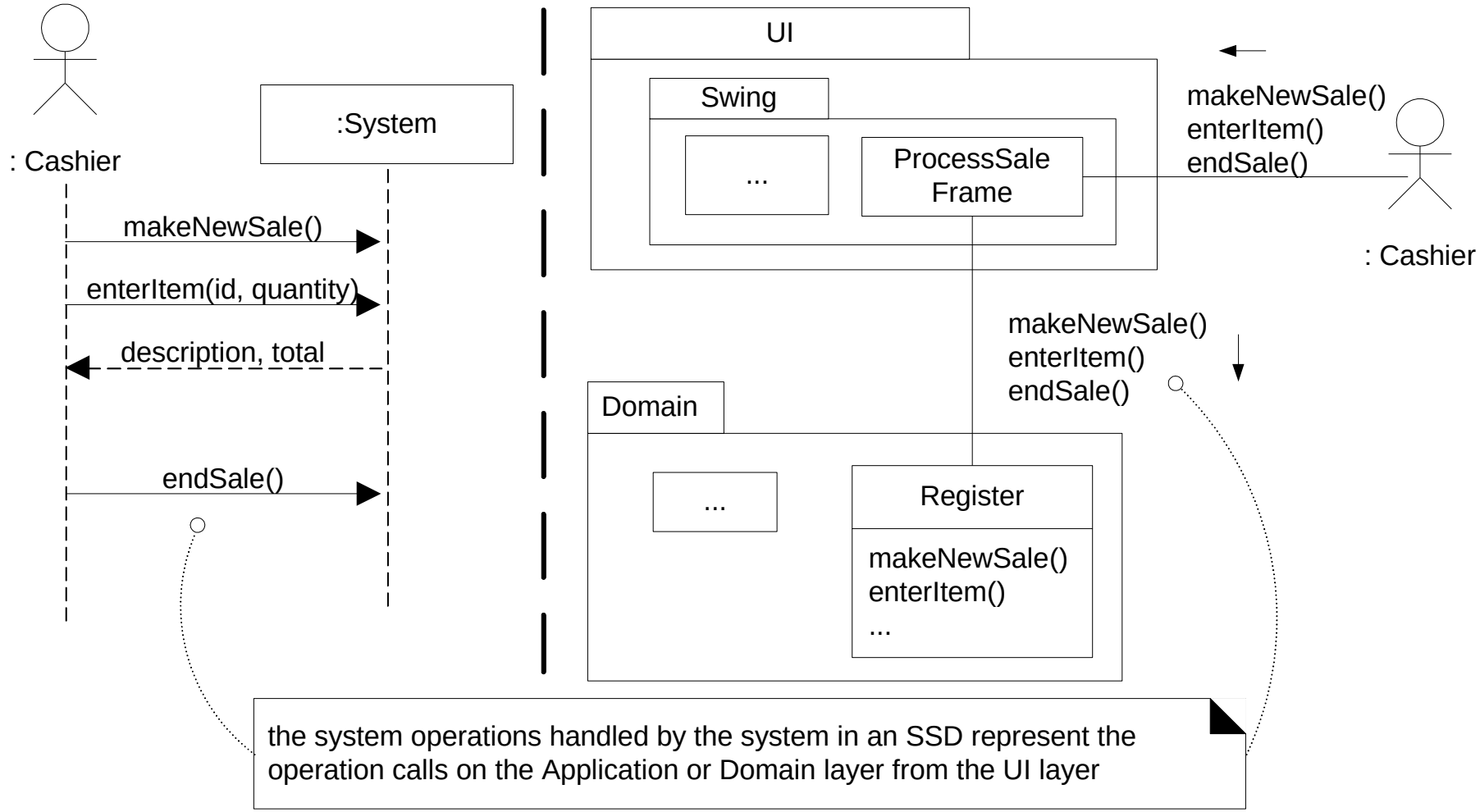
The SSD is focussing on events between Actor and system

- The UI are newer shown in SSDs

In design it will normally be UI objects that captures the events from users



System Sequence Diagrams (SSD) and Layers



[Larman, 2005] Figure 13.8