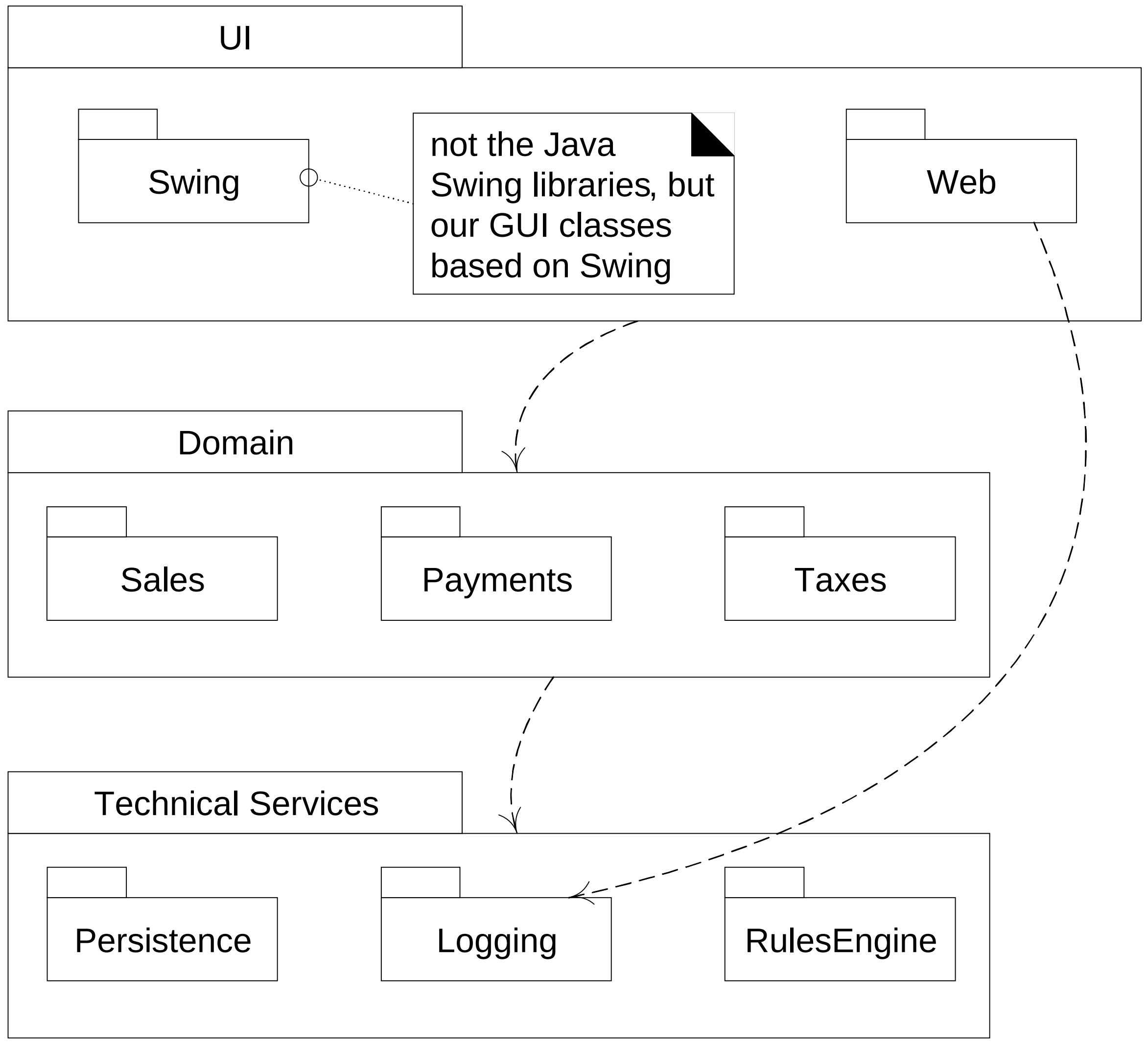
Logical Architecture and UML Package Diagrams

The Large-Scale

- At this level, the design of a typical OO system is based on several architectural layers, such as
  - UI layer,
  - Application logic (or "domain") layer,
  - Technical Services



Layers

- **Layer** is a coarse-grained grouping of classes, packages, or subsystems that has cohesive (strongly related) responsibilities for a major aspect of the system.
  - E.g. Application Logic and Domain Objects— software objects representing domain concepts (for example, a software class Sale) that fulfill application requirements, such as calculating a sale total.
- Application layer is the focus of Use Cases.

- Higher layers (such as UI layer) call upon services of lower layers, but not normally vice versa.

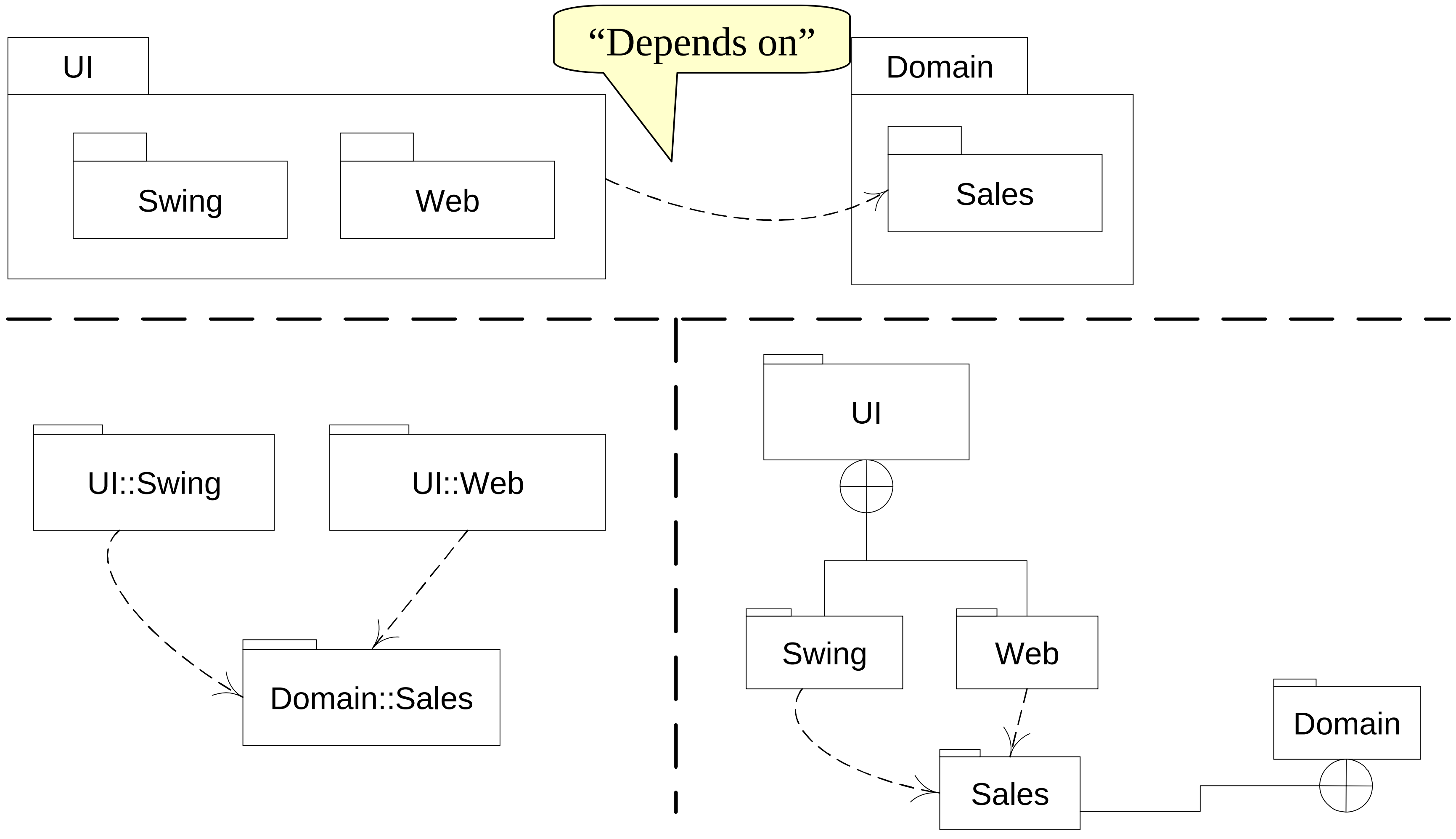
- In a strict layered architecture, a layer only calls upon the services of the layer directly below it.
  - Common in network protocols
- But not in information systems, which have a relaxed layered architecture, in which a higher layer calls upon several lower layers.
  - For example, UI layer may call upon its directly subordinate application logic layer, and also upon elements of a lower technical service layer, for logging and so forth.

Software Architecture

- A software architecture is
  - the set of **significant decisions** about the organization of a software system,
  - the **selection of the structural elements** and their interfaces by which the system is composed,
  - together with their **behavior** as specified in the **collaborations** among those elements, and
  - the **composition** of these structural and behavioral elements into progressively larger subsystems.

UML Package Diagrams

Alternate notations for the same thing



Problems when not using layers

- Application logic is intertwined with the user interface, so
  - it cannot be reused with a different interface or distributed to another processing node.
- General technical services are intertwined with application-specific logic, so they
  - cannot be reused,
  - distributed to another node, or
  - easily replaced with a different implementation.
- There is high coupling across different areas of concern.
  - It is thus difficult to divide the work along clear boundaries for different developers.

Mapping Code Organization to Layers and UML Packages

```
// --- UI Layer

com.mycompany.nextgen.ui.swing
com.mycompany.nextgen.ui.web


// --- DOMAIN Layer

   // packages specific to the NextGen project
com.mycompany.nextgen.domain.sales
com.mycompany.nextgen.domain.payments


// --- TECHNICAL SERVICES Layer

   // our home-grown persistence (database) access layer
com.mycompany.service.persistence

   // third party
org.apache.log4j
org.apache.soap.rpc

// --- FOUNDATION Layer

   // foundation packages that our team creates
com.mycompany.util
```

To support cross-project reuse, we avoid using a specific application qualifier ("nextgen") in the package names unless necessary.

The UI packages are related to the NextGen POS application, so they are qualified with the application name …nextgen.ui.*.

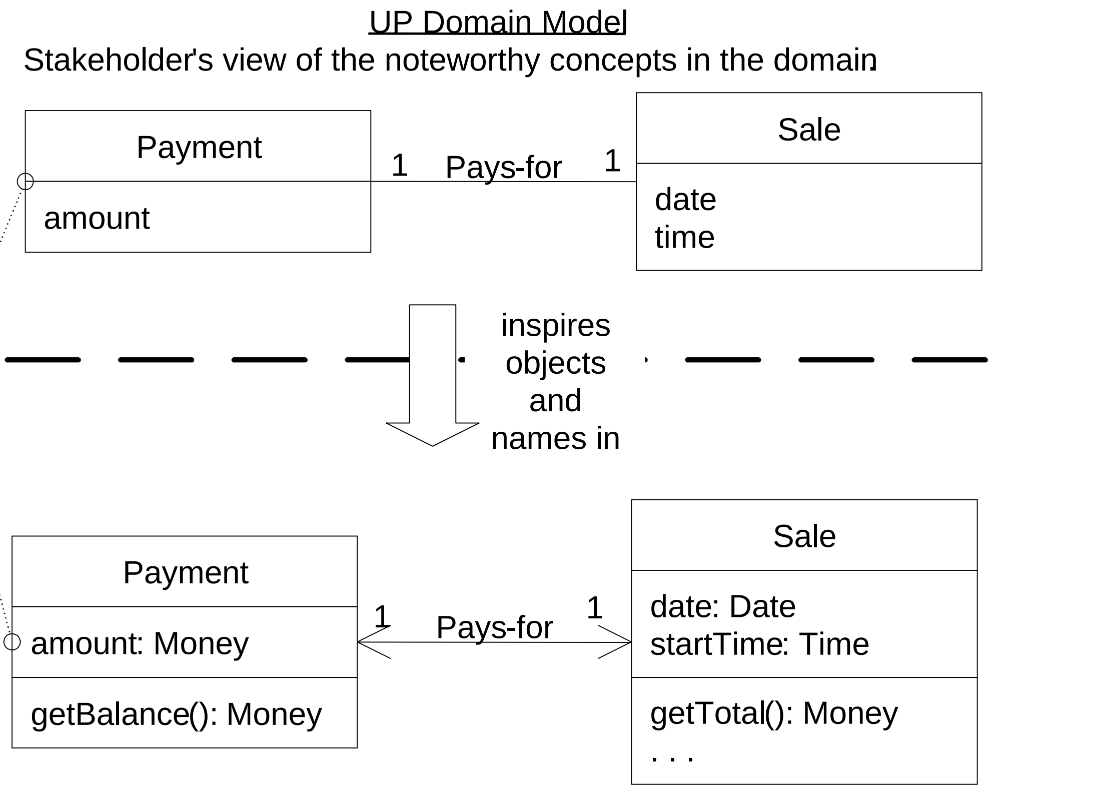
But utilities could be shared across many projects, hence the package com.mycompany.utils, not com.mycompany.nextgen.utils

Relationship Between the Domain Layer and Domain Model

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.



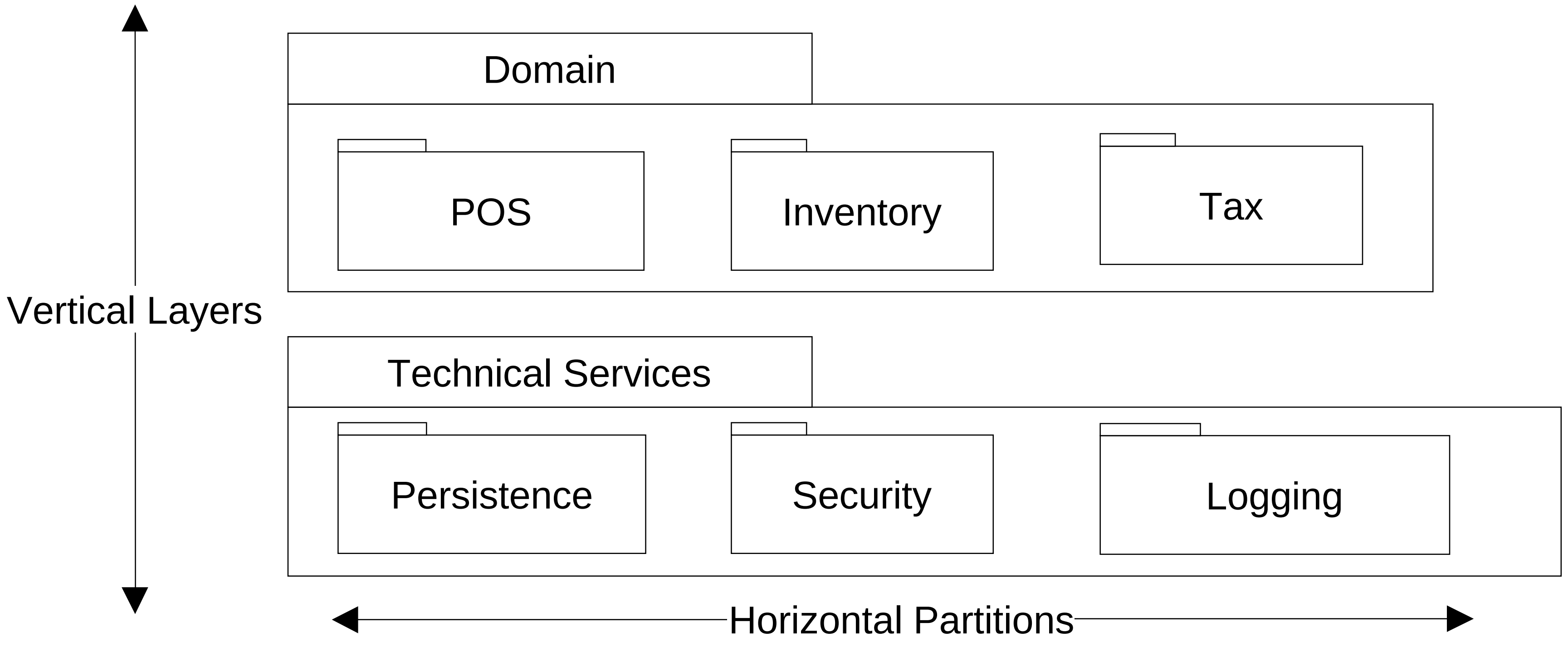
Domain layer of the architecture in the UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Layers and Partitions

- **Layers** of an architecture represent the vertical slices, while **partitions** represent a horizontal division of relatively parallel subsystems of a layer.
  - E.g., **Technical Services** layer may be divided into partitions such as *Security* and *Reporting*.



Model-View Separation Principle

- **Model** is a synonym for the **domain layer** of objects (it's an old OO term from the late 1970s).
- **View** is a synonym for UI objects, such as windows, Web pages, applets, and reports.

Principle:

- Model (domain) objects should not have direct knowledge of view (UI) objects.
  - E.g. **Register** or **Sale** objects should not directly send a message to a GUI window object **ProcessSaleFrame**, asking it to display something, change color, close, and so forth.

History:

- Pattern Model-View-Controller (MVC) originally a small-scale Smalltalk-80 pattern.
- The Model is the Domain Layer, the View is the UI Layer, and the Controllers are the workflow objects in the Application layer.

Further part of this principle

- Domain classes encapsulate the information and behavior related to application logic.
- Window classes are thin; they are responsible for
  - input and output, and
  - catching GUI events,
- Window classes do not maintain application data or directly provide application logic. E.g.,
  - A Java JFrame window or a Web JSP page should not have a method that does a tax calculation.
- These UI elements should delegate to non-UI elements for such responsibilities.

Legitimate relaxation

- **Observer pattern**, where domain objects can send messages to UI objects viewed only in terms of an interface such as **PropertyListener** (a common Java interface for this situation).

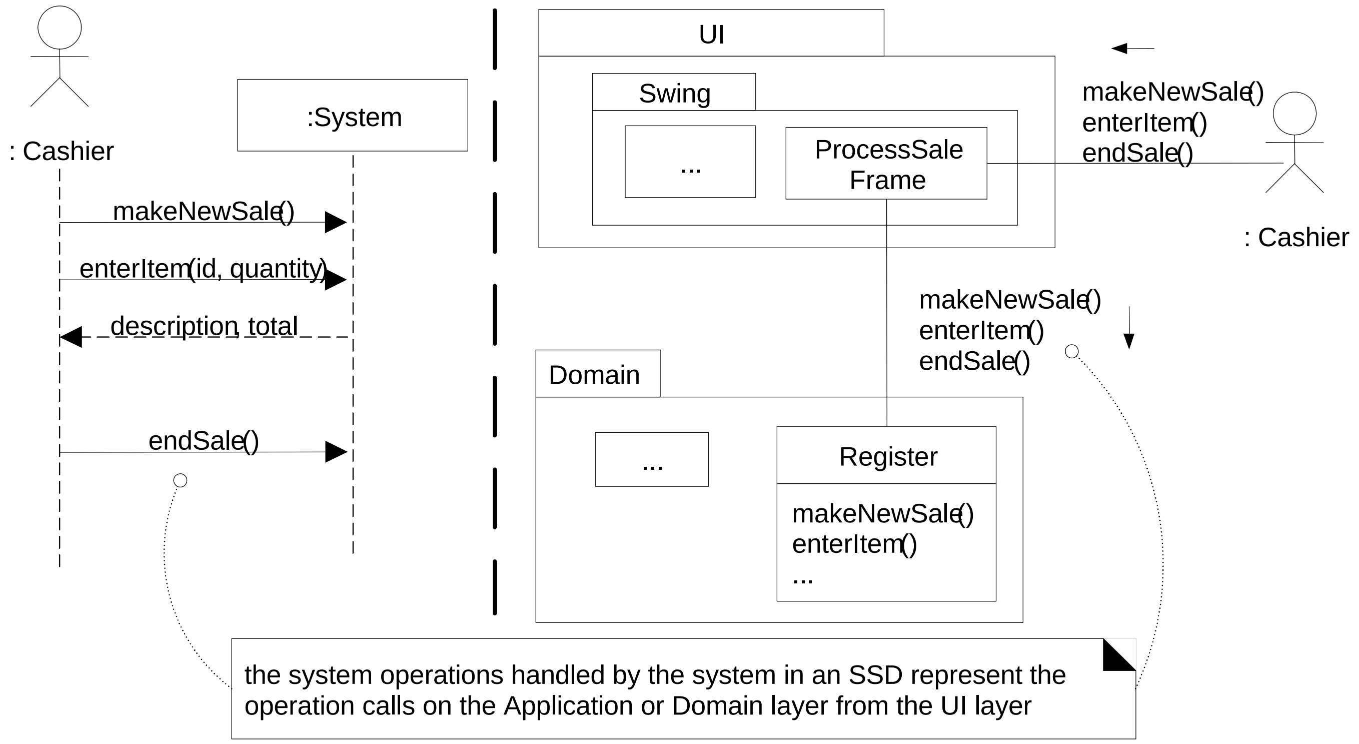
- Then, the domain object doesn't know that the UI object is a UI object—it doesn't know its concrete window class. It only knows the object as something that implements the **PropertyListener** interface.

Motivation for M-V Separation

- To allow separate development of the model and user interface layers.
- To minimize the impact of requirements changes in the interface upon the domain layer.
- To allow new (multiple simultaneous) views to be easily connected to an existing domain layer, without affecting the domain layer.
- To allow execution of the model layer independent of the user interface layer, such as in a batch-mode system.

Connection between SSDs and Layers

- SSDs illustrate system operations, but hide the specific UI objects.
- Nevertheless, it will be objects in the UI layer of the system that capture these system operation requests.
  - UI layer objects will then forward—or delegate—the request from the UI layer onto the domain layer for handling.


: Cashier
:System
makeNewSale()
enterItem(id, quantity)
description, total
endSale()
UI
Swing
...
ProcessSale Frame
makeNewSale()
enterItem()
endSale()
: Cashier
makeNewSale()
enterItem()
endSale()
Domain
...
Register
makeNewSale()
enterItem()
...
the system operations handled by the system in an SSD represent the operation calls on the Application or Domain layer from the UI layer