AS - Teoria 2

Unit 2.2. Object Oriented Architecture

Context

A system can be seen as a collection of objects that in response to certain external stimulus or internal events, interchange information, change their state and eventually provide observable results.

Problem

It is necessary to design the system as a collection of objects, having each object assigned some responsibilities.

The system must combine two viewpoints:

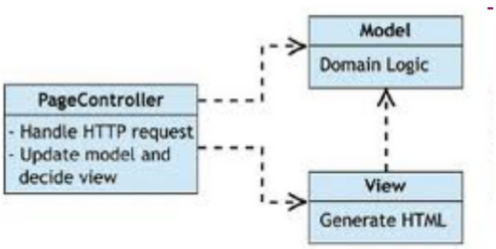
- STATIC VIEW object properties that configure the system state
- DYNAMIC VIEW responses that objects produce to different events (state changes, interchange of information, observable results,...)

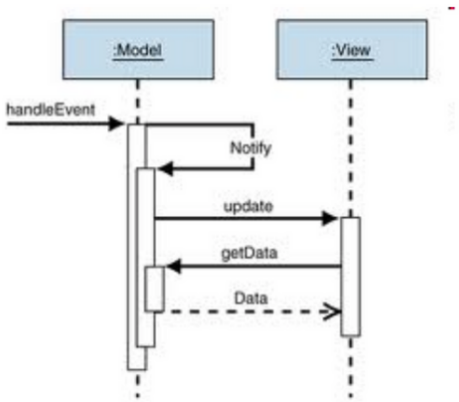
- The system shall be portable to other platforms
- Code changes should not propagate throughout all the system, Similar responsibilities should be grouped
- Components should be reused and substituted to use them in alternative implementations

Solution

Structure the system as a collection of objects that together configure (part of) the
architecture.

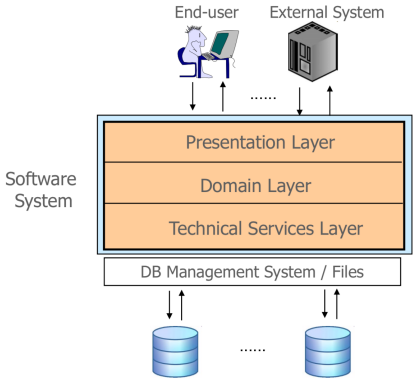
- Assign responsibilities to these objects in a systematic way.
- Provide a static view of the system, declaring the classes which the objects belong to, with their attributes, operations, interrelations (associations, inheritance,...)
  and all the information considered important at design level (visibility, ...).
- Provide a dynamic view of the system, that identifies the events that cause changes in
  the system, and for each event the resulting sequences of action.

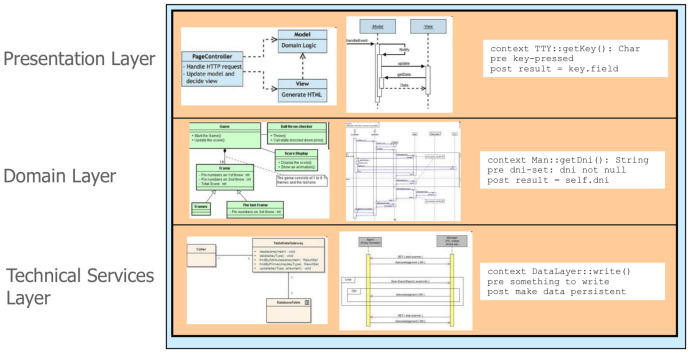





```
context TTY::getKey(): Char
pre key-pressed
post result = key.field
```

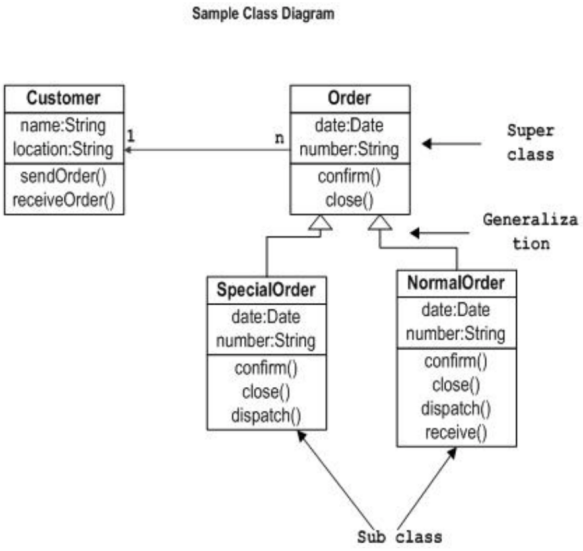
Application to Software Systems

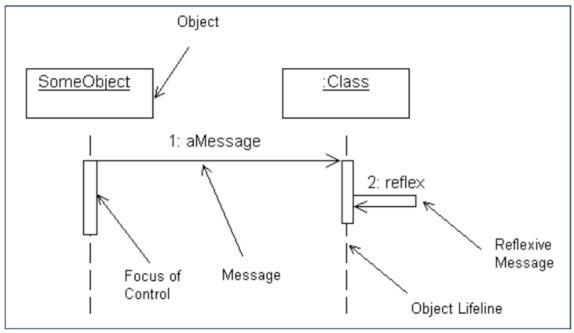




UML Elements for Design

Classes, attributes, associations, operations, sequence diagrams.

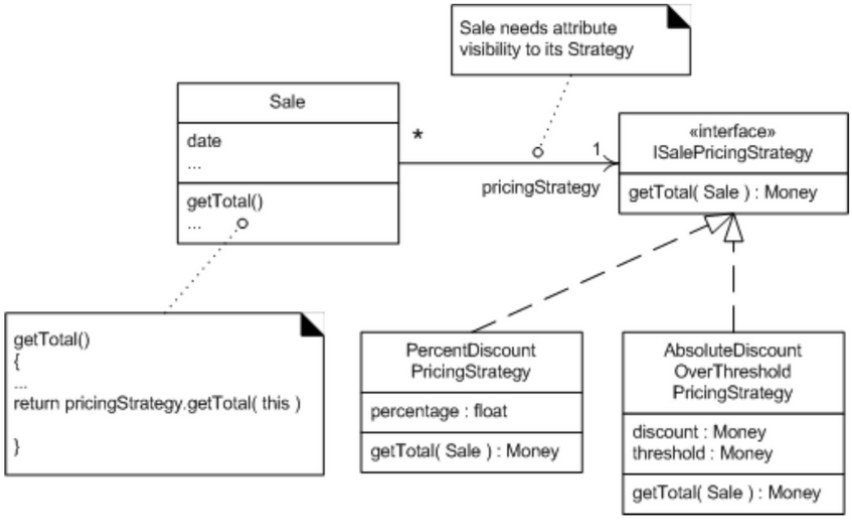




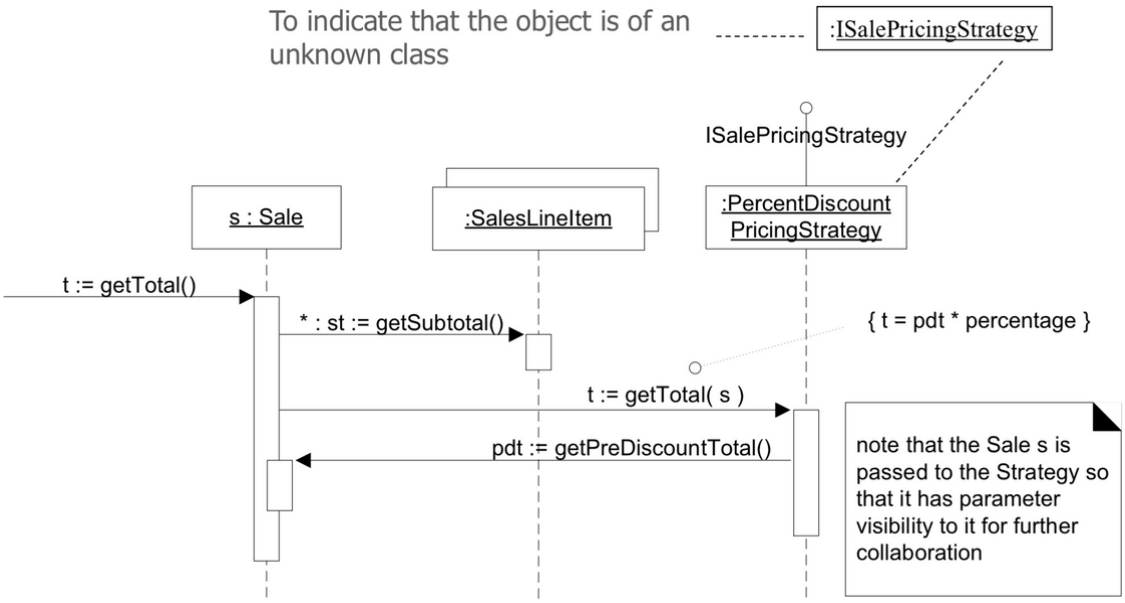
An **interface** represents a declaration made available to or required from anonymous classifiers. The purpose of interfaces is to decouple direct knowledge of classifiers
that must interact to implement behavior.

An interface is essentially equivalent to an abstract class with no private attributes and no methods and only abstract operations. All the features of an interface have public visibility and has not direct instances.

Interface Notation:



Sequence Diagram Notation with Interfaces:



Differences between **interfaces** and **abstract classes**

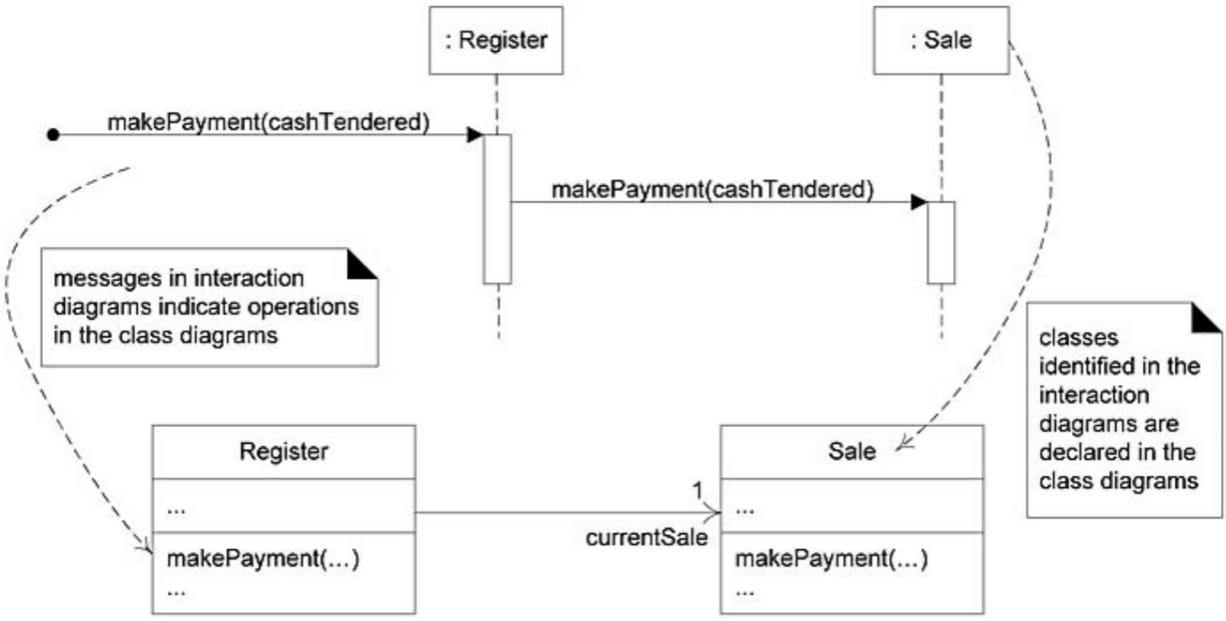
- At the same time multiple interfaces may be implemented. A class may only inherit from one (abstract) class.
- An abstract class may have some method implementation (non- abstract methods, constructors, instance initializers and instance variables) and non-public members.

Interfaces or **abstract classes**?

- To provide common implementation to subclasses an abstract class is used
- To declare non-public members, use abstract classes.
- Use abstract classes if new public methods need to be added in the future.
- To provide the implementing classes the opportunity to inherit from other sources at the same time, use an interface.

Object Oriented Architecture Example

- NextGen is a point-of-sale system (POS) used to record sales and handle payments.
- Class diagram and sequence diagram (Domain Layer)

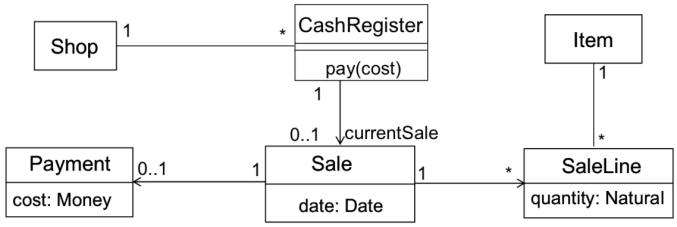


Object Oriented Architecture Design Principles

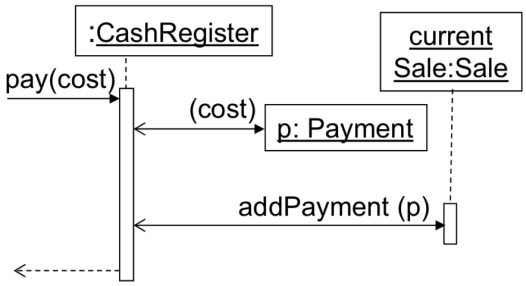
Low Coupling Principle

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

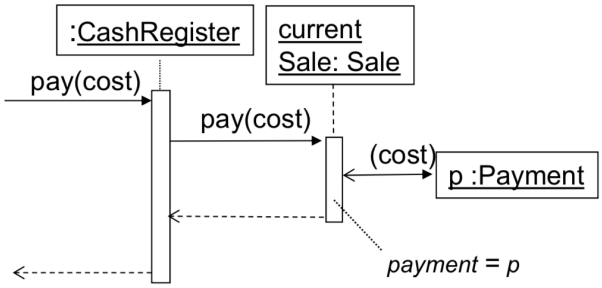
- Low Coupling Principle: Keep coupling as low as is possible. An element with low (or weak) coupling is not dependent on too many other elements.
- Demeter's law ("Do not talk to strangers") helps maintaining a low coupling.



Low Coupling Violation



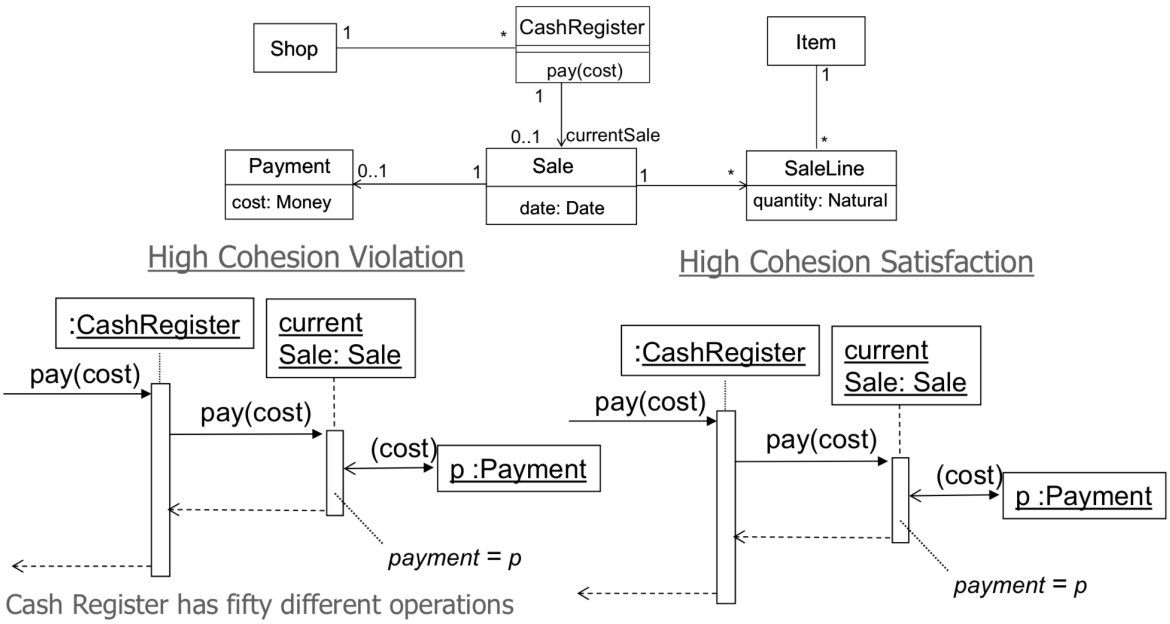
Low Coupling Satisfaction



High Cohesion Principle

Cohesion is a measure of how strongly related and focused the responsibilities of an element are.

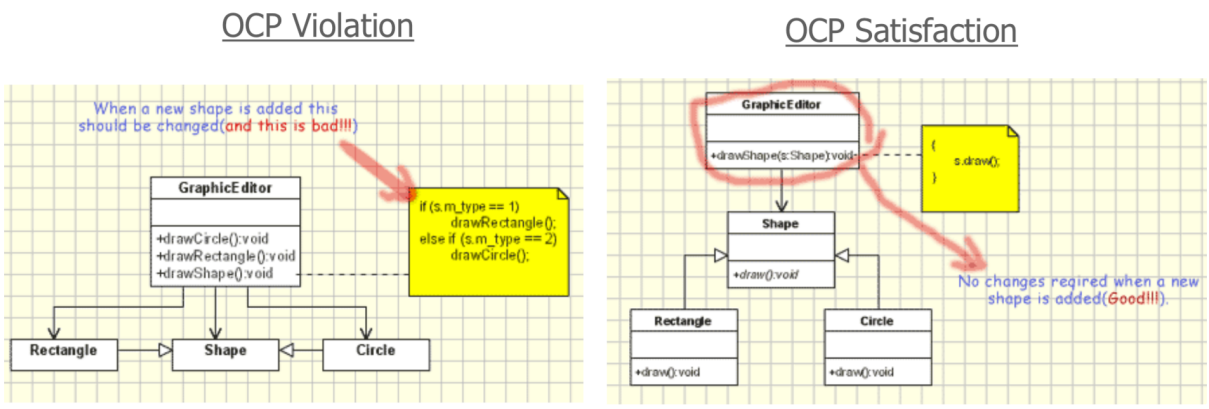
- The High Cohesion Principle: Keep cohesion as high as is possible. An element with highly related responsibilities that does not do a tremendous amount of work has high
  cohesion.



Type '/' for commands

Open Closed Principle

The Open Closed Principle (OCP): A module should be open for extension but closed for modification



Dependency Inversion Principle

- The Dependency Inversion Principle (DIP): High level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

DIP Violation

DIP Satisfaction