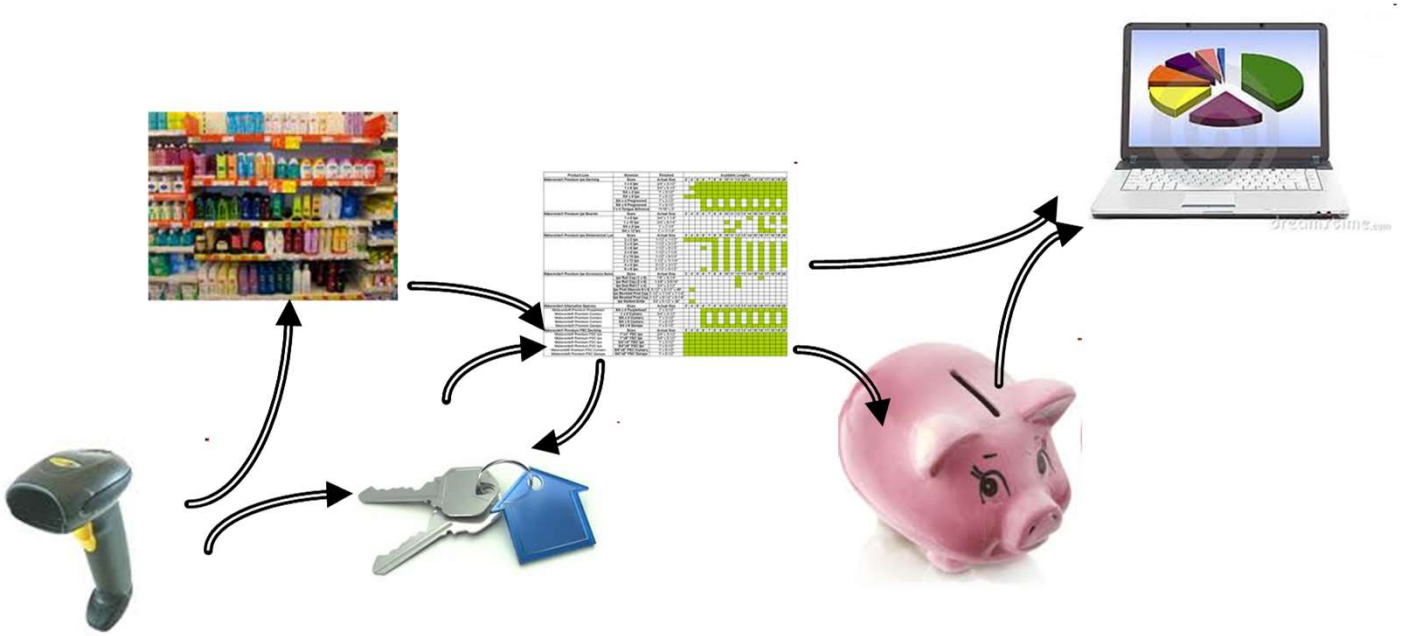
Object Oriented Architecture

Object Oriented Architecture

- Context
- Problem
- Solution
- Application to Software Systems
- UML Elements for Design
- Object Oriented Architecture Example
- Object Oriented Architecture Design Principles
- References

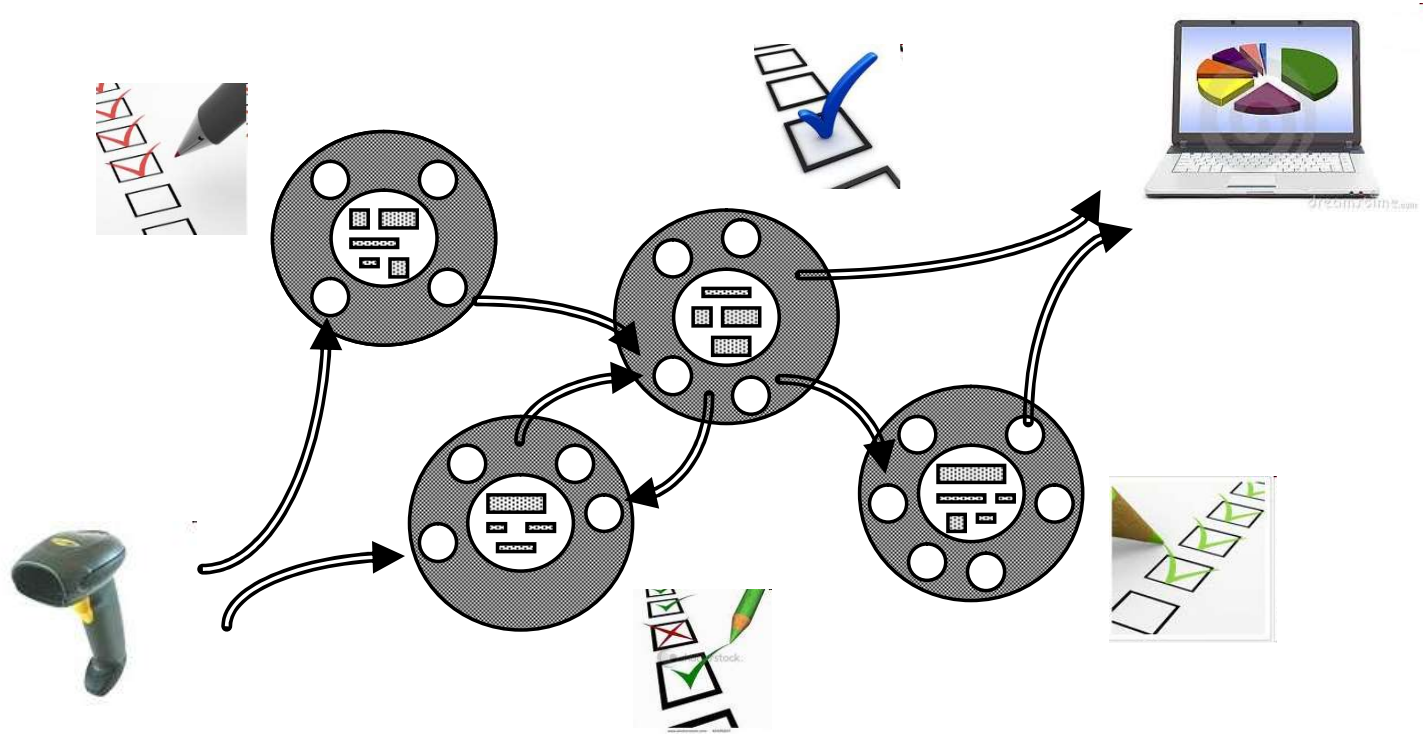
Context

- 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.



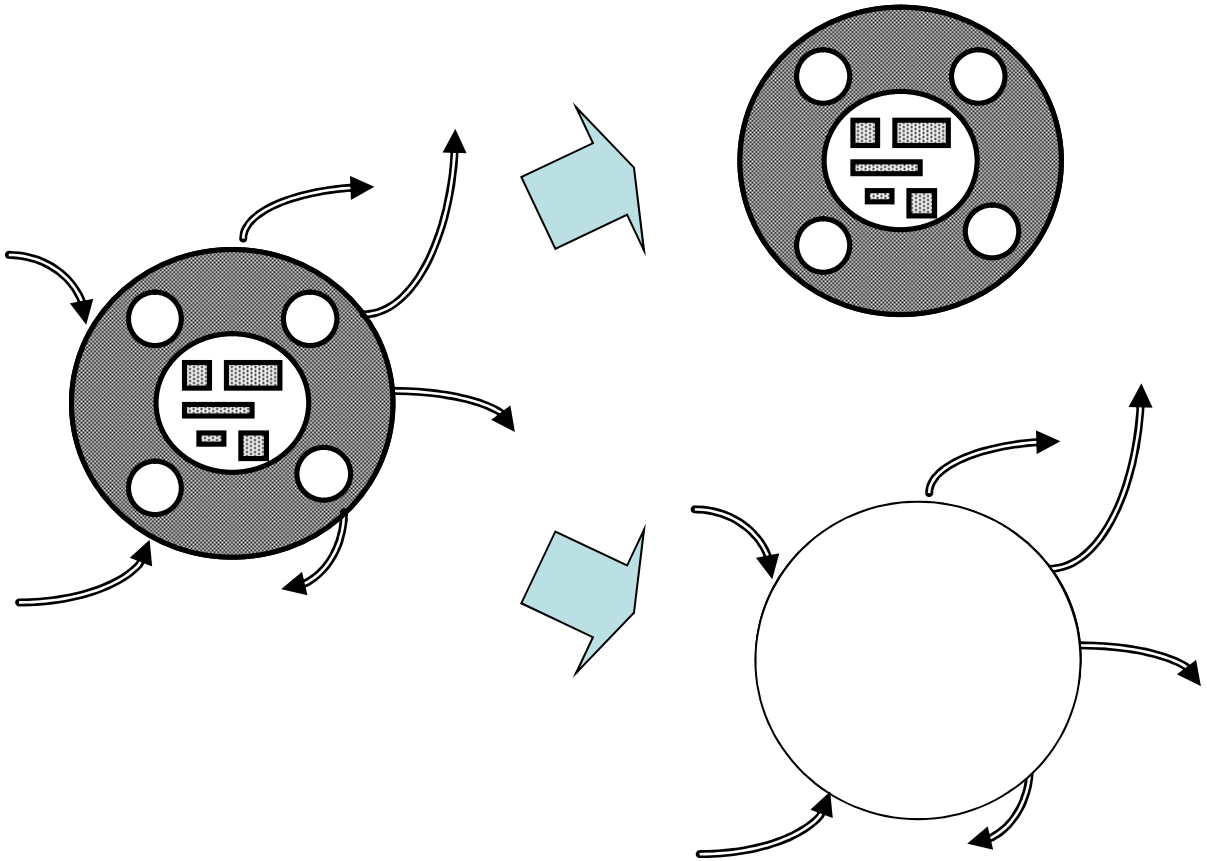
Problem

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



Problem

- The system must combine two viewpoints:



STATIC VIEW

object properties that configure the system state

DYNAMIC VIEW

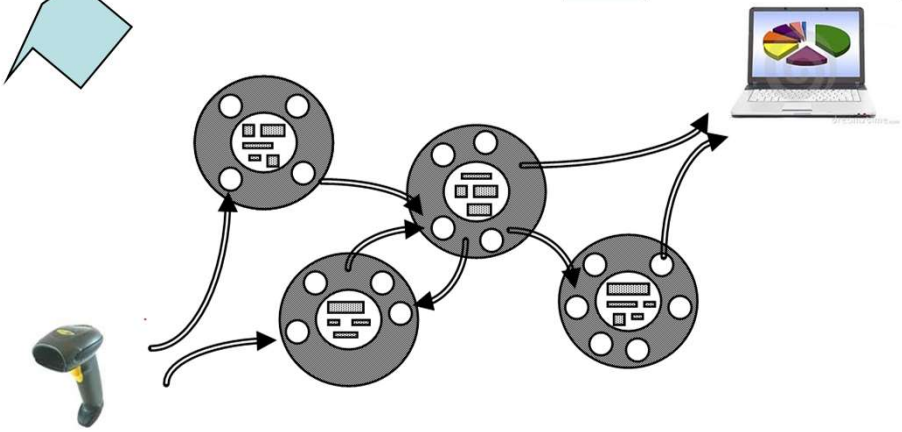
responses that objects produce to different events (state changes, interchange of information, observable results,…)

Problem

- Code changes should not propagate throughout all the system
- Similar responsibilities should be grouped

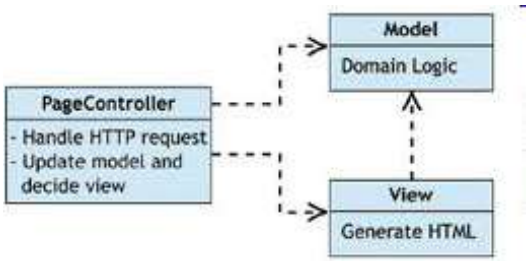
Components should be reused and substituted to use them in alternative implementations

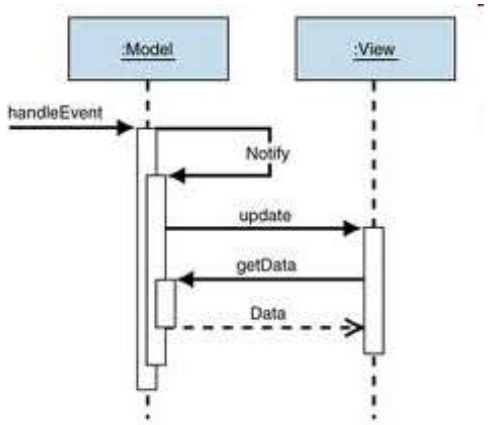
The system shall be portable to other platforms



Solution

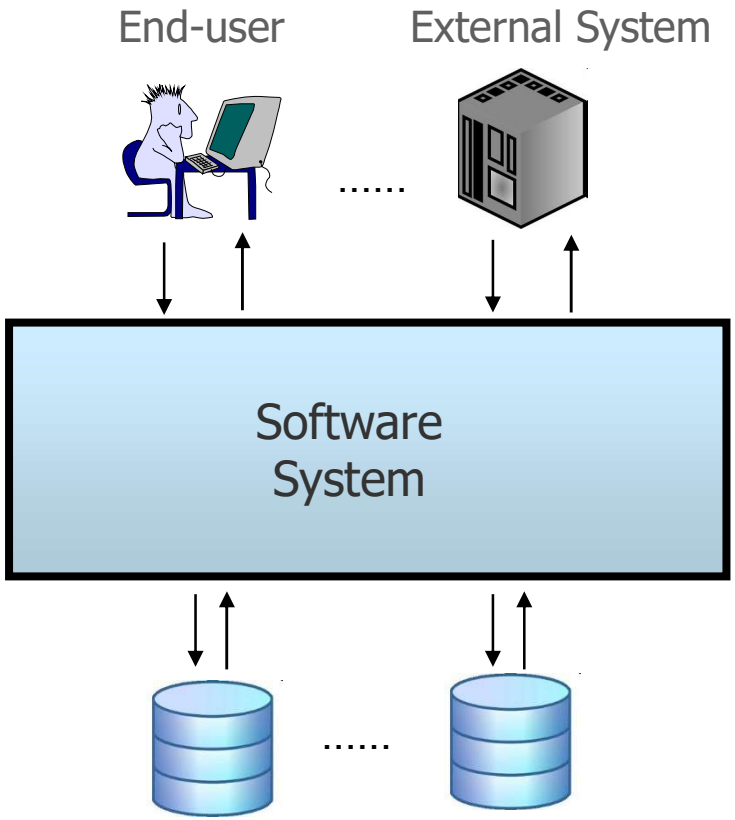
- Structure the system as a collection of objects that together configure (part of) the architecture.
- Assign responsibilities to these objects in a systematic way.
- Provide a static view of the system, declaring the classes which the objects belong to, with their attributes, operations, interrelations (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.



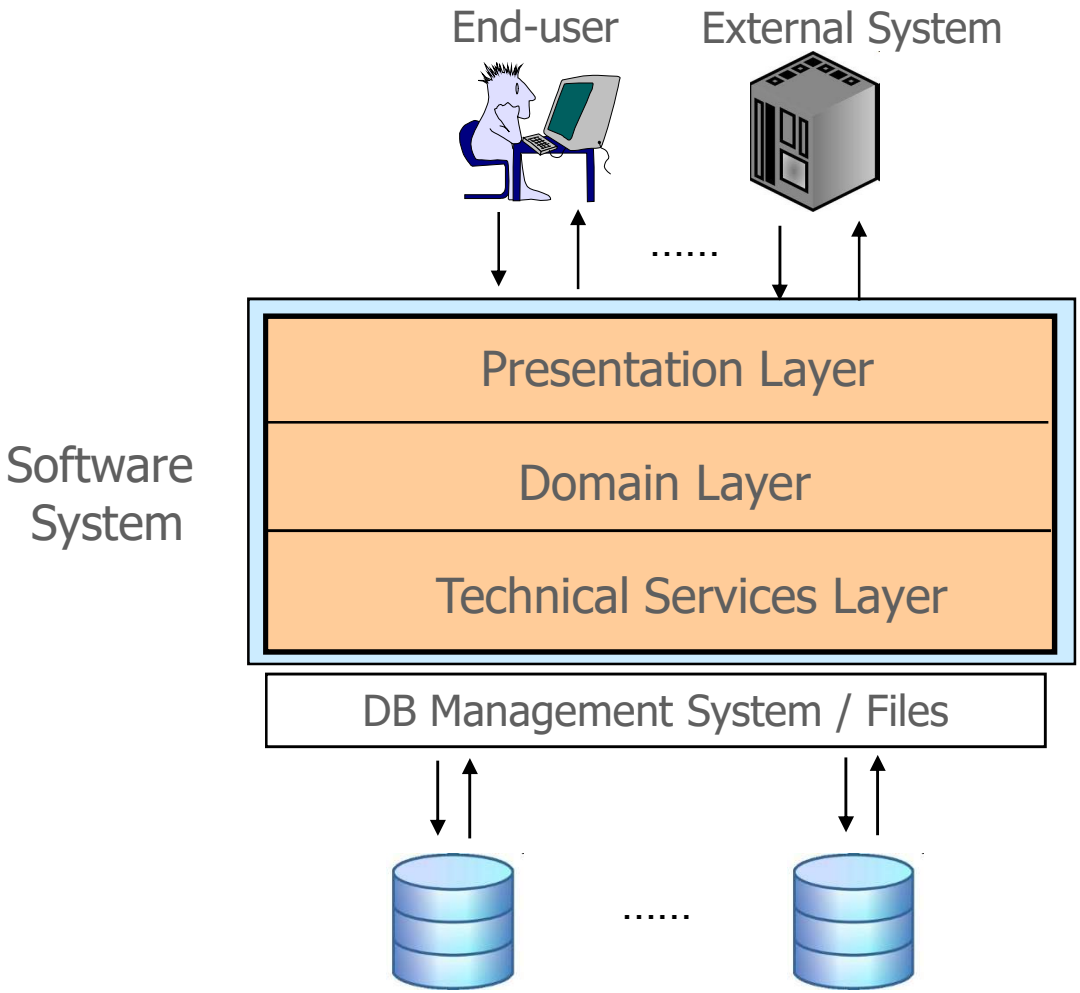



```
context TTY::getKey(): Char
pre key-pressed
post result = key.field
```

Application to Software Systems

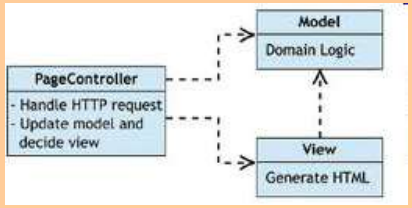


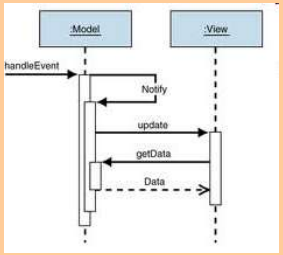
Application to Software Systems



Application to Software Systems

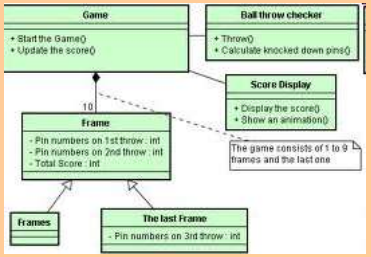
Presentation Layer

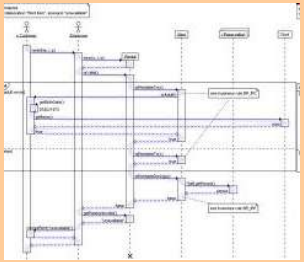





```
context TTY::getKey(): Char
pre key-pressed
post result = key.field
```

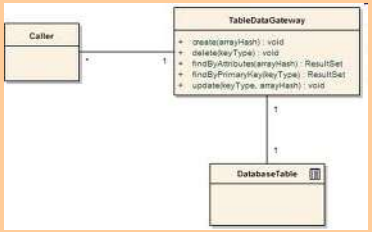
Domain Layer

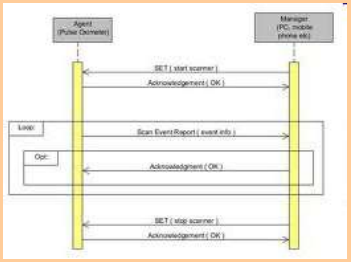




```
context Man::getDni(): String
pre dni-set: dni not null
post result = self.dni
```

Technical Services Layer

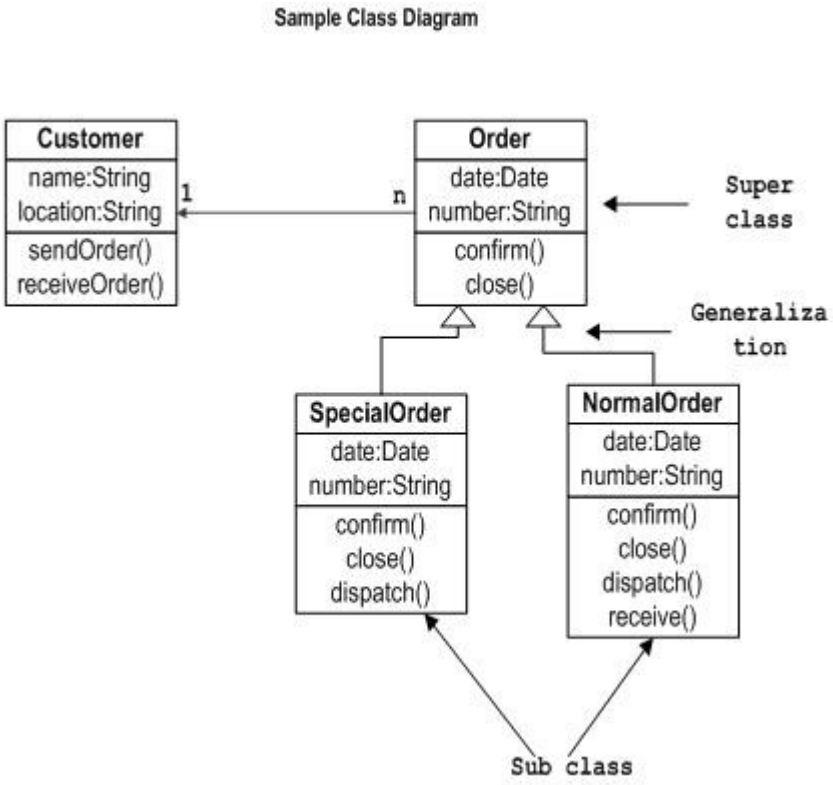


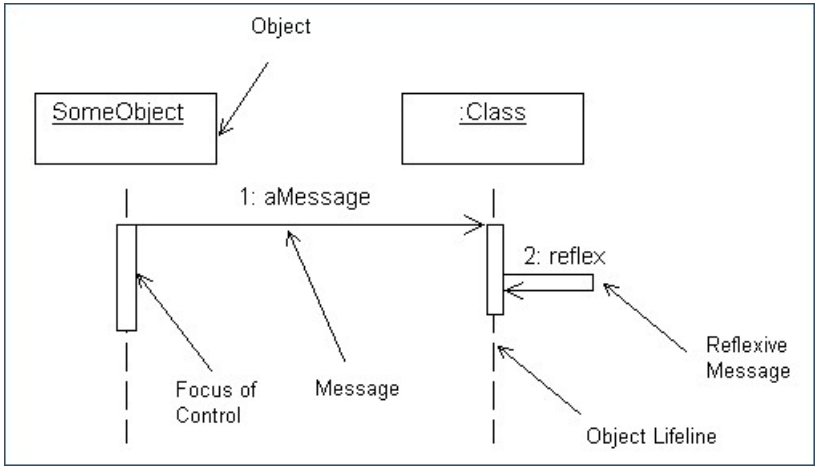


```
context DataLayer::write()
pre something to write
post make data persistent
```

UML Elements for Design

- Classes, attributes, associations, operations, sequence diagrams





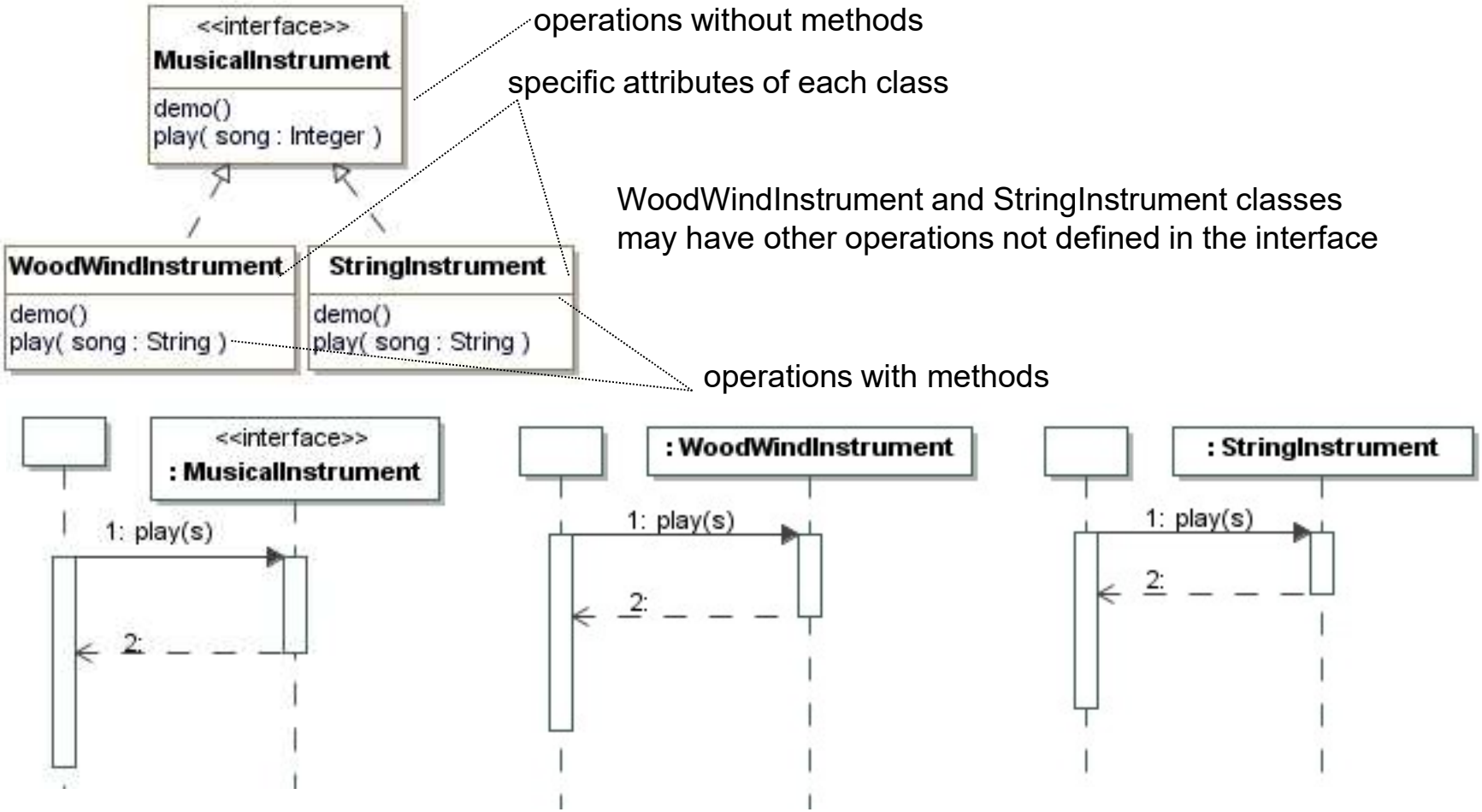
UML Elements for Design

- 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.

UML Elements for Design

- Interface Notation



UML Elements for Design

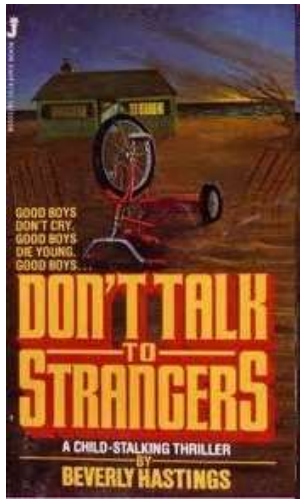
- **Interfaces** or **abstract classes**?
  - To provide common implementation to subclasses an abstract class is used.
  - To declare non-public members, use abstract classes.
  - Use abstract classes if new public methods need to be added in the future.
  - To provide the implementing classes the opportunity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

Object Oriented Architecture Design Principles: Low Coupling Principle

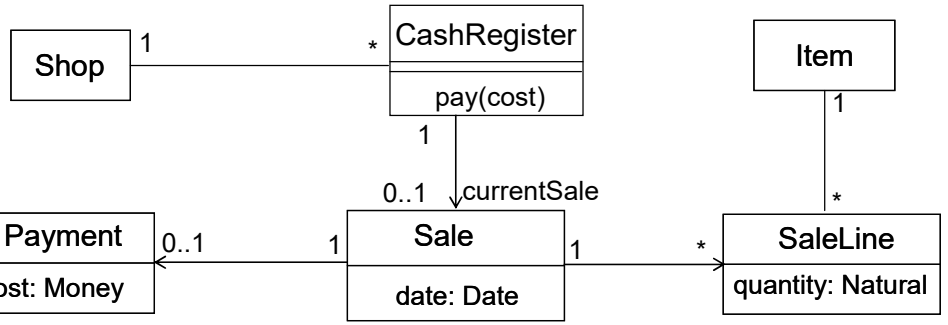
- **Coupling** is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements (*Applying UML and Patterns.* C. Larman).

- 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.

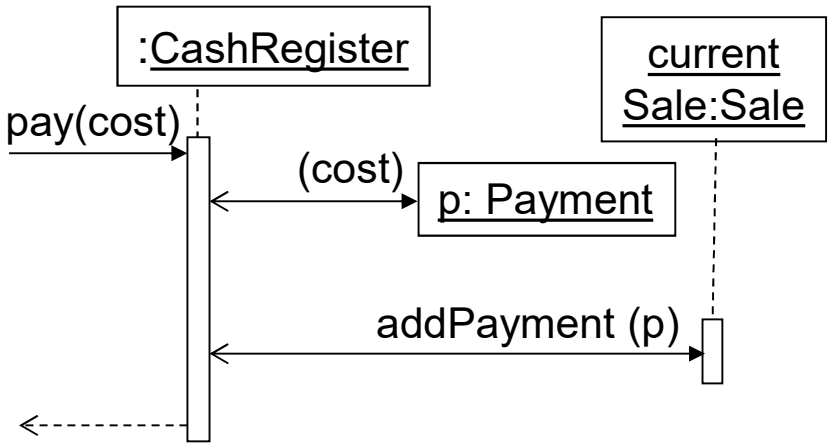
- Demeter's law ("Do not talk to strangers") helps maintaining a low coupling.



Object Oriented Architecture Design Principles: Low Coupling Principle

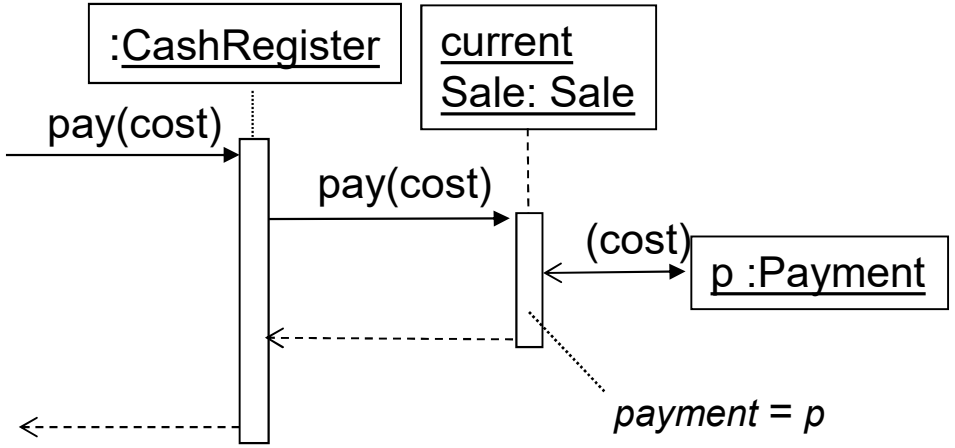


High Coupling Violation



This solution generates a new coupling from CashRegister to Payment

Low Coupling Satisfaction



This solution does not generate a new coupling.

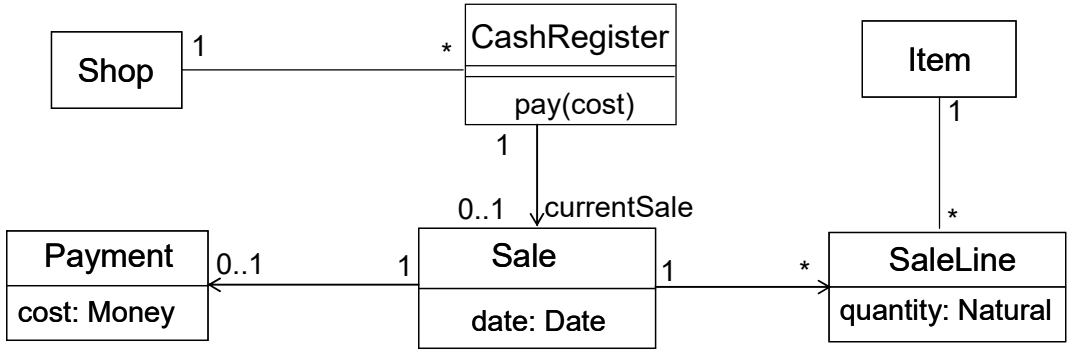
*Example extracted from *Applying UML and Patterns*. C. Larman

Object Oriented Architecture Design Principles: High Cohesion Principle

- **Cohesion** is a measure of how strongly related and focused the responsibilities of an element are (*Applying UML and Patterns.* C. Larman).

- 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.

Object Oriented Architecture Design Principles: High Cohesion Principle



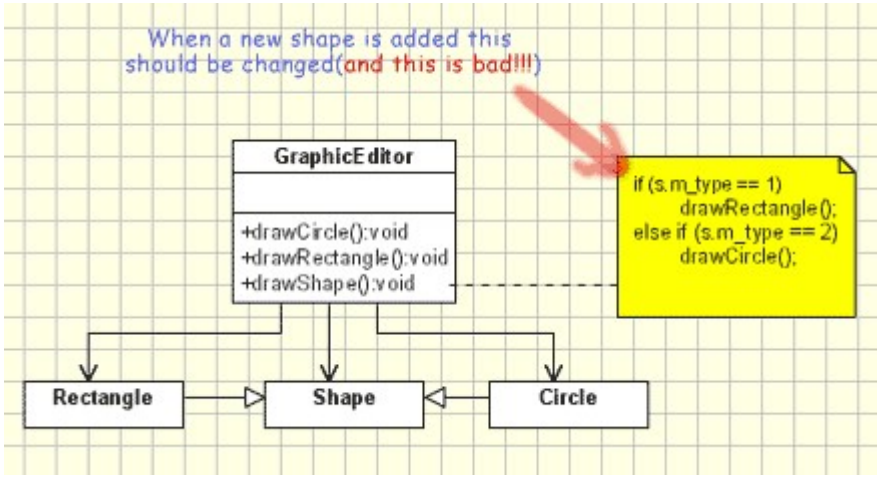
High Cohesion Violation

High Cohesion Satisfaction

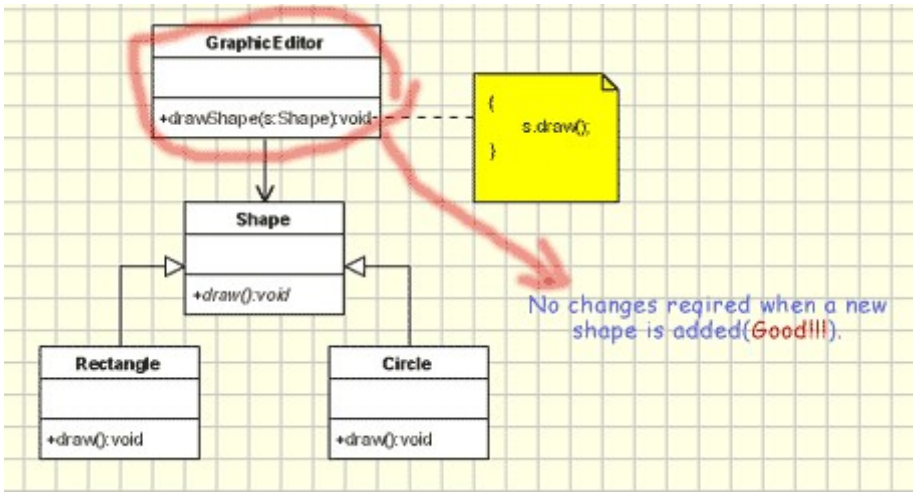
Object Oriented Architecture SOLID Design Principles: Open Closed Principle

- The Open Closed Principle (OCP): A module should be open for extension but closed for modification (*Agile Software Development: Principles, Patterns and Practices*, R.C. Martin).

OCP Violation



OCP Satisfaction

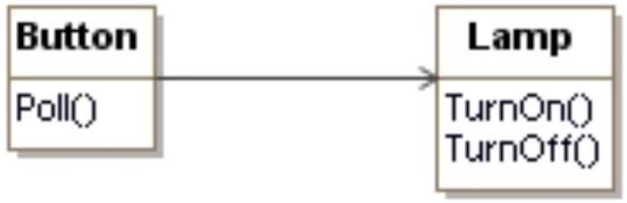


*Example extracted from http://www.oodesign.com/open-close-principle.html

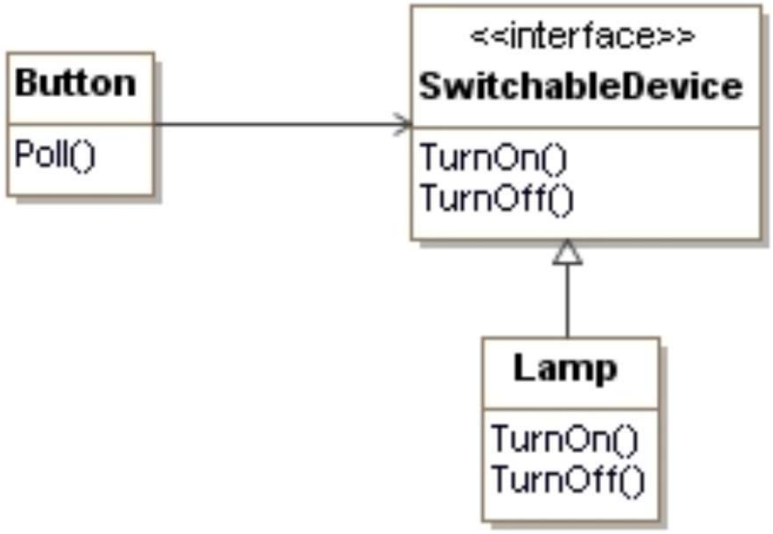
Object Oriented Architecture SOLID Design Principles: Dependency Inversion Principle

- The Dependency Inversion Principle: 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 (*Agile Software Development: Principles, Patterns and Practices*. R.C. Martin).

DIP Violation



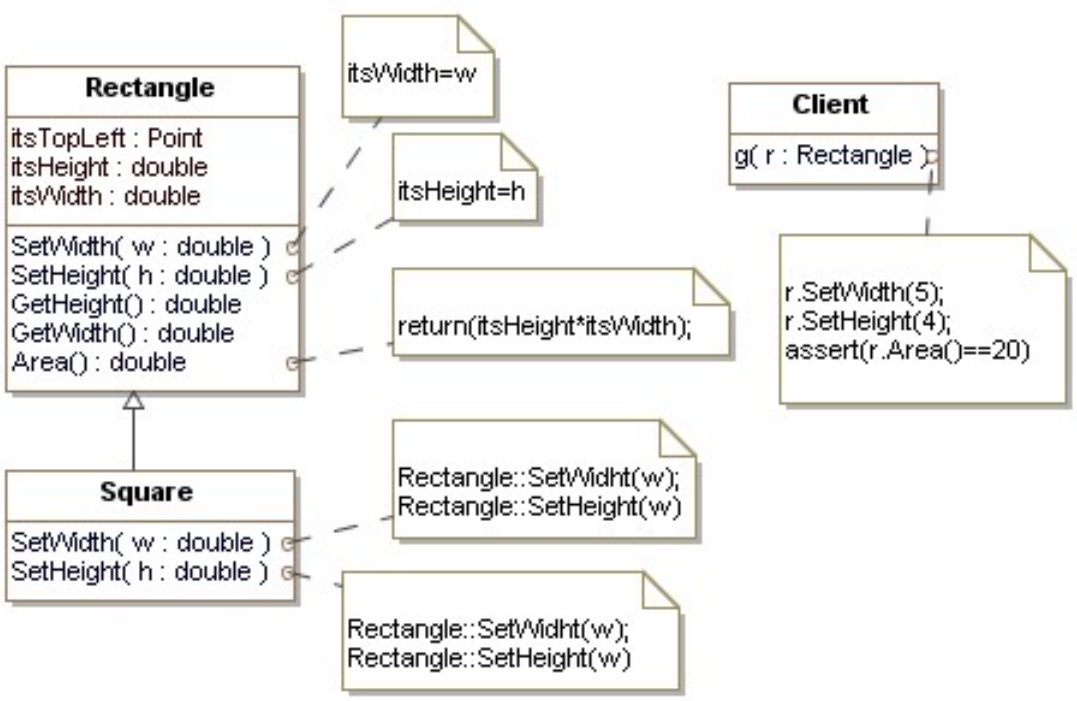
DIP Satisfaction



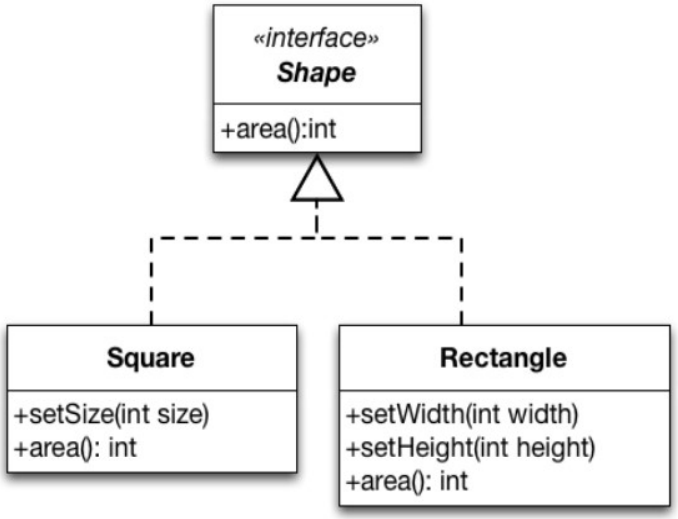
Object Oriented Architecture SOLID Design Principles: The Liskov Substitution Principle

- The Liskov Substitution Principle: Subclasses should be substitutable for their base classes (*Agile Software Development: Principles, Patterns and Practices*. R.C. Martin).

Liskov Violation



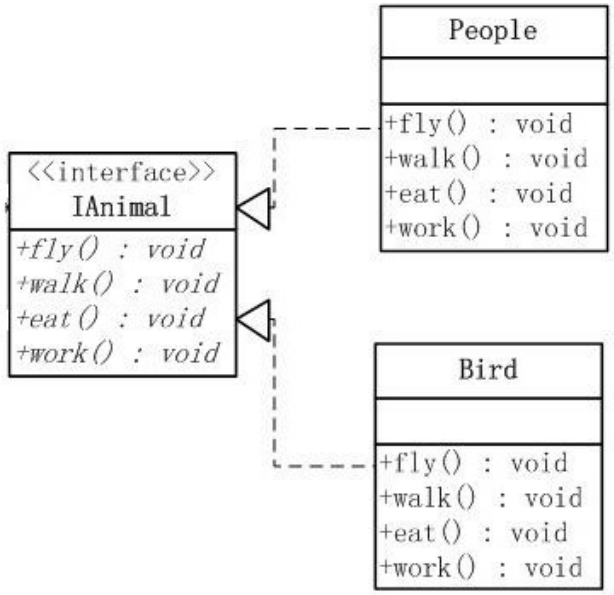
Liskov Satisfaction



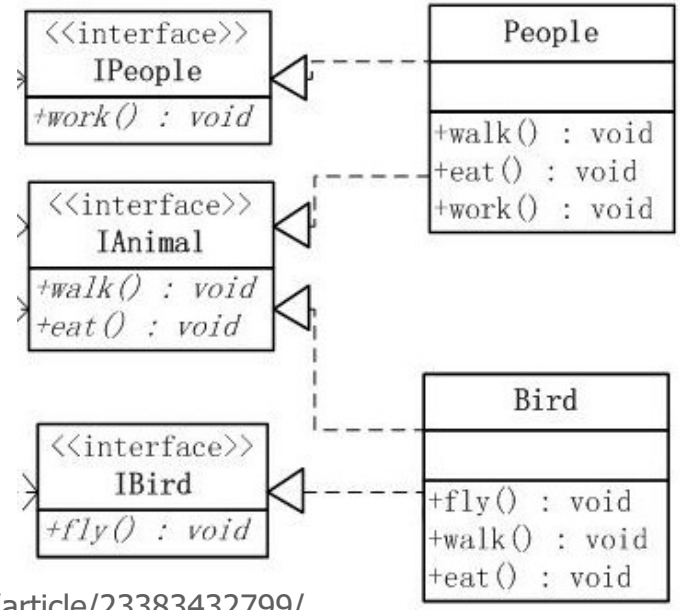
Object Oriented Architecture SOLID Design Principles: Interface Segregation Principle

- The Interface Segregation Principle (ISP): Clients should not be forced to depend on methods that they do not use (*Agile Software Development: Principles, Patterns and Practices*. R.C. Martin).

ISP Violation



ISP Satisfaction



*Example extracted from https://www.programmersought.com/article/23383432799/

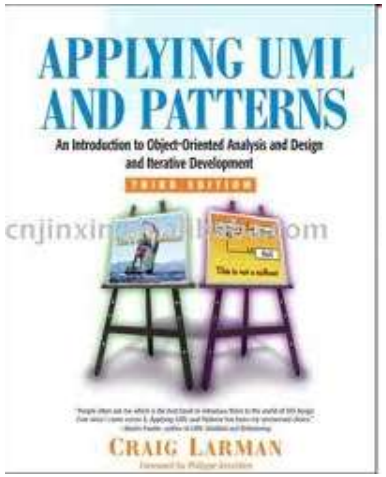
Object Oriented Architecture SOLID Design Principles: Single Responsibility Principle

- The Single Responsibility Principle: a class should have only one reason to change. (*Agile Software Development: Principles, Patterns and Practices*. R.C. Martin). Related with the cohesion principle

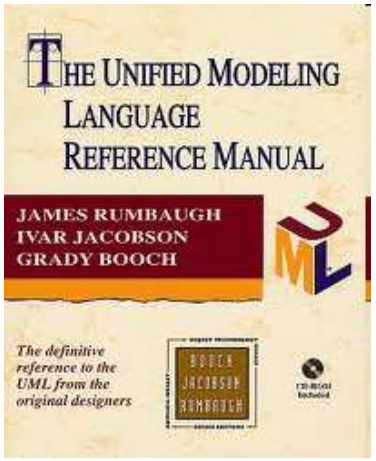
SRP Violation

SRPSatisfaction

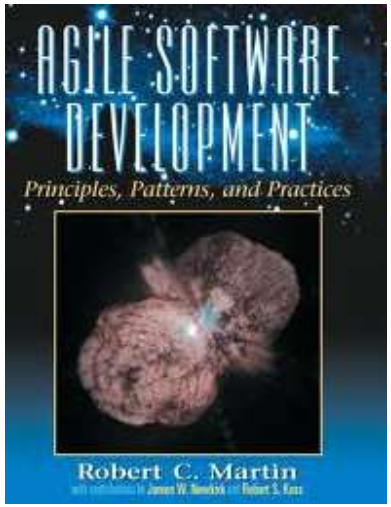
References



Chapters 16, 17 and 32



Pages 413-418



http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

References

- *Applying UML and Patterns*
  C. Larman
  Prentice Hall, 2005 (Third edition), ch. 16, 17 and 32

- *The Unified Modeling Language Reference Manual.*
  J. Rumbaugh, I. Jacobson, G. Booch.
  Addison-Wesley, 2004, pp. 413-418

- *Agile Software Development: Principles, Patterns and Practices*
  R.C. Martin
  Prentice Hall, 2003

- Design principles and design patterns
  R.C. Martin
  http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf