# SOLID Principles

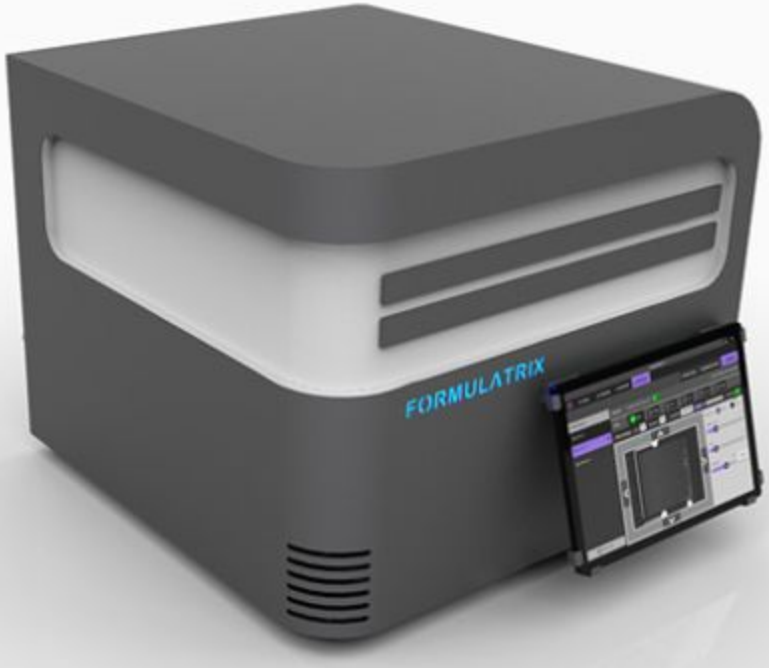
By Irwansyah :)
A C Developer wannabe

# Your Expectations?

# Why SOLID?

# CONSTELLATION® Digital PCR System

*Mainstream Digital PCR*



# Constellation Challenges

1. The hardware (mechanical and electronics) not existed, yet
2. The computer vision functionality not existed, yet
3. No prior experience in Robotics
4. Computer vision functionalities not functioning properly when integrated

SOLID HELPS A LOT!!!!

irwansyahwii Initial version — Latest commit 7dc9d02 on Nov 15, 2013

| | | |
|---|---|---|
| .. | | |
| ApplicationStates.cs | Initial version | 4 years ago |
| IActiveObject.cs | Initial version | 4 years ago |
| IApplication.cs | Initial version | 4 years ago |
| IApplicationComposer.cs | Initial version | 4 years ago |
| IApplicationState.cs | Initial version | 4 years ago |
| IContinousActiveObject.cs | Initial version | 4 years ago |
| IFileSystem.cs | Initial version | 4 years ago |
| IFullMethodName.cs | Initial version | 4 years ago |
| IMainWindow.cs | Initial version | 4 years ago |
| IMapBetweenTwoIntervals.cs | Initial version | 4 years ago |
| IMessageBox.cs | Initial version | 4 years ago |
| IOSVersionInfo.cs | Initial version | 4 years ago |
| IResourceFile.cs | Initial version | 4 years ago |
| ISplashScreen.cs | Initial version | 4 years ago |
| ISystemLog.cs | Initial version | 4 years ago |
| ITimer.cs | Initial version | 4 years ago |
| IUShortToBitArray.cs | Initial version | 4 years ago |
| IXmlDeserializer.cs | Initial version | 4 years ago |
| IXmlSerializer.cs | Initial version | 4 years ago |

# IrwanCordova - SpecFlow

irwansyahwii Initial version 9e73fb3 on Nov 15, 2013

1 contributor

27 lines (21 sloc) | 1.25 KB Raw Blame History

```
Feature: Device Module

@mytag
Scenario: Machine Id key is not exists and a newly generated MachineId will be stored in the registry
        Given This GUID "C62A91D0-482A-449B-B955-D7911FEB5A8F"
        And Machine id key is not exists
        When Device.AcquireUniqueId() is called
        Then A new key with name MachineId must be created in registry \CurrentUser\Software\Intel\Cordova
        And it's value must be set to "C62A91D0-482A-449B-B955-D7911FEB5A8F"
        And The Device.UniqueId property value must be the same as "C62A91D0-482A-449B-B955-D7911FEB5A8F"

@mytag
Scenario: Machine id key is already exists, don't overwrite the existing MachineID
        Given This GUID "2B17C4B0-4999-4CFE-AB3B-C7882D4395B6"
        And MachineID key is exists with this GUID "6DEA3BB5-DBCF-4497-899C-61DED0011606"
        When Device.AcquireUniqueId() is called
        Then Existing MachineID must be the same as "6DEA3BB5-DBCF-4497-899C-61DED0011606"
        And The Device.UniqueId property value must be the same as "6DEA3BB5-DBCF-4497-899C-61DED0011606"


Scenario: Passing action "getDeviceInfo" the matching method must be called
        Given This action "getDeviceInfo" and callback id "1"
        When Device.Execute is called
        Then Device.GetDeviceInfo() must be called with callback id "1"
        And Success callback is called with the correct result

```

# PREREQUISITES

Because diamond is forever

# Tools and Techniques for Software Development

1. Structured Programming

Developing application software that breaks large problems into smaller, simpler ones, and provides an integrative structure

2. Object-Oriented Programming

Development of a collection of existing modules of code or objects that can be used across any number of applications without being rewritten

http://facpub.stjohns.edu/~wolfem/4322/chapter13/sld028.htm

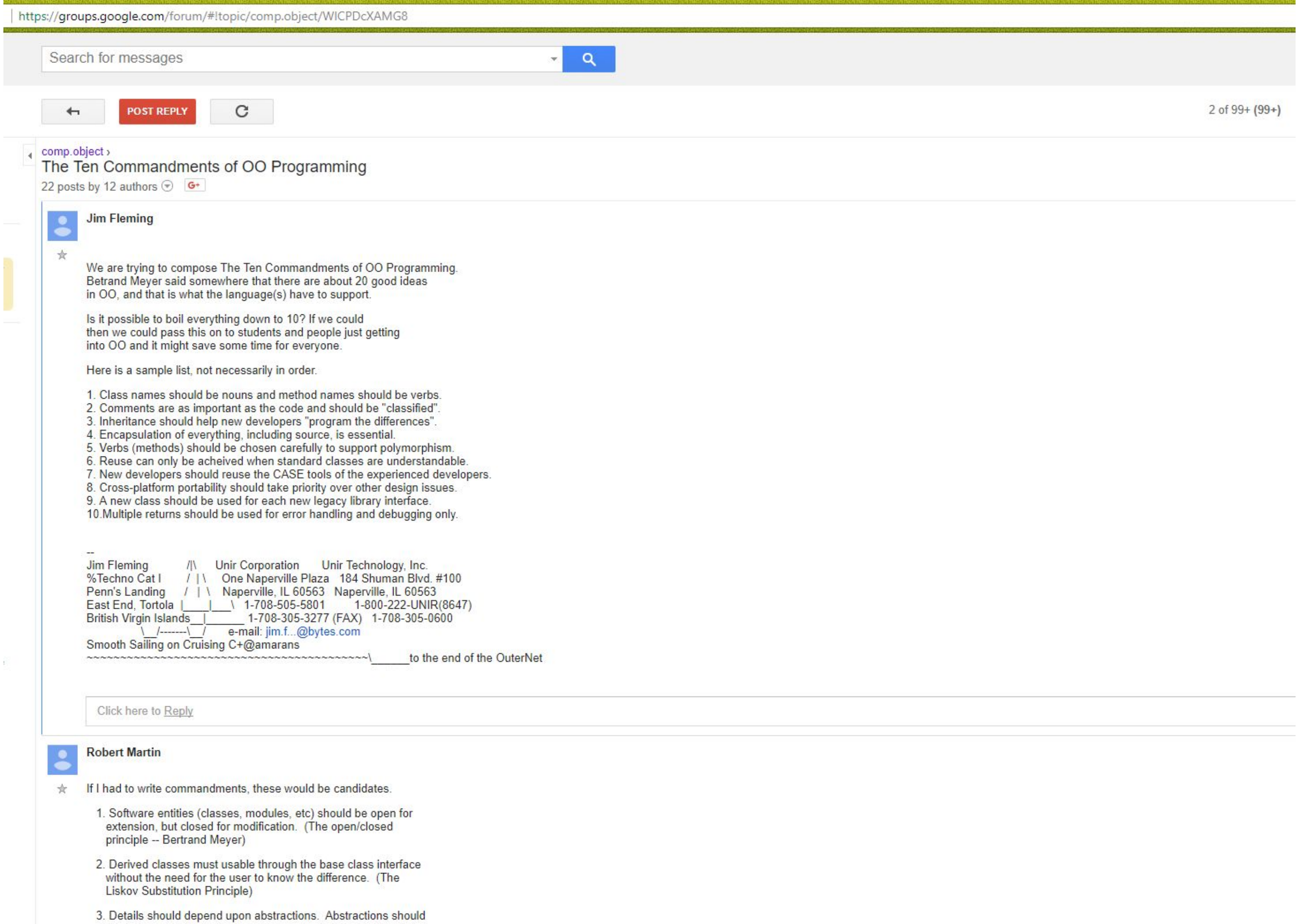
# Structured Programming Characteristics

1. Program code is broken into modules
2. Each module has one and only one function. Such modules are said to have tight internal cohesion
3. The modules are loosely coupled
4. GOTO statements are not allowed

# Basic OOP Concepts

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction
5. Interface/Abstract Class/Pure Virtual Functions → HIGH REUSABILITY!!!
   (Deep understanding of SCRUM)
6. Virtual Method Table
7. Method Overloading
8. Objects and Classes

# SOLID History



comp.object ›

The Ten Commandments of OO Programming

22 posts by 12 authors

**Jim Fleming**

We are trying to compose The Ten Commandments of OO Programming.
Betrand Meyer said somewhere that there are about 20 good ideas
in OO, and that is what the language(s) have to support.

Is it possible to boil everything down to 10? If we could
then we could pass this on to students and people just getting
into OO and it might save some time for everyone.

Here is a sample list, not necessarily in order.

1. Class names should be nouns and method names should be verbs.
2. Comments are as important as the code and should be "classified".
3. Inheritance should help new developers "program the differences".
4. Encapsulation of everything, including source, is essential.
5. Verbs (methods) should be chosen carefully to support polymorphism.
6. Reuse can only be acheived when standard classes are understandable.
7. New developers should reuse the CASE tools of the experienced developers.
8. Cross-platform portability should take priority over other design issues.
9. A new class should be used for each new legacy library interface.
10.Multiple returns should be used for error handling and debugging only.

```
--
Jim Fleming          /|\      Unir Corporation       Unir Technology, Inc.
%Techno Cat I       / | \     One Naperville Plaza   184 Shuman Blvd. #100
Penn's Landing     /  |  \    Naperville, IL 60563   Naperville, IL 60563
East End, Tortola |____|___\  1-708-505-5801         1-800-222-UNIR(8647)
British Virgin Islands__|______ 1-708-305-3277 (FAX)   1-708-305-0600
                 \__/-------\__/       e-mail: jim.f...@bytes.com
Smooth Sailing on Cruising C+@amarans
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\______to the end of the OuterNet
```

Click here to Reply

**Robert Martin**

If I had to write commandments, these would be candidates.

1. Software entities (classes, modules, etc) should be open for
   extension, but closed for modification. (The open/closed
   principle -- Bertrand Meyer)

2. Derived classes must usable through the base class interface
   without the need for the user to know the difference. (The
   Liskov Substitution Principle)

3. Details should depend upon abstractions. Abstractions should

# Jim Fleming - The Ten Commandments of OO Programming

1. Class names should be nouns and method names should be verbs.
2. Comments are as important as the code and should be "classified".
3. Inheritance should help new developers "program the differences".
4. Encapsulation of everything, including source, is essential.
5. Verbs (methods) should be chosen carefully to support polymorphism.
6. Reuse can only be acheived when standard classes are understandable.
7. New developers should reuse the CASE tools of the experienced developers.
8. Cross-platform portability should take priority over other design issues.
9. A new class should be used for each new legacy library interface.
10. Multiple returns should be used for error handling and debugging only.

# Original Version of SOLID

1. Software entities (classes, modules, etc) should be open for extension, but closed for modification. (The open/closed principle -- Bertrand Meyer)
2. Derived classes must usable through the base class interface without the need for the user to know the difference. (The Liskov Substitution Principle)
3. Details should depend upon abstractions. Abstractions should not depend upon details. (Principle of Dependency Inversion)
4. The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused.
5. Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed. What affects one, affects all.

# Original Version of SOLID (Cont.)

6. Classes within a released componen should be reused together. That is, it is impossible to separate the components from each other in order to reuse less than the total.

7. The dependency structure for released components must be a DAG. There can be no cycles.

8. Dependencies between released components must run in the direction of stability.  The dependee must be more stable than the depender.

9. The more stable a released component is, the more it must consist of abstract classes.  A completely stable component should consist of nothing but abstract classes.

# Original Version of SOLID (Cont.)

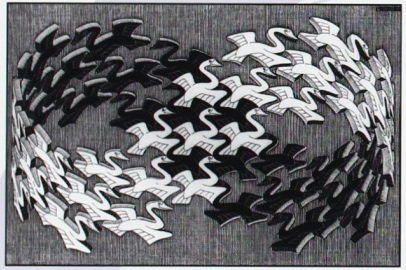
10. Where possible, use proven patterns to solve design problems.

11. When crossing between two different paradigms, build an interface layer that separates the two.  Don't pollute one side with the paradigm of the other.

# Design Patterns

## Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides





Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# What is a Design Pattern?

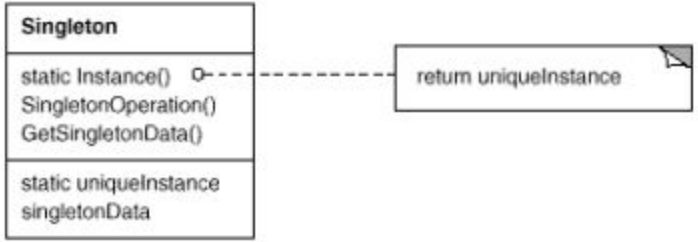
Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS+77]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context.

# Singleton

**Intent:** Ensure a class only has one instance, and provide a global point of access to it.

**Motivation:** It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company

## Structure

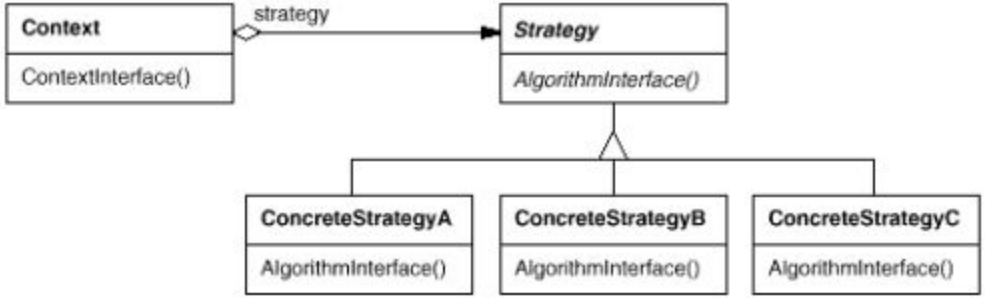

Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

# Strategy

**Intent:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**Motivation:** Robotic instruments for analyzing crystal protein needs different cameras with different models from different vendors. The camera is different but the operation is the same. So we need a pluggable camera driver to handle specific camera hardware.

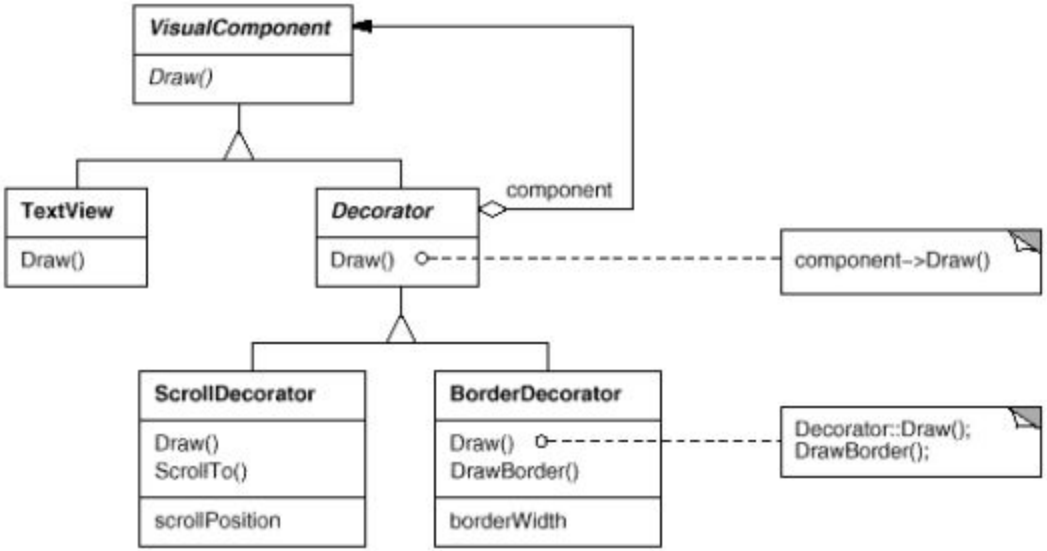
## Structure


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

# Decorator

**Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Motivation:** A repository class handles a domain model persistence mechanism and it will connect let say to a RDBMS and load the object properties. When the user grows we need to add caching mechanism. We can use decorator to decorate the existing repository class with the caching algorithm. If on another occasion we want to add logging capability we can decorate the existing object with a logging decorator.

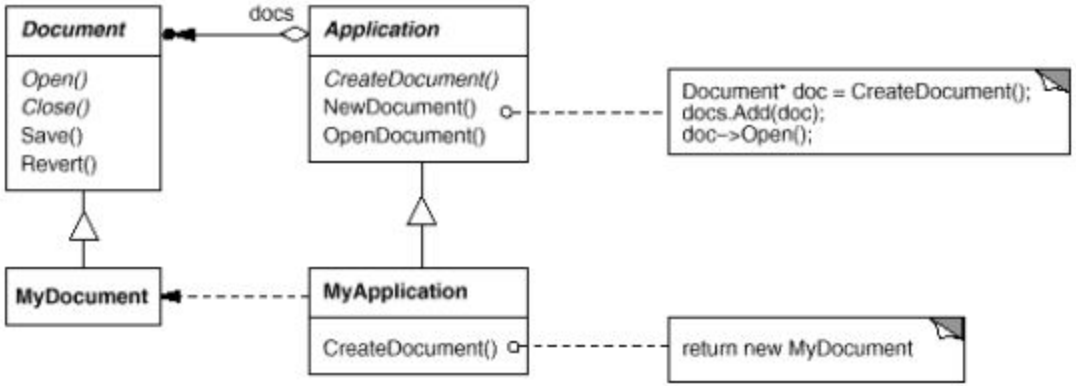

VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

# QUIZ - Water Jugs Problem

# Factory Method

**Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

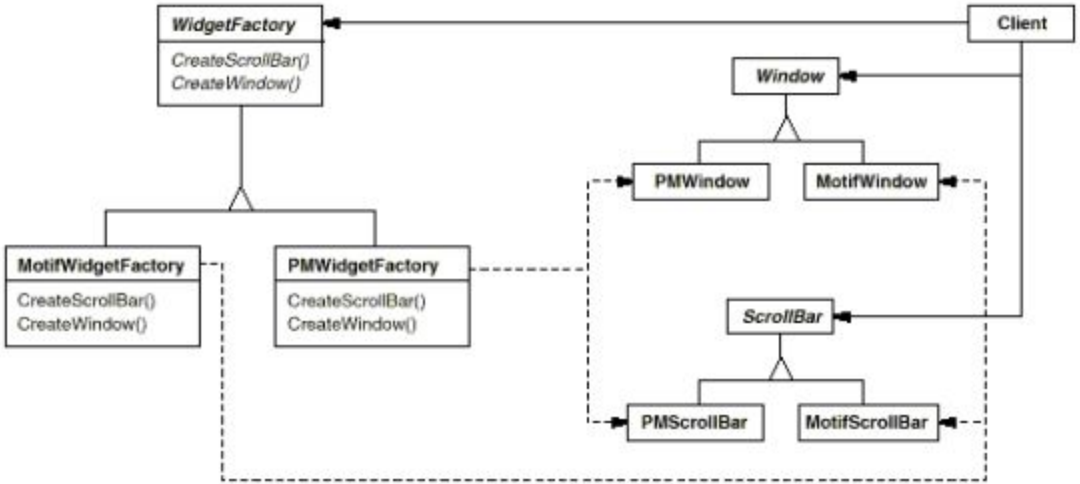
**Motivation:** MDI (Multiple Document Interface) framework


Document
Open()
Close()
Save()
Revert()
docs
Application
CreateDocument()
NewDocument()
OpenDocument()
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
MyDocument
MyApplication
CreateDocument()
return new MyDocument

# Abstract Factory

**Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

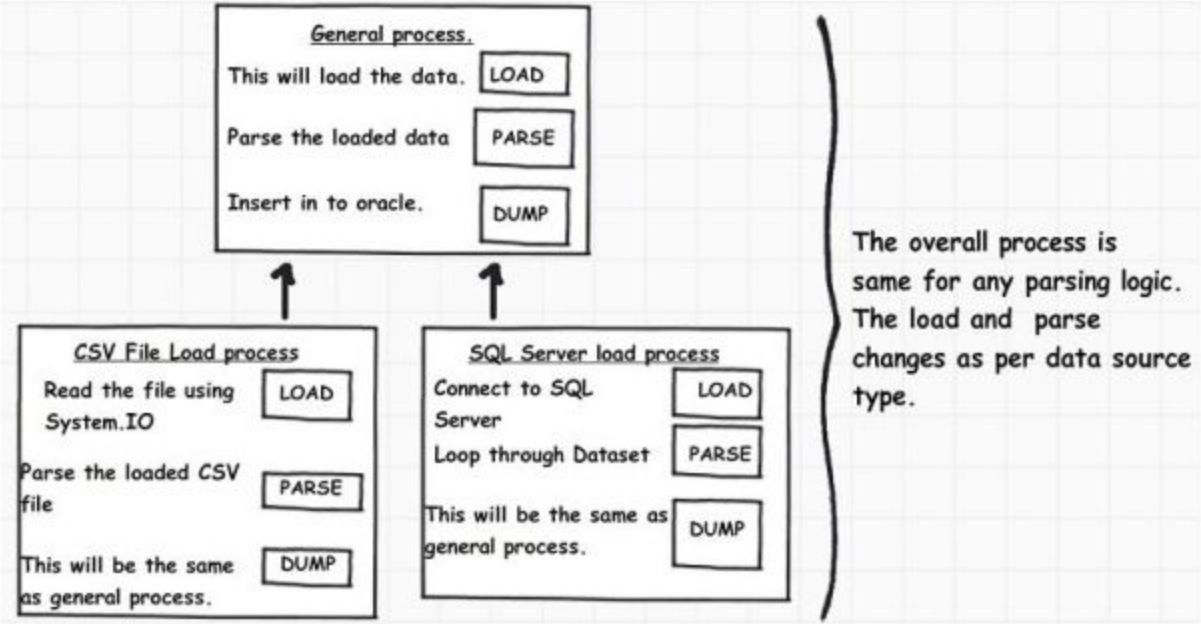
**Motivation:** When creating mobile apps, we have to target different platforms with different screen sizes. We can use abstract factory to create each screen or each UI components.


WidgetFactory
CreateScrollBar()
CreateWindow()
Client
Window
PMWindow
MotifWindow
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
ScrollBar
PMScrollBar
MotifScrollBar

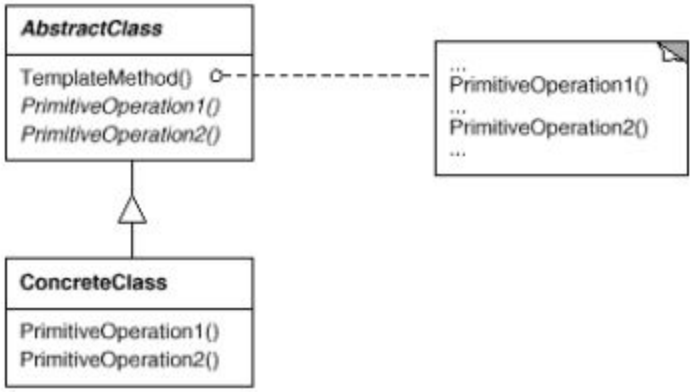
# Template Methods

**Intent:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

**Motivation:** Let say we want to create an application to migrate data to an Oracle database. This application must be able to work on various of data sources, e.g. for SQL Server data source or CSV data source. We can define the steps of the process into Load(), Parse(), Dump().


General process.
This will load the data.
LOAD
Parse the loaded data
PARSE
Insert in to oracle.
DUMP
CSV File Load process
Read the file using System.IO
LOAD
Parse the loaded CSV file
PARSE
This will be the same as general process.
DUMP
SQL Server load process
Connect to SQL Server
LOAD
Loop through Dataset
PARSE
This will be the same as general process.
DUMP
The overall process is same for any parsing logic. The load and parse changes as per data source type.

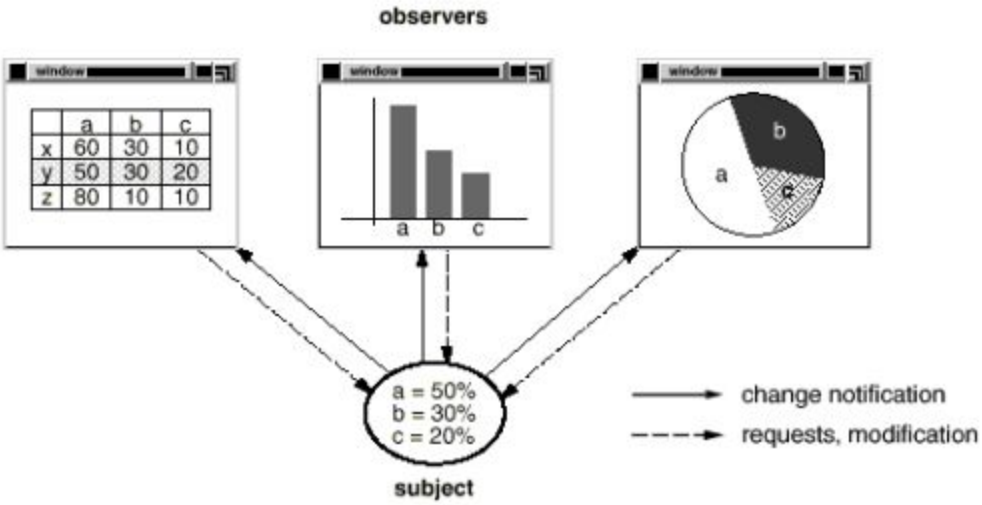
## Structure


AbstractClass
TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()

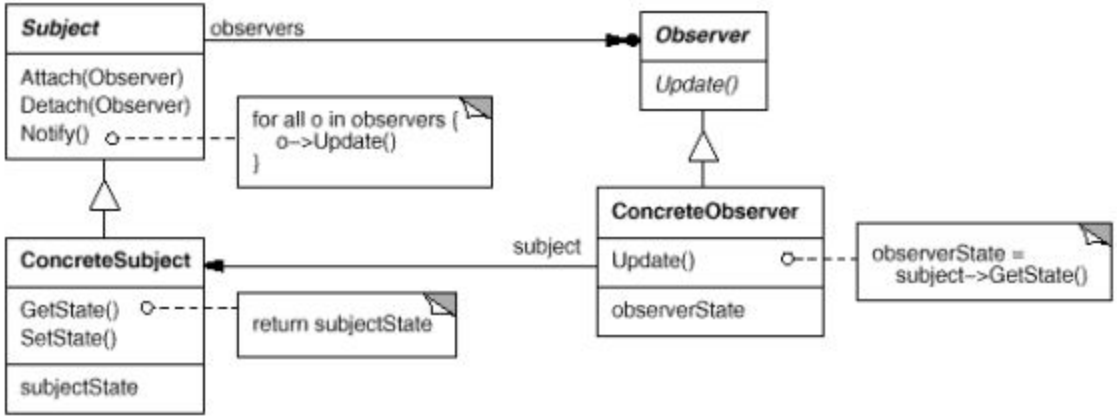
# Observer

**Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Motivation:** We want to notify other objects. We can use events or write our own implementation of Observer.


observers
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
a
b
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

## Structure


Subject
observers
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState

# SOLID Principles (5 Yin)

1. A class should have only one reason to change.
2. Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
3. Subtypes must be substitutable for their base types.
4. High-level modules should not depend on low-level modules. Both should depend on abstractions.
5. Abstractions should not depend upon details. Details should depend upon abstractions.
6. Clients should not be forced to depend on methods they do not use.

# Tools (5 Yang)

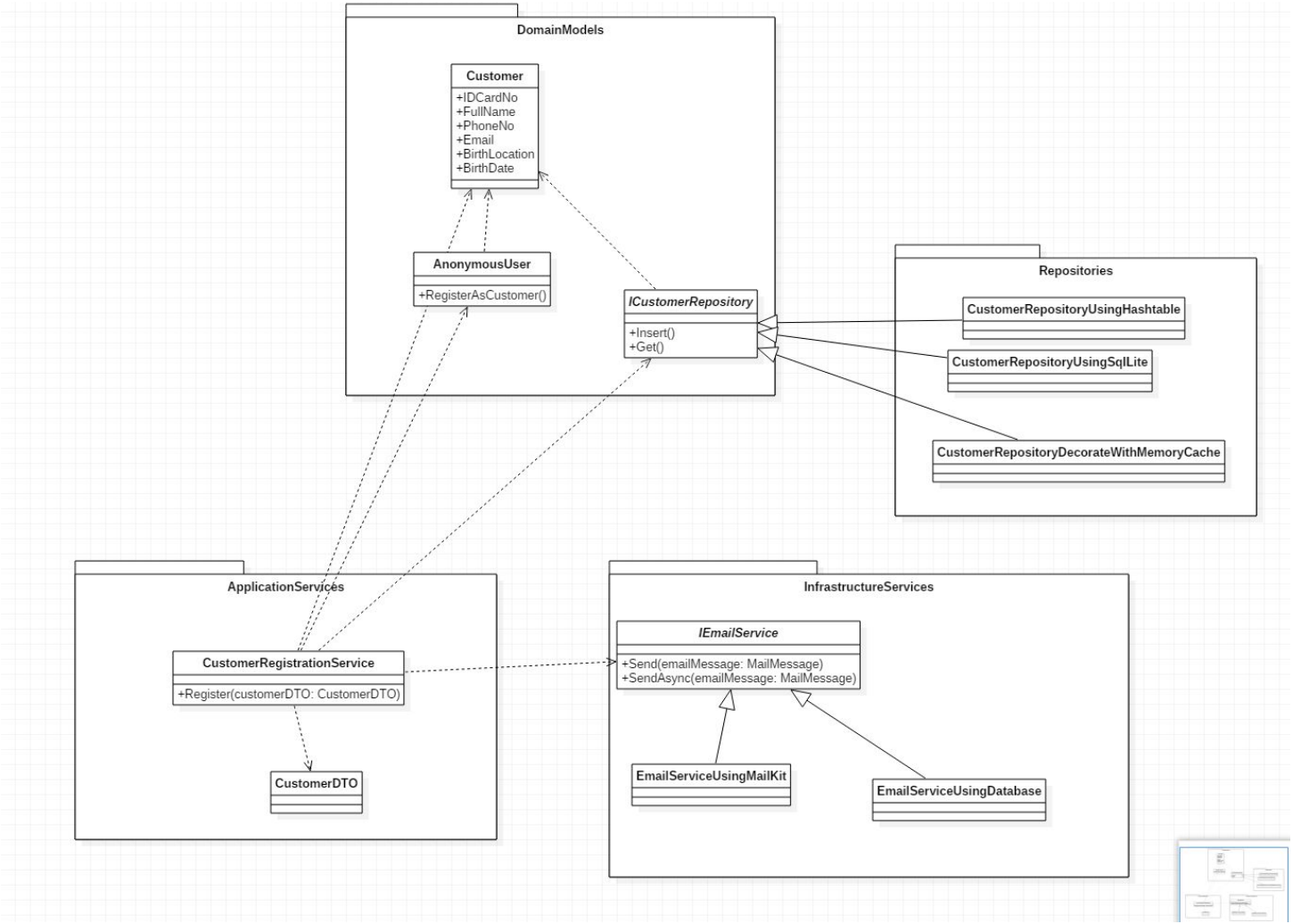
1. Strong Typing
2. Static Typing
3. Object Oriented
4. Dependency Injection Tools
5. Mocking Framework

5 Yin + 5 Yang = Magic Hand

A Force of 1,000 Pounds can be deflected with a Force of Four Ounces.

# SOLID Workshop

(A customer can register then receiving a registration notification email. We have to be able to view registered customers.)


DomainModels
Customer
+IDCardNo
+FullName
+PhoneNo
+Email
+BirthLocation
+BirthDate
AnonymousUser
+RegisterAsCustomer()
ICustomerRepository
+Insert()
+Get()
Repositories
CustomerRepositoryUsingHashtable
CustomerRepositoryUsingSqlLite
CustomerRepositoryDecorateWithMemoryCache
ApplicationServices
CustomerRegistrationService
+Register(customerDTO: CustomerDTO)
CustomerDTO
InfrastructureServices
IEmailService
+Send(emailMessage: MailMessage)
+SendAsync(emailMessage: MailMessage)
EmailServiceUsingMailKit
EmailServiceUsingDatabase

# How to know if your design is “correct”?

# How to know if your design is correct?

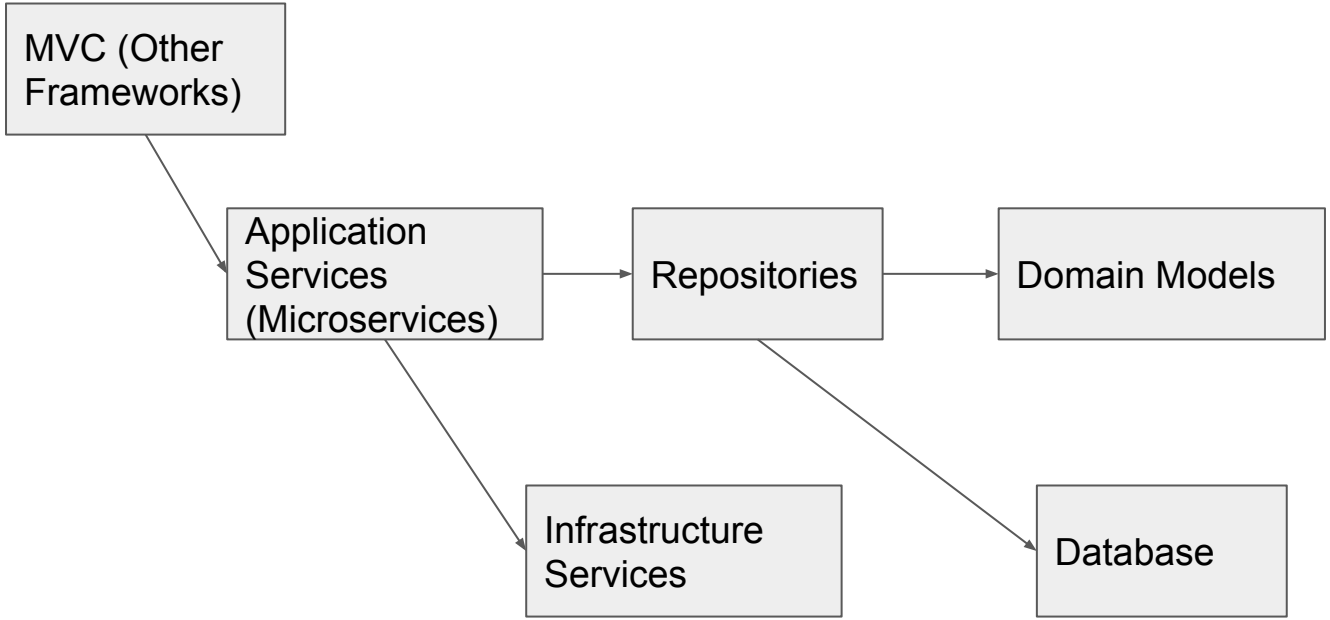
1. It is very easy to take out your defecting code and test it in isolation.
2. It is very easy to replace your code with a new one

IF YOU DON'T WAKE UP IN THE MORNING EXCITED TO PICK UP WHERE YOU LEFT YOUR WORK YESTERDAY,

YOU HAVEN'T FOUND YOUR CALLING YET.

@awesome_on_purpose

# The Architecture with DDD



# The Process

1. Start with activity diagram
2. Analyze using ERD and Class Diagram (DDD)
3. Code using Domain Model Pattern
4. Write Automated Tests (Unit Tests/Integration Tests)

# Workshop

- Food ordering system
- Online shop
- Hospital Information System
- Etc

# Kaizen/Retrospective

PENUTUP

https://www.facebook.com/andika.alramadhan/videos/1020979644529860 9/