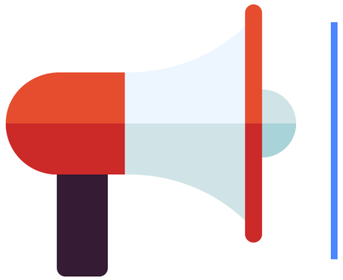
DePaul University

Design Principles: S.O.L.I.D.
Design Patterns: Intro

Object-oriented Software Development
SE 350– Spring 2021

Vahid Alizadeh

Week 7.1
May 11, 2021



Announcements

Future Schedule

Assignment 2 is graded

Midterm is graded

- ~~**Assignment 1**~~
- ~~**Assignment 2**~~
- ~~**Mid Term Exam**~~
- **Assignment 3:**
  - Release: Week 7 (TODAY)
  - Due: Week 8
- **Assignment 4:**
  - Release: Week 8
  - Due: Week 9
- **Bonus Research Project:**
  - Presentation Due: Week 10
  - Report Due: Week 11
- **Final Exam:**
  - Week 11

Assignment 3

Due: May 20, 2021

SE 350: OO Software Development

Assignment 3: Design Principles and Design Patterns

Instructor: Vahid Alizadeh

Email: v.alizadeh@depaul.edu

Quarter: Spring 2021

DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

Last update: May 11, 2021

UML Class Diagram Q&A Session

Wednesday May 12, 2021

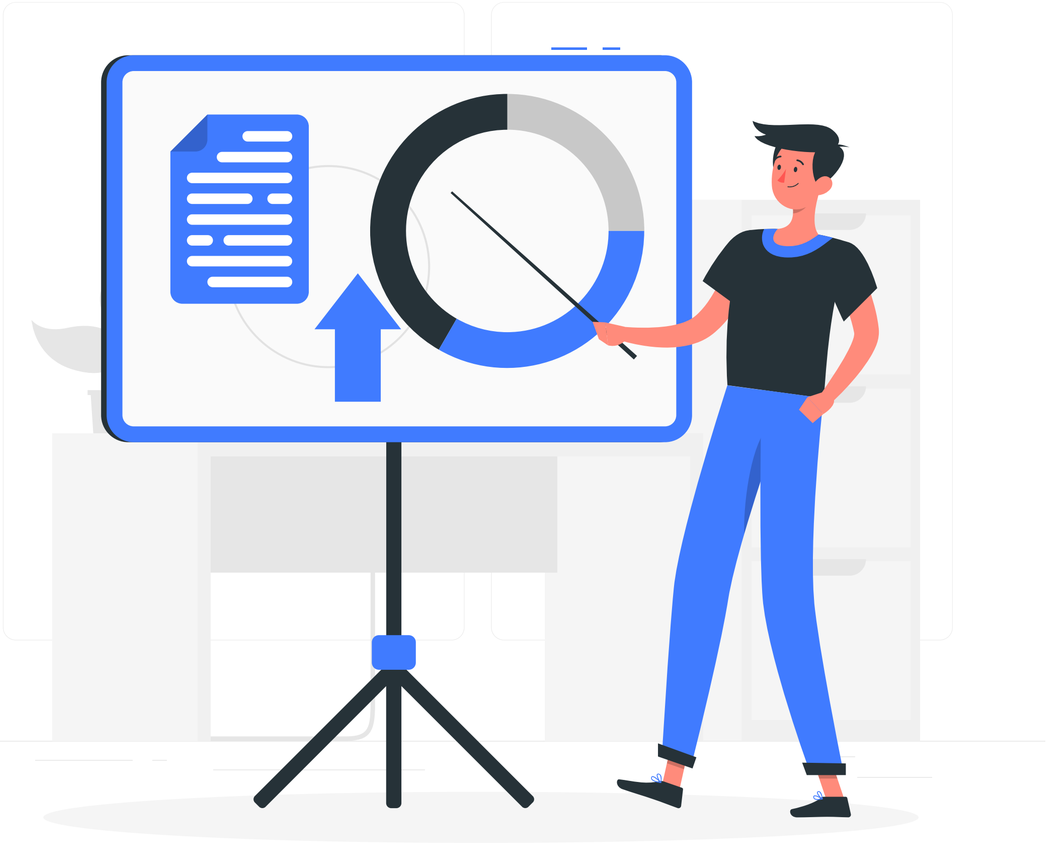
3:00 PM – 4:30 PM

Bonus Credits: Research Paper & Presentation

- **Research Presentation**
  - **Due** Week 10
  - Max 10 slides - ~7 min talk
  - **Template**: your choice!
- **Research Report**
  - **Due** Week 11
  - Writing requirement 3-4 pages
  - At least 2 external references
    - (Conference paper, articles, journals, books)
  - **Template**: ACM Proceedings (Link)
    - LaTex or Word Template (Also uploaded on D2L)
    - Overleaf Latex template (Link)
- **Research on**
  - Object-oriented programming related concepts/principles
  - Design Patterns topics.
- **Select and Announce Your Topic On:**
  - MS Teams: Research Project Channel

Confirm your project and topic by:

May 16, 2021



Bonus Credits: Research Topics and Resources

- **Some resources to find research articles and books:**
  - Google Scholar
  - Scopus
  - IEEEXplore
  - DePaul Library
- **A great resource to find related topics/resources:**
  - SemanticScholar

- **Some Topics:**
  - Design Patterns
    - **NO** Singleton, Abstract Factory, Builder, Factory method, Decorator, Adapter, Proxy
  - Object oriented metrics: QMOOD, MOOD, C&K, …
  - Design Antipatterns
  - Impact of Design Principles and Patterns
    - Pros and Cons

- **Some papers:**
  - Olague, Hector M., et al. "**Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes.**" IEEE Transactions on software Engineering 33.6 (2007): 402-419.
  - Maurer, S.. "**Design Patterns Explained A New Perspective On Object Oriented Design**." (2016).
  - Subburaj, R. et al. "**Impact of Object Oriented Design Patterns on Software Development**." (2015).
  - Dong, J. et al. "**A Review of Design Pattern Mining Techniques**." Int. J. Softw. Eng. Knowl. Eng. 19 (2009): 823-855.
  - Jiang, S. and Huaxin Mu. "**Design patterns in object oriented analysis and design**." 2011 IEEE 2nd International Conference on Software Engineering and Service Science (2011): 326-329.
  - Din, Jamilah et al. "**A Review of the Antipatterns Detection Approaches in Object-Oriented Design.**" Journal of Convergence Information Technology 8 (2013): 518-527.
  - Aras, Mehmed Taha and Y. Selçuk. "**Metric and rule based automated detection of antipatterns in object-oriented software systems.**" 2016 7th International Conference on Computer Science and Information Technology (CSIT) (2016): 1-6.
  - Khomh, F.. "**Patterns and quality of object-oriented software systems**." (2010).
  - Abbes, Marwen et al. "**An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension**." 2011 15th European Conference on Software Maintenance and Reengineering (2011): 181-190.
  - Plösch, Reinhold et al. "**Measuring, Assessing and Improving Software Quality based on Object-Oriented Design Principles**." Open Computer Science 6 (2016): 187 - 207.

Sample Research Topics

Presenter	Title
1	Impact of Design Patterns (Pros & Cons)
2	Anti-patterns and their impact on programming
3	Flyweight Design Pattern (What is it, how can it be used, other patterns that can complement it)
4	Visitor pattern
5	Observer Design Pattern
6	Mediator Pattern
7	MVC Design Pattern
8	Object oriented metrics : CK metrics

Presenter	Title
9	Iterator Pattern
10	Impact of object-oriented design principles on Software quality
11	Strategy Design Pattern
12	Composite Design Pattern
13	Broker Pattern
14	Object-oriented metrics: MOOD and QMOOD
15	State Pattern
16	Template method pattern

Design Principles

SOLID

Interface Segregation Principle

- **Many client-specific interfaces are better than one general-purpose interface.**
- **Reduce the side effects of using larger interfaces by breaking application interfaces into smaller ones.**
- **Similar to SRP:**
  - Each class or interface serves a single purpose.
- **It take more time and effort in the design phase .**
- **It increase the code complexity.**
- **It leads to flexible code.**
- **Examples:**
  - logging interface for writing and reading logs – DB vs Console
  - Reportable interface: generateExcel() and generatedPdf().
  - Large Employee class:
    - EmployeeTimeLogController, EmployeeTimeOffController, EmployeeSalaryController
- **How to make sure your code follows the ISP?**

"Clients should not be forced to implement unnecessary methods which they will not use"

ISP Example: Java AWT event handlers

- **Java AWT event handlers**
  - Some of the listeners are –
    - FocusListener
    - KeyListener
    - MouseMotionListener
    - MouseWheelListener
    - TextListener
    - WindowFocusListener

```java
public class MouseMotionListenerImpl implements
MouseMotionListener
{
    @Override
    public void mouseDragged(MouseEvent e) {
        //handler code
    }

    @Override
    public void mouseMoved(MouseEvent e) {
        //handler code
    }
}
```

Dependency Inversion Principle

- **Depend on abstractions, not on concretions.**
  - Design software in such a way that various modules can be separated from each other using an abstract layer to bind them together.
- **DIP Fundamentals:**
  1) **High-level** modules should not depend on **low-level** modules. Both should depend on **abstractions**.
  2) **Abstractions** should not depend on **details**. Details should depend on abstractions.
- **Benefits:**
  - Extensible, Testable, Maintainable
- **Example: The electricity in your house and all devices that can plug in and use it.**

> **"Modules should depend upon interfaces or abstract classes, not concrete classes."**

- **High level modules**
  - bring real value
  - solve real problems and use cases
  - what the software should do
- **Low level modules**
  - are implementation details that are required to execute the business policies.
  - how the software should do various tasks
- **Abstraction**
  - not concrete
  - interfaces and abstract classes

Understanding DIP

DIP Example: Payment Database

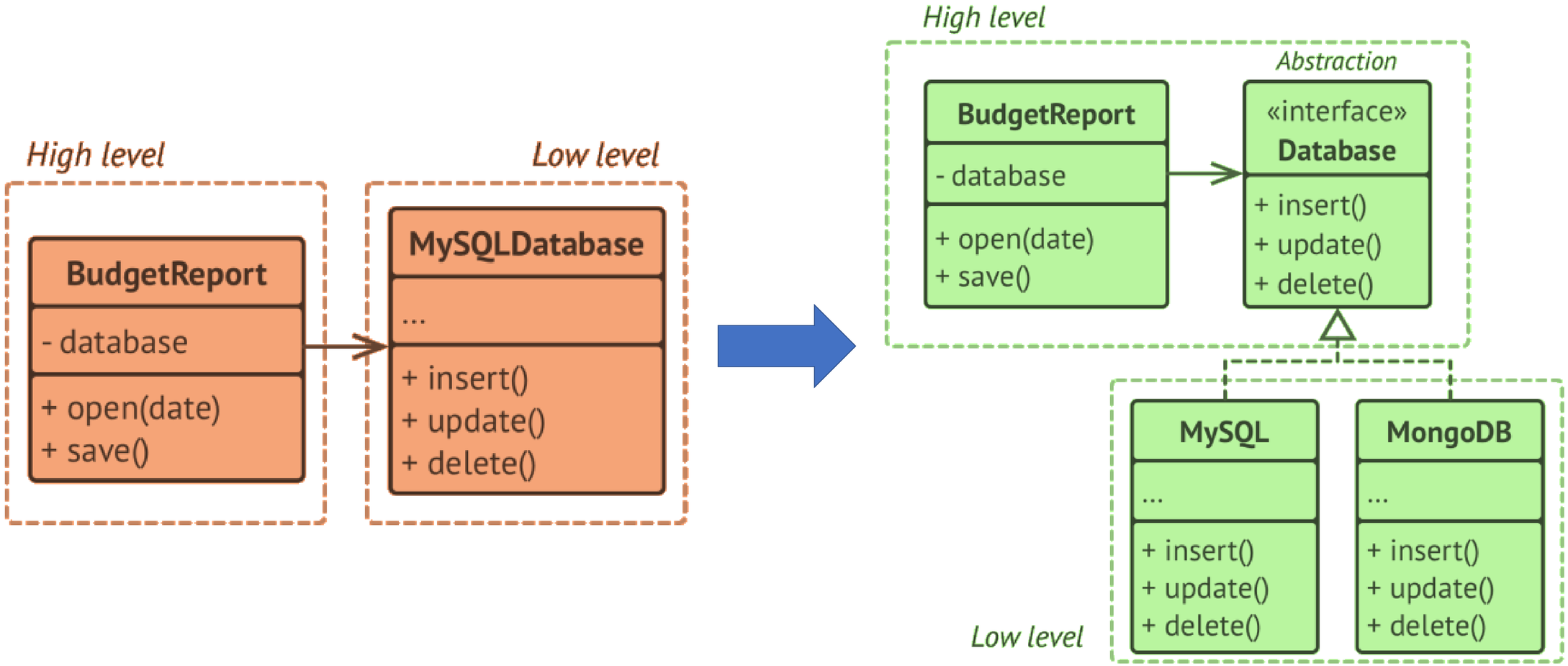
DIP Violation

```java
// low level class
// It's a concrete class that use SQL to return products
  from the database.
class SqlProductRepo {
    public Product getById(String productId) {
        // grab product from SQL database
    }
}

// High level class
class PaymentProcessor {
    public void pay(String productId) {
        SqlProductRepo repo = new SqlProductRepo();
        Product product = repo.getById(productId);
        this.processPayment(product);
    }
}
```

DIP Compliance

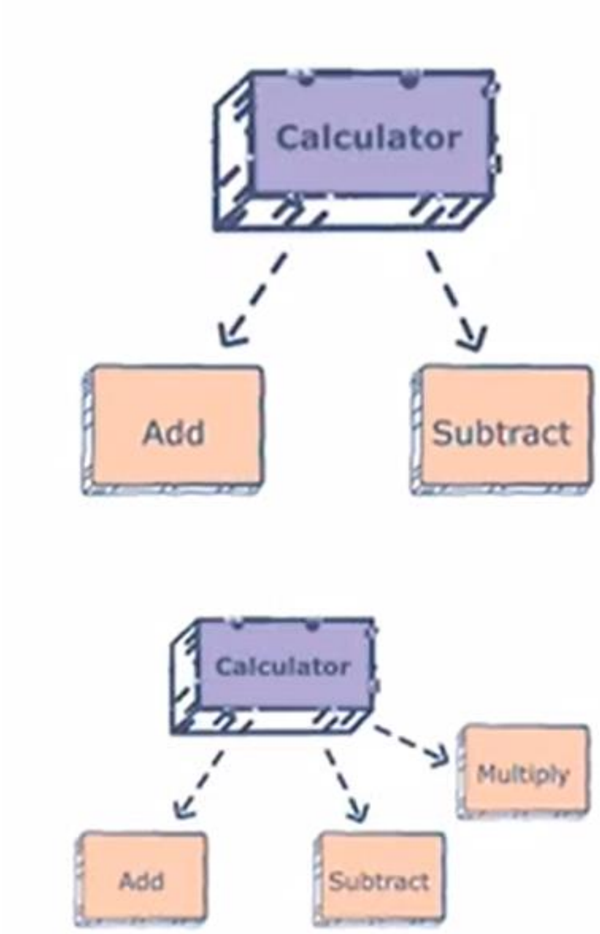
```java
interface ProductRepo {
    Product getById(String productId);
}

// low level class depends on abstraction
class SqlProductRepo implements ProductRepo {
    @Override
    public Product getById(String productId) {
        // concrete details for fetching a product
    }
}

class PaymentProcessor {
    public void pay(String productId) {
        ProductRepo repo = ProductRepoFactory.create();
        Product product = repo.getById(productId);
        this.processPayment(product);
    }
}

class ProductRepoFactory {
    public static ProductRepo create(String type) {
        if (type.equals("mongo")) {
            return new MongoProductRepo();
        }

        return new SqlProductRepo();
    }
}
```

DIP Example: Budget Report Database

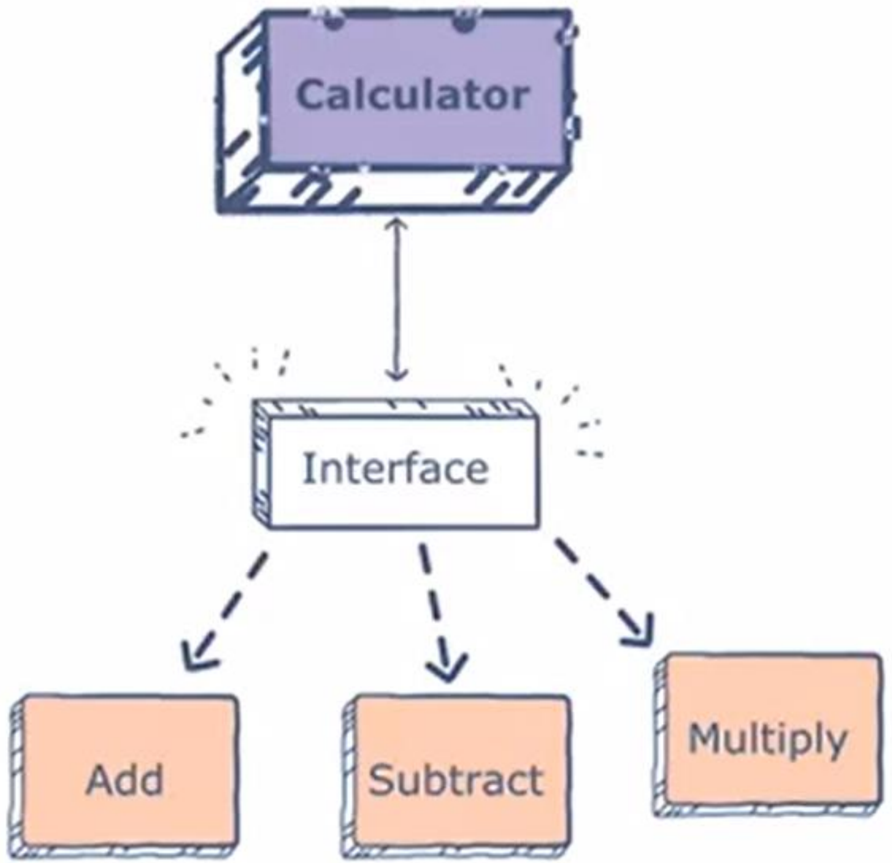


DIP Example: Calculator

DIP Violation



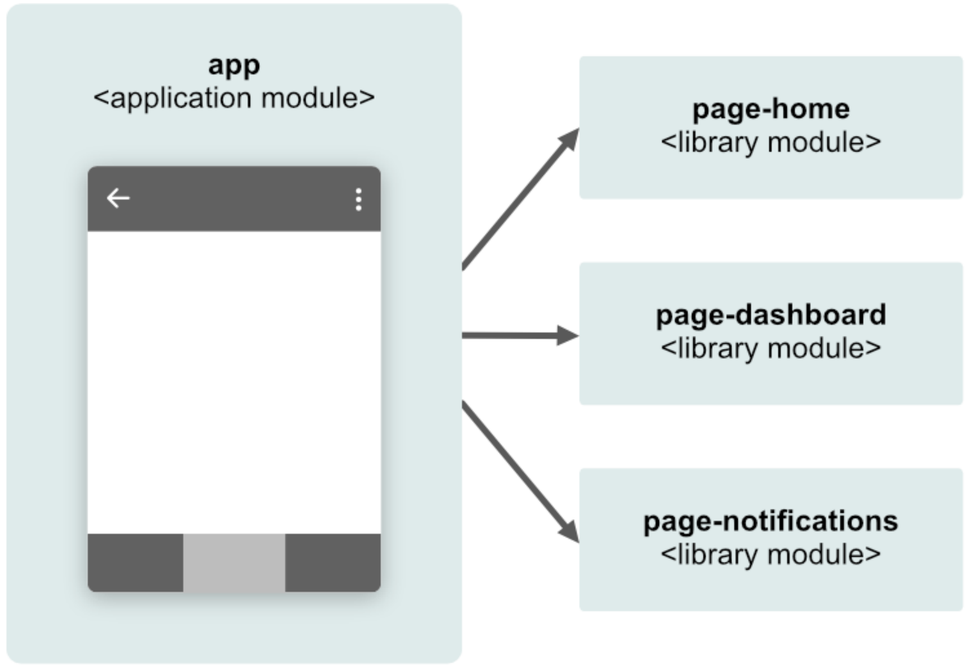
DIP Compliance



DIP Example: App Pages

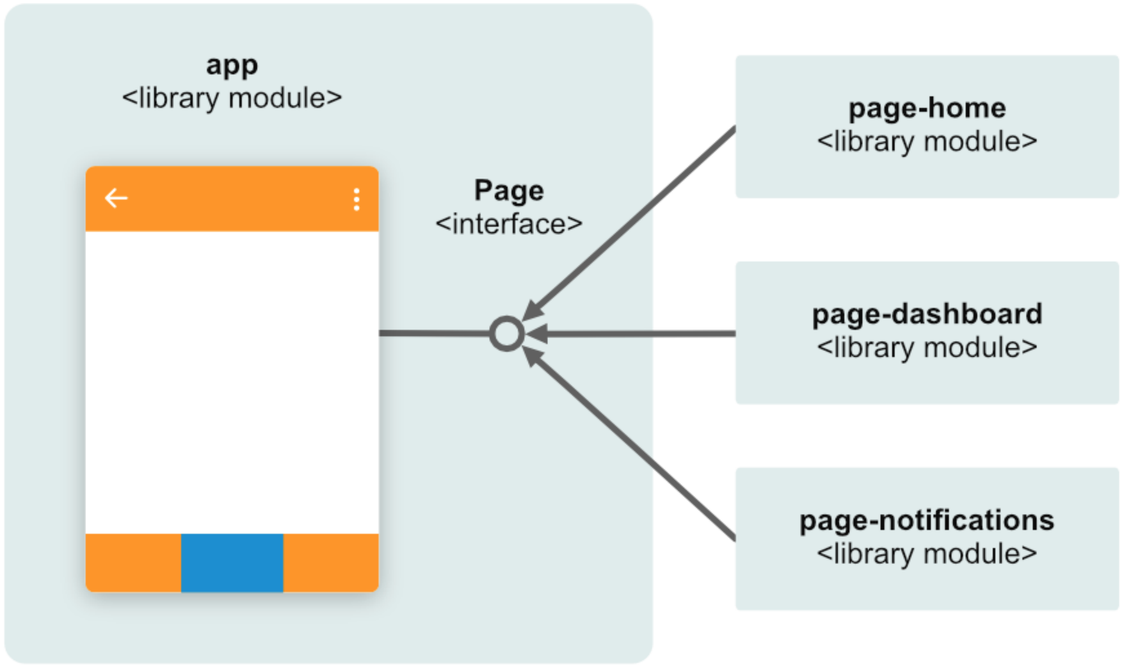
DIP Violation

Traditional layer pattern



DIP Compliance

Dependency inversion pattern



DIP Example: Windows Machine

DIP Violation

```
1  public class Windows98Machine {
2
3      private final StandardKeyboard keyboard;
4      private final Monitor monitor;
5
6      public Windows98Machine() {
7          monitor = new Monitor();
8          keyboard = new StandardKeyboard();
9      }
10
11 }
```

DIP Compliance

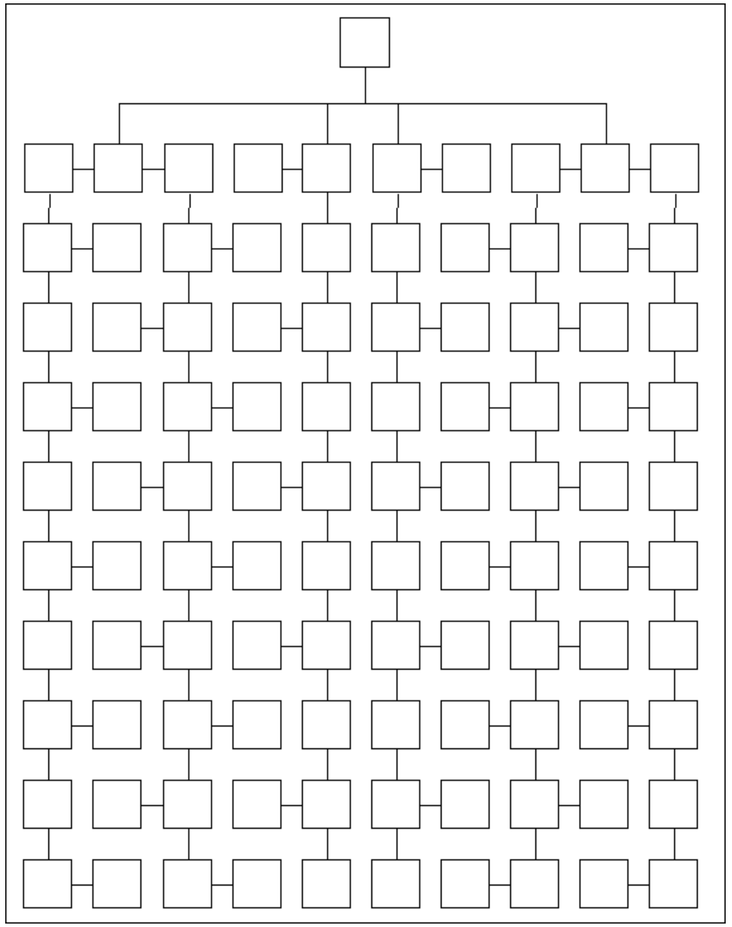
```
1  public interface Keyboard { }
2  //=========================================================
3  public class Windows98Machine{
4
5      private final Keyboard keyboard;
6      private final Monitor monitor;
7
8      public Windows98Machine(Keyboard keyboard, Monitor
   monitor) {
9          this.keyboard = keyboard;
10         this.monitor = monitor;
11     }
12 }
13 //=========================================================
14 public class StandardKeyboard implements Keyboard { }
```

Criticisms of SOLID

- **Criticism**
  - Vague principles
  - lead to complex and unintelligible code
  - focuses too much on dependencies
  - lead to long inheritance chains
  - leads to inconsistencies

Read More:

Marston, Tony. 2011.
"Not-so-SOLID OO Principles."



Too much separation and abstraction can make code unreadable.

SOLID Summary

Single Resposibility Principle

A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

Open / Closed Principle

A software module (it can be a class or method) should be open for extension but closed for modification.

Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Interface Segregation Principle

Clients should not be forced to depend upon the interfaces that they do not use.

Dependency Inversion Principle

Program to an interface, not to an implementation.

Timeline 1

Oct 1987

Barbara Liskov of MIT presents at a conference a paper titled *Data Abstraction and Hierarchy*. She uses the term "substitution property" and explains,*

> “Data abstractions provide the same benefits as procedures, but for data. Recall that the main idea is to separate what an abstraction is from how it is implemented so that implementations of the same abstraction can be substituted freely.

1988

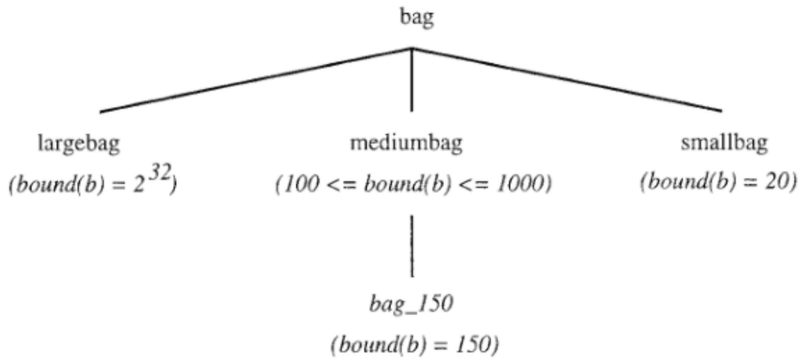
Bertrand Meyer, the creator Eiffel programming language, publishes a book titled *Object-Oriented Software Construction*.* Meyer is credited with introducing the Open-Closed Principle.*

1990

With growing codebases and complexity, object-oriented software design becomes popular to better manage software. But this change of programming paradigm from procedural to object-oriented does not automatically lead to clean code. Developers write large classes and methods. Code is duplicated. Old habits continue. There arises a need to guide developers design object-oriented software the right way.*

Timeline 2

1994



Barbara Liskov and Jeannette Wing present the case that anyone making use of supertype objects should not see any difference in behaviour even if using a subtype object instead.* This later becomes a principle of SOLID. This principle is also related to what Bertrand Meyer calls **Design by Contract**.*

Mar 1995

On com.object group, Robert Martin mentions a number of commandments for OOP. Three of these would later become part of SOLID: Open-Closed, Liskov Substitution, Dependency Inversion.*

2000

Robert Martin writes in detail about four of the SOLID principles. The Single Responsibility Principle doesn't appear on this list.*

2002

Robert Martin publishes the book *Agile Software Development: Principles, Patterns, and Practices*. He explains in detail all the five SOLID principles under a section named "Agile Design".* Thus, SOLID becomes an essential aspect of Agile methodology. Sometime later Michael Feathers coins the term SOLID as a useful way to remember the principles.*

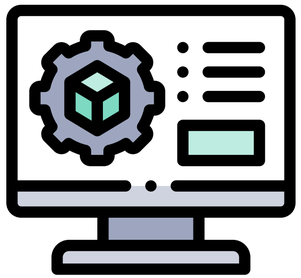
Timeline 3

2017

InterVenture publish extensions of SOLID. They identify six more principles. These are by no means universal in the industry but are worth studying.*

Sep 2020

Paulo Merson notes in a blog article that SOLID was designed OOP. It may not exactly fit microservices. For microservices, he therefore proposes **IDEALS**: interface segregation, deployability (is on you), event-driven, availability over consistency, loose-coupling, and single responsibility.*

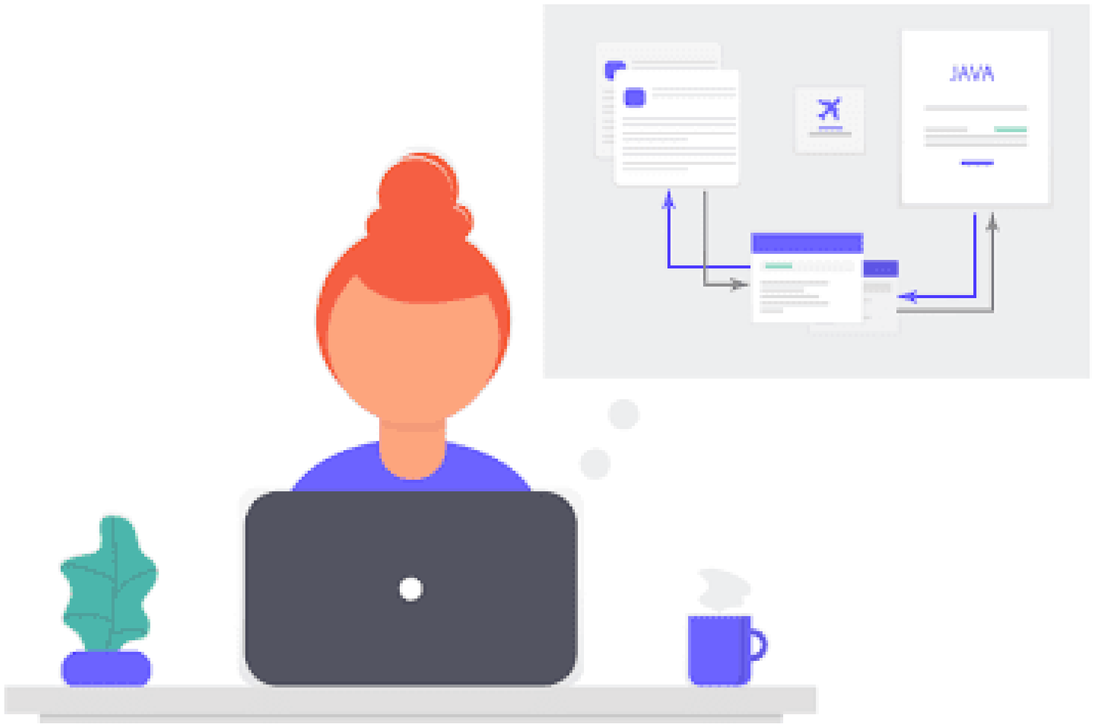


Software Design Patterns

GoF Patterns

Software Design Patterns

- **What's a Design Pattern?**
  - A solution to commonly occurring problems.
  - They are customizable blueprints.
  - They are NOT libraries where you can find and copy into your program.
  - Pattern (High level description) vs. Algorithms (clear set of actions)



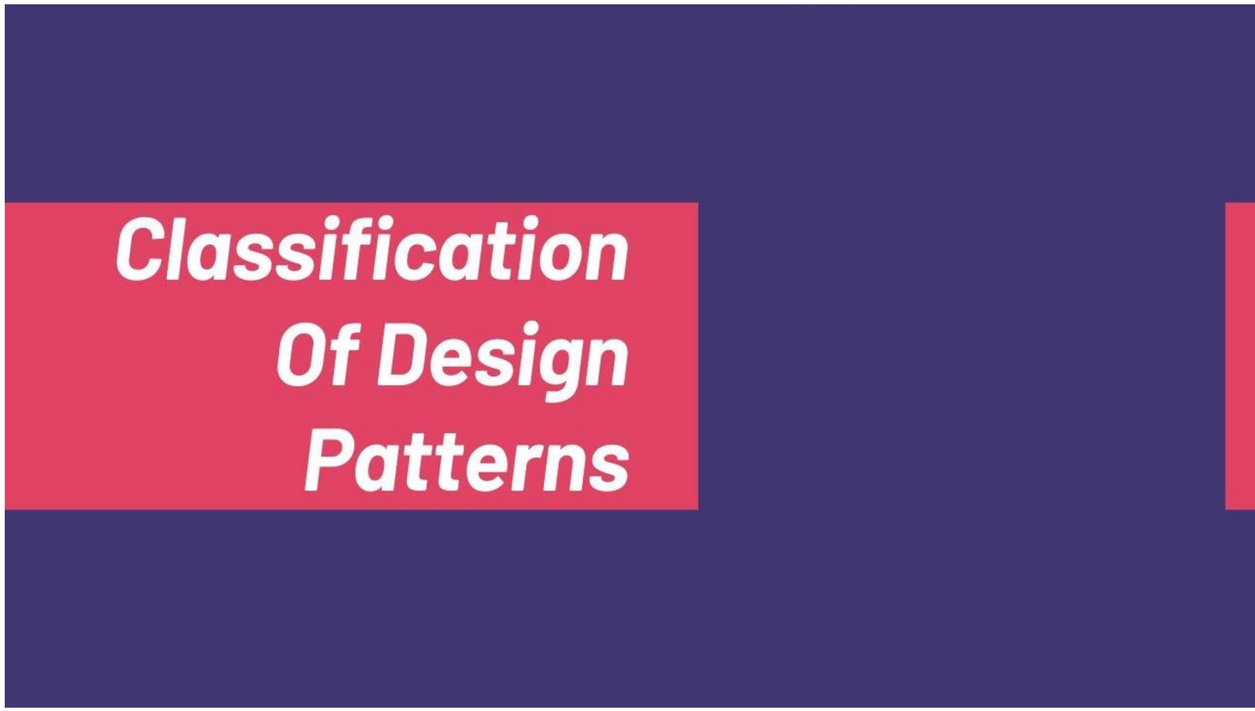
How to Describe Design Patterns?

- **Structure of a Design Pattern:**
  - Formal description of patterns has these sections:
  - **Intent**
    - States the problem that the pattern addresses
  - **Motivation**
    - A scenario that illustrates the problem
  - **Structure**
    - Normally displayed as some sort of object or class diagram
  - **Code Example**
    - A set of code that uses the pattern in one of the popular programming languages
  - **Other possible sections:**
    - Applicability, relations with other patterns, consequences, known uses, participants, collaborations

Intent	Collaborations
Motivation	Consequences
Applicability	Implementation
Structure	Known uses
Participants	Related patterns

Classification of Design Patterns

- **Based on Complexity, level of details, scale of applicability to the system:**
  - **Idioms**:
    - Basic and low-level
  - **Architectural patterns**:
    - Universal and high-level
- **Based on intent or purpose:**
  - Creational
  - Structural
  - Behavioral
- **Based on Scope:**
  - Class patterns
  - Object patterns



	Class Patterns	Object patterns
Creational	Can defer object creation to its subclasses	Can defer object creation to another object
Structural	Focuses on the composition of classes (primarily uses the concept of inheritance)	Focuses on the different ways of composition of objects
Behavioral	Describes the algorithms and execution flows	Describes how different objects can work together and complete a task

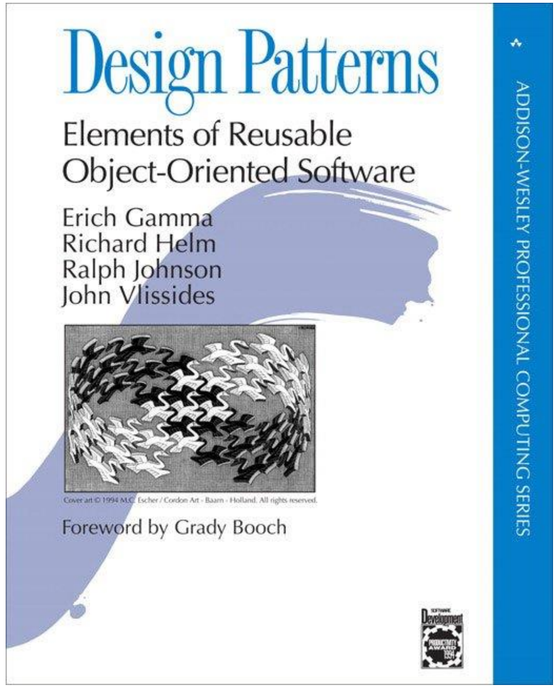
Who invented patterns?

Erich Gamma

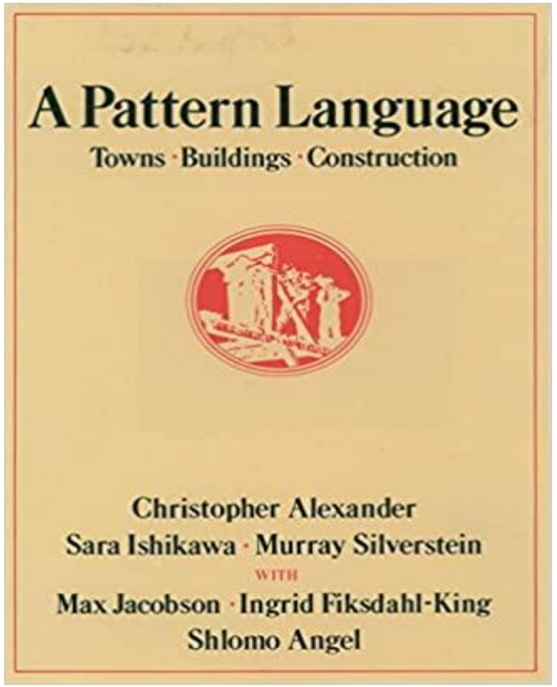
Richard Helm

Ralph Johnson

John Vlissides



Link (1995)



Link (1977)

Describes a “language” for designing the urban environment

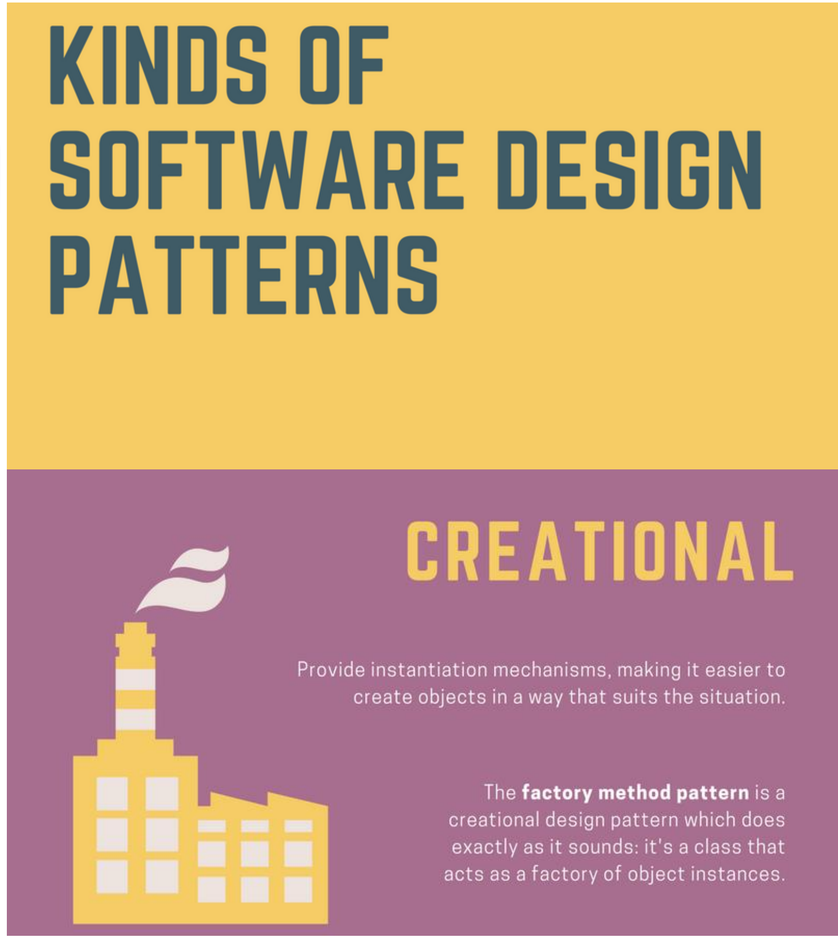
Software Design Patterns Categories: Creational

- **Related to object creation mechanisms**
- **Benefits:**
  - Increase flexibility
  - Increase reuse of existing codes
- **Main Ideas:**
  - Knowledge encapsulation
  - Hiding details about the actual creation and how objects are combined

Object Scope	Class Scope
Abstract factory pattern	Factory pattern
Builder pattern	Simple factory pattern
Prototype pattern	Singleton pattern

Creational Patterns

- *Builder pattern*
- *Singleton pattern*
- *Prototype pattern*
- *Factory method*
- *Abstract factory*



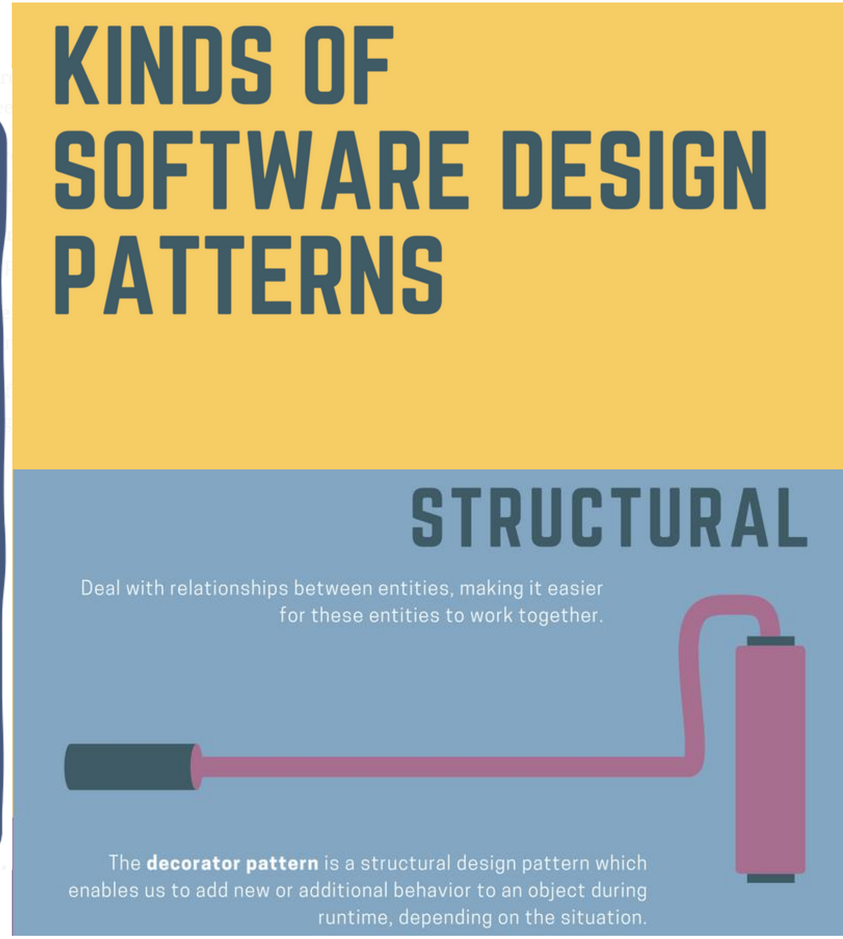
Software Design Patterns Categories: Structural

- **How to assemble objects and classes into larger structures.**
- **Benefits:**
  - Keeping the structure flexible and efficient
- **Main Ideas:**
  - The use of the composition to combine the implementations of multiple objects
  - Build a large system by maintaining a high level of flexibility

Object scope	Class scope
Adapter object pattern	Adapter class pattern
Bridge pattern	
Composite pattern	
Decorator pattern	
Facade pattern	
Flyweight pattern	
Proxy pattern	

Structural Patterns

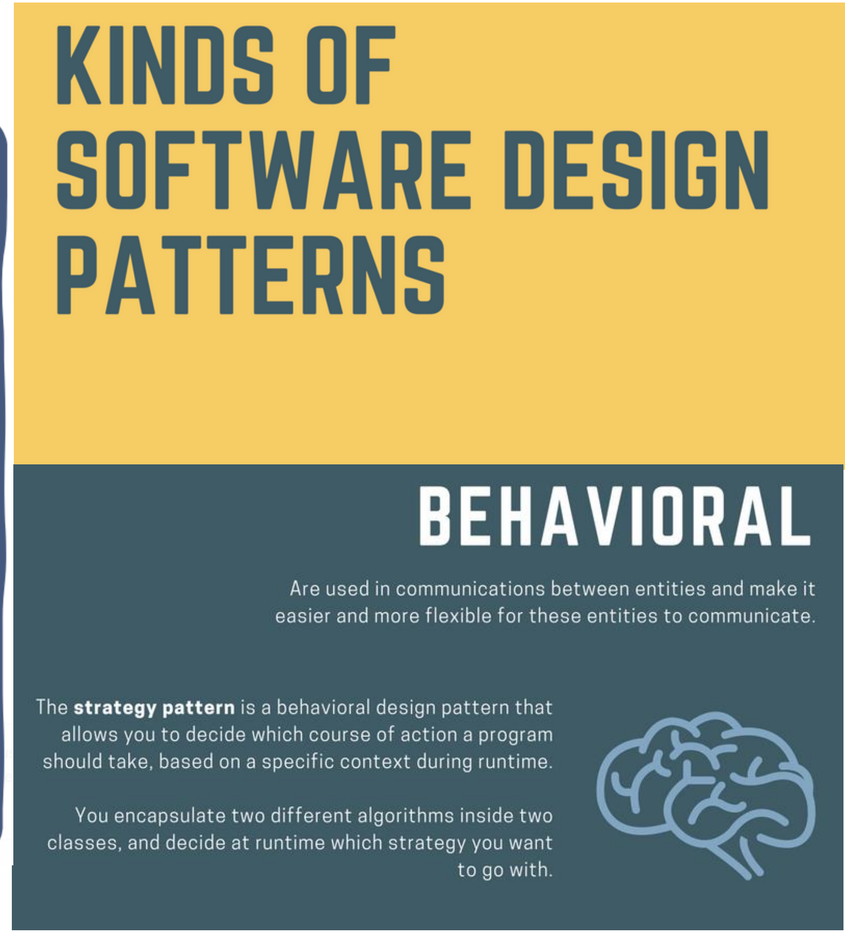
- *Adapter pattern*
- *Bridge pattern*
- *Filter pattern*
- *Composite pattern*
- *Facade pattern*
- *Decorator pattern*
- *Proxy pattern*
- *Flyweight pattern*



Software Design Patterns Categories: Behavioral

- **Concerned with algorithms and the assignments of responsibilities between objects.**
- **Main features:**
  - What is being described is a process
  - The flows are simplified
  - They accomplish tasks that would be difficult to achieve with objects

Object Scope	Class Scope
Chain of responsibility pattern	Interpreter pattern
Command pattern	Template method pattern
Iterator pattern	
Mediator pattern	
Memento pattern	
Null object pattern	
Observer pattern	
State pattern	
Strategy pattern	
Visitor pattern	

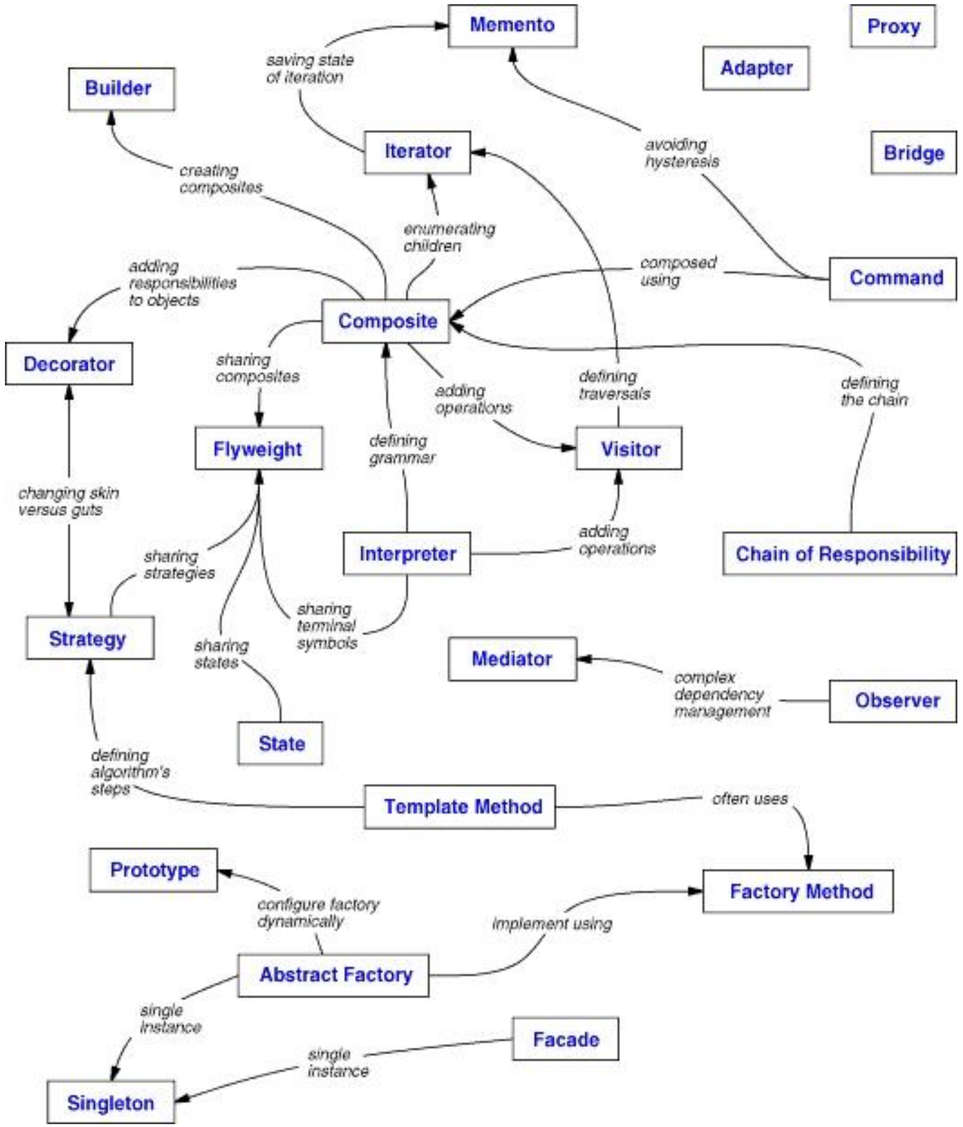


Why Learn/Use Design Patterns?

- **Define a common language for efficient communications.**
  - *“Oh, just use an Observer for that!”*
- **A toolkit of tried and tested solutions**
  - *“Oh! this is a job for an Observer!”*
- **Provide a generic solution.**
- **Testing systems are easier.**
- **Enhance the readability of the code.**
- **Improving software development process.**

Choosing a Design Pattern

- **Some approaches that helps to find appropriate pattern:**
  - How design patterns solve design problems
  - Read intent section
  - Analyze interrelation of patterns
  - Study the comparison of patterns
  - Consider the root cause of re-design
  - Understand the variable in your design



Any Question

??????????????

How do you feel about the course?

Please Send Your Question or Feedback...

Top **New**