OO ontwerpen

Samenvatting

Inhoud

[0. The OO Triangle 2](#_Toc472512554)

[1. SOLID Design Principles 3](#_Toc472512555)

[1.0 SOLID 3](#_Toc472512556)

[1.1 SRP 3](#_Toc472512557)

[1.2 OCP 4](#_Toc472512558)

[1.3 LSP 6](#_Toc472512559)

[1.4 ISP 7](#_Toc472512560)

[1.5 DIP 8](#_Toc472512561)

[2. Design Patterns 9](#_Toc472512562)

[2.1 Strategy 9](#_Toc472512563)

[2.2 State 10](#_Toc472512564)

[2.3 Observer 11](#_Toc472512565)

[2.4 Singleton 14](#_Toc472512566)

[2.5 Simple Factory 16](#_Toc472512567)

[2.6 Adapter 18](#_Toc472512568)

[2.7 Façade 19](#_Toc472512569)

[2.8 Template 21](#_Toc472512570)

[3. Refactoring 22](#_Toc472512571)

[3.1 What is it 22](#_Toc472512572)

[3.2 Test driven development 22](#_Toc472512573)

[4. Swing 22](#_Toc472512574)

[5. Enum 22](#_Toc472512575)

[6. Java Reflection API (Class) 22](#_Toc472512576)

[7. Properties 23](#_Toc472512577)

[7.1 Read file 23](#_Toc472512578)

[7.2 Use properties 24](#_Toc472512579)

[7.3 Read from XML 24](#_Toc472512580)

[7.4 Write 24](#_Toc472512581)

[7.5 Nice solution 24](#_Toc472512582)

[8. Inheritance, Aggregation & Composition 24](#_Toc472512583)

[9. MVC 26](#_Toc472512584)

[9.1 What? 26](#_Toc472512585)

[9.2 How? 26](#_Toc472512586)

[9.3 General UML 26](#_Toc472512587)

[9.4 Controller vs. Façade 26](#_Toc472512588)

# 0. The OO Triangle

State

Strategy

Façade

Adapter

Singleton

Template

Factory

Observer

Patterns

LSP

DIP

SRP

ISP

OCP

Inheritance

Polymorfism

Encapsulation

Abstraction

Basic OO

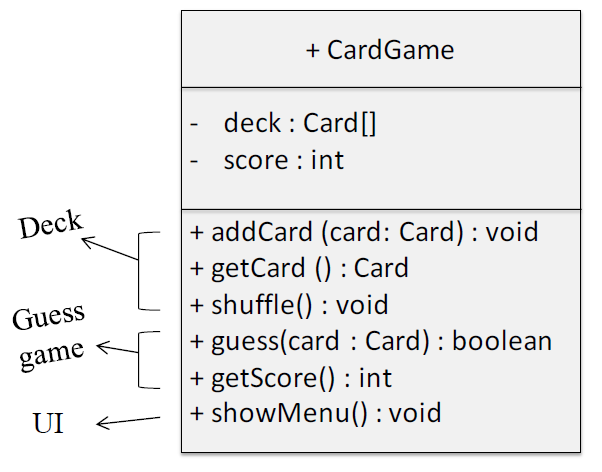
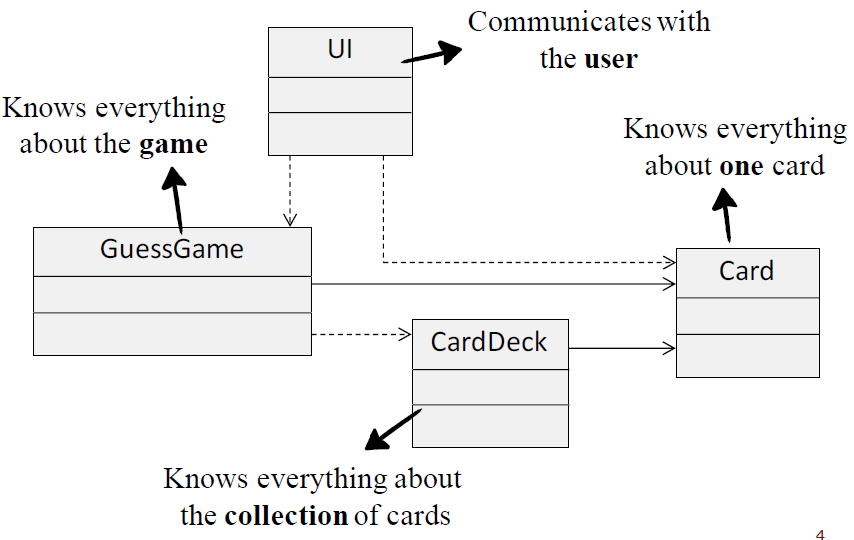
Principles

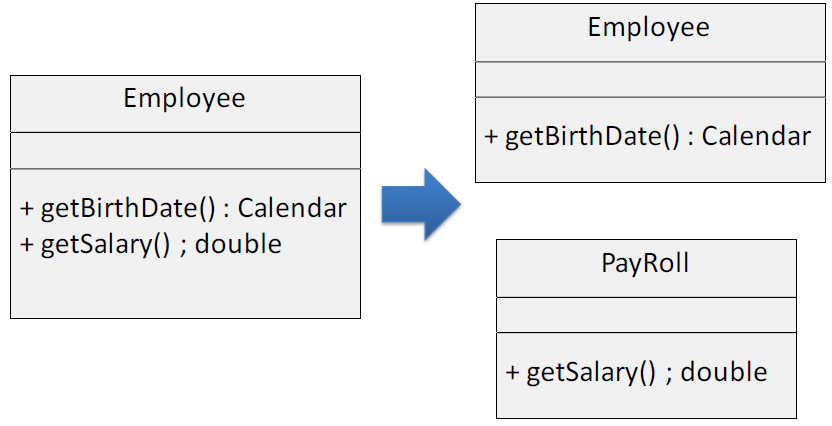
# 1. SOLID Design Principles

## 1.0 SOLID

* Founder: **Robert Martin** (Uncle Bob)
* Well recognized **set of principles**
* Useful **guidelines** for better OO programming (NOT rules)
* <https://www.youtube.com/watch?v=U0mNwweUF_s>

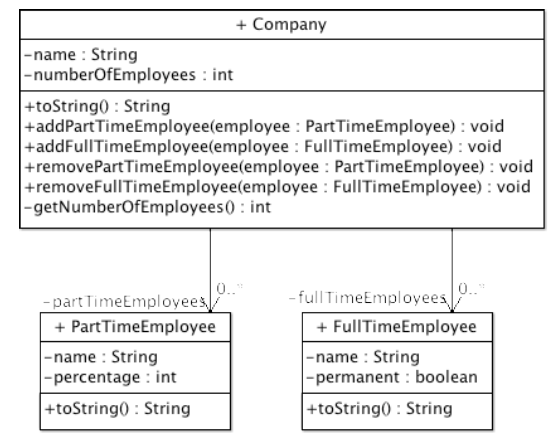
## 1.1 SRP

* **Single Responsibility Principle**
* *“A class should have one reason to exist/to change – one primary responsibility”*
* Each class has **one responsibility**
* When multiple responsibilities: **split class**
* **Cohesion**: more general concept but is closely related. When a class or module is designed around a set of related functions, it has *high cohesion*. Classes that adhere to SRP generally have high cohesion.
* **Example 1**
* **Example 2**

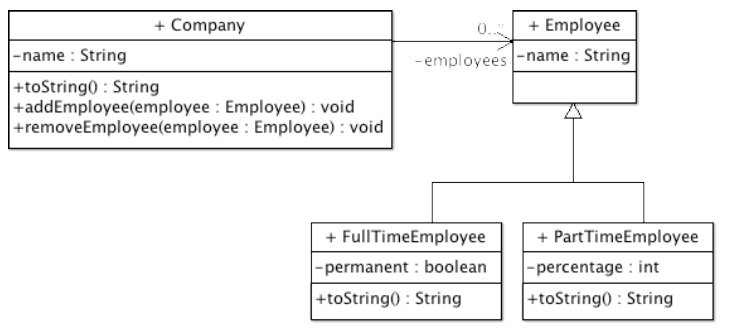


## 1.2 OCP

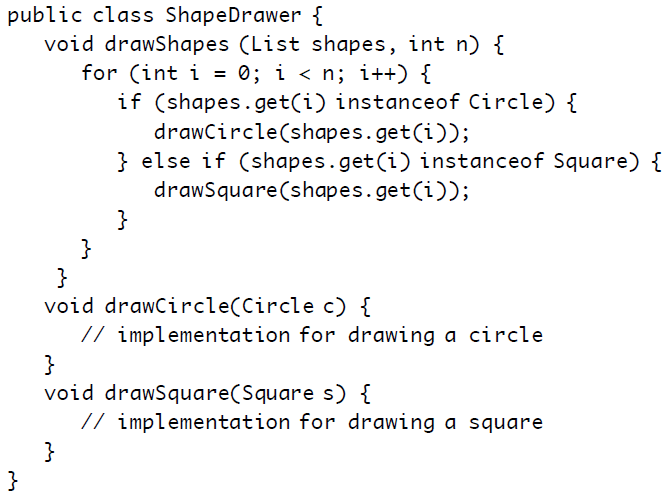
* **Open Closed Principle**
* *“Software entities (classes, functions, modules, etc.) should be open for extension, but closed for modification.”*
* New functionality should be added with **minimum changes in the existing code**
* Adding new functionality is done by **writing new code, not by rewriting existing code**
* **Example 1**



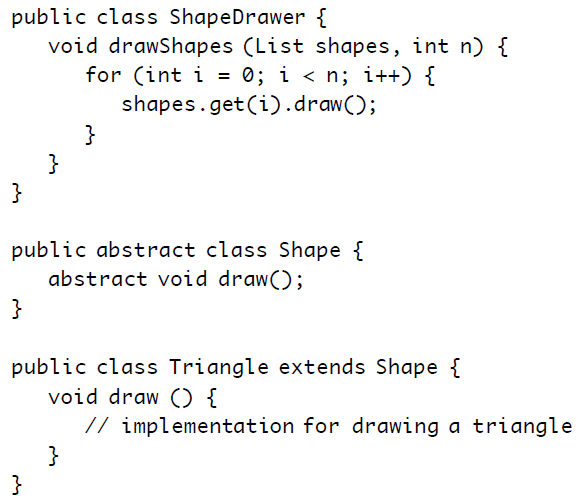
*Refactored for OCP:*



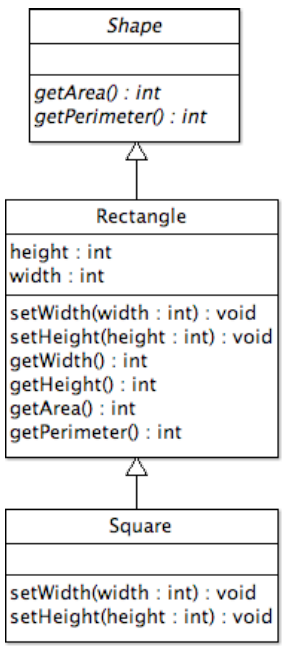
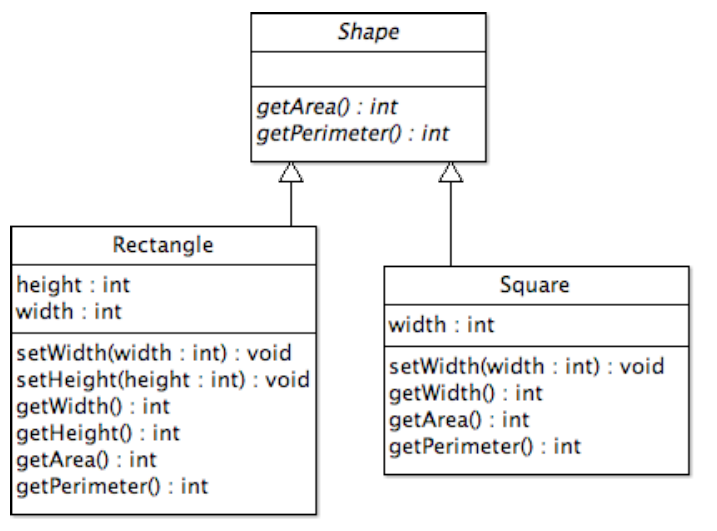
* **Example 2**



*Refactored for OCP:*

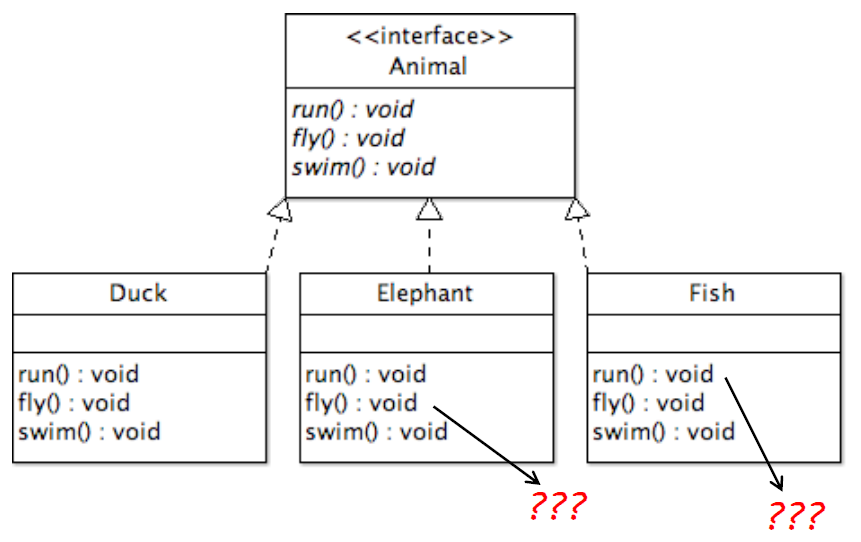


## 1.3 LSP

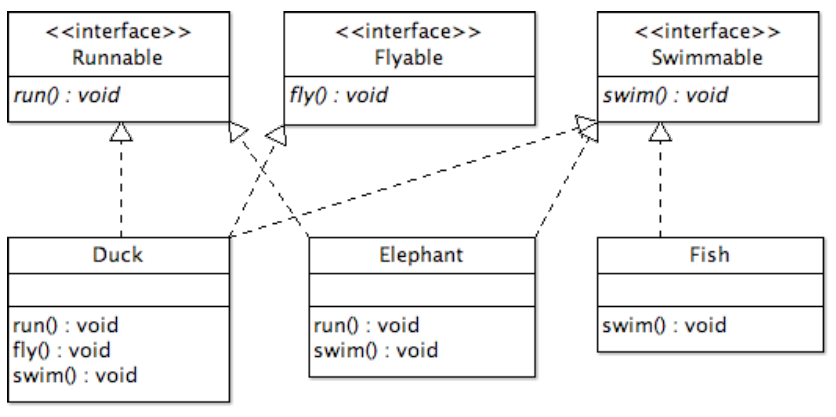
* **Liskov Substitution Principle**
* *“If a program module is using the reference of a base class* (super class)*, then it should be able to replace the base class with a derived class (sub class) without affecting the functioning of the program module”*
* Everywhere you use an **instance of a base class**, you should be able to **replace that instance with an instance of a sub class**
* Sub classes are allowed to do more than a super class, but **not less**
* You should always be able to treat each derived class like it’s the parent class
* No derived class should have to be treated specially
* **Example:**

## 1.4 ISP

* **Interface Segregation Principle**
* *“Clients should not be forced to depend upon interfaces that they don't use.”*
* **Multiple specific interfaces are better** than one general purpose interface.
* Interfaces should be as **small and specific as possible (high cohesion)**
* If interfaces get too much responsibilities, they should be **split up**
* No class should be forced to support a huge list of methods
* **Example:**

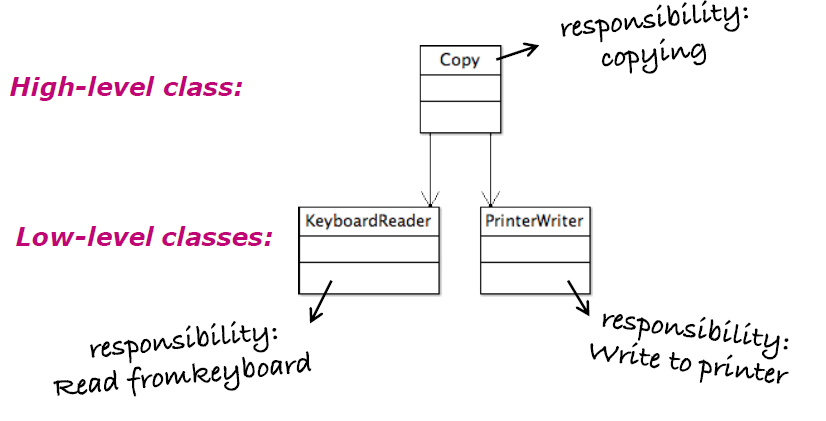


*Refactored for ISP*

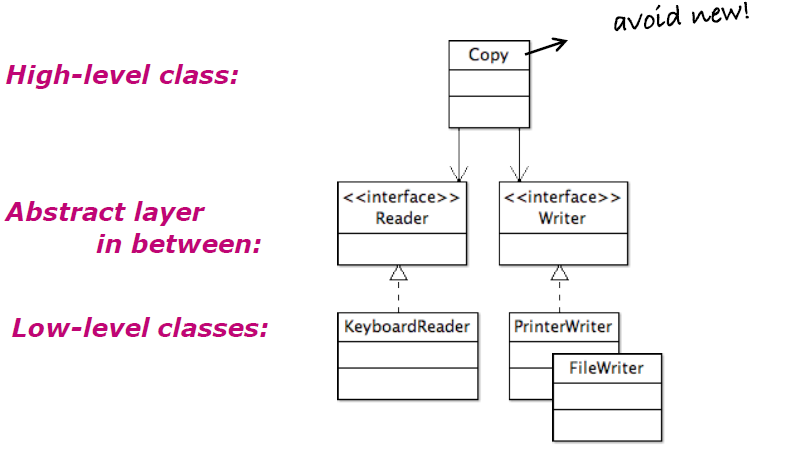


## 1.5 DIP

* **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.”*
* High-level classes do not use concrete low-level classes, they use **interfaces** (interface in the general sense) **as an abstract layer in between**
* Low-level classes do not know the high-level class, **they know the abstract layer**
* More flexibility: low level classes could be any class that inherits from the abstraction
* It allows flexibility, substitutions
* **Example:**



*Refactored for DIP*



# 2. Design Patterns

## 2.1 Strategy

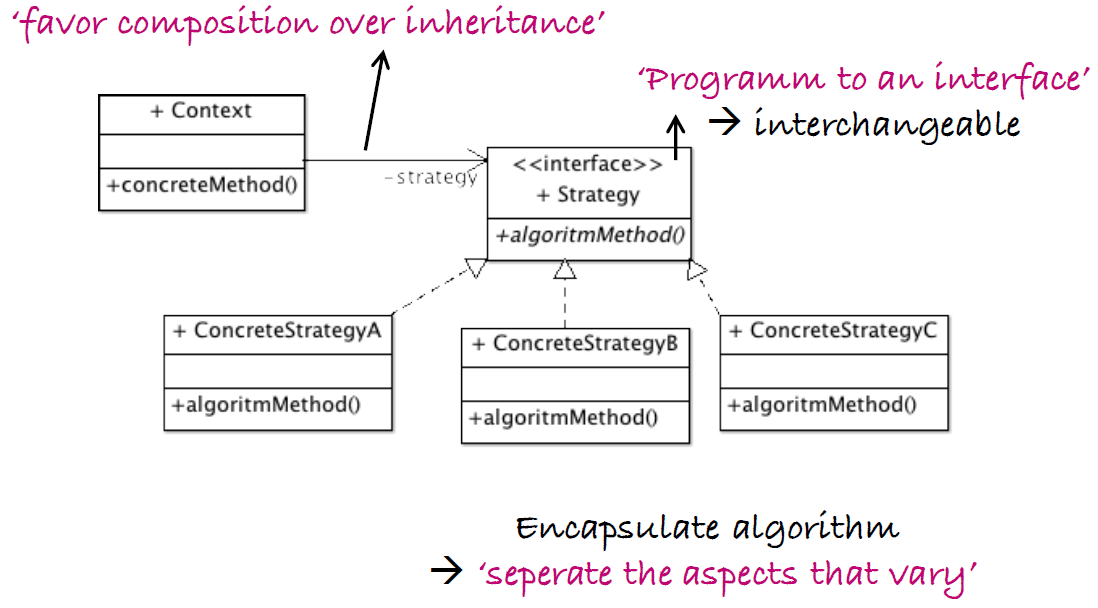
### Why?

* Define **family of algorithms**:
  + **encapsulate** each algorithm (= make separate class for this algorithm)
  + which makes them **interchangeable**
* Ensures the code is…
  + **flexible**: choose a different algorithm @runtime (i.e. with a setter)
  + **extendable**: add a new algorithm without modifying existing code (OCP)
* **Alternative to subclassing**: not stuck with behavior you don’t need

### How?

* Create **interface** for algorithm family
* Create **implementing classes** for algorithm(s)
* Method in context class **uses the interface** to execute the algorithm, instead of doing it itself
* **Favor composition over inheritance:** instead of inheriting behavior, behavior is composed with the right behavior object

### General UML example



### SOLID?

* **Single Responsibility?**
  + OK for ConcreteStrategy classes: each strategy class has 1 and only 1 responsibility: execute the algorithm it encapsulates
  + Pay attention: also check the rest of your design!
* **Open Closed?**
  + Almost OK!
  + Add new strategy, minimal rework in your client class

## 2.2 State

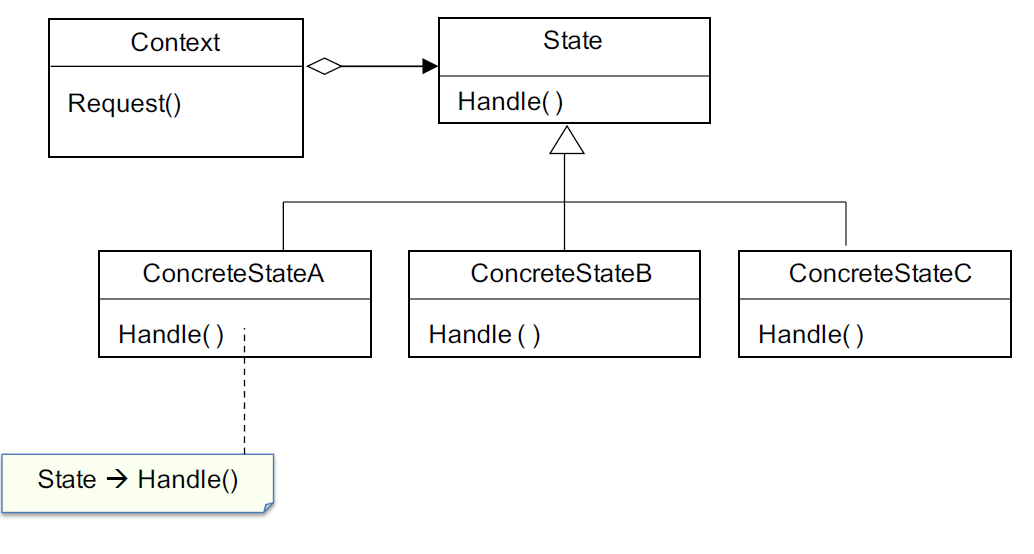
### Why?

* To **change behavior** changes **when state of object changes**
* To replace high-maintenance **if-else** structures
* Easy to **expand**

### How?

* Create **interface for state**
* Create **implementing classes** for states
* Context class contains **all possible states**
* Context class knows its **current state**
* Method in context class uses the current state to **handle the request**, instead of doing it itself

### General UML example



### SOLID?

* **SRP?**
  + OK. Each state is encapsulated in its own class
* **Open Closed?**
  + NOK: introducing a new state means modifying
    - the context class
    - (at least some of) the other states
* **Interface segregation?**
  + NOK: all states have to implement all methods

### Strategy vs. State

|  |  |
| --- | --- |
| State | Strategy |
| Client doesn’t choose, the context does (following a certain scheme) | Client chooses strategy |
| Alternative to if-else structures | Alternative to subclassing |

## 2.3 Observer

### Why?

* Defines a **one­to-many** relationship:
  + the **state** of one object **changes** (the subject)
  + all of its dependents are **notified and updated automatically** (the observers)

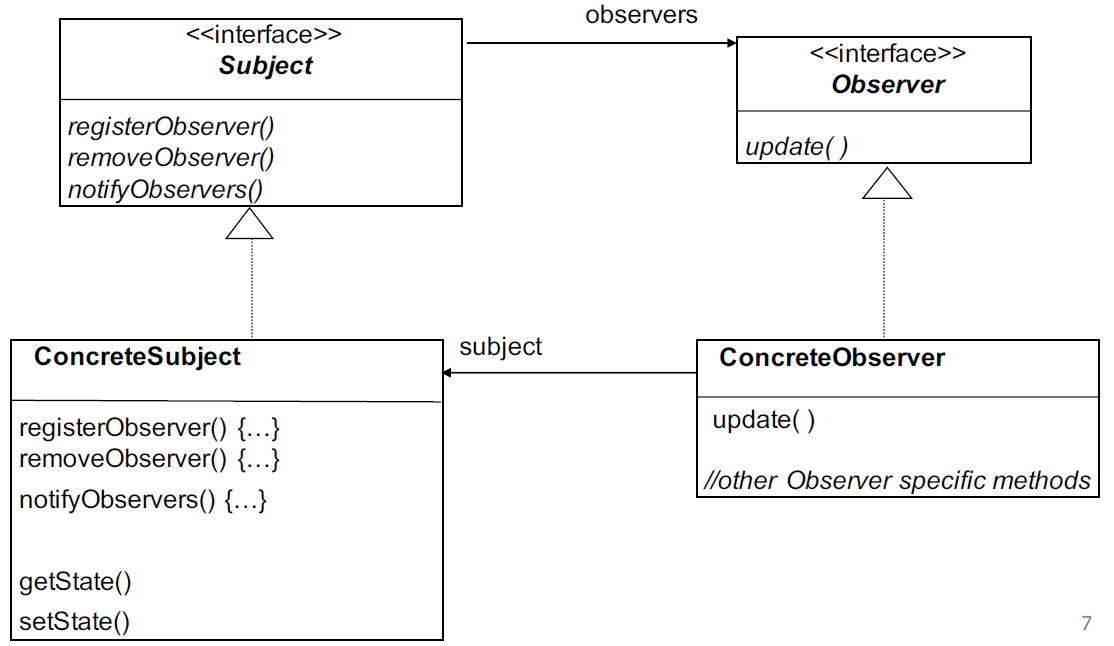
### How?

* Create **interface** for subject and observer
* Subject has **list of observers**
* When a change happens: **subject notifies observers**:
  + method notifyObservers() of subject calls…
  + method update() of observers

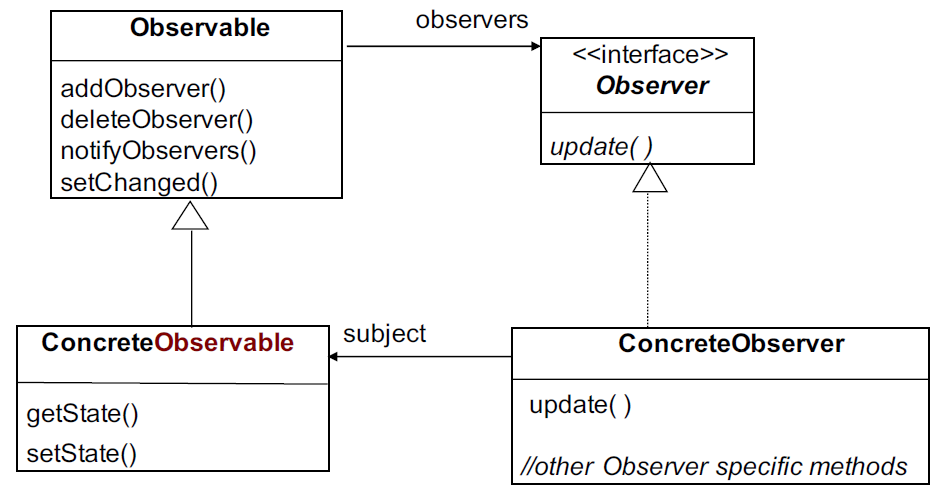
### Advantage

* **Without observer**
* Class with core information has fields for each interested class
* When we add an ‘interested class’, we need to modify ‘information’ class
* **With observer**: loose coupling
* Subject does not know the concrete observers, …
* Subject knows only their interface, that they have an update() method
* We can add new observers at any time
* We never need to modify the subject to add new types of observers

### General UML example



### Standard Java Observer UML



### Custom Observer vs. Java Observable

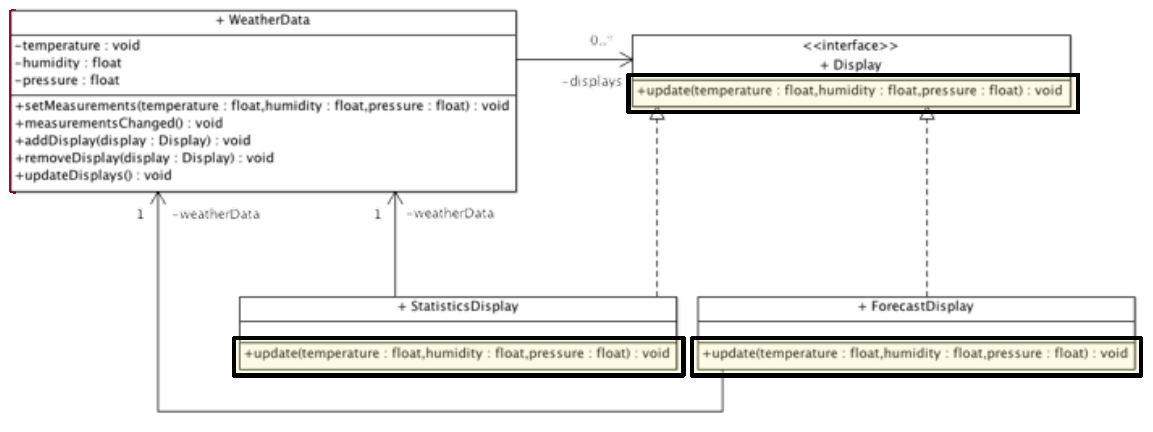
Java built-in Observable has a few disadvantages. The main reason is because Observable is **a class that needs to be extended in order to use it,** not an interface:

* You cannot add Observable behavior to a class that already extends another superclass
* You cannot create your own implementation because there isn’t an observable interface
* You cannot create an instance of the observable class and compose it with your own objects

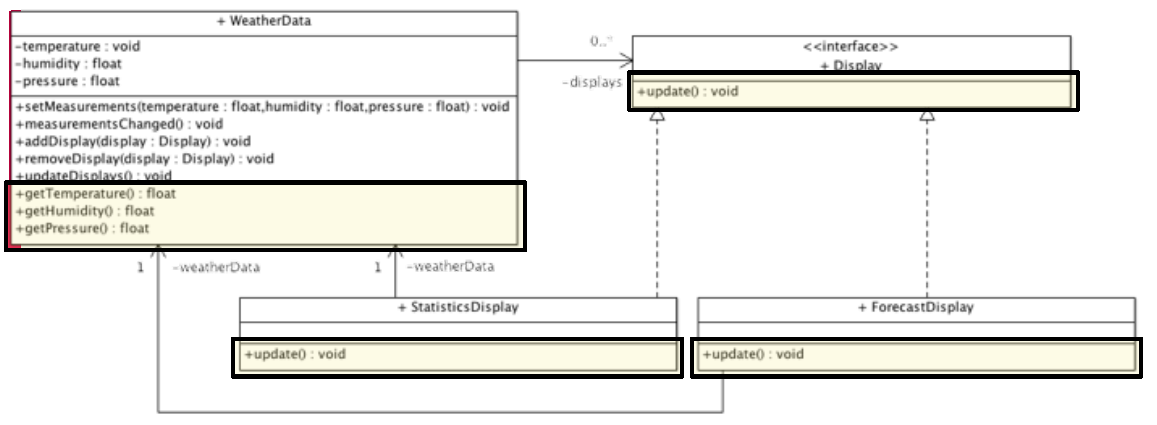
Java observable also has a **setChanged()** method. This method is called whenever the data has changed. It does not automatically call notifyObservers() in order to gain more control over when data is passed to the observers (see example p. 66).

### Push vs. Pull

* **Push**



* + **All data is pushed as parameters via update() method**
  + + The observers get all data in one notification
  + - It forces all displays to get all data, including data they don’t need
  + - If new instance variables need to be added, this is not possible
* **Pull**



* + **Observers get data with getters from subject**
  + - The displays may need to call several get methods to get all the data they need
  + + Displays can just request the data they really need

### Remarks

* Pull is considered more correct
* Don’t depend on a specific order of notification for your observers
* Swing, other GUI frameworks, JavaBeans and RMI make heavy use of the Observer pattern

## 2.4 Singleton

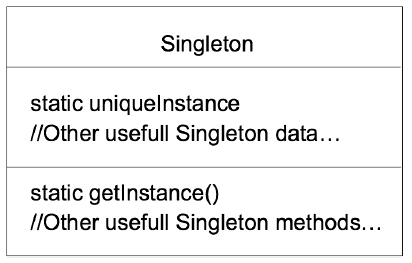
### Why?

* Only **ONE instance** of a class that is **globally** **accessible**
* **Avoid duplicates** of e.g. databases, settings, …
* Doesn’t have to be created when not needed (in contrast to global variables)

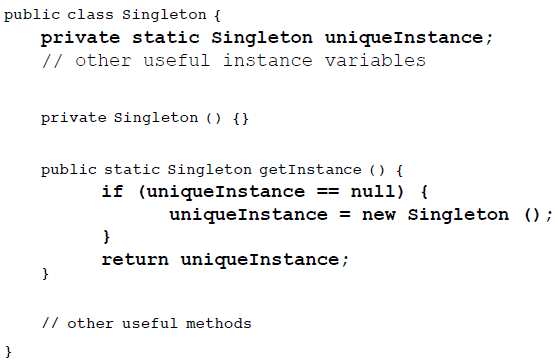
### How?

* **Static** **method**
* that calls **private constructor**
* making sure **only one instance** is created (if uniqueInstance == null...)

### General UML



### Code example

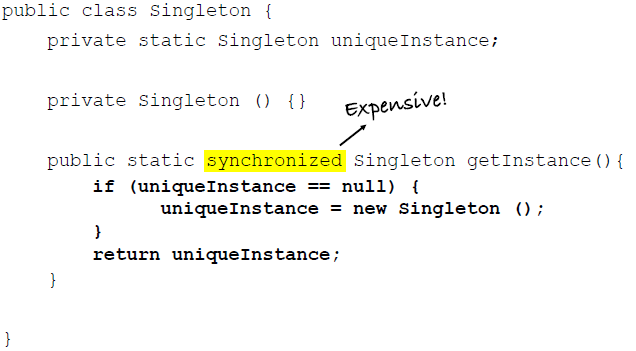


### Multithreading issues

If more than 1 thread runs the getInstance code at the same time, more than one instance could be created.

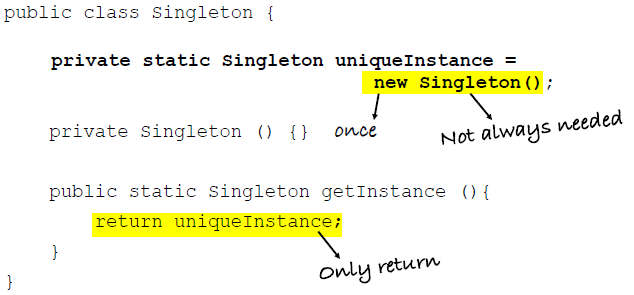
* **Solution 1**

*Synchronized* keyword makes threads wait turn to run method. This is useful *before* the initialization, but after it just takes up memory.



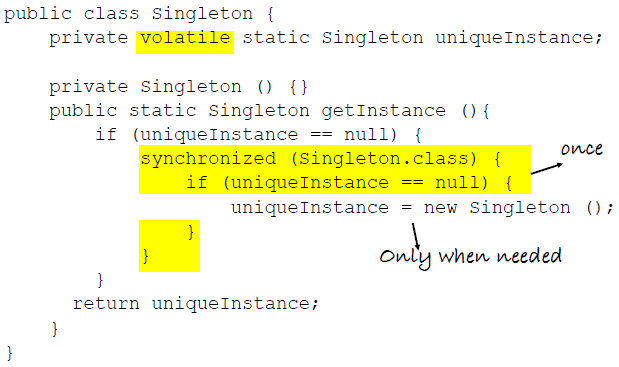
* **Solution 2**

Ensures the initialization happens only one, but the object may not always have to be initialized in the program, thus taking up memory.



* **Solution 3: double-checked locking**

This makes the *synchronized* block run only once: at initialization. The *volatile* keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.



### SOLID?

* **SRP**
  + Not really: singleton class manages its own unique instance + fulfills a role in the main purpose of the application
* **OCP**
  + NOK 🡪 use only when needed
  + private constructor 🡪 subclass not possible

## 2.5 Simple Factory

### Why?

* **Creating objects**:
  + without exposing **instantiation logic** to the customer
  + through a **common interface**

### How?

* Factory­class with method **createProduct(type: Type): Product**
  + Creates a **concrete instance** of product
  + … based on **type­parameter**
  + … and **returns it**
* Client
  + does **not** need to use **new** and
  + does **not** need to know the **dynamic type**

### Static or not?

* Not static

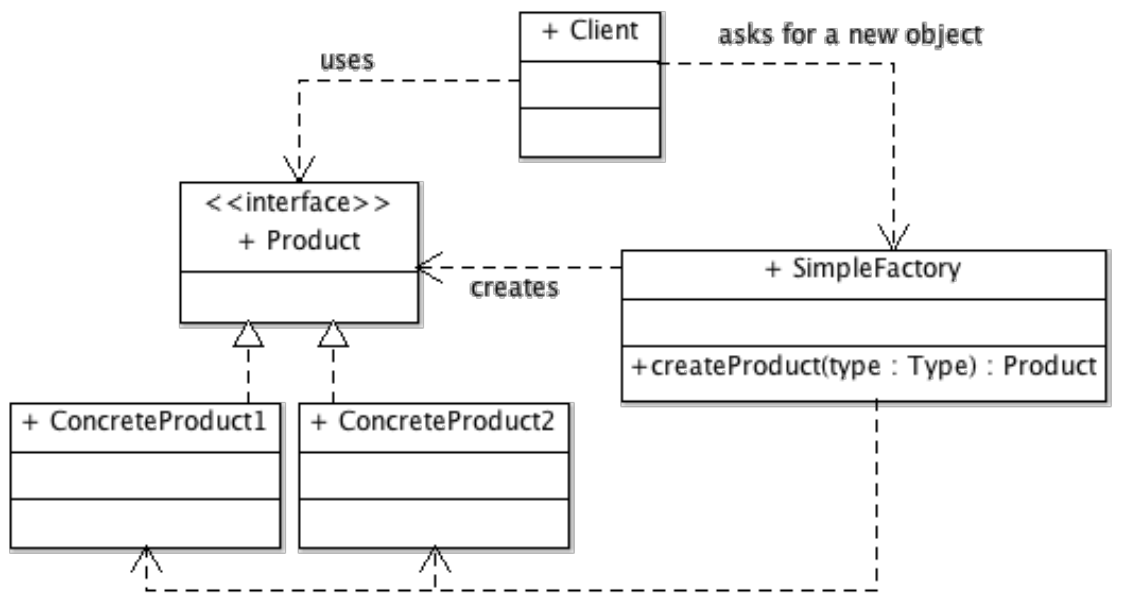
public Product createProduct…

* + Inheritance still possible
  + More flexible (subclassing possible)
* Static

public static Product createProduct…

* + No instance of Factory needed
  + Inheritance not possible (can’t subclass and change behavior of create method)

### General UML Example



### SOLID?

* **SRP**
  + OK 🡪 single responsibility = create object
* **OCP**
  + Client OK, Factory NOK
  + Not a major problem: no important logic and limited to one class
  + Supports OCP for the rest of the application
* **DIP**
  + Client and ConcreteProduct both depend on Product interface

### Simple Factory, Factory & Abstract Factory

* Simple factory is actually not considered a real, fully fledged design pattern
* Factory and Abstract Factory are the “real” patterns, provide more functionality
* See Head First Design Patterns, p.122

## 2.6 Adapter

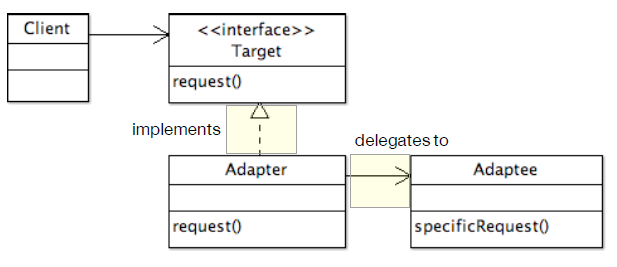
### Why?

* **Adapt the interface** of a class
* In order to enable **classes with different** (incompatible) **interfaces to work together**
* We **do not need to change code** of the target or the adaptee
* I.e. to adapt old legacy classes, libraries with a rewritten interface,…

### How?

* **Adaptee**
  + Has functionality **needed by the client**
  + But does **not implement expected** (target) **interface**
* **Adapter**
  + **Implements target interface**
  + Has an **instance of adaptee**
  + Does not execute request himself, **delegates** to adaptee

General UML example

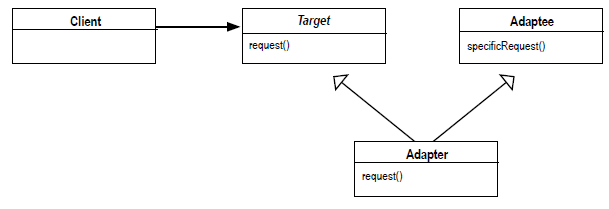


### SOLID?

* **SRP**
  + OK: adapter adapts
* **OCP**
  + OK
* **DIP**
  + OK

### Object vs. Class adapter

* Class adapter needs **multiple inheritance**,not possible in Java
* Class adapter: inheritance ⬄ Object adapter: composition



## 2.7 Façade

### Why?

* To **simplify** an interface
* A façade represents a **set of interfaces in a subsystem**

### How?

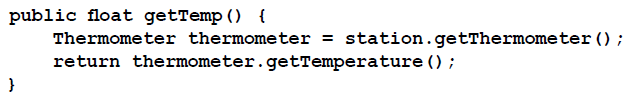
* **On top** of all classes of subsystem sits the façade class
* Client **talks to façade**
* Client **does** **not need to know** all the other classes
* Client **still has access to subsystem** if needed
* When a change is made to the subsystem, only façade code needs to be refactored, not client code (because it’s loosely coupled)

### Principle of least knowledge

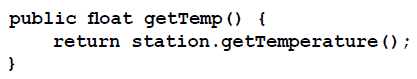
Guideline: only invoke methods that belong to:

* The **object itself**
* **Objects received as a parameter** to the method
* Any **object the method creates** or instantiates
* Any **components** of the object

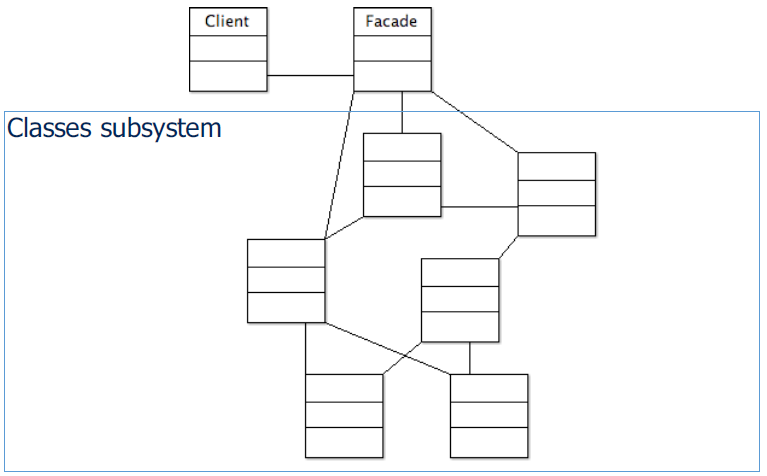
**Without the principle:**



**With the principle:**



### General UML example



### SOLID?

* **SRP**
  + OK: only intermediary
* **OCP**
  + NOK: new functionality 🡪 new methods

### Adapter vs. Facade

* **=**
  + wrap existing class(es)
  + does not add new functionality
* **Adapter**
  + adapt existing interface
* **Facade**
  + simplify existing interfaces



## 2.8 Template

### Why?

* To **enforce a specific algorithm**
* With substep **variations**

### How?

* **Template method** – skeleton (template) of algorithm in a (final) method in the superclass
* Each step is **represented by a method**
  + Some are handled by the superclass
  + Some by the subclass(es) -> defined abstract in superclass
* Superclass can provide a **hook**
  + Default or empty method in superclass
  + Is added to template method
  + Can be overridden by subclass to add steps to algorithm (optional)

### General UML Example



### Template vs. Strategy

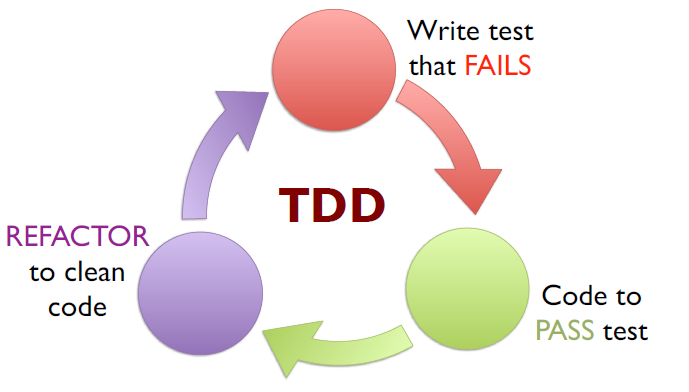
|  |  |
| --- | --- |
| Template | Strategy |
| Enforce algorithm | Make algorithm interchangeable |
| Choice at design time | Choice at runtime |
| Inheritance | Composition |

# 3. Refactoring

## 3.1 What is it

* **Restructuring existing code**
* To **improve readability and maintainability**
* **Without changing its external behavior**
* Risk: breaking existing code
* Use **tests** to make sure that the code still works
* **Never refactor and add new functionality at the same time!**
* Write code for who comes after you, not for the computer

## 3.2 Test driven development



**Step 1: refactoring**

1. Evaluate existing code
2. Evaluate test classes
3. Write new test cases if necessary
4. Refactor existing code if necessary

**Step 2: add new functionality**

1. Write new test cases
2. Add new code

# 4. Swing

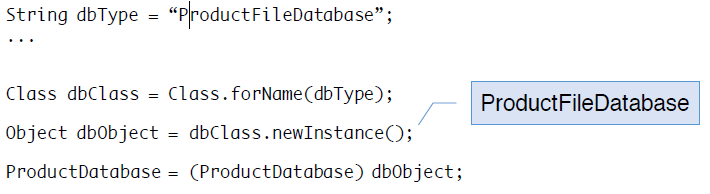
See slides & lab.

# 5. Enum

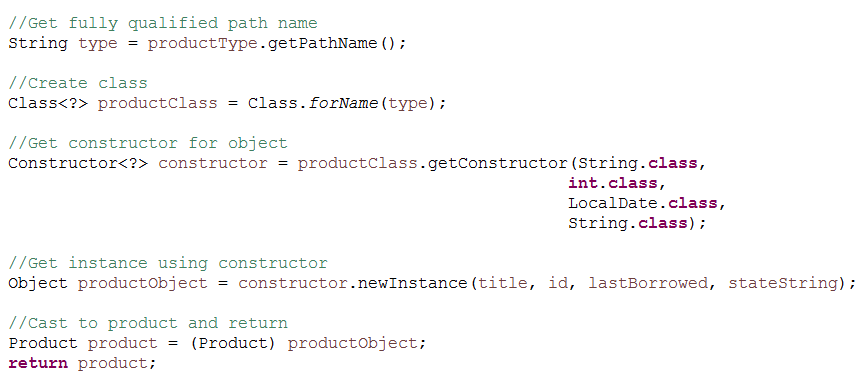
* Special kind of class
* ***“enum”*** keyword instead of “*class”*
* **Fixed** **set of objects**, i.e:
  + Months
  + Days of the week
  + Wind directions
* See also: *Example use of enum in Java*

# 6. Java Reflection API (Class)

* **Reflection** is the ability of a computer program to *examine* and to *modify* the structure and behavior of the program **at runtime**
* Example in Java: **Class**
  + Used to **manipulate** **classes** ad everything in a class
  + Can **slow down** a program (because JVM can’t optimize the code)
  + Can’t be used with **applets**
  + Should be used **sparingly**
* **Example:**



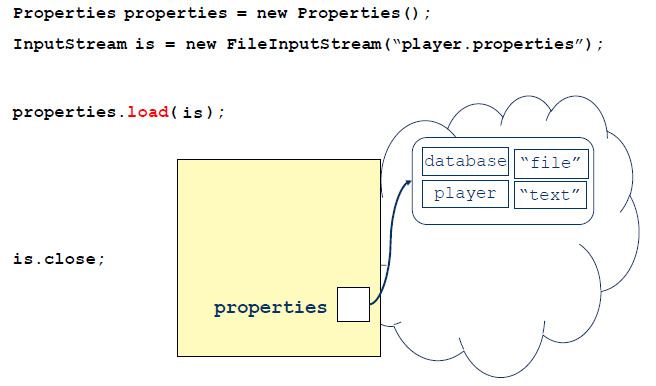
* The newInstance() method uses the default constructor, therefore the “empty” constructor 🡪 solution: use Constructor class
* **Practical example:**



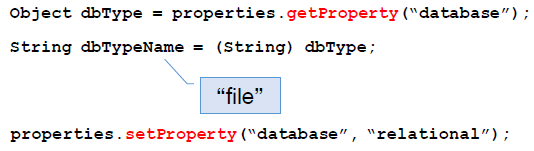
# 7. Properties

*Properties* Class: used for **reading**, **using** and **writing** properties of an application

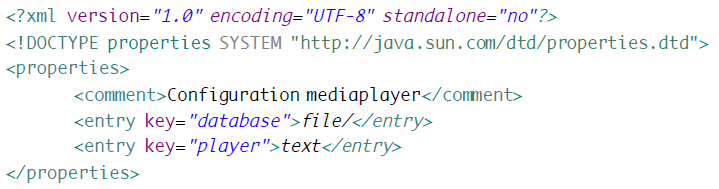
## 7.1 Read file

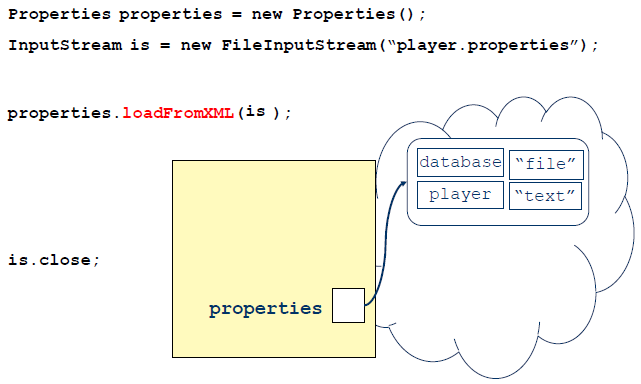


## 7.2 Use properties

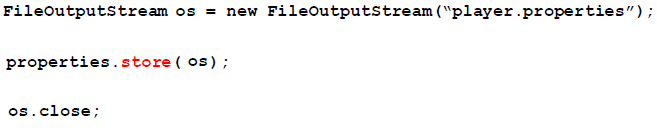


## 7.3 Read from XML





## 7.4 Write



## 7.5 Nice solution

* A nice solution to always having to load and save a properties file with multiple lines of code, is to encapsulate this behavior
* E.g. a class **PropertiesFile** with a **properties** field and methods like
  + **read()**
  + **write()**
  + **get(String property)**
  + **set(String property, String value)**
* For example, see shop project

# 8. Inheritance, Aggregation & Composition

* Design principles
* <https://youtu.be/d5ecYmyFZW0>
* **Inheritance** can be defined as an “**is a**” relationship
  + Empty, closed arrow in UML
* **Aggregation** can be defined as a “**has a**” relationship
  + Empty diamond shape in UML
* **Composition**
  + Is a more **specific type of aggregation**
  + Implies **ownership**
  + **When the owning object is destroyed, so are the contained objects**
  + Closed diamond in UML

# 9. MVC

## 9.1 What?

MVC isolates the application logic from the UI, permitting independent development, testing and maintenance of each.

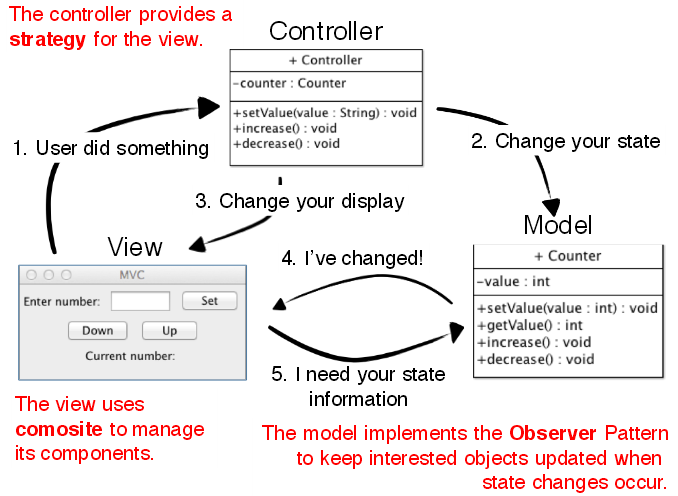
* The **model** holds all the data, state and application logic.
* The **view** gives a presentation of the model.
* The **controller** takes user input and figures out what it means to the model.

Java MVC tutorial: <http://www.newthinktank.com/2013/02/mvc-java-tutorial/>

## 9.2 How?

* MVC as a **compound pattern:**
  + A **combination** of a few patterns
  + To solve a **general problem**
* Controller is **strategy** for view
* **Observer**
  + Model is observable
  + View registers as an observer
* View uses **composite** to manage its components

## 9.3 General UML



## 9.4 Controller vs. Façade

* They are both **intermediary**
* **Difference:**
  + Façade is part of the model 🡪 hides complexity
  + Controller translates between view and model