**Facade Pattern**

Facades are all around us in the real world.  Operating systems are one such example - you don't see all the inner workings of your computer, but the OS provides a simplified interface to use the machine. Buildings also have a facade - the exterior of the building.

**Motivation**

* Structuring a system into subsystems helps reduce complexity
* Subsystems are groups of classes, or groups of classes and other subsystems
* The interface exposed by the classes in a subsystem or set of subsystems can become quite complex
* One way to reduce this complexity is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem

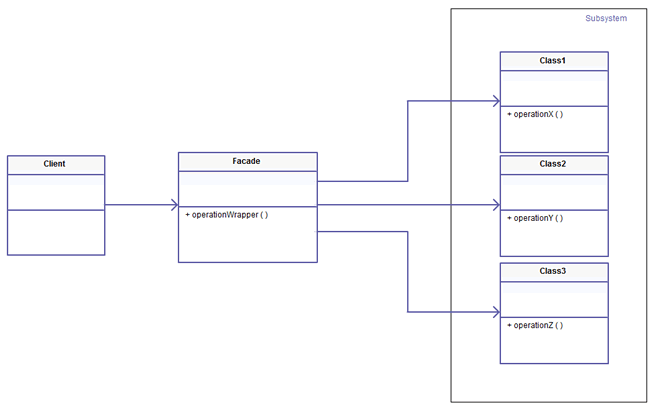


**Intent**

* Provide a unified interface to a set of interfaces in a subsystem.
* Façade defines a higher-level interface that makes the subsystem easier to use.

**Implementation**

The figure below shows a UML class diagram for the Facade Pattern:



**Applicability & Examples**

Use the Facade pattern:

* To provide a simple interface to a complex subsystem. This interface is good enough for most clients; more sophisticated clients can look beyond the facade.
* To decouple the classes of the subsystem from its clients and other subsystems, thereby promoting subsystem independence and portability

**Code Sample**

Without Façade

<?php

// Various classes we want to shield the Client from.

**class** Adder

{

**public function** add($a, $b)

{

**return** $a + $b;

}

}

**class** Subtractor

{

**public function** subtract($a, $b)

{

**return** $a - $b;

}

}

**class** Multiplier

{

**public function** multiply($a, $b)

{

**return** $a \* $b;

}

}

**class** Divider

{

**public function** divide($a, $b)

{

**if** ($b == 0) {

**throw new** Exception('Division by zero.');

}

**return** $a / $b;

}

}

// Client code

$adder = **new** Adder();

**echo** '254 + 113 = ', $adder->add(254, 113), "\n";

$divider = **new** Divider();

**echo** '256 / 8 = ', $divider->divide(256, 8), "\n";

Applying Façade

<?php

// Various classes we want to shield the Client from.

**class** Adder

{

**public function** add($a, $b)

{

**return** $a + $b;

}

}

**class** Subtractor

{

**public function** subtract($a, $b)

{

**return** $a - $b;

}

}

**class** Multiplier

{

**public function** multiply($a, $b)

{

**return** $a \* $b;

}

}

**class** Divider

{

**public function** divide($a, $b)

{

**if** ($b == 0) {

**throw new** Exception('Division by zero.');

}

**return** $a / $b;

}

}

**class** CalculatorFacade

{

**public function** \_\_construct(Adder $adder,

Subtractor $subtractor,

Multiplier $multiplier,

Divider $divider)

{

$this->\_adder = $adder;

$this->\_subtractor = $subtractor;

$this->\_multiplier = $multiplier;

$this->\_divider = $divider;

}

**public function** calculate($expression)

{

**list** ($a, $operator, $b) = explode(' ', $expression);

// eliminating switch constructs is not in the intent of this pattern

**switch** ($operator) {

**case** '+':

**return** $this->\_adder->add($a, $b);

**break**;

**case** '-':

**return** $this->\_subtractor->subtract($a, $b);

**break**;

**case** '\*':

**return** $this->\_multiplier->multiply($a, $b);

**break**;

**case** '/':

**return** $this->\_divider->divide($a, $b);

**break**;

}

}

}

**class** CalculatorFactory

{

**public function** getCalculator()

{

**return new** CalculatorFacade(**new** Adder(), **new** Subtractor(), **new** Multiplier(), **new** Divider());

}

}

// Client code

$calculatorFactory = **new** CalculatorFactory;

$calculator = $calculatorFactory->getCalculator();

**echo** '254 + 113 = ', $calculator->calculate('254 + 113'), "\n";

**echo** '256 / 8 = ', $calculator->calculate('256 / 8'), "\n";

In the example, the library vendor or the component's owner is now free to change the internal concrete classes.

Moreover, the vendor can still **change signatures and move methods** between classes, a feature that even by using Abstract Factories to return Adder and its sibling classes cannot be achieved.

Only external interfaces are published, while the contracts between internal classes are not referred to in any way, stimulating internal refactoring and shifting of responsibilities between classes.

For instance, we can easily **merge or extract classes and introduce collaborators** without the Client code being affected.

Facade is one of the most powerful decoupling patterns because it hides every unnecessary assumption beyond a heavy curtain, thus preventing change in a software module, which will certainly happen, from spreading in the whole application.

**Specific problems and implementation**

The following need to be avoided while implementing a Facade:

* It **should not be a Singleton** or a static class; otherwise the global state of the subsystem will be hidden under the Facade, effectively preventing isolation of tests that involve. Imagine test against a database schema you cannot ever reset.
* The Facade **should only return and accept interfaces or value objects** in its method signatures, so that there are no transitive dependencies. If the Facade gives away references to the internal collaborators of its module, it won't achieve its decoupling goal.

**Consequences**

**Benefits**

* It hides the implementation of the subsystem from clients, making the subsystem easier to use
* It promotes weak coupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.
* It reduces compilation dependencies in large software systems
* It simplifies porting systems to other platforms, because it's less likely that building one subsystem requires building all others
* It does not prevent sophisticated clients from accessing the underlying classes
* Note that Facade does not add any functionality, it just simplifies interfaces

**Liabilities**

* It does not prevent clients from accessing the underlying classes!