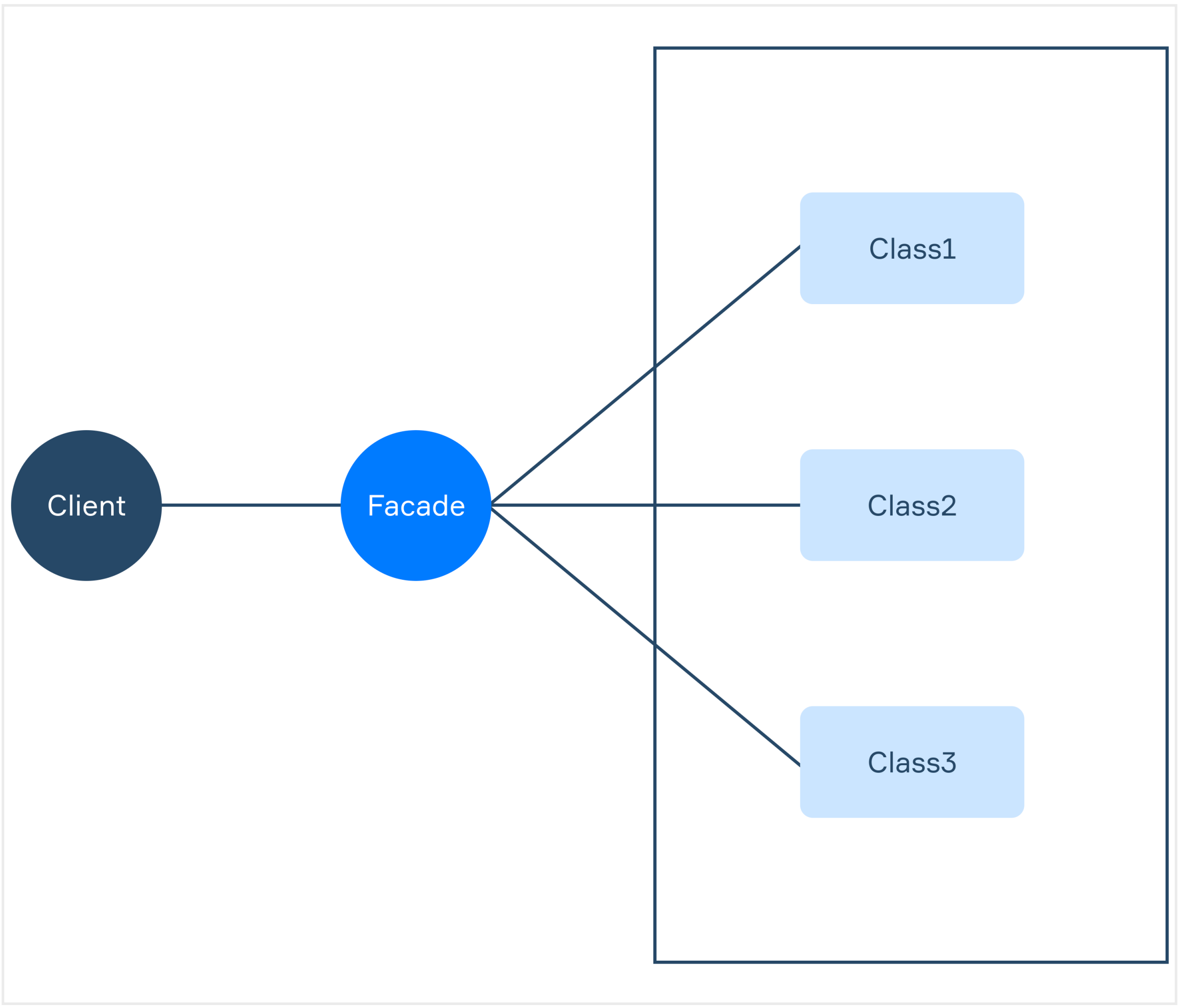
Facade pattern

The **facade design pattern** is a structural design pattern that provides the client with an interface for accessing parts of the system which could be hidden to avoid showing the complexity of the code.

Basically, a facade can be applied, when you are working with a complex set of objects, which you do not want to track.

Usually, this pattern is used when you need to provide a simple way to access a library or a framework.



Facade pattern goal

A facade works through an interface with the same name.

If you have worked with a library or a framework with lots of different classes, you could have worked with them using special objects.

These objects were probably created as a facade.

The facade interface provides simplified access to methods of classes that are hidden from the client.

It could be constricted access, compared to direct access to these classes.

But this approach will relieve clients from functions in which users are not interested. For example, when you play some video game, all of the components that you see are a finished graphical representation of application functions, that hides all the code from you.

But, as you can guess, a facade object could become a god object, meaning that it could do or know too much. So, if it becomes too big, consider diverting some of its functions by creating other facade objects.

Facade example

As an example of the facade pattern, we will take a look at video games.

A game itself consists of complex code that has a lot of hidden functions that players don't need to think about.

After a player's actions, the game needs to calculate geometry, render graphics, load data, and many other things. These classes cannot be accessed directly by the player, and frankly shouldn't be, or they can break the game:

```
class Render is ...
  ...


class Calculate is ...
  ...


class Load is ...
  ...
```

So, let's consider that in this game you can jump, walk, start a new game, and do other explicit actions.
Players get to know about them through the UI and can execute them by pressing physical or virtual buttons.

These functions allow players to change parameters in classes that are usually unavailable to players.

As a simple example:

- When we walk, the game needs to render new objects that weren't visible before.
- When we jump, the game is supposed to calculate our new position.
- When we start a new game, files need to be loaded from the storage.

The players' actions will be executed through a class called Game that represents our facade:

```
class Game is
  method walk() is
    ...
    return new Render()

  method jump() is
    ...
    return new Calculate()

  method startGame() is
    ...
    return new Load()
```

Facade implementation

Facade implementation in the client code will look pretty boring:

```
Game game = new Game()
game.startGame()
game.walk()
game.jump()
```

As we can see, all the actions and functions are happening somewhere behind the scene. This way, client code looks like some objects giving commands to other hidden parts of the code that we cannot access easily.