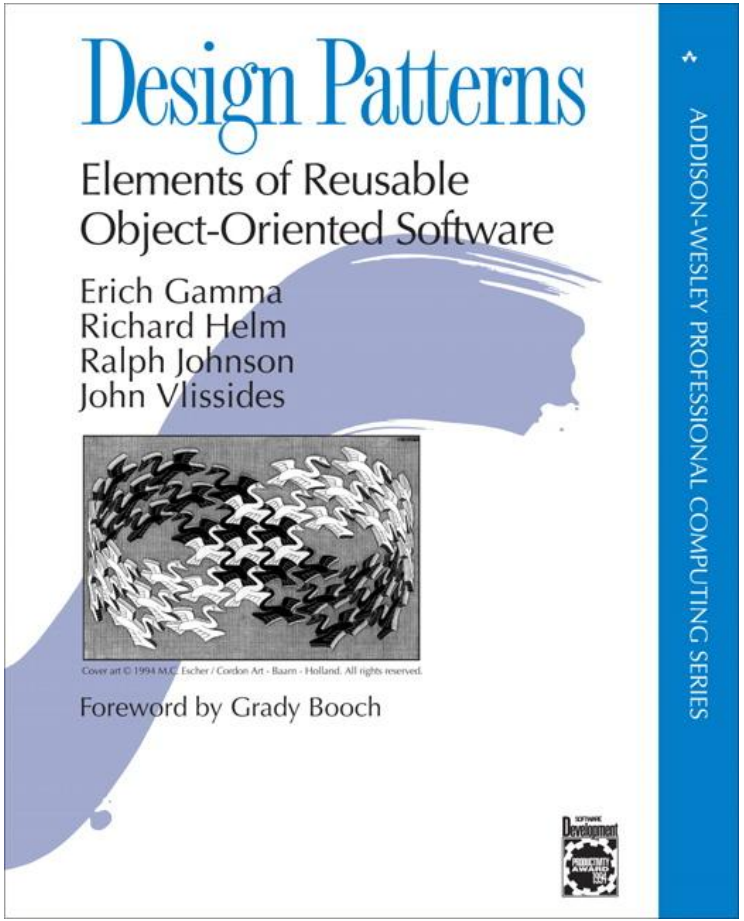
Design patterns for dummies

A short introduction to software design patterns

A dated, but still useful book



What is a software design pattern?

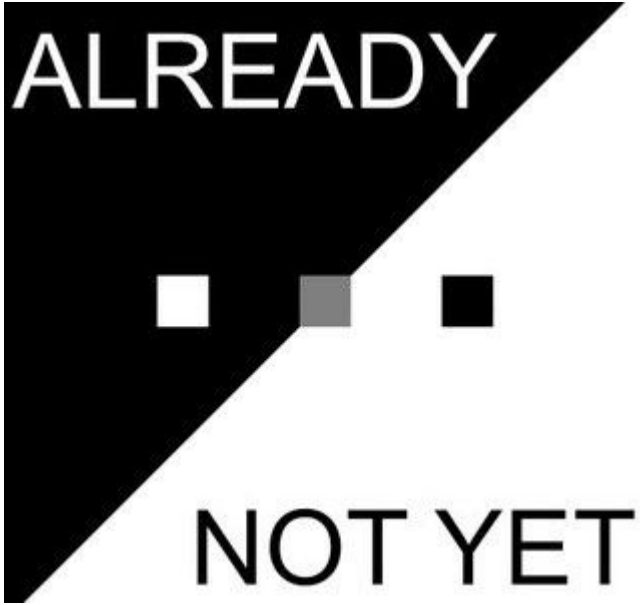
In software engineering, a software design pattern is a general **reusable solution to a commonly occurring problem** within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. **Design patterns are formalized best practices** that the programmer can use to solve common problems when designing an application or system.

Why are they important?

- Common vocabulary
- Common solutions to common problems
- Accepted as good solutions
- Prevent custom bad design
- Help make better architecture
- Improved readability
- Less code
- S.O.L.I.D.

F.A.Q.

- Should I use design patterns always?
  - No. Use them when they actually improve the software architecture.
- Can I use them?
  - Yes, each design pattern specify when, why and how to use it.
- Should I know them all?
  - You should know all the concepts behind basic patterns, and google them when you want to use one.
- When should I use them?
  - Some books use the concept of "***Not yet***". A pattern is a generalization of a problem. You should generalize when not doing so will violate the S.O.L.I.D. rules.



Creational patterns

Creational patterns provide ways to instantiate single objects or groups of related objects.

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns

Structural patterns provide a manner to define relationships between classes or objects.

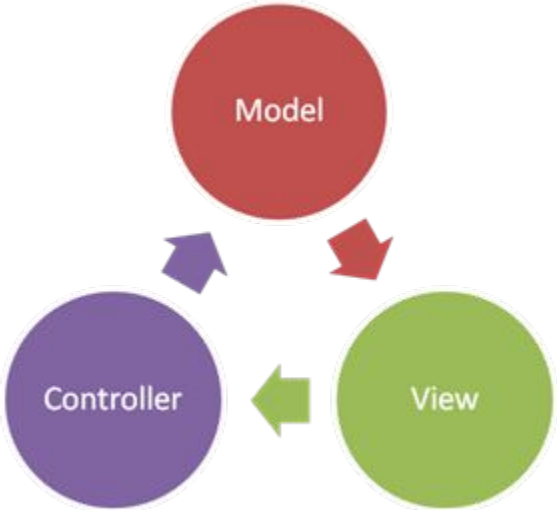
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioural Patterns

Behavioural patterns define manners of communication between classes and objects.

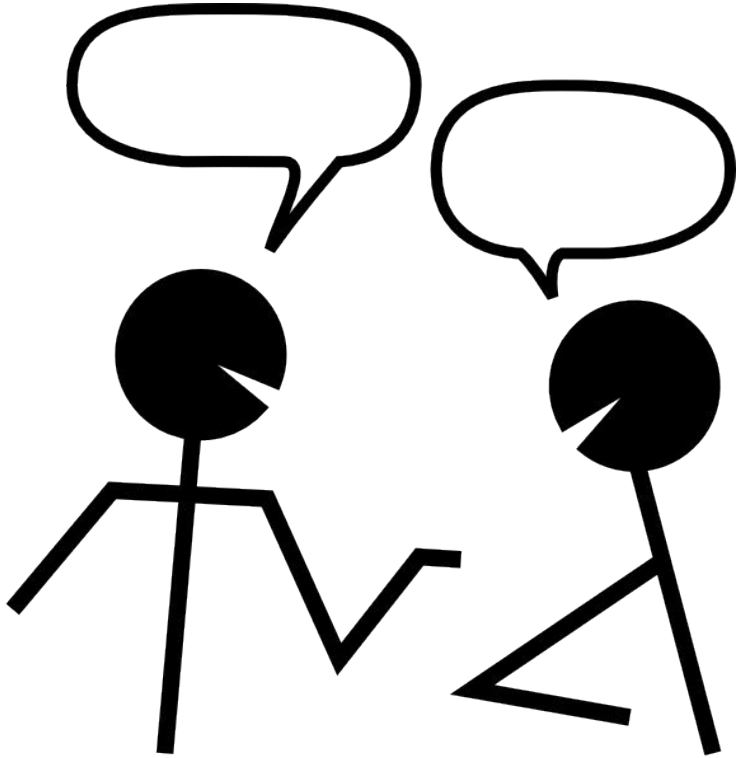
- Command
- Interpreter
- Iterator
- Mediator
- Template Method
- Null Object
- Observer
- State
- Strategy
- Visitor
- Memento

An important guest is missing: MVC!



What we will speak about

- Real cases where a pattern would have helped:
  - TodoMVC: jQuery example without M.V.C.
  - java.io.File: NullObject instead of null checks.
- Adapter, Decorator, Proxy: are they the same pattern?



MVC: ToDo app

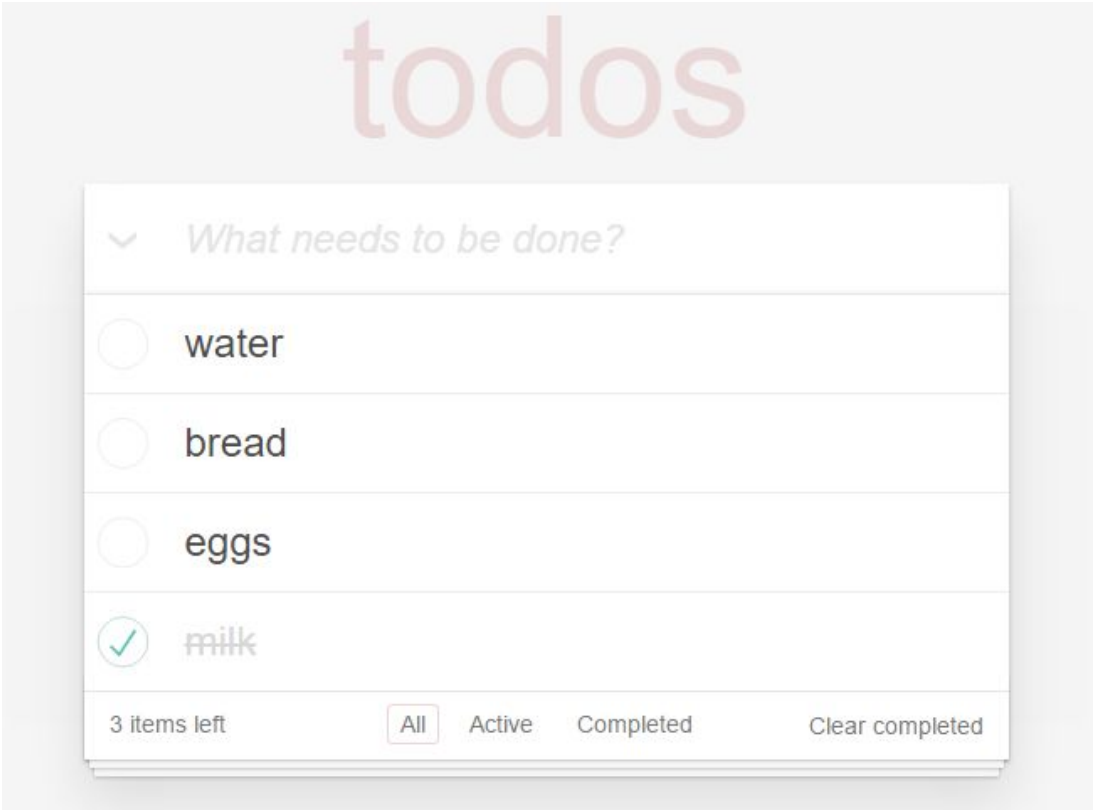
A real example taken from http://todomvc.com/

In the jquery example, MVC is violated.

Let’s analyze what should happen when a user ticks off an item on the list.

MVC: ToDo app, jQuery implementation

```
var App = {
        ...
        bindEvents: function () {...},
        render: function () {...},
        indexFromEl: function (el) {...},
        create: function (e) {...},
        toggle: function (e) {...},
        update: function (e) {...},
        ...
};
```



MVC: ToDo app, jQuery implementation

```
var App = {
        ...
        bindEvents: function () {...},
        render: function () {...},
        indexFromEl: function (el) {...},
        create: function (e) {...},
        toggle: function (e) {...},
        update: function (e) {...},
        ...
};
```

```
toggle: function (e) {
      var i = this.indexFromEl(e.target);
      this.todos[i].completed = !this.todos[i].completed;
      this.render();
},
```

MVC: ToDo app, jQuery implementation

```
toggle: function (e) {
      var i = this.indexFromEl(e.target);
      this.todos[i].completed =
            !this.todos[i].completed;
      this.render();
},
```

Considerations:

- The code that handles the browser event fired when the user checks off an item, also handles the model update.
- There is a weird `this.indexFromEl(e.target);`
- `this.indexFromEl(...)` is O(n), which is not necessary
- What if we want to programmatically mark items as done?

Can we do better? Yes...

ToDo App, a better M.V.C.

Model

In the previous example a JS array is the model.

In a real application, an array may be not enough, but it is good enough for this example.

View

In the previous example the code reacts directly to the browser events, without any distinction between the view and the controller.

The view is responsible to listen to events, and notify the controller that the User did something.

Controller

The controller actually implements the behaviour of the application.

Doing this means also updating the model and notifying to the view the updated model.

ToDo App, a better M.V.C.

```
class Controller {

init(view) {
     view.onItemTickClick((index) => {
          this._model[index].done =
               !this._model[index].done;
          view.render(this._model);
     });
}
```

```
class View {
onItemMarkClick(callBack) {
     this._tickCallback = callBack;
}
render(itemList) {
     this._clearList();
     itemList.forEach((item, index) => {
          this._createItem(item, index);
     });
}
_createItem(item, index) {
// Creates html elements, uses _tickCallback
// to notify the item has been ticked off
```

MVC: ToDo app, jQuery implementation

Int the original jQuery implementation:

- has unnecessary code
- is tied with user interface code, making difficult to programmatically command the application (Eg. a wizard)
- is hard to test

java.io.File, from OpenJDK

A piece of code from java.io.File implementation

```
public boolean exists() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
                security.checkRead(path);
        }
        if (isInvalid()) {
                return false;
        }
        return ((fs.getBooleanAttributes(this) & FileSystem.BA_EXISTS) != 0);
}
```

The line `if (security != null)` is repeated 25 times!`if (security != null) {`

java.io.File, from OpenJDK

If we change the System.getSecurityManager(); implementation from

```
public static SecurityManager getSecurityManager() {
        return security;
}
```

to

```
public static SecurityManager getSecurityManager() {
        return security != null ? security : NoSecurity.getInstance();
}
```

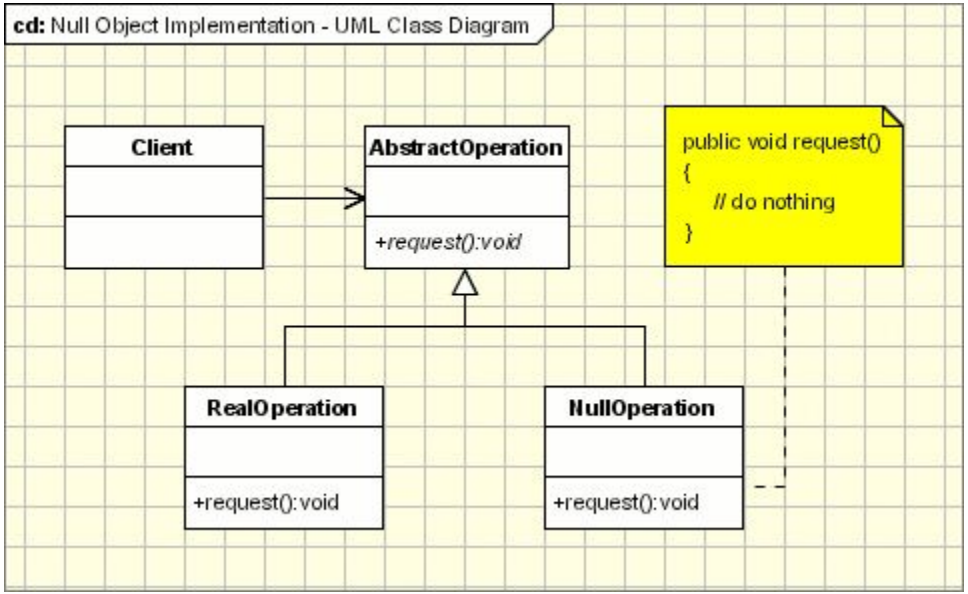
we can refactor the exist method into:

```
public boolean exists() {
        System.getSecurityManager().checkRead(path);

        if (isInvalid()) {
                return false;
        }
        return ((fs.getBooleanAttributes(this) & FileSystem.BA_EXISTS) != 0);
}
```

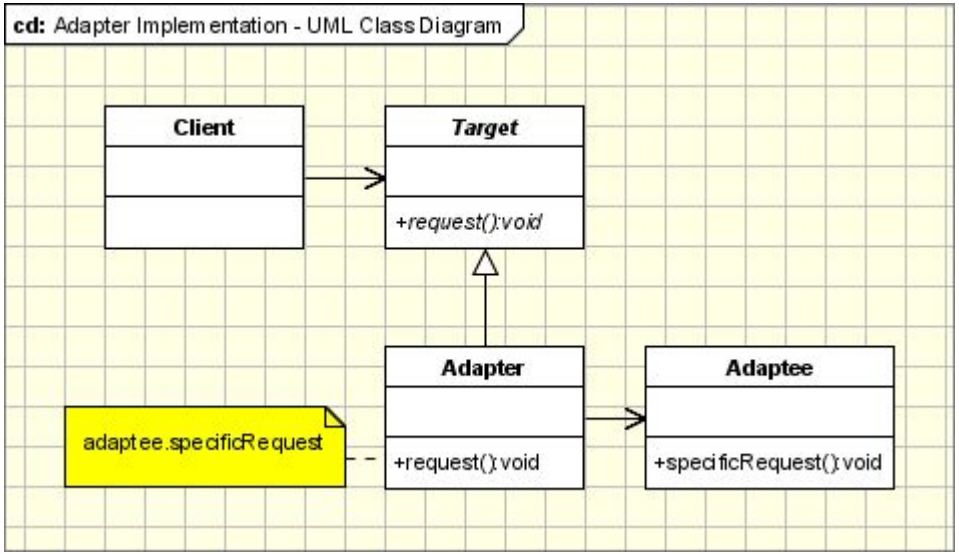
NullObject pattern

NoSecurity is an example of the NullObject pattern:



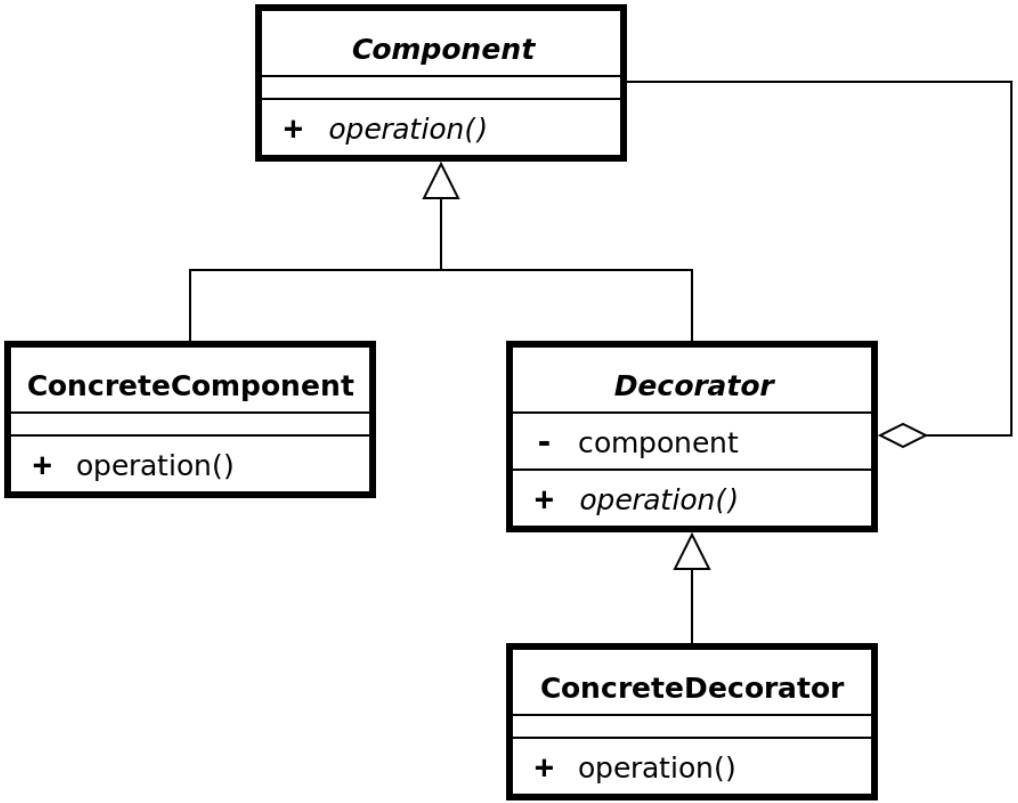
Adapter

Convert the interface of a class into another interface clients expect.



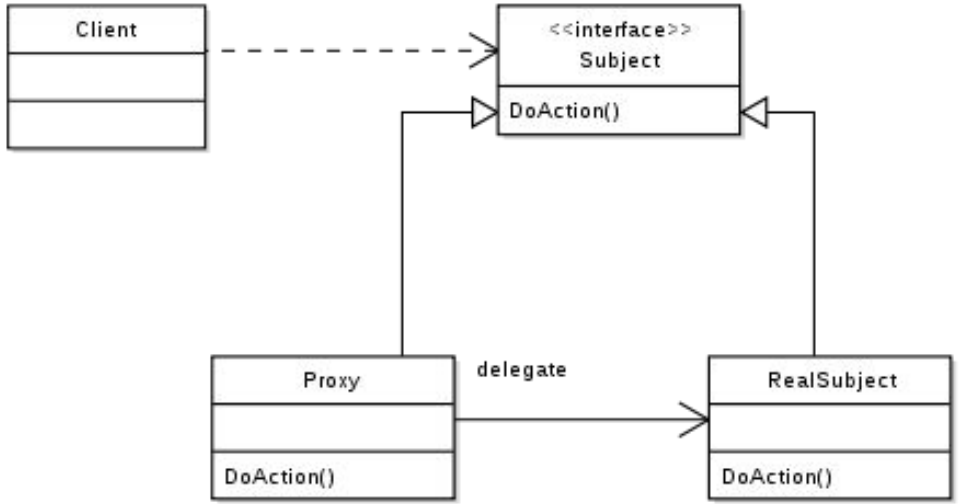
Decorator

Adds additional responsibilities dynamically to an object.



Proxy

Provides a surrogate or a Placeholder for an object to control references to it.



Adapter vs Decorator vs Proxy

A: a class, **B**: another class, $\mathbf{A}_{\mathbf{sub}}$: a subclass of A, $\mathbf{B}_{\mathbf{sub}}$: a subclass of B

Adapter: A -> B adapts from a A to B

Decorator: A -> A wraps A to dynamically extends its behavior

Proxy: $\mathbf{A}_{\mathbf{sub}}$ -> $\mathbf{B}_{\mathbf{sub}}$ limits access to $\mathbf{B}_{\mathbf{sub}}$


THANK
YOU
return