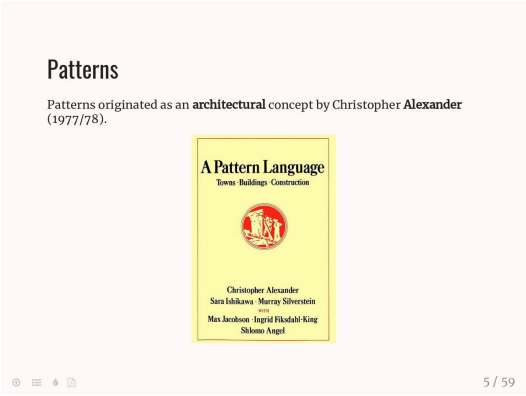
Design Patterns #1

Introdução

Hoje vamos finalmente começar a falar sobre uma das componentes mais importantes desta unidade curricular: os **design patterns**.



Antes de começarmos a falar dos padrões propriamente ditos, um bocado de história.

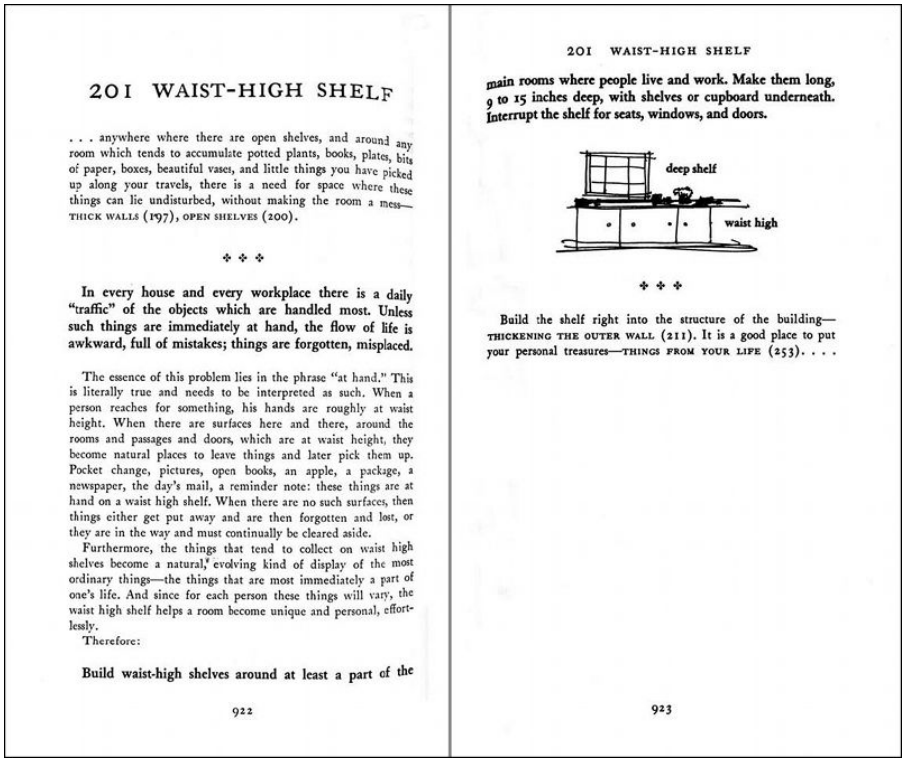
Em 1977, o arquitecto **Christopher Alexander** escreveu um livro (“A Pattern Language: Towns, Buildings, Construction”) contendo um conjunto de padrões sobre **arquitectura de edíficios e cidades, e design urbano**.

Mas mais do que o conteúdo em si, inventou um novo conceito: o de uma **linguagem de padrões**.

Neste livro, cada padrão apresenta um **problema comum** em arquitectura e uma **solução genérica** para esse problema.

Cada padrão tem um **nome** e um **número**, e está **interligado** com vários outros padrões.

Podemos ver aqui um exemplo de um destes padrões:



201 WAIST-HIGH SHELF

. . . anywhere where there are open shelves, and around any room which tends to accumulate potted plants, books, plates, bits of paper, boxes, beautiful vases, and little things you have picked up along your travels, there is a need for space where these things can lie undisturbed, without making the room a mess—THICK WALLS (197), OPEN SHELVES (200).

❖ ❖ ❖

In every house and every workplace there is a daily "traffic" of the objects which are handled most. Unless such things are immediately at hand, the flow of life is awkward, full of mistakes; things are forgotten, misplaced.

The essence of this problem lies in the phrase "at hand." This is literally true and needs to be interpreted as such. When a person reaches for something, his hands are roughly at waist height. When there are surfaces here and there, around the rooms and passages and doors, which are at waist height, they become natural places to leave things and later pick them up. Pocket change, pictures, open books, an apple, a package, a newspaper, the day's mail, a reminder note: these things are at hand on a waist high shelf. When there are no such surfaces, then things either get put away and are then forgotten and lost, or they are in the way and must continually be cleared aside.

Furthermore, the things that tend to collect on waist high shelves become a natural, evolving kind of display of the most ordinary things—the things that are most immediately a part of one's life. And since for each person these things will vary, the waist high shelf helps a room become unique and personal, effortlessly.

Therefore:

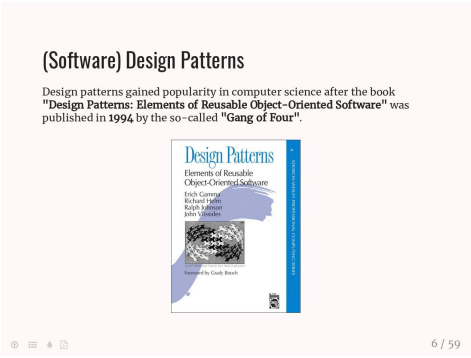
Build waist-high shelves around at least a part of the

main rooms where people live and work. Make them long, 9 to 15 inches deep, with shelves or cupboard underneath. Interrupt the shelf for seats, windows, and doors.

❖ ❖ ❖

Build the shelf right into the structure of the building—THICKENING THE OUTER WALL (211). It is a good place to put your personal treasures—THINGS FROM YOUR LIFE (253). . . .

O livro **não** é só um conjunto de receitas, mas uma nova linguagem. Cada padrão é apresentado **não** como a solução definitiva para o problema mas como uma **template** que tem as suas vantagens e desvantagens — **forças** que têm de ser tomadas em consideração, cada qual a puxar para o seu lado — e tem de ser **adaptada** a um **problema específico**.

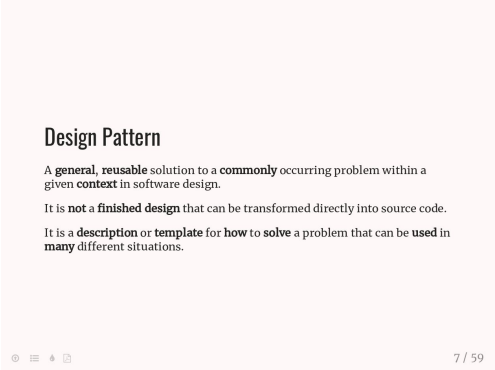


Em 1994, surge um novo livro, com **forte inspiração** no livro do Christopher Alexander, com o título "**Design Patterns: Elements of Reusable Object-Oriented Software**".

Este livro, que é considerado um livro fundamental para qualquer engenheiro informático, contém 23 padrões clássicos e foi escrito pelos:

- Erich Gamma (um dos criadores do JUnit que já usaram para implementar testes unitários)
- Richard Helm
- Ralph Johnson
- John Vlissides
- E com um prefácio do Grady Booch (um dos criadores do UML).

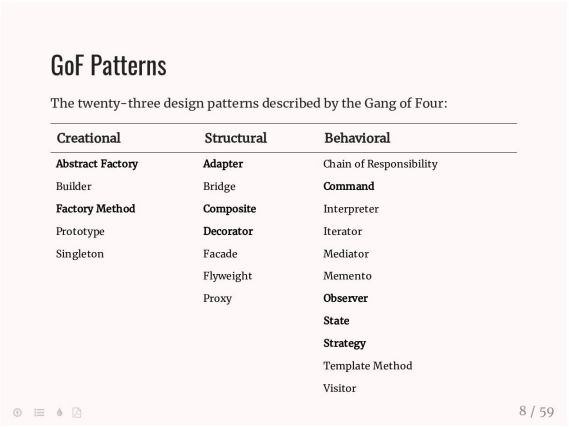
Juntos os 4 autores são conhecidos como o "Gang of Four" ou apenas **GoF**.



É fundamental perceber, que estes padrões **não são receitas**, **bibliotecas**, ou **código** que possa ser simplesmente copiado para os vossos projectos. Mas antes, **sugestões** de como resolver problemas **comuns** e **recorrentes**.

Design Pattern: "*A **general**, **reusable** solution to a **commonly occurring** problem within a given **context** in software design.*"

É também importante perceber que, normalmente, quem escreve o padrão, não é quem inventou a solução (*a solução já foi reinventada várias vezes, e os padrões servem exactamente para deixarmos de ter de reinventar a roda*). Apenas se apercebeu que a solução que está a descrever é uma solução recorrente para um problema comum e que pode, e deve, ser considerada um padrão.



Estes são os 23 padrões descritos pelo GoF. Como podem ver, dividem-se em 3 categorias:

- **Creacionais**, que se preocupam com diferentes formas de **criar objectos**, que não a instanciação directa.
- **Estruturais**, que se preocupam com a forma como os objectos são **organizados** e **compostos** de maneira a criar **estruturas** úteis.
- E **comportamentais**, que se preocupam com a forma como os objectos **interagem** e comunicam.

Podemos ver como é que os **padrões** se **interligam** nesta imagem:

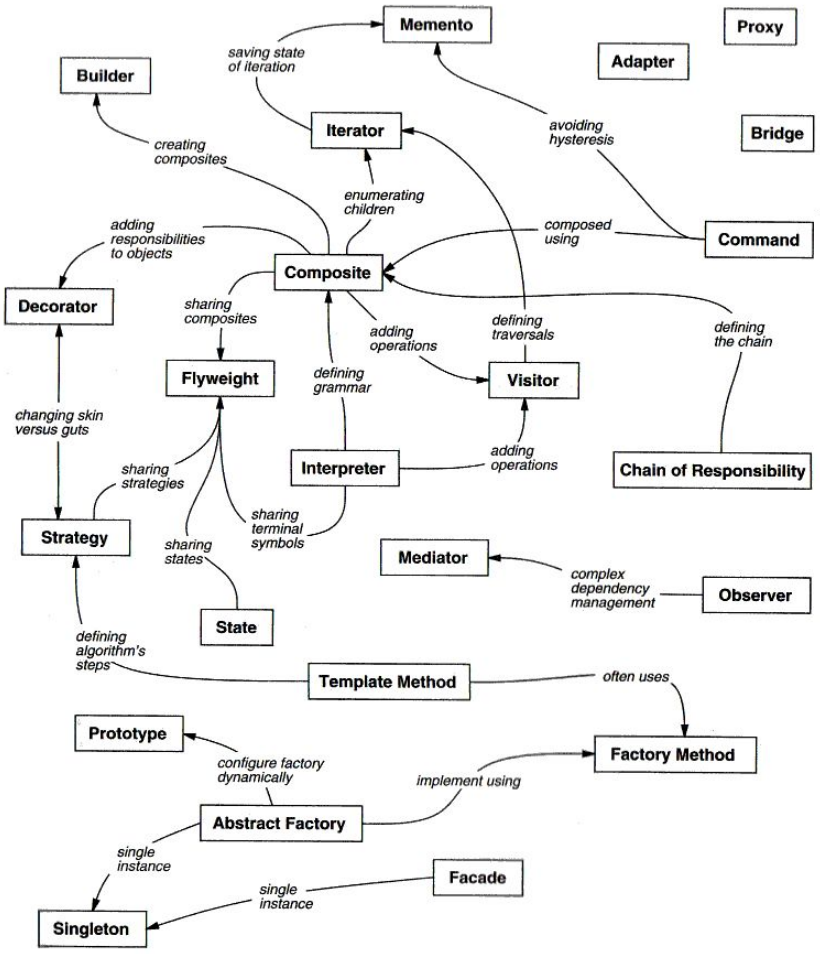
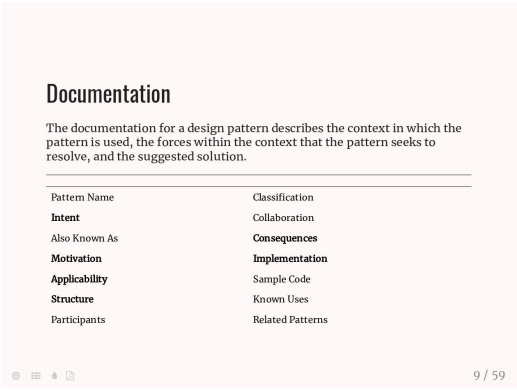


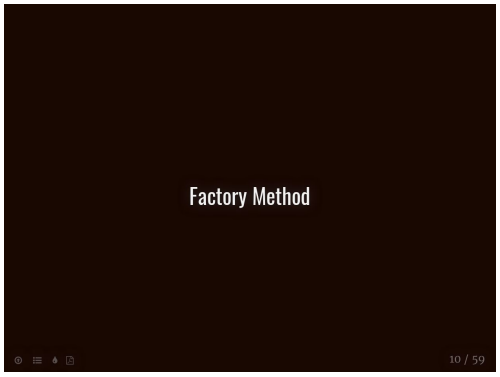
Figure 1.1: Design pattern relationships



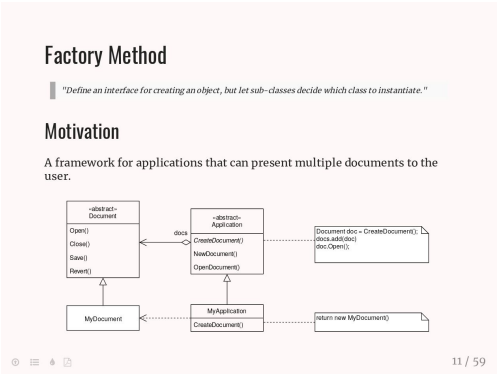
A descrição de cada um destes padrões contém os mesmos elementos:

- **Nome do Padrão**, um elemento fundamental que permite que o padrão se torne parte da linguagem corrente.

- **Intenção**, de um forma muito concisa, qual a intenção do padrão.
- **Também conhecido como**, nomes alternativos pelo qual o padrão é conhecido.
- **Motivação**, um exemplo motivacional, simples mas concreto que nos ajuda a perceber em que casos o padrão é útil.
- **Aplicabilidade**, em que situações é que o padrão pode ser implementado.
- **Estrutura**, a estrutura genérica do padrão de uma forma abstracta (ou seja sem se focar num problema específico).
- **Participantes**, uma descrição mais detalhada das classes participantes no padrão.
- **Classificação**, qual o tipo de padrão.
- **Colaboração**, uma descrição de como as classes ou objectos colaboram entre si de forma a realizar o padrão.
- **Consequências**, quais as vantagens e **desvantagens** de se usar este padrão. O que é que estamos a ganhar e o que é que estamos a perder. Quais os efeitos secundários. Quais são as forças.
- **Implementação**, a solução, ou como implementar o padrão.
- **Código Exemplo**, algum código para exemplificar a implementação.
- **Usos Conhecidos**, casos reais de utilização do padrão.
- **Padrões Relacionados**, padrões que podem ser usados em alternativa ou em colaboração com este padrão.



Durante esta *“aula”*, vamos explorar 5 padrões diferentes, começando pelo padrão **Factory Method**.



De vez em quando queremos que uma superclasse permita instanciar objectos de **um certo tipo** (ou seja, que implementem uma certa interface), mas queremos que sejam as **subclasses a decidir qual o tipo específico de objectos** que queremos que seja instanciado.

Como exemplo **motivacional**, vamos imaginar uma *framework* para **edição de documentos** que pode ser ***extendida*** para ser usada com **qualquer tipo** de documentos.

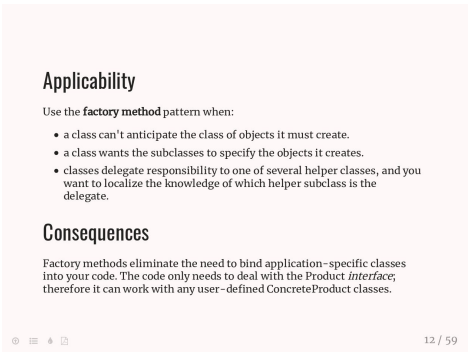
Na parte superior do diagrama, podemos ver que a **framework** é composta por **duas classes abstractas**: a *aplicação* e o *documento*.

Do lado direito temos uma **anotação** UML (papel com o canto dobrado) que mostra o código do método ***NewDocument()***. Como podemos ver, este método **chama** o método ***CreateDocument()*** para criar um **novo documento**.

Mas a classe abstracta *Application* **não sabe que tipo de documento deve criar** (depende da aplicação concreta). Por esse motivo o método *CreateDocument()* é **abstracto**. As classes derivadas vão ser obrigadas a implementar esse método.

A parte de baixo do diagrama mostra uma aplicação (*MyApplication*) baseada nesta framework, e que **implementa** o método *CreateDocument()* de forma a que um documento do tipo *MyDocument* seja criado.

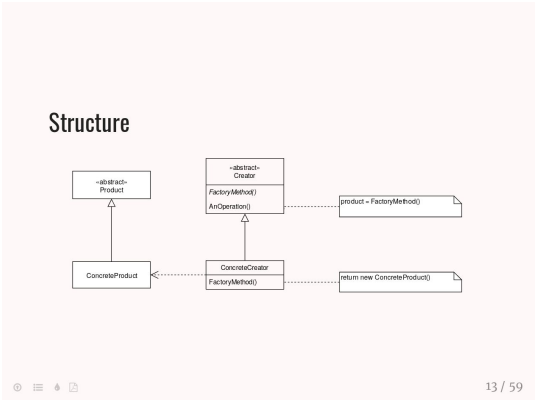
Desta forma, a framework **não tem de ser modificada** sempre que um novo tipo de aplicação é adicionado. Por exemplo, com métodos diferentes para cada aplicação criada, ou um *switch case* que escolhe o tipo de documento a criar.



Devemos **usar este padrão** sempre que:

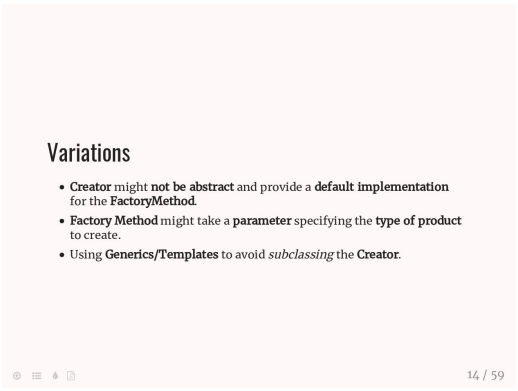
- Uma classe **não tem forma de saber** que tipo de objecto instanciar.
- Uma classe **quer que sejam** as subclasses a decidir que tipo de objecto instanciar.
- Uma classe delega a decisão a classes *helper* mas são as subclasses que sabem **que classe** *helper* **deve ser usada**.

Como **consequência**, deixamos de ter de introduzir conhecimento sobre classes específicas a cada caso de uso, em código que não precisa de as conhecer.



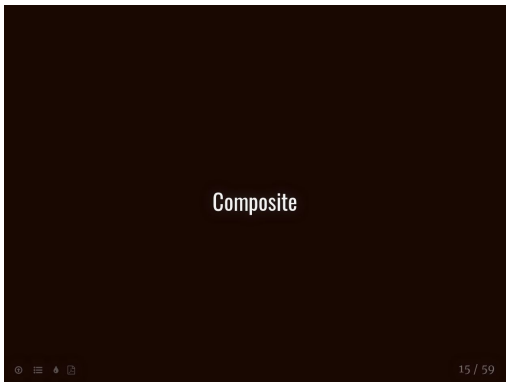
A estrutura abstracta do padrão é a mesma do exemplo mas em vez de chamarmos às classes *Application* e *Document*, chamamos ***Creator*** e ***Product***.

As subclasses concretas são a ***ConcreteCreator*** e ***ConcreteProduct***. Estes nomes representam o **papel** que cada uma das classes do vosso código, caso usem este padrão, vai tomar.

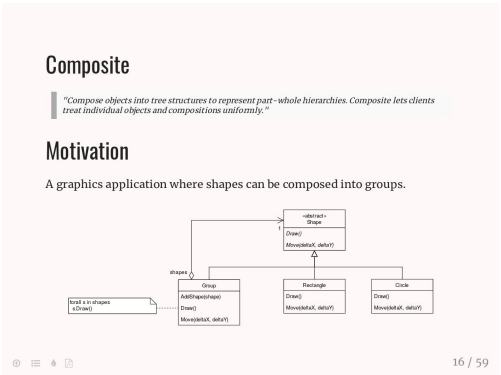


Algumas variações do padrão (só algumas):

- A classe *Creator* pode **não ser abstracta** e definir um tipo de objectos como sendo o tipo criado por omissão.
- O método que cria objectos pode receber um **parâmetro** que especifica o tipo de objectos a criar (ou outro tipo de informação).
- Há a possibilidade de usar **genéricos** (o mesmo que templates de C++) para evitar ter de criar subclasses da classe *Creator*. Vamos ver isto mais tarde...



Vamos passar agora ao segundo padrão: o **Composite**.



Muitas vezes precisamos não só que um objecto **contenha vários objectos** de um determinado tipo, mas que também possa ter **objectos do seu próprio tipo**.

Como exemplo **motivacional**, imaginemos uma aplicação de **desenho vectorial** (tipo o draw.io, o inkscape ou o illustrator).

Já vimos várias vezes o exemplo de termos uma classe ***Shape* abstracta** que tem várias subclasses que representam ***shapes* concrectas** (rectângulo, círculo, triângulo, …).

A maior parte destas aplicações permite que a gente faça **grupos** com estas shapes que depois podem ser **alterados em conjunto** (translação, alteração da cor, rotação, …).

Para isso podemos criar uma **class *Group*** que é uma **agregação** (♢) de *Shapes*.

Mas agora temos **dois problemas**:

1. As **operações** que se pode fazer a um grupo **são as mesmas** que se pode fazer a uma *shape*.
2. Se calhar gostaríamos de poder fazer **grupos de grupos**, tal como fazemos grupos de shapes.

Isto tudo indica que se calhar a classe ***Group*** também devia ser **subclasse da classe *Shape***.

Dessa forma, tudo o que **posso fazer** com uma *Shape*, **também posso fazer** com um *Group*.

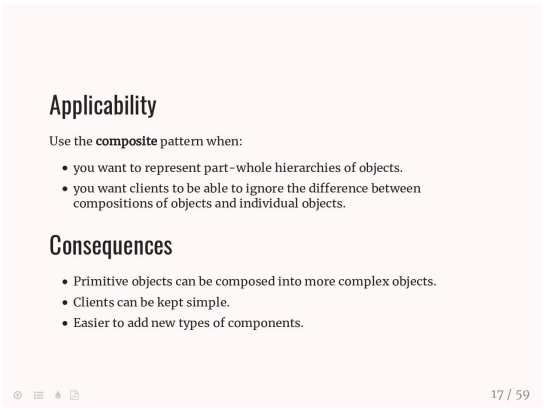
E qualquer método que **receba** uma *Shape* **pode receber** um *Group.*

Para além disso, como um *Group* é uma agregação de *Shapes*, e um *Group* **é** uma *Shape*, os *Groups* passam a poder ter *Groups* lá dentro. Ficamos assim com uma **árvore de Shapes**.

Vamos imaginar como poderia ser o método ***move(deltax, deltay)*** de um *Group*:

(alguém quer tentar)

```
public void move(int deltax, int deltay) {
  for (Shape shape : shapes)
    shape.move(deltax, deltay);
}
```

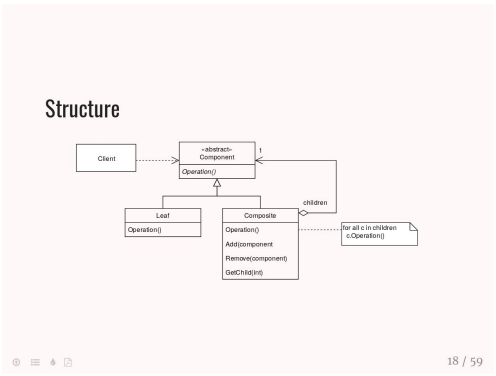


Devemos **usar** este padrão quando:

- Queremos estabelecer uma **hierarquia de objectos** em árvore.
- Queremos que o **cliente** dessa hierarquia possa **ignorar** se está a trabalhar com **um só elemento** da hierarquia, ou com uma **composição** de elementos.

As **consequências** de usar este padrão são:

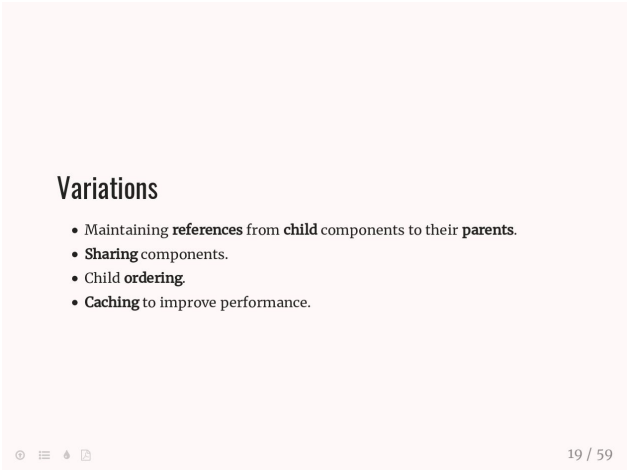
- Podemos criar **objectos** mais **complexos** criando grupos de objectos.
- Os **clientes**, ou seja, quem usa os objectos, ficam mais **simples** porque **não** se têm de **preocupar** com saber se estão a trabalhar com um grupo ou com um objecto isolado.
- Torna-se **simples** adicionar **novos tipos** de objectos.



A **estrutura** abstrata do padrão é similar à do exemplo. A classe ***Client*** representa **qualquer classe** que use a árvore de objectos.

Em vez de *Shape*, dado que isto pode ser usado em outros casos que não aplicações de desenho, temos a class ***Component***.

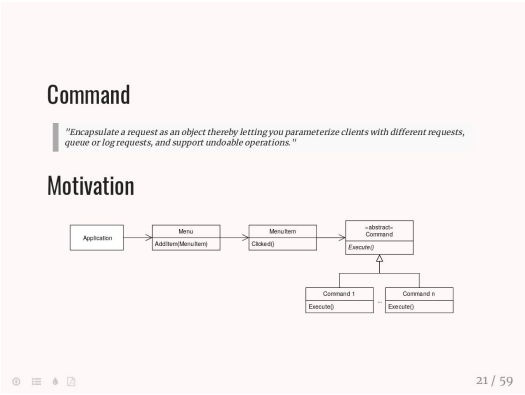
A cada um dos componentes chamamos ***Leaf***. E ao grupo chamamos ***Composite*** (que é o nome do padrão).



Podemos ainda considerar várias **variações** a este padrão:

- Podemos querer que as *Leafs* saibam **de que grupo** fazem parte. Pode dar jeito em alguns casos.
- Podemos querer que grupos (*Composites*) **partilhem** *Components*.
- Pode ser necessário saber a **ordem** dos filhos de um *Composite*.
- E em alguns casos, pode ser útil poder fazer ***cache*** do resultado de operações. Por exemplo, a classe *Shape* pode saber calcular a área, e a área de um *Group* ser a soma das áreas das *Shapes*. Se tivermos uma árvore muito profunda, este cálculo pode tornar-se lento. Mas os *Groups* podem **guardar** a área das *Shapes* e só **recalcular** se as *Shapes* mudarem.

Vamos agora ver o próximo padrão que se chama **Command**. Este padrão é extremamente simples mas abre a porta a muitas possibilidades.



Este padrão usa-se quando queremos encapsular um pedido num objecto de forma a que este possa ser **parametrizável**, que um objecto possa ser parametrizado com um pedido, que os pedidos possam estar numa **lista de espera**, possam ser **repetíveis** ou **desfazíveis** (operação de *undo*), …

Reparem que quando falo em **pedido**, estou a falar de, tipicamente, uma **chamada a uma função**.

Imaginem que têm uma classe com um método *execute()* e que querem que quando esse método for chamado, algo aconteça, mas querem poder ser vocês, como **cliente** da classe, a **decidir** o que **acontece** quando o método é chamado.

Como exemplo **motivacional**, vamos imaginar uma aplicação de *desktop* com um ménu (classe *Menu*). A esse ménu podemos adicionar vários **items** (classe *MenuItem*) e queremos que cada um desses items faça algo **diferente** quando a sua função *clicked()* é **executada**.

Podemos, obviamente, ter várias subclasses de *MenuItem*, cada uma com a sua **implementação** do método *clicked()*.

Mas agora imaginemos que queremos que a operação que o *MenuItem* executa possa ser executada de **outra forma**. Por exemplo, através de um *shortcut*, ou noutro sítio da interface gráfica.

Ou imaginemos que queremos criar comandos que não são mais do que uma **composição** de dois comandos. Por exemplo: *Open* = *New* + *Load*, ou *Save and Quit* = *Save* + *Quit*, …

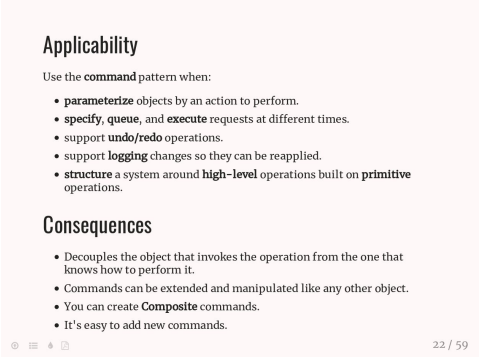
Ou que queremos que os utilizadores possam fazer ***undo*** das operações. **Onde** é que guardamos cada uma das operações que já foi feita?

A solução para isto é criar uma classe *abstract* chamada ***Command***, que possa ser **extendida** de forma a criarmos os vários comandos que a aplicação aceita como **subclasses**.

Cada *MenuItem* depois pode ter um ou vários **Commands** associados, e executar cada um deles.

Podemos guardar a **lista** de *Commands* que foi **executada** numa *Stack* e deixar que cada comando saiba como fazer ***undo*** da sua operação.

Podemos ter comandos que são uma **composição** de vários comandos. Podemos até juntar este padrão ao padrão ***Composite*** e ter comandos que são **composições de composições** de comandos. As hipóteses são imensas.

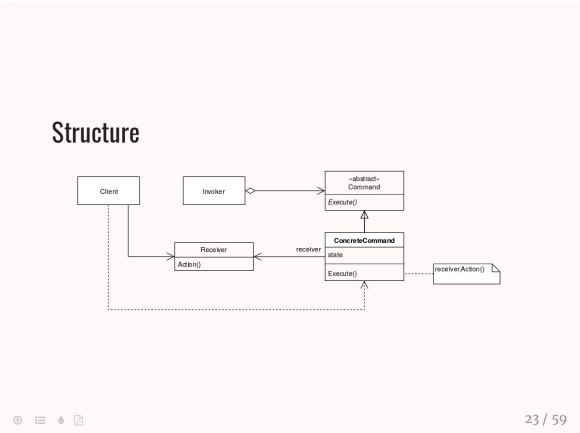


Devemos **usar** este padrão quando:

- Queremos **parametrizar** os **objectos** com uma acção a ser executada.
- Especificar a acção num determinado sítio mas só a executar **mais tarde**. Por exemplo, colocando as acções numa **fila de processamento**.
- Suportar operações de ***undo*** ou ***redo***.
- Ter um **log** das alterações efectuadas de forma a que as possamos **voltar a fazer**. Útil por exemplo caso algo corra mal num comando e tenhamos perdido as alterações efectuadas. Vários sistemas tolerantes a falhas usam sistemas deste género.
- Implementar sistemas baseados em operações de baixo-nível muito **simples** que depois são **compostas** em operações mais **complexas** (padrão *Composite*).

As **consequências** de usarmos este padrão são:

- **Separamos as responsabilidades** de saber quando executar um comando e como executar o comando (*Single Responsibility Principle*).
- Os comandos podem ser **extendidos** e manipulados como qualquer outro objecto. Por exemplo o comando *SaveAs* pode ser derivado do comando *Save*. Ou um comando pode ser parametrizado com o nome do ficheiro, …
- Torna-se fácil ter comandos **compostos**.
- Torna-se fácil **adicionar** novos comandos.



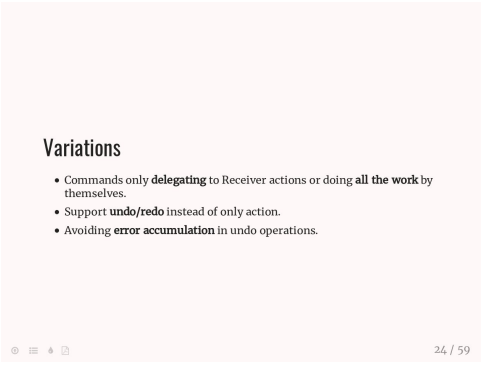
A **estrutura** abstracta deste padrão pode parecer um pouco mais complexa, mas não é. Vamos partir isto em bocados mais simples.

Do lado direito temos a classe abstracta ***Command*** e um ***ConcreteCommand*** que é uma implementação de um comando **específico** (por exemplo *SaveCommand*). Reparem que o comando concreto pode ter um estado interno (por exemplo para ajudar a fazer *undo*).

É normal que cada comando tenha de saber **em que objecto** vai actuar. Por essa razão, é normal os comandos estarem associados a um ***Receiver***. É também normal que esse *receiver* seja passado ao comando no **construtor** (por exemplo: *new CloseCommand(document)*).

O ***Client*** é a class que **cria** o comando, e o ***Invoker*** é a classe que depois o vai **executar**.

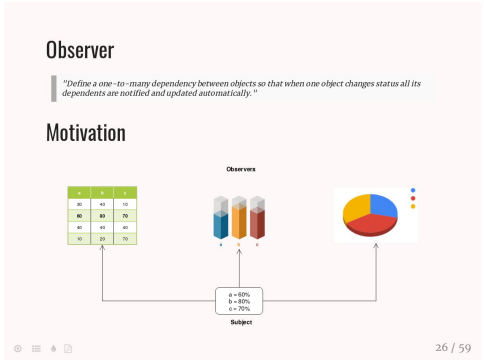
Reparem que **nenhuma** destas classes (*Client*, *Invoker* e *Receiver*) tem de **existir**. São só representações **abstractas** de classes que vão existir no vosso código (por exemplo, *MenuItem* ou *Application*)



Algumas (só algumas) da variações possíveis para este padrão:

- O comando **não** ser o **responsável** por executar realmente a acção mas apenas **delegar** no método correto do *Receiver*. Ou seja, o método *execute()* de cada comando realmente ter apenas **uma linha**.
- Suportar operações de *undo* e *redo*, acrescentando um método *undo()* ao comando.
- Garantir que erros (*bugs*) em undos múltiplos **não** se **propagam** pelos vários undos a serem executados (guardando por exemplo o **estado** do *Receiver* e recusando-se a fazer *undo* se o estado for diferente).

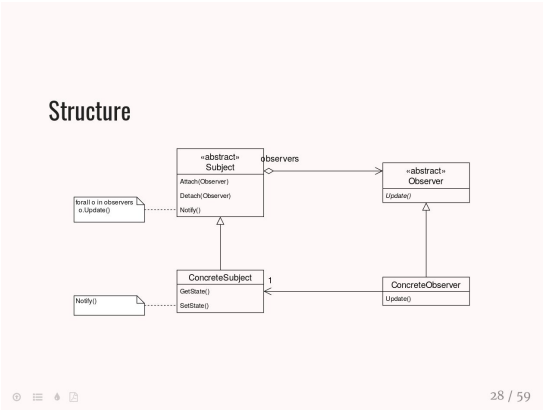
Vamos ver agora um dos padrões mais usados em linguagens orientadas a objectos, o padrão **Observer**. Também conhecido como **Listener** ou **Publisher/Subscriber**.



Para este padrão vamos ter uma **motivação** um bocado diferente. Em vez de termos um exemplo em UML, imaginem apenas que têm alguns **dados** (guardados algures numa ou mais classes) e que querem **mostrar** os dados de **formas diferentes** na interface gráfica.

Será que são as classes que **guardam os dados** que devem ser **responsáveis** por **avisar** as classes responsáveis pela **visualização** sempre que os **dados mudarem**?

O que acontece se tivermos de acrescentar uma nova classe de visualização. Lá vamos nós ter de alterar a classe que guarda os dados (uma clara violação do *Open Closed Principle*).



Como o nosso exemplo não tinha um diagrama UML, desta vez vamos ver primeiro a estrutura do padrão.

Em relação ao nosso exemplo, as vistas são os ***Observers*** e a estrutura de dados é o ***Subject***.

Começamos por declarar duas classes **abstractas**: *Subject* e *Observer*.

Ao *Subject* podem ser **adicionados** *Observers*, para além disso, um *Subject* tem um método concreto (***Notify()***) que **notifica** todos os *Observers*.

Um Observer tem um método (***Update()***) que é chamado pelo *Subject* sempre que este for alterado.

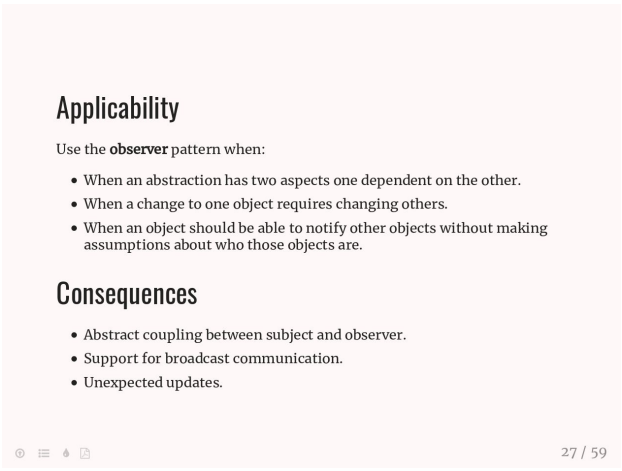
Para usar estas duas classes, basta *extendê-las*. No ***ConcrectSubject***, sempre que o estado é alterado (por exemplo, porque um método ***set…()*** foi chamado) basta chamar o ***Notify()***. O Update() do ***ConcreteObserver*** é automaticamente chamado.

Exemplo de utilização:

```
ConcreteSubject subject = new ConcreteSubject();
ConcreteObserver observer = new ConcreteObserver();
subject.attach(observer);
subject.changeSomething();
```

Quando o método ***attach()*** é chamado, o *observer* é adicionado a uma lista de *observers*.

Quando o método ***changeSomething()*** é chamado, o método ***notify()*** é invocado de forma a que o método ***update()*** de todos os observers também seja chamado.

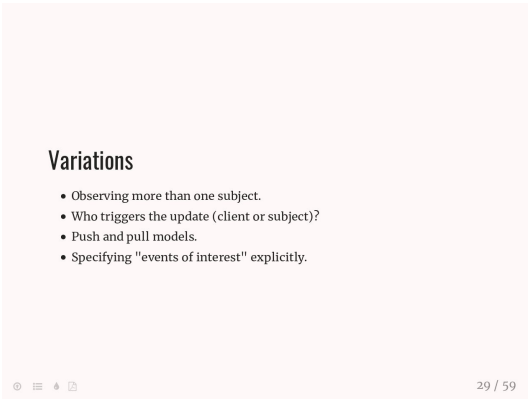


Este padrão **usa-se** quando:

- Uma abstração tem **dois aspectos** em que um **depende** do outro.
- Quando uma **mudança** num objecto implica **alterar** outros.
- Quando um objecto pretende **notificar** outros de alterações **sem ter de saber** que tipo de objectos está a **notificar**.

As **consequências** da utilização deste padrão são:

- O **acoplamento** entre o sujeito e o observador passa a ser **abstracto**. Ou seja, não implica que saibam qual o tipo concreto um do outro.
- Passa a ser possível ter comunicação por **broadcast**, ou seja, um sujeito notificar facilmente **vários** observadores.
- Como os observadores não têm conhecimento um dos outros, não existe a noção do **custo**/**resultado** de actualizar um objecto.

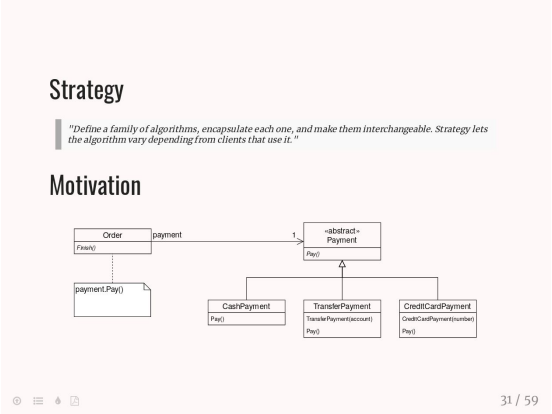


Variações a este padrão incluem:

- Observadores que conseguem observar **mais do que um** *subject*.

- Quem é **responsável** por fazer chamar o método **update()**, o próprio *subject*? Ou deve ser o **cliente** (que o modificou) a **decidir** se deve ou não **notificar** todos os *subjects*.
- O *subject* deve enviar as modificações para o observador (**push**), ou deve ser o observador a pedir ao *subject* qual o novo estado (**pull**)?
- O observador poder declarar em que **tipos** de alterações está **interessado**. Útil quando o *subject* pode ser modificado de **variadas formas** mas só nos interessa **algumas** delas.

Finalmente o último padrão de hoje: o **Strategy**.



O padrão **Strategy** permite **encapsular algoritmos** dentro de objectos. Isto permite que o algoritmo usado pelo cliente possa ser **parametrizado** sem ser necessário alterá-lo ou *extendê-lo*.

Como **motivação**, vamos imaginar que temos uma classe ***Order*** que representa uma **encomenda**. E que a encomenda pode ter **várias formas de pagamento** (e.g., dinheiro, transferência bancária ou por cartão de crédito).

Podemos criar uma classe abstracta que representa uma estratégia de pagamento (**Payment**) que depois é especializada nos **vários tipos de pagamentos** possíveis. Desta forma, basta-nos passar à encomenda (por exemplo no constuctor) qual a estratégia que deve ser utilizada, e depois, quando o método ***pay()*** da encomenda é invocado, basta-nos chamar o método ***pay()*** da estratégia.

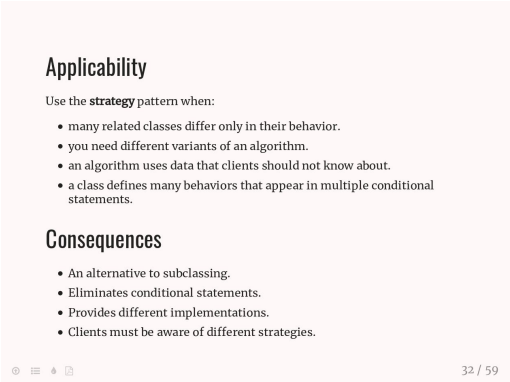
Exemplo de código:

```
Order order = new Order(new CashPayment());
order.pay();
```

Quando o método ***pay()*** da encomenda é chamado, a encomenda **delega** a execução à sua **estratégia** de pagamento (neste caso ao ***CashPayment***).

Porque é que não podemos simplesmente *extender* a classe Order e criar 3 tipos de Order? Porque isso iria, em primeiro lugar, violar o *Single Responsibility Principle*.

Mas há **mais razões**. Imaginem que para além de pagamentos diferentes, a encomenda tinha outro método ***shipping()*** que tratava da **forma de entrega** da encomenda. E que podia haver **dois tipos de entrega** (recolha na loja e entrega por correio). Agora em vez de termos **três** subclasses de Order, precisamos de **seis**!!!



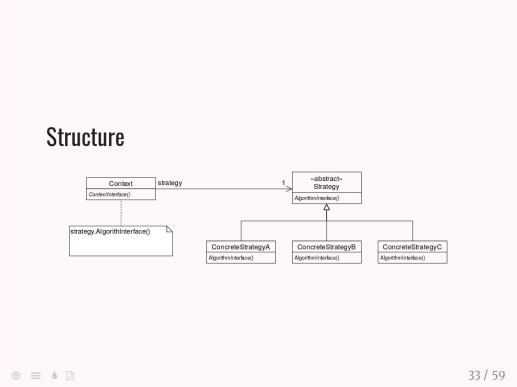
Este padrão deve ser **usado** quando:

- Muitas classes relacionadas **diferem apenas no algoritmo**.
- Pode haver **várias variantes** de um algoritmo.
- Um algoritmo usa **dados** que **não queremos** que o cliente **conheça**.
- Uma classe define **vários comportamentos** que aparecem em vários ***switch cases***.

As **consequências** de usar este padrão são:

- Pode ser usado como uma **alternativa** a criar **subclasses** de uma classe cliente.

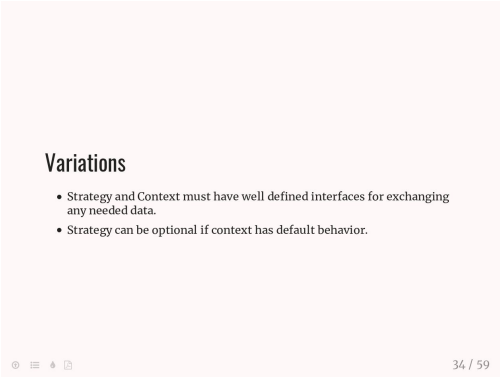
- **Elimina *switch cases*** complexos.
- Permite ter **várias implementações** alternativas de um **algoritmo**.
- Os clientes **têm de saber** que quais as estratégias que existem.



A **estrutura** abstracta deste padrão, dá o nome de ***Context*** à classe cliente da estratégia.

Essa classe terá uma forma de lhe ser **passada** a estratégia (por exemplo no constructor) a ser usada e um, ou mais, métodos que **chamam essa estratégia** (***ContextInterface()***).

A **estratégia** será uma **classe abstracta** com um método que permite **invocar a estratégia** (***AlgorithmInterface()***) e **várias subclasses concretas**.



Em termos de **variações**, a mais importante prende-se com o facto de podermos ter uma **estratégia por omissão** (por exemplo quando é usado o construtor vazio).