0.1 Factory Method

Define uma interface para criar um objeto, mas deixa as subclasses decidir qual classe será instanciada.

Usa-se quando:

- uma classe não consegue antecipar a classe dos objetos a serem criados.
- uma classe pretende que as subclasses especifiquem os objetos a criar.

Elimina a necessidade de ligar classes específicas ao código.

0.2 Composite

Compôr objetos em estruturas de árvores para representar hierarquias parte-todo. Permite que os clientes tratem objetos e composições uniformemente.

Usa-se quando:

- se pretende representar hierarquias parte-todo de objetos.
- se pretende que os clientes ignorem a diferença entre composições e objetos individuais.

Tipos primitivos podem ser compostos em objetos complexos.

0.3 Command

Encapsular um pedido como um objeto permitindo parametrizar clientes com diferentes pedidos, filas ou registo de pedidos e suportar operações reversíveis.

Usa-se quando:

- se especifica, enfileira e executa pedidos em momentos diferentes.
- se suporta operações de undo/redo.
- se estrutura um sistema envolvido em operações de alto nível baseadas em operações primitivas.

Comandos podem ser estendidos e manipulados como qualquer outro objeto.
É fácil adicionar comandos.
Tipos primitivos podem ser compostos em objetos complexos.

0.4 Observer

Define uma dependência um-para-muitos entre objetos para que quando um objeto muda o seu estado todos os dependentes são notificados e atualizados.

Usa-se quando:

- uma abstração tem dois aspetos, um dependente do outro.
- a mudança de um objeto implica a mudança de outro.

- um objeto deve notificar outros sem assumir quem são.

Emparelhamento entre sujeito e observador.
Suporte para comunicação.
Atualizações inesperadas.

0.5 Strategy

Define-se uma família de algoritmos, encapsula-se cada um e fazem-se permutáveis. Permite que o algoritmo varie dependendo do cliente que o usa.

Usa-se quando:

- muitas classes relacionadas só diferem no seu comportamento.
- se precisa de variantes diferentes de um algoritmo.
- um algoritmo usa dados que um cliente não deve conhecer.
- uma classe define muitos comportamentos que aparecem em múltiplas declarações condicionais.

Alternativa a ter subclasses.
Elimina condições e providencia implementações diferentes.

0.6 State

Permite que um objeto mude o seu comportamento quando os seus estados internos mudam. O objeto aparecerá para mudar a sua classe.

Usa-se quando:

- o comportamento de um objeto depende do seu estado e precisa de o mudar em run-time.
- as operações têm grandes condições que dependem de uma ou mais enumerações.

Faz com que as transições entre estados sejam explícitas.

0.7 Adapter

Converte a interface de uma classe noutra interface esperada pelo cliente. Permite que classes trabalhem juntas e não o poderiam fazer noutro caso devido à incompatibilidade de interfaces.

Usa-se quando:

- se quer utilizar uma classe existente e a sua interface não combina com a que se precisa.
- se quer criar uma classe reutilizável que trabalhe com classes imprevistas.

0.8 Decorator

Anexar responsabilidades adicionais a um objeto dinamicamente. Providencia uma alternativa flexível a criar subclasses por extensão.

Usa-se quando:

- se quer adicionar responsabilidades a objetos individuais dinamicamente.
- se quer retirar responsabilidades.
- estender por criação de subclasses é impraticável.

Mais flexível que herança estática. Evita classes com muitas funções/responsabilidades.

0.9 Singleton

Assegurar que uma classe só tem uma instância e providenciar uma forma global para a aceder.

Usa-se quando:

- tem que haver uma e uma só instância da classe.
- a única instância tem que ser estensível para subclasse.

É um considerado um **anti-pattern**:

- difícil de testar.
- difícil de implementar com multi-threading.

O que fazer em vez de usar o padrão? Criar uma instância da classe e propagá-la para o sítio onde será utilizada.

0.10 Abstract-Factory

Providenciar uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar a classe concreta.

Usa-se quando:

- um sistema deve ser independente de como os seus produtos são criados, compostos e representados.
- um sistema deve ser configurável com mais do que uma família de produtos.

Isola classes concretas.
Promove a consistência entre os produtos.
É difícil suportar novos tipos de produtos.
subsectionMode-View-Controller (MVC)
Divide o sistema em 3 partes:

- o modela representa os dados.
- a vista mostra o modelo e envia ações do utilizador ao controlador.
- o controlador providencia o modelo à vista e interpreta as ações do utilizador.