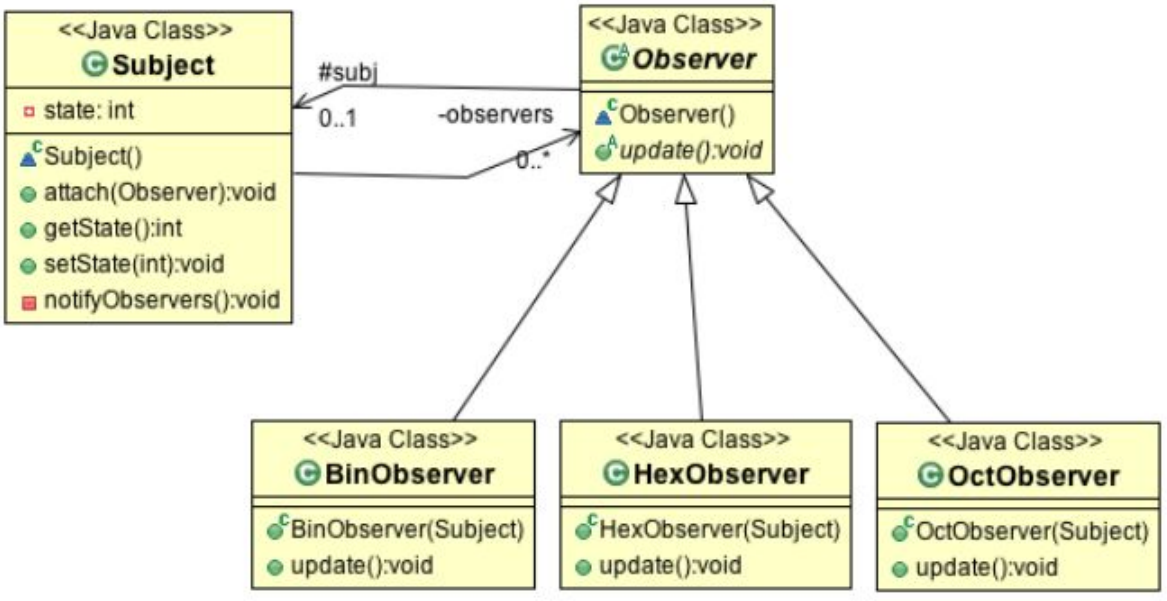
ATENÇÃO QUE FALTA O VISITOR

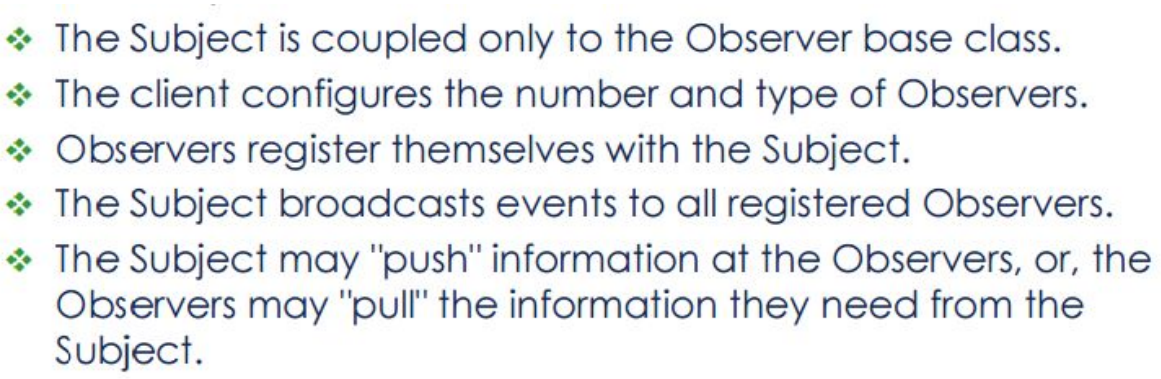
Observer

Objectivo

- O Observer é um padrão de projeto de software que define uma dependência um-para-muitos entre objetos de modo que quando um objeto muda o estado, todos seus dependentes são notificados e atualizados automaticamente. Permite que objetos interessados sejam avisados da mudança de estado ou outros eventos ocorrendo num outro objeto.

- Os observadores devem conhecer o objecto de interesse
- O objecto de interesse (subject) deve notificar os observadores quando for atualizado





State

Objectivo

- State é um padrão de projeto de software usado quando o comportamento de um objeto muda, dependendo do seu estado.

O padrão State é motivado por aqueles objetos que, em seu estado atual, varia o seu comportamento devido as diferentes mensagens que possa receber. Como exemplo, tomamos uma classe Livro, um objeto desta classe terá respostas diferentes, dependendo do seu estado(Disponível ou Emprestado). Por exemplo invocando o método reservar de um objeto da classe Livro seu comportamento será diferente, se o Livro está no estado Disponível ou no estado Emprestado.

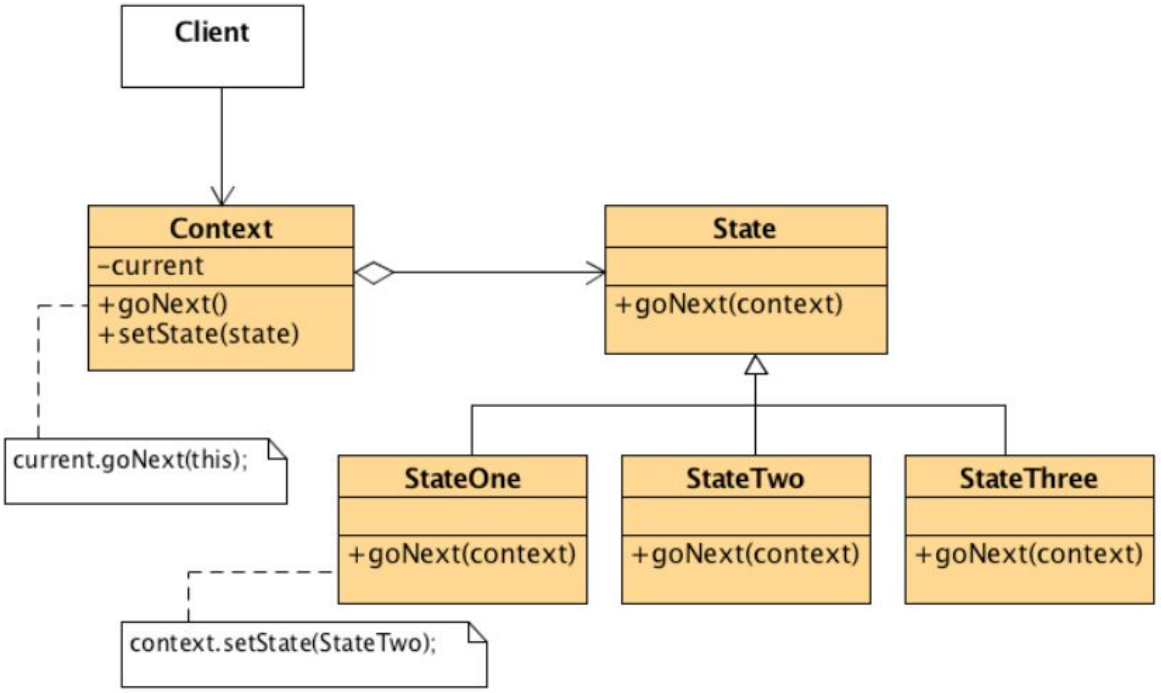
Problema

Há uma extrema complexidade no código quando tentamos gerenciar comportamentos diferentes, dependendo de um número de estados diferentes. Também manter o código torna-se difícil, e mesmo em alguns casos, podem apontar a uma inconsistência de estados atuais na forma de implementação dos diferentes estados no código (por exemplo, com variáveis para cada estado

Solução

Se implementa uma classe para cada estado diferente do objeto e o desenvolvimento de cada método para cada estado em particular. O objeto da classe a que pertencem esses estados resolvem os diferentes comportamentos, dependendo de sua condição, com as

instâncias das classes de estado. Assim, sempre está presente em um objeto o seu estado atual e se comunica com ele a resolvendo suas responsabilidades.



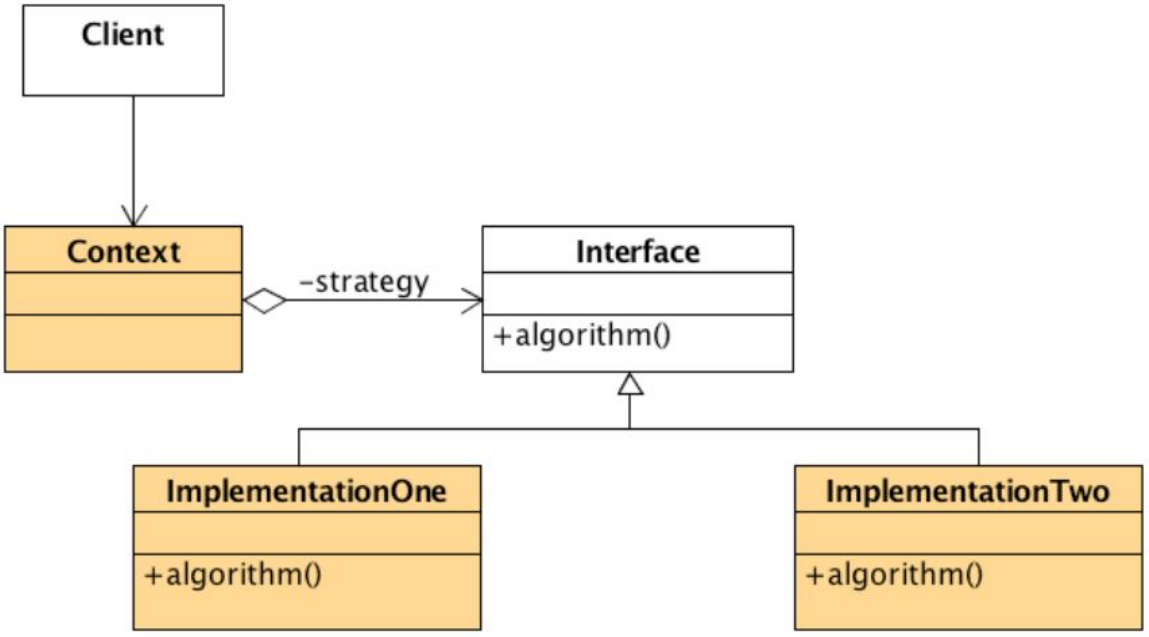
Strategy

Objectivo

Definir uma família de algoritmos, encapsular cada algoritmo e torná-los intercambiáveis. O padrão strategy permite o algoritmo varie independentemente dos clientes que o usam.

Capturar a abstração numa interface e fazer a implementação numa classe derivada.

O padrão strategy é semelhante ao padrão state, excepto no seu objectivo. Strategy permite mudar o interior de um objecto enquanto um decorador muda a “pele”.

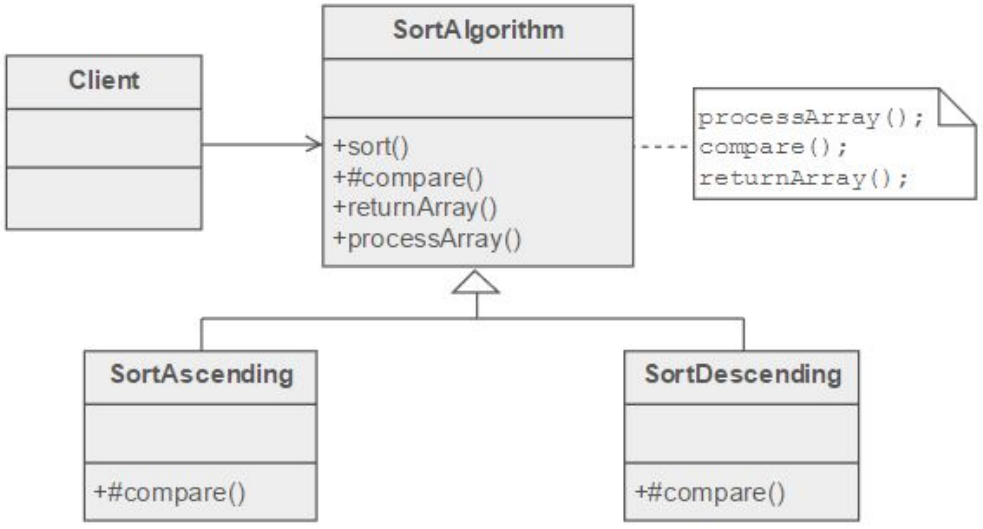


Template

Define o esqueleto de um algoritmo numa operação, deixando que subclasses completem algumas das etapas. O padrão Template Method permite que subclasses redefinam determinadas etapas de um algoritmo sem alterar a estrutura do algoritmo

Quando usar?

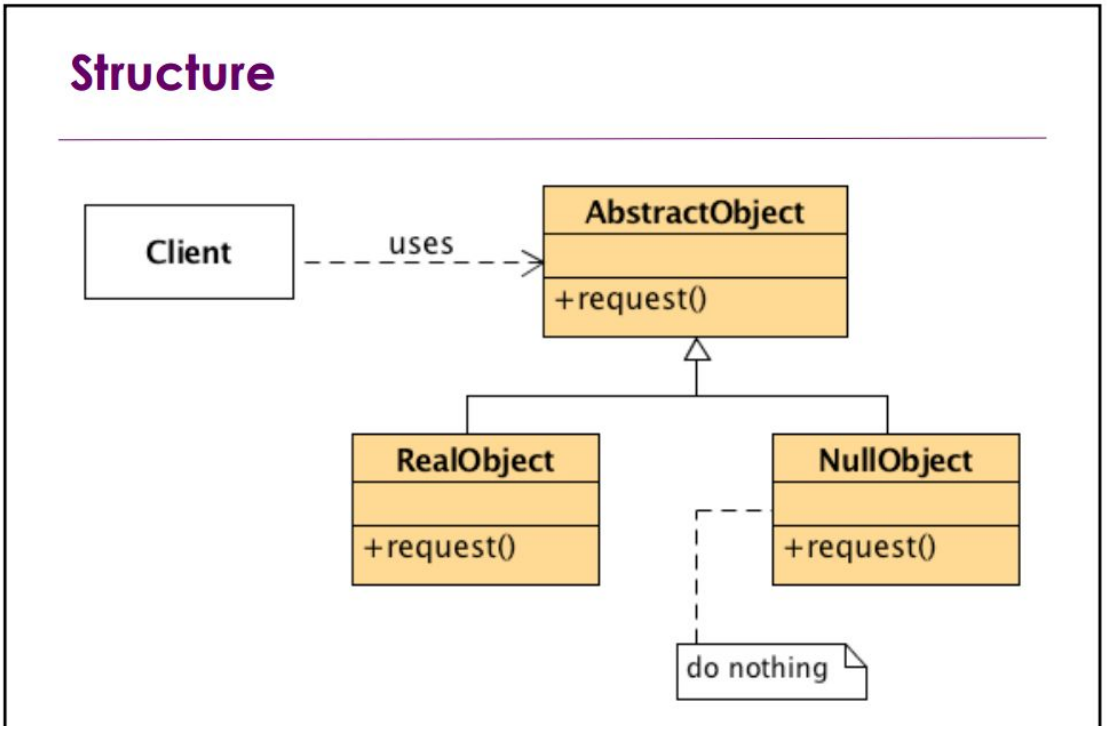
- Para implementar partes invariantes de um algoritmo uma única vez e deixar subclasses implementarem o comportamento variável



Null Object

É uma objeto que é criado para simular outro objeto com os mesmos contratos, mas sem funcionalidade alguma. Sem estado e sem comportamento real. Os dois tipos são compatíveis entre si já que atendem o mesmo contrato, implementam as mesmas interfaces e possivelmente estendem o mesmo tipo.

É uma forma de evitar o uso do estado nulo de objetos que a maioria das linguagens usam. Quando realmente existe uma necessidade semântica, uma situação que um objeto de um tipo seja nulo, usa-se este objeto que atende os mesmos contratos mas não faz nada.



Padrões de Arquitetura de Software

Arquitetura multicamada (Layered Architecture)

- A maioria das arquiteturas multicamadas consistem em 4 camadas principais:
    - Apresnetação (Presentation)
    - Negócio (Business)
    - Persistência (Persistence)
    - Banco de Dados (Database)
- Cada camada tem uma funcionalidade e responsabilidade específica dentro da aplicação
- Separa os conceitos
- Componentes de uma camada em específico, lidam somente com a lógica relacionada com essa camada

Layers fechadas (Closed Layers)

- Uma cada so consegue falar directamente com a camada seguinte. Não pode aceder a outras camadas que não essa.
- Promove a independência entre camadas, o isolamento e reduz dependências.
- Mudanças feitas em uma camada, geralmente não afetam os componentes das outras camadas.

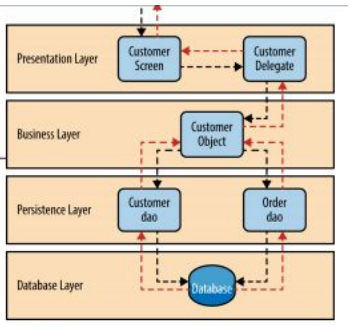
Layers abertas (Open Layers)

- As camadas conseguem aceder a camadas que não a directamente abaixo.

Considerações

- Bom ponto de partida para a maior parte das aplicações
- É necessário ter cuidado para que as requests não passem por várias camadas da arquitetura sem ter lógica associada em cada camada. Não ser só um ponto de passagem.

Pattern Example 1



- The **customer screen** is responsible for accepting the request and displaying information.
  - does not know where the data is, how it is retrieved, which database tables must be queried
  - it forwards the request onto the **customer delegate** module.
  - This module is responsible for knowing which modules in the business layer can process that request
- The **customer object** is responsible for aggregating all of the information needed by the business request.
  - This module calls out to the **customer dao** (data access object) module in the persistence layer to get customer data, and also the **order dao** module to get order information.
- These modules in turn execute SQL statements to retrieve the corresponding data and pass it back up to the customer object in the business layer.
  - Once the customer object receives the data, it aggregates the data and passes that information back up to the customer delegate, which then passes that data to the customer screen to be presented to the user.

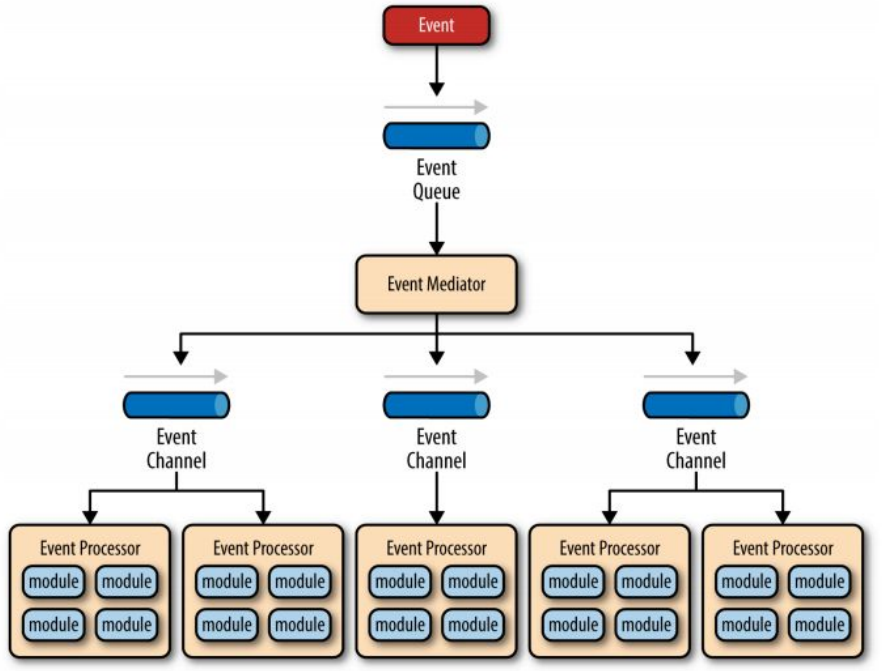
User interface layer WPF / ASP.NET / Console App / WinRT / …
Service layer WCF / ASMX / …
Business logic layer
Data access layer EF / ADO.NET / …
Database SQL Server / Oracle / MySql / …

Event-Driven Architecture

- Um padrão de arquitetura distribuído assíncrono usado frequentemente para produzir aplicações altamente escaláveis. É muito adaptável de modo que pode ser usado tanto para aplicações pequenas como para aplicações complexas.
- É composto por componentes, com com grande grau de desacoplamento, que recebem e processam eventos assíncronos.
- Consistem em duas topologias principais: mediator e broker.

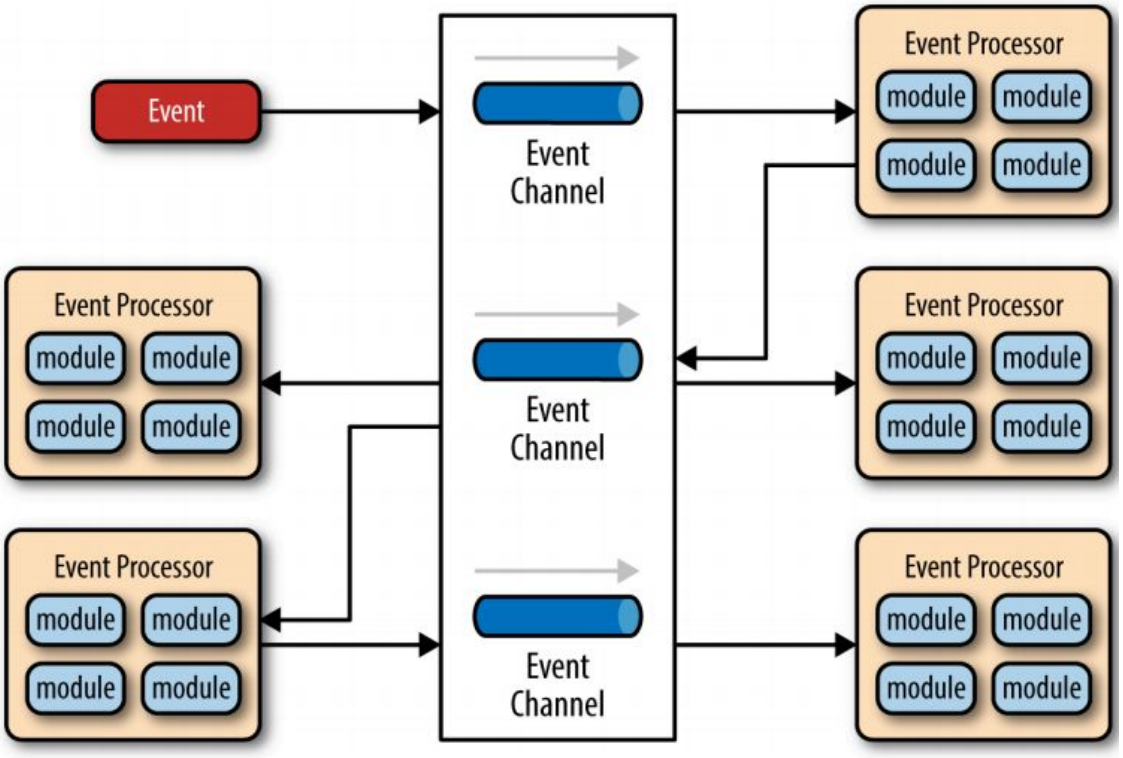
Topologia de mediador

- É útil para eventos que têm vários passos e necessitam de algum nível de orquestração para processar o evento.
    - Exemplo: um evento para comprar/vender uma ação da banca, precisa que primeiro a troca seja validada, verificar se respeita as várias regras, atribuir a troca a um broker, calcular comissão e finalmente colocar a transação com o broker.
- Existem 4 componentes principais na topologia de mediador:
    - Event queues
    - Event mediator
    - event channels
    - even processors



Topologia de broker

- Não existe um mediador de eventos central
    - As mensagens são distribuídas pelos componentes que processam eventos em modo de cadeia através de um “lightweight message broker”
- É uma topologia útil quando temos fluxo simples de processamento de eventos.
- Os dois componentes principais são:
    - Broker
    - Componente processador de eventos

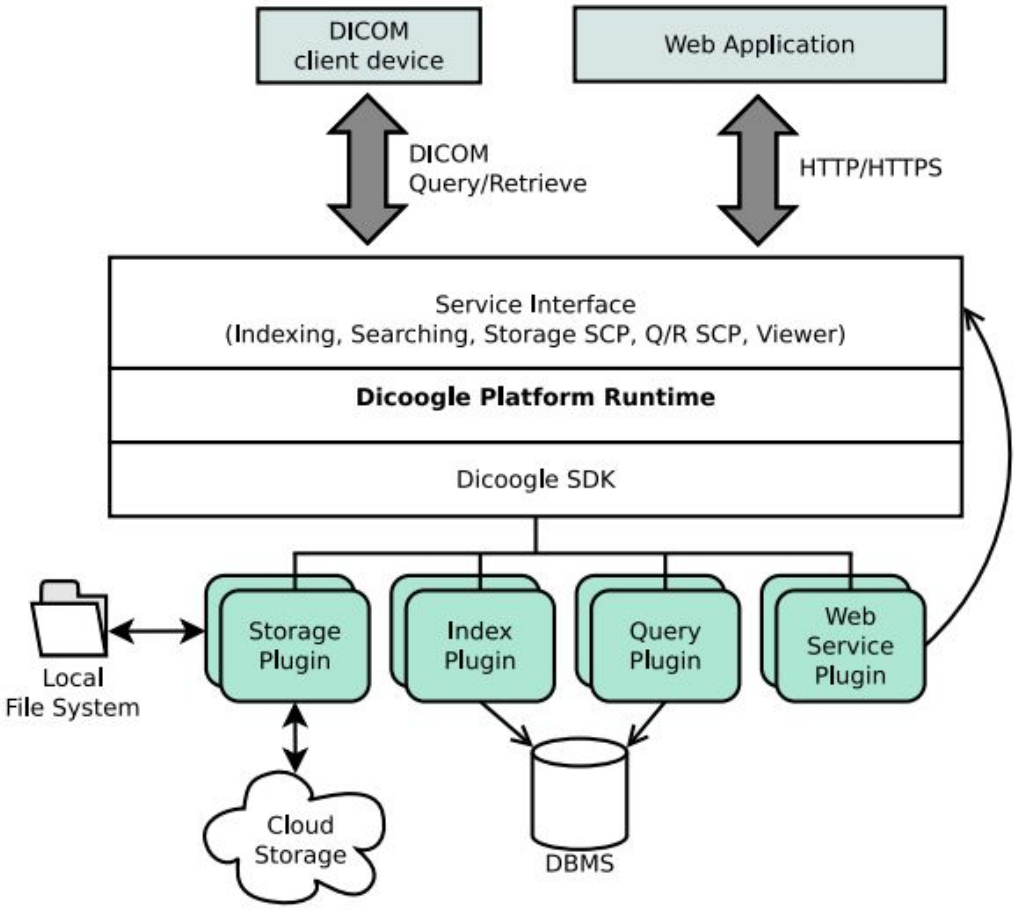


Considerações

- É uma arquitetura complexa de implementar principalmente devido à sua natureza assíncrona.

Microkernel Architecture

- É um padrão que permite adicionar funcionalidades à aplicação como plugins à base da aplicação, dando assim extensibilidade á aplicação, separação de funcionalidades e isolamento.
- É também referenciada como plug-in architecture pattern e é o padrão natural para implementar aplicações baseadas em produtos.
- Muitos sistemas operativos implementam a arquitetura microkernel, daí a origem do nome.
- Dois componentes principais
    - O sistema base (core)
        - Normalmente contém as funcionalidades mínimas que permitem o sistema funcionar
    - Módulos (plug-in)
        - Podem ser ligados de várias maneiras ao sistema base
            - OSGi (open service gateway initiative), messaging, web-services, etc...



Considerações

- Um dos maiores benefícios deste padrão é que pode ser usada em conjunto ou como parte de outro padrão de arquitetura
- Alto suporte para um design evolutivo e desenvolvimento incremental
- Ideal para aplicações “product-based”
    - Particularmente produtos que vão lançar novas funcionalidades ao longo do tempo e querem controlar que utilizadores recebem essas funcionalidades
    - É sempre possível fazer refactor à aplicação para outro padrão mais adequado aos requisitos.

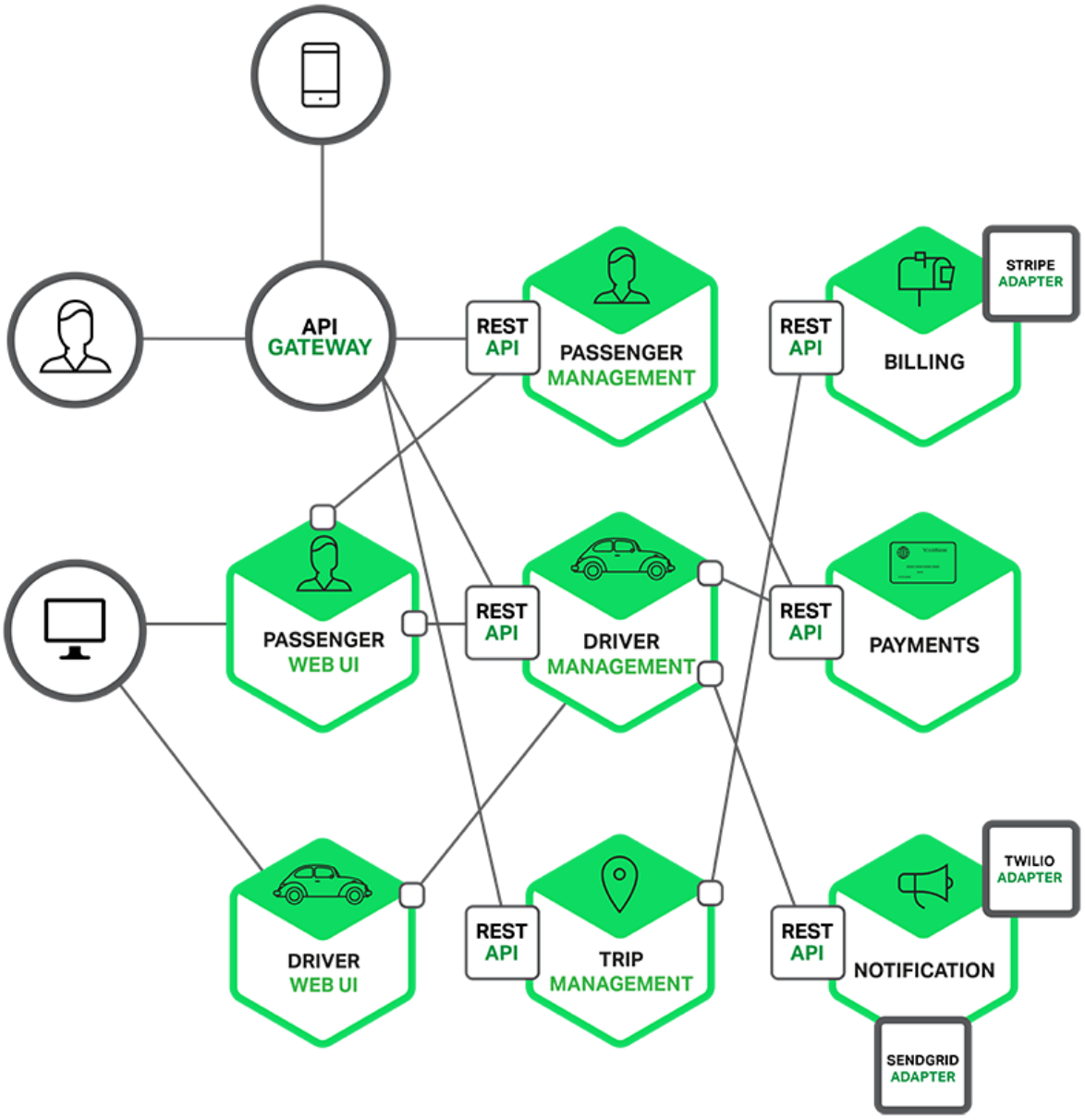
Microservices Architecture Pattern

- Cada componente é implantado como uma unidade separada, permitindo uma implantação mais simples e sem acoplamento.
- Arquitetura distribuída
    - Os componentes estão completamente desacoplados
    - Comunicação é feita através de REST, SOAP, RMI, etc…
- Evolui naturalmente a partir de duas fontes
    - Aplicações monolíticas desenvolvidas usando o padrão de arquitetura por camadas
    - Aplicações distribuídas desenvolvidas através do padrão de arquitetura orientado a serviços

Os micro serviços são de baixo acoplamento, pequenos e focados numa responsabilidade, “language neutral e bounded context”.

Existem várias maneiras de implementar este padrão, mas destacam-se três como as mais populares:

- API Rest-based
- application REST-based
- centralized messaging topology



Considerações

- Aplicações são geralmente mais robustas, mais escaláveis e mais fáceis de suportar.
- Só os componentes que são modificados necessitam de ser implantados.

“

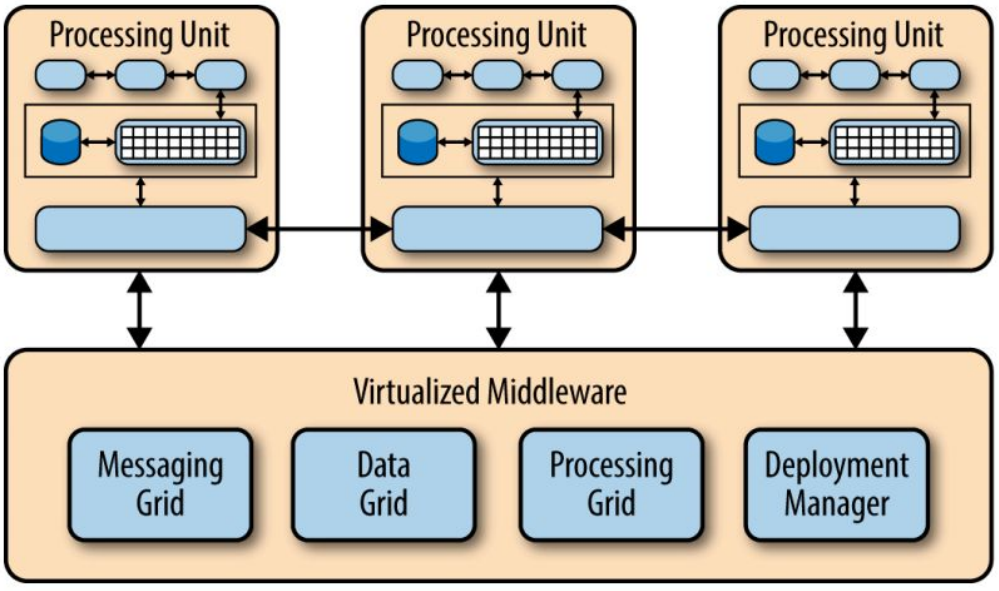
- But .. distributed architecture – it shares some of the same complex issues found in the event-driven architecture pattern, including contract creation, maintenance, and government, remote system availability, and remote access authentication and authorization. “

Space-Based Architecture

- A maioria das aplicações web seguem o mesmo fluxo de pedidos:
    - Um pedido do browser chega ao web-server, posteriormente ao servidor da aplicação e finalmente ao servidor do banco de dados
- Começam a aparecer bottlenecks à medida que o user load aumenta.
    - Numa primeira instância na camada do servidor-web, depois na camada do servidor da aplicação e finalmente no servidor de banco de dados
- O padrão de arquitetura “space-based” é desenhado especialmente para resolver problemas de escalabilidade e concorrência
- É também referido como padrão de arquitetura cloud
- A escalabilidade é atingida através da substituição da base de dados central com data grids replicadas.
- Os dados da aplicação são mantidos em memória e replicados em todas as unidades de processamento activas.
- As unidades de processamento podem ser inicializadas e paradas dinamicamente à medida que o user load aumenta e diminui.

Existem dois componentes principais neste padrão

- Componente da unidade de processamento que contém os componentes da aplicação
    - Inclui componentes web-based assim como a lógica de negócios backend.
- Componentes virtualizados middleware que tratam das comunicações e “housekeeping”



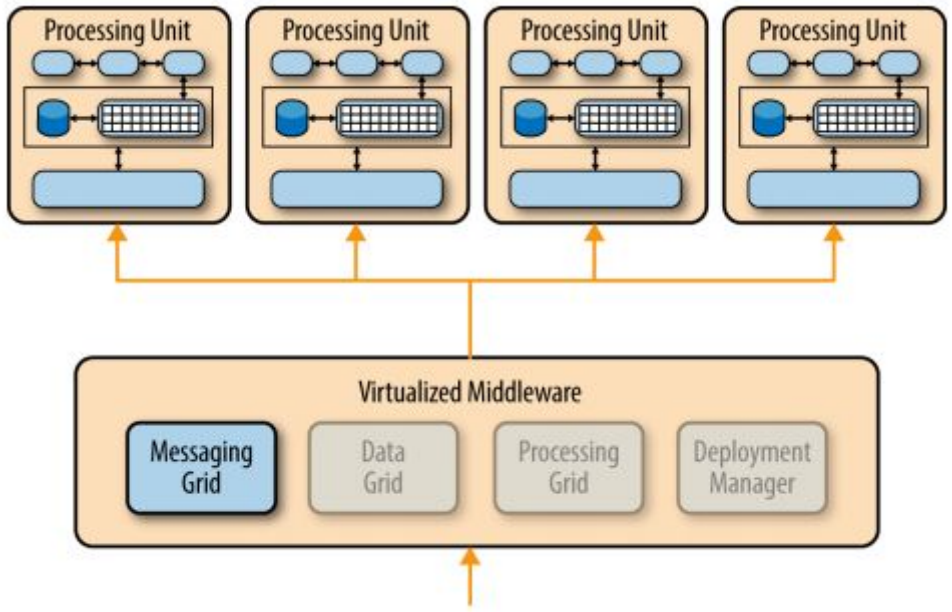
Middleware virtualizado

- Controla os pedidos, sessões, replicação de dados, processamento distribuído de pedidos e o deployment de unidades de processamento
- Quatro componentes principais
    - Messaging grid
    - Data grid
    - processing grid
    - deployment manager

Messaging grid

Controla os pedidos e informação das sessões.

Para cada pedido, o messaging grid determina os componentes activos de processamento capazes de tratar do pedido. Pode utilizar um algoritmo round robin, ou outro algoritmo mais complexo.



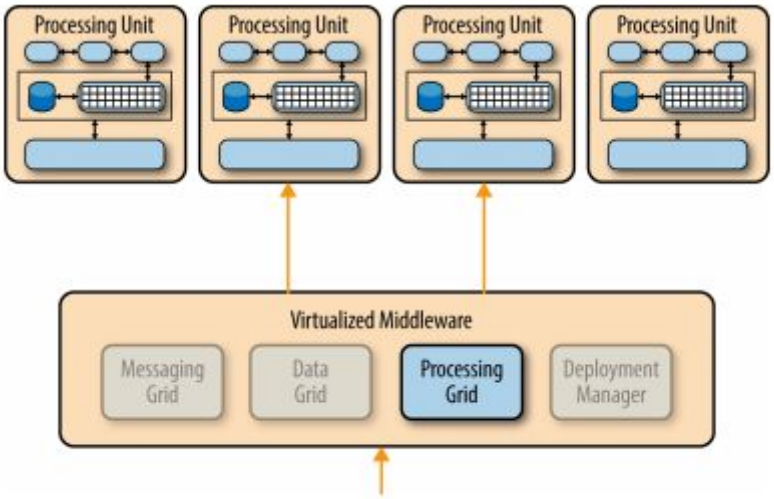
Data grid

- Interage com o motor de replicação de dados de cada unidade de processamento.
- Tem como objectivo controlar a replicação de dados entre unidades de processamento quando existe uma atualização destes.

- Cada unidade de processamento tem que conter exatamente os mesmos dados na “in-memory datagrid”
- Isto tem que ser atingido de modo paralelo, assíncrono e rápido, muita vezes em microssegundos.

Processing grid

- É um componente opcional na virtualização middleware.
    - É responsável por controlar o processamento dos pedidos distribuídos quando existem várias unidades de processamento.
    - É responsável pela coordenação entre as várias unidades de processamento.



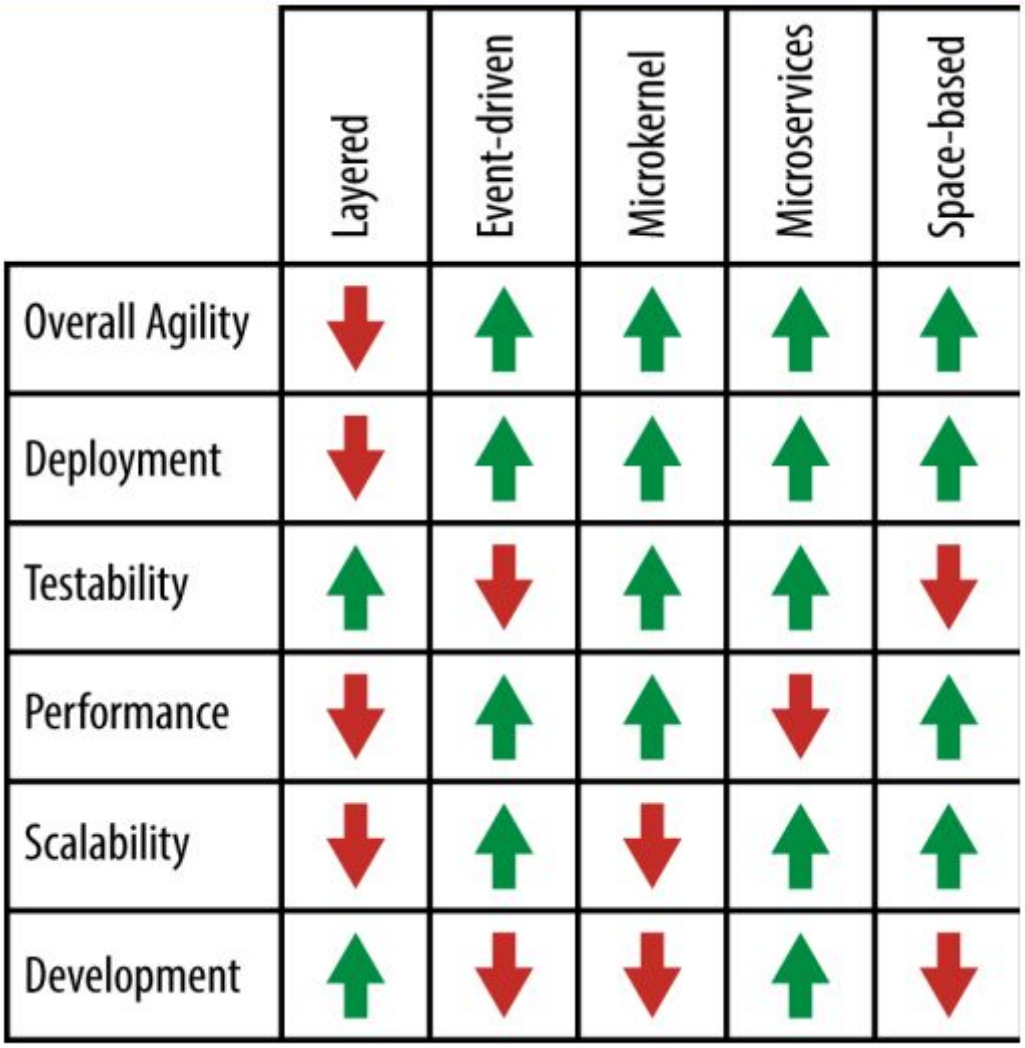
Deployment manager

- Controla dinamicamente a inicialização e a terminação das unidades de processamento, baseado nas condições de carga (load).
- Monitoriza constantemente os tempos de resposta e o user-load.
    - Inicializa novas unidades de processamento quando o load aumenta
    - Termina unidades de processamento quando o load diminui.
- É um componente crítico para as necessidades de escalabilidade variáveis de uma aplicação

Considerações

- É um padrão complexo e caro (expensive) de implementar
- É uma boa escolha para aplicações web pequenas com load variável (social media websites, leilões online, etc…)

- Não é adequado para aplicações com bases de dados relacionais de larga escala com quantidades grandes de dados operacionais.



Java reflection

Reflection é um termo usado para indicar a capacidade de obter metadados sobre o próprio programa compilado.

Permite criar chamadas em tempo de execução, sem precisar conhecer as classes e objetos envolvidos quando escrevemos nosso código (tempo de compilação). Esse dinamismo é necessário para resolvermos determinadas tarefas que nosso programa só descobre serem necessárias ao receber dados, em tempo de execução.

Classe Class

Para cada objecto carregado pela JVM, existe um objecto do tipo Class associado. Os tipos primitivos também são representados por objetos Class.

As instâncias do tipo class armazenam informações sobre a classe:

- Nome da classe
- Herança
- Interfaces implementadas
- Métodos
- Atributos

Comandos importantes (?)

Class.forName(“nomeDaClasse”).newInstance()

- Devolve um objecto conforme o nome pedido e inicializa-o.

String nome = “nomeDaClasse”;
nome.getClass();
nome.getConstructors():

- Devolve os construtores da classe pedida

nome.GetFields()

- Devolve os atributos da classe pedida

ATENÇÃO QUE FALTA O VISITOR