Dublês de Testes

Os dublês de testes segundo o Livro Gerard Mezanos, existem em cinco variações e podemos agrupa-los em dois como mostrado abaixo.

Duble de Testes	Agrupamento do duble	Descrição do duble de testes
Mock	Mocks	Mocks são dublês de testes criados apartir de uma ferramenta para simular o comportamento de uma dependência. ele tem um comportamento mais complexo, para simular a dependência, ele tem um conjunto e valores esperados e resposta
Spy	Mocks	É igual ao mock, entretanto são escritos manualmente, e tem como objetivo armazenar informações das sua chamadas, ou seja ele vai armazenar os parâmetros passados para spy, a quantidade de vezes que ele e chamado.
dummies	Stubs	É um valor ficciono ou inventado, como valores nulos ou strings quais quer para resolver chamada de um método. São valores que unicamente para preencher um espaço, eventualmente eu não vou usar eles.
fake	Stubs	É similar o dummy, entretanto tem um caráter de substituir uma dependência que ainda não existe, então eles tem alguma implementação, para se passar por uma dependência, mas ele corta vários caminhos, ele e muito mais simples que a implementação real
stubs	Stubs	Ele server muito para consulta, ele se parece com o mock, entretanto ele não tem a inteligência do mock. O stub é o cara que eu vou utilizar para mandar uma requisição e ele vai me retornar o mesmo valor.

Comportamento Observável e Detalhes implementação

Um Conceitos que vai nos ajudar a entender alguns dos próximos conceitos é o que se trata de entender quais partes do código representam Detalhes de Implementação e quais são Comportável Observável, e necessário entender que funcionalidade ou são do comportamento observável ou de detalhes de implementação.

▼ Comportamento Observável

São partes que expõem métodos para que os clientes consigam atingir os seus objetivos. Sendo a exposição de uma operação (calculo ou decorrente de evento colateral, ou ambos) ou a condição do sistema.

▼ Detalhes de Implementação

Tudo que não tiver haver com os conceitos apresentados, pelo comportamento observável, se caracteriza como detalhe de implementação.

Vale trazer algumas ressalvas que podem confundir a muitos, quando a palavra cliente é abordada, dependendo do código, ou parte ao qual estamos nos debruçando podem ter comportamentos diferentes. Muitas vezes podemos retratar o cliente como um serviço que consome do sistema, ou até mesma a interface com o usuário.

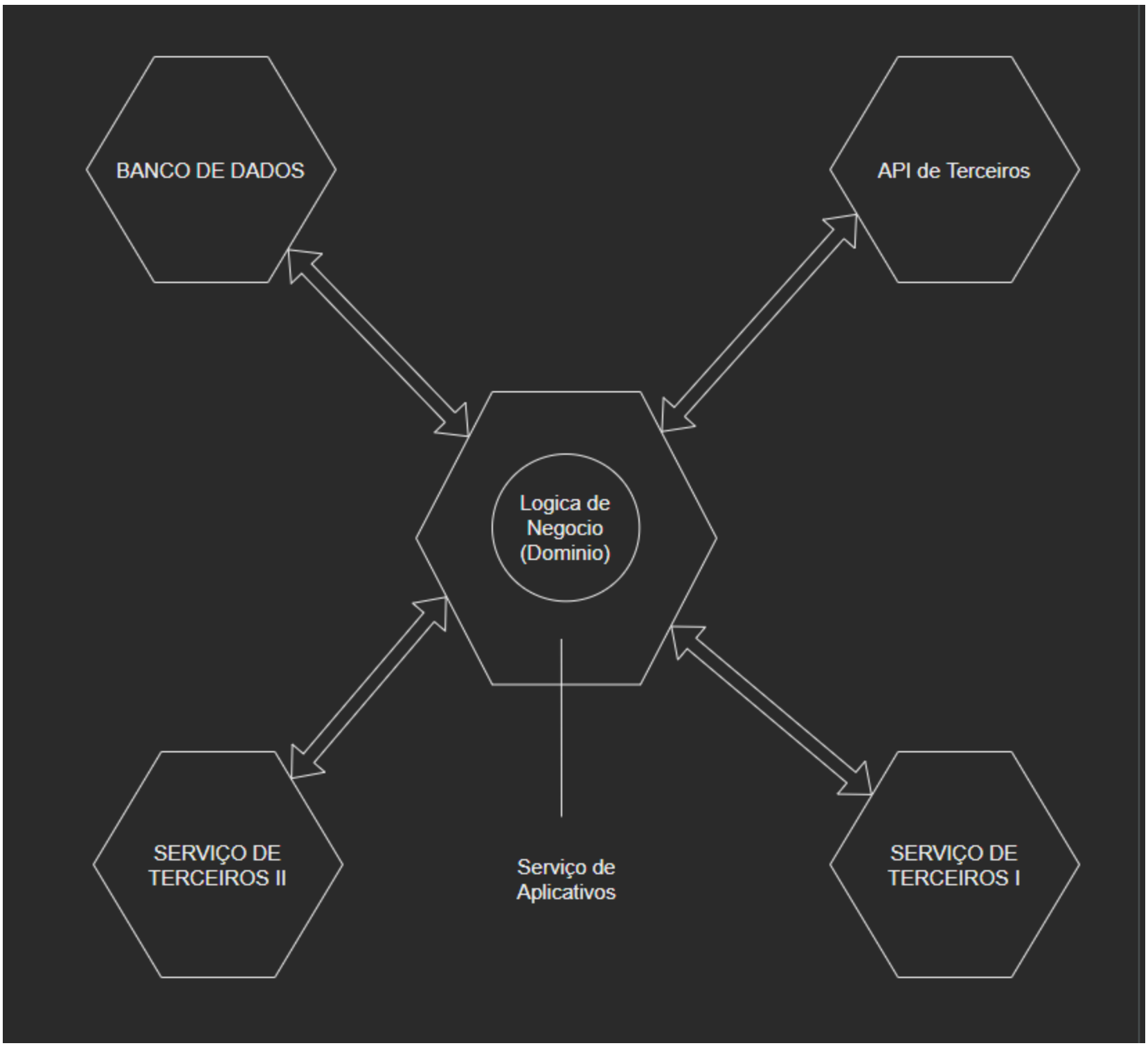
Dividindo entre APIs Publicas e Privadas

Quando temos um sistema, e estamos falando de comportamento observável e detalhes de implementação, temos que relevar outros dois conceitos para garantir que o nosso sistema e bem projetado, são APIs publicas e privadas. Mas o que define uma API publica ou privada. e especificamente o acesso as operações e métodos expostos ai cliente em um sistema bem projetado e esperado que a API publica é o comportamento observável, representem a mesma coisa, assim como APIs privadas e os detalhes de implementação. Enquanto um software mal projetado espontem detalhes de implementação em sua API publica.

Esse pratica de expor os detalhes de implementação, apresenta um problema no encapsulamento das funções que não deveriam ser visíveis e manipuladas pelo cliente, pois também apresentam regras de negocio, e expõem um vulnerabilidade, que nos conhecemos como violação invariante.

Arquitetura Hexagonal

A Principal vantagem da arquitetura Hexagonal, esta na separação por camada onde são atribuídas duas camadas, uma que implementa o logica de negocio, denominada Domínio e a que implementa a parte de comunicação tanto com o cliente como com outros serviços. Na diagramação da arquitetura hexagonal podemos ver que o domínio se apresenta dentro do centro do hexágono e tudo fora do domínio como serviços de aplicativo. Na arquitetura hexagonal um aplicação pode se comunicar com outros serviços e comum acreditar que pelo ideia de ser um hexágono, 6 lados cada lado condiz com a possibilidade de uma conexão com um outros serviço assim limitando a quantidade de conexões. porem realmente não existe essa limitação e uma aplicação pode se comunicar com quantos serviços houver necessidade.



Agora veremos um exemplo da aplicação da Arquitetura Hexagonal:

Vamos pensar no cenário, onde eu tenho um formulário qualquer, ai a pessoa coloca o nome e o e-mail, apartir dessas informações eu crio, um cadastro para a pessoas dentro do meu banco de dados

```
import re

class Form:

  def __init__(self, title, description = None):
    self._title = title
    self._description = description

  def fill_form(self, name, email):
    self._name = name
    self._email = email

  def generate_nickname(self):
    self._nickname = self.name[:6] + self;name[:-4]

  def get_information_user(self):
    return {
      "nickname": self._nickname
      "name": self._name
      "email": self._email
      }


class controllerForms:

  def save_information_userDB(self, title, name, email, descrition = None):

    newform = Form(title, description)
```

```
    newform.fill_form(name, email)
    newform.generate_nickname()

    # classe que permite conexão com o baco de dados mas
  # so realiza operações na tabela de usuarios
    db = UserDB()
    db.create_user(newform.get_information_user())
```

Observer que no ***controllerForms***, apartir do momento que eu instancio o meu objeto ***form***, eu chamo duas funções, a primeira ***fill_form*** para o cliente colocar as informações, e a segunda para gerar um *nickname*.

Concordem comigo essa função de gerar o *nickname*, apartir de uma regra de negocio, aplicada em cima do nome, é um detalhe de implementação, e isso não deveria estar visível para o meu cliente. E quem é o meu cliente aqui o ***ControllerForms***, isso é na verdade uma violação invariante, que eu tenho no meu código.

Okay, E como eu corrijo isto, simples encapsulo a metodo, e chamo ela dentro da função do escopo observável ***fill_form***, você passa a responsabilidade da execução desse detalhe de implementação para o escopo observável:

```
class Form:

  def __init__(self, title, description = None):
    self._title = title
    self._description = description

  def fill_form(self, name, email):
    self._name = name
    self._email = email
    self._nickname = self._generate_nickname(self._name)

  def _generate_nickname(self, name):
    return name[:6] + name[:-4]

  def get_information_user(self):
    return {
      "nickname": self._nickname
      "name": self._name
      "email": self._email
      }


class controllerForms:

  def save_information_userDB(self, title, name, email, descrition = None):

    newform = Form(title, description)
    newform.fill_form(name, email)

    # classe que permite conexão com o baco de dados mas
  # so realiza operações na tabela de usuarios
    db = UserDB()
    db.create_user(newform.get_information_user())
```

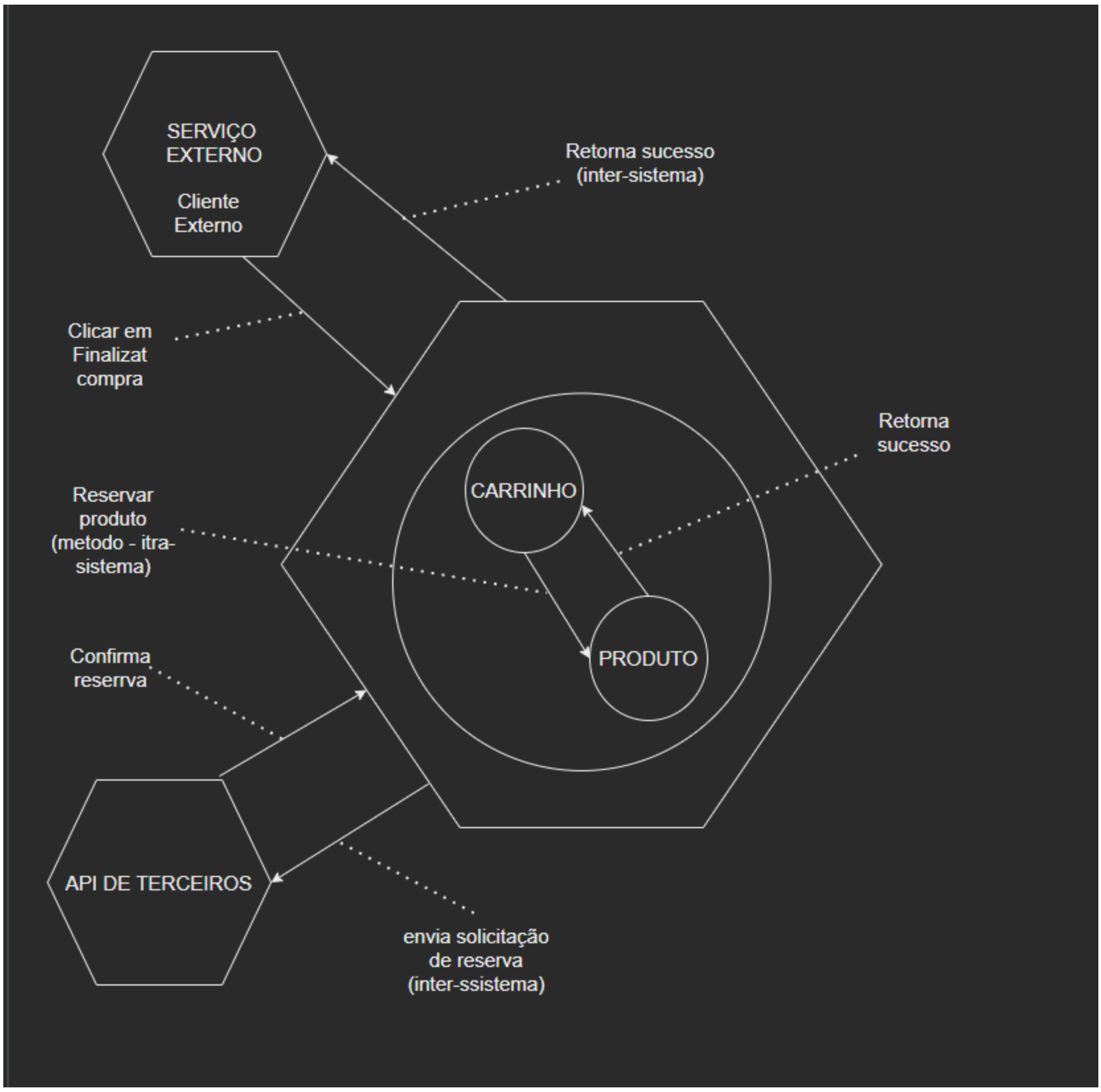
Observer que as principais mudanças desse em relação ao outro código e agora no só chamado um linha para criar todas as informações do usuário. E agora o método ***_generate_nickname*** é privado, o cliente não tem mais acesso, e você consegui separa o domínio do comportamento observável.

Comunicações intra-sistema e inter-sistema

AS conexões intra-sistema, são conexões entre classes internas do próprio sistema, e são parte dos detalhes de implementação, isso se ta ao fato de que as conexões entre as classes internas, não fazem parte do comportamento observável e nem acessível para o cliente. Enquanto comunicações inter-sistema, são comunicações com outros serviços ou aplicativos, a são partes do comportamento observável, uma das características que nos ajuda a observar

isso é é que conforme esses serviços vão evoluindo, as comunicações também tem que evoluir, a maneira como eles conversam devem ser modificadas.

Para exemplificar vamos pensar em um carinho de compra, nos temos uma serviço de externo que é responsável pela comunicação do cliente com a nossa aplicação, nos recebemos um evento de compra onde estamos realizando a finalização da compra, então nos temos uma classe carinho, ela chama uma classe que é o produto, tentando reservar o produto, o produto por sua vez passa a responsabilidade para a serviço de aplicação que se comunica com uma API de terceira para reservar, essa e uma comunicação inter-sistema, de terceiro confirma a reserva, passar para produto, produto responde para carinho que passar a resposta para o serviço de aplicação que por sua vez via comunicação inter-sistema, responde ao cliente externo.



Veja que somente as comunicações entre o Carinho e Produto eram comunicações intra-sistemas, enquanto as outras eram comunicações inter-sistema