

## **ARQUITETURA DE REFERÊNCIA - GUIA GERAL**

**Versão: 1.2**

Data	Versão	Descrição	Responsável
13/03/2019	1.0	Criação do Guia Geral da Arquitetura de Referência – Alinhada a MDS ágil	Carlos Alberto Rodrigues de Oliveira Santana Francisco Ernesto Diaz Teixeira Neto José Augusto de Jesus
19/03/2019	1.1	Homologação do Guia Geral	Carlos Alberto Rodrigues Santana José Arthur Souza de Macedo José Augusto de Jesus Leonardo Moraes Borges Théo Alves Monteiro Valdecy Lourenço de Araújo Júnior
13/05/2019	1.2	Alteração do item 5 - Anexo A para referenciar o Guia de Frontend de forma geral.	Nirian Martins Silveira dos Santos

## Sumário

<b>1</b>	<b>INTRODUÇÃO.....</b>	<b>5</b>
<b>2</b>	<b>PROPÓSITO.....</b>	<b>5</b>
<b>3</b>	<b>REPRESENTAÇÃO ARQUITETURAL .....</b>	<b>5</b>
3.1	METAS E RESTRIÇÕES ARQUITETURAIS.....	6
3.1.1	<i>Camada .....</i>	<i>9</i>
3.1.2	<i>Frontend.....</i>	<i>9</i>
3.1.3	<i>Backend.....</i>	<i>9</i>
3.1.4	<i>Namespace.....</i>	<i>9</i>
3.1.5	<i>Padrões .....</i>	<i>10</i>
3.1.5.1	Tecnologias.....	10
3.1.5.2	Componentes.....	10
3.1.5.3	Nomenclatura .....	10
3.1.6	<i>Padrões para construção de estória de usuários.....</i>	<i>11</i>
3.2	VISÕES.....	12
3.2.1	<i>Visão lógica (ou de projeto) .....</i>	<i>12</i>
3.2.1.1	Pacotes .....	13
3.2.1.1.1	Pacote extensões.....	13
3.2.1.1.2	Pacote utilitários.....	13
3.2.1.1.3	Pacote modelos.....	13
3.2.1.1.4	Pacote persistência .....	14
3.2.1.1.5	Pacote regras .....	14
3.2.1.1.6	Pacote serviços - camada apresentação.....	14
3.2.1.1.7	Pacote serviços - camada de agregação.....	14
3.2.1.1.8	Pacote diretivas.....	15
3.2.1.1.9	Pacote controladores.....	15
3.2.1.1.10	Pacote configurações.....	15
3.2.1.2	Camadas .....	15
3.2.1.2.1	Camada de acesso de dados.....	17
3.2.1.2.2	Camada de apresentação .....	18
3.2.1.2.3	Camada de serviços.....	19
3.2.2	<i>Visão de implementação (ou de componente) .....</i>	<i>21</i>
3.2.3	<i>Visão de processos (ou concorrência).....</i>	<i>22</i>
3.2.4	<i>Visão de dados .....</i>	<i>22</i>
3.2.5	<i>Visão física (ou implantação) .....</i>	<i>22</i>
3.3	DEVOPS.....	23
3.3.1	<i>Comunicação .....</i>	<i>23</i>
3.3.2	<i>Colaboração .....</i>	<i>23</i>

<b>3.3.3</b>	<b>Automação</b>	23
<b>3.3.4</b>	<b>Monitoramento</b>	23
<b>3.3.5</b>	<b>Padrão para execução da automação</b>	23
3.3.5.1	Kubernetes	23
3.3.5.2	GitOps	24
3.3.5.3	Ambientes de execução	24
3.3.5.4	Pipeline	24
3.3.5.4.1	Feature branch	25
3.3.5.4.2	hotfix branch, release branch	25
3.3.5.4.3	Tags	26
3.3.5.5	Scripts	27
3.3.5.5.1	Scripts de banco	27
3.3.5.6	Operators	27
3.4	CONTROLE DE VERSÃO	28
3.4.1	Fluxo de trabalho	29
3.5	REQUISITOS DE QA	31
3.5.1	Pré-requisitos de qualidade	31
3.5.2	Automação	32
3.5.3	Cobertura de código por teste	32
4	REFERÊNCIAS	33
5	O ANEXO A – GUIAS DE ARQUITETURAS	34

## 1 Introdução

Este documento descreve uma arquitetura padrão para a construção de aplicativos orientados a recursos REST e interação rica com o usuário. Essa arquitetura será aplicada as seguintes plataformas: Java, PHP, Python e Javascript. Por tratar-se de uma discussão com o enfoque no *design* da aplicação, foram consideradas as diferenças entre as plataformas a serem tratadas em documentos específicos sobre a arquitetura de cada plataforma e o seu respectivo guia operacional para a construção do código.

Aqui, são apresentadas as motivações e diretrizes que orientam o *design* da arquitetura, orientações de como o desenvolvedor pode empregar diretrizes para resolver problemas de sua aplicação, e, por fim, a descrição dos elementos que compõem a arquitetura, tanto na sua estrutura de pacotes quanto na divisão de camadas.

## 2 Propósito

O objetivo principal deste documento é demonstrar as características que orientam a arquitetura, em especial, orientar o desenvolvedor no que concerne na divisão da aplicação em camadas, em pacotes e em estereótipos de classe. Assim, não tem a intenção de ser um documento definitivo de arquitetura para cada sistema individual. Ao invés disso, tenta traçar os objetivos de alto nível que levem a uma boa arquitetura de software. Cada sistema em particular deve ser analisado junto com a equipe de desenvolvimento e um dos arquitetos de sistemas da CAPES.

## 3 Representação Arquitetural

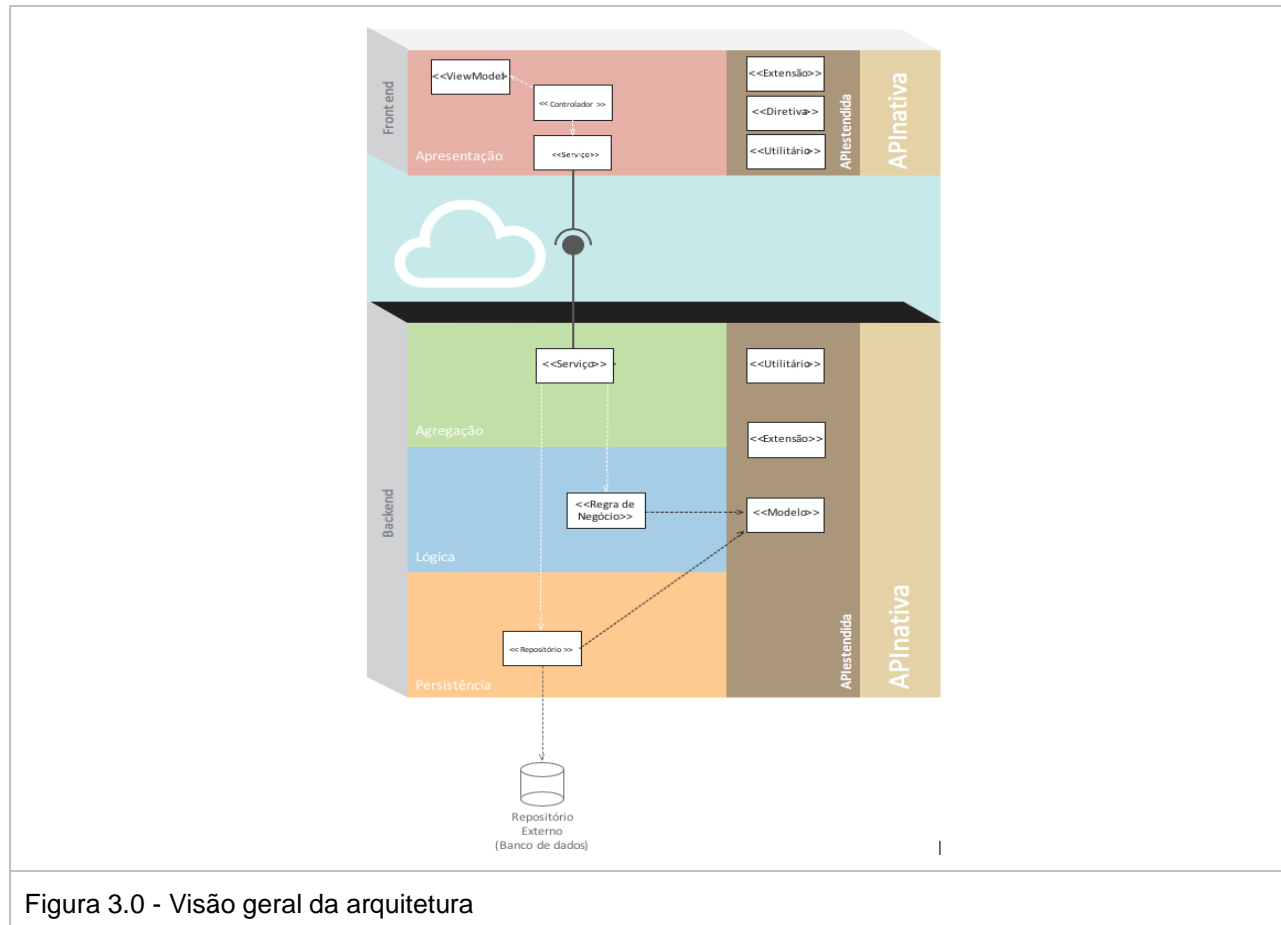


Figura 3.0 - Visão geral da arquitetura

Agrupamento lógico de fragmentos de códigos que implementa um componente, funcionalidade ou aplicação. Conforme figura 3.0, na vertical encontram-se dispostas quatro camadas lógicas: camada de apresentação (vermelho), de agregação (verde), de lógica (azul) de persistência (amarelo). Cada camada está disposta em um ambiente de execução: *Frontend* e *Backend*. Esses ambientes de execução são interligados por serviços.

Alguns elementos, como os que pertencem aos pacotes extensões e utilitários, não pertencem a nenhuma camada específica, por terem a finalidade de estender as capacidades das plataformas que hospedam as aplicações. Nesses casos, para fins da arquitetura, esses elementos serão considerados parte da API básica de cada plataforma.

Na horizontal, são representadas as camadas físicas de cada ambiente. As seções mais à direita (nos tons de marrom e dourado) representam dois níveis de API das plataformas hospedeiras (nativa e estendida), que reúnem os elementos que não fazem parte da aplicação em si, ou seja, linguagem de programação e componentes de terceiros utilizados na codificação.

As caixas brancas representam os estereótipos das classes que serão construídas para a aplicação. As linhas pontilhadas, o relacionamento de dependência entre as classes na sua implementação: *viewmodels*, controladores, serviços, regras de negócio, repositórios, utilitários, extensões, diretivas e modelos.

O espaço azul entre os dois ambientes de execução representa o meio utilizado para o tráfego de informações.

A comunicação entre os elementos da arquitetura hospedadas nos diferentes ambientes dar-se-á em único ponto representada pela conexão entre os estereótipos Serviço da camada de apresentação e Serviço de agregação por meio do protocolo aberto de rede HTTP e a notação JSON.

### 3.1 Metas e Restrições arquiteturais

A arquitetura de software apresentada nesse documento objetiva descrever aplicações modularizadas hospedadas em plataformas web, orientadas a serviços REST, com forte distinção entre seus elementos operacionais. Dentre suas características principais, podemos citar:

- *Alta coesão*: caracteriza-se pela composição de elementos de baixa complexidade e com funções bem definidas, permitindo alto grau de reuso e baixo esforço de manutenção;
- *Baixo acoplamento*: o código produzido será composto por elementos independentes, passíveis de substituição a baixo custo a qualquer momento do ciclo de vida da aplicação. Permite, por exemplo, a troca do gerenciador de banco de dados por um competidor com ajustes pontuais em um único componente da aplicação (o repositório), sem a necessidade de modificação dos demais;
- *Interoperabilidade*: baseada em serviços que funcionam sobre padrões abertos, esse modelo de aplicação permite, ao longo do tempo, o reaproveitamento de funcionalidades completas entre sistemas executados em plataformas distintas. Por exemplo: uma funcionalidade implementada a partir de um serviço na plataforma Java é passível de utilização imediata em um sistema da plataforma PHP ou Python;

- *Interface intercambiável*: a interação com o usuário é executada em plataforma independente dos demais elementos da aplicação. Essa característica é evidente ao observar-se no diagrama de representação arquitetural, que a camada de apresentação tem um ambiente de aplicação (frontend) para si, isolada das demais camadas. Dessa forma, quando conveniente, é possível acrescentar-se uma nova interface com o usuário a um custo mínimo.
- *Design responsivo*: permite a disponibilização de aplicações construídas seguindo as orientações dessa arquitetura a partir de dispositivos com fatores de interface distintas. Isso significa que, sem esforço adicional, essas aplicações são imediatamente utilizáveis a partir de smartphones e tablets;

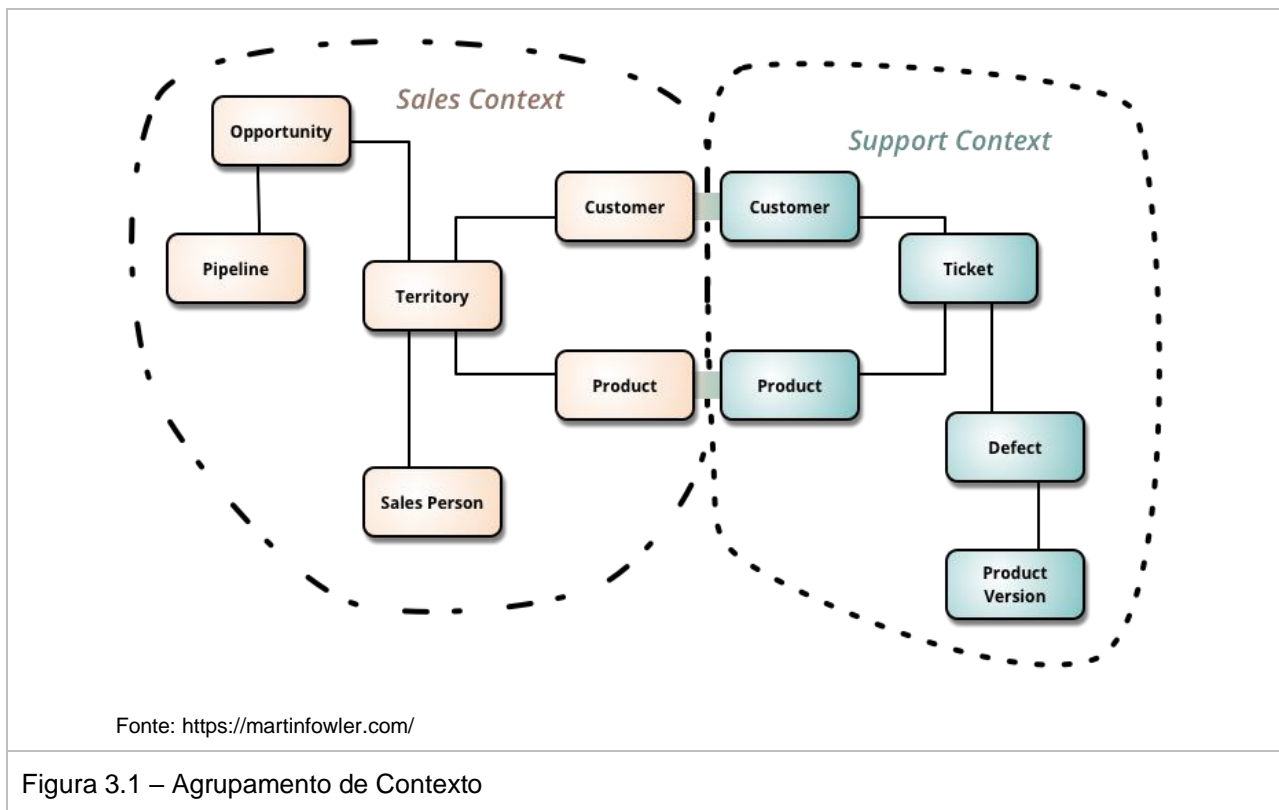
*Testabilidade*: fronteiras claras e distintas entre os componentes da aplicação lhe concede a capacidade inerente de testar esses elementos de forma automática e independente (testes unitários) ou em conjunto (teste de integração e teste funcional).

Além dos princípios acima, a arquitetura das aplicações preza:

- *Alinhamento do código com o negócio*: o contato dos desenvolvedores com especialistas do domínio.
- *Favorecer a reutilização*: códigos que possibilitem o real reaproveitamento.
- *Independência da tecnologia*: soluções que não foquem em tecnologia e sim, no atendimento do negócio.

Dado a separação de responsabilidades apresentada pela *figura 02*, técnicas e padrões do DDD (*Domain-Driven Design*) serão adotadas para o agrupamento de contexto de pacotes, que, segundo Martin Fowler, é o padrão central do DDD.

Um *Agrupamento de Contexto* permite criar seções com responsabilidades específicas dentro de um sistema de contexto mais amplo. Agrupamentos de Contexto se traduz em pacotes da aplicação com responsabilidades e fronteiras bem definidas.



Um *grupamento de Contexto* encapsula todas as funcionalidades necessárias ao atendimento de uma funcionalidade.

A exemplo, um módulo *Foo* terá seu namespace próprio e todas as suas camadas agrupadas sob este. Exemplo de agrupamento de camadas em um pacote:

```
br.gov.capes.<NomeSistema>.foo.camadaApresentação
br.gov.capes.<NomeSistema>.foo.camadaServiço
br.gov.capes.<NomeSistema>.foo.camadaAcessoADados
```

Tabela 3.0 – Tabela de agrupamento de camadas em pacotes

O módulo **Foo** conterá todos os recursos para atender ao que se propõe e deverá ser possível seu reuso por outros módulos, o que se traduz em um isolamento de seu modelo de negócio.



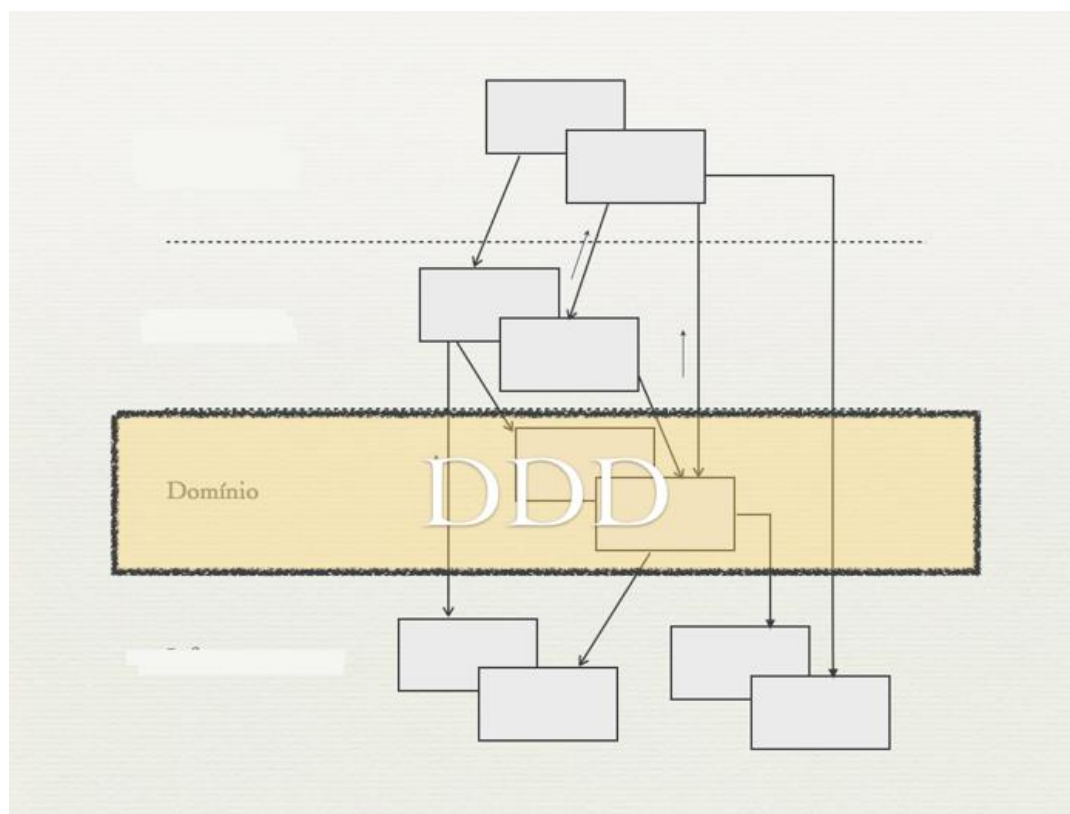


Figura 3.2 - Arquitetura em camadas, utilizada para separar o domínio do resto da aplicação.

### 3.1.1 Camada

Agrupamento lógico de códigos que visa a identificar inequivocamente determinada implementação do sistema. Seu agrupamento consistirá em domínios de negócio ou pacotes.

### 3.1.2 Frontend

O *Frontend view* consiste da camada de apresentação da aplicação ao usuário final. Em se tratando de uma aplicação com interação com usuário via tela gráfica, essa camada será o agrupamento de todos os componentes gráficos necessários para interação entre o usuário e a aplicação. Em sendo uma aplicação que demande interação em modo linha de comando, essa camada assume o papel de detalhar o processo de integração entre o que está sendo configurado e aplicação.

### 3.1.3 Backend

O Backend consiste no agrupamento de todas as tecnologias, separadas por camadas, necessárias ao funcionamento da aplicação e que consome o *Frontend view* como entrada ou saída dos dados processados.

### 3.1.4 Namespace

Por sua vez, camadas são logicamente agrupadas em pacotes designados por namespaces. A Capes adotará o namespace padrão:

br/gov/capes/<nomeSistema>

Tabela 3.1 – Namespace padrão Capes

O namespace se reflete na estrutura de diretórios da aplicação, que será iniciada por:

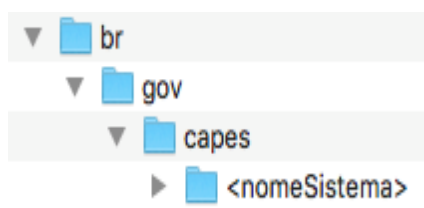


Tabela 3.2 – Estrutura de diretórios base de aplicações Capes

### 3.1.5 Padrões

#### 3.1.5.1 Tecnologias

As tecnologias utilizadas no processo de desenvolvimento de aplicações Capes serão definidas em documentos inerentes a cada pilha de desenvolvimento (Java, PHP, Python).

#### 3.1.5.2 Componentes

Haverá um repositório de componentes homologado pela Capes e referenciado nos respectivos guias de arquitetura. O *Anexo A – Guias de arquiteturas*, deste documento, elenca os Guias de arquitetura e a que tecnologia se destina.

#### 3.1.5.3 Nomenclatura

Para os projetos das plataformas PHP e Python, quando não houver regras que associem os nomes de arquivos e diretórios aos elementos implementados ali, a estrutura de pastas apresentada a seguir será aplicada. Para os projetos da plataforma Java, será necessário seguir a estrutura inerente, em que os diretórios e nomes de arquivos correspondem, respectivamente, à estrutura de pacotes da aplicação e ao nome de cada classe. Os elementos da camada de apresentação, em todos os casos, residem sob o diretório correspondente ao conteúdo estático, conforme citado adiante.

/projeto		Diretório raiz do projeto.
	/api	Contem o código necessário à disponibilização da api REST da aplicação.
	/utils	Contém os componentes da API estendida da aplicação, tais quais interfaces com componentes de logging, geração de documentos PDF ou acessos a sistemas externos.
	/modelos	Contém os modelos da aplicação, tanto os utilizados para persistência quanto aqueles necessários para implementar as APIs estendidas.

	/repositorios	Reúne as classes necessárias para as funcionalidades de persistência, em especial os repositórios e as anotações de mapeamento objeto-relacional.
	/servicos	Conjunto de classes de manipulação pura das informações da aplicação, como a geração de dados agregados ou transformação de dados para a utilização nas demais camadas (e.g. conversão de unidade ou agregação de dados para geração de relatórios ou visualizações alternativas).

Tabela 3.3 – Estrutura de pastas de um projeto CAPES

### 3.1.6 Padrões para construção de estória de usuários

Existem três perguntas primordiais na construção da estória do usuário:

- Quem? - indica o autor da ação ou para quem ela é direcionada ou vai se beneficiar da ação;
- O quê? - indica o que se deseja realizar com a funcionalidade;
- Por quê? - indica o valor e propósito, a justificativa, da funcionalidade.

Um exemplo de como escrever a estória do usuário seria: "Como [Quem?], quero [O quê?] para [Por quê?]."

Também é importante, na estória escrita, definir seus critérios de aceite. Os quais irão validar e orientar o resultado almejado na implementação da funcionalidade entregue.

Em um exemplo mais completo:

"Como ADMINISTRADOR, preciso de acesso a um relatório de vendas para saber quanto recebi em determinado período."

Critérios de aceite:

- Vendedores não poderão ter acesso ao relatório;
- Os relatórios devem ter a opção de imprimir;
- Os relatórios devem ter a opção de exportar para planilha eletrônica;
- Deverá ter filtro por data, para que o administrador possa definir o período desejado.

Pode-se também apontar nos critérios de aceite o detalhamento das informações que desejam ser vistas, ordenações, etc.

### 3.2 Visões

Existem várias formas de se observar um sistema em construção, cada pessoa envolvida resalta propriedades que lhe interessa e omitem as não relevantes. A exemplo:

- O modo como as pessoas que desempenham papéis diferentes dentro do processo de desenvolvimento de software ou solução veem o problema;
- O modo como cada entidade (componente) da arquitetura de software pode ser observado (perspectivas diferentes).

Assim, conseguimos delimitar suas abstrações nas seguintes visões:

- Visão lógica (ou de projeto);
- Visão de implementação (ou de componente);
- Visão de processos (ou concorrência);
- Visão de dados;
- Visão física (ou implantação).

#### 3.2.1 Visão lógica (ou de projeto)

A visão lógica é a área que detém as funcionalidades do sistema ou solução a ser elaborada.

Essa visão contempla os diagramas de caso de uso e classes (UML).

Existe a possibilidade de haver uma visão em modelo de processos (BPMN) para melhor entendimento e complementação do mesmo.

Deve-se levar em consideração que em um modelo ágil tais documentos e diagramas não são obrigatórios em sua totalidade. Podendo haver a eleição de quais poderão ser entregues mediante acordo e nível de complexidade da operação.

A partir do momento em que esta documentação faça parte de um projeto, é imprescindível seu alinhamento com a visão de implementação e processos para um correto acompanhamento e histórico.

Em resumo:

- Os atores dessa visão são os clientes do produto, os analistas e os desenvolvedores;
- Existe uma forte ligação ao problema do negócio;
- Independe de decisões de projeto;
- Descreve e especifica a estrutura estática do sistema e as colaborações dinâmicas entre objetos via mensagens para realizarem as funções do sistema;
- Contém a coleção de pacotes, classes e relacionamentos.

### 3.2.1.1 Pacotes

Os elementos da aplicação hospedadas em *servidor de aplicação* (i.e. *camadas de persistência, lógica e agregação*), serão distribuídos em pacotes principais, tais como: *extensões, utilitários, modelos, persistência, regras, serviços*.

A camada de apresentação tem uma divisão de pacotes particulares, a saber: *extensões, utilitários, diretivas, serviços, controladores e configurações*. No âmbito dessa camada, pacotes são denominados *módulos*. A distribuição e denominação distintas se dá a fim de adequar os elementos dessa camada aos mecanismos inerentes da plataforma que será utilizada na sua implementação (i.e. *Java*).

Ambas as lista de pacotes não são extensivas e, quando conveniente, é prerrogativa do desenvolvedor subdividir os pacotes sugeridos ou acrescentar novos, desde que a nova organização não viole a que se encontra descrita neste documento e se mostre benéfica à compreensão dos mecanismos implementados para aquela aplicação específica.

#### 3.2.1.1.1 Pacote extensões

Em todas as plataformas escolhidas, é-se possível estender as classes das APIS nativas, acrescentando métodos personalizados nas classes nativas (Texto, Vetores, entre outros). O material desenvolvido deverá ser reunido neste pacote, o qual, quase sempre, se resume a código.

Métodos de extensão têm, por natureza, alto potencial de reaproveitamento em qualquer aplicação da mesma plataforma, na forma de *bibliotecas de extensão*.

As classes desse pacote podem ser referenciadas pelo código de qualquer pacote da aplicação.

#### 3.2.1.1.2 Pacote utilitários

Reúne as classes com funções das mais diversas. Escritas para realizar tarefas comuns referentes à aplicação que não se categorizam como regras de negócio, eliminando assim a redundância de código.

Conversão de tipos e formatação de dados específicas de uma aplicação são exemplos de funções comumente realizadas em classes de pacote *utilitários*.

As classes desse pacote podem ser referenciadas pelo código de qualquer pacote da aplicação.

#### 3.2.1.1.3 Pacote modelos

As classes de estereótipo *Model* (conforme descrito mais adiante) e classes utilizadas para o transporte de informações na forma de parâmetros ou resultados de operações são reunidas neste pacote.

Possuem, em essência, apenas atributos e sua lógica interna e simplificada. Na maior parte do tempo, contém apenas código com o objetivo de facilitar a conversão ou agregar os atributos ali armazenados como, por exemplo, propriedades calculáveis a partir das demais (exemplo: o valor agregado de um produto: preço unitário versus quantidade).

As classes desse pacote podem ser referenciadas pelo código de qualquer pacote da aplicação. Não é permitido que faça referência a classes pertencentes aos pacotes *regras, persistência* ou *controladores*.

#### 3.2.1.1.4 Pacote persistência

As classes do pacote *persistência* realizam as funções de atualização e recuperação de informações a partir de repositórios destinados ao seu armazenamento.

No caso mais comum esses repositórios correspondem a bancos de dados relacionais, mas é possível a utilização de qualquer dispositivo que permita as operações necessárias para esse fim. Alguns exemplos são arquivos de texto, arquivos binários ou bancos de dados não estruturados (NoSQL, sistema de arquivos, serviços externos, entre outros).

A função primordial das classes desse pacote é tornar os mecanismos de persistência transparentes da perspectiva das demais classes da aplicação, concentrando todos os detalhes necessários para essas operações.

As classes pertencentes a esse pacote são do estereótipo repositório e devem ser referenciados somente a partir de classes pertencentes ao pacote *serviços*.

#### 3.2.1.1.5 Pacote regras

Concentra as classes que implementam as regras de negócio de uma aplicação. São classes de um único estereótipo (*regra de negócio*), que realizam seus objetivos por meio da alteração do estado interno de classes individuais ou conjuntos de classes da aplicação. As funcionalidades necessárias para a efetivação dessas regras (exemplo; recuperação e persistência das informações) não fazem parte das implementadas pelas classes desse pacote. Desse modo, é comum a ocorrência de código composto de classes desse pacote e do pacote *persistência* lado a lado.

As classes pertencentes a esse pacote serão referenciadas somente a partir de classes pertencentes ao *pacote serviços*.

#### 3.2.1.1.6 Pacote serviços - camada apresentação

Contém classes análogas às aquelas encontradas no pacote *serviços*, agora na função de cliente. Em essência, fazem a comunicação (via rede) com os serviços hospedados no *backend* (da camada de agregação) e conversões de dados necessárias para realizar os *ViewModels* (explicados mais adiante).

O objetivo principal dessas classes é realizar o isolamento da comunicação com o *backend* das classes dos demais pacotes da camada de apresentação.

#### 3.2.1.1.7 Pacote serviços - camada de agregação

Concentra as classes as quais se encontram as implementações da fronteira de serviços em uma aplicação. Nessas classes, encontra-se o código necessário para coordenar as funcionalidades encontradas nas demais. Rotinas que recuperam modelos a partir de repositórios é comum em uma classe de serviço. Processar informações utilizando *regras de negócio*, validar

parâmetros recebidos na sua chamada e atualizar informações da aplicação são tarefas comuns deste pacote.

Deste pacote, permiti-se fazer referência a classes de todos os demais pacotes. Mas o contrário não é permitido.

#### **3.2.1.1.8 Pacote diretivas**

Esse módulo pertence ao código que implementa a interação com o usuário, e contém as construções necessárias para a implementação de diretivas de comportamento. Em geral, nele, encontram-se os componentes (visuais ou não) necessários para a interação os quais não estão disponíveis naturalmente na plataforma ou composições de componentes que evitam redundância de código ao longo de toda aplicação.

Por exemplo, um componente visual para o preenchimento e validação de números de CEP pertenceria a essa categoria.

Classes do módulo diretivas são acessadas apenas pela infraestrutura da interface ou camada de apresentação.

#### **3.2.1.1.9 Pacote controladores**

As classes desse módulo realizam as funções de orquestração dos componentes necessários para implementar a interação com o usuário. Em essência, ela utiliza as classes do módulo de serviços, implementam os comportamentos esperados dos componentes visuais (exemplo: respostas a eventos capturados junto ao usuário) e ajustes dos elementos gráficos necessários.

Uma classe típica desse módulo é aquela que implementa o comportamento de formulários: ela inicializa componentes visuais, recupera informações utilizando classes do módulo *serviços*, implementa o comportamento da interação com o usuário (e.g. validação de informações entradas pelo usuário no lado cliente) e utiliza novamente classes do módulo *serviços* para atualizar informações ao final da operação.

#### **3.2.1.1.10 Pacote configurações**

Em alguns casos, em especial na implementação de comportamentos não nativo da interface do usuário, a execução de código específico para efetivar as configurações necessárias ao ambiente cliente é requerida.

Para manter o código organizado, as configurações necessárias para a interação com o usuário serão centralizadas no módulo configurações.

#### **3.2.1.2 Camadas**



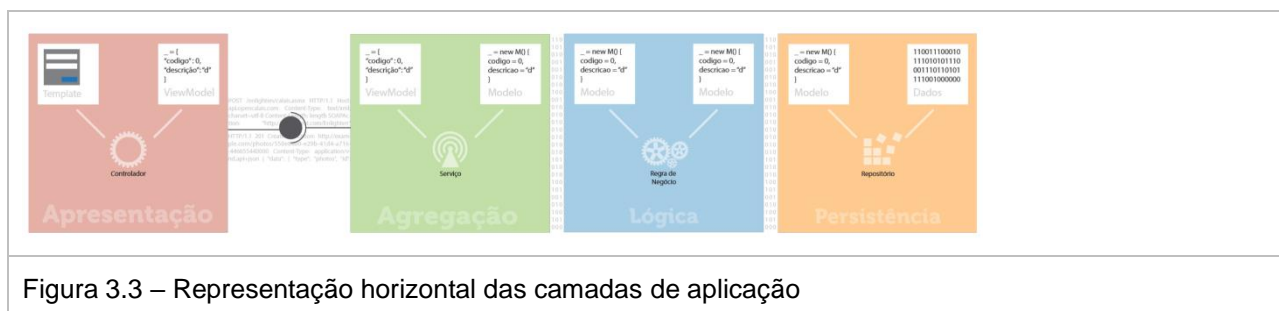


Figura 3.3 – Representação horizontal das camadas de aplicação

Os elementos que implementam uma aplicação são segregados em camadas lógicas. Essa separação permite a determinação precisa das funções realizadas por cada conjunto de classes. Alguns elementos, como os que pertencem aos pacotes extensões e utilitários, não pertencem a nenhuma camada específica, por se tratarem de elementos que têm, por fim, entender as capacidades das plataformas que hospedam as aplicações. Nesses casos, para fins da arquitetura, esses elementos serão considerados parte da API básica de cada plataforma.

Portanto, uma aplicação é conceitualmente dividida em quatro camadas:

- A *camada de persistência* reúne as classes, as quais recuperam e atualizam informações em repositórios de dados. Esses elementos abstraem as complexidades inerentes do acesso a componentes externos especializados em armazenamento e recuperação de informações.
- A *camada de lógica* contém classes chamadas "regras de negócio", que implementam a lógica particular ao sistema, elementos que manipula as informações do sistema para refletir mudanças de estado.
- A *camada de agregação* reúne o conjunto de elementos, os quais formam a fronteira de serviços expostos pela aplicação. Sua principal função é isolar a camada de apresentação das camadas de lógica e de persistência. Com esse objetivo, a camada de agregação opera, principalmente, pela orquestração dos elementos dessa camada e pela conversão de dados entre os tipos binários (classes utilizadas na plataforma de execução) e os tipos serializados (representação JSON utilizada pela camada de apresentação, por exemplo).
- A *camada de apresentação* captura estímulos dos usuários, interpreta suas instruções, envia e recebe dados contra os elementos da camada de agregação e apresenta as informações de forma humanamente compreensível. Camada mais externa do sistema e única que realiza interações diretas com o usuário. Composta pelos elementos dos pacotes extensões, utilitários, diretivas, controladores e configurações.



Essa divisão aplica-se tanto à camada de visão lógica quanto à implementação do sistema: os elementos pertencentes a uma camada restringem-se aos de pacotes pré-determinados, os quais contém apenas elementos de determinados estereótipos. Essa correspondência entre elementos de implementação e elementos lógicos permite a verificação da consistência de uma aplicação frente a arquitetura pela utilização de regras de validação.

Há, por exemplo, regras para a referência em código de elementos de uma determinada camada para as demais, conforme citado ao longo deste texto. Considerando-se a camada de persistência como a mais interna e a de apresentação, a mais externa, em especial não é permitido a elementos de uma determinada camada fazer referência a elementos de outras mais exteriores. Há também regras para a implementação de funcionalidades em determinados pacotes, os quais permitirão o rastreio da divisão de responsabilidades entre os elementos de uma aplicação.

#### 3.2.1.2.1 Camada de acesso de dados

Na camada de acesso a dados, estão codificadas todas as operações pertinentes a persistência de informações. Nenhuma manipulação de dados será realizada em componentes que pertençam a quaisquer outras camadas, em exceção às necessárias e convenientes durante a interação com o usuário.

Nesta camada, encontram-se classes de três naturezas: aquelas que representam as informações manipuladas, de estereótipo *Model*; aquelas que cuidam da persistência e recuperação dessas informações, de estereótipo *repositório*; e aquelas que realizam operações sobre essas informações, de forma individual ou em conjuntos (ou coleções), de estereótipo regras de negócio.

*Classes de estereótipo Model* representam informações com correlação semântica, desde que essa semântica não seja a de coleção de itens ou de coleções aninhadas. Uma classe *Model* contém, por exemplo, informações pertinentes a uma pessoa, ou um lugar, ou uma operação transacional. Classes desse estereótipo podem conter anotações e metadados necessários para o funcionamento dos mecanismos de persistência, mas não deverão ter nenhum tipo de dependência das classes dessas camadas que pertençam aos demais estereótipos, direta ou indiretamente. É vetado, também, que represente um conjunto de objetos resultados de uma consulta ou operação (semântica de coleção).

Não há exigências quanto aos critérios de normalização dos dados de uma classe *Model*. Ela poderá conter atributos *multivalorados* ou de tipo *coleção*, desde que essa coleção tenha relacionamento semântico com o restante das informações e que o propósito da classe não seja servir exclusivamente como resultado de operações (como consultas ou processamentos). Para essa finalidade, devem ser utilizadas as classes disponíveis na plataforma de implementação.

*Classes de estereótipo repositório* têm a finalidade primária de concentrar as operações de persistência de dados em repositórios diversos.

As *classes de estereótipo regra de negócio* concentram as operações, as quais não têm por consequência implícita a persistência de dados. Entre elas, encontram-se operações que alteram o estado interno das classes de estereótipo *Model* e processamento de coleções de objetos dessa classe. Por exemplo, uma operação que processe um conjunto de itens referentes a uma nota fiscal e realize as operações necessárias no sistema para sua efetivação (ajuste de estoque, financeiro, etc).

É vedado que operações das classes de estereótipo *regra de negócio* realizem ou dependam diretamente da persistência das informações que opera. A orquestração entre persistência e operações de regra de negócio deve ser realizadas no ponto em que as essas classes são originalmente instanciadas, em geral em classes da *camada de agregação*.

Quando conveniente, classes de estereótipo repositório poderão ser, também, do estereótipo *regras de negócio*, desde que os métodos de cada tipo sejam claramente segregados no código fonte e que sigam as diretivas aqui indicadas. Essa mescla é indicada em situações de aplicações com poucas regras de negócio ou quando estas forem simples ou, ainda, quando julgar que diminui benéficamente a complexidade do código fonte.

#### **3.2.1.2.2 Camada de apresentação**

A camada de apresentação contém código e recursos necessários para capturar ações realizadas pelo usuário e representar as informações da aplicação de forma acessível a ele. Entre os recursos, encontram-se gráficos, textos, marcações e elementos de interação (e.g. botões, caixas de texto, etc).

Aqui, as figuras de classes e objetos dão lugar a um conjunto de aplicações especializadas que interagirão contra a camada de agregação para realizar as ações do sistema.

A camada de apresentação pode conter os seguintes elementos em sua composição: controladores, *viewmodels* e *templates*.

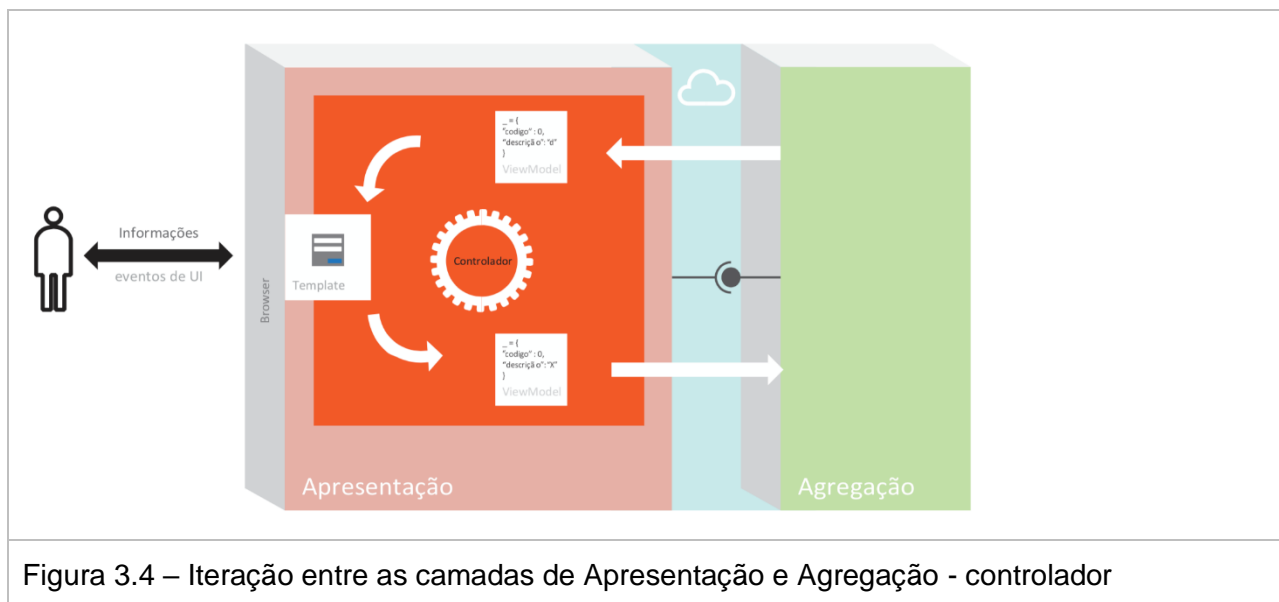
Controladores são os elementos centrais que orquestram o funcionamento dos demais. Suas funções principais são recuperar modelos através de chamadas à camada de agregação, associar modelos a *templates*, reagir aos eventos capturados e informações recebidas, e disparar processos solicitados pelos usuários, novamente a partir de chamadas à camada de agregação. Em suma, controladores representam o corpo do código da camada de apresentação.

*ViewModels* (DTO) são containers para os dados que trafegam entre serviços, controladores e templates. A grosso modo, têm as mesmas características de objetos *Model*: contém apenas

atributos e nenhuma lógica interna, a não ser a necessária para manipular seus valores internos (setters e getters) ou gerar atributos calculados. A priori, *viewmodels* são criados e utilizados nas chamadas e respostas dos serviços, mas a criação de *ViewModels* internos é conveniente para simplificar a interação de controladores com *templates*. Por serem gerados a partir das informações recuperadas de serviços, *ViewModels* coincidem regularmente com classes de persistência (quando o recurso exposto por um serviço de consulta e atualização de dados refletir fielmente uma classe de persistência, por exemplo). Porém, não existe relação estrita entre esses dois elementos: em chamadas serviços compostos, por exemplo, *ViewModels* refletirão, de forma geral, os parâmetros necessários à execução daquele serviço.

*Templates*, por sua vez, são os elementos da interface do usuário: construções que exibem e capturam informações. Em geral, *templates* são associados a *ViewModels* e geram a marcação do código visual necessário para exibir e/ou manipular as informações contidas ali. Por exemplo, um *ViewModels* associado a um *template* de formulário gerará os rótulos e caixas de texto necessárias para capturar e editar aquelas informações.

Quando necessário construir uma interação mais complexa ou um componente visual que não esteja disponível ou possa ser composto pelos disponíveis na biblioteca padrão, novos *templates* podem ser construídos por código.



### 3.2.1.2.3 Camada de serviços

A camada de serviços concentra classes de dois estereótipos (serviços e *ViewModels*).

Os serviços implementarão os canais de serviços REST da aplicação e servirão como fronteira mais externa da aplicação em relação à infraestrutura de execução, tanto pública (internet) quanto privada (intranet). O conjunto de serviços de uma aplicação formam sua API pública. Esta

serve como ponto de integração para todas funcionalidades da aplicação, tanto aquelas que necessitam de uma interface para interação com o usuário, quanto aquelas disponíveis para integração com outros sistemas.

Cada serviço é implementado como uma rota dentro de um controlador MVC (*classes de estereótipo controlador*). Controladores, portanto, podem ser vistos como grupos de serviços. O critério para o agrupamento poderá ser funcional (ou horizontal), quando um controlador reúne serviços que realizam operações semelhantes em classes diferentes, ou semântico (ou vertical), quando um controlador reúne serviços referentes a um único recurso (ou entidade, ou classe de negócio) do sistema. A escolha por um ou outro tipo de agrupamento deverá levar em conta a conveniência e a quantidade de itens que serão agrupados em cada controlador.

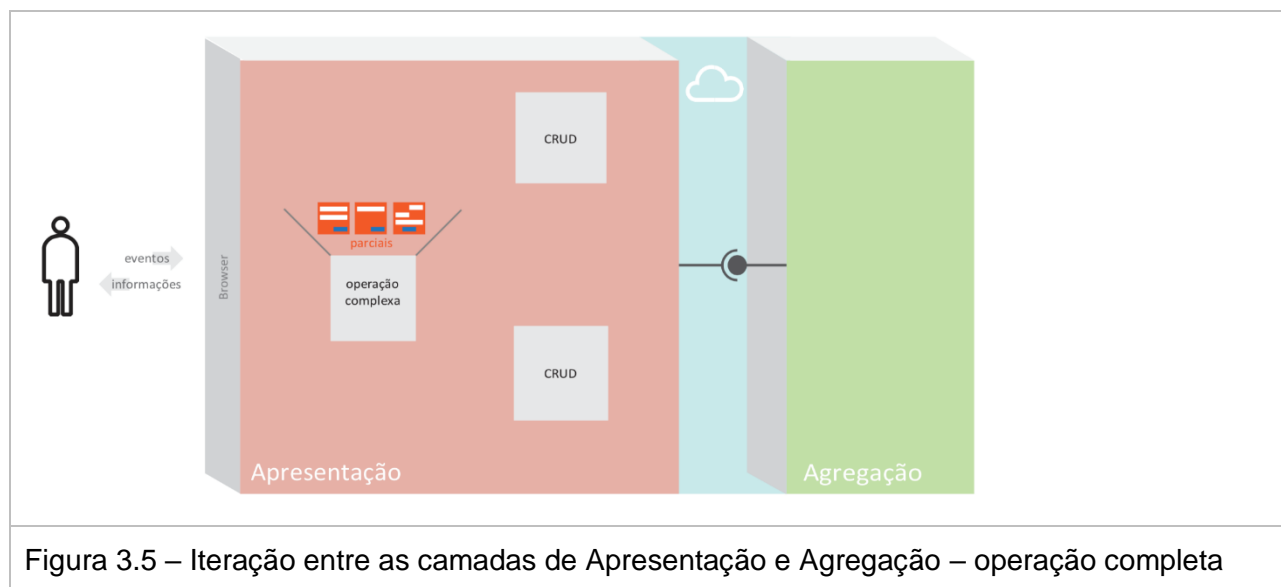
A priori, cada serviço pertence a uma das seguintes categorias: serviços de consulta e atualização de recursos, que costumam ser agrupados pelo critério funcional; ou serviços compostos (ou de operações transacionais), cuja implementação realizará operações sobre recursos (ou entidades) diferentes, que costumam ser agrupados pelo critério semântico e que, quando realizam atualização de informações, envolvem uma transação ACID implícita.

Como exemplos de serviços de consulta e atualização de recursos, cita-se a exposição de serviços REST tradicionais sobre as entidades da aplicação: por exemplo, um serviço de consulta e atualização do catálogo de produtos em um sistema de controle de estoque. Não é mandatório que um recurso exposto corresponda exatamente a uma entidade representada em uma classe *Model* persistida em um repositório, o recurso pode ser composto internamente a partir de repositórios e classes diferentes, criando funcionalidades similares aos serviços implementados sobre uma única classe *Model*. Nesse caso, o serviço deverá realizar as composição e decomposição necessárias para prover a funcionalidade descrita. Cada serviços de consulta e atualização de recursos costuma ter por denominação um substantivo (e.g. / produto) e implementar diversos verbos REST sobre um mesmo recurso.

Serviços compostos utilizam elementos da camada de acesso a dados para compor operações complexas e realizar a persistência das informações atualizadas ao seu final. Também representa um serviço composto válido aquele que é formado de diversas consultas, sobre uma mesma entidade ou sobre entidades diferentes, com a finalidade de montar representações de dados que exijam estruturas de informação complexas como, por exemplo, grafos e árvores. Esse caso distingue-se dos serviços de consulta e atualização de recursos com composição, conforme descrito anteriormente, porque não permitirá atualização de dados e por não ser sua finalidade causar uma ilusão de recurso unificado para o exterior da aplicação. Um exemplo típico de serviço composto é aquele que provê uma API para operações transacionais típicas de um sistema de controle de estoque: de saída, entrada ou transferência de produtos nesse sistema. Serviços

compostos costumam ter verbos por denominação (e.g. /vender) e implementar apenas um verbo REST.

Os *ViewModels* são representações simplificadas das classes de estereótipo *Model*, pertencentes à camada de acesso a dados. A priori, não é vedada a reutilização das classes de estereótipo *Model* na função de *ViewModels*. Porém, como as classes de estereótipo *Model*, em geral, contém mais informações que o necessário ou desejável para sua utilização, além da fronteira de serviços da aplicação. A criação de uma representação simplificada (*ViewModel*) é um artifício utilizado de forma regular. Apesar de representar, de uma certa forma, redundância de código, essas classes têm a virtude de simplificar conversões de dados na fronteira de serviços (de ambos os lados, cliente e servidor), e obfuscar as estruturas internas da aplicação para o mundo exterior, aumentando tanto a performance da comunicação em rede quanto o grau de segurança da aplicação.



### 3.2.2 Visão de implementação (ou de componente)

Nessa visão, existe o desenvolvimento do que foi especificado na visão lógica. Entretanto, em um modelo ágil, no qual é possível haver mudanças ou ajustes, existe uma interação periódica do que está sendo desenvolvido para melhor aderência ao plano da visão lógica e minimizar os problemas que podem surgir durante o desenvolvimento.

Em resumo:

- Envolve diretamente os desenvolvedores, ainda que exista também um direcionamento pela equipe de análise e gerência do projeto;
- Implementação dos módulos e suas dependências são descritos;

- É utilizada para saber como distribuir o trabalho de implementação e manutenção entre os membros da equipe considerando aspectos de reuso, subcontratação e aquisição de softwares que auxiliem o projeto.

A CAPES utiliza a arquitetura em camadas já descritas e os pacotes, as nomenclaturas, entre outros aspectos estruturais.

### **3.2.3 Visão de processos (ou concorrência)**

Na visão de processos é explicitado pontos tais como desempenho, escalabilidade, vazão das informações e tolerância a falhas. Além disso, os processo do sistema e como eles se comunicam e as atividades do seus funcionamentos deverão ser descritos

Em resumo:

- Trata a divisão do sistema em processos e processadores (propriedade não funcional);
- O sistema é dividido em linhas de execução de processos concorrentes;
- Esta visão de concorrência deverá mostrar como se dá a comunicação e as concorrências;
- Considera questões de desempenho, confiabilidade, tolerância a falhas.

### **3.2.4 Visão de dados**

Fornece uma perspectiva de armazenamento de dados persistentes do sistema. A visão de dados é opcional se os dados persistentes forem poucos ou inexistentes ou se a conversão entre o Modelo de Design e o Modelo de Dados for trivial.

### **3.2.5 Visão física (ou implantação)**

A visão de implementação mostra a distribuição física do processamento no sistema. Nessa visão, são especificados os maquinários e meios envolvidos para o funcionamento do sistema implementado.

O advento da virtualização e containerização (ver DevOps) não exige a existência deste modelo, muito pelo contrário, se torna imprescindível pois existe o aumento da quantidade de pontos de falhas e exige uma análise mais rigorosa. Felizmente existem formas de mitigar esse problema (ver DevOps).

Em resumo:

- Conter a parte física do sistema e a conexão entre suas sub-partes, interação hardware e software, com objetivo de colocar o sistema em operação;

- Mostrar a organização física do sistema, os computadores, os periféricos e como eles se conectam entre si;
- Esta visão será executada pelos desenvolvedores, integradores e testadores, e será representada pelo diagrama de implantação, pois considera o ambiente de desenvolvimento, teste e produção.

### **3.3 DevOps**

DevOps tem em seu cerne a evolução e melhoria contínua do processo de entrega de software e é fundamentado em 4 pilares: Comunicação, Colaboração, Automação e monitoria.

#### **3.3.1 Comunicação**

Ao criar o software, a equipe precisa de constante comunicação para o planejamento e liberação do código, bem como resolução de impedimentos. Na CAPES, foi adotado as ferramentas Redmine e Mattermost. Redmine é utilizado para gerenciamento da metodologia ágil; Mattermost, para comunicação.

Para maiores informações sobre ambas as ferramentas, analisar a documentação de uso do Mattermost e Redmine em seus respectivos sites oficiais.

#### **3.3.2 Colaboração**

O processo de colaboração é executado pelo próprio time. Devendo o gerente ser o responsável por facilitar o processo em casos de impedimentos.

#### **3.3.3 Automação**

A automação é o coração de todo o processo DevOps e deve tornar os procedimentos determinísticos e estáveis. Os procedimentos serão detalhados na seção padrões para execução da automação.

#### **3.3.4 Monitoramento**

O monitoramento pelas equipes DevOps tem como papel fornecer informações, as quais garantam o tempo de atividade do serviço e o desempenho ideal em tempo de execução (*runtime*).

#### **3.3.5 Padrão para execução da automação**

Para o correto entendimento de onde a automação se encaixa dentro do processo, exige-se o necessário entendimento dos elementos, abaixo elencamos, que compõem a automação :

##### **3.3.5.1 Kubernetes**

A CAPES utiliza o Kubernetes como orquestrador do ambiente de execução das aplicações. Mais especificamente a distribuição OpenShift. As aplicações são disponibilizadas em ambientes containerizados e colocados em execução pelo orquestrador.

Alguns dos benefícios da adoção da ferramenta são:

- Igualdade dos ambientes de desenvolvimento das aplicações até a produção



- Criação de ambientes dinâmicos
- Auto *scale* das aplicações em caso de picos de utilização

### 3.3.5.2 GitOps

GitOps é a utilização do Git como fonte única verdadeira para implantações de ambientes. Geralmente as modificações são aplicadas ao git por meio de um commit, gerando a execução de um pipeline para a execução de uma atividade qualquer relacionada a informação que foi colocada no repositório.

Na CAPES, o padrão gitops é utilizado para o desenvolvimento de aplicações, de forma que o commit do usuário acione o *pipeline* que efetuará a validação, building, testes e *deployment* do código alterado para o ambiente de desenvolvimento até a produção.

### 3.3.5.3 Ambientes de execução

A ferramenta OpenShift permite que sejam criados quantos ambiente forem necessários para a implantação de aplicações. Entretanto definimos 4 ambientes distintos para o ambiente CAPES:

*Revisão* - são ambientes dinâmicos criados sob demanda e temporários para a apresentação de uma funcionalidade, eles trabalham ligados a uma feature branch do gitflow.

*Homologação* - ambiente esse que é utilizado pelo cliente para a homologação do sistema.

*Integração contínua* - ambiente dinâmico usado para a execução automatizada dos testes de integração, carga e performance.

*Produção* - ambiente onde a aplicação é executada para atender a demanda externa dos usuários.

Embora esses sejam os ambientes padrões para os sistemas, nada impede que possam se criados novos ambientes para atendimento das especificidades das aplicações.

### 3.3.5.4 Pipeline

Conjunto de passos executados em uma sequência lógica para que um objetivo final seja atendido. O *pipeline* é integrado ao Git de forma que *commits* do usuário acionem sua execução.

Abaixo, vemos um exemplo de *pipeline* com 4 fases: preparação, building, teste, análise de segurança.





Em caso de falha de uma das etapas, o commit é marcado como defeituoso pois não atendeu as exigências do pipeline e isso bloqueia a continuidade da execução do *pipeline*.

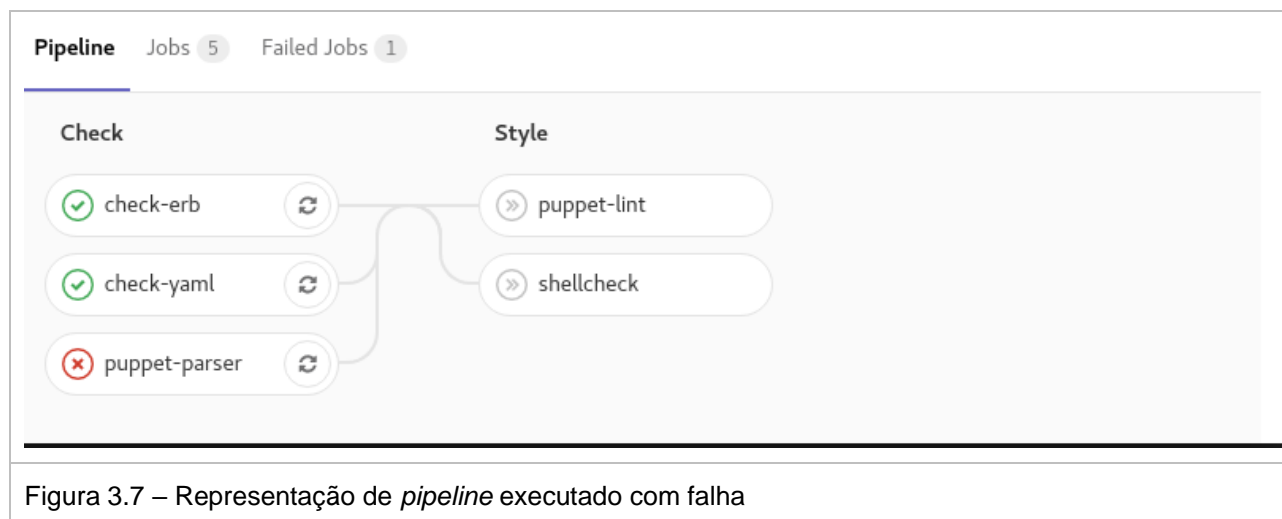


Figura 3.7 – Representação de *pipeline* executado com falha

Um pipeline pode ser customizado para se adequar as necessidades do projeto. É no pipeline que são definidos quando os testes rodam, quando os procedimentos de deployment são executados ou quando uma análise de código é necessária.

Na CAPES, um pipeline básico deverá conter as seguintes fases para cada tipo de branch segundo o gitflow:

#### 3.3.5.4.1 Feature branch

São executados os seguintes passos a depender do suporte da linguagem:

- *Lint do código* – analisar se o código foi bem construído segundo os padrões estabelecidos para a linguagem ou pela CAPES.
- *Testes unitários* – executar testes unitários codificados na aplicação.
- *Análise de vulnerabilidades (SAST)* – executar análise de dependências e se não possui vulnerabilidades reportadas no CVE.
- *Compilação* – executar resolução de dependências, building da aplicação, para os casos de aplicações compiladas, e empacotamento em container para execução no OpenShift.
- *Review* – acionamento manual e sob demanda. Criar ambiente que executa a aplicação para uma visualização da funcionalidade implementada.
- *Limpeza* – acionamento manual e sob demanda. Remover ambiente de revisão.

#### 3.3.5.4.2 hotfix branch, release branch

- *Lint do código* – analisar se o código foi bem construído segundo os padrões estabelecidos para a linguagem ou pela CAPES.

- *Testes unitários* – executar testes unitários codificados na aplicação.
- *Compilação* – executar resolução de dependências, building da aplicação, para os casos de aplicações compiladas, e empacotamento em container para execução no OpenShift.
- *Testes de integração / aceitação / pentest (DAST)* – executar testes de integração ou aceitação (via Selenium) em conjunto com *pentest* (via OWASP zed *Attack proxy*) para análise DAST.
- *Testes de carga / performance* – executar testes de carga e de análise de tempo de resposta da aplicação
- *Review* – acionamento manual e sob demanda. Permitir criação de um ambiente que executa a aplicação para uma visualização da funcionalidade implementada.
- *Limpeza* – acionamento manual e sob demanda. acionamento manual e sob demanda. Remover ambiente de revisão.

#### 3.3.5.4.3 Tags

- *Compilação* – executar resolução de dependências, building da aplicação, para os casos de aplicações compiladas, e empacotamento em container para execução no OpenShift.
- *Deployment* – acionamento manual e sob demanda. Publicar aplicação em ambiente de *staging*.
- *Liberação* – acionamento manual e sob demanda. executar testes de liberação da aplicação para acesso externo.
- *Limpeza* – acionamento manual e sob demanda. acionamento manual e sob demanda. Remover ambiente de *Liberação*.

O arquivo de definição do pipeline é o `.gitlab-ci.yml`. Esse arquivo define a estrutura de execução do pipeline.

Abaixo vemos um exemplo de um pipeline do Gitlab:

```
.gitlab-ci.yml 371 Bytes
1  stages:
2    - Testes
3    - Deploy
4
5  Validação código PHP:
6    stage: Testes
7    script:
8      - composer install
9      - ./vendor/phpunit/phpunit/phpunit --testdox
10   except:
11     - tags
12
13  Deploy em homolog:
14    stage: Deploy
15    script:
16      - sh ./deploy-hom
17    only:
18      - homolog
19
20  Deploy em produção:
21    stage: Deploy
22    when: manual
23    script:
24      - sh ./deploy-prod
25    only:
26      - tags
```

Figura 3.8 – Exemplo de configuração de .gitlab-ci.yml.

Neste arquivo, ficam apenas as definições do *pipeline*. Devendo os scripts de execução das operações serem colocados na pasta *devops/pipeline-scripts*.

### 3.3.5.5 Scripts

Todos os procedimentos executados devem ser dispostos em *scripts*. Essa diretriz visa garantir a replicabilidade da execução dos procedimentos. Por isso, os scripts devem ser armazenados na DevOps.

#### 3.3.5.5.1 Scripts de banco

Os scripts de banco de dados têm como finalidade o controle das modificações necessárias para a correta execução da aplicação, quer seja para desenvolvimento, quer seja para execução de testes automatizados ou aplicação em produção.

Eles devem estar no formato SQL e estarem sob a pasta *devops/scripts-banco*. Deve ser utilizado o *Liquibase* para gerenciamento das modificações de banco de dados. Os arquivos contendo a instrução sql de modificações de estruturas (DDL) ou migração / ajustes de dados (DML) devem usar o padrão *YYYY-MM-DD-HH-mm\_<descrição simples da modificacao>.sql*, conforme recomendação do liquibase.

### 3.3.5.6 Operators

Operators é o novo padrão para gerenciamento do ciclo de vida de aplicações expostas no cluster Kubernetes.

A função principal de um *Operador* é encapsular o conhecimento de infraestrutura de uma aplicação. De forma a tornar mais simples os procedimentos necessários a manutenção da aplicação.

Existe o operador SDK que é a ferramenta a ser utilizada para o desenvolvimento de operadores para a plataforma OpenShift.

Os operadores são registrados no repositório do núcleo da arquitetura no Git da CAPES. (<https://git.capes.gov.br/cgs/narq/openshift/operators>)

O funcionamento de um operador permite estender as funcionalidades do OpenShift de forma a gerenciar objetos personalizados. Abaixo vemos um exemplo de um objeto personalizado:

```
1  apiVersion: "apps.capes.gov.br/v1"
2  kind: "Sistema"
3  metadata:
4    name: "financeiro"
5  spec:
6    servicos:
7      - imagem: "financeiro-web:1.1.8"
8        replicas: 1
9        recursos:
10          cpu: 400m
11          memoria: 2Gi
12        firewall:
13          portas-liberadas:
14            - nome: financeiro-ws
15              porta: 8080
16              protocolo: TCP
17            - nome: financeiro-web
18              porta: 8080
19              protocolo: TCP
20          saida:
21            - nome: acesso site serpro
22              dns: serpro.gov.br
23          acesso-externo:
24            - rota: financeiro.capes.gov.br
25              porta-liberada: financeiro-web
26
```

Figura 3.10 – Exemplo de configuração de Operator

Ao aplicar esse objeto customizado o sistema é capaz de implantar a versão 1.1.8 do sistema financeiro. De forma que as regras descritas nesse documento sejam aplicadas ao cluster OpenShift.

Para uma descrição completa da sintaxe desse arquivo verifique a versão mais recente da documentação do operador sistemas-capes, que encontra-se no Git da CAPES.

### 3.4 Controle de versão

Para o gerenciamento de versão de código, a CAPES adotará o Git e Gitflow Workflow (*Gitflow*) como fluxo de versionamento. No Gitflow, há dois branches essenciais: *develop* e *master*. O *develop* serve de ponto de integração das *Features* (funcionalidades) finalizadas e estáveis, as quais precisam ser integradas ao

projeto. O *develop* deverá ser composto apenas por código estável. O *master* tem dois propósitos: i) versionar o código a ser publicado em produção, e; ii) servir de ponto de partida para o código que será usado para correção em produção (*hotfix*).

Além dos *branches develop* e *master*, há, no Gitflow, mais 3 prefixos: *Feature*, *Release* e *Hotfix*.

*Feature*<sup>1</sup> representa uma marcação no tempo de versionamento que aponta para um conjunto de código. Uma *feature* é criada para a codificação de novas funcionalidades no projeto e tem origem a partir do *develop*.

*Release* representa um conjunto de *features* que serão disponibilizadas em produção. Uma *Release* é criada sempre a partir do *develop*.

*Hotfix* representa uma marcação no versionamento que surge sempre a partir do *master* para corrigir um código em produção.

### 3.4.1 Fluxo de trabalho

O fluxo de trabalho com o Gitflow descreve o passo a passo no gerenciamento de *branches*, sua origem, transição e fechamento <sup>2 3</sup>.

---

<sup>1</sup> Uma *Feature* jamais deverá interagir com o *Master*

<sup>2</sup> A Capes fornecerá o endereço do repositório e credenciais em momento oportuno durante a execução do projeto.

<sup>3</sup> Sem prejuízo das etapas de QA e demais processos especificados neste documento.

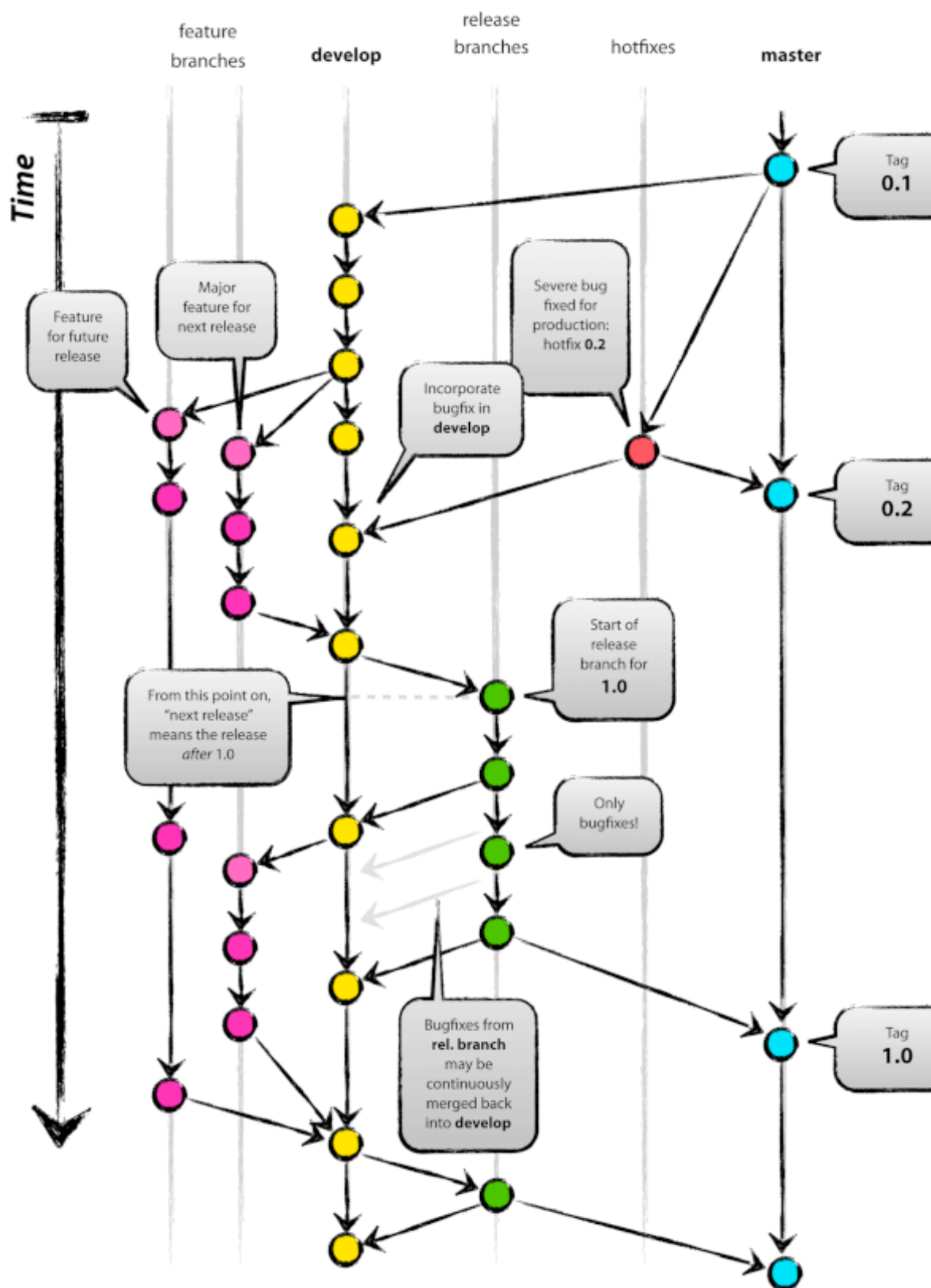


Figura 3.11 - Representação gitflow

Para cada funcionalidade, criar-se-á uma *feature branch*. A esta *feature*, poder-se-á efetuar *commits* parciais até que seu código esteja pronto para se juntar ao *develop*.

A entrega do *feature* se dá com o merge da *feature branch* no *develop*.

*Release branch* é usado para juntar um conjunto de funcionalidades a serem disponibilizadas em produção. Todas as possíveis correções necessárias a estabilização dessas funcionalidades, dar-se-ão diretamente na própria *release branch*. Tendo a *release branch* sido estabilizada, seu fechamento implicará em *merge* tanto no *master* quanto no *develop*.

Correções em código de produção, *master*, dar-se-á com o uso de *hotfix*. Ao término, fechamento do *hotfix*, implicará em *merge* tanto no *master* quanto no *develop*.

### 3.5 Requisitos de QA

Sem comprometimento dos itens relacionados à qualidade de software descritos noutros tópicos deste documento, a Capes, com o objeto de garantir a qualidade interna e externa de seus sistemas, adotará como requisitos de qualidade de software os elencados no item *Pré-requisitos de qualidades*. Tais requisitos visam a garantia dos tópicos do Modelo de qualidade para qualidade externa e interna definida na ISO/IEC 9126:2003 - Tópico 6 da norma, resumido na Figura 3.12.

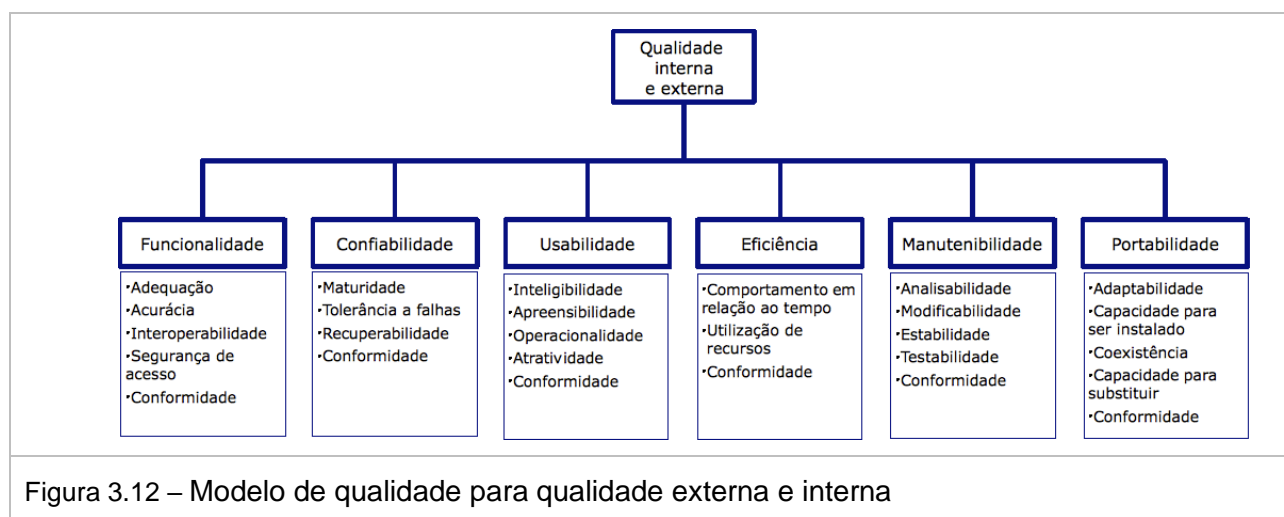


Figura 3.12 – Modelo de qualidade para qualidade externa e interna

#### 3.5.1 Pré-requisitos de qualidade

Cada um dos itens elencados na *tabela 00* será detalhado nos guias de arquitetura de tecnologia específica, bem como sua exigibilidade será avaliada pela CAPES conforme o projeto.

Requisito	Descrição	Entregável
<b>Teste de Unidade</b>	Teste de componente ou classe.	Sim
<b>Teste de Integração</b>	Teste que garante que componentes funcionem de forma combinada	Sim
<b>Teste de Regressão</b>	Teste que garante que uma alteração seja compatível com os demais componentes	Sim

Requisito	Descrição	Entregável
<b>Teste de Caixa Preta</b>	Teste que entrada e saída esteja em conformidade com o especificado	Sim
<b>Teste Funcional</b>	Teste de funcionalidades, requerimentos, regras de negócio	Sim
<b>Teste de Interface</b>	Teste da navegabilidade e os objetos de tela	Sim
<b>Teste de Performance</b>	Teste de tempo de resposta	Sim
<b>Teste de Carga</b>	Teste do sistema com um número elevado de usuários simultâneo	Sim
<b>Teste de Stress</b>	Teste de avaliação do tempo de resposta de uma ou mais funcionalidade em seu caminho feliz	Sim
<b>Teste de Configuração</b>	Teste de avaliação do funcionamento em diferentes ambientes de hardware/software	Sim
<b>Teste de Instalação</b>	Teste de instalação da aplicação se foi bem sucedida seguindo o roteiro de instalação	Sim
<b>Teste de Segurança</b>	Teste de segurança da aplicação contra falhas conhecidas (CVE)	Sim

Tabela 3.4 – Pré-requisitos de qualidade

### 3.5.2 Automação

A automação de testes, quando aplicável, será detalhada no Guia de Arquitetura específico de cada linguagem de programação.

### 3.5.3 Cobertura de código por teste

O percentual de cobertura de código será definido nos Guias de Arquitetura específicos de cada linguagem de programação (*Guia Arquitetura Java*, *Guia Arquitetura Javascript*, *Guia Arquitetura PHP*, *Guia Arquitetura Python*) e quanto cabível, elencado por camada da aplicação.



## 4 Referências

- ISO/IEC 9126 - <https://www.iso.org/standard/22749.html>
- The Art of Computer Programming: D. E. Knuth, Addison-Wesley (volumes 1--3, 4A), 1998.
- Spring Guides: <https://spring.io/guides>
- Hibernate Docs: <http://hibernate.org/orm/documentation/5.0/>
- Docker para desenvolvedores: Rafael Gomes, Leanpub, 2017

## 5 O Anexo A – Guias de arquiteturas

Nome do guia de referência	Tecnologia
Arquitetura de referência - Java	Java
Arquitetura de referência – PHP	PHP
Arquitetura de referência – Python	Python
Arquitetura de referência - Frontend	-