Unidade 5

# Aula 1

# Padrões de Projeto - Parte 1

Os padrões de projeto são muito importantes no desenvolvimento de aplicativos móveis. Os smartphones possuem recursos limitados, então qualquer erro de lógica de programação pode acarretar em perda de desempenho, alto consumo de bateria e travamentos, então preparamos duas aulas com alguns padrões de projetos importantes para você. Em aulas anteriores vimos o padrão Composite que nos ajuda a entender como funciona a estrutura e relações de Views e ViewGroups em um layout. Esta aula será voltada totalmente aos padrões de projeto. Começando pelo padrão Template Method, você irá entender como funciona o ciclo de vida de uma Activity e um Fragment. Depois com o Singleton veremos como acessar instâncias compartilhadas, que o próprio SDK do Android disponibiliza. Entendendo estes padrões fica mais fácil de entender o que acontece “por trás dos panos” no Android, e ainda é um diferencial importante para um profissional desenvolvedor mobile.

## Template Method

### Objetivo

* Define uma classe esqueleto de um algoritmo em uma operação, ou seja, define uma sequencia de passos a ser executado sendo que alguns destes passos são adiados para suas subclasses. O Template Method permite que subclasses redefinam determinadas etapas de um algoritmo sem alterar a estrutura do mesmo.
* Serve como classe base que declara um algoritmo de "espaços reservados", e as classes derivadas devem implementar estes espaços reservados. Estes espaços reservados podem ser entendidos como um método abstrato por exemplo, onde as subclasses irão implementá-lo, ou seja, preencher este espaço.

### Problema

Dois componentes diferentes podem ter semelhanças significativas, mas não demonstram a reutilização de interface ou de aplicação de algum código comum. Se esta semelhança tiver que ser modificada, é necessário um esforço maior para tratar o comportamento de todas as subclasses (ou componentes).

### Discussão

O projetista do componente decide quais passos de um algoritmo são invariáveis (ou padrão), e quais são variáveis (ou personalizados). Os passos invariáveis são implementados em uma classe base abstrata, enquanto os passos variáveis são implementados em suas subclasses. Os passos variáveis representam "ganchos", ou "espaços reservados", que podem, ou devem, ser fornecidos pelo cliente (entende-se como cliente, uma subclasse, ou a classe que o consome) do componente em uma classe derivada concreta (entende-se como classe concreta, a classe que não é abstrata).

**DICA:** Você acabou de ler um bombardeiro de termos técnicos. Não se preocupe, vamos explicar. Esta é uma definição formal e bastante técnica do Template Method que encontramos em livros deste assunto, mas vamos entender melhor estes termos.

O **projetista de componentes** somos nós, programadores, pois quando programamos em Linguagem Orientada a Objetos, devemos nos preocupar a todo tempo em desenvolver algo desacoplado, pronto para reúso, e esta é a característica de um componente.

O **algoritmo invariável** seria um trecho de código que já está determinado e não precisa ser alterado em tempo de execução (enquando o software está em execução). Já o **algoritmo variável**, é o trecho de código que é sensível ao ambiente, ou seja, de acordo com o uso do software, uma determinada funcionalidade tem comportamento diferente.

Os **ganchos** e **espaços reservados** se tratam deste algoritmo variável, ou seja, são métodos abstratos que devem ser implementados pelas subclasses (ou **classe derivada concreta**).

O projetista do componente obriga os passos necessários de um algoritmo, e a ordem dos passos, mas permite que o cliente do componente estenda ou substitua uma quantidade destes passos.

Template Method é usado de forma proeminente nos frameworks. Cada framework (estruturas de reuso de larga escala) implementa as peças invariáveis de arquitetura de um domínio, e define "espaços reservados" para todas as opções de personalização do cliente necessárias ou interessantes. Ao fazê-lo, o framework torna-se o "médico", e as personalizações do cliente são os "enfermeiros", ou seja, o médico tem o procedimento e o enfermeiro executa o procedimento dado ao médico. Mas o médico também executa partes deste procedimento. Então fazendo uma analogia, o framework determina um procedimento e delega algumas partes as classes clientes, mas alguns passos são executados pelo próprio framework. Esta estrutura de controle invertida foi carinhosamente denominada "o princípio Hollywood" - "não nos chame, nós o chamaremos".

Estas definições são sempre um tanto complicadas. Isto porque, para obtermos total compreendimento destes padrões teremos que executá-los na prática. O diagrama a seguir e a aplicação em um exemplo do SDK Android podem esclarecer um pouco este padrão, mas dica é: Esteja sempre programando e buscando soluções baseadas nos padrões de projeto existentes. Além de aumentar a sua produtividade (pois os padrões de projeto se tratam de soluções prontas para serem aplicadas)

### Estrutura

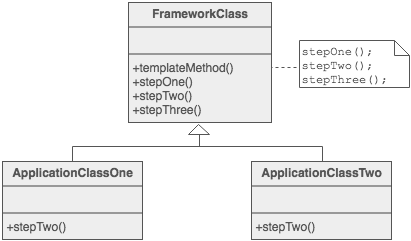


Figura - Diagrama conceitual do padrão Template Method

A implementação do método **templateMethod()** é: chamar **stepOne()**, chamar **stepTwo(),** e chamar **stepThree()**. Vendo pelo diagrama de classes, o **stepTwo()** é um método “gancho” – ou um espaço reservado.

Ele é declarado na classe base (Framework Class) e então é implementado nas subclasses (classes clientes). Muitos frameworks conhecidos usam Template Method. Todo código reusável é definido nas classes base do framework, e então os clientes deste framework são livres para customizar funcionalidades criando classes derivadas.

**DICA**: Podemos chamar estes ganchos também de **método de retorno de chamada**, que é o nome dado na documentação do Android. Dá-se este nome devido a uma tradução livre do termo **call-back**, onde um objeto delega certas tarefas e outro objeto, e fica esperando uma resposta que será enviada quando a tarefa delegada terminar. Geralmente é utilizado para tarefas assíncronas como obter dados de um serviço da web, receber conexão com um dispositivo Bluetooth ou monitorar o estado da Wi-fi do aparelho.

### Exemplo em Android

Grades frameworks que utilizam o Template Method na API Android são a Activity e o Fragment. Vamos analisar a Activity.



Figura - Ciclo de vida de uma Activity

A imagem acima, já mostrada no item 1.1.1.1 na Unidade 4, representa o ciclo de vida de uma Activity, então temos aqui a primeira característica de um Template Method: Passos ou um fluxo de chamada de métodos definido.

Se você olhar a implementação do método **onCreate()**  na classe Activity, verá que ele não está vazio, isto porque a Activity não é uma classe abstrata. Veja:

@MainThread

@CallSuper

**protected void** onCreate(@Nullable Bundle savedInstanceState) {

**if** (***DEBUG\_LIFECYCLE***) Slog.v(***TAG***, **"onCreate "** + **this** + **": "** + savedInstanceState);

**if** (**mLastNonConfigurationInstances** != **null**) {

**mFragments**.restoreLoaderNonConfig(**mLastNonConfigurationInstances**.**loaders**);

}

**if** (**mActivityInfo**.**parentActivityName** != **null**) {

**if** (**mActionBar** == **null**) {

**mEnableDefaultActionBarUp** = **true**;

} **else** {

**mActionBar**.setDefaultDisplayHomeAsUpEnabled(**true**);

}

}

**if** (savedInstanceState != **null**) {

Parcelable p = savedInstanceState.getParcelable(***FRAGMENTS\_TAG***);

**mFragments**.restoreAllState(p, **mLastNonConfigurationInstances** != **null**

? **mLastNonConfigurationInstances**.**fragments** : **null**);

}

**mFragments**.dispatchCreate();

getApplication().dispatchActivityCreated(**this**, savedInstanceState);

**if** (**mVoiceInteractor** != **null**) {

**mVoiceInteractor**.attachActivity(**this**);

}

**mCalled** = **true**;

}

Apesar de não ser abstrata, nós, projetistas dos componentes, quase nunca iremos utilizá-la puramente, sem fazer uma extensão. Então na prática utilizamos apenas suas subclasses. Mas a vantagem desta não ser uma classe abstrata é que seus métodos não requerem uma implementação obrigatória de suas subclasses.

Se você analisar bem a classe Activity, verá que existem métodos que cuidam do ciclo de vida como se fossem uma máquina de estados. Quando o método onCreate() é chamado, por exemplo, o framework (classe Activity) nos permite adicionar comportamentos adicionais quando o método é chamado. Temos aí outra característica do Template Method.

Você pode interceptar qualquer método do ciclo de vida de uma Activity e adicionar comportamentos a eles, basta você implementá-lo na sua subclasse. Mas não se esqueça: todos os métodos do ciclo de vida que você sobrepõe devem possuir a chamada **super**, senão a Activity não funcionará corretamente.

**DICA:** Quando um método é sobreposto (técnica de Polimorfismo), ou seja, quando substituímos um método da classe mãe por um outro de mesmo nome na classe filha, temos ainda a chance de executar o código contido no método da classe mãe. Isto é possível utilizando a chamada **super**. Sua utilização é parecida com o **this**. O this é utilizado para diferenciar um atributo da classe de uma variável local de mesmo nome, enquanto a chamada super é utilizada para diferenciar um método da classe mãe de um método de mesmo nome na classe filha. Veja nos exemplos que já construímos:

@Override  
**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
  
 setContentView(R.layout.***activity\_login***);  
 ButterKnife.*bind*(**this**);  
  
}

Por fim, a característica “não me chame, nós te chamaremos” também está presente. veja todo código que programamos até agora e responda: Alguma vez nós precisamos invocar um método do ciclo de vida da Activity manualmente?

A resposta é não. Nunca invocaremos o **onPause()** por exemplo, apenas o implementaremos e a máquina de estado cuidará de chamá-lo em momento oportuno.

### Checklist para utilizar o Template Method

1. Examine o código e identifique passos de execução que são padrões e passos que são peculiares.
2. Defina uma classe abstrata (ou não) que consiga dizer: “não me chame, nós te chamaremos”.
3. Mova a casca do algoritmo (agora chamado "template method") e a definição de todas as etapas padrões para a nova classe base (se você optou por utilizar uma classe abstrata).
4. Definir espaços reservados ou métodos "ganchos" na classe base para cada etapa que requer muitas implementações diferentes.
5. Invoque os ganchos a partir do template method.
6. Cada uma das classes existentes declara um relacionamento "é-um" à nova classe base abstrata.
7. Retire das classes existentes todos os detalhes de implementação que foram movidos para a classe base.
8. Os únicos detalhes que permanecerão nas classes existentes serão os detalhes de implementações peculiares a cada classe derivada.

## Singleton

### Objetivo

* Certificar-se que a classe possui apenas uma instância, e provê acesso global a esta instância.
* "just-in-time initialization" ou "inicialização apenas no primeiro uso" é encapsulada.

### Problema

A aplicação precisa de uma, e somente uma, instância de uma determinada classe. Adicionalmente, inicialização “preguiçosa” ( ou em inglês “lazy initialization”- termo dado a inicialização apenas no primeiro uso) e acesso global são necessários.

### Discussão

Nosso objetivo é fazer uma classe de única instância responsável pela própria criação, inicialização, acesso e execução. Para isto declaramos um atributo privado e estático que armazenará uma instância da própria classe. Ao ponto que este membro é estático e privado, devemos fornecer um método estático e público que encapsula todo o código de inicialização e fornece acesso para a instância armazenada neste atributo. Chamamos ele de método **instance()** ou **sharedInstace().**

O cliente chamará apenas este método assessor, e os demais métodos da classe são chamados apenas por intermédio deste método.

Por segurança você pode **privatizar** os construtores da classe para que ninguém consiga inicializar esta classe, já que o objetivo é utilizar uma instância única dela. Quem irá inicializá-la é o método **sharedInstance()** caso já não esteja.

**DICA:** Um ponto a ressaltar é que o **método estático assessor da instância** ou **sharedInstance()** não é estendido para subclasses, a final, métodos e atributos estáticos pertencem unicamente a classe que foram criados, então os Singletons devem ser projetados suficientemente bem para não necessitarem subclasses.

### Estrutura



Figura - Diagrama de classe conceitual do padrão Singleton

Faça a classe da instância única responsável pelo acesso e **inicialização na primeira utilização**. A instância única é um atributo estático e privado. A função de assessor é um método estático público.

### Exemplo em Android

Para desenvolvimento Android iremos sentir uma grande necessidade de utilizar este padrão de projeto. A própria API do Android utiliza bastante Singletons.

#### SharedPreferences

A classe SharedPreferences é um típico Singleton da API Android. Com ela podemos armazenar em cache, permanentemente, alguns dados simples, como preferências do usuário, sessão e etc. Para esclarecer melhor, vamos fazer um exemplo:

Para execução desta atividade, preparamos um projeto inicial para você. Se trata do nosso app MailList, e já contém as atividades feitas nas aulas passadas. Você pode seguir este passo-a-passo e testar num exemplo próprio. Para facilitar, utilizaremos nosso app MailList.

Descompacte o arquivo **Unidade\_5\_-\_Aula\_1\_-\_Exemplo\_1.zip** e abra o projeto no Android Studio e execute.

Nossa primeira tela voltou a ser nossa LoginActivity.

A esta altura quando abrimos o app algumas vezes, nos sentimos frustrados em sempre ter que inserir o e-mail e senha. Para resolver este problema, vamos salvar estes dados em cache e fazer com que a tela login apareça somente se o usuário nunca digitou o seu e-mail e senha. Siga os seguintes passos:

1. Na nossa classe LoginActivity. Vamos inserir uma constante que será a chave do local onde será salvo o e-mail do usuário. Insira a seguinte constante:

**public static final** String ***USER\_DATA*** = **"user\_data"**;

1. Insira o método que é responsável por salvar um dado no cache:

**public void** saveUserData () {

SharedPreferences sharedPreferences = getSharedPreferences(***USER\_DATA***, ***MODE\_PRIVATE***);

SharedPreferences.Editor editor = sharedPreferences.edit();

editor.putString(**"email"**, **emailEditText**.getText().toString());

editor.commit();

}

**Entendendo o código:** Na primeira linha temos o acesso ao **sharedPreferences()**:

SharedPreferences sharedPreferences = getSharedPreferences(***USER\_DATA***, ***MODE\_PRIVATE***);

Nesta linha utilizamos a constante criada no passo anterior, que acessa o local onde nossas coisas serão salvas. A constante ***MODE\_PRIVATE*** faz com que este cache se torne particular do app. Nenhum outro app pode acessá-lo. Em seguida criamos o Editor:

SharedPreferences.Editor editor = sharedPreferences.edit();

Este editor é uma interface que habilita o cache para alterações. Por fim inserimos o e-mail:

editor.putString(**"email"**, **emailEditText**.getText().toString());

editor.commit();

Os dados inseridos no SharedPreferences são organizados em chave-valor, ou seja, para uma chave temos um valor. Então cria-se a chave e-mail, e guarda o valor contido no **emailEditText**. O método commit() submete as alterações.

1. Insira a chamada deste método dentro da condição de sucesso do método **submit()**:

@OnClick(R.id.login)

public void submit(View view) {

User user = new User(emailEditText.getText(), passwordEditText.getText());

if (!user.isValidEmail()){

emailEditText.setError("Email inválido");

}

if (!user.isValidPassword()){

passwordEditText.setError("Senha inválida. Sua senha deve conter pelo menos 4 caracteres.");

}

if (user.isValid()) {

*// navega para próxima tela*

saveUserData();

startNextActivity();

}

}

1. O passo final é fazer a verificação se existe algo neste cache criado, para evitarmos que a tela de login apareça sempre. Insira o método **onResume()** e faça a verificação dentro dele:

@Override

**protected void** onResume() {

**super**.onResume();

SharedPreferences sharedPreferences = getSharedPreferences(***USER\_DATA***, ***MODE\_PRIVATE***);

**if** (sharedPreferences.contains(**"email"**)) {

startNextActivity();

}

}

Execute o app e veja o resultado. A primeira vez, a tela de login aparecerá, então “faça o login” e rode o app novamente. A tela de login será evitada. Veja como nosso será nosso fluxo:



Figura 4 - Fluxo que o app seguirá após a implementação dos SharedPreferences

## Resumo

Nesta aula aprendemos dois conceitos importantes do Android com a ajuda dos padrões de projeto, o funcionamento do ciclo de vida de uma Activity e da classe SharedPreferences. Sabemaos agora a importância dos padrões de projeto quando nosso objetivo é entender o que se passa em um projeto pronto. Para fixar o conhecimento criamos um fluxo alternativo no nosso app utilizando o padrão Singleton que evita que a tela de login fique aparecendo sempre. Na aula seguinte veremos o outro lado dos padrões de projeto, ele nos auxiliará a solucionar problemas conhecidos na hora que estivermos desenvolvendo um produto novo.

Aula 2

# Padrões de Projeto - Parte 2

Iniciando agora nossa segunda parte de padrões de projetos. Até então utilizamos padrões para demonstrar como certas coisas funcionam no Android, ou seja, fizemos uso deles para **interpretação de um projeto.** Agora mostraremos o outro lado da moeda, onde utilizaremos padrões para **solucionar problemas conhecidos**. Nesta aula mostraremos dois padrões de projeto, o Memento, que basicamente serve para guardar histórico de estados de um objeto e posteriormente resgatá-los, e o padrão State, que serve para definir estados do sistema. Cada um deles possui um exemplo prático e no final da aula você mesmo irá aplicar um padrão de projeto no nosso app MailList.

## Memento

O memento basicamente captura e externaliza estados de um objeto, então o objeto pode retornar para um estado passado. Vejamos mais detalhes.

### Objetivo

* Sem violar a encapsulação, o memento captura e externaliza estados de um objeto, então o objeto pode retornar para um estado passado.
* Permite “desfazer” completamente estados de um objeto.
* Pode criar espécies de “checkpoints” do objeto.

### Problema

Muitas vezes precisamos restaurar ou fazer com que um objeto retorne para um estado anterior (geralmente ações de “desfazer” ou “rollback”).

### Discussão

Memento vem do inglês e pode ser traduzido como lembrança, e é exatamente este o objetivo deste padrão de projeto. Quando fazemos alterações em objetos, pode ser que não queiramos descartar as alterações anteriores, então neste caso precisamos “lembrar” destes estados.

O cliente solicita uma lembrança do objeto de origem quando ele precisa de um ponto de verificação do estado do objeto original. O objeto de origem inicializa o Memento com uma caracterização de seu estado. O cliente é o **cuidador** (Caretaker) do Memento, mas apenas o objeto de origem pode armazenar e recuperar informações do Memento (o Memento é **opaco,** ou seja, não é transparente ao cliente e todos os outros objetos). Se o cliente posteriormente precisar "reverter" o estado do objeto, ele entrega o Memento de volta para o objeto de origem para uma reintegração.

Com este padrão temos a capacidade de realizar ações “desfazer” e “refazer” ilimitadamente e pode ser implementado com um conjunto de objetos.

Este design pattern define três regras:

1. Originator - o objeto que sabe como salvar a si mesmo.
2. Caretaker - o objeto que sabe por que e quando o originador necessita salvar ou recuperar seu estado.
3. Memento - a caixa fechada que é escrita e lida pelo Originator, e mantida pelo Caretaker.

### Estrutura

## https://lh4.googleusercontent.com/GP-TuP9m-dffVk80wnj-g86v-95C0G-9ux3v0e24GywzcXkKqcmk0scptbpdmsp-N_LK-TlAdhUTpqGp0K-ocuTLgpyVKYFdsxo6s2DNVmwranCRANIuOoYvNSBYF7YnsQet7A12

Figura 5 - Diagrama de classes conceitual do padrão Memento

### Exemplo em Android

Neste caso não temos um exemplo direto da API Android, mas podemos criar um exemplo nosso. Vamos inserir a funcionalidade “Desfazer” na nossa tela **ComposeActivity.** Funcionará da seguinte maneira:

Quando o usuário digitar textos no campo “Mensagem” o botão “Desfazer” ficará habilitado. Quando o usuário tocá-lo, a última alteração feita no campo será descartada.

Como nosso exemplo possuirá apenas o “Desfazer”, preparamos um projeto para ser utilizado como ponto de partida deste exemplo. Este projeto tem uma implementação do Memento simplificada, onde o Caretaker descarta a última alteração feita. Vamos lá:

1. Descompacte o arquivo **Unidade\_5\_-\_Aula\_2\_-\_Exemplo\_1.zip** e abra o projeto contido nele no seu Android Studio.
2. Este projeto, é o nosso app MailList. Foram feitas algumas melhorias em relação à aula passada. Veja que temos o botão “Desfazer” na nossa Toolbar da ComposeActivity. Então abra a ComposeActivity e insira o seguinte atributo:

**boolean isUndoEnabled** = **false**;

Este será o controle que faremos para habilitar o botão **desfazer** ou não. No caso, a regra será: habilitado somente quando houver algum texto no campo **Mensagem.**

1. Vamos então criar nosso Originator e nosso Caretaker. Para isto, simplesmente adicione os seguintes atributos:

Caretaker **caretaker** = **new** Caretaker();

Originator **originator** = **new** Originator();

1. Agora precisamos capturar o evento que indica mudanças no nosso EditText de mensagem. Então insira o seguinte método:

@OnTextChanged(R.id.***message***)

**public void** textChanged(CharSequence s, **int** start, **int** before, **int** count) {

**originator**.set(s.toString());

**caretaker**.pushMemento(**originator**.saveToMemento());

**isUndoEnabled** = **true**;

}

**Entendendo o código:** Seguindo o padrão memento, fazemos com que o Originator configure um estado, logo em seguida o Caretaker recebe o memento oferecido pelo Originator e salva em uma pilha da maneira mais apropriada (para o Caretaker). Por fim, habilitamos nosso botão **desfazer**, já que temos alguma coisa dentro do memento.

1. Agora, a última coisa que precisamos é capturar o toque do botão “desfazer”. Para isto adicione o método abaixo, já conhecido por você, no tópico 1.2 da unidade 4:

@Override

**public boolean** onOptionsItemSelected(MenuItem item) {

**switch** (item.getItemId()) {

**case** R.id.***undo***:

**if** (**isUndoEnabled**) {

**messageEditText**.setText(**originator**.restoreFromMemento(**caretaker**.popMemento()));

           }

**return true**;

   }

**return super**.onOptionsItemSelected(item);

}

**Entendendo o código:** Dado que o botão pressionado é o **desfazer** e que estamos autorizados a realizar esta ação, fazemos que o originator nos ofereça o estado registrado no Memento. Memento este especificado pelo caretaker. O método popMemento() do caretaker faz com que nunca mais consigamos resgatar a alteração desfeita.

Isto basta. Execute e teste seu código.

Como implementamos somente uma ação de desfazer alterações, o Memento não está sendo usado em sua força total. A ideia é que ele consiga restaurar qualquer ponto salvo, e no nosso caso, só restauramos alterações passadas ao nosso estado atual, mas o Memento deveria possibilitar que “voltemos para frente” também.

Fique a vontade em criar um botão **refazer** e fixar seus conhecimentos neste padrão de projeto.

## State

Agora veremos mais um padrão. O State permite que um objeto altere seu comportamento quando seu estado interno muda.

### Objetivo

* Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto aparece para mudar sua classe.
* É uma máquina de estado orientada a objeto

### Problema

Geralmente sentimos a necessidade de fazer com que um objeto mude seu comportamento, dependendo do estado em que ele se encontra. Esta necessidade pode ser suprida utilizando uma máquina de estado ou o padrão State.

### Discussão

O padrão de State é uma solução para o problema de como tornar o comportamento dependente de seu estado. Com ele podemos:

* Definir uma classe **contexto** para apresentar uma interface única para o mundo exterior.
* Definir uma classe base abstrata State.
* Representar os diferentes **estados** da máquina de estados como classes derivadas da classe base Estado. Melhor dizendo, podemos estender nossa máquina de estado deixando-a escalável para suporte de mais estados.
* Definir o comportamento específico do State nas classes States derivadas apropriadas.
* Manter um ponteiro para o **estado** atual na classe contexto.
* Alterar o estado da máquina de estado, alterando o ponteiro estado atual.

O padrão de State não especifica onde as transições de estado serão definidas. Mas possuímos duas opções: especificar o objeto contexto, ou cada classe individual derivada de State. A vantagem desta última opção é a facilidade de adição de novos estados derivados de State. A desvantagem é que cada classe State derivada tem conhecimento (acoplamento) dos seus irmãos, que introduz dependências entre subclasses.

A abordagem baseada em tabelas de projetar máquinas de estados finitos faz um bom trabalho de especificar transições de estado, mas dificulta a adição de ações que acompanham as transições de estado. A abordagem baseada em padrões usa código (em vez de estruturas de dados) para especificar as transições de estado, mas ele faz um bom trabalho de acomodar as ações de transição de estado.

### Estrutura

A interface da máquina de estado é encapsulada na classe **wrapper**. A interface da hierarquia wrappee espelha a interface do wrapper com a exceção de um parâmetro adicional. O parâmetro extra permite que as classes wrappee chamem de volta a classe de wrapper conforme necessário. Veja o diagrama.

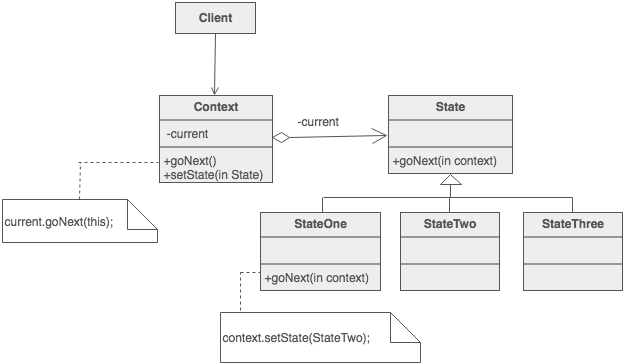


Figura 6 - Diagrama de classes conceitual do padrão State

### Exemplo

O padrão de State permite que um objeto mude seu comportamento quando seu estado interno muda. Este padrão pode ser observado em uma máquina de venda automática. Máquinas de venda automáticas têm estados com base no inventário, a quantidade de moedas depositadas, a capacidade de fazer a mudança, o item selecionado, etc. Quando a moeda é depositado e uma seleção é feita, uma máquina de venda automática irá ou entregar um produto e nenhuma mudança, entregar um produto e mudança, entregar nenhum produto devido a moedas insuficientes em depósito, ou entregar nenhum produto devido ao esgotamento de estoques.

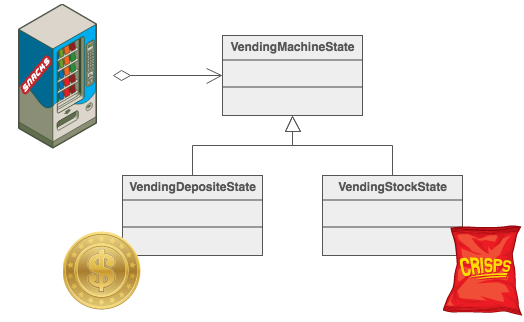


Figura 7 - Exemplo do padrão State com uma máquina de vendas automática

### Faça você mesmo

A explicação é bem robusta, não é mesmo? Como dissemos anteriormente, pode parecer difícil, interpretar, implementar e dominar um padrão de projeto no início, mas sabemos que eles são extremamente importantes em mobile. Para treinar, faça sua própria implementação do padrão State. Utilize o conhecimento aprendido aqui e complemente com pesquisas na internet. Conteúdo e exemplos sobre isso é o que não falta.

Para facilitar, lembra-se da lógica que fizemos na tela de Login, que se o usuário **não está logado** a tela de Login deve ser apresentada, mas se o usuário **já está logado** evitamos esta tela e pulamos para dentro da app. Pois, tente implementar um controle de estado para esta lógica.

### Resumo

Nesta aula aprendemos a importância de conhecermos padrões de projetos para solucionar problemas em específico. Com o padrão Memento pudemos criar uma a funcionalidade **desfazer** na nossa tela Escrever E-mails. Com o padrão State poderemos definir melhor o fluxo de login do nosso app. Não se esqueça de finalizar a tarefa do tópico 2.2.6, ela é muito importante para que você domine a prática dos padrões de projeto. Na próxima aula aprenderemos a criar listas no nosso App.