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, sendo assim, foram preparadas duas aulas com alguns padrões de projetos importantes para você. Em aulas anteriores, foi visto o padrão *Composite* que ajuda a entender como funciona a estrutura e as relações de *Views* e *ViewGroups* em um *layout*. Essa aula será voltada totalmente aos padrões de projeto. Começando pelo padrão *Template Method*, você entenderá como funciona o ciclo de vida de uma *Activity* e um *Fragment*. Depois, com o *Singleton,* você verá como acessar instâncias compartilhadas, que o próprio SDK do Android disponibiliza. Entendendo esses padrões fica mais fácil de entender o que acontece internamente 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 sequência de passos a ser executada sendo que alguns desses passos são adiados para suas subclasses. O *Template Method* permite que subclasses redefinam determinadas etapas de um algoritmo sem alterar sua estrutura;
* Serve como classe base que declara um algoritmo de "espaços reservados" e as classes derivadas devem implementar esses espaços. Eles podem ser entendidos como um método abstrato, por exemplo, no qual as subclasses vão implementá-lo, ou seja, preencher esse espaço.

### Problema

Dois componentes diferentes podem ter semelhanças significativas, mas não demonstram a reutilização de interface ou de aplicação de algum código comum. Se essa 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 do componente (entende-se como cliente uma subclasse ou a classe que o consome) em uma classe derivada concreta (entende-se como classe concreta a classe que não é abstrata).

**DICA:** Você acabou de sofrer um bombardeio de termos técnicos. Não se preocupe, eles serão explicados. Essa é uma definição formal e bastante técnica do *Template Method* que é encontrada em livros desse assunto, mas você vai entender melhor esses termos.

Os **projetistas de componentes** são os programadores, ou seja, você. Quando programa em Linguagem Orientada a Objetos, você deve se preocupar a todo o tempo em desenvolver algo desacoplado, pronto para reuso, e essa 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 do *software*. 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** tratam-se desse algoritmo variável, ou seja, são métodos abstratos que devem ser implementados pelas subclasses ou **classe derivada concreta**.

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

O *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 pelo médico. Mas o médico também executa partes desse procedimento. Então fazendo uma analogia, o *framework* determina um procedimento e delega algumas partes às classes clientes, mas alguns passos são executados pelo próprio *framework*. A relação com esta analogia pode ser chamada de **princípio de Hollywood**, que é: “Não nos chame. Nós o chamaremos”.

Essas definições são sempre um tanto complicadas. Isso porque, para obter total compreensão desses padrões, é necessário executá-los na prática. O diagrama a seguir e a aplicação em um exemplo do SDK Android podem esclarecer um pouco esse padrão, mas a dica é: esteja sempre programando e buscando soluções baseadas nos padrões de projeto existentes. Isso pode aumentar a sua produtividade, pois os padrões de projeto são como soluções prontas para serem aplicadas.

### Estrutura

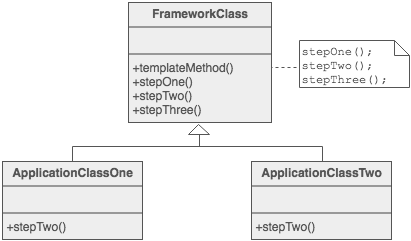


Figura 1 - Diagrama conceitual do padrão Template Method

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 reutilisável é definido nas classes bases do *framework* e então os clientes desse *framework* são livres para customizar funcionalidades criando classes derivadas.

**DICA**: É possível chamar esses ganchos também de **método de retorno de chamada**, que é o nome dado na documentação do Android. Dá-se esse nome por causa de uma tradução livre do termo ***call-back***, em que um objeto delega certas tarefas a 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 do Wi-Fi do aparelho.

### Exemplo em Android

Grades *frameworks* que utilizam o *Template Method* na API Android são a *Activity* e o *Fragment*. Agora será analisada a *Activity*.

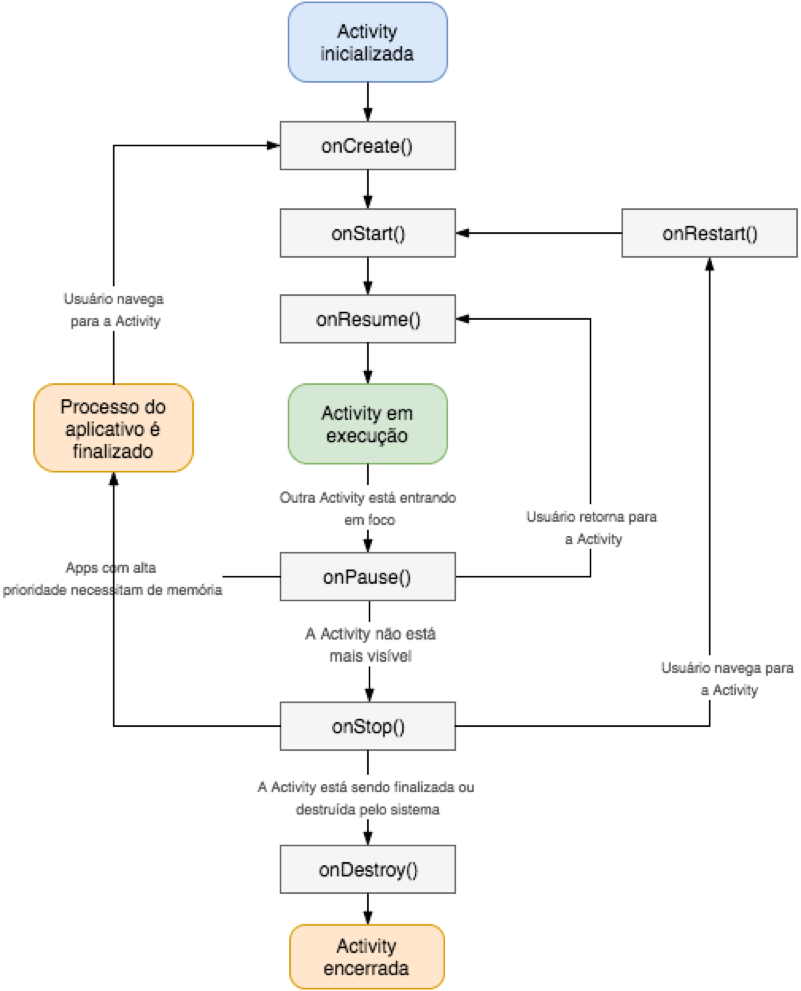


Figura 2 - Ciclo de vida de uma Activity

A imagem acima, já mostrada no item 1.1.1.1 da Unidade 4, representa o ciclo de vida de uma *Activity*, então há 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, isso 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 ela não ser abstrata, os projetistas dos componentes quase nunca utilizam-na puramente, sem fazer uma extensão. Então na prática são utilizadas apenas suas subclasses. Mas a vantagem dela não ser uma classe abstrata é a de 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*) permite inserir comportamentos adicionais. Há aí outra característica *do Template Method*.

Você pode interceptar qualquer método do ciclo de vida de uma *Activity* e adicionar comportamentos a ele, basta você implementá-lo na sua subclasse. Mas não se esqueça de que 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 se substitui um método da classe mãe por outro de mesmo nome na classe filha, há ainda a chance de executar o código contido no método da classe mãe. Isso 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 da classe filha. Veja nos exemplos já construídos:

@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 o código que foi programado até agora e responda: alguma vez será preciso invocar um método do ciclo de vida da *Activity* manualmente?

A resposta é não. Nunca será invocado o **onPause(),** por exemplo, ele apenas será implementado 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 de que a classe possui apenas uma instância e provê acesso global a ela;
* "*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 à inicialização apenas no primeiro uso) e acesso global são necessários.

### Discussão

O objetivo é fazer uma classe de única instância responsável pela própria criação, inicialização, acesso e execução. Para isso declarou-se um atributo privado e estático que armazenará uma instância da própria classe. Uma vez que esse membro é estático e privado, deve-se fornecer um método estático e público que encapsula todo o código de inicialização e fornece acesso para a instância armazenada nesse atributo. Ele será chamado de método **instance()** ou **sharedInstace().**

O cliente chamará apenas esse método assessor e os demais métodos da classe são chamados apenas por intermédio desse método.

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

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

### Estrutura



Figura 3 - Diagrama de classe conceitual do padrão Singleton

Faça a classe da instância única responsável pelo acesso e pela **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, você sentirá uma grande necessidade de utilizar esse padrão de projeto. A própria API do Android utiliza bastante *Singletons*.

#### *SharedPreferences*

A classe *SharedPreferences* é um típico *Singleton* da API Android. Com ela pode-se armazenar em *cache*, permanentemente, alguns dados simples, como preferências do usuário, sessão etc. Para esclarecer melhor, você fará um exemplo.

Para execução desta atividade, foi preparado um projeto inicial para você. Trata-se do seu *app* MailList, que já contém as atividades feitas nas aulas passadas. Você pode seguir este passo a passo e testar num exemplo próprio.

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

Sua primeira tela voltou a ser sua LoginActivity.

A esta altura, quando você abrir o *app* algumas vezes, talvez você se sinta frustrado por sempre ter de inserir o *e-mail* e a senha. Para resolver esse problema, você vai salvar esses dados em *cache* e fazer com que a tela *login* apareça somente se o usuário nunca digitou o *e-mail* e a senha. Siga os seguintes passos:

1. Na sua classe LoginActivity, insira uma constante que será a chave do local em que 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 você tem o acesso ao **sharedPreferences()**:

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

Nessa linha foi usada a constante criada no passo anterior, que acessa o local em que suas coisas serão salvas. A constante ***MODE\_PRIVATE*** faz com que esse *cache* torne-se particular do *app*. Nenhum outro *app* pode acessá-lo. Em seguida, crie o Editor:

SharedPreferences.Editor editor = sharedPreferences.edit();

Esse editor é uma interface que habilita o *cache* para alterações. Por fim, insira 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 tem 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 desse 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 a próxima tela*

saveUserData();

startNextActivity();

}

}

1. O passo final é fazer a verificação. Se existe algo nesse *cache* criado, é para evitar 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 será o seu fluxo:



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

## Resumo

Nesta aula você aprendeu dois conceitos importantes do Android com a ajuda dos padrões de projeto: o funcionamento do ciclo de vida de uma *Activity* e o da classe SharedPreferences. Você sabe agora a importância dos padrões de projeto quando seu objetivo é entender o que se passa em um projeto pronto. Para fixar o conhecimento, foi criado um fluxo alternativo no seu *app* utilizando o padrão *Singleton,* que evita que a tela de *login* fique sempre aparecendo. Na aula seguinte, você verá o outro lado dos padrões de projeto: eles ajudarão a solucionar problemas conhecidos na hora que você estiver desenvolvendo um produto novo.

Aula 2

# Padrões de Projeto - Parte 2

Você iniciará agora a segunda parte de padrões de projetos. Até então foram utilizados padrões para demonstrar como certas coisas funcionam no Android, ou seja, foi feito uso deles para a **interpretação de um projeto.** Agora será mostrado o outro lado da moeda: serão utilizados padrões para **solucionar problemas conhecidos**. Nesta aula serão mostrados 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 aplicará um padrão de projeto no seu *app* MailList.

## *Memento*

O *memento* basicamente captura e exterioriza estados de um objeto, o qual pode retornar para um estado passado. Veja mais detalhes.

### Objetivo

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

### Problema

Muitas vezes é necessário 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 esse o objetivo desse padrão de projeto. Quando são feitas alterações em objetos, pode ser que não se queira descartar as alterações anteriores, então nesse caso é preciso “lembrar-se” desses 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 a 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 esse padrão, tem-se a capacidade de realizar ações “desfazer” e “refazer” ilimitadamente e ele pode ser implementado com um conjunto de objetos.

Este *design pattern* define três regras:

1. *Originator* – é o objeto que sabe como salvar a si mesmo;
2. *Caretaker* – é o objeto que sabe por que e quando o originador necessita salvar ou recuperar seu estado;
3. *Memento* – é 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

Nesse caso não há um exemplo direto da API Android, mas você pode criar um exemplo seu. Insira a funcionalidade “Desfazer” na sua 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 seu exemplo possuirá apenas o “Desfazer”, foi preparado um projeto para ser utilizado como ponto de partida. Esse projeto tem uma implementação do *Memento* simplificada, em que o *Caretaker* descarta a última alteração feita.

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

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

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

1. Você vai então criar seu *Originator* e seu *Caretaker*. Para isso, simplesmente adicione os seguintes atributos:

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

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

1. Agora é preciso capturar o evento que indica mudanças no seu 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*, é feito 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, é habilitado o botão **desfazer**, já que há alguma coisa dentro do *memento*.

1. Agora, a última coisa que você precisa é capturar o toque do botão “desfazer”. Para isso adicione o método abaixo, já conhecido 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 você está autorizado a realizar essa ação, faça com que o *originator* ofereça o estado registrado no *Memento*, o qual está especificado pelo *caretaker*. O método popMemento() do *caretaker* faz com que nunca mais se consiga resgatar a alteração desfeita.

Isso basta. Execute e teste seu código.

Como foi implementada 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, mas no seu caso só foram restauradas alterações passadas ao seu estado atual, embora o *Memento* devesse possibilitar que “se voltasse para a frente” também.

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

## *State*

Agora você verá 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 sente-se a necessidade de fazer com que um objeto mude seu comportamento, dependendo do estado em que ele se encontra. Essa necessidade pode ser suprida utilizando uma máquina de estado ou o padrão *State*.

### Discussão

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

* 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, pode-se estender sua 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 você possui duas opções: especificar o objeto contexto ou especificar 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, gerando dependências entre subclasses de State e seus irmãos.

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 essas transições. 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 essas ações de transição.

### 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.

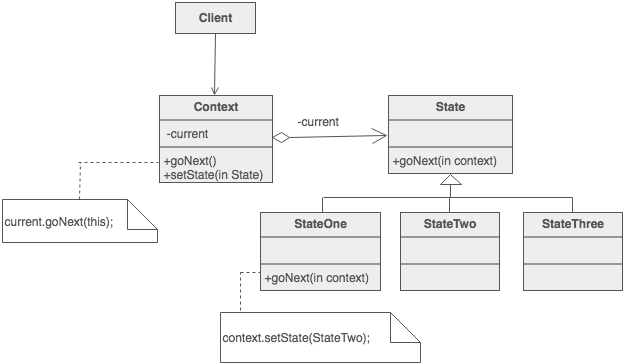


Figura 6 - Diagrama de classes conceitual do padrão State

### Exemplo

O padrão de *State* permite que um objeto mude seu comportamento quando seu estado interno muda. Esse 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 depositada e uma seleção é feita, uma máquina de venda automática irá ou entregar um produto e nenhuma mudança, entregar um produto e uma mudança, entregar nenhum produto devido a moedas insuficientes em depósito ou entregar nenhum produto devido ao esgotamento de estoques.

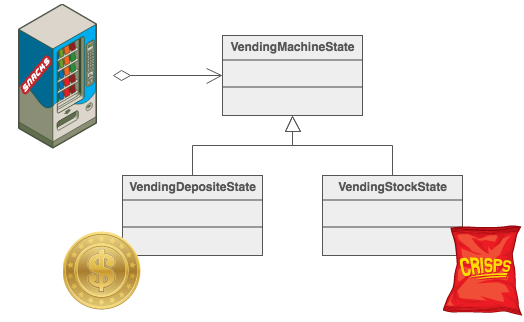


Figura 7 - Exemplo do padrão State com uma máquina de vendas automática

### Faça você mesmo

A explicação é bem robusta, não é mesmo? Como dito anteriormente, pode parecer difícil interpretar, implementar e dominar um padrão de projeto no início, mas é sabido 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 faltam.

Para facilitar, lembra-se da lógica feita 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,** evita-se essa tela e pula-se para dentro do *app*? Tente implementar um controle de estado para essa lógica.

### Resumo

Nesta aula você aprendeu a importância de conhecer padrões de projetos para solucionar problemas específicos. Com o padrão *Memento* você pôde criar uma funcionalidade **desfazer** na sua tela “Escrever *E-mails”*. Com o padrão *State* você poderá definir melhor o fluxo de *login* do seu *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, você aprenderá a criar listas no seu *app*.