**Exercício de Padrões de Projeto**

Ciência da Computação - UFPA

Disciplina: Análise e Projeto de Software

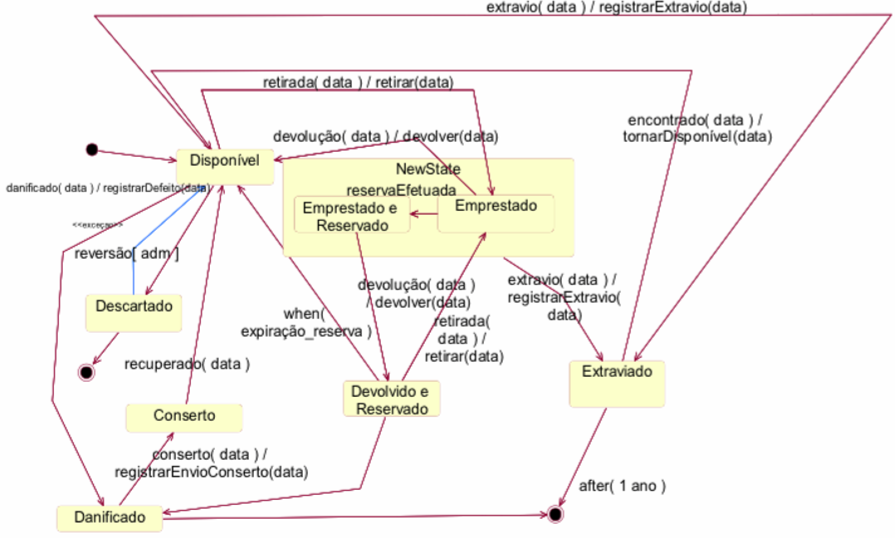
Professor: Rodrigo Quites Reis

Alunos: Marcos Senna Benaion Leal. Matrícula: 201204940019.

Walisson Cardoso Gomes Matrícula: 201204940001.

Para todos os exercícios aqui listados foram realizadas suas implementações na linguagem Java e o respectivo código para teste submetido para o e-mail ‘rqreis@gmail.com’.

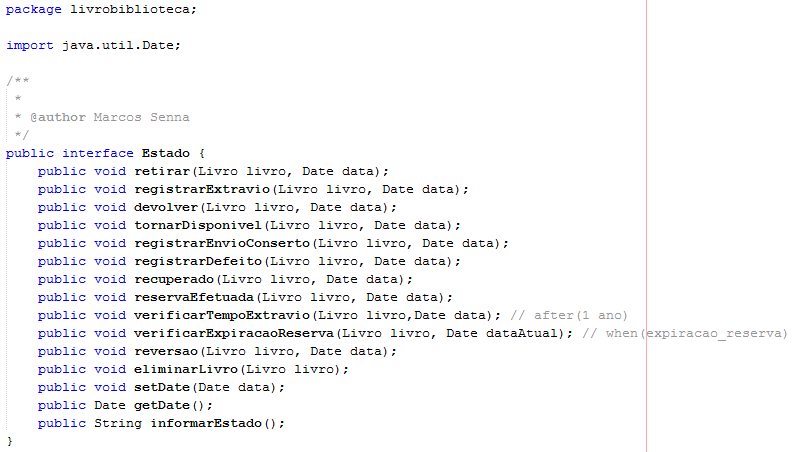
**1ª questão)** Implementar o statechart abaixo com o uso do pattern State:



Resolução:

Para implementar este Statechart foi utilizado o padrão de projeto State, conforme recomendado pelo enunciado da questão.

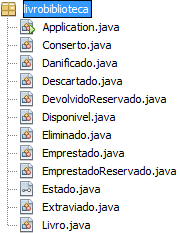
Inicialmente foi criada a interface Estado (State), a qual terá suas versões concretas de todos os estados visíveis no diagrama.



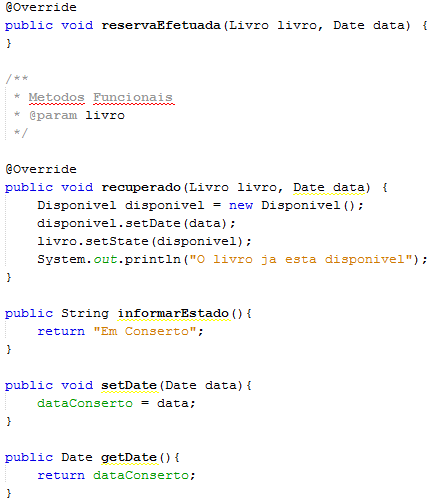
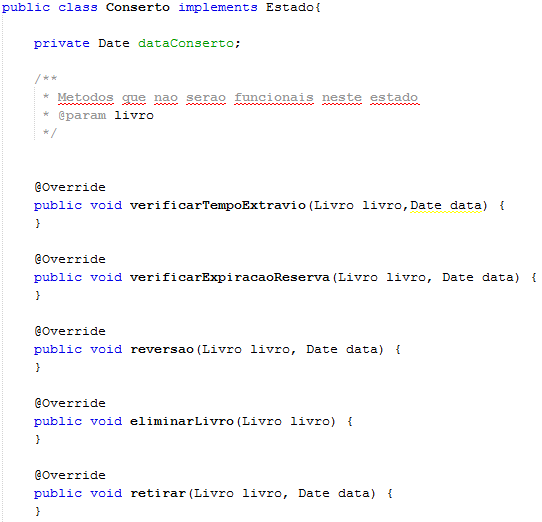
**Figura 1**. Trecho de código da interface Estado

Esta interface contém todas as possíveis ações (transições) de estado que o conjunto de estados pode oferecer. Entretanto, dependendo do estado atual, nem todas as funções serão oferecidas para o mesmo, para assegurar a consistência com o Statechart.

A seguir, são criados todos os estados disponíveis no Statechart fornecido.

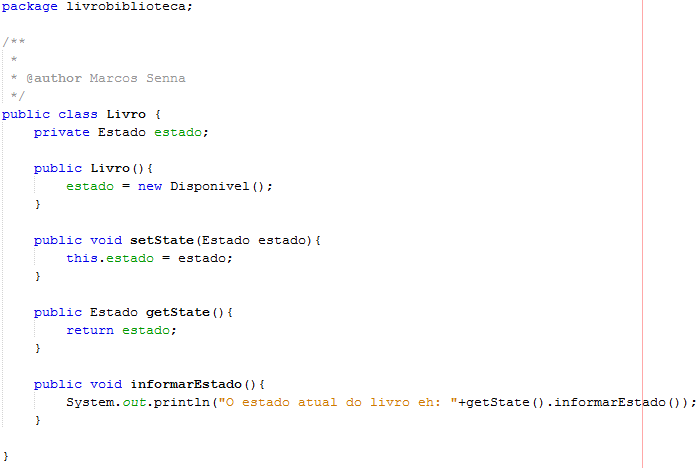


**Figura 2**. Lista de Classes da implementação do Statechart. Observe que há a implementação concreta de todos os estados possíveis disponíveis no Statechart (ex.: Conserto, Disponível, Extraviado).



**Figura 3**. Trecho do código da classe e estado Conserto. Observe que um livro em conserto pode apenas ser recuperado (método recuperar() ). Outros métodos seguem não implementados.

Após a criação de todos os estados e definição de todas as funcionalidades disponíveis e quais não devem ser fornecidas, foi realizada a criação do Contexto, que neste caso é o Livro, conforme mostrado abaixo.



**Figura 4**. Trecho de código da Classe Livro, a qual será o contexto utilizado para manipular seus possíveis estados. Observe que um livro criado já começa em no estado Disponível.

Após isso, iremos testar a execução do código e observar as mudanças de estado de um objeto Livro. Para isso, inicialmente, definimos um objeto do tipo *Date* (Data), pois uma parte das transições de estado necessitam de um parâmetro do tipo Data. Criamos este parâmetro e armazenamos numa variável chamada DataAtual, correspondente a data de criação. Há também um outro objeto do tipo Data que armazena a data atual após ter decorrido um ano, sendo mostrado sua finalidade mais adiante.

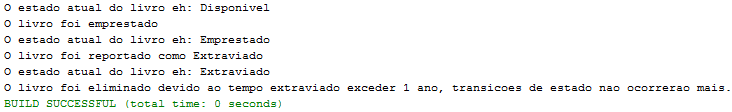
Com isso, podemos instanciar um objeto do tipo Livro e começar a manipulá-lo, sendo mostrado na figura 5. Ao criar um livro, seu estado é automaticamente definido para Disponível, então iremos utilizar as operações que estão disponíveis para o estado citado, as quais são: Retirar (corresponde a realizar empréstimo), Registrar extravio, Descartar e Registrar Defeito.

Para isso, selecionamos o objeto Livro e solicitamos seu estado atual (através do getState() ), e executamos a operação de retirada do livro, informando ele mesmo e a data de retirada. Com isso seu estado mudará, conforme a figura 6 mostra. A figura 7 mostra mais detalhes de como ocorre a transição do estado Disponível para Emprestado. Uma nova instância do estado Emprestado é criada, passando a data de empréstimo e após isso o Livro muda seu estado de Disponível para Emprestado (observado em ‘livro.setState(emprestado)’).

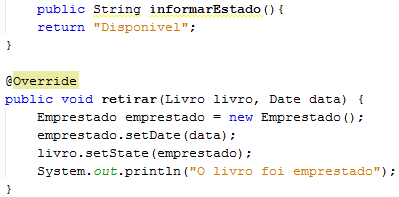
Adicionalmente outras operações foram realizadas, como por exemplo Registrar Extravio, mudando o estado do livro de Emprestado para Extraviado. Ainda neste estado foram realizados dois testes para comprovar a funcionalidade de eliminação do livro dos registros após 1 ano de extravio. O método verificarTempoExtravio() é responsável por essa verificação. No primeiro caso, ele é executado comparando a data atual com a data de extravio (que também foi a data atual). Como neste caso a condição se verifica falsa, não há mudança de estado do Livro. Entretanto ao utilizar como data atual da verificação de tempo uma data correspondente ao dia atual mais 1 ano decorrido, a condição se torna verdadeira e o Livro muda de estado, seguida de uma mensagem na tela. Mais detalhes da



**Figura 5**. Comandos utilizados para manipular uma instância do Livro

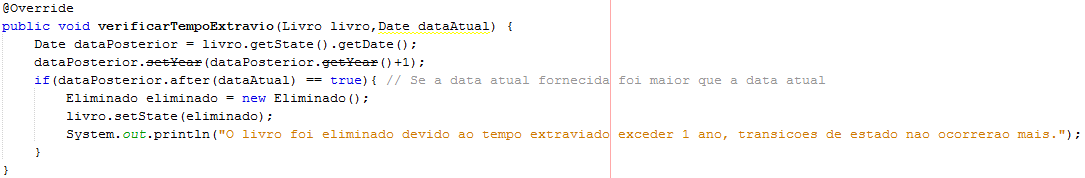


**Figura 6**. Saída do programa após a manipulação do Livro conforme a figura 5.



**Figura 7**. Implementação do método ‘Retirar’ dentro da classe ‘Disponível’

Implementação do método verificarTempoExtravio() são mostrados na figura 8.

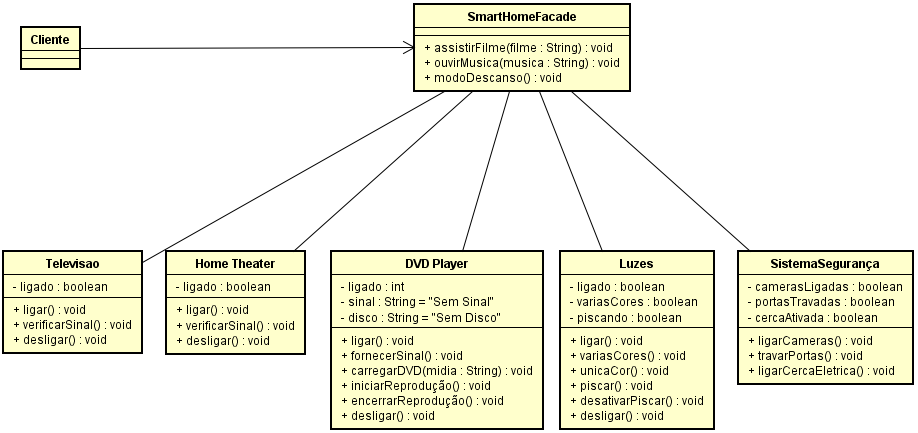


**Figura 8**. Implementação do método verificarTempoExtravio().

**2ª questão)** Construa uma façade com, no mínimo, 3 serviços, que encapsulem funcionalidades que estão dispersas em, no mínimo, 4 classes. Esta façade deve ser singleton.

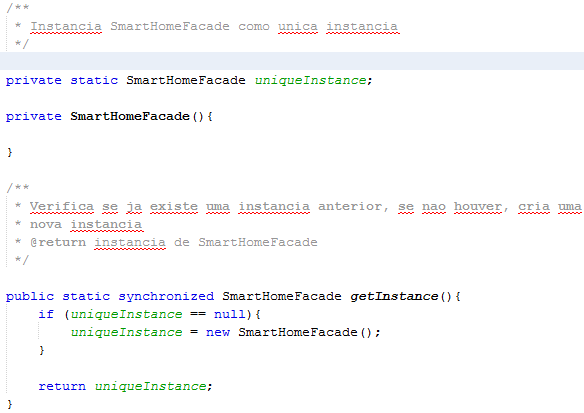
Resolução:

Após a leitura do enunciado foi definida a estruturação/tema para a implementação do Façade Singleton. Com isso a abordagem escolhida foi a simulação de uma *Smart Home* (Casa Inteligente), onde um Cliente/Usuário se comunica com uma interface Façade Smart Home para realizar múltiplas tarefas na casa. A Façade, por sua vez, irá chamar funcionalidades dispersas nos dispositivos para realizar a função escolhida pelo usuário. Segue abaixo na figura 9 uma exibição das classes utilizadas neste problema.



**Figura 9**. Diagrama de classes de Smart Home.

Na Façade SmartHomeFacade, como esperado, fornece 3 funcionalidades para o cliente: Assistir filme, Ouvir música ou Modo descanso, este último o qual desliga os aparelhos da casa e ativa os sistemas de segurança. A Façade, além disso é Singleton, isto é, permitindo apenas uma única instância da mesma.



**Figura 10**. Exibição da obtenção da instância única da Facade.



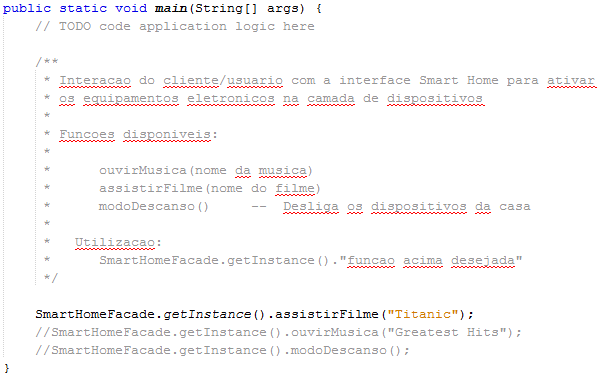
**Figura 11**. Detalhe da implementação da funcionalidade assistir filme na classe SmartHomeFacade.

Devido à natureza da implementação da Smart Home, cada dispositivo também possui apenas uma instância única.

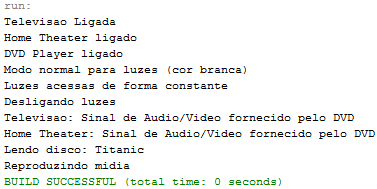


**Figura 12**. Funcionalidades que a Televisão pode realizar.

Com isso, pode ser realizada a utilização dos serviços da Facade. Neste exemplo será utilizado a funcionalidade assistir filme com o parâmetro “*Titanic*”. Podemos observar na saída de dados que a SmartHomeFacade acessou os vários dispositivos conectados e os manipulou de tal forma que possibilitou a reprodução do filme.



**Figura 13**. Execução da funcionalidade assistir filme.

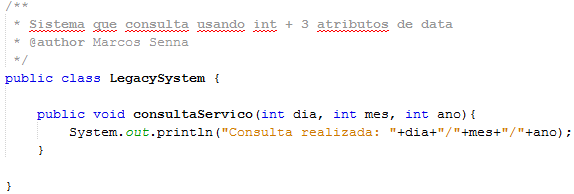


**Figura 14**. Saída exibida na tela após a execução do método assistir filme mostrado na figura 13.

**3ª questão)** Considere um sistema legado que fornece o serviço consultaServiço que recebe três parâmetros: dia, mês e ano (todos int), e que apenas os imprime na tela. Construa, usando o padrão adapter, um software que envia mensagens para este sistema legado fornecendo um parâmetro do tipo java.util.Date (ou equivalente em outra linguagem de programação).

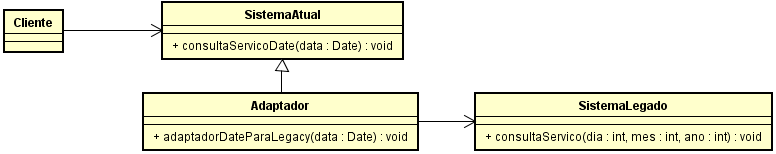
Resolução:

Assim como proposto no enunciado, foi realizada a criação do sistema legado, o qual possui uma funcionalidade ‘consultarServiço’ a qual recebe 3 parâmetros: dia, mês e ano.



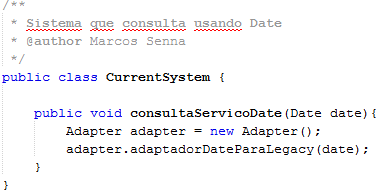
**Figura 15.** Sistema Legado o qual irá ser adaptado.

Posteriormente foi definido como a relação entre o adaptador irá se modelar entre o sistema legado e o sistema atual.

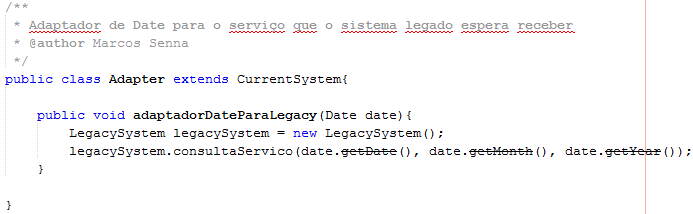


**Figura 16**. Diagrama de classes do adaptador entre sistema legado e sistema atual.

Após isso foi criado o sistema atual, o qual chama o adaptador e o Adaptador de Date para parâmetros separados.

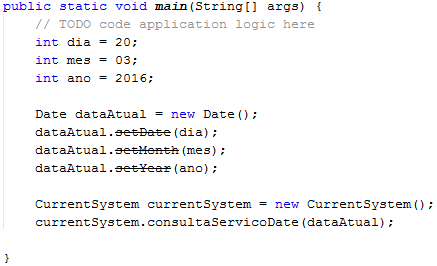


**Figura 17**. Sistema atual de consulta usando *Date*



**Figura 18**. Adaptador entre Sistema Atual e Sistema Legado

Com isso podemos testar a aplicação atual para verificar se a comunicação entre os respectivos sistemas, através do adaptador, é efetiva. A partir da figura 20 podemos ver que isso se verifica verdadeiro.



**Figura 19**. Execução do Sistema atual, passando o objeto ‘dataAtual’.



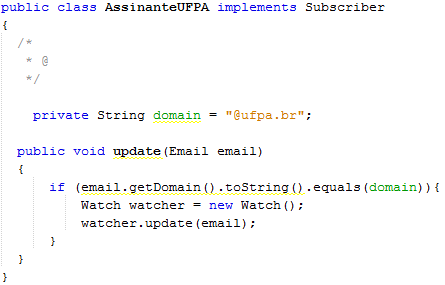
**Figura 20**. Resultado da execução do método ‘consultaServicoDate’

**4ª questão)** Na página http://www.ybadoo.com.br/tutoriais/poo/10/28/ há um programa que adota a abordagem Observer para notificar o recebimento de um e-mail por diversos meios, virtuais, representando dispositivos reais. Complementar o programa para que a notificação para Watch ocorra apenas para mensagens cujo remetente seja do domínio "@ufpa.br".

obs: em Java, pode-se usar a interface Observable já fornecida pela linguagem.

Resolução:

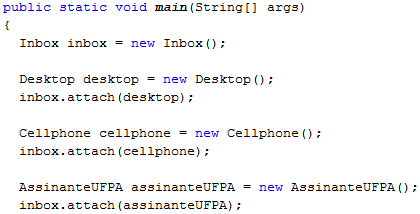
De forma simples, o aspecto mais importante nesta questão é a criação de um novo observador, o qual daremos o nome de AssinanteUFPA, sendo responsável por gerenciar os e-mails do domínio ‘@ufpa.br’. As outras classes não foram modificadas.



**Figura 21**. Observador para os e-mails do domínio ‘@ufpa.br’.

Conforme o enunciado direciona, apenas os e-mails do domínio ‘@ufpa.br’ devem entrar na notificação para Watch, consequentemente os outros tipos de email irão para notificação Desktop ou Cellphone. Executando a aplicação, não irá mais anexar o observador Watch na caixa de entrada. No seu lugar estará o observador AssinanteUFPA. Os outros não serão modificados.

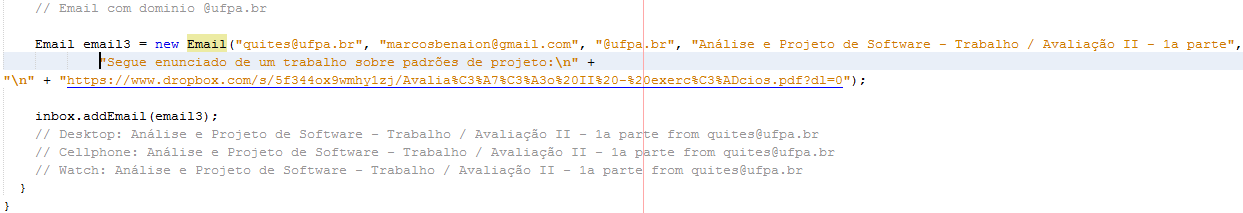
Após isso vários e-mails de teste serão criados para validar a execução do programa. Através da saída mostrada na figura podemos ver que apenas os remetentes com e-mails de domínio ‘@ufpa.br’ alertaram o destinatário através da notificação Watch.



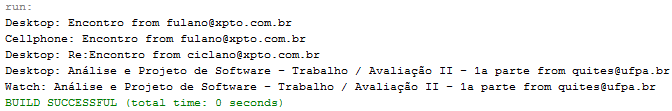
**Figura 22**. Anexando o observador AssinanteUFPA à caixa de entrada.



**Figura 23**. Email de teste sem o domínio da UFPA.



**Figura 24**. Email de teste da UFPA.



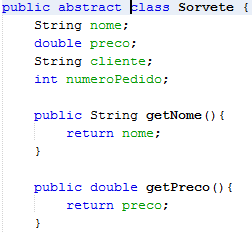
**Figura 25**. Saída exibida na tela após as notificações dos email serem realizadas. Observe que apenas o email com domínio da UFPA foi notificado por Watch.

**5ª questão)** Obs: Adaptado de <https://sites.google.com/site/diogodmoreira/padroes20142>

Uma sorveteria precisa criar um sistema para cadastrar os pedidos dos seus clientes. Os sorvetes podem ser servidos em Copo (R$0.20), Taça (sem custo) ou Casquinha (R$1.50). Atualmente, eles contam com 5 sabores de sorvete: ’Chocolate’, ’Morango’, ’Flocos’, ’Pavê’ e ’Napolitano’ (R$1.50 cada bola) ou ‘Chocolate Diet’ (R$ 2.00 a bola). Além disso, os clientes podem acrescentar: Cobertura de Chocolate, Morango ou Caramelo (que custa R$ 0.50). Utilizando o Padrão Decorator para resolver este problema.

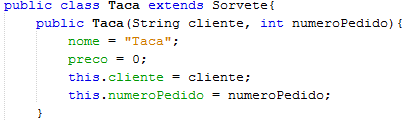
Resolução:

Começando a implementação utilizando Decorator, foi criada a classe abstrata Sorvete, a qual irá ter informações de nome e preço do sorvete selecionado conforme o tipo em que serão servidos.



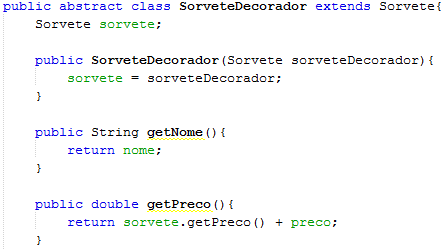
**Figura 26**. Trecho de código da classe abstrata Sorvete

Assim, para cada tipo em que será servido o sorvete há uma implementação correspondente.

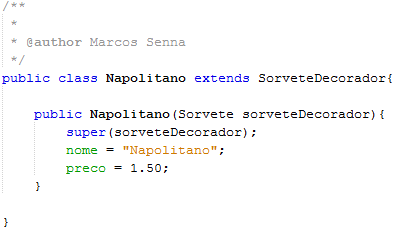


**Figura 27**. Classe Taca que é uma extensão de Sorvete

Feito isso, há a escolha dos sabores para cada sorvete, sendo necessário implementar um decorador para esses sabores, e para cada sabor, novamente, uma implementação de seu tipo.

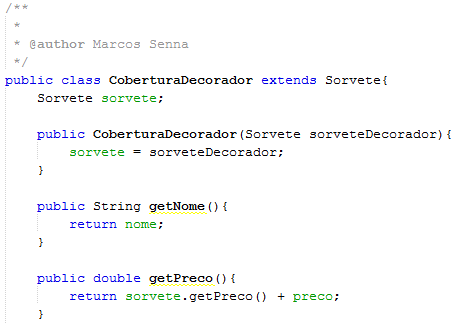


**Figura 28**. Trecho de código do decorador de sorvete para sabores



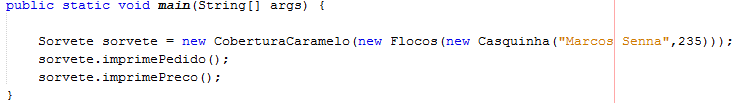
**Figura 29**. Implementação concreta de decorador de sabores

Por fim, há a escolha de adicionais para o sorvete através de coberturas e da mesma forma que realizado anteriormente, há um decorador para as coberturas e suas implementações.



**Figura 30**. Decorador de coberturas

Para realizar os testes devemos instanciar um objeto do tipo Sorvete e apenas agrega-lo com suas responsabilidades/decorações através de novas instâncias de outros tipos até chegar a super classe.



**Figura 31**. Utilização da agregação de decorações em um objeto Sorvete

Através da saída abaixo mostrada, pode-se notar que o objeto decorado sorvete estará cumprindo suas responsabilidades em cascata, exibindo as mensagens da classe anterior e incluindo a sua própria, além de realizar a adição dos valores de cada objeto escolhido.



**Figura 32**. Saída exibida através da execução dos comandos na figura 31