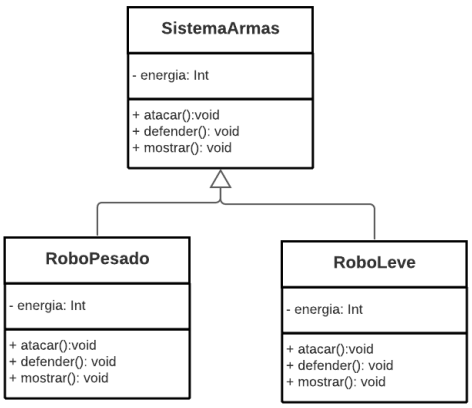
Objetivo

Identifique e modele as classes e interfaces envolvidas na instrução, especificando seus atributos, responsabilidades e relacionamentos entre elas.

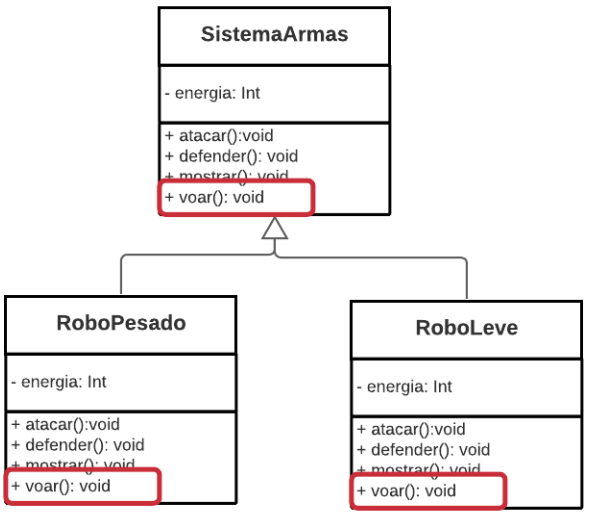
Apresentação do caso “Batalha do futuro”

Suponha que você precise modelar um jogo de estratégia/guerra em tempo real: **“Batalha do futuro”**. Entre as classes do jogo, teremos os diferentes tipos e suas características. A primeira etapa é definir um robô com suas operações básicas:



A primeira coisa que podemos fazer é criar uma classe SistemaArmas, com as operações comuns: atacar, defender e se mostrar (na tela, com seus dados). Digamos que uma próxima atualização do jogo apresentará sistemas de armas voadoras. Seria importante manter as "atualizações" do aplicativo no mínimo e interromper o sistema o menos possível. Então, o que poderia ser feito com o design atual?

Normalmente, a operação voar() seria incluída na classe SistemaArmas. Mas se entre os sistemas de armas que desejo incluir houver um tanque? Os tanques não voam! Vejamos um exemplo:

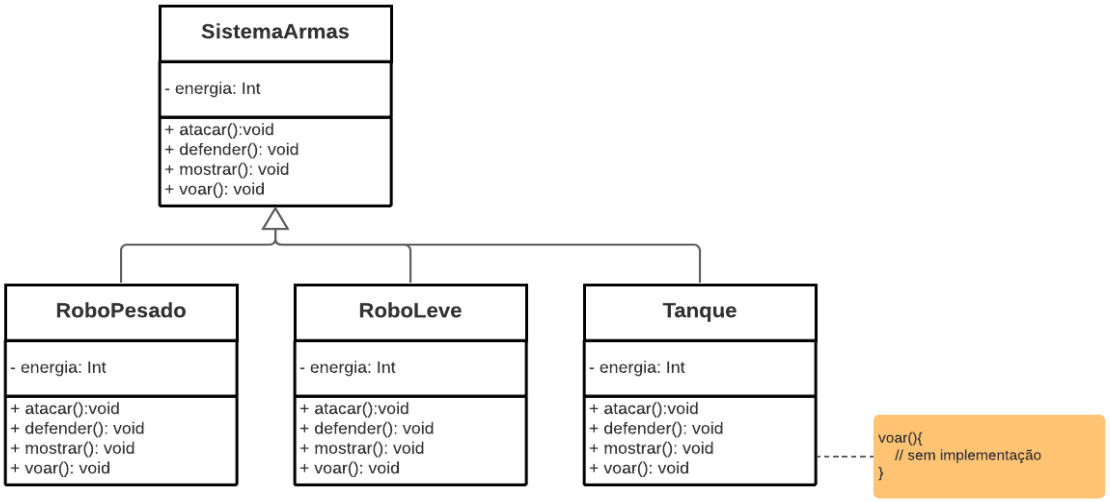


À primeira vista, um sistema de armas pode ser um robô, um tanque, um bombardeiro, então, adicionar o método voar() quebra o design, porque se houvesse uma classe filha Tanque, teríamos um tanque voador. Portanto, nem todos os veículos devem voar! Não é um bom design. Mesmo o design original não era bom, já que uma mudança local para uma classe gerava um efeito colateral generalizado.

Portanto, quando se trata de manutenção, **a herança nem sempre é a melhor opção.**

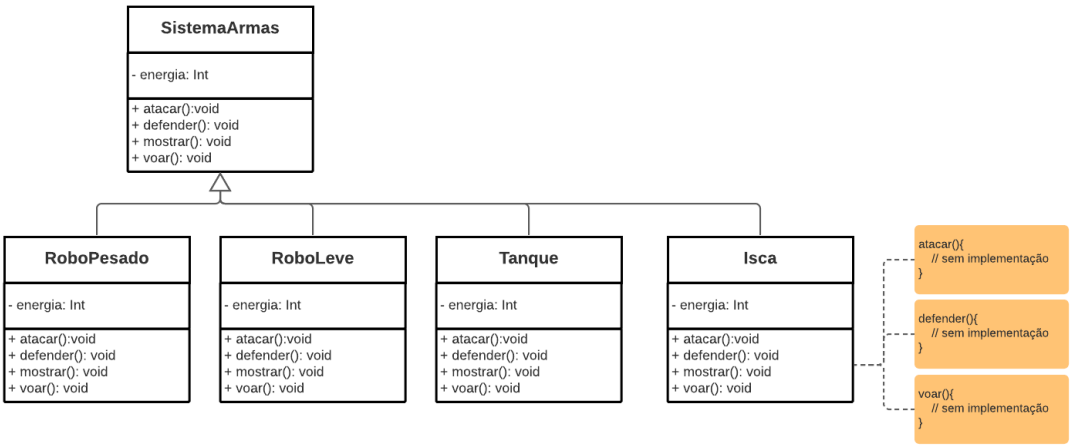
Proposta de soluções

Que tal sobrescrevermos a operação voar() e deixá-la "vazia" (sem implementação) em todas as classes que "não deveriam voar" (tanque, submarino, etc.).



Se sobrescrevermos o voo para deixar uma implementação vazia, o que acontecerá se eu adicionar um novo tipo de veículo, por exemplo, um porta-aviões? Um navio não deve voar: novamente, tenho que deixar uma implementação vazia.

Para piorar a situação, o que acontece se eu adicionar um veículo isca? Esse também não deve "voar" e não queremos continuar adicionando implementações vazias. Atenção com este caso em particular, pois não deve atacar, nem se defender. Nesse caso, você não pode começar a deixar implementações vazias por todo o lugar - vimos que essa não é a solução certa, por uma série de razões. Além disso, estaríamos duplicando o código em todos os lugares e deixando o software suscetível a erros.



Poderíamos fazer classes abstratas, nas quais temos implementações vazias por padrão. Porém, se precisarmos de uma classe que tenha diferentes combinações de ataque(), defender(), voar(), mergulho() e assim por diante, perderíamos muito quando se trata de polimorfismo.

Os problemas das soluções levantadas

Com esses aspectos em mãos, podemos ver quais são as desvantagens da herança no estabelecimento do comportamento de um sistema de armas.

- O código é duplicado nas filhas.
- Uma simples mudança pode afetar todo o modelo.
- Mudar o comportamento dos veículos em tempo de execução é quase impossível.

Dessa forma, que outra construção vimos que poderia resolver este problema? Crie o diagrama de classe com a sua solução.