Universidade Federal do Rio Grande do Norte
Escola de Ciências e Tecnologia
ECT2540 – Programação Orientada a Objetos 2019.1
Prof. Bruno Silva

Projeto: Arena de Robôs

1 Introdução

O projeto final da disciplina de Programação Orientada a Objetos consiste em implementar um programa em Python com interface gráfica TK que represente uma arena composta por vários robôs. Para isto, você deve implementar as hierarquias de classes envolvidas no processo:

- Classes para os tipos de robôs
- Classe para a arena
- Classes para os tipos de sensores
- Classe(s) para a interface gráfica do programa

Na nossa abstração, uma arena é uma área retangular no plano 2D na qual várias formas geométricas, representando robôs e obstáculos, estão inseridas. Um robo deve se mover na arena e detectar obstáculos e outros robôs através dos seus sensores.

O trabalho deve fazer uso de funcionalidades vistas na disciplina, tais como abstração, encapsulamento, herança, polimorfismo e associação entre objetos.

1.1 Informações Sobre a Entrega do Projeto

- **Grupo:** máximo 2 alunos (recomendável: 2 alunos)
- **Entregável:** código fonte e (possível) diagrama de classe em um arquivo .zip
- **Prazo:** até 11:59 de 26/06 (SIGAA)
- **Apresentação:** aula do dia 26/06 (arguição com professor)

O restante deste documento apresenta os detalhes sobre as classes envolvidas no programa. O seu grupo pode implementar a modelagem sugerida como segue ou implementar uma nova modelagem, que deve ser bem justificada e estar documentada em um diagrama de classes.

2 Abstração para Elementos Gráficos

Uma maneira de se trabalhar com interfaces gráficas é separar a parte do código correspondente ao domínio do problema da parte que contém as funcionalidades relacionadas à exibição e tratamento de eventos.

Com esta estratégia, achar falhas no código ou estendê-lo se torna mais fácil, uma vez que as funcionalidades do domínio do problema e interface gráfica estarão desacopladas. Além disto, o código final possui tamanho menor por ter sido implementado com o reuso de funcionalidades.

2.1 Atributos e Métodos da Classe ItemTK

Com o prpósito de facilitar a implementação do projeto conforme discutido, é proposto o uso de uma classe base abstrata para representar um item no Canvas TK. Esta classe, chamada de `ItemTK`, é exibida na Figura 1.

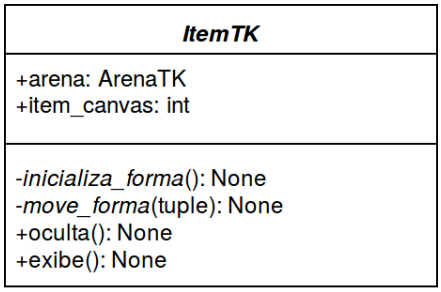


Figura 1: Classe representando um item no canvas TK.

2.1.1 Atributos

- `arena`: referência para instância da classe Arena onde o item está desenhado. Na nossa abstração, uma Arena é um `tk.Canvas`
- `item_canvas`: inteiro contendo o número do item, recebido após as funções `create_<forma>` da classe `tk.Canvas` serem chamadas

Outros atributos podem ser utilizados, tais como cor de contorno do item, cor de preenchimento, etc.

2.1.2 Método `inicializa_forma`

Este método é responsável por chamar as funções do tipo `create_<forma>` da classe `tk.Canvas` para desenhar uma forma e obter o `item_canvas` correspondente a esta forma.

O método é abstrato, ou seja, cada implementação concreta desta classe é que deve chamar `create_oval`, `create_rectangle`, `create_polygon`, etc. conforme o caso.

2.1.3 Método `move_forma`

Este método é responsável por mover a forma geométrica associada ao `item_canvas` no `tk.Canvas` em que ele se encontra.

O método é abstrato, ou seja, cada implementação concreta desta classe é que deve chamar o método `move` ou o método `coords` da classe `tk.Canvas`.

2.1.4 Método `oculta`

Este método é responsável por ocultar a forma geométrica associada ao `item_canvas` no `tk.Canvas` em que ele se encontra.

Para isto, utilize o método `itemconfig` da classe `tk.Canvas` com a configuração `state='hidden'`.

2.1.5 Método `exibe`

Este método é responsável por exibir a forma geométrica associada ao `item_canvas` no `tk.Canvas` em que ele se encontra.

Para isto, utilize o método `itemconfig` da classe `tk.Canvas` com a configuração `state='normal'`.

3 Visão Geral do Sistema

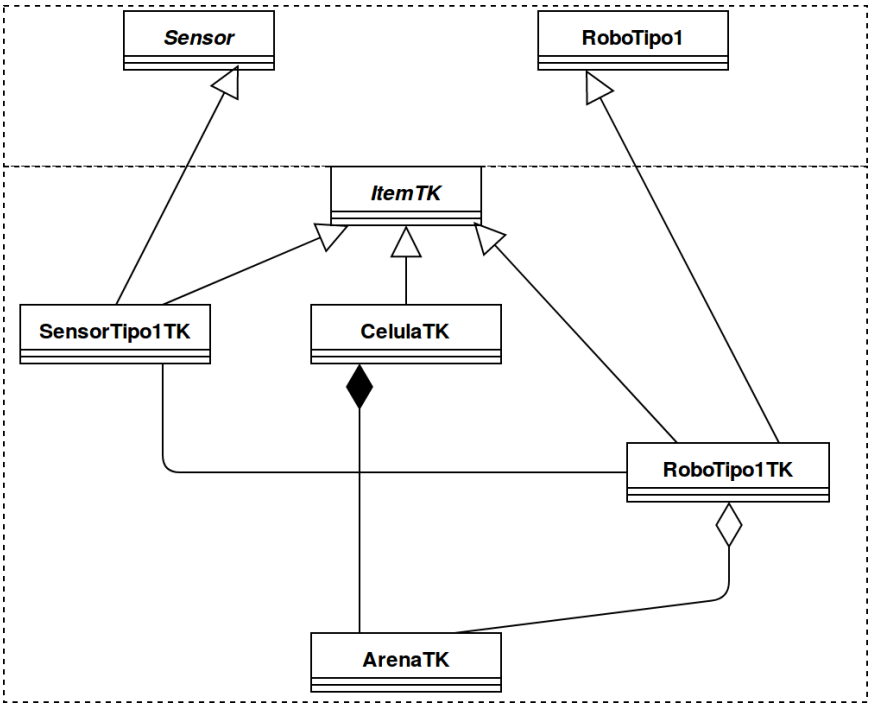


Figura 2: Visão geral das classes do sistema. As classes na parte superior fazem parte do domínio do problema. As classes na parte inferior compatibilizam as classes do domínio com a interface gráfica TK.

Na Figura 2 é possível visualizar as classes a serem desenvolvidas para o sistema e o relacionamento entre elas. Observe a divisão entre classes que pertencem ao domínio do problema (parte superior do diagrama) e classes com funcionalidades relacionadas à interface gráfica (parte inferior do diagrama). Observe ainda que o recurso de herança múltipla é utilizado para simplificar a implementação do sistema.

4 Hierarquia de Robôs

A hierarquia de robôs do sistema é formada por uma classe base abstrata representando um robô e três classes concretas, que modelam robôs de três tipos diferentes. Cada classe implementa um comportamento específico quanto ao movimento do robô na arena. A Figura 3 ilustra esta hierarquia.

4.1 Atributos e Métodos da Classe Base Abstrata Robô

A classe base abstrata que modela um robô contém o comportamento comum a todos os robôs. Seus atributos e métodos são descritos como segue.

4.1.1 Atributos

- `nome`: string representando o nome do robô

4.1.2 Método `move`

Move o robô de acordo com um vetor deslocamento dado por uma tupla.

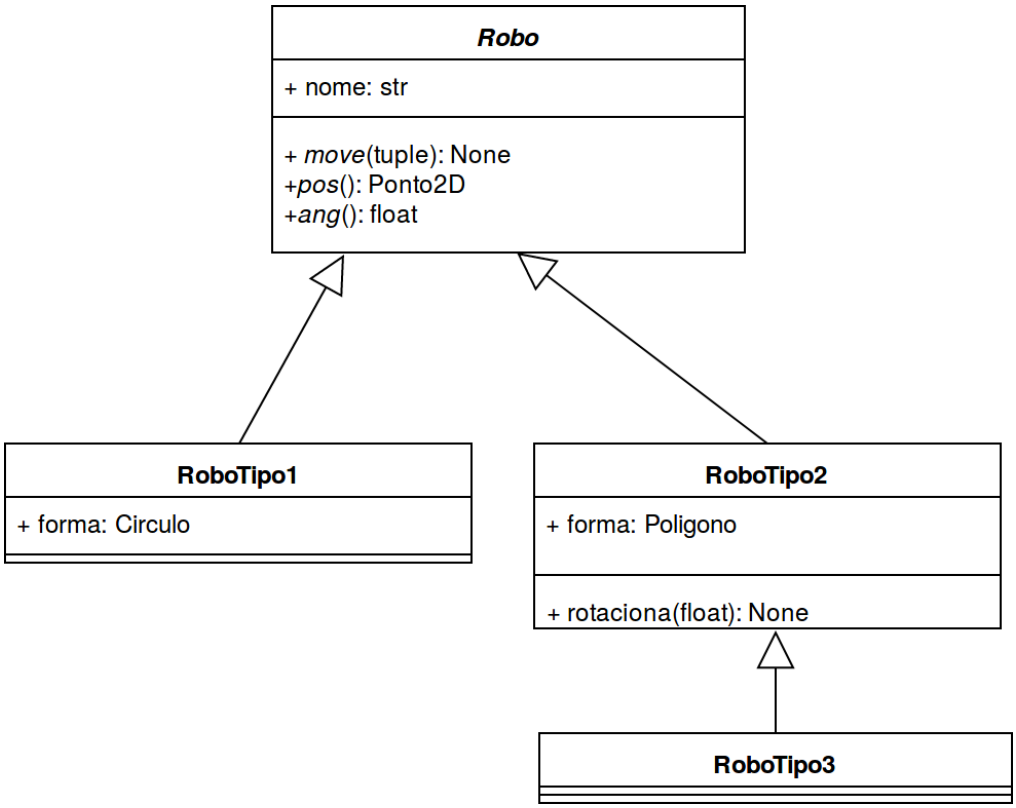


Figura 3: Hierarquia das classes de robôs a serem implementadas.

4.1.3 Método `pos`

Retorna a posição do robô (coordenadas x e y) em relação ao ponto $(0, 0)$ (origem da arena na qual o robô está situado).

4.1.4 Método `ang`

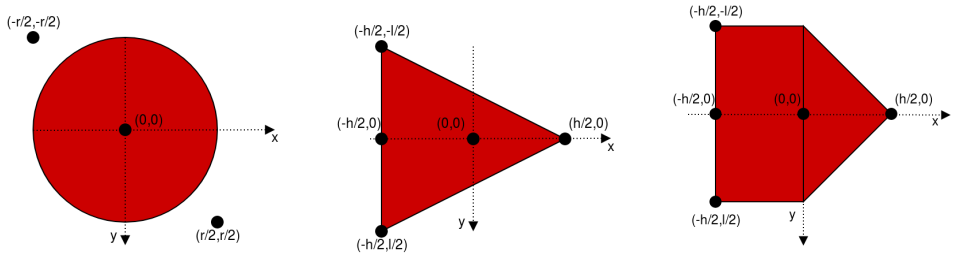
Retorna o ângulo de orientação do robô (valor entre 0 e 359 graus).

4.2 Comportamentos Concretos para os Robôs

Cada tipo concreto de robô tem uma forma geométrica associada. conforme mostra a Figura 4. No seu código, você precisa implementar pelo menos um tipo de robô dentre os exibidos a seguir, mas o seu programa precisa ser compatível com todos os tipos de robôs descritos neste documento.

A forma geométrica que representa o robô é quem de fato armazena os dados sobre a sua posição e orientação, daí a necessidade dos métodos `pos` e `ang`. Além disto, ela também armazena os atributos que serão utilizados para desenhar cada robô na tela, como por exemplo o centro e raio do círculo (para robôs tipo 1) e o centro e vértices do polígono (para robôs tipo 2 e 3).

Além disso, cada classe concreta deve implementar o método `move` (e o método `rotaciona`, se for o caso) de acordo com o diferentes características, como descrito a seguir.



(a) Robô tipo 1 (b) Robô tipo 2 (c) Robô tipo 3

Figura 4: Formas geométricas relacionadas aos tipos de robôs no sistema proposto.

4.2.1 Robô Tipo 1

Este robô possui a forma geométrica de um círculo, e denota um robô que pode se mover ao longo das duas direções (horizontal e vertical) da arena. Neste robô, não há a ideia de ângulo de orientação, ou seja, não existe a "frente" do robô. Os seus eixos de orientação, mostrados como X_r e Y_r na Figura 5, estão sempre alinhados com os eixos X e Y que formam o sistema de coordenadas fixo na arena (sistemas de coordenadas da classe Canvas do TK).

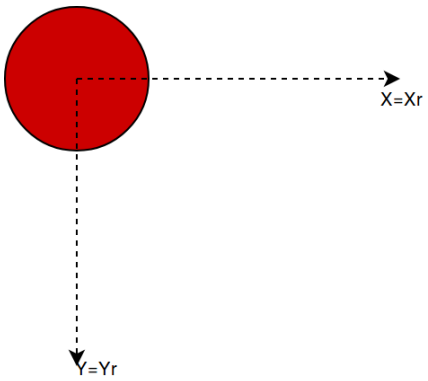


Figura 5: Movimento do robô tipo 1.

Este tipo de robô se movimenta para cima (direção $Y-$ da arena), para baixo (direção $Y+$), para a esquerda (direção $X-$) e para a direita (direção $X+$)

4.2.2 Robô Tipo 2

Este robô possui a forma geométrica de um triângulo isósceles (dois lados iguais). O vértice oposto à base do triângulo denota a frente do robô. Esta informação é importante porque este tipo de robô somente se movimenta em direção à sua frente e às suas costas.

Para que ele possa se movimentar para todas as posições da arena, ele também pode rotacionar em torno do seu centro. Com este movimento, as direções X_r e Y_r, correspondentes aos eixos do robô costas-frente e esquerda-direita, respectivamente, passam a estar rotacionados de um ângulo θ em relação aos eixos X e Y que denotam o sistema de coordenadas fixo na arena. A Figura 6 ilustra a mudança nos eixos.

Para determinar as coordenadas de um ponto (x^r, y^r) no sistema de coordenadas do robô (sistema rotacionado), devemos transformar as coordenadas originais do ponto (x, y), referenciado no sistema da arena, através da aplicação de três passos:

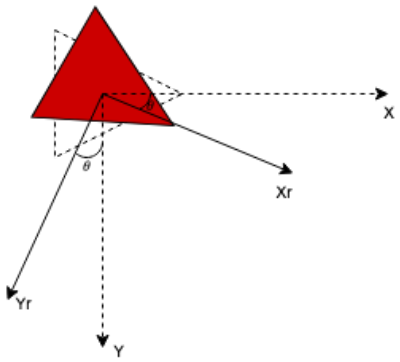


Figura 6: Movimento do robô tipo 2.

1. Mover o ponto (x, y) para onde está o eixo de rotação. Isto é feito aplicando uma translação para a posição do robô $(r_x, r_y)^t$:
$$(x', y') = (x, y) - (r_x, r_y)$$
2. Aplicar uma transformação de rotação em torno da origem no ponto (x', y') obtido no passo anterior:
$$x'' = \cos\theta x' - \sin\theta y'$$
$$y'' = \sin\theta x' + \cos\theta y'$$
3. Mover o ponto (x'', y'') obtido no passo anterior de volta para a sua posição original:
$$(x^r, y^r) = (x'', y'') + (r_x, r_y)$$

Ao mover um robô deste tipo, a rotação de pontos deve ser aplicada ao vetor deslocamento para que seja determinado o deslocamento no sistema de coordenadas do robô, determinando se o movimento será para frente ou para trás.

4.2.3 Robô Tipo 3

Este tipo de robô deve possuir uma geometria diferente da utilizada para o robô tipo 2. Esta geometria pode ser a sugerida na Figura 4, mas qualquer outro polígono em que a orientação do robô esteja visível pode ser utilizado.

Este robô é um robô do tipo 2 com uma capacidade a mais: ele também se desloca lateralmente, isto é, ao longo dos dois sentidos da sua direção y_r.

4.2.4 Código Utilitário

Para implementar as classes concretas descritas nesta seção, utilize o módulo contido no arquivo `geometria.py`. Este módulo utilitário contém classes e métodos relacionados a pontos 2D e polígonos. Por exemplo, com ele é possível chamar um método para rotacionar um ponto em torno de outro. Obviamente, o módulo pode ser estendido para suportar outras formas geométricas.

4.3 Classe `RoboTipo<Num>TK`

Conforme exibido na Figura 2, classes do tipo `RoboTipo<Num>TK`, que implementam um robô tipo 1, 2 ou 3 na interface gráfica TK, devem herdar simultaneamente de um robô concreto (tipo 1, 2 ou 3) e da classe base abstrata `ItemTK`.

Assim, esta classe deve implementar os métodos abstratos da classe `ItemTK`, definindo qual a forma geométrica associada a um robô e como ela deve se mover em uma arena.

Além disto, esta classe necessita ter um novo método que deve ser chamado quando houver eventos do teclado. Este método de *callback*, chamado de `processa_teclado`, é responsável por tratar eventos do teclado detectados pela interface gráfica TK e realizar ações do robô.

Como exemplo, um robô tipo 1 pode ir para frente caso seja pressionada a tecla w, para trás caso seja pressionada a tecla s, para cima caso seja pressionada a tecla a e para baixo caso seja pressionada a tecla d.

Um robô tipo 2 pode também rotacionar no sentido horário caso seja pressionada a tecla q e rotacionar no sentido anti-horário caso seja pressionada a tecla e.

Um robô tipo 3 se movimenta como robôs tipo 2, podendo ainda realizar um deslocamento para a sua esquerda (direção y_r-) caso seja pressionada a tecla a e um deslocamento para a sua direita (direção y_r+) caso seja pressionada a tecla d.

Importante: lembre-se que para processar eventos do teclado, o método `focus_set` deve ter sido chamado anteriormente pelo objeto da classe `tk.Canvas` (ou seja, pelo objeto Arena).

5 Arena

A classe que modela uma arena é um Canvas TK composto de várias células. Cada célula é um quadrado simbolizando uma porção da área da arena, que pode se encontrar livre para navegação (um robô pode navegar pela célula) ou ocupada com algum obstáculo (um robô não pode navegar pela célula). A arena mantém também vários robôs em uma agregação.

A Figura 7 exibe um robô navegando por uma arena à medida que vai descobrindo se cada célula da arena é navegável (cor branca) ou não (cor preta) com os seus sensores. Por ora, desconsidere o fato de um robô poder atravessar uma célula representando um obstáculo (esta funcionalidade não precisa ser implementada).

O que você deve implementar é inicializar cada célula como invisível e fazer com que elas se tornem visíveis a medida em que os sensores dos robôs as detectem em suas medições.

5.1 Atributos e Métodos da Classe Arena

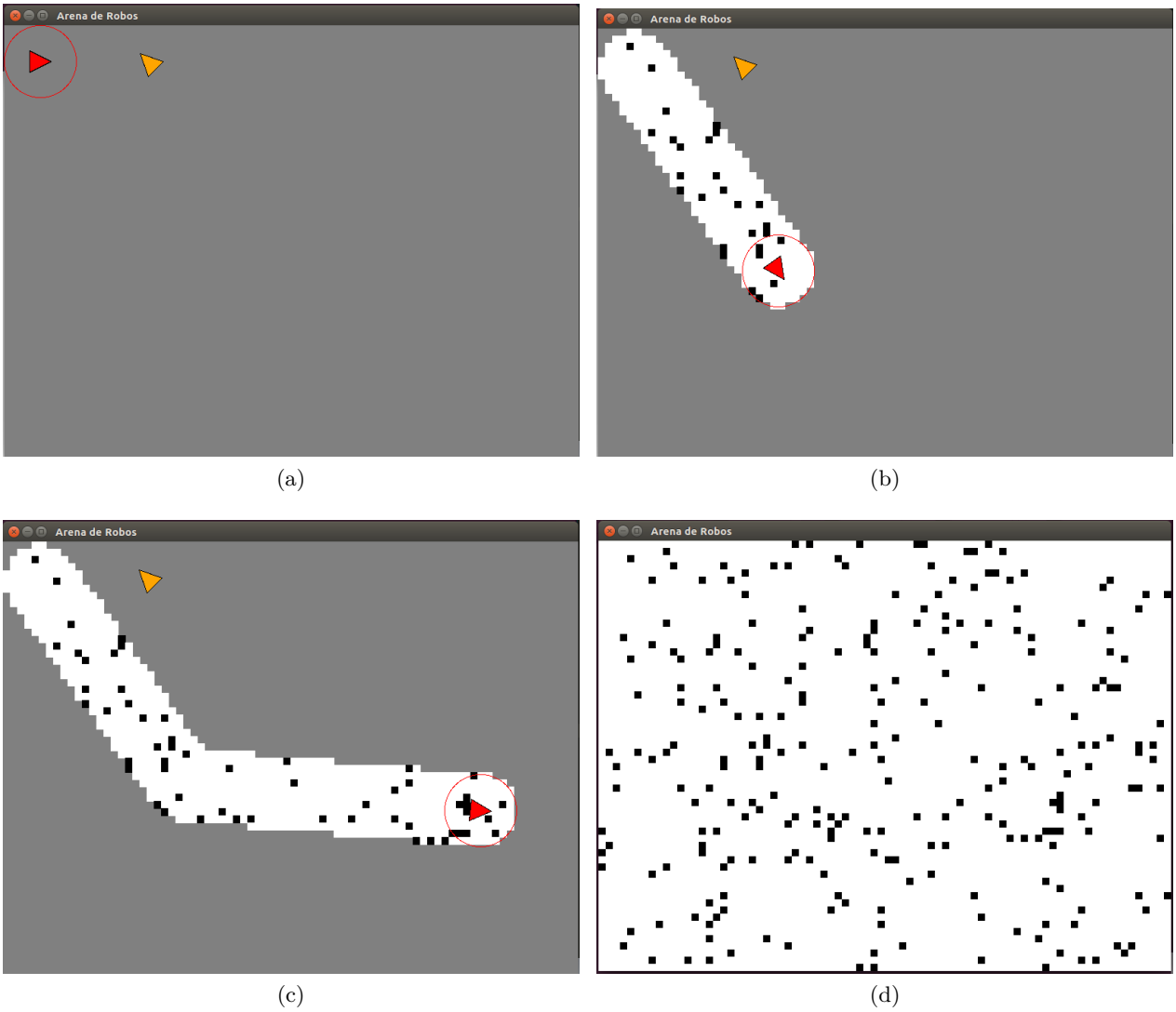
Os atributos e métodos que formam o comportamento de uma arena (exibidos na Figura 8) são descritos como segue.

5.1.1 Atributos

- `largura`: inteiro com o número de células para a largura
- `altura`: inteiro com o número de células para a altura
- `tam_celula`: inteiro com o tamanho de cada célula (em pixels)

Desta forma, a largura em pixels da arena é dada pela sua largura multiplicada pelo tamanho da célula e a altura em pixels é dada pela sua altura multiplicada pelo tamanho da célula.

Para a agregação de robôs/composição de células, recomenda-se utilizar um dicionário formado por pares `item_canvas: objeto`: a chave é o item do canvas (forma geométrica) associado à instância da classe que tem o item como atributo de instância.



(a) (b)

(c) (d)

Figura 7: Robô navegando em uma arena e descobrindo se as células estão livres (brancas) ou possuem obstáculos (pretas).

5.1.2 Método `desenha`

No TK, sempre que um item é criado no canvas, ele permanece desenhado, a não ser que o seu estado (`state`) seja alterado para oculto (`hidden`) ou que ele seja removido com o método `delete`. É necessário portanto um método que controle o desenho da arena, ou seja, desenhe os robôs e as células percorridas por cada robô. Controlar se estes últimos devem ou não ser desenhados, de acordo com os sensores de cada robô presente na arena, é responsabilidade deste método.

Assim, este método deve ser chamado a cada vez que o TK irá redesenhar a arena. O TK não possui uma função *callback* de desenho, mas qualquer *widget* possui um método que possibilita chamar uma função de *callback* a cada intervalo de tempo, especificado em milissegundos. Tal método se chama `after`, e deve receber como parâmetros o intervalo de tempo em que ele deve chamar o *callback* e a referência da função de *callback*. Como exemplo, `arena.after(10, arena.desenha)` registra como *callback* o método `desenha` do objeto `canvas`, fazendo com que este método seja chamado após 10 milissegundos que o `after` foi chamado. Observe que o *callback* é

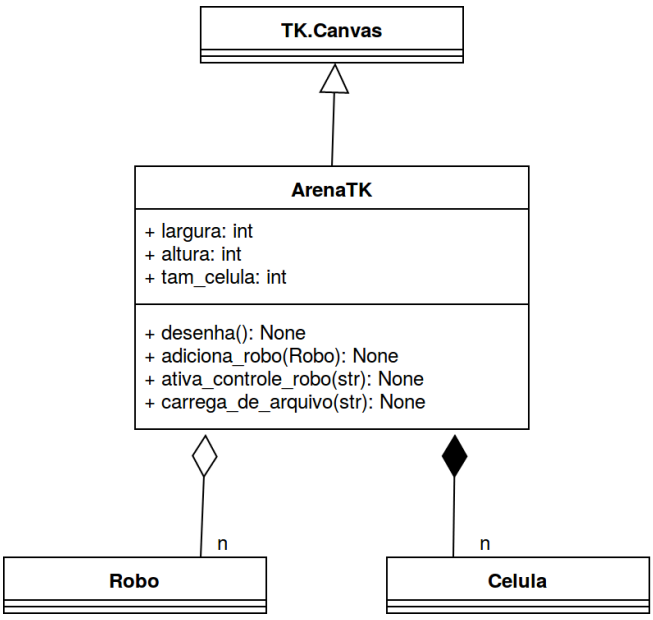


Figura 8: Classe arena e suas relações com as classes robô, obstáculo e TK.Canvas.

chamado somente uma vez depois que o método `after` é chamado: para que a mesma função de *callback* seja chamada continuamente, ela deve chamar novamente o método `after` com os mesmos parâmetros.

Dentro do método `desenha`, deve haver chamadas a outros método privados, responsáveis por desenhar/ocultar as células, desenhar/ocultar os sensores e desenhar/ocultar os robôs.

5.1.3 Método `adiciona_robo`

Este método deve ser chamado para associar um robô à uma arena. Para isto, o trabalho do método é adicionar o robô ao atributo de instância da classe arena que é responsável por manter todos os robôs. Utilizando dicionários, a chave é o item do canvas associado ao robô e o valor é a instância da classe robô que tem a chave correspondente como item no canvas.

5.1.4 Método `ativa_controle_robo`

Este método deve ser chamado quando um robô com nome passado como parâmetro deve ser ativado para controle. Mais especificamente, o método deve buscar pelo robô de nome informado, ativá-lo (fazer com que ele receba eventos do teclado) e desativar todos os outros robôs.

5.1.5 Método `carrega_de_arquivo`

Este método carrega uma arena a partir do arquivo de texto com nome passado como parâmetro. O arquivo que representa uma arena deve possuir a estrutura a seguir:

```
# arena <largura> <altura> <tam_cel>
arena 10 5 50
# as proximas LxA linhas especificam se uma célula contém ou não um obstáculo,
# com valor 1 ou 0 respectivamente.
# após esta linha não há mais comentários.
0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 1 1 1 1 0 0 0
0 0 0 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

5.2 Classe Utilitária Célula

Uma célula que compõe uma arena é um item desenhável em um `tk.Canvas` e portanto, deve herdar da classe base abstrata `ItemTK`. Esta classe utilitária possui alguns atributos a mais como mostra a Figura 9.

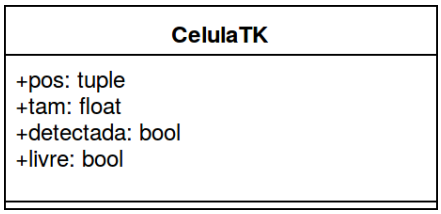


Figura 9: Classe para representar uma célula que compõe uma arena TK.

- `posicao`: posição 2D no canvas contendo onde está situada a célula
- `tam`: inteiro contendo o tamanho (em pixels) de cada célula
- `detectada`: booleano que informa se a célula foi detectada pelos sensores do robô ou não
- `livre`: booleano que informa se a célula é uma posição livre na arena (pode ser navegável por robôs) ou não

Em relação aos métodos abstratos das suas classe base, uma célula deve implementar o método `inicializa_forma` chamando o método `create_rectangle` para desenhar um quadrado. O conjunto de todas as células que formam a arena devem estar inicialmente ocultos. Uma célula não se move e portanto, o método `move_forma` deve ser sobrescrito com uma implementação sem código (corpo vazio).

6 Sensores

Além de se mover na arena na qual está inserido, um robô interage com ela detectando posições livres e posições ocupadas com algum obstáculo. Isto é feito com o uso de sensores. Para isto, um robô deve ser construído passando para ele uma referência da classe sensor.

Equipe cada robô com cada tipo de sensor e navegue o robô na arena para detectar objetos (robôs/células) presentes na arena.

6.1 Atributos e Métodos da Classe Base Abstrata Sensor

Sensor é uma classe base abstrata que define o comportamento de objetos deste tipo. Os atributos e métodos que compõem a classe (Figura 10) são descritos no que segue.

6.1.1 Atributos

- `robo`: referência para objeto da classe robô com o qual o sensor está associado
- `robos_detec`: lista de robôs detectados em uma medição do sensor
- `celulas_detec`: lista de células detectadas em uma medição do sensor

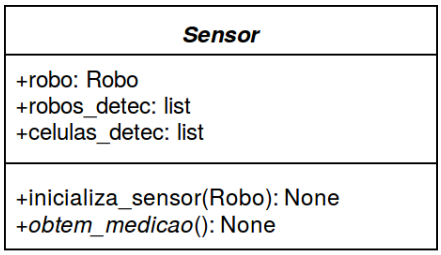


Figura 10: Classe base abstrata para representar um sensor.

6.1.2 Método `inicializa_sensor`

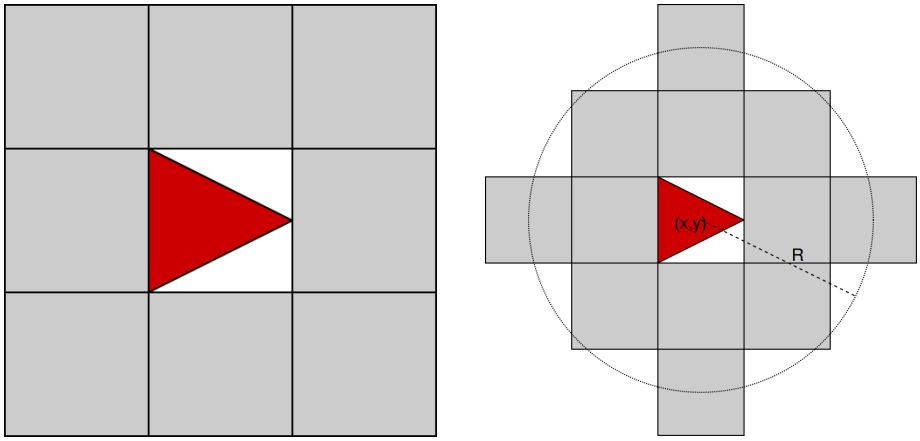
Recebe um robô como parâmetro, ao qual o sensor deve ser associado. Após a chamada deste método, o atributo `robo` passa a ser válido, habilitando chamadas ao método `obtem_medicao`.

6.1.3 Método `obtem_medicao`

Este método realiza uma medição sensorial. Na nossa abstração, ele deve limpar as listas de robôs/células detectadas na medição anterior e inserir nestas mesmas listas novas medições. Isto simula o comportamento real de robôs no qual uma medição sensorial retorna apenas os obstáculos/robôs detectados pelos sensores no momento em que eles são acionados, sem acumular com os detectados em instantes anteriores.

6.2 Classe `SensorTipo<Num>TK`

Como mostra a Figura 2, classes do tipo `SensorTipo<Num>TK` implementam um sensor tipo `<Num>` na interface gráfica TK, herdando simultaneamente das classes base abstratas `Sensor` e `ItemTK`. Os sensores são modelados como `ItemTK` porque eles podem ser desenhados em um canvas.



(a) Sensor de proximidade 1. (b) Sensor de proximidade 2.

Figura 11: Tipos de sensores e respectivas formas de detecção de obstáculos presentes na arena.

Esta classe deve implementar de fato como os sensores devem detectar células/robôs presentes na arena. A Figura 11 sugere dois tipos de sensores de proximidade. Com formas geométricas arbitrárias compondo um sensor, os métodos de busca de itens da classe `tk.Canvas` (`find_overlapping`, `find_all`, etc.) podem ser utilizados para inserir robôs/células detectados pelo sensor nas suas listas de detecções.

Além de implementar os métodos abstratos das suas classes base, esta classe deve também implementar o método `desenha_deteccoes`, que será chamado pela classe `Arena` (que é quem controla o desenho da arena).

Os sensores devem ser acionados pelo método *callback* de teclado do robô, de duas formas: 1. automaticamente, à medida em que o robô se move ou 2. manualmente, pressionando uma tecla especial para ativar o sensor sempre que for o usuário controlando o robô deseje realizar uma medição sensorial.

7 Aplicação TK com Interface Gráfica para Usuário

A classe Arena é um `tk.Canvas`. Sendo assim, ela deve ser inserida em uma janela de uma aplicação TK que comporte funcionalidades básicas.

Dentre elas, deve ser possível adicionar um robô na arena (com nome, posição e orientação informados em algum widget), escolher um robô presente na arena para ser controlado (via clique de mouse) e visualizar qual o robô ativo no momento (nome, posição e orientação).