PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

INF1301 Programação Modular - Período 2015-2

Prof. Flavio Bevilacqua

TRABALHO 3

Fernanda de Miranda Carvalho - 1411287

Marcelo Paulon Jucá Vasconcelos - 1411029

Renan da Fonte Simas dos Santos - 1412122

1. Especificação de Requisitos

- Requisitos Funcionais

RF01 - Criar Tabuleiro de Gamão

Neste jogo haverá um tabuleiro contendo 24 casas, sendo 12 delas na parte superior, e 12 na parte inferior. Cada conjunto de 6 casas constitui um quadrante. Os dois quadrantes à direita do tabuleiro são chamados “internos”.

RF02 - Criar peças

Criar peças brancas e pretas para serem movimentadas ao longo da partida.

RF03 - Dispor peças no Tabuleiro para início de partida

Para possibilitar o início de uma nova partida é necessário dispor as peças no tabuleiro da seguinte forma:

- 5 peças brancas na décima segunda casa da parte superior (da direita para a esquerda)

- 5 peças pretas na décima segunda casa da parte inferior (da direita para a esquerda)

- 5 peças brancas na sexta casa da parte superior (da direita para a esquerda)

- 5 peças pretas na sexta casa da parte inferior (da direita para a esquerda)

- 3 peças pretas na oitava casa da parte superior (da direita para a esquerda)

- 3 peças brancas na oitava casa da parte inferior (da direita para a esquerda)

- 2 peças brancas na primeira casa da parte superior (da direita para a esquerda)

- 2 peças pretas na primeira casa da parte inferior (da direita para a esquerda)

Totalizando 15 peças brancas e 15 peças pretas no tabuleiro.

RF04 - Disponibilizar 2 dados para movimentação das peças

Criar dois dados. Cada dado possui seis lados numerados de 1 a 6.

RF05 - Verificar jogador que inicia a partida

Os dados devem ser lançados pelos jogadores e quem obtiver o maior valor será o primeiro a jogar. Em caso de empate os dados devem ser lançados até obter valores diferentes.

RF06 - Iniciar partida

Disponibilizar tabuleiro e dados para início de partida e informar qual jogador deverá iniciar.

RF07 - Dobrar valor da partida

Deve ser possível, a qualquer momento da partida, que um jogador dobre o valor da aposta. Se o oponente recusar, a partida deve ser encerrada e o jogador que propôs a dobra a vence. Se aceitar, ele também poderá propor a dobra. Uma partida deve ser iniciada valendo 1 ponto.

RF08 - Jogar dados

A cada rodada dois dados são lançados, obtendo números de 1 a 6 cada.

RF09 - Movimentar peças

Dada uma casa de origem e uma casa de destino válidas deve ocorrer a movimentação de uma peça.

RF10 - Permitir jogada dupla

Se os valores obtidos nos dois dados lançados forem os mesmos, o turno atual passa a ter 4 movimentações.

RF11 - Capturar uma peça

Se uma peça estiver sozinha em uma casa e uma peça oponente for movimentada para lá, esta será capturada (a peça capturada vai para o BAR).

RF12 - Retornar com uma peça

Cada peça capturada deve ser devolvida para uma casa livre antes de qualquer outra movimentação. O que determina o número da casa é o valor tirado em cada dado, caso seja possível a movimentação.

RF13 - Verificar se uma casa está livre

Uma casa é dita livre para receber uma peça se nela houver apenas outras peças da mesma cor ou somente uma da cor do oponente (ou se estiver vazia).

RF14 - Impedir uma movimentação

Se a casa de destino não estiver livre ou o valor percorrido até ela não corresponder ao obtido no dado, a peça não será movimentada e o jogador será notificado.

RF15 - Retirar peça do Tabuleiro

É possível iniciar a retirada de uma peça do tabuleiro caso todas as peças dessa mesma cor (inclusive ela própria) estiverem no quadrante interno do jogador.

RF16 - Finalizar uma partida

Uma partida deve ser finalizada quando um dos dois jogadores retirar todas as suas peças do tabuleiro.

RF17 - Salvar pontuação da partida

Salvar pontuação acumulada ao término de uma partida.

- <u>Requisitos Não Funcionais</u>

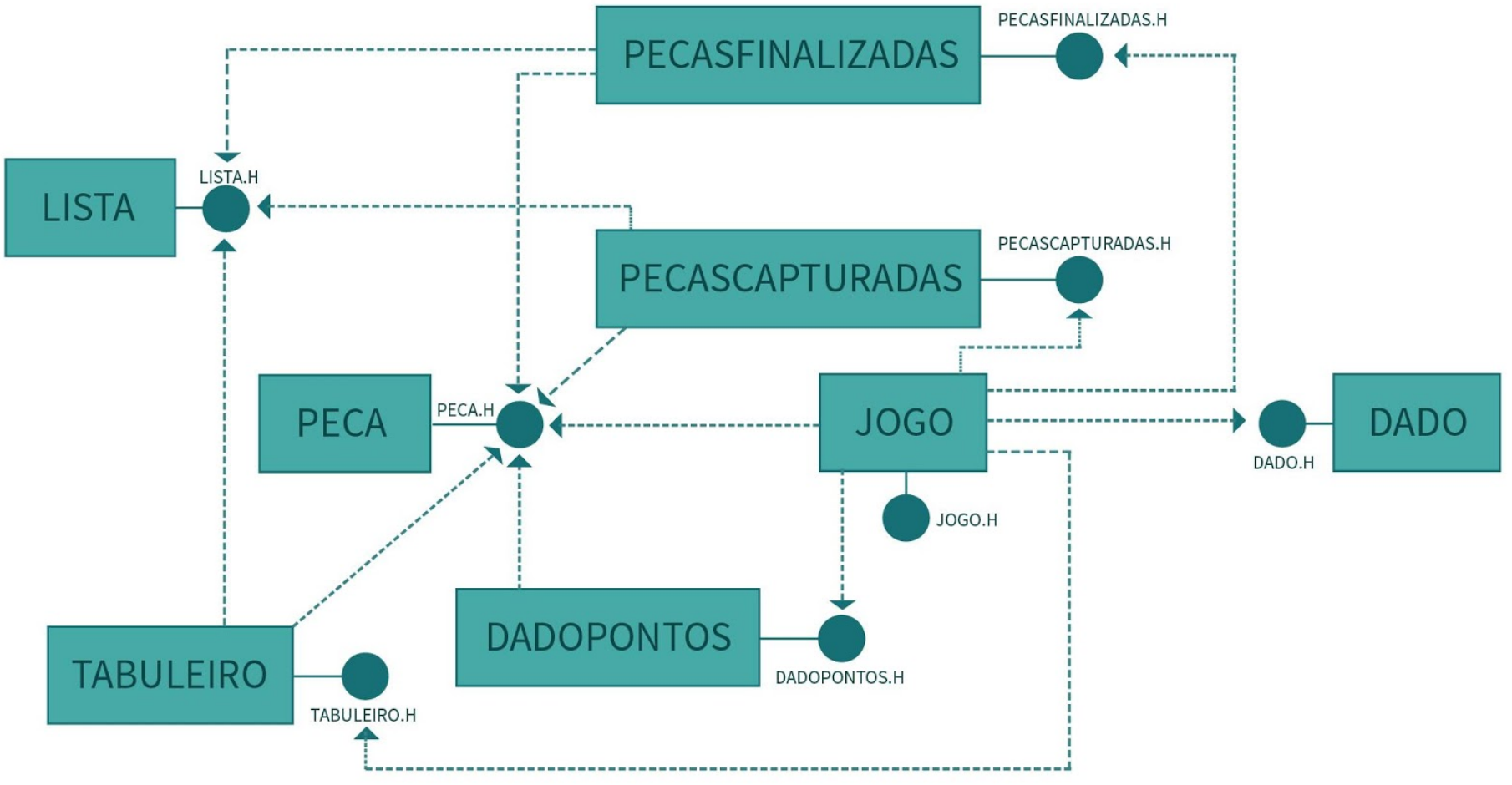
RNF01 - Execução

Este jogo deve ser compatível com sistemas operacionais Windows e não deve exigir nenhuma instalação. Para iniciá-lo será necessário apenas executar o programa.

RNF02 - Manutenção

O programa é modularizado e contém comentários sobre as funções de cada módulo nas respectivas interfaces, permitindo e facilitando, assim, a sua manutenção.

2. Modelo de Arquitetura



Funções disponibilizadas em cada interface:

I. Lista - LIS

LIS_tppLista LIS_CriarLista (void (* ExcluirValor) (void * pDado));

Assertivas de Entrada:

- Deve existir um ponteiro para uma função de exclusão;
- Deve existir um ponteiro genérico para o tipo de dado que se deseja armazenar na lista criada.

Assertivas de Saída:

- Lista foi criada.

void LIS_DestruirLista(LIS_tppLista pLista);

Assertivas de Entrada:
- Deve existir uma lista para ser destruída.

Assertivas de Saída:
- Lista foi destruída.

void LIS_EsvaziarLista(LIS_tppLista pLista);

Assertivas de Entrada:
- Deve existir uma lista para ser esvaziada.

Assertivas de Saída:
- Lista foi esvaziada, não armazena valores.

LIS_tpCondRet LIS_InserirElementoAntes(LIS_tppLista pLista, void * pValor);

Assertivas de Entrada:
- Deve existir uma lista onde será inserido o valor;
- Deve existir um ponteiro para um valor a ser armazenado na lista.

Assertivas de Saída:
- Se pValor não é nulo, o valor para o qual ele aponta é inserido no início da lista.
- A lista tem um novo início e ponteiro corrente aponta pra ele.

LIS_tpCondRet LIS_InserirElementoApos(LIS_tppLista pLista, void * pValor);

Assertivas de Entrada:
- Deve existir uma lista onde será inserido o valor;
- Deve existir um ponteiro para um valor a ser armazenado na lista.

Assertivas de Saída:
- Se pValor não é nulo, o valor para o qual ele aponta é inserido no final da lista.
- A lista tem um novo fim e ponteiro corrente aponta pra ele.

LIS_tpCondRet LIS_ExcluirElemento(LIS_tppLista pLista);

Assertivas de Entrada:
- Deve existir uma lista de onde será excluído um elemento.

Assertivas de Saída:
- Se lista não é vazia, elemento é excluído.

void * LIS_ObterValor(LIS_tppLista pLista);

Assertivas de Entrada:
- Deve existir uma lista de onde será obtido um valor;

Assertivas de Saída:
- Se lista não é vazia obtém-se um valor, por referência.

void IrInicioLista(LIS_tppLista pLista);

Assertivas de Entrada:
- Deve existir uma lista.

Assertivas de Saída:
- Se lista é vazia ou tem apenas um elemento não há alterações no ponteiro corrente.
- Se lista tem mais de um elemento, ponteiro corrente avança para início da lista e passa a apontar para primeiro elemento.

void IrFinalLista(LIS_tppLista pLista);

Assertivas de Entrada:
- Deve existir uma lista.

Assertivas de Saída:
- Se lista é vazia ou tem apenas um elemento não há alterações no ponteiro corrente.
- Se lista tem mais de um elemento, ponteiro corrente avança para final da lista e passa a apontar para último elemento.

LIS_tpCondRet LIS_AvancarElementoCorrente(LIS_tppLista pLista, int numElem);

Assertivas de Entrada:
- Deve existir uma lista.
- Deve existir um valor inteiro referente à quantidade de elementos que o ponteiro corrente deve ser avançado.

Assertivas de Saída:
- Se lista é vazia ou tem apenas um elemento, ponteiro corrente não pode avançar;
- Se numElem é maior que tamanho da lista, ponteiro corrente não pode avançar;
- Se lista tem mais de um elemento e numElem não é maior que tamanho da lista, ponteiro corrente avança numElem elementos.

LIS_tpCondRet LIS_ProcurarValor(LIS_tppLista pLista, void * pValor);

Assertivas de Entrada:
- Deve existir uma lista onde um valor será procurado;
- Deve existir um ponteiro para valor que será procurado.

Assertivas de Saída:
- Se valor existe na lista, é retornado por referência.

II. Pecas - PCA

PCA_tpCondRet PCA_CriarPeca(PCA_tpPeca **pPeca, PCA_tpCorPeca CorPeca);

Assertivas de Entrada:

- Deve existir um ponteiro por onde será passada, por referência, a peça criada;
- Deve existir uma cor a ser associada à peça.

Assertivas de Saída:

- Peça foi criada com a cor passada e ponteiro aponta para ela.

PCA_tpCondRet PCA_ObterCorPeca(PCA_tpPeca *pPeca, PCA_tpCorPeca *CorPeca);

Assertivas de Entrada:

- Deve existir um ponteiro que apontará para a peça da qual será obtida a cor;
- Deve existir um ponteiro cujo conteúdo será atualizado com a cor da peça.

Assertivas de Saída:

- Caso a peça exista, o conteúdo do ponteiro CorPeca é atualizado com a cor da peça apontada pelo ponteiro pPeca.
- Caso a peça não exista, o conteúdo do ponteiro CorPeca não é alterado.

PCA_tpCondRet PCA_DestruirPeca(PCA_tpPeca **pPeca);

Assertivas de Entrada:

- Deve existir um ponteiro para a peça a ser destruída, passado por referência;

Assertivas de Saída:

- Peça foi destruída e ponteiro para ela passa a apontar para NULL.

III. Tabuleiro - TAB

TAB_tpCondRet TAB_CriarTabuleiro(TAB_tpTabuleiro **pTabuleiro);

Assertivas de Entrada:

- Deve existir um ponteiro por onde será passado, por referência, o tabuleiro criado.

Assertivas de Saída:

- Tabuleiro foi criado com uma lista de casas e para cada casa uma lista de peças;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

TAB_tpCondRet TAB_InserirPeca(TAB_tpTabuleiro *pTabuleiro, PCA_tpPeca *pPeca, int NumeroCasa);

Assertivas de Entrada:

- Deve existir um ponteiro para o tabuleiro onde será inserida a peça;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- Deve existir um ponteiro para uma peça que exista e tenha uma cor definida;
- Há um valor que corresponde a casa onde será inserida a peça.

Assertivas de Saída:

- Se tabuleiro e casa existem, então peça é inserida e ponteiro corrente aponta para ela;
- Se tabuleiro e casa não existem, então peça não é inserida e ponteiro corrente não muda;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

TAB_tpCondRet TAB_RemoverPeca(TAB_tpTabuleiro *pTabuleiro, PCA_tpCorPeca CorPeca, PCA_tpPeca **pPeca, int NumeroCasa);

Assertivas de Entrada:

- Deve existir um ponteiro para o tabuleiro de onde a peça será removida;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- Deve existir um ponteiro que receberá por referência a peça removida;
- Há um valor que corresponde à casa de onde será removida a peça.

Assertivas de Saída:

- Se tabuleiro ou casa ou peça não existem, a peça não é removida;
- Se tabuleiro, casa e peça existem a peça é removida, e o ponteiro passado por referência; passa a ter o endereço desta peça;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

TAB_tpCondRet TAB_MoverPeca(TAB_tpTabuleiro *pTabuleiro, PCA_tpCorPeca CorPeca, int NumeroCasaOrigem, int NumeroCasaDestino);

Assertivas de Entrada:

- Deve existir um ponteiro para o tabuleiro onde está a peça a ser movida;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- A casa de origem deve conter uma peça da mesma cor que foi passada;
- Devem existir valores correspondentes aos números das casas de origem e destino.

Assertivas de Saída:

- Se tabuleiro ou casa não existirem, a peça não será movida;
- Se na casa de destino existirem duas ou mais peças de outra cor, a peça não será movida;
- Se na casa de destino existirem apenas peças da mesma cor ou ainda, se estiver vazia, a peça será movida e ponteiro corrente apontará para ela;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

TAB_tpCondRet TAB_ContarPecas(TAB_tpTabuleiro *pTabuleiro, int NumeroCasa, PCA_tpCorPeca CorPeca, int *pContagem);

Assertivas de Entrada:

- Deve existir um ponteiro para o tabuleiro onde estão as peças que serão contadas;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- Deve existir um valor correspondente à casa onde será feita a contagem de peças;
- Deve existir um ponteiro para que a contagem das peças seja armazenada.

Assertivas de Saída:

- Se tabuleiro ou casa não existirem não haverá contagem de peças;
- Se tabuleiro e casa existirem, o ponteiro para a contagem será atualizado.
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

TAB_tpCondRet TAB_DestruirTabuleiro(TAB_tpTabuleiro **pTabuleiro);
Assertivas de Entrada:
- Deve existir um ponteiro para o tabuleiro a ser destruído;

Assertivas de Saída:
- Se tabuleiro existe ele é destruído e ponteiro para ele passa a apontar para NULL.

IV. DadoPontos - DPT

DPT_tpCondRet DPT_CriarDadoPontos(DPT_tpDadoPontos **pDadoPontos);
Assertivas de Entrada:
- Deve existir um ponteiro por onde será passado, por referência, o dado de pontos criado.

Assertivas de Saída:
- Dado de pontos foi criado e pDadosPontos foi atualizado.

DPT_tpCondRet DPT_AtualizarJogadorDobra(DPT_tpDadoPontos *pDadoPontos, PCA_tpCorPeca CorPeca);
Assertivas de Entrada:
- Deve existir um ponteiro para o dado de pontos que indica a dobra;
- Deve existir uma cor associada ao jogador que está dobrando a pontuação.

Assertivas de Saída:
- O jogador que pode dobrar a partida é atualizado através da cor da peça;

DPT_tpCondRet DPT_DobrarPontuacaoPartida(DPT_tpDadoPontos *pDadoPontos, PCA_tpCorPeca CorPeca);
Assertivas de Entrada:
- Deve existir ponteiro para o dado de pontos que será utilizado;
- Deve existir uma cor de peça associada ao jogador que está dobrando a pontuação.

Assertivas de Saída:
- Se dado pontos não existir ou cor da peça não for igual a cor autorizada a dobrar a pontuação o ponteiro pDadosPontos não é atualizado;
- Caso contrário, a pontuação é dobrada e o ponteiro é atualizado.

DPT_tpCondRet DPT_ObterPontuacaoPartida(DPT_tpDadoPontos *pDadoPontos, int *pPontuacao);
Assertivas de Entrada:
- Deve existir um ponteiro para o dado de pontos que será utilizado;
- Deve existir um ponteiro para ser atualizado com a pontuação da partida.

Assertivas de Saída:
- Se dado de pontos existir, a pontuação será obtida e o conteúdo do ponteiro pPontuacao será atualizado.

DPT_tpCondRet DPT_ObterJogadorDobraPartida(DPT_tpDadoPontos *pDadoPontos, PCA_tpCorPeca *pCorPeca);

Assertivas de Entrada:

- Deve existir um ponteiro para o dado de pontos que será utilizado;
- Deve existir um ponteiro para uma cor de peça, que será atualizado com a cor da peça do jogador que estiver com o dado.

Assertivas de Saída:

- Se dado de pontos existir e um dos jogadores estiver com ele, a cor do jogador detentor do dado será obtida e o conteúdo do ponteiro pCorPeca será atualizado.
- Se o dado de pontos existir mas nenhum dos jogadores estiver com ele, o conteúdo do ponteiro pCorPeca não será alterado.

DPT_tpCondRet DPT_DestruirDadoPontos(DPT_tpDadoPontos **pDadoPontos);

Assertivas de Entrada:

- Deve existir um ponteiro para o dado de pontos que será destruído, passado por referência.

Assertivas de Saída:

- Dado de pontos foi destruído e ponteiro agora aponta para NULL.

V. Dado - DAD

DAD_tpCondRet DAD_CriarDados(DAD_tpDado **pDados);

Assertivas de Entrada:

- Deve existir um ponteiro por onde serão passados, por referência, os dados criados.

Assertivas de Saída:

- Dados foram criados e ponteiro pDados está apontando para eles.

DAD_tpCondRet DAD_JogarDados(DAD_tpDado *pDados);

Assertivas de Entrada:

- Deve existir um ponteiro para o par de dados que será utilizado;

Assertivas de Saída:

- Dados foram lançados.

DAD_tpCondRet DAD_ObterValores(DAD_tpDado *pDados, int *pValorDado1, int *pValorDado2);

Assertivas de Entrada:

- Deve existir um ponteiro para o par de dados que será utilizado;
- Devem existir dois ponteiros cujos valores serão atualizados com os valores de cada dado.

Assertivas de Saída:

- O conteúdo de pValorDado1 e pValorDado2 foi atualizado com os valores de cada dado.

DAD_tpCondRet DAD_DestruirDados(DAT_tpDado **pDados);

Assertivas de Entrada:

- Deve existir um ponteiro para o par de dados que será destruído, passado por referência;

Assertivas de Saída:

- Dado foi destruído e ponteiro passa a apontar para NULL.

VI. PecasCapturadas - BAR

BAR_tpCondRet BAR_CriarListaPecasCapturadas(BAR_tpPecasCapturadas **pPecasCapturadas);

Assertivas de Entrada:

- Deve existir um ponteiro por onde será passada, por referência, a lista de peças capturadas que será criada.

Assertivas de Saída:

- Lista de peças capturadas foi criada.
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

BAR_tpCondRet BAR_InserirPeca(BAR_tpPecasCapturadas *pPecasCapturadas, PCA_tpPeca *pPeca);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista onde será inserida a peça;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- Deve existir um ponteiro para uma peça a ser inserida.

Assertivas de Saída:

- Se lista e peça existem, então peça é inserida e ponteiro corrente aponta para ela;
- Se lista ou peça não existem, então peça não é inserida e ponteiro corrente não muda;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

BAR_tpCondRet BAR_RemoverPeca(BAR_tpPecasCapturadas *pPecasCapturadas, PCA_tpCorPeca CorPeca, PCA_tpPeca **pPeca);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista de onde a peça será removida;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- Deve existir um ponteiro que receberá, por referência, a peça removida;
- Deve existir uma cor associada à peça que será removida.

Assertivas de Saída:

- Se lista é vazia ou não existe peça da cor associada, a peça não é removida;
- Se lista não é vazia e existe peça da cor associada, a peça é removida e o ponteiro passado por referência passa a ter o endereço desta peça;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

BAR_tpCondRet BAR_ContaPecas(BAR_tpPecasCapturadas *pPecasCapturadas, PCA_tpCorPeca CorPeca, int *pContagem);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista de peças capturadas que serão contadas;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- Deve existir uma cor associada às peças que devem ser contadas;
- Deve existir um ponteiro para que a contagem das peças seja armazenada.

Assertivas de Saída:

- Se lista é vazia não haverá contagem de peças;
- Se lista não é vazia e contém peças da cor buscada, o ponteiro para a contagem será atualizado.
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

BAR_tpCondRet BAR_DestruirListaPecasCapturadas(BAR_tpPecasCapturadas **pPecasCapturadas);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista de peças que será destruída;

Assertivas de Saída:

- Se lista de peças existe, ela é destruída e ponteiro para ela passa a apontar para NULL.

VII. PecasFinalizadas - PCF

PCF_tpCondRet PCF_CriarListaPecasFinalizadas(PCF_tpPecasFinalizadas **pPecasFinalizadas);

Assertivas de Entrada:

- Deve existir um ponteiro por onde será passada, por referência, a lista de peças finalizadas que será criada.

Assertivas de Saída:

- Lista de peças finalizadas foi criada.
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

PCF_tpCondRet PCF_InserirPeca(PCF_tpPecasFinalizadas *pPecasFinalizadas, PCA_tpPeca *pPeca);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista onde será inserida a peça;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- Deve existir um ponteiro para uma peça a ser inserida.

Assertivas de Saída:

- Se lista e peça existem, então peça é inserida e ponteiro corrente aponta para ela;
- Se lista ou peça não existem, então peça não é inserida e ponteiro corrente não muda;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

PCF_tpCondRet PCF_ContaPecas(PCF_tpPecasFinalizadas *pPecasFinalizadas, PCA_tpCorPeca CorPeca, int *pContagem);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista de peças finalizadas que serão contadas;
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça;
- Deve existir uma cor associada às peças que devem ser contadas;
- Deve existir um ponteiro para que a contagem das peças seja armazenada.

Assertivas de Saída:

- Se lista é vazia não haverá contagem de peças;
- Se lista não é vazia, o ponteiro para a contagem será atualizado.
- Valem as assertivas estruturais da Lista Duplamente Encadeada com cabeça.

PCF_tpCondRet PCF_DestruirListaPecasFinalizadas(PCF_tpPecasFinalizadas **pPecasFinalizadas);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista de peças que será destruída;

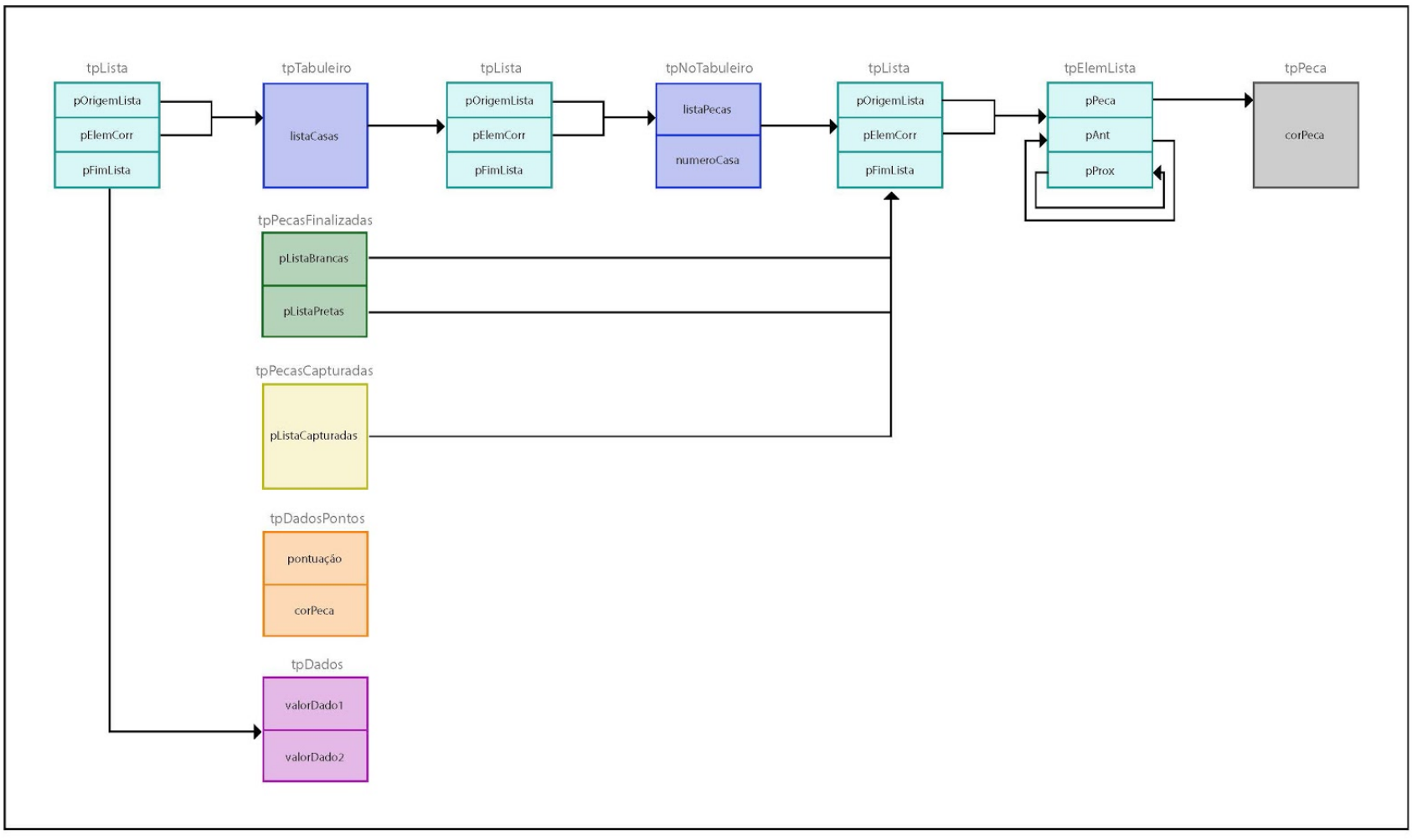
Assertivas de Saída:

- Se lista de peças existe, ela é destruída e ponteiro para ela passa a apontar para NULL.

VIII. Jogo - GAM

O módulo jogo não tem funções externas.

3. Modelo Estrutural



- <u>Assertivas estruturais:</u>

Tabuleiro -> Lista composta de 24 elementos possuindo, cada um, um ponteiro para um tipo Lista (onde cada elemento aponta para um tipo Peca). Valem as assertivas estruturais da lista duplamente encadeada com cabeça.

PecasFinalizadas -> Estrutura composta de duas listas onde:

- A primeira possui ponteiros para o tipo Peca (na cor branca);
- A segunda possui ponteiros para o tipo Peca (na cor preta).
- Valem as assertivas estruturais da lista duplamente encadeada com cabeça.

PecasCapturadas -> Estrutura composta de uma lista com ponteiros para tipo Peca (nas cores branca e preta). Valem as assertivas estruturais da lista duplamente encadeada com cabeça.

- Exemplo: