Engenharia de Software – Profª Ana C. V. de Melo
Projeto Kindred – Fase 3

Bruno Guilherme Ricci Lucas (4460596)
Leonardo Pereira Macedo (8536065)
Vinícius Bitencourt Matos (8536221)

1 de dezembro de 2015

O *use-case* que definimos na primeira fase contempla a criação e realização de uma partida completa do jogo, o que inviabiliza a criação de um teste para ele. O mesmo se aplica a um dos *statecharts* definidos na segunda fase, correspondente a uma partida inteira.

Desta forma, decidimos criar *use-cases* de partes do sistema, de modo a testar seus componentes.

1 Dados de teste

- **testClient_WrongIP:**
  - **Entrada:** Um client é criado e manipulado para se conectar a um servidor inexistente: um que possui o nome*fakehost*.
  - **Saída:** A saída esperada seria ocorrer um `RuntimeException`, mostrando que houve erro na conexão com o servidor.
- **testClient_CorrectIP:**
  - **Entrada:** Um client é criado e manipulado para se conectar a um servidor já previamente rodando no *localhost*.
  - **Saída:** A saída esperada seria não ocorrer um `RuntimeException`, visto que a conexão foi feita com sucesso.
- **testQuit:**
  - **Entrada:** É criado e inicializado um Client, que logo envia um comando para sair.
  - **Saída:** A saída esperada seria receber uma afirmação positiva do `assertFalse`, ou seja, é falso que o Client esteja conectado após o comando `QUIT`.
- **testQuit:**
  - **Entrada:** É criado e inicializado um Client, que logo envia um comando para sair.
  - **Saída:** A saída esperada seria receber uma afirmação positiva do `assertFalse`, ou seja, é falso que o Client esteja conectado após o comando `QUIT`.
- **testNick_Invalid:**
  - **Entrada:** Cliente envia `NICK` (comando de terminal) com os seguintes parâmetros, todos inválidos:

    ```
    aa
    3444
    abc%
    ```
  - **Saída:** Espera-se que o *nickname* do cliente não mude em nenhum dos casos. Ou seja, que `client.getNickname()` seja `null` (valor para *nickname* indefinido) em todos os casos testados.
- **testNick_Valid:**
  - **Entrada:** Cliente envia `NICK` com o seguinte parâmetro: `niceNick`, um parâmetro válido.
  - **Saída:** Esperamos que o Cliente de fato recebe e guarde a *string* `niceNick` como seu *nickname*, pois ela é válida.

- **testHost_Invalid:**
  - **Entrada:** Um Cliente vira *host* de uma partida. Temos três situações disponíveis:
    1. Cliente sem *nickname* usando um mapa que não existe. No caso, os parâmetros são: `null` (*nickname*) e "`thisIsFake`" (nome do mapa).
    2. Cliente sem *nickname* usando um mapa válido. No caso, os parâmetros são: `null` e "`simpleMap`".
    3. Cliente com *nickname* válido usando um mapa inválido. No caso, os parâmetros são: "`hostInvalid`" (um *nickname* válido) e "`thisIsFake`".
  - **Saída:** Espera-se que os testes nos digam que:

    Em 1, *nickname* e nome de mapa de fato não valem.

    Em 2, *nickname* não vale, mas nome do mapa vale.

    Em 3, *nickname* vale, mas nome do mapa não.

- **testHost_Valid:**
  - **Entrada:** Cliente com *nickname* válido vira *host* de uma partida usando um mapa válido. Após um tempo, ele muda para um outro mapa válido. Os parâmetros usados são: "`hostValid`" (*nickname* do *host*), "`simpleMap`" e "`happyPlains`" (nome dos mapas válidos).
  - **Saída:** Espera-se que o Cliente de fato sirva de *host* para os mapas escolhidos, mesmo após a mudança.

- **testUnhost_Invalid:**
  - **Entrada:** Há dois casos:
    1. Cliente com nickname inválido e sem ser um *host* (não deu o comando ou não escolheu um mapa válido) tenta dar `UNHOST`. Neste caso, os parâmetros são nulos.
    2. Cliente com nickname válido e sem ser um *host* tenta dar `UNHOST`. Neste caso, os parâmetros acabam sendo nulos, já que não há sala.
  - **Saída:** Espera-se que, por o Cliente não ser *host*, o comando `UNHOST` não funcione.

- **testeHost_Valid:**
  - **Entrada:** Cliente com nickname válido começa a ser *host* de uma sala usando um mapa válido. Após um tempo, ele dá o comando *unhost*. Os parâmetros são: "`unhostVal`" (nickname) e "`testmap`".
  - **Saída:** Espera-se que o Cliente possa se tornar *host* e cancelar o comando sem problemas.

- **testJoin_Valid:**
  - **Entrada:** Dois Cliente são conectados ao servidor com nicknames válidos. Um será o *host* e o outro o *guest*. O *host* cria uma sala usando um mapa válido e o *guest* entra nela. Os parâmetros são: "`hostJoin`" e "`guestJoin`" (nomes do *host* e do *guest*) e "`testmap`" (nome do mapa). Em seguida, eles desconectam.
  - **Saída:** Espera-se que o servidor reconheça que os dois Clientes estão em jogo antes de eles se desconectarem.

A partir daqui, temos os testes para partidas em si. Usaram-se mapas especialmente preparados para testes, *testmap* (para quase todos os casos) e *testmatch* (para testar o fim da partida), além de uma unidade de debug chamada *Master*.

- **testSurrender:**
  - **Entrada:** Dois Clientes, um *host* e um *guest*, se conectam ao servidor e uma sala é criada. O jogo começa e o *host* (primeiro a jogar) dá o comando `SURRENDER` para desistir da partida. Logo após os dois se desconectam. Os parâmetros são "`hostSur`" e "`guestSur`" como nicknames do *host* e do *guest* e "`testmap`" como mapa.
  - **Saída:** Espera-se que os dois Clientes sejam reconhecidos como "`isPlaying`" até o momento em que o *host* dá o comando `SURRENDER`. Na averiguação subsequente, espera-se que nem o *host* nem o *guest* sejam reconhecidos como "`isPlaying`".

- **testEndTurn:**
  - **Entrada:** Um Cliente é criado e vira um *host*. É verificado o turno (-1). Então um *guest* entra na sala e o jogo começa. É verificado o turno (1-turno do *host*). O *host* dá o comando endTurn. É verificado o turno (2-turno do *guest*). O *guest* dá o comando endTurn. É verificado o turno (1-turno do *host*). Os parâmetros são "`hostEnd`", "`guestEnd`" e "`testmap`". Além dos já mencionados -1, 1 e 2 para as comparações de turno.

  - **Saída:** Espera-se que de fato os turnos mudem da forma descrita acima.

- **testMove_Valid:**
  - **Entrada:** Um *host* e um *guest* estão em jogo. O *host* manda a unidade na localização 0 0 se mover para a localização 1 0. Após uma pausa ele tenta movê-la de novo.
  - **Saída:** Espera-se que de fato o *host* consiga mover a unidade da primeira vez, mas que ele falhe na segunda tentativa.

- **testMove_Invalid:**
  - **Entrada:** Durante uma partida, um *host* tenta mover uma unidade usando diversos comandos inválidos, seja tentando mover uma unidade em uma posição inválida, seja tentando colocar uma unidade em uma posição inválida, ou partindo de uma posição inválida para tentar posicionar em outra posição inválida. Também é testado posicionar uma unidade em uma tile ocupada, ou posicionar em um lugar que extrapole o movimento da unidade selecionada, assim como tentar mover uma unidade inimiga.
  - **Saída:** Espera-se que nenhum dos comandos testados sejam aceitos, já que são todos inválidos.

- **testAttack_Valid:**
  - **Entrada:** Durante uma partida, o *host* comanda uma unidade para atacar um inimigo. É dada uma pausa e o *host* tenta mover e atacar com a mesma unidade novamente. Os parâmetros são "`hostAtkVl`", "`guestAtkVl`" e "`testmap`".
  - **Saída:** Espera-se que a unidade selecionada ataque com sucesso na primeira vez. Em seguida, é esperado que o *host* falha em tentar movimentar ela e atacar com ela.

- **testAttack_Invalid:**
  - **Entrada:** Durante uma partida, o *host* tenta atacar. São testadas diversas coordenadas inválidas, tiles vazias como sendo alvo, atacar uma unidade além do range da unidade sendo usada, atacar usando uma unidade inimiga e atacar uma unidade aliada. Os parâmetros são "`hostAtkIn`", "`guestAtkIn`" e "`testmap`".
  - **Saída:** Espera-se que nenhuma das entradas testadas seja bem-sucedida, já que são inválidas.

- **testUnitsNextTurn:**
  - **Entrada:** Durante uma partida, um *host* movimenta uma unidade e ataca uma unidade inimiga, encerrando seu turno em seguida. O *guest* então encerra seu turno, passando a vez para o *host*, que tenta mover uma unidade e atacar usando ela. Os parâmetros são "`hostCtrl`", "`guestCtrl`" e "`testmap`".
  - **Saída:** Espera-se que o *host* possa usar suas unidades normalmente após o *guest* encerrar seu turno.

- **testMatch:**
  - **Entrada:** Durante uma partida, o *host* destrói todas as unidades adversárias, efetivamente vencendo o oponente.
  - **Saída:** Espera-se que tanto o *host* quanto o *guest* automaticamente saiam da partida e voltem para o menu principal, onde podem interagir com o servidor e procurar novos oponentes. Curiosamente, o programa passa neste teste, mas notamos que os jogadores ficam travados na partida após o ataque final quando experimentamos com o projeto.

2 Teste de classes

Teste de Classes

A classe Map (em kindred/client/model/Map.java) foi escolhida para o teste de classes. Os 4 métodos escolhidos, com a aplicação da técnica estrutural de teste “todos os comandos”, encontram-se abaixo. Os métodos em si não foram colocados aqui por questões de estética neste relatório; eles podem ser encontrados no Map.java. Além disso, para maiores detalhes sobre o funcionamento de cada método, consulte os Javadocs disponíveis na pasta *doc/*.

1) validPosition

Para exercitar todos os comandos possíveis, podemos usar as seguintes entradas (com suas saídas esperadas) abaixo.

x	y	Saída
`-1`	`0`	`false`
`getMapHeight() + 1`	`0`	`false`
`getMapHeight() - 1`	`-1`	`false`
`getMapHeight() - 1`	`getMapWidth() - 1`	`true`

2) placeUnit

Para exercitar todos os comandos possíveis através dos atributos do método, podemos usar as seguintes entradas (com suas saídas esperadas) abaixo. Note que, dentro do primeiro **if**, temos:

```
tiles[x][y].getUnit() != null
```

Esta última condição não pode ser controlada apenas pelos parâmetros `unit`, `x` e `y` do método. Além disso, o método `getTeam()` de Unit também cria diferentes caminhos possíveis.

Portanto, para exercitar todos os caminhos, precisa-se externamente manipular a mencionada parte.

unit	x	y	Condições adicionais	Saída
null	0	0		false
!= null	-1	-1		false
!= null	0	0	getUnit() != null	false
!= null	0	0	getUnit() == null; getTeam() == 1	true
!= null	0	0	getUnit() == null; getTeam() == 2	true

3) move

Os comandos possíveis através dos atributos do método, podemos usar as seguintes entradas (com suas saídas esperadas) abaixo. Novamente, não é possível exercitar todos os caminhos apenas com os parâmetros do método devido a `getUnit()` e ao uso da variável local `move`, ligada a `getMove()` e `getMovePenalty()`.

xi	yi	xf	yf	Condições adicionais	Saída
-1	0	0	1		false
0	0	0	-1		false
0	0	1	1	Unidade no tile (xf, yf) não é null	false
0	0	1	1	Unidade no tile (xf, yf) não é null; unidade no tile (xi, yi) é null	false
0	0	0	0	Unidade no tile (xf, yf) não é null; unidade no tile (xi, yi) é null; move ≤ 0;	false
0	0	0	0	Unidade no tile (xf, yf) não é null; unidade no tile (xi, yi) é null; move > 0;	false
0	0	2	2	Unidade no tile (xf, yf) não é null; unidade no tile (xi, yi) é null; move ≤ 0; dx + dy > move	false
0	0	2	2	Unidade no tile (xf, yf) não é null; unidade no tile (xi, yi) é null; move > 0; dx + dy > move	false
0	0	2	2	Unidade no tile (xf, yf) não é null; unidade no tile (xi, yi) é null; move ≤ 0; dx + dy ≤ move	false
0	0	2	2	Unidade no tile (xf, yf) não é null; unidade no tile (xi, yi) é null; move > 0; dx + dy ≤ move	true

4) causeDamage

Os comandos possíveis através dos atributos do método, podemos usar as seguintes entradas (com suas saídas esperadas) abaixo. Não é possível exercitar todos os caminhos apenas com os parâmetros do método; alguns caminhos dependem de `getCurrentHp()` e `getTeam()`, ambos pertencentes à classe Unit.

x	y	damage	Condições adicionais	Saída
-1	0	4		false
0	0	3	Unidade no tile (x, y) é null	false
0	0	5	Unidade no tile (xf, yf) não é null; isDead = false; unit.getTeam() == 1	true
0	0	6	Unidade no tile (xf, yf) não é null; isDead = false; unit.getTeam() != 1	true
0	0	4	Unidade no tile (xf, yf) não é null; isDead = true; unit.getTeam() == 1	true
0	0	5	Unidade no tile (xf, yf) não é null; isDead = true; unit.getTeam() != 1	true

3 Erros encontrados nos testes

Tivemos alguns problemas para iniciar os testes. A parte que gerencia as trocas entre cliente e servidor não estava implementada de uma maneira que permitisse a produção de testes de uma maneira intuitiva e simples. Tivemos grandes dificuldades inicialmente, pois não sabíamos exatamente como proceder. Após um tempo, tivemos a ideia de fazer uma refatoração do código usando o método *test-first*. Primeiro pensamos nos testes a serem feitos e então passamos a alterar o código com isso em mente. Com isso, foi muito mais simples retrabalhar o código e gerar os testes. A maioria foi feita sem grandes dificuldades, excetuando os casos abaixo:

1. Comando `QUIT` fazia com que todos os testes parassem. Foi devido a comandos `System.exit()` no cliente.

   SOLUÇÃO: Retirada de uns `System.exit()` no cliente, deixando-o mais limpo.

2. Comando `NICK`, sendo usado simultaneamente por 2 clientes, fazia com que o segundo tivesse um erro de *nickname* já definido. Isso ocorria devido à estrutura *static* nas `ClientToServerMessages`.

   SOLUÇÃO: Refatoração nas mensagens cliente-servidor.

3. Fim de Partida: Este é um caso à parte. Todos os testes feitos foram aprovados, no entanto, ao se jogar o jogo, vemos que ele não tem fim. Quando todas as unidades de um dos lados são destruídas, o laço do jogo não se encerra, mesmo que seja detectado que não há mais unidades de um dos Clientes. Ainda não foi encontrada uma solução para este problema.

Por último, gostaríamos de dizer que tentamos, mas não conseguimos usar o build.xml para executar os testes devido a problemas de inclusão do *JUnit* no Classpath do *ant*. Como alternativa, usamos as facilidades do Eclipse para executar os testes.

4 Análise comparativa

Infelizmente, devido aos problemas apresentados no relatório de erros encontrados durante os testes, a grande maioria dos diagramas que havíamos feito se tornaram ultrapassados. No entanto, como refatoramos o código pensando nos testes a serem feitos, acabamos tendo o trabalho simplificado na hora de gerar os testes.

Se não houvesse a necessidade de refatorar o código, não há dúvidas de que os diagramas que já havíamos gerado serviriam de base para boa parte dos testes, e esses diagramas nos ajudariam muito no decorrer do desenvolvimento de tais testes. Fazer testes para *use-cases* em que não há diagramas é algo mais complicado, visto que é preciso ter uma ótima ideia do funcionamento da implementação a ser testada.

Em suma, não podemos fazer uma comparação entre a geração de testes para os quais haviam diagramas e para os quais não haviam, no entanto é do entendimento do grupo que os diagramas com certeza facilitariam o trabalho.

5 Problemas no design e na implementação

Como abordado nos demais relatórios, o grupo precisou refatorar toda a parte do código responsável por gerenciar as trocas entre cliente e servidor, pois fazer testes com o código implementado da maneira que estava antes era algo extremamente complicado. Sendo assim, o grupo agiu para se adequar aos requisitos da Entrega 3.

Tal refatoração basicamente anulou boa parte dos diagramas de design que havíamos feito, embora, a um alto nível, alguns ainda pudessem ser utilizados. O jogo em si era funcional, mas a necessidade de implementar testes falou mais alto.

Tendo em vista que há uma seção específica no enunciado para abordar esse assunto, é claro que algo assim era esperado, o que, sinceramente, não é surpreendente. Foi a primeira vez que o grupo implementou testes automatizados, e o programa começou a ser feito antes de nos aprofundarmos nesse assunto em questão durante as aulas. O grupo todo sabe da importância dos testes e que escrever um programa pensando nos testes a serem feitos facilita o trabalho em diversos sentidos. É uma lição que não será esquecida.