Testinator client bot
Design overview

Introduction
This app solves the following programming challenge:

http://testinator-project.appspot.com/

Basically it is a client (bot) that talks to the server. Server asks simple questions in natural language and client attempts to answer these questions.

Requirements
This is a list of requirements taken as a base for the implementation.

R01	**Game rules** • Bot initializes the session. • Then bot asks questions in a loop. • For every question bots calculates and sends the answer. • Server validates the answer. • Negative validation implies end of the game. • Positive validation implies that the game can continue. • After several rounds server signals game over by returning “The game finished” as a next question.
R02	**Client-Server protocol** This is request-response API. All requests are invoked by issuing HTTP GET to specific URL. All responses are expressed as text in the body of HTTP response.
R03	**Server API** **As methods:** • **sessionInit(userAlias: String): String** • **askNextQuestion(): String** • **sendResponse(text: String): Boolean** **As mapped to the protocol:** • **session init:** GET http://host:port/startTest/userAlias - returns sessionToken • **ask next question:** GET http://host:port/sessiontoken/nextQuestion - returns question text • **send response:** GET http://host:port/token/answer - returns “pass” or “fail”
R04	**Bot operation** After starting bot plays one game and exits.
R05	**Configuration of the connection** Server address, server port and user alias are configurable.
R06	**Max number of rounds** Maximal number of question-answer rounds of the game is configurable. After reaching this value bot stops the game and exits.
R07	**Communication problems** On encountering any communication problem with the server bot stops the game and exits.
R08	**Questions format validation** Questions have format: “What is NNN OPERAND MMM?” .. where NNN, MMM are Int values (possibly negative) and OPERAND is one of: “plus”, “minus”, “multiplied by” Bot strictly validates the format of question. If the question is malformed, it stops the game and exits.
R09	**Answer format**

	Answer is just integer value (as string).
R10	**Bot implementation extensibility** Implementation of the bot should be prepared to the following future extensions: • adding more operands in questions • adding more types of questions • playing concurrently several games (possibly against multiple servers) • plugging another communication protocol • storing the history of each game played, with detailed steps

Implementation – technology stack

Implementation language: Scala 2.10.4
Build: SBT 0.13.5
JVM: 1.7.0_60
Logging: slf4j + logback-classic
Dependency injection: manual
HTTP client: naive-http-client (https://github.com/timt/naive-http-server)

Implementation

Diagrams

There are 4 diagrams attached to this document:

- **components-structure:** shows high-level components and dependencies (visibility) between them
- **implementation-structure:** shows main classes of the implementation; interface-implementation separation is ignored for clarity
- **data-model:** shows structure of classes that are "data"
- **game-state-machine:** game finite state machine layout (as implemented by the bot)

Application startup

Startup class is: **heuriqo.hsbctest.Startup**.
Command-line parameters are not needed, because currently the configuration is hadrcoded (see below).

Startup sequence just initializes singletons pool (=manual dependency injection) and asks the bot instance (singleton) to play a new game.

Config

Configuration is per-game. In this version configuration is hardcoded in class: HardcodedGameConfig.

Client

TestinatorClient encapsulates the client-server connectivity. It works as a factory of sessions (TestinatorSession).

TestinatorSession offers API to ask questions and send answers.

Questions parser

Questions parser encapsulates the structure of questions. For simplicity it currently keeps a collection of regexp patterns to parse questions.

There is a hierarchy of classes below GameQuestion trait. This hierarchy is encoding different types of questions and at the same time serves as a place where answer generating logic is placed.

Questions parser returns an instance of (concrete subclass of) GameQuestion.

Bot

Bot is conceptually a factory of games. Given a GameConfig, bot creates a new Game and runs it (currently this is hidden, i.e. Game instances are not visible at Bot API).

A Game is where the state of the game being played is kept. Internally this is finite state machine. A custom implementation of synchronous finite state machine is used.

Game.runTheGame() method is where the main loop of running transitions of this state machine happens. States and Events are implemented as nested class hierarchies. States have “isFinal” flag, each state having isFinal=true is terminal (the FSM processing stops there).

States are implemented as singleton objects. Events are used as a medium of transferring data between states.

The state machine operates as follows:

- it starts in NewGame and the fists event is GameStarted(host, port, userAlias)
- while processing an event, each state selects state to go next and construct next event instance
- the states processing loop continues as long as the current state is not final

This is slightly different compared to the “classic” operation of FSMs, because we do not have external events here.