Inspired Gaming (UK) Limited

Programming Exercise: Sic Bo

Contents

Abstract

‘Sic Bo’ (骰寶) is a casino dice game popular in Asia.
This exercise involves writing a program which simulates a simplified version of Sic Bo.

Rules of Sic Bo

These rules describe the simplified version of Sic Bo which you will implement.

A round of Sic Bo consists of three distinct phases:

1. players are invited to place bets on a *selection*, until
2. the dealer rolls three dice[1] and reveals their values (the *result*), after which
3. the players’ bets are *settled* by the dealer (they become either winners or losers).

The mechanics of the game are as follows:

- Players may place bets on either *big* (大), or *small* (小).
- If the result is that all three dice show the same value, this is called a *triple*.
- Any bet placed on *big* will win if the dice:
  - show a total value of 11 or more, and
  - not a *triple*.
- Any bet placed on *small* will win if the dice:
  - show a total value of 10 or fewer, and
  - not a *triple*.
- If the result is a *triple*, no bets will win.
- The amount of money which a player bets in phase 1 is called his *stake*.
- The amount of money given back to a player in phase 3 is called his *prize*.
- If a player’s bet loses, the value of his *prize* will be zero.
- If a player’s bet wins, the value of his *prize* will be double that of his *stake*[2].

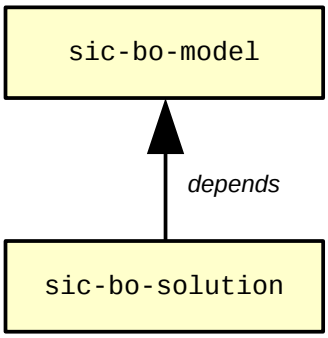
1 A video showing an electronic Sic Bo dice shaker can be seen here.
2 This is known as “even money”.

Environment

In addition to this document, you have been supplied with two Maven modules.

`sic-bo-model`

This module contains the interfaces etc. described in this document which you will require to complete the exercise.



Do not modify any code in this module, simply install it into your local Maven repository in the usual fashion.

`sic-bo-solution`

This module is where you will write your solution to the exercise.

The POM already includes a compile-scope dependency upon `sic-bo-model`.

A skeletal class `SicBo` is included which is the starting point for your solution.

You should:

- modify the POM of `sic-bo-solution` to include:
  - your name and email address where indicated,
  - any dependencies on libraries from Maven Central you require,
- add your name as the `@author` of the `SicBo` class,
- complete the `SicBo` class to satisfy the requirements in this document,
- keep any additional types you create in the same package (or subpackages).

Any open source libraries you use must be suitably licensed for commercial use[3].

Submission

Having completed the exercise, your solution should be submitted as follows:

- Run `mvn clean` on the project (we do not want to receive `.class` files).
- Create a `.zip` file containing the `sic-bo-solution` directory.
- Name the file `sic-bo-solution.zip` .
- Email the file to ikernel.team@ingg.com with the subject "`Sic Bo Solution`".

3 Permissive licenses are acceptable (e.g. BSD, Apache), whereas GPL / LGPL etc. are not.

Requirements

General

Write production-quality code, assuming a Java 7 target.

There is no requirement to:

- write log files,
- write anything directly to `stdout` or `stderr`[4],
- make your application configurable, or
- to localize your application.

We are interested in:

- the thread-safety of your application,
- the clarity of your code, and
- your adherence to the instructions in this document.

Each instance of the `SicBo` class represents an independent Sic Bo casino table.

To complete the `SicBo` class you will realize these interfaces:

Interface	Purpose
`Table`	• controls the lifecycle of each table • accepts bets from players
`BetFuture`	• allows prizes to be delivered back to players

The interfaces are supplied with JavaDoc which complements this document.
After you submit your solution to Inspired Gaming (UK) Limited, we will:

- create (many) instances of your `SicBo` class using the existing constructor[5],
- invoke methods via the `Table` and `BetFuture` interfaces to verify your code.

You will have no control over the threads we use to invoke the methods on the interfaces.

4 Although, as discussed later, an implementation of `ResultDisplay` may write to `stdout`.
5 See the `Main` class in the `sic-bo-solution` module for an example of this.

`Table` interface

The `SicBo` class must realize the `Table` interface.

The phrase “each table” should be interpreted as “each separate instance of `SicBo`”.

```
public interface Table {

    void open();

    void close();

    BetFuture acceptBet( Selection selection, Integer stake )
                throws TableClosedException;

}
```

Each table can be opened, and closed.

Players are modelled as clients of the `acceptBet` method.

`open` will be called:

- once in the lifetime of each table.

`close` will be called:

- once in the lifetime of each table,
- at an arbitrary time after `open` has returned.

`acceptBet` will be called:

- at any time in the lifetime of each table,
- from multiple threads concurrently.

Before `open` is invoked, no bets can be accepted by that table.

Once `open` has been invoked, the table should begin accepting bets from players and executing rounds of Sic Bo continuously, until `close` is invoked.

Once `close` is invoked, you should complete the current (i.e. final) round of Sic Bo, then no further rounds should execute on that table. No bets can be accepted by a table after the final round of Sic Bo has completed.

Each round of Sic Bo should have the following structure (maintain this sequence):

- generate a globally-unique `String` which identifies the round,
- accept bets for a period of approximately 5 seconds,
- generate a result,
- display the result (see later),
- use the result to settle all the bets which were accepted, and finally
- deliver the prizes to the players.

Each player is represented as a thread which might concurrently invoke `acceptBet`.

The number of players on each table will vary between 0 and 64.

A player (i.e. thread) may place multiple (possibly conflicting) bets is a single round.

Each player will place bets at a maximum frequency of 1 per second.

There is no requirement to identify the players, or to track their balance over time.

The parameters of `acceptBet` completely describe a bet from the player's perspective.

If a bet is accepted, you should return an instance of `BetFuture` to the client (see later).

If `acceptBet` is invoked, but the table is not open (i.e. bets are not being accepted), you should throw `TableClosedException`.

If `acceptBet` is invoked, but the 5 second period for accepting bets in the current round has elapsed (e.g. a result is being generated), the bet should be accepted for the *subsequent* round, if any[6]. The `acceptBet` method should block in this circumstance.

When `close` is invoked, you *may* wish to prematurely curtail the 5 second period for accepting bets and conclude the final round early, *or* you could allow the 5 second period to complete naturally. Either solution is acceptable.

Result Generation

You must simulate the roll of three six-sided dice to generate a result.

The dice are independent from one another; each round of Sic Bo on a table is independent; each table is independent.

`ResultDisplay` interface

Each instance of `SicBo` will hold a reference to a `ResultDisplay`.

```
public interface ResultDisplay {

    void displayResult( String roundId, Iterable<Integer> result );

}
```

Once a result is generated, you must pass it to the the `ResultDisplay`, along with the unique identifier for that round.

A simple implementation is included in `sic-bo-model` which writes the result to `stdout`.

You are not required to implement the `ResultDisplay` interface.

6 If there is no subsequent round, you should throw `TableClosedException`.

`BetFuture` interface

Once bets are accepted from clients of the `Dealer` interface (the players), you will return instances of `BetFuture` to them.

```
public interface BetFuture {

    String getRoundId();

    Integer getPrize() throws InterruptedException;

}
```

The client will invoke `getRoundId` to obtain the unique identifier for the round which has concluded.

The client will invoke `getPrize`, which should block until the bet is settled, and the prize value can be returned to the client.

The client may invoke `getPrize` immediately the bet is accepted, or arbitrarily later.

Once the prize value is delivered, all state associated with the bet may be discarded.

A losing bet has a prize value of zero.

License

7 We explicitly forbid use of our code for your own commercial purposes.