UNIVERSITY OF TÜBINGEN

BACHELOR THESIS

Functional Universe: A Gaming Client in Racket for First-Year Students

Author:
Louisa LAMBRECHT

Reviewer:
Prof. Dr. Torsten GRUST

Supervisor:
Daniel O'Grady

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Database Systems Research Group
Department of Computer Science

September 24, 2018

Declaration of Authorship

I, Louisa LAMBRECHT, declare that this thesis titled, “Functional Universe: A Gaming Client in Racket for First-Year Students” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UNIVERSITY OF TÜBINGEN

Abstract

Faculty of Science

Department of Computer Science

Bachelor of Science

Functional Universe: A Gaming Client in Racket for First-Year Students

by Louisa LAMBRECHT

In the era of technology smartphones and computers play a significant role in daily life. More and more computer scientists are requested to fill the need of advancing technology. For many students games on the phone or computer are a part of their leisure time. Therefore games in general but multi-player games especially are an opportunity of creating strong motivation among first-year students to learn programming and design concepts. This thesis explores the possibility of providing a game specific frame geared to the concept of big-bang. This frame allows the students to program a simple Racket client without being distracted by data-interchange formats or transmission protocols. A model is presented to develop an interlayer corresponding to the game. This interlayer provides an interface to enable the students to program their own game client suitable for their skills. Specifically the messages communicated by the server are preprocessed and trigger game specific events to not overburden first-year students with data-interchange formats in addition to distributed programming. The model also indicates an abstraction to easily make use of different games and servers.

Contents

Declaration of Authorship **iii**

Abstract **v**

1 Introduction **1**
1.1 Pedagogical Value of Multi-Player Games 1
1.2 Why Racket? 2
1.2.1 Racket - A Lisp-based Language 2
1.2.2 Why Racket? 2
1.3 Aim of this Thesis 3

2 Multi-Player Games in Racket **5**
2.1 The *Universe* Teachpack 5
2.2 Worlds 5
2.2.1 Simulations 6
2.2.2 Interactive Games 6
2.2.3 An Interactive Example 7
2.3 A Universe Represents a Server 8
2.3.1 The Server 8
2.3.2 Role of the Server 9
2.3.3 The Client 9
2.3.4 Communication 9
2.3.5 A Distributed Example 10
2.4 How About a Different Server? 12
2.5 The *Dragonland* Server 12
2.5.1 Description of the Game *Dragonland* 12
2.5.2 Role of the *Dragonland* Server 13
2.5.3 Communication of the *Dragonland* Server 13

3 Development of a Client **15**
3.1 Idea 15
3.2 Connecting to the Server 15
3.3 Steps of Programming 17
3.3.1 Parsing JSON 17
3.3.2 Analysing JSON 18
3.3.3 Processing Data 18
3.4 Workflow of the Client 18

4 Abstraction to a General Model **21**
4.1 Turning the Idea into a Model 21
4.1.1 Model of the Communication 21
4.1.2 Model of the Logical Units 22
4.2 Student Client: The Template 23

4.3 JSON Module 24
4.3.1 Documentation of the Functions 25
4.4 Game Client 26
4.5 Testing the Model 27
4.5.1 The Client from a Student's Point of View 27

5 Discussion and Conclusion **31**
5.1 The Task 31
5.2 Summary of the Results 31
5.3 Discussion: Changing the Game 32
5.4 Concluding Remarks 32

Bibliography **35**

List of Figures

2.1 Workflow of the worlds in a simulation 6
2.2 Workflow of the worlds in an interactive game 7
2.3 Communication between a Racket server and a Racket client 10
2.4 Screenshot: a simple game where two players can move around 12

3.1 Workflow of the client 19

4.1 Communication between server, game client and student client 21
4.2 Model of the logical units 22
4.3 Screenshot: the implementation of a *Dragonland* client in Racket 28

5.1 Model of the logical units with interchangeable parts 33

List of Listings

2.1 A typical big-bang function call using mouse, key and tick events. . . 7
2.2 on-key: changing the position according to the key pressed 8
2.3 to-draw: rendering the pawn at its correct position 8
2.4 big-bang: a pawn moves on an empty canvas 9
2.5 A typical universe function call with connection, message and tick events 9
2.6 big-bang as a client 10
2.7 on-new: this server accepts only two clients 11
2.8 on-msg: passing through the messages of the clients 11
2.9 universe: the server connects two players 11
2.10 An example of a JSON object sent by the *Dragonland* server 13
3.1 Simple TCP client to test the connection 16
3.2 Source code of the *universe* teachpack: establishing the connection . . . 16
3.3 Java method to send the acknowledging message 17
3.4 The simplest big-bang client 17
4.1 The ideal set-up of a game-specific big-bang function 23
4.2 Optional keyword arguments offer high resemblance to the ideal . . . 24
4.3 The general set-up of the game client 26
4.4 Analysing the server input and triggering events 27

List of Abbreviations

HtDP	**H**ow **t**o **D**esign **P**rograms
DMdA	**D**ie **M**acht **d**er **A**bstraktion
GUI	**G**raphical **U**ser **I**nterface
TCP	**T**ransmission **C**ontrol **P**rotocol
JSON	**J**avaScript **O**bject **N**otation

Chapter 1

Introduction

Daily life cannot be imagined without computers, smartphones or other technological devices any more. But developing these technologies needs good programmers. The key to becoming a good programmer is good education. Good education does not only become apparent at its (formal) end when a student leaves university with a bachelor or master degree. Good education is visible during the process by the students' motivation to visit lectures and exercise classes and to invest time on their own. A vague interest in a general topic does not carry a student through university studies. It takes initiative, willingness to learn and hard work that can be triggered and supported by challenging motivation. Motivation is a key to learning success.

1.1 Pedagogical Value of Multi-Player Games

Triggering motivation is said easily but not always done so. How to motivate students to engage with abstract concepts of programming? A real reference point is always a good starting point; something everybody knows and can connect to. Computer games pose such a point. Nowadays every student will have had some contact with video games: either with a game console, a gaming app on the smartphone or a good (old) computer game like Tetris or DotA.[1]

The best way to encourage students to learn is to use their leisure activities as a reference point. The combination of gaming (entertainment) and the possibility of teammates from all over the world (networking) form a unique motivation for students: not only learning with games but also learning through games (by programming games themselves).

A beginners course in computer science ought to be exciting. It is the corner stone of the course itself and maybe the rest of the studies. Several studies[2] showed that programming games is a successfull tool to teach students program design principles and create exciting motivation[Mor18]. Therefore programming multi-player games in teaching can be considered as having high pedagogical value.

[1] see `https://tetris.com/` and Defense of the Ancients

[2] Courtney et al. (2003) [CNP03]; Achten (2008) [Ach08]; Felleisen et al. (2009) [Fel+09]; Morazán (2011) [Mor11]; Morazán (2015) [Mor15]

1.2 Why Racket?

As multi-player games are a great opportunity to encourage students to learn, it is beneficial to use them in programming education. But why should that be done in combination with Racket? What makes Racket the programming language for distributed game programming?

1.2.1 Racket - A Lisp-based Language

Racket belongs to the Lisp family.[3] Significant characteristics of Lisp are the lambda calculus by Alonzo Church, the concept of everything being a function, and list processing [Fel+13].

In the late 1950s John McCarthy and his researchers developed Lisp. Almost 15 years later in 1972 Guy Steele and Gerald Sussman studied the ideas of Alonzo Church and also object-oriented programming. They concluded that Church's programming language should be extended with assignment statements and jumps in control flow [Fel+13]. They implemented this new language in Lisp and called it Scheme. Soon after Scheme became popular and many experimented with the language. So did a research group at Rice University in Houston, Texas in the 1990s.

Matthias Felleisen, Robby Findler, Matthew Flatt and Shriram Krishnamurthi wanted to develop a language that would not overburden students who just began to study programming [Fel+15]. They aimed to create a simple programming language with a supporting programming environment for beginners. Thus, a new version of Scheme was developed and refined repeatedly until it was renamed to Racket in June 2010 to emphasize the differences between Scheme and this by then fully-fledged programming language.[4]

1.2.2 Why Racket?

The reason why Racket was created in the first place is the reason for using Racket in distributed game programming. Matthias Felleisen, Robby Findler, Matthew Flatt and Shriram Krishnamurthi et al. explicitly state in their *Racket Manifesto* that their intention for Racket was to create a teaching language for beginners [Fel+15]. Further they expressed their wishes in *Realm of Racket* that they wanted to give beginners the chance to learn programming without simple exercises, such as calculating the first 100 primes, but to trigger excitement among the students that programming can be fun [Fel+13].

Racket is not only a programming language with simple keywords that are easy to understand for beginners. It also comes with a programming environment that is very supportive and not overloaded for students who have never programmed before. This environment offers several Lisp dialects and allows the inclusion of different teachpacks to learn step by step.

The *universe* teachpack by Matthias Felleisen contains all the functions needed for programming multi-player games [Fel14]. Furthermore it offers other simpler animation tools to provide a smooth transition from easy animations to more complex

[3] see http://docs.racket-lang.org/guide/dialects.html
[4] see http://racket-lang.org/new-name.html

games and finally to distributed games. The extensive documentation for Racket and the teachpacks is a secure guide that accompanies the students on their journey to become good programmers.

1.3 Aim of this Thesis

The aim of this thesis is to explore the possibility of using Racket multi-player games in first-year teaching. More specifically the potential of connecting a `big-bang` client with a Java server is examined. To make the task of programming a client suitable for first-year students an interlayer should preprocess incoming data. This interlayer should provide an interface to a small part of the client such that the students would not have to care about understanding JSON data or the details of network protocols. The messages from the server should be analyzed and trigger game-specific events similar to the functioning of the `big-bang` clauses. In the end a template should indicate the use of the interface and provide a frame for the students to easily implement their own game client.

The code described in this thesis can be found in the internal git repository "Info1Universe" of the chair of database systems.

Chapter 2

Multi-Player Games in Racket

As mentioned before Racket is a teaching language and since multi-player games have proven to be very encouraging for students, Racket also offers some useful functions to implement your own game. The *universe* teachpack provides powerful animation tools and simple functions to design a game. It is used in combination with the *image* teachpack for graphical user interface (GUI) programming. Both teachpacks were developed in the course of *How to Design Programs*[1] *(HtDP)*, a textbook that introduces systematic design of computer programs in Scheme. The *universe* teachpack is the basis for this thesis. Thus, the following chapter will give a brief introduction into the concept of the *universe* teachpack and its implementation with short examples.

2.1 The *Universe* Teachpack

The concept underlying this teachpack is an analogy of our universe. Each computer or program is represented as a world. A simulation or a normal interactive game that runs on exactly one computer is a world without contact to other worlds. Only the universe (a server) can connect the worlds and provide the facility of interacting with each other.

To guide the students slowly into the concept of the *universe* teachpack, user interaction and GUI programming, there are some straightforward functions to start with before rushing into multi-player game programming. Simulations without interaction are a good opportunity to introduce the *universe* teachpack and the GUI functions of the *image* teachpack. Interactive games like *Breakout*[2] form a good follow-up, as they introduce the function that is also needed for distributed game programming and broaden the knowledge of GUI programming. After programming an interactive game the students will have understood the concept of the universe analogy sufficiently to take a look into the design of multi-player games in Racket.

2.2 Worlds

This section describes what kind of worlds exist in the *universe* teachpack and how to create them. A world consists of at least two features: a world state that represents

[1] see Official Website and Dokumentation
[2] see https://en.wikipedia.org/wiki/Breakout_(video_game)

the world's set-up and a rendering function that produces an image to display the world.

2.2.1 Simulations

A simulation is a simple world whose state is only marked by a number. This number is increased by every clock tick. Thus, the world constantly changes. Every change of the world state invokes the rendering function. It produces only a single scene but the world changes with every clock tick, 28 frames per second form a simulation (see figure 2.1).

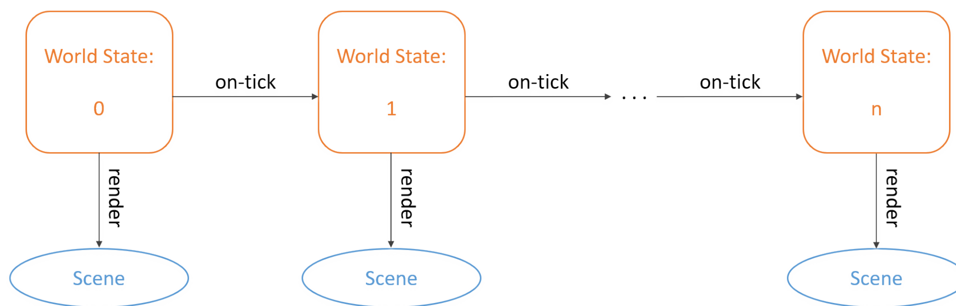

FIGURE 2.1: Workflow of the worlds in a simulation: The world state of a simulation changes on every clock tick. The state is an increasing number. Every change of the world invokes the rendering function that creates a scene out of the world state.

Such simulations can be created by the `animate` tool that takes a single function as an argument. This function is the rendering function that has to create a scene out of a number (the world state). A first result could be a person moving from one edge to the other or a UFO gliding down from the top of the window.

2.2.2 Interactive Games

Simulations are entertaining to watch but lack an activity. Interaction is a new aspect that can transform a simulation into a game. This section describes how interaction is managed by different events.

Interactive games are event based. Apart from the clock tick other events such as a mouse click or a key stroke can occur. The events belong to clauses. A clause accepts a function and sometimes other parameters as well that handle the corresponding event. For example the `on-tick` clause that was integrated in `animate` calls on `tock` whenever the event of a clock tick takes place.

Every event may change the world state and therefore the visualization of the game. These new possibilities of changing the world state yield in the necessity of a more flexible world state that is able to contain complex information. Hence the world state can be basically anything: a number, a list, a structure. That offers the potential of elaborate GUI programming since the rendering function now has access to complex information to turn into a scene. All these features are combined in a function called `big-bang`.

```racket
(big-bang ws
        (to-draw render)
        (on-tick tock)
        (on-mouse click)
        (on-key react))
```

LISTING 2.1: A typical big-bang function call using mouse, key and tick events.

The big-bang function offers many clauses and specifications of these clauses. The four most used clauses - though not necessarily in one program - are displayed in listing 2.1. But these four are most likely to be used - though not necessarily in one program. The rendering clause and the on-tick clause are integrated in animate. All clauses apart from the rendering clause to-draw are optional. The functions that pose the arguments of the clauses have the world state as a given argument and as their outcome. That results in a similar workflow to that of a simulation (see figures 2.1 and 2.2 for comparison).

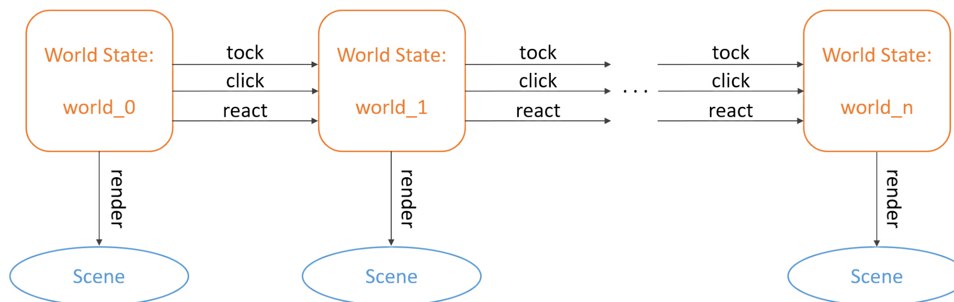

FIGURE 2.2: Workflow of the worlds in an interactive game: Not only clock tick events can change the world state. There are also mouse clicks and key strokes. The world state can be a more complex structure than a number. Still every world state is rendered to a scene.

2.2.3 An Interactive Example

The following example demonstrates the use of the introduced big-bang function. A simple but interactive example (as a preparation for the game that is used in this thesis) could be a pawn moving on a canvas. The world state represents the position of the pawn. The player can change that position with the arrow keys.

The position is stored in a mutable prefab struct. The prefab feature will be of importance when transforming the example into a distributed game. The initial world state using the posn struct could be (posn 100 100). The react function updates the position according to the key pressed as presented in listing 2.2.

Whenever the on-key event occurs with the arrow keys the world state changes. That invokes the rendering function that has to plot the game. Listing 2.3 shows the rendering function that creates a scene by placing the icon of the pawn onto a panel.

In the end the call on big-bang that can be seen in listing 2.4 connects these functions to a simple interactive program. Though the working of the game is never revealed in that function but in the handling of the events.

```racket
(struct posn (x y) #:mutable #:prefab)

(define (react ws key)
  (cond [(key=? key "up") (set-posn-y! ws (- (posn-y ws) 5)) ws]
        [(key=? key "down") (set-posn-y! ws (+ (posn-y ws) 5)) ws]
        [(key=? key "right") (set-posn-x! ws (+ (posn-x ws) 5)) ws]
        [(key=? key "left") (set-posn-x! ws (- (posn-x ws) 5)) ws]
        [else ws]))
```

LISTING 2.2: on-key: changing the position according to the key pressed

```racket
(define (render ws)
  (place-image pawn1 (posn-x ws) (posn-y ws)
               (empty-scene 300 300)))
```

LISTING 2.3: to-draw: rendering the pawn at its correct position

Even though this interactive example is very easy, it is possible to create complex programs with `big-bang`. If a game like *Breakout* is chosen that resembles the students' experience with professional games, they have a reference point that can motivate them to spend time and effort on it.

2.3 A Universe Represents a Server

The *universe* teachpack provides everything needed for creating distributed multi-player games. This section presents the tools for creating a universe server, a `big-bang` client and their communication.

A distributed game usually consists of a server that provides the game and forms the point of intersection. On the other side there are clients. A client accesses the server and thereby takes part in the game. The universe analogy presents the server as the universe and the clients as worlds that are part of the universe. Since a `big-bang` program creates a world, it acts as a client in distributed games.

2.3.1 The Server

The `universe` function works similar to `big-bang`. A universe state represents the current status of the universe and clauses handle the different events that can occur. The most common events are a new connection with a world (`on-new` clause) and receiving messages from a world (`on-msg` clause). Like in `big-bang`, event handling functions consume the state and the new information - e.g. the identity of a new client that first connects - as arguments. In addition there is an extra clause to specify the port that the server listens on. Listing 2.5 shows the connection of the events in the `universe` function.

```
(big-bang (posn 100 100)
  (to-draw render)
  (on-key react))
```

LISTING 2.4: big-bang: a pawn moves on an empty canvas

```
(universe state
        (on-new new-expr)
        (on-msg msg-expr)
        (on-tick tick-expr)
        (port port-expr))
```

LISTING 2.5: A typical universe function call with connection, message and tick events

2.3.2 Role of the Server

The server can engage in different roles:

- it can simply pass through all messages coming in
- the server allows only send-listen-send so that every world can send a message and before sending again has to wait for an answer
- the server can act as an administrator that controls everything and keeps track that the clients only make valid moves.

If the server implements a game where the clients have to follow some rules, the server will need to make sure that nobody is trying to manipulate the game since every student can implement his/her own client. Thus, the server has to act as an administrator to enforce the logic of the game.

If the server offers the facility of e.g. a chatroom it only has to pass through and broadcast the messages to all other clients. But whatever role the server takes, it does not change the set-up of the `universe` function.

2.3.3 The Client

Since `big-bang` acts as a client in a distributed game, some additional events are needed. The set of clauses introduced before is kept and expanded with three other clauses needed for building a connection: register, port and on-receive. They are used to define the IP, port to connect to and how to process incoming server messages (see listing 2.6).

2.3.4 Communication

Enabling communication between the server and a client does not require a lot of changes in the `big-bang` client. Instead of reacting to an event with a new world or universe state, functions can also return a combination of a (new) state and a message. clients can send a “package” with a world state and a message while the server works with bundles containing the universe state, messages to clients and

```racket
(big-bang ws
        (to-draw render)
        (on-key react)
        ...
        (on-receive listen)
        (register IP)
        (port PORT-NUM))
```

LISTING 2.6: `big-bang` as a client

a list of clients which are to be removed from the universe. Figure 2.3 shows this communication graphically.

The messages are sent via TCP therefore only so called S-expressions can be sent. S-expressions are basic data like strings, numbers etc. and lists of S-expressions or prefab structs of S-expressions. Thus, the `posn` struct needs the prefab feature so that it can be sent to the server.

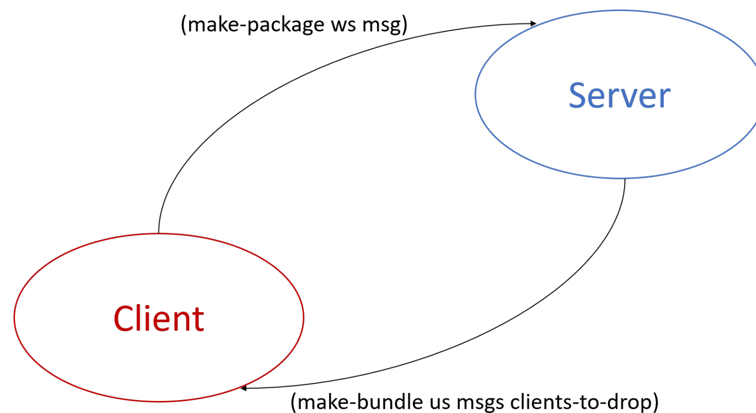

FIGURE 2.3: Communication between a Racket server and a Racket client: The `big-bang` client returns a package with a world state and a message. The world state replaces the current world state of the `big-bang` program. The message is sent to the server. The universe server creates bundles with a new universe state, messages to the clients and a list of clients that are to be disconnected from the server.

2.3.5 A Distributed Example

Extending the example from before will demonstrate a universe server and the use of `big-bang` as a client. The server will implement the facility that two clients can connect and see each other moving.

To use the `big-bang` example from before the program has to be changed slightly. The world state has to be adjusted to save not only the player's own position but also the other's, e.g. in a list. Furthermore whenever a player changes his/her position, not only the world state has to adapt accordingly but the position needs to be sent to the server. That is done by sending a package consisting of the world state and the new position as a message: `(make-package modified-ws new-position)`. The client also needs to listen when the server broadcasts the messages of the other player and

update this position in the world state. In addition the rendering function must be able to display more than just the own position.

The Exemplary Server

The server only passes through the positions of the other client. Therefore the state of the universe is represented by a list of the current clients. In this case the server only accepts two clients to keep it simple. Every new connection either adds the client to the list or disconnects the world from the universe if two clients are already connected (see listing 2.7).

```racket
(define (new-expr us client)
  (if (> (length us) 1)
    ; disconnect if more than two players
    (make-bundle us '() (list client))
    ; add new player if not
    (cons client us)))
```

LISTING 2.7: on-new: this server accepts only two clients

The data representation of a client in a universe program is an *iworld*. Since *iworlds* are only a depiction of a client and not the client itself, *iworlds* don't have a constructor. However, most of the other functions available for structures can be found as well, e.g. `iworld=?` is used as a comparison operator. In listing 2.8 the server forwards the received message to the other party by creating a bundle of the same universe state, a mail with the new position and an empty list because no clients are to be disconnected.

```racket
; forward position to the correct client
(define (msg-expr us from msg)
  (if (and (= (length us) 2) (iworld=? from (first us)))
      (make-bundle us (list (make-mail (second us) msg)) '())
      (make-bundle us (list (make-mail (first us) msg)) '())))
```

LISTING 2.8: on-msg: passing through the messages of the clients

This is basically what is needed for this simple universe. The call on `universe` is composed of these two functions and the port (see listing 2.9). The result of these few lines of code can be seen in figure 2.4. The code for the whole example can be found in the `Universe` folder in the git repository.

```racket
(universe empty
          (on-new new-expr)
          (on-msg msg-expr)
          (port 1234))
```

LISTING 2.9: universe: the server connects two players

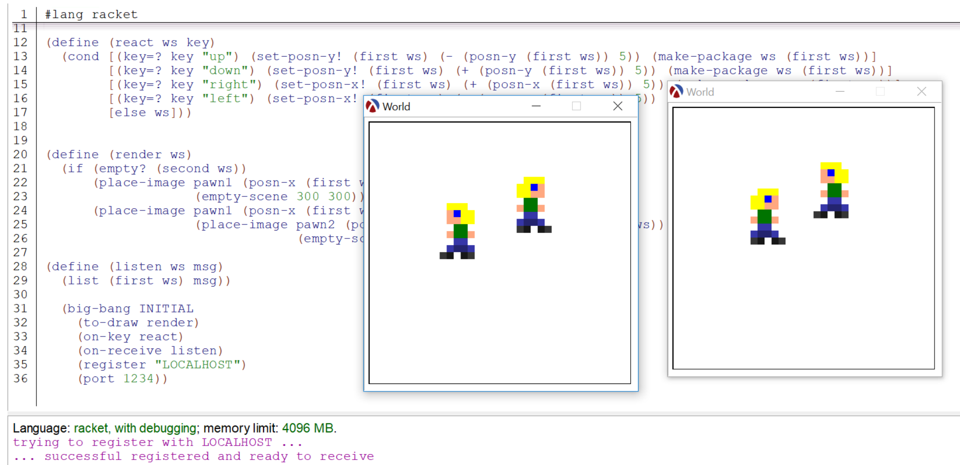

FIGURE 2.4: Screenshot: 60 lines of code can create a simple game where two players can move their pawns

2.4 How About a Different Server?

The sections above were an impression of the realization of multi-player games in Racket. But Racket may not always be the first choice of language to implement a server. One of the tasks for this thesis was to research the possibility of connecting `big-bang` with a non-Racket server; in this case a Java server. The Java server and `big-bang` make use of the Transmission Control Protocol (TCP) that allows language and platform independent communication and therefore should enable a stable connection. The following section gives a full description of the server and the implemented game.

2.5 The *Dragonland* Server

The server that was used for this work implements a game that was originally invented by Roland Mühlenbernd[3] to use it for teaching. For the sake of this thesis it was called *Dragonland*. The implementation in Java[4] was done by Daniel O'Grady[5].

2.5.1 Description of the Game *Dragonland*

Dragonland has a grid structure with different fields. The players can move their pawns on the board. The fields represent various types of terrain e.g. water, forest or paths. Some fields can be walked on some cannot. *Dragonland* is bordered with walls. In some parts of the forest the players can hunt. Now and then a dragon leaves its hiding place and enters the game.

Whenever two players meet they can decide to interact and test their fighting skills in a minigame. There are three minigames available: *Skirmish*, *Staghunt* and *Dragonhunt*. All of them belong to the field of game theory similar to rock, paper, scissors.

[3] see https://www.muehlenbernd.net/
[4] see https://github.com/ogrady/inf3project
[5] see https://db.inf.uni-tuebingen.de/team/DanielO-Grady.html

Dragonhunt can only be played when two players and a dragon meet on the same field. To play *Staghunt* two players have to be on a huntable field together in the forest. *Skirmish* can be fought whenever two players meet.

The players collect points depending on the minigame and the choices made. The player with the highest score wins - though technically there is no official end of the game. In addition all connected players can communicate with each other via a chat function.

2.5.2 Role of the *Dragonland* Server

The server acts as an administrator to make sure that everyone plays by the rules. Whenever a player wants to perform an action, e.g. move on the board from one field to the next, he/she has to send a request to the server. According to the map (whether the field is "walkable") the server either grants or denies the request by sending No or Ok as an answer. If the request is granted the server will also broadcast a message to all participants containing the new position of that player.

2.5.3 Communication of the *Dragonland* Server

The server communicates via TCP and uses the JavaScript Object Notation (JSON) to wrap the relevant information that is sent. JSON is a format to bundle information in a certain structure and access it again easily.

The JSON objects sent by the *Dragonland* server have a unique single key on the first level that identifies the information in the value e.g. like in listing 2.10 a new chat message is always contained in a JSON object that has "mes" as a head key. This enables an easy recognition and dissection of the incoming information.

```json
{"mes":
        {"senderid":0,
         "sender":"player42",
         "text":"Hello!"}
}
```

LISTING 2.10: An example of a JSON object sent by the *Dragonland* server

The communication from a client to the server is specified by a strict command structure that the client has to adopt.[6] All requests for retrieving information from the server are marked by the key word `get`. Inquiries for performing some kind of action like moving or challenging another player start with the key word `ask`. If "player42" wants to send a chat message that results in the JSON object in listing 2.10 he/she needs to send the request `ask:say:Hello!` to the server.

[6] see https://github.com/ogrady/inf3project/wiki

Chapter 3

Development of a Client

The aim of this thesis was to assess whether it is possible to connect a Racket `big-bang` client with the *Dragonland* server and to what extend the server may have to change, furthermore to find out how a future task for first-year students could be developed fitted to the desired game. This chapter describes how to set up a connection and introduces some of the changes to the server.

3.1 Idea

First-year students should benefit from the encouraging effect of multi-player games in teaching. Thus, they should develop their own client to learn and use concepts like higher-order programming etc. To not overburden them with design concepts, distributed programming, network communication and JSON data, an interlayer should be created that partly frees the students from that. That way the task for the students can be focused on the subjects that are important in first-year teaching in programming. The first draft of a client that could act as such an interlayer is described in the following sections.

3.2 Connecting to the Server

In the beginning the development of a client proved to be difficult. The server as well as `big-bang` use TCP but a simple `big-bang` client like in listing 3.4 would not connect to the server. Therefore as an experiment a small TCP client was created to explore the behaviour of the client itself and that of the server (see listing 3.1). Furthermore it created the opportunity to analyse the structure of JSON messages.

There are several functions that can be used to write to an output port which can be narrowed down to:

- `(display datum [out])`
- `(write datum [out])`
- `(print datum [out quote-depth])`

It is also possible to use different reading functions such as

- `(read [in])` and
- `(read-line [in mode])`.

```racket
(define client
  (lambda (port ip)
    (define-values [in out] (tcp-connect ip port))
    (display (read-line in))     ; (diplayln (read in))
    (display "get:map" out)      ; (write "get:map" out)
                                 ; (print "get:map" out)
    (close-output-port out)
    (display (read-line in))
    (close-input-port in)))

(client 1234 "LOCALHOST")
```

LISTING 3.1: Simple TCP client to test the connection

Partially the behaviour of these functions vary a lot which is discussed further in sections 3.3.1 and 4.5.1.

But the notion of these differences in behaviour did not solve the problem of the failed connection. The humble result of trying to connect a big-bang client with the server remained an abandoned connection due to protocol problems.

Taking a look at the source code of the *universe* teachpack revealed the origin of that error. The responsible function is presented in listing 3.2.

```racket
(define (tcp-register in out name)
  (define msg `(REGISTER ((name ,(if name name (gensym 'world))))))
  (tcp-send out msg)
  (define ackn (tcp-receive in))
  (unless (equal? ackn '(OKAY))
    (raise tcp-eof)))
```

LISTING 3.2: Source code of the *universe* teachpack: establishing the connection

When connecting to a (universe) server big-bang sends a message to register by a name with the server. The name can be given with a big-bang clause (name "player42") or generated automatically (line 2 in listing 3.2). The (universe) server then replies to the registration with an acknowledging message. That message has to equal (OKAY) (line 5[1]). Otherwise an end-of-file error is raised that closes the connection.

In the teaching language *Die Macht der Abstraktion (DMdA)* symbols are treated slightly differently which changes the acknowledging message to (|OKAY|). This also works in normal Racket language. The curious part is that the tubes only surround the text and not the complete symbol.

Due to the necessity of the acknowledging message the server has to be changed to send it to the new client before broadcasting any other information about the entrance of that client. This modification was necessary even though it maybe lessens the server's universal usage.

[1] The apostrophe marks a symbol in Racket even though it may look like a list with the parentheses at first sight.

```java
public synchronized void sendRegisterOk() {
    send("(OKAY)"); // or more general: "(|OKAY|)"
}
```

LISTING 3.3: Java method to send the acknowledging message

3.3 Steps of Programming

After a stable connection between server and client could be established the actual development started with the client in listing 3.4.

```racket
(define (analyse ws msg) (print msg))

(big-bang #f
    (to-draw (lambda (ws) (empty-scene 200 200)))
    (on-receive analyse)
    (register "LOCALHOST")
    (port 1234))
```

LISTING 3.4: The simplest `big-bang` client

3.3.1 Parsing JSON

The preprocessing of the data began with the parsing of the JSON data that was received as a message in `analyse` (line 1 of listing 3.4). The first problem occurred here. As mentioned before there are several reading functions to read in data from an input port. The *universe* teachpack makes use of the `read` function that only reads one datum after each other. A datum is anything that can be a single element.[2]

This element is then interpreted by Racket. For a JSON object that means: curly and squared brackets evolve into normal parentheses and commas stand for unquote. Fortunately a JSON object is always just one datum because the surrounding curly brackets are interpreted by Racket as a list. Hence a JSON object is received by Racket as a list of lists.

For example the JSON object
`{"mes": {"senderid": 0, "sender": "player42", "text":"Hello!"}}`
is received as
`'("mes" : ("senderid" : 0 ,"sender" : "player42" ,"text":"Hello!"))`
which is equal to
`(list "mes" ': (list "senderid" ': 0 (list 'unquote "sender") ': "player42" (list 'unquote "text") ': "Hello!"))`.

The behaviour of the `read` function can be further explained with the following example. If the server were to send the message "Hello player42!", `read` would be called twice to first read *Hello* as a symbol and secondly *player42!* as another symbol. Strings are only read as one string by `read` if surrounded by quotation marks. Thus,

[2] see `https://docs.racket-lang.org/reference/reader.html#%28tech._datum%29`

surrounding every message to be sent with quotation marks is a sensible modification of the server. Otherwise the resulting list has to be parsed into a string containing a valid JSON object first before being able to make use of the JSON library. As the task required to change the server as little as possible such a function was developed. It is presented in detail in section 4.3.

3.3.2 Analysing JSON

In the next step the string containing the JSON object could be converted into a JSON hash by using Racket's JSON library.[3] Then the hash had to be analysed. A hash can be simply searched with a loop to find key-value pairs. The structure of the JSON objects lead to a few functions that allow dissecting the hash easily. In particular the feature of having one head key is of importance. This feature is used to analyse and identify the JSON object correctly to be able to act accordingly.

3.3.3 Processing Data

Based on the type of information, e.g. the update of a player or the transmission of a chat message, various functions can be called to process the data further. For that matter functions were developed that offer access to hashs easily and extract relevant data. This data constitutes the arguments of the functions that actually deal with the information to manipulate the world state. Their implementation belongs to the part that the students have to fulfill. A presentation of all functions to parse, analyse and process JSON data can be found in section 4.3.

3.4 Workflow of the Client

With the preprocessing of the input from the server and the functions that manipulate the world state and represent the game the client was fully working. Figure 3.1 shows the workflow and the use of the JSON functions and the JSON library in the client graphically. Every time a message from the server is received, the following steps are executed.

1. The input data from the server is parsed into a string containing a valid JSON object.
2. The achieved string is converted into a JSON hash by using Racket's JSON library.
3. Then the message can be analysed. The head key reveals its identity. For example the key "mes" announces a new chat message while "challenge" reports that the player was challenged by someone else etc.
4. Depending on the head key the respective arguments can be extracted so that in the end the appropriate event can be triggered by calling the student's function.

The work of the client was set but it was *one* client. There were no clean borders between a possible interlayer and what could be the students' task. Furthermore if

[3] see http://docs.racket-lang.org/json/index.html?q=json

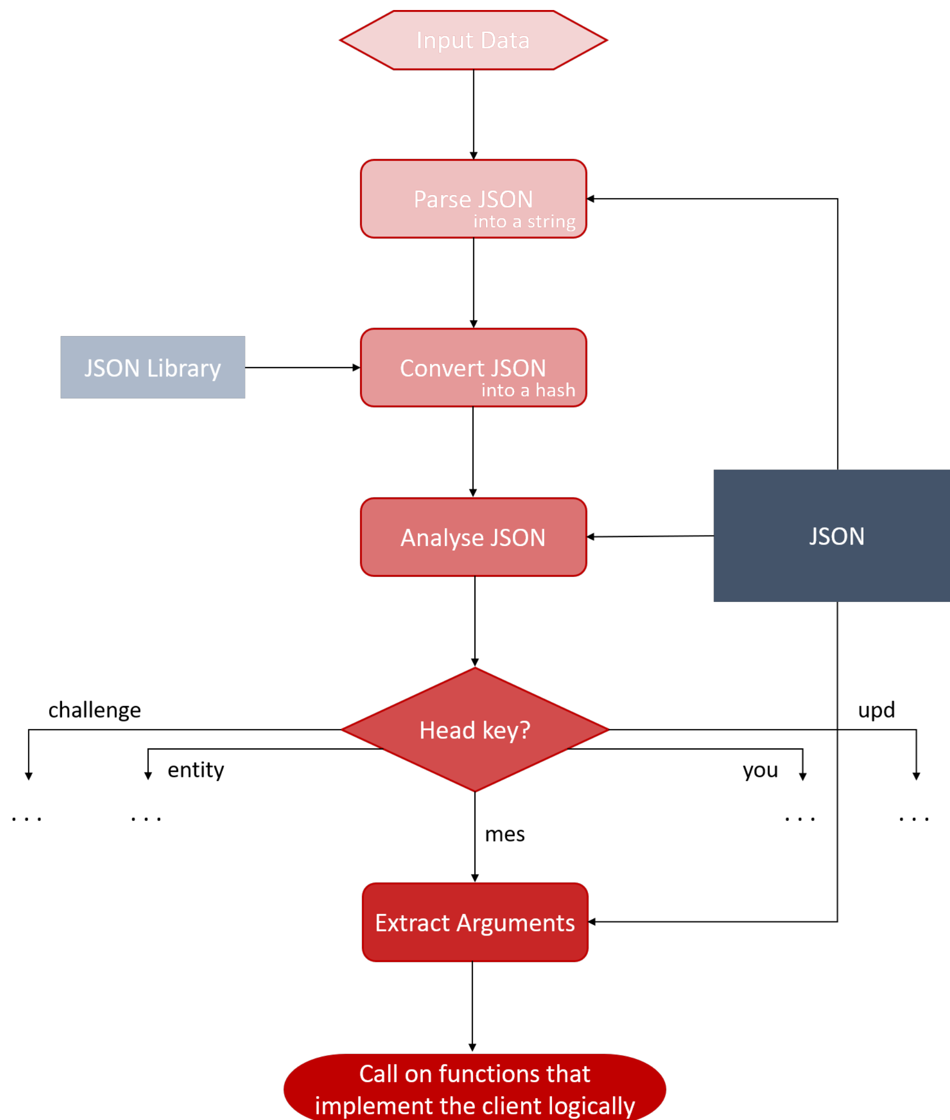

FIGURE 3.1: Workflow of the client: The steps of preprocessing server input with the use of JSON functions.

the server implemented a different game it would be unclear which part of the code could be reused.

Chapter 4

Abstraction to a General Model

The client introduced in chapter 3 comprised all functions. A division was necessary to separate the intended work for the students from the code to run in the background and to depict reusable code. This chapter introduces a model that fulfills these requirements and explains its different parts.

4.1 Turning the Idea into a Model

4.1.1 Model of the Communication

The idea introduced in the last chapter was to create an interlayer that frees the students from working with JSON data and network communication. Only the relevant data should be presented to the students in a template.

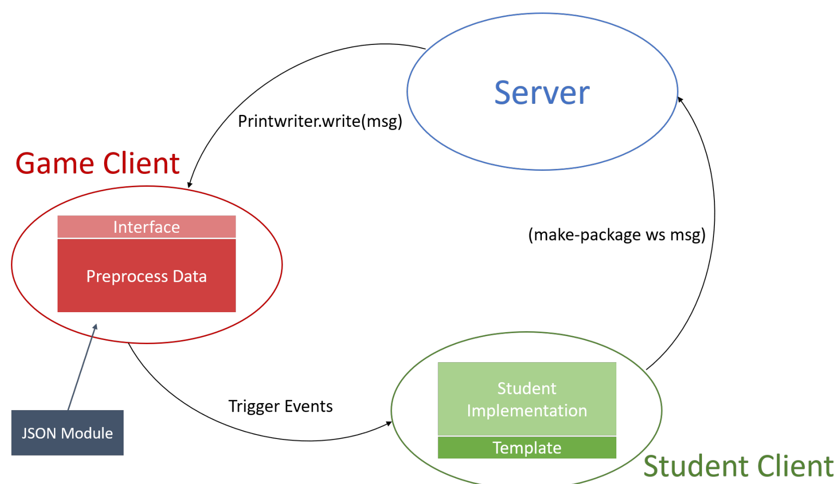

FIGURE 4.1: Communication between server, game client and student client: The server writes to the output stream of a client by using the `PrintWriter.write()` method. The JSON object is received by the game client that manages the connection. The game client preprocesses the data and provides an interface that implements game-specific events. The events are triggered by the game client and the corresponding function from the template is called. These functions are implemented by the students. The template and its implementation form the student client. The student client communicates directly with the server by returning packages and making use of the command structure of the server.

The abstraction to a general model was driven by the set-up of the template. Hence the creation of the model was carried out bottom-up. The template should be oriented towards the known concept of big-bang, simply with new features specific to the game. It should provide some guidance and secure correct usage of the interface created by the interlayer. The template and its implementation form the student client. The communication from the student client to the server should be direct and without interference by making use of the server's command structure.

The interface provided by the interlayer should be easily adapted if the server or the game is modified. The workflow of the client described in section 3.4 suggests that the game-specific dissection of incoming JSON data results in a game-specific function. Thus, a game client should combine the establishment of a connection to the server, all the steps that preprocess game-specific data and the interface that provides this data. The events presented by the interface and the corresponding functions that are to be implemented by the students are triggered by the game client. Figure 4.1 illustrates the different compounds and their interference with each other.

This division of game client and student client enables the implementation of the student client in the same way as the big-bang events were handled before when programming interactive games.

4.1.2 Model of the Logical Units

The model of the communication illustrates the separation of game client and student client but it doesn't reveal anything about the reusability of the code.

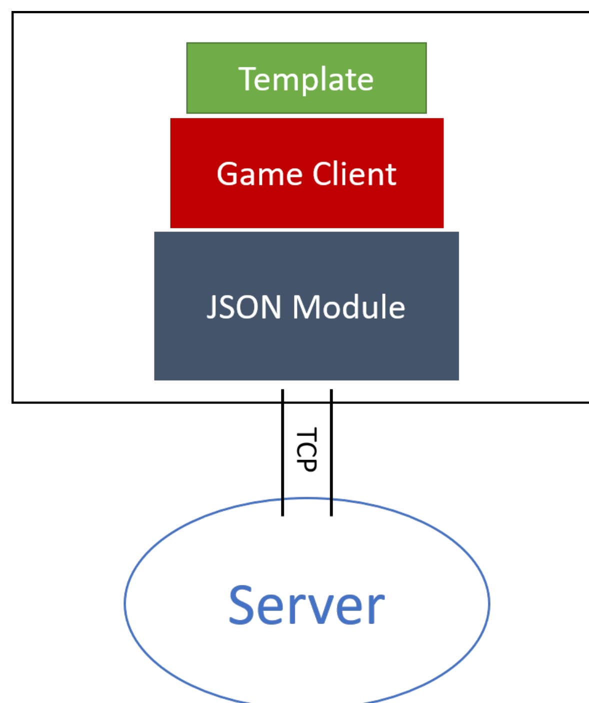

FIGURE 4.2: Model of the logical units: Everything is based on the JSON module. It provides functions for parsing, analysing and extracting JSON data. The game client merges these functions and the data to create an interface that the template reverts to.

Changing the game should be a possibility that requires little modification and high usage of the existing code. To illustrate which parts of the code can be used again completely, need only minor modifications and which part holds game-specific code that is interchangeable, the following division was made (see figure 4.2). Firstly, a JSON module was created that contains functions to convert the input data into a JSON string and access JSON hashs. Secondly, there is a file that provides the game-specific function for the students to use and therein offers an event based interface. The template that the students are supposed to use forms the third part.

This division was chosen to have a clear separation of the interchangeable game part and the unit of code that can be reused in order to need as little effort as possible to change the game. The following sections describe these units and their potential of reusability. The code for the units can be found in the `Racket` folder in the git repository.

4.2 Student Client: The Template

The template should make use of a function similar to `big-bang` with new features specific to the game. This function should resemble the structure of `big-bang` to make the transition from an interactive game using `big-bang` to a game client for multi-player games easy. The ideal was adding new clauses to `big-bang` and renaming it to clearly show the difference. This ideal is indicated in listing 4.1 but unfortunately could not be put into practice accordingly.

```
(dragonland ws
  (on-key react)
  (on-mouse click)
  (to-draw render)

  (on-dragon dragon)
  ...

  (register "LOCALHOST")
  (port 1234))
```

LISTING 4.1: The ideal set-up of a game-specific `big-bang` function

Partially copying the source code of the *universe* teachpack to create a customised version of `big-bang` was not an option due to the inclusion of the existing code that is based on the use of `big-bang` already. An acceptable solution was the use of optional keyword arguments[1]. A function with keyword arguments resembles the look of the clauses in `big-bang` and will therefore ease the transfer. Every keyword argument represents an event. Thus, the function belonging to the keyword appears in the template, takes the world state as an argument and returns with a new world state either in a package or as a single expression.

In addition to the resemblance in appearance optional keyword arguments have the advantage that minor events of the game that are not essential to the functioning of the client do not have to be implemented. Furthermore most of the `big-bang` clauses

[1] see Optional Arguments and Keyord Arguments

are optional as well and the order of the optional keyword arguments can be chosen randomly like the clauses in big-bang. This not only avoids mistakes when using them but proves their similarity (see listing 4.2).

```racket
(dragonland ws
  #:on-key react
  #:on-mouse click
  #:to-draw render

  #:on-dragon dragon
  ...

  "LOCALHOST"
  1234)
```

LISTING 4.2: Optional keyword arguments offer high resemblance to the ideal

The optionality of arguments also has the advantage of being able to structure the task of implementing the client into several smaller tasks. The template can simply be expanded. Since optional arguments need a default argument that is used if the argument is not specified, a fully implemented function can be employed instead of one that simply keeps the game from crashing. That way the game will work correctly even though not everything was implemented. Unfortunately not every event handling function can be replaced.[2] But since these functions run in the background in Racket language, they could enable side effects like printing if a teaching language is used in the template or manage complex GUI programming.

However, keyword arguments are not available in the teaching languages *HtDP* and *DMdA*. Consequently this possibility is only of value when using a Racket language. The less elegant but also working solution is a normal function that takes all event based functions as arguments. The disadvantages are the necessity of all functions, the strict order they have to follow and the loss of the big-bang-like composition of the function. Nevertheless the concept of higher-order programming is present in that solution as well.

4.3 JSON Module

The JSON module holds all functions for parsing and accessing JSON data. It is completely detached from the game or the set-up of the server. It forms the section of completely reusable code. No matter the game, the structure of JSON objects is always the same and allows the universal application of the functions. The first part of the module is the conversion of the raw JSON object (the list of lists that Racket recognises as a datum as described in section 3.3.1) into a string. The second part contains all the functions necessary to work with a JSON hash and access its data.

[2] If the world state is a structure that has to be manipulated, the structure specific functions are not accessible from the location of the default functions (since mutual imports are not allowed).

4.3.1 Documentation of the Functions

```
(js-to-string x) → string?                                   procedure
  x : any/c
```

The `js-to-string` function turns the input from the server that Racket interprets as a list of lists into a string containing a valid JSON object.

```
(head-key json) → jsexpr?                                    procedure
  json : jsexpr?
```

Depending on the js-expression the output is either

- the first key if *json* is a hash
- the first element if *json* is a list
- itself otherwise.

```
(get-value json key) → (or/c jsexpr? #f)                     procedure
  json : jsexpr?
  key : (or/c symbol? string?)
```

`get-value` returns the JSON value for the given key from the first layer of the JSON hash, #f if non-existent.

```
(path json keys) → (or/c jsexpr? #f)                         procedure
  json : jsexpr?
  keys : string?
```

This function is useful to extract a lot of arguments from a big JSON object. `path` specifies the path of the keys that are to be followed to extract the last value. Lists can be accessed by numbers via `list-ref`. If the path doesn't exist, it returns #f.

The string containing the path has to be formed thus: "key1/key2/.../keyN". The keys can either refer to a key in hash or an element in list. As an example "root/4" would find the fifth element in the list that is stored in the key "root".

Other Functions

Some other functions were developed for accessing JSON hashs but in the end not used for the thesis. Their description can be found in the scriblings of the code in the git repository under `Racket > scribbles > json.scrbl`.

4.4 Game Client

The interchangeable part of the model is the game-specific function dragonland/racket[3] that manages the connection to the server with big-bang. It defines the default functions to use if the optional argument is not set. The game client also makes use of the JSON module and thus combines everything to form the interlayer. Changes in the set-up of a server message result in small modifications here without affecting the template.

```racket
(define
  (dragonland/racket ws
                     #:on-key [key-fun keyF]
                     #:on-player [player-fun playerF]
                     ...
                     ip
                     port-no)

  (set! playerF player-fun)
  ...

  (big-bang ws
    (on-key key-fun)
    (to-draw draw-fun)
    (on-mouse mouse-fun)
    (on-receive analyse)
    (register ip)
    (port port-no)))
```

LISTING 4.3: The general set-up of the game client: The function is defined with the world state as its first argument followed by all optional keyword arguments (lines 3-5). IP and port number also have to be specified. All optional arguments have a global default argument like the playerF function. It has to be set to the student's function to be able to use it in analyse. big-bang is called with its clauses and the corresponding functions. analyse is not provided by the template but handles the preprocessing of data and triggers the appropriate events.

The "look" of the dragonland/racket function was determined in the development of the student client. Its set-up can be seen in listing 4.3 and remains the same independent from the game: the optional keyword arguments with the flags as well as setting the global functions (playerF etc.) to the student's functions.

The heart of the game client that reflects the game is the analyse function - the only function not implemented in the template. In this function the messages from the server are analysed and according to the content the events are triggered, e.g. such that the player function from the template is called on when new information about a player is received.

[3] There also is a dragonland/teaching function that works without optional keyword arguments and thus can be used in the teaching languages *DMdA.* and *HtDP*

As described before the *Dragonland* server structures the JSON objects in such a way that there is only one key on the upper level that characterises the message (see section 2.5.3). Therefore after parsing the input into a JSON hash, the head key of the hash is analysed to extract the necessary arguments and trigger the corresponding events. `analyse` and the example of triggering the chat message event can be seen in listing 4.4 lines 12 et sqq.

```racket
; analyse the message from server
; WorldState tcp-input -> HandlerResult (by invoking the student's
↪ functions)
(define (analyse ws input)
  ; parsing tcp input to json hash
  (let* ([js (string->jsexpr (js-to-string input))]
         [root (head-key js)])
    ; disassembly the server message based on the game
    (cond [(symbol? root)
           (cond

             ; trigger the chat-message event
             [(symbol=? root 'mes)
              (msgF ws
                    (path js "mes/senderid")
                    (path js "mes/sender")
                    (path js "mes/text"))]

             ...)]...)))
```

LISTING 4.4: Analysing the server input and triggering events: The input is parsed into a string and then into a hash (line 5). The head key is extracted next (line 6). According to its value the relevant function from the template is called with the extracted data es arguments (lines 12-16 for a chat message).

4.5 Testing the Model

The model was developed and tested on the provided server with the game *Dragonland*. The units interlock such that the template imports the game client that provides the game-specific function. The game client in turn imports the JSON module and Racket's JSON library. To test that all units engage correctly a student client was created by fully implementing the template.

4.5.1 The Client from a Student's Point of View

The template for the student client was already developed as a part of the model. Therefore it was possible to completely take over a student's point of view. Within the frame provided by the template it was possible to complete most of the client without difficulty. Figure 4.3 shows the final result. However, programming the student client also presented some challenges:

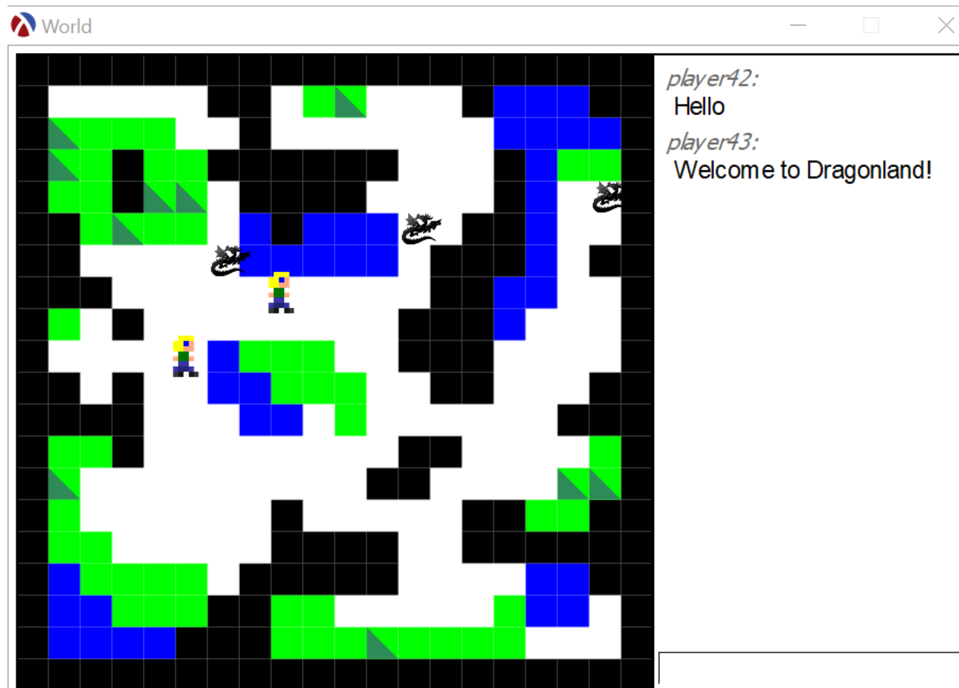

FIGURE 4.3: Screenshot: the implementation of a *Dragonland* client in Racket

- The communication from client to server is set by specific commands that the client has to send. When sending these commands another problem of how Racket reads and sends data occurred. When reading data the JSON objects were not read as a string but as a list (see section 3.3.1). When writing data to a port Racket offers several functions (see section 3.2). The *universe* teachpack uses `write` which (unlike `display`) sends the text of a string with the quotation marks. The server reads everything as a string which causes two quotation marks around the messages and hinders the server from recognising the commands. Only symbols are sent normally. But symbols do not allow whitespaces - or rather a symbol containing whitespaces is automatically surrounded by tubes which again makes the message unrecognisable.

  It is a special case and only occurs when a player wants to send a chat message to another player. Since the message itself is part of the command (and usually contains whitespaces) it poses a problem. Other special characters like parentheses or commas also produce this error of tubes surrounding the message. This behaviour requires either complicated workarounds in the client with symbols which is not a universally valid solution[4] or a simple modification of the server. The latter means removing the extra quotation marks of every message and presents the only acceptable solution for public use.

- Especially the GUI of the chat functionality was very challenging with `big-bang`. With the *universe* and the *image* teachpack the whole GUI is a composition of overlying images. There are no GUI elements like a text input field that contains accessible text. Every typed letter had to be stored individually in the world state of the client. Displaying text also poses a problem since there

[4] It was done in the exemplary implementation of the student client since the server should remain unchanged and as it is not part of the model that might be used by others.

are no automatic linebreaks when working with images. Nevertheless it was possible to create an aesthetically pleasing GUI, but the effort was out of all proportion compared to the rest of the client.

- The implementation was done in Racket language. A very small part was translated to the teaching language *DMdA*. This caused more modifications than expected. A game like *Dragonland* with its complexity is not easily implemented in a teaching language that does not offer side effects.

Chapter 5

Discussion and Conclusion

5.1 The Task

The purpose of this thesis was to find out whether it is possible to use `big-bang` and its TCP connection as a client for any server not written in Racket; furthermore to explore the details of that connection such as demands on the server and treatment of data by a `big-bang` client. Then these findings should be used with a given Java server to implement such a client and create an interlayer. This interlayer should enable easy adaptation to changes of the server and pose an interface for a simple version of a client for first-year-students to implement themselves.

5.2 Summary of the Results

The model describes the set-up of the interlayer, its different units and how they interlock. The chosen division into the units JSON module, game client and student template makes it easy to keep a structure even with complex games.

The steps in this section form a summary of the results according to the model. They also present the directions for reproducing this work with any server and thereby creating a frame for first-year-students to implement a game client.

Server specification:

1. The server needs to send an acknowledging message to the client before starting any other communication. Otherwise `big-bang` will abandon the connection (see section 3.2).
2. The communication between server and client is easier if the server wraps quotation marks around every message to send (see section 3.3.1) and removes extra quotation marks from every message received. (see section 4.5.1)
3. No matter whether the server communicates via JSON or uses a different data-interchange format, it is advantageous to have an identifier for the kind of message, e.g. only a single key on the first layer of the JSON object to differentiate between a new player who entered the game or a chat message that was received from another player (see section 2.5.3).

Creating an interlayer:

4. A normal `big-bang` client can be used in the interlayer following the example in `game-client.rkt` as described in section 4.4. It creates the interface for the student template.
5. The major change is implemented in the function called by the `on-receive` clause. There the messages from the server have to be dissected and the corresponding events need to be triggered. If JSON is used, the JSON module (see section 4.3) created for this work will support the analysis of the data.
6. The decision of the students' language is set here since keyword arguments are not available in teaching languages. Optional keyword arguments require default functions to prevent the client from crashing (see section 4.2).
7. Racket and teaching languages behave differently in case sensitivity. Thus, names of provided functions should contain lower case characters only.

Supplying students with sufficient information:

8. Providing a template shows the students a frame in which to operate and prevents problems.
9. Working without a template can be more challenging but will push the students to deeply dive into the topic. Extensive documentation is needed in this case.

5.3 Discussion: Changing the Game

Changing the game leads to modifications in the interlayer. During development the server did undergo a few changes. That made it possible to estimate the effort of work that has to be invested if the communication of the server or the game changes slightly. A difference in the communication or the set-up of a JSON object that does not effect the logic of the game only requires small changes in the game client. Only large variations in the logic of the game like adding or removing an event leads to adjustments in the game client and the student template.

According to the model of logical units a change of game (that also uses JSON) results in a new game client and student client (see figure 5.1). The JSON module remains unaffected and can be used completely. The amount of code that has to be created to change the game client is higher than wished for but it depends a lot on the complexity of the game. In addition, following the steps from the summary makes it straightforward to create a new game client and template. It is done in the style of the interlayer for the *Dragonland* game.

5.4 Concluding Remarks

Programming a client for a game can reach high complexity for first-year students and require much time and effort - especially when using a teaching language. The

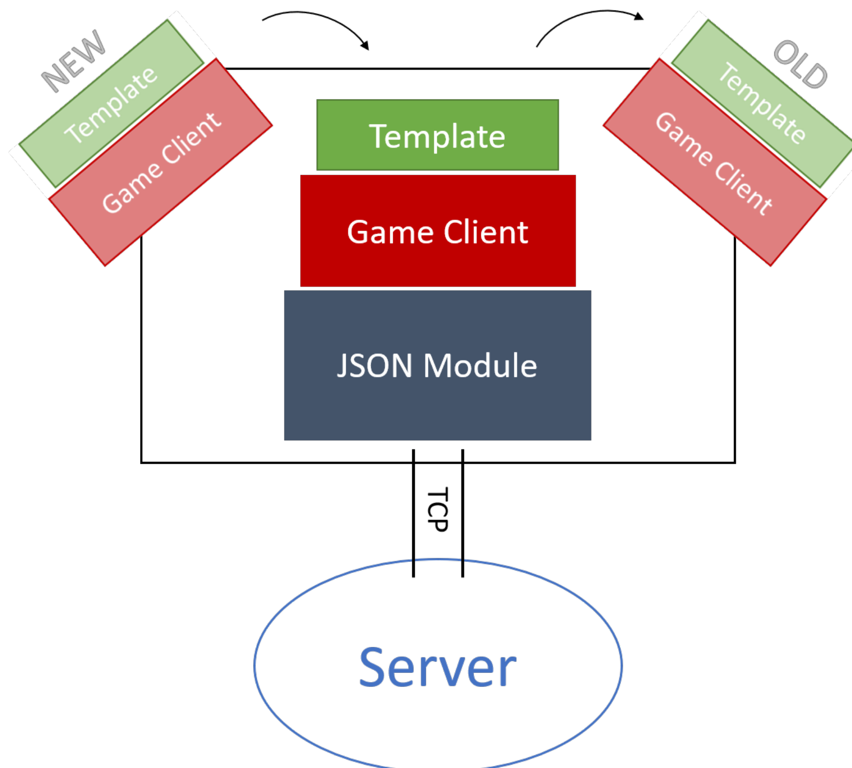

FIGURE 5.1: Model of the logical units with interchangeable parts: The game client and the student template have to be replaced completely. The structure of both units remains the same though which eases the exchange.

game is very motivating but the testing of the model indicated that the complexity of *Dragonland* is perceived as very high. As described in section 4.2 the assignment could be split into several subtasks or some code could be provided in the template already.

The inclusion of files is another matter. The template simply *requires* the files which means that they have to be in the same folder as the template. Apart from including the files directly it could be possible to also provide them in a teachpack to avoid confusion and keep the code from the students if a division into several subtasks is desired.

Future work should test this model on a different game server. It would be encouraging to see this work in practice with the new first-year students and an easier game. The high motivational factor of multi-player games in first-year teaching suggest a great success and a lot of fun for the students.

Bibliography

[Ach08] Peter Achten. "Teaching Functional Programming with Soccer-fun". In: *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*. FDPE '08. Victoria, BC, Canada: ACM, 2008, pp. 61–72. ISBN: 978-1-60558-068-5. DOI: 10.1145/1411260.1411270. URL: http://doi.acm.org/10.1145/1411260.1411270.

[CNP03] Antony Courtney, Henrik Nilsson, and John Peterson. "The Yampa Arcade". In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. Haskell '03. Uppsala, Sweden: ACM, 2003, pp. 7–18. ISBN: 1-58113-758-3. DOI: 10.1145/871895.871897. URL: http://doi.acm.org/10.1145/871895.871897.

[Fel+09] Matthias Felleisen et al. "A Functional I/O System or, Fun for Freshman Kids". In: *SIGPLAN Not.* 44.9 (Aug. 2009), pp. 47–58. ISSN: 0362-1340. DOI: 10.1145/1631687.1596561. URL: http://doi.acm.org/10.1145/1631687.1596561.

[Fel+13] Matthias Felleisen et al. *Realm of Racket: Learn to Program, One Game at a Time!* San Francisco, CA, USA: No Starch Press, 2013. ISBN: 1593274912, 9781593274917.

[Fel+15] Matthias Felleisen et al. "The Racket Manifesto". In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by Thomas Ball et al. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 113–128. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.113. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5021.

[Fel14] Matthias Felleisen. *Worlds and the Universe: "universe.rkt"*. 2014. URL: https://docs.racket-lang.org/teachpack/2htdpuniverse.html.

[Mor11] Marco T. Morazán. "Functional Video Games in the CS1 Classroom". In: *Trends in Functional Programming*. Ed. by Rex Page, Zoltán Horváth, and Viktória Zsók. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 166–183. ISBN: 978-3-642-22941-1.

[Mor15] Marco T. Morazán. "Generative and accumulative recursion made fun for beginners". In: *Computer Languages, Systems & Structures* 44 (2015). SI: TFP 2011/12, pp. 181 –197. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2015.08.001. URL: http://www.sciencedirect.com/science/article/pii/S1477842415000524.

[Mor18] Marco T. Morazán. "Infusing an HtDP-based CS1 with distributed programming using functional video games". In: *Journal of Functional Programming* 28 (2018), e5. DOI: 10.1017/S0956796818000059.