Prof. Dr. Peter Thiemann
Manuel Geffken
Matthias Keil Summer Term 2012

Softwaretechnik

http://proglang.informatik.uni-freiburg.de/teaching/swt/2012/

Portfolio

The elaboration of this portfolio is admission requirement for the *Softwaretechnik* exam. The portfolio consists of a software project described in section *Task Description*. The implementation should be done in groups up to 6 participants. Each participant has to write an individual portfolio explaining his involvement in the project.

1 Requirements

1.1 Style

The portfolio has to be written in LaTeX, document class *article* with font size *11*. Use the template provided on our webpage.

1.2 Submission

Your submission must contain:

- **The written portfolio as a single pdf file** (*NAME_MATRICULATIONNUMBER*_portfolio.pdf)
- **The source code as a zipped file** (*NAME_MATRICULATIONNUMBER*_sourcecode.zip)

It must adhere to the naming convention. Submit your solution to the top directory of your subversion repository.

1.3 Subversion

We provide a subversion repository for each group for development, group communication, and for the final submission.

- `https://proglang.informatik.uni-freiburg.de/svn/swt2012GROUPNUMBER`

You have to register for the repository with your group members till **Wednesday, May 23.**.

- `http://proglang/teaching/swt/2012/registration/`

To get access to the repository you have to login with your *TF user name* and the *WWW Password*. To set the *WWW Password* use the following link:

- `https://support.informatik.uni-freiburg.de/cgi/support/fawmgr.cgi?wpassword:en`

1.4 Deadline

Deadline for the submission is:

- **Saturday, July 28.**

2 Assessment

Your submission will be checked for completeness.

3 Implementation

The implementation should be done in groups up to 6 participants. Three participants have to handle the server part. The other three have to handle the client side. Provide a division of tasks to work together. Take use of the provided subversion repository. Each participant has to write an individual portfolio explaining his involvement in the project.

3.1 Language

Provide the implementation in Java.

4 Topics

Treat the following topics in your portfolio. Write about 1-2 pages per task.

4.1 Specification and requirements

Determine the important functions and requirements of your purpose. Furthermore classify each requirement with a priority of importance, elicit interactions between agents (people, peripherals, ...) as well as purposes and processes. Elaborate requirements for usability, technique and environment. Provide development plans for workflows, timelines and milestones including metrics for evaluation and performance and testing mechanisms.

Task 1: Functional Specification Document Provide a *Functional Specification Document* encompassing your part of the task.

Task 2: Product Requirements Document Provide a *Product Requirements Document* encompassing encompassing your part of the task.

4.2 Cost estimation

Investigate the *function point method* for estimating the cost of a software development and evaluate its usability.

Task 3: Cost estimation Use the *function point method* to provide a cost estimation to accomplish the specified product requirements of your part. Compare this at the end with the actually used time.

4.3 Unified Modeling Language (UML)

Use the *Unified Modeling Language* to provide an abstract description of your software system.

Task 4: Class diagram Provide a class diagram for the client or server part of the implementation including interfaces to adjoining and extending parts. Describe the classes and interfaces, their attributes and methods together with the relationships among the elements.

Task 5: State chart diagram Provide a state chart diagram describing the behavior of your part as an abstract description of the system.

4.4 Design pattern

Familiarize yourself with the concept of *Design pattern*.

Task 6: Design pattern Elaborate the use and usability of design patterns in your implementation. Work out advantages and disadvantages and consider suitable intended uses, and evaluate actual uses in your implementation.

4.5 Design by contract

Familiarize yourself with the concept of *Design by contract*.

Task 7: Design by contract Use *JML* to specify invariants for all classes and pre- / postconditions for all methods. Use the *ESC/Java2* (JML tool) to perform contract monitoring and report your experience.

4.6 Testing / Debugging

Revise the introduced techniques for testing and debugging. Develop a test plan for the task specification.

Task 8: Unit testing Your source code must come with unit tests that demonstrate that it satisfies the product requirements. Create unit tests from the JML specifications for all methods using *JUnit*. There are tools like *JMLUnit* that generate unit tests from JML annotated Java files. Which other kinds of testing are applicable and make sense?

Task 9: Debugging Document your debugging efforts. Did you employ techniques from the lecture? If not, why not?

5 Task Description

5.1 Overview

Recent breakthroughs in higher-order, statically-typed, metaspatial communication will enable data to be transferred between Mars and Earth almost instantaneously. As such, NASA is seeking examples of real-time control software to operate its latest model Martian rovers from control centers on Earth. Since it is well known that the University of Freiburg attracts the *crème de la crème* of students from around the world, NASA has decided to use the current SWT Portfolio Project as a means of gathering software prototypes for their new control system. We are pleased to announce that this year's most promising project team will win the tender for the supply of the actual rover control software for NASA's very next mission to Mars![1] Good luck, and may yours be the winning entry, to be used on Mars itself.

Your control software will communicate with the rover over a network socket. Its object is to guide the rover safely from a given starting location to its home base. The controller's primary function is to avoid the boulders and craters that litter the Martian surface. As an added nuisance, the local inhabitants, who are decidedly hostile, will immediately destroy any rover they can get their fourteen sticky fingers on. Note that Martians, like dogs, vary in intelligence.

As NASA cannot provide you with a Martian rover to run your tests and does not provide a simulator yet either your job is to build both the control software and a simulator. However, NASA will eventually evaluatate your software with its own simulator. Control software prototypes will be evaluated according to their performance in a series of trials, the details of which are given below. These trials are to be modelled by your simulator. Each trial consists of five runs on a given region of Mars. As a means of preparing for these trials, this document and its accompanying infrastructure provide sufficient details for the development of both prototype controller candidates and rover simulators.

Among the usual requirements for safety-critical software are extensive documenation and the adherence to strict processes. Hence, to demonstrate your competence regarding these requirements, NASA requests you to provide them with a **portfolio document** accompanying your controller/simulator prototype implementation.

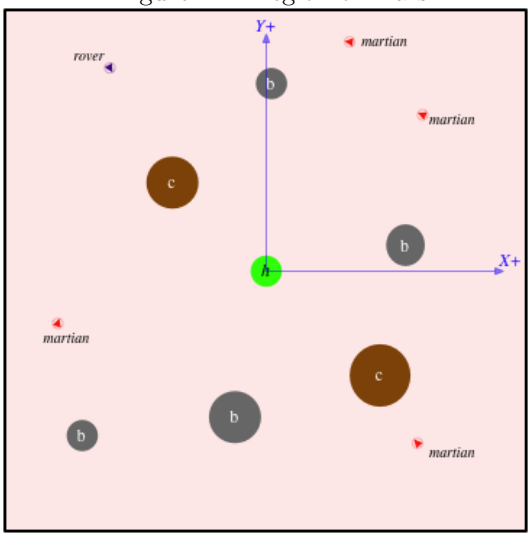
5.2 Mars rover behavior

The rover is a circular vehicle of radius 0.5m, which your controller must guide across the Martian terrain. To do so, your controller must establish a connection to the rover over a TCP/IP socket. This socket it used for all communication to and from the rover. While metaspatial communication is very fast, there is some latency in the connection (on the order of 75 microseconds).

Once the connection is established, the controller will be sent an initial message about the dimensions of the world and the physical characteristics of the vehicle. The world is modeled as a rectangular segment of the xy-plane centered at the origin. The *home base* – the rover's intended destination – is a circle (radius 5m) located at the origin. The vehicle's characteristics include its maximum speed, its maximum rotational speed when turning,

[1] Subject to budget constraints.

Figure 1: A region of Mars



and a few other facts. NASA is testing various different models of rovers, which have varying performance. They are also testing different regions of Mars, with a wide range of characteristics. For a given trial, the rover's performance and the map will be fixed, but these will vary from trial to trial. Complete information on the initial message is furnished in Section 5.3.2 below.

About one second after the initial message is sent, the first run starts and the server begins sending a stream of vehicle telemetry data to the controller. Telemetry data consists of location, speed, and information about the local terrain (see Section 5.3.3 for full details). At any time after the telemetry data has started streaming to the controller, the controller may issue commands back to the server to direct the vehicle towards home base.

The rover must avoid three kinds of perils on its way home. If the rover hits a boulder, or the edge of the map, it bounces off and loses speed. Hitting a boulder happens if the distance between the center of the boulder and the center of the rover is less than the sum of their radii. If the center of the rover enters a crater, it falls in and explodes. If a vehicle is caught by a Martian, it is destroyed and its remains are sacrificed to the Gods of Barsoom, of whom it is best not to speak further.

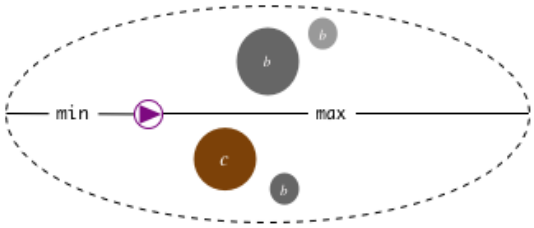
Martians, while hostile, possess no special physical abilities; that is, they cannot drive through craters, pass through objects, or escape the boundary of the map. The physics of Martian movement is the same as for the rover, although they may be faster or slower, *etc.* Martians are slightly smaller than the rover, having a radius of 0.4m.

An illustration of a typical Martian region appears in Figure 1. Boulders, craters and home base are marked b, c and h respectively.

5.2.1 Vision

The rover's visual sensors cover an elliptical area that extends further in the direction that the rover is facing. Figure 2 depicts this region.

Figure 2: A sketch of the vision model



The rover is oriented toward the right in this illustration. Implicitly, there is an ellipse defined by min and max, with the rover always positioned at one of the foci. The rover can see everything within this ellipse, with the exception of those objects that are occluded by boulders. In the figure, the rightmost boulder is not visible to the rover because of the larger boulder blocking it. The lowermost boulder, on the other hand, is visible, since craters do not occlude vision.

5.2.2 Speed

The linear speed of the rover at time t' ($s_{t'}$) is computed according to its speed at its predecessor time t according to the following formula.

$$s_{t'} = max(s_t + (t' - t)a - k(t' - t){s_t}^2, 0)$$

The latter term is the simulated *drag* on the vehicle. Note a, the *acceleration*, can be negative if the rover is braking. The rover's acceleration and braking rates are not known, although they can be determined by experiment. The effect of drag is to limit the maximum speed of the rover. The maximum speed is known (it is communicated as part of the initial message), but the drag coefficient is not known.

5.2.3 Turning

The rover has two turning speeds – regular turns and hard turns – in both directions. When the rover receives a command to turn left, its turning state moves one "notch" in the leftward direction; likewise for right. Note that while the turn rate and hard-turn rates are known, the rotational acceleration of the vehicle is finite and thus it will take some time change from one turning mode to another. Section 5.3.7 addresses the mechanics of steering messages in greater detail.

5.3 Network protocol

Communication between the server and controller will be over a TCP/IP socket using plain-text messages encoded in ASCII. The controller will be given a server hostname and port

number as command-line arguments at the beginning of each trial. The controller should establish a client-side TCP/IP connection to the server; this socket will be used by the controller to send commands to the vehicle and by the server to send telemetry data to the controller.
A *message* is a sequence of tokens delimited by the ASCII space character (`0x20`) and terminated by a semicolon. The tokens in a message represent quantities of various kinds and have the following formats:

- Distances, lengths, and locations (x and y coordinates) are given in meters in fixed-point representation rounded to the nearest thousandth (millimeter).
- Angles are given in degrees in fixed-point representation rounded to the nearest tenth.
- Angular velocities are given in degrees per second in fixed-point representation rounded to the nearest tenth.
- Speeds are given in meters per second in fixed-point representation rounded to the nearest thousandth.
- Durations are given in whole milliseconds (since the start of the simulation).

5.3.1 Messages from the server to the controller

There are a variety of messages that the rover sends to the controller. Each message begins with a single character that denotes the message kind.

5.3.2 Initialization

The exact characteristics of the vehicle are unspecified and may differ between trials, but information about the vehicle will be given at the beginning of each trail. Once the connection to the server is established, the controller will receive an initial message with the following format:

`I` *dx dy time-limit min-sensor max-sensor max-speed max-turn max-hard-turn* `;`

where

`I` is the message tag signifying initialization.

dx is the span of the map's x-axis (meters). A map with dx 100.000 extends from -50.000 to 50.000 on the x-axis.

dy is the span of the map's y-axis (meters). A map with dy 100.000 extends from -50.000 to 50.000 on the y-axis.

time-limit is the time limit for the map (milliseconds). Map time limits are discussed in Section 5.4 below.

min-sensor is the minimum range of the vehicle's visual sensors (meters). See the discussion of the vision model in Section 5.2.1 above.

max-sensor is the maximum range of the vehicle's visual sensors (meters). See the discussion of the vision model in Section 5.2.1 above.

max-speed is the maximum speed of the vehicle (meters per second).

max-turn is the maximum rotational speed when turning (degrees per second).

max-hard-turn is the maximum rotational speed when turning hard (degrees per second).

5.3.3 Telemetry stream

During a run, the server sends a steady stream of telemetry data to the vehicle controller (roughly one message every 100 milliseconds). This data includes information about the vehicle's current state (control-state, heading, velocity, *etc.*) as well as information about the local map conditions (obstacles and enemies).

`T` *time-stamp vehicle-ctl vehicle-x vehicle-y vehicle-dir vehicle-speed objects* `;`

where

`T` is the message tag signifying telemetry data.

time-stamp is the number of milliseconds since the start of the run.

vehicle-ctl is the current state of the vehicle controls. It is a two-character sequence with the first character specifying the acceleration state (`a` for accelerating, `b` for braking, or `-` for rolling, *i.e.*, moving at a constant speed) and the second character specifying the turning state (`L` for hard-left turn, `l` for left turn, `-` for straight ahead, `r` for right turn, and `R` for hard-right turn). Note that the rover will gradually slow down when rolling, because of drag.

vehicle-x is the x-coordinate of the vehicle's current position.

vehicle-y is the y-coordinate of the vehicle's current position.

vehicle-dir is the direction of the vehicle measured as a counterclockwise angle from the x-axis.

vehicle-speed is the vehicle's current speed (meters per second).

objects is a sequence of zero or more obstacles and/or enemies that are visible to the vehicle. An item is *visible* if it falls in the range of the vehicle's visual sensors; recall that range is part of the rover characteristics given in the server's initial message. Object messages have two different formats depending on the type of object. If the object is a boulder, crater, or home base, the format is

object-kind object-x object-y object-r

where

object-kind is one of `b` (for a boulder), `c` (for a crater), or `h` (for home base).

object-x is the x-coordinate of the object's center.

object-y is the y-coordinate of the object's center.

object-r is the radius of the object.

If the object is a Martian, the description has the format

`m` *enemy-x enemy-y enemy-dir enemy-speed*

Here is an example telemetry message. Note that we have split this message over multiple lines to improve readability – the actual message would not contain any newline characters.

```
T 3450 aL -234.040 811.100 47.5 8.450
b -220.000 750.000 12.000
m -240.000 812.000 90.0 9.100 ;
```

This message describes the vehicle's state at 3.45 seconds after the start of the run. It is currently accelerating and turning hard to the left. Its position is $(-234.040, 811.100)$, its direction is 47.5 degrees (roughly NE), its velocity is 8.450 meters per second, and it sees one boulder and one Martian.

5.3.4 Adverse events

There are also messages to signal unhappy occurrences. These messages have the format:

message-tag time-stamp `;`

where the *message-tag* is one of

`B` for a crash against a boulder or the map edge. When the rover crashes into a boulder, it bounces off and loses speed. Each crash message is immediately followed by a telemetry message so that the controller can update its state.

`C` if the vehicle fell into a crater. Falling into a crater destroys the rover and ends the run.

`K` if the vehicle was killed by a Martian, which ends the run.

5.3.5 Success message

The server sends the message "`S` t `;`" when the vehicle reaches home base safely. The current run is terminated on success.

5.3.6 End-of-run message

At the end of a run, the server sends the message "`E` t s `;`" where t is the time since the beginning of the run, and s is the score (*i.e.*, the run time plus any penalties). Note that each run will end with exactly one of the following sequences of server messages:

- `C` t `;`, then `E` t s `;`
- `K` t `;`, then `E` t s `;`

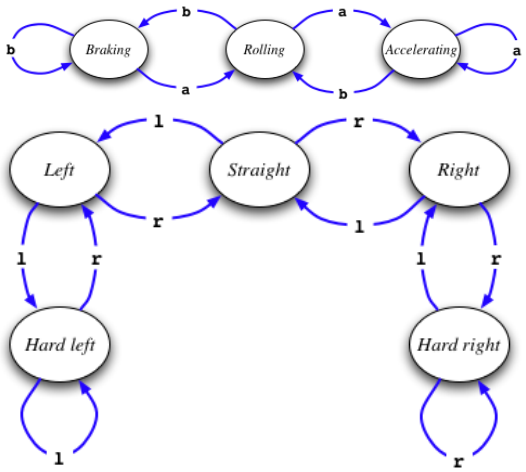
- `S` *t* `;`, then `E` *t s* `;`
- `E` *t s* `;` preceded by none of `C`, `K`, or `S`, indicating that the time limit has been reached

Once a run has terminated, there will be a pause of at least one second before the start of the next run. Note that the controller should not exit at the end of the run, but instead should prepare for another run. Also note that the initialization message described in Section 5.3.2 is only sent once per trial. Each run of the trial uses the same map, although the rover's initial position and the number and location of Martians can vary from run to run.

5.3.7 Messages from the controller to the server

The rover behavior is controlled by a pair of state machines (Figure 3), which, in turn, are controlled by commands sent by the controller.

Figure 3: Vehicle-control state machines



Each command consists of an optional acceleration (`a`) or braking (`b`) command, followed by an optional turning command (`l` for left and `r` for right), and followed by a semicolon (`;`). Thus, the grammar of controller-to-server messages is

Message::= `;` | `a;` | `b;` | `l;` | `r;` | `al;` | `ar;` | `bl;` | `br;`

No other characters (including whitespace) should be sent over the command stream!

While communication with the rover is fast, there may be some latency (less than 20 milliseconds) in processing the commands. The controller may send messages as often as it likes, although flooding the network can negatively affect performance. We recommend only sending messages in response to telemetry data, although you may need a sequence of messages to reach the desired control state.

5.4 Scoring

NASA will evaluate control software prototypes in a series of trials of varying difficulty. Your simulator shall support the same evaluation scheme. A *trial* consists of five *run*s on the same map. Each map has an associated *time limit* of some number of milliseconds. A *limit-n map* has an upper limit of n milliseconds.
The *score* for a run is the amount of time it takes to complete the run or be destroyed, plus any penalties.

- If a rover reaches home base on a given limit-n map in some $t \leq n$ number of milliseconds, the run score is t.
- If the rover fails to reach home base on a given limit-n map, the run score is given by the equation $2\ n\ -\ t\ +\ p$, where t is the elapsed time, and p is the penalty as follows:
  - 100 if the time limit has been exceeded,
  - 600 if the rover was destroyed by a Martian, or
  - 1000 if the rover fell into a crater.

  Note that t is at most n in this formula, since the run is halted when t exceeds n. Therefore $(2n-t)$ will always be between n and $2n$ inclusive.

As in ski racing, lower scores are better.
The *trial score* is the sum of the three lowest scores in the trial.

5.5 How to test your program

NASA is providing sample maps for teams to test their code on while developing their simulators. The maps are available for download from

`http://proglang.informatik.uni-freiburg.de/teaching/swt/2012/sample-maps.tgz`

Details on the map file format are addressed in Section 5.7.2.

5.6 Implementation hints

Because the controller program is sensitive to network latency, you should disable Nagle's algorithm for the socket. You can do this using the `Socket.setTcpNoDelay(boolean)` method.

5.7 Simulator

NASA demands that your simulator (sometimes referred to as "server") fullfils the following requirements.

5.7.1 Running the simulator

To run the server, you must supply it with the name of a map file, so NASA is providing some sample maps for download. For example, the command

```
./server -v map1.wrld
```

will run the server on `map1.wrld` with a graphical view of the simulation. When the simulator starts up, it prints the message:

`waiting for client connection on port` *n*

where *n* is the port number used to connect to the server. When running graphical mode, the simulator can be terminated by typing the '`q`' key. Assuming your program resides is in the `bin` directory, you can then run it with the command

`bin/run` *hostname n*

where *hostname* is the name of the machine on which the server is running. You can also specify the port using the `-p` option. For example,

```
./server -p 19023 map1.wrld
```

will run the server in non-graphical mode using port 19023.

5.7.2 Map-file format

NASA has supplied some sample maps for download from

`http://proglang.informatik.uni-freiburg.de/teaching/swt/2012/sample-maps.tgz`,

but you may wish to define your own.
Maps are represented as JSON (JavaScript Object Notation) files. (See `http://www.json.org/` for details on JSON.) The format of a map file is as follows:

```
    {
"size" :  INT,
"timeLimit" :  INT,
"vehicleParams" :  PARAMS,
"martianParams" :  PARAMS,
"craters" :  [ { "x" :  FLOAT, "y" :  FLOAT, "r" :  FLOAT }, ...  ],
"boulders" :  [ { "x" :  FLOAT, "y" :  FLOAT, "r" :  FLOAT }, ...  ],
"runs" :  [ RUN, ...  ]
}
```

where *PARAMS* is a JSON object with the following format:

```
    {
"maxSpeed" :  FLOAT,
"accel" :  FLOAT,
```

```
"brake" :  FLOAT,
"turn" :  FLOAT,
"hardTurn" :  FLOAT,
"rotAccel" :  FLOAT,
"frontView" :  FLOAT,
"rearView" :  FLOAT
}
```

and a *RUN* is a JSON object with the following format:

```
    {
"vehicle" :  VEHICLE,
"enemies" :  [ ENEMY, ...  ]
}
```

A *VEHICLE* object has the form

```
    {
"x" :  FLOAT,
"y" :  FLOAT,
"dir" :  FLOAT
}
```

and an *ENEMY* object is a vehicle object extended with the following extra fields:

```
    {
...  as per VEHICLE ...
"speed" :  FLOAT,
"view" :  FLOAT
}
```